# 期末大作业小组报告

**21307071 詹迪　21307275 许嘉瑋**

## 一、项目简介

**【项目名称】**结合光线追踪之二维图片的三维重建动画

**【项目简介】**我们的项目将静态的二维图像转换为动态的三维模型。通过对二维图像进行分析和处理，从中提取出三维信息，并将其表现为具有动态变化视觉效果的三维模型动画。具体的视觉效果包括：

1、以自定义的“体素”——三维模型的基本单元来构建三维模型。每个自定义的“体素”单元都可以自旋转动态展示，从而将静态的二维图像转换为动态的三维模型。

2、不同距离视角下，动态三维模型的多角度旋转动画展示。

3、不同光照下的全局光线追踪。

4、多个三维模型轮流切换与动画表现。

我们的项目还实现简单的交互功能，包括三维动画模型的切换、光源位置的修改等。

**【开发环境】**Qt Creator 5.13.0

**【小组成员】**21307071 詹迪、21307275 许嘉瑋

**【分工合作】**

21307071 詹迪主要负责项目中：

①自定义“体素”——三维模型中基本单元的设计与实现

②多视角下，三维模型的多角度旋转动画

③与组员合作完成三维模型的全局光线追踪算法

21307275 许嘉瑋主要负责项目中：

①二维图片的三维建模与渲染

②程序交互功能的实现

③与组员合作完成三维模型的全局光线追踪算法

## 二、21307275 许嘉瑋个人实现报告

**【个人实现内容 1】**

**二维图片的三维建模与渲染**

总体上要实现二维图像的简单三维重建，首先要做的是解析二维图像，并自定义规则重建出能描绘该图像的三维轮廓模型，最后使用自定义元素构建该模型并渲染出来。

**【实现思路】**

为了解析出的二维图像信息能在后续被更方便的使用，我们选择先将图像转为纹理对象，再读取纹理对象的各项数据（包括位置、颜色等）。其中为了实现轮廓模型的三维化，核心思想是基于像素颜色对像素进行了离散化分层。最终再通过自定义的着色器初步渲染出该模型。

**【代码实现】**

①搜集想重建的图像，并用 XML 语言将数据编写进 Qt 的 qrc 文件以便后续使用。


（background.qrc 内代码）

②构建顶点着色器，进而构建片段着色器。

```
135
136  static const char *vertexShader =
137      "layout(location = 0) in vec4 vertex;\n"
138      "layout(location = 1) in vec3 normal;\n"
139      "out vec3 vert,vertNormal,color;\n"
140      "uniform mat4 projMatrix,camMatrix,worldMatrix,myMatrix;\n"
141      "uniform sampler2D sampler;\n"
142      "void main() {\n"
143      "    ivec2 pos = ivec2(gl_InstanceID % 32, gl_InstanceID / 32);\n"
144      "    vec2 t = vec2(float(-16 + pos.x) * 0.8, float(-18 + pos.y) * 0.6);\n"
145      "    float val = 2.0 * length(texelFetch(sampler, pos, 0).rgb);\n"
146      "    mat4 wm = myMatrix * mat4(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, t.x, t.y, val, 1) * worldMatrix;\n"
147      "    color = texelFetch(sampler, pos, 0).rgb * vec3(0.6, 0.8, 0.8);\n"
148      "    vert = vec3(wm * vertex);\n"
149      "    vertNormal = mat3(transpose(inverse(wm))) * normal;\n"
150      "    gl_Position = projMatrix * camMatrix * wm * vertex;\n"
151      "}\n";
```

（顶点着色器定义）

声明 vec4 类型输入变量 vertex，代表顶点位置信息。

声明 vec3 类型输入变量 normal，代表顶点法线信息。

声明 vec3 类型输出变量 vert（顶点位置）、vertNormal（顶点法线）、color（颜色信息）。

声明 mat4 类型的 uniform 变量，projMatrix（投影矩阵）, camMatrix（相机矩阵）, worldMatrix（世界矩阵）, myMatrix（自定义矩阵）。

声明 sampler2D 类型（2D 纹理采样器）的 uniform 变量 sampler。

主函数中进行如下计算：

- 当前实例在纹理中的坐标位置 pos。
- 在原始纹理坐标基础上进行一些调整求得纹理坐标 t。
- 通过纹理采样函数 texelFetch 获取指定坐标 pos 处像素的颜

色并计算其长度 val，这是轮廓模型最终表现为立体的重要实现部分。

- 变换矩阵 wm，包含多次矩阵乘法和纹理坐标 t、val 的运算。
- 将 worldMatrix 乘到现有的 wm 矩阵上。
- 通过纹理采样函数 texelFetch 获取指定坐标 pos 处像素的颜色，并与一个固定的颜色向量相乘进行适当颜色、亮度变换求得颜色值 color。
- 对顶点坐标 vertex 应用矩阵变换求得变换后的顶点位置 vert。
- 使用变换矩阵 wm 对法线数据进行变换求得变换后的顶点法线 vertNormal。
- 应用投影矩阵、相机视图矩阵和变换矩阵 wm 对顶点位置进行变换，并赋值给 gl_Position（OpenGL 中表示顶点位置的特殊变量）。

```
152
153 static const char *fragmentShader =
154     "in highp vec3 vert,vertNormal,color;\n"
155     "out highp vec4 fragColor;\n"
156     "uniform highp vec3 lightPos;\n"
157     "void main() {\n"
158     "   highp vec3 L = normalize(lightPos - vert);\n"
159     "   highp float NL = max(dot(normalize(vertNormal), L), 0.0);\n"
160     "   highp vec3 col = clamp(color * 0.2 + color * 0.8 * NL, 0.0, 1.0);\n"
161     "   fragColor = vec4(col, 1.0);\n"
162     "}\n";
163
```

（片段着色器定义）

声明 vec3 类型输入变量，分别是顶点位置 vert、顶点法线 vertNormal 和颜色 color。

声明 vec4 类型输出变量 fragColor，表示片段的最终颜色值。

声明 vec3 类型 uniform 变量，表示光源位置。

主函数中进行如下计算。

- 光源位置减去顶点位置、取其单位向量求出入射光线方向 L。
- 计算顶点法线和光线方向的点积求出光照强度 NL。

- 根据光照强度调整颜色计算最终颜色值 col。
- 将最终的颜色值赋给片段颜色输出变量 fragColor

③着色器的使用

```
263  void GLWindow::initializeGL()
264  {
265      QOpenGLFunctions *f = QOpenGLContext::currentContext()->functions();
266
267      if (m_texture) {
268          delete m_texture;
269          m_texture = nullptr;
270      }
271      QImage img(QString(":/%1.png").arg(count));
272      Q_ASSERT(!img.isNull());
273      m_texture = new QOpenGLTexture(img.scaled(32, 36).mirrored());
274
275      if (m_program) {
276          delete m_program;
277          m_program = nullptr;
278      }
279      m_program = new QOpenGLShaderProgram;
280      m_program->addShaderFromSourceCode(QOpenGLShader::Vertex, versionedShaderCode(vertexShader));
281      m_program->addShaderFromSourceCode(QOpenGLShader::Fragment, versionedShaderCode(fragmentShader));
282      m_program->link();
283
284      m_projMatrixLoc = m_program->uniformLocation("projMatrix");
285      m_camMatrixLoc = m_program->uniformLocation("camMatrix");
286      m_worldMatrixLoc = m_program->uniformLocation("worldMatrix");
287      m_myMatrixLoc = m_program->uniformLocation("myMatrix");
288      m_lightPosLoc = m_program->uniformLocation("lightPos");
289
290      if (m_vao) {
291          delete m_vao;
292          m_vao = nullptr;
293      }
294      m_vao = new QOpenGLVertexArrayObject;
295      if (m_vao->create()) m_vao->bind();
296
297      if (m_vbo) {
298          delete m_vbo;
299          m_vbo = nullptr;
300      }
301      m_program->bind();
302      m_vbo = new QOpenGLBuffer;
303      m_vbo->create();
304      m_vbo->bind();
305      m_vbo->allocate(m_logo.constData(), m_logo.count() * sizeof(GLfloat));    ⚠implicit conversion c
306      f->glEnableVertexAttribArray(0);
307      f->glEnableVertexAttribArray(1);
308      f->glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), 0);
309      f->glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat),
310                               reinterpret_cast<void *>(3 * sizeof(GLfloat)));
311      m_vbo->release();
312
313      f->glEnable(GL_DEPTH_TEST);
314      f->glEnable(GL_CULL_FACE);
315  }
316
```

- 加载纹理：首先通过 QImage 加载了一个图像文件（文件路径是类似于 :/1.png 的形式），然后使用 QOpenGLTexture 创建了一个 OpenGL 纹理对象。缩放这个纹理对象到 32x36 的大小，并且做镜像翻转处理。

- 创建了一个 QOpenGLShaderProgram 对象，向其中添加了顶点

着色器和片段着色器的源代码，然后链接这些着色器。

- 获取着色器中的 uniform 变量位置
- 创建顶点数组对象（VAO）和顶点缓冲对象（VBO）并进行绑定和配置。顶点缓冲对象存储了顶点数据，并配置了顶点属性指针，告诉 OpenGL 如何解释顶点数据。
- 启用 OpenGL 深度测试和面剔除。

④初步绘制。

```
327 void GLWindow::paintGL()
328 {
329     QOpenGLExtraFunctions *f = QOpenGLContext::currentContext()->extraFunctions();
330 
331     f->glClearColor(0.2f, 0.2f, 0.2f, 1);
332     f->glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
333 
334     m_program->bind();
335     m_texture->bind();
336 
337     if (m_uniformsDirty) {
338         m_uniformsDirty = false;
339         QMatrix4x4 camera;
340         camera.lookAt(m_eye, m_eye + m_target, QVector3D(0, 1, 0));
341         m_program->setUniformValue(m_projMatrixLoc, m_proj);
342         m_program->setUniformValue(m_camMatrixLoc, camera);
343         QMatrix4x4 wm = m_world;
344         wm.rotate(m_rotation, 1, 1, 0);
345         m_program->setUniformValue(m_worldMatrixLoc, wm);
346         QMatrix4x4 mm;
347         mm.setToIdentity();
348         mm.rotate(-m_rotation2, 1, 0, 0);
349         m_program->setUniformValue(m_myMatrixLoc, mm);
350         m_program->setUniformValue(m_lightPosLoc, QVector3D(0, 0, lz-Rg/2));
351     }
352 
353     f->glDrawArraysInstanced(GL_TRIANGLES, 0, m_logo.vertexCount(), 32 * 36);
354 }
355 
```

（初步绘制代码）

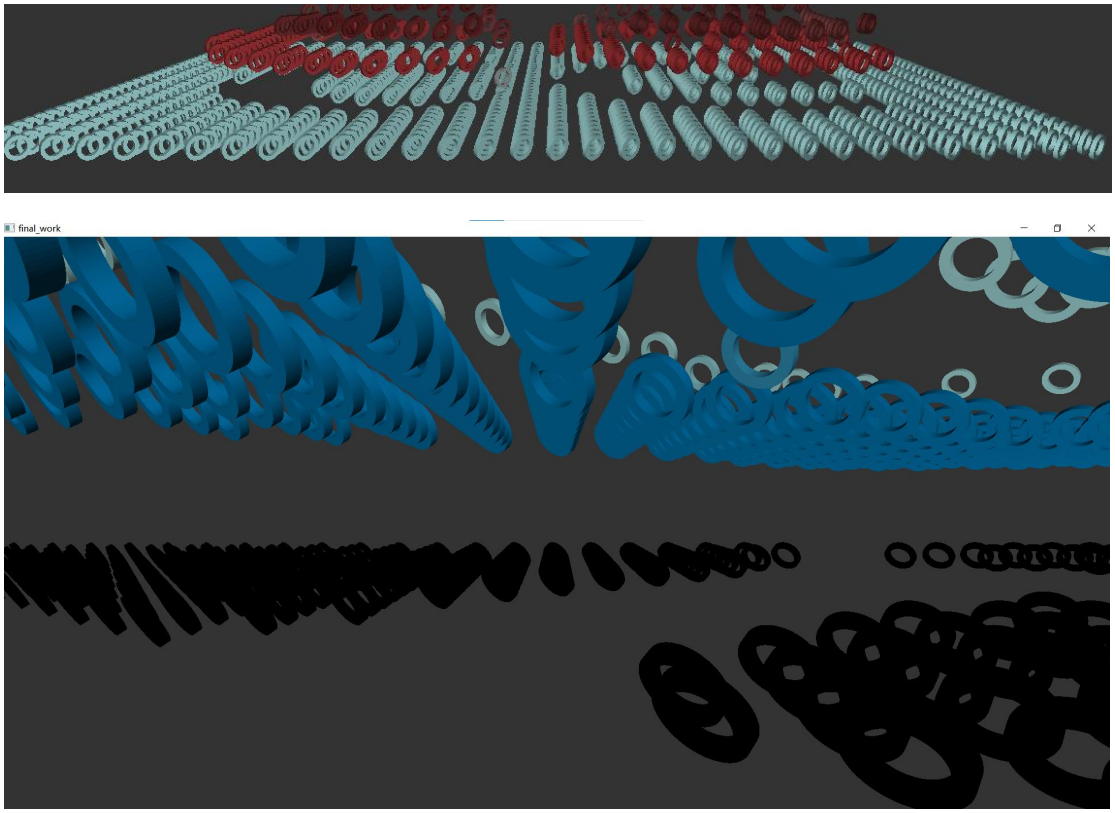
获取 OpenGL 额外函数的指针，以便调用 OpenGL 的额外函数。

- 设置清除颜色为灰色并清除颜色缓冲和深度缓冲。
- 绑定着色器程序，准备使用它进行渲染。
- 绑定纹理，以便在渲染时使用。
- 检查是否需要更新 uniform 变量。如果需要更新，进行下列操

作：创建投影矩阵、相机视图矩阵、旋转世界矩阵和自定义矩阵 并设为 uniform 变量；设置光源位置的 uniform 变量。

- 使用当前的着色器程序，绑定的纹理以及之前设置的 uniform 变量，以实例化的方式绘制图形。

**【效果展示】**




（基于颜色的层次化分离效果）

**【个人实现内容 2】**

**程序交互功能的实现**

为了验证我们程序的对于 RGB 颜色空间的二维图像的通用性，我们原本使用了定时器周期性（约 20s）替换需要渲染的二维图像。为了更加灵活的控制程序，我们加入按键检测功能，实现按下 1 时可以立刻切换图像；同时为了能更好的展示在不同光照位置的效果，我们

实现了按下 2 时改变光源的 z 位置。

## 【代码实现】

```
    m_timer = new QTimer(this);
    connect(m_timer, &QTimer::timeout, this, &GLWindow::changeBackground);
    m_timer->start(20000);
    count = 0;
    lz = Rg;
```

（构造函数中周期修改图像的代码及一些需要的变量的初始化代码）

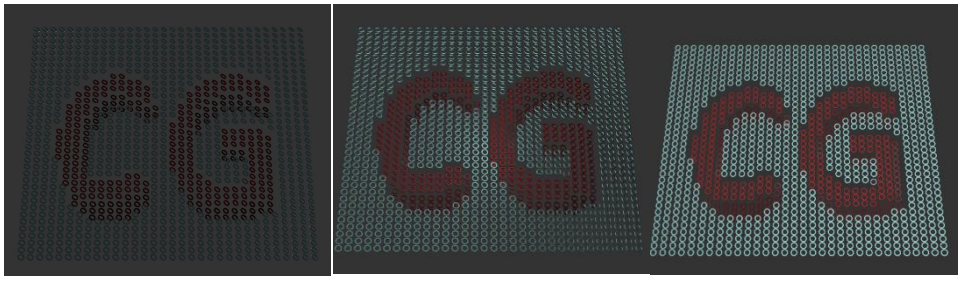
```
void GLWindow::changeBackground()
{
    count = (count+1) % 5;
    qDebug("Img %d",count);
    QImage img(QString(":/%1.png").arg(count));
    Q_ASSERT(!img.isNull());
    m_texture = new QOpenGLTexture(img.scaled(32, 36).mirrored());
    update();
}
```

（上述定时器对应的槽函数代码）

```
void GLWindow::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_1) {
        changeBackground();
    }
    else if(event->key() == Qt::Key_2){
        lz = (lz+5)%Rg;
        qDebug("lz=%d",lz-Rg/2);
        m_uniformsDirty = true;
        update();
    }
}
```

（按键检测实现代码）

## 【效果展示】



（负距离光照，近距离光照，足够距离光照）

详细按键功能请实际运行程序自行体验。

## 【个人实现内容 3】

## 协助完成光线追踪算法

内容大致包括各个 类 的定义及其中函数的实现，具体实现可看源代码、解释可看最后算法实现，此处大致展示各部分作用。

```
#define TOTALDEPTH 2 // 光线递归深度
#define INFINITY 1000000.0f
#define SMALL 0.0001f
enum INTERSECTION_TYPE {INTERSECTED_IN = -1,MISS = 0,INTERSECTED = 1};
```

（一些简化程序的宏定义）

```
class Ray
{
public:
    Ray();
    Ray(const QVector3D& org, const QVector3D& drct);

    const QVector3D& get_origin() const;
    const QVector3D& get_direction() const;
    void set_origin(const QVector3D& org);
    void set_direction(const QVector3D& drct);
    QVector3D get_point(float d) const;

private:
    QVector3D m_origin;
    QVector3D m_direction;
};
```

（光线类，便于追踪算法中获取入射方向、反射方向）

```
class Object
{
public:
    Object();
    float get_spec();
    float get_refl();
    float get_diffuse();
    void reset(float spec, float refl);
    virtual QVector3D get_color(QVector3D position) = 0;
    virtual QVector3D get_normal(QVector3D point) = 0;                  // 获取物体表面一点的法线
    virtual INTERSECTION_TYPE is_intersected(Ray ray,float& dst) = 0;   // 判断光线是否与物体相交
    int style;
protected:
    float m_spec;                                                       // 镜面反射强度
    float m_refl;                                                       // 环境反射强度
};
```

（物体类，获取物体法线等）

```
class Help
{
public:
    Help();
    ~Help();

    void init();
    int get_object_count();
    int get_light_count();
    void set_ambient_light(QColor amb);
    QColor get_ambient_light();
    Object* get_object(int idx);
    PointLight get_light(int idx);
    Object** m_obj;
    PointLight* m_light;
    int loadHelp(char* filename);
    void parse_check(char* expected, char* found);
    void parse_doubles(FILE* file, char* check, double p[3]);
    void parse_rad(FILE* file, double* r);
    void parse_shi(FILE* file, double* shi);

private:
    int m_object_count;
    int m_light_count;
    QColor ambient_light;
};
```

（用于最终帮助实现追踪算法的类）

## 三、21307071 詹迪个人实现

**【个人实现内容 1】**

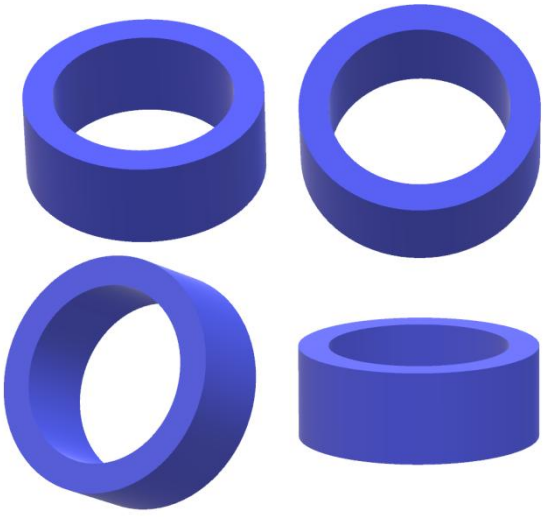
**自定义“体素”——三维模型中基本单元的设计与实现**

我们的项目将静态的二维图像重建为动态的三维模型，而三维模型以我们自定义的“体素”来构建。每个自定义的“体素”，其实就是我们自己设计的简单三维几何体。通过这些小的简单三维几何体的“堆砌”，从而构建出二维图像的三维模型。

为什么要使用自定义的三维几何体来构建三维模型呢？

这是因为，我们的项目并不追求将二维图片重建为三维模型的逼真效果（因为这可能需要使用到深度学习算法和庞大的模型训练），我们追求的是，根据二维图片重建出基本的三维轮廓，所以不需要非常精细的体素，简单的三维几何体即可。而且，使用自定义的几何体来构建三维模型，三维模型中的每一个“体素”都可以进行一些旋转之类的动态视觉效果，这是我们想要的。

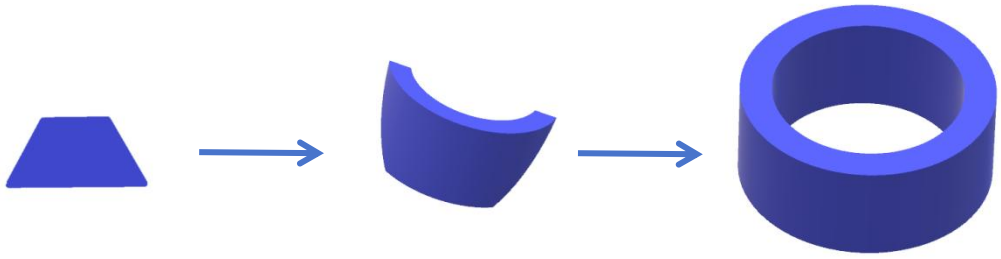
在这里，我们将自定义“体素”设计为圆柱环，具体设计见图 1。



（图 1：自定义“体素”设计图）

【实现思路】

为了构建如图 1 的圆柱环，我们可以按照以下步骤来进行：首先，在二维平面中，构建一个四边形（使用两个三角形拼接而成的梯形），然后在三维空间中，通过拉伸将梯形挤压成一个具有高度和深度的扇形柱面。使用若干个小扇形柱面，就可以拼接成圆柱环。并且，扇形柱面的个数越多，圆柱环平滑度越高。实现流程如图 2 所示



（图 2：圆柱环的构建过程）

【代码实现】

①定义了一个名为 Element 的类，用于创建和管理图形元素。

私有成员变量：

• m_data：存储顶点信息的 QVector 容器。

• m_count：存储顶点数量的整型变量。

公共成员函数：

• constData()：返回一个指向常量 GLfloat 数据的指针。m_data 是存储数据的 QVector 容器，constData()函数返回容器中数据的指针。

• count()：返回顶点数据的数量。

• vertexCount()：返回顶点的数量，通过将顶点数据数量除以 6 计算。

私有成员函数：

• quad()：根据给定的四个点的坐标，生成一个四边形。

• extrude()：根据给定的两个点的坐标，挤压出一个侧面。

• add()：将顶点和法向量添加到数据容器中。

```
#ifndef ELEMENT_H
#define ELEMENT_H

#include <qopengl.h>
#include <QVector>
#include <QVector3D>

class Element
{
public:
    Element();
    const GLfloat *constData() const { return m_data.constData(); }
    int count() const { return m_count; }
    int vertexCount() const { return m_count / 6; }

private:
    void quad(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2, GLfloat x3, GLfloat y3, GLfloat x4, GLfloat y4);
    void extrude(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);
    void add(const QVector3D &v, const QVector3D &n);

    QVector<GLfloat> m_data;
    int m_count;
};

#endif // ELEMENT_H
```

（图 3：Element 类定义）

②在 Element 类的构造函数中：首先使用成员初始化列表来初始化 m_count，并设置为 0。然后调整了 m_data 的大小，以容纳 2500 个元素，每个元素包含 6 个坐标值。接着，定义了一个常量 NumSectors，表示在创建元素形状时将被分割的扇区数量。

开始循环迭代 NumSectors 次。在每次迭代中，计算了两个角度 angle 和 angleSin、angleCos，然后根据这些角度计算了四个点 (x5, y5)、(x6, y6)、(x7, y7)、(x8, y8) 的坐标值。同时，调用 quad 函数创建一个四边形，并调用两次 extrude 函数分别对两个边进行挤压（extrude），从而创建了一个立体形状（圆柱环）。构造函数的具体实现见下图。

```
Element::Element()
    : m_count(0)
{
    m_data.resize(2500 * 6);

    const int NumSectors = 100;
    for (int i = 0; i < NumSectors; ++i) {
        GLfloat angle = (i * 2 * M_PI) / NumSectors;
        GLfloat angleSin = qSin(angle), angleCos = qCos(angle);     ⚠ implicit conversion loses floating-point pre
        const GLfloat x5 = 0.30f * angleSin, y5 = 0.30f * angleCos, x6 = 0.20f * angleSin, y6 = 0.20f * angleCos;

        angle = ((i + 1) * 2 * M_PI) / NumSectors;
        angleSin = qSin(angle);     ⚠ implicit conversion loses floating-point precision: 'qreal' (aka 'double') t
        angleCos = qCos(angle);     ⚠ implicit conversion loses floating-point precision: 'qreal' (aka 'double') t

        const GLfloat x7 = 0.20f * angleSin, y7 = 0.20f * angleCos, x8 = 0.30f * angleSin, y8 = 0.30f * angleCos;
        quad(x5, y5, x6, y6, x7, y7, x8, y8);
        extrude(x6, y6, x7, y7);
        extrude(x8, y8, x5, y5);
    }
}
```

（图 3：Element 构造函数的具体实现）

③在 quad 函数中：首先，通过 QVector3D::normal 函数计算出第一个三角形的法线向量 n。再分别将两个三角形的三个顶点的坐标以及法线向量添加到元素的数据中。这样，整个 quad 函数完成了一个由两个三角形组成的四边形的创建，并将这些点的坐标和法线信息添加到元素的数据中。quad 函数的具体实现见图 4。

```
void Element::quad(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2, GLfloat x3, GLfloat y3, GLfloat x4, GLfloat y4)
{
    QVector3D n = QVector3D::normal(QVector3D(x4 - x1, y4 - y1, 0.0f), QVector3D(x2 - x1, y2 - y1, 0.0f));

    add(QVector3D(x1, y1, -0.05f), n);
    add(QVector3D(x4, y4, -0.05f), n);
    add(QVector3D(x2, y2, -0.05f), n);

    add(QVector3D(x3, y3, -0.05f), n);
    add(QVector3D(x2, y2, -0.05f), n);
    add(QVector3D(x4, y4, -0.05f), n);

    n = QVector3D::normal(QVector3D(x1 - x4, y1 - y4, 0.0f), QVector3D(x2 - x4, y2 - y4, 0.0f));

    add(QVector3D(x4, y4, 0.05f), n);
    add(QVector3D(x1, y1, 0.05f), n);
    add(QVector3D(x2, y2, 0.05f), n);

    add(QVector3D(x2, y2, 0.05f), n);
    add(QVector3D(x3, y3, 0.05f), n);
    add(QVector3D(x4, y4, 0.05f), n);
}
```

（图 4：quad 函数的具体实现）

④extrude 函数用于对两个点所确定的线段进行挤压（extrude），从而创建一个有厚度的立体形状：首先，通过 QVector3D::normal 函数计算两个向量的法线向量 n，这里一个向量是 (0.0f, 0.0f, -0.1f)，表示挤压方向，另一个向量是 (x2 - x1, y2 - y1, 0.0f)，表示挤压的平面。接着，调用 add 函数，将上表面的三个顶点 (x1, y1, +0.05f)、(x1, y1,

-0.05f)、(x2, y2, +0.05f) 和法线向量 n 添加到 m_data 中。最后，再次调用 add 函数，将下表面的三个顶点 (x2, y2, -0.05f)、(x2, y2, +0.05f)、(x1, y1, -0.05f) 和法线向量 n 添加到 m_data 中。extrude 函数的具体实现见图 5。

```
61  void Element::extrude(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2)
62  {
63      QVector3D n = QVector3D::normal(QVector3D(0.0f, 0.0f, -0.1f), QVector3D(x2 - x1, y2 - y1, 0.0f));
64
65      add(QVector3D(x1, y1, +0.05f), n);
66      add(QVector3D(x1, y1, -0.05f), n);
67      add(QVector3D(x2, y2, +0.05f), n);
68
69      add(QVector3D(x2, y2, -0.05f), n);
70      add(QVector3D(x2, y2, +0.05f), n);
71      add(QVector3D(x1, y1, -0.05f), n);
72  }
73
```

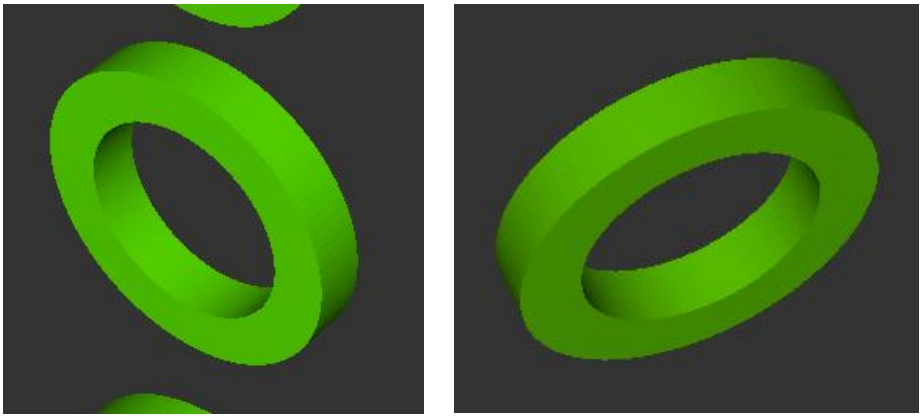
（图 5：extrude 函数的具体实现）

⑤add 函数用于将一个顶点的坐标和法线向量添加到 m_data 中。add 函数的实现比较简单：

```
26  void Element::add(const QVector3D &v, const QVector3D &n)
27  {
28      GLfloat *p = m_data.data() + m_count;
29      *p++ = v.x();
30      *p++ = v.y();
31      *p++ = v.z();
32      *p++ = n.x();
33      *p++ = n.y();
34      *p++ = n.z();
35      m_count += 6;
36  }
```

（图 6：add 函数的实现）

**【实现效果】**



（图 7：Element 类实例效果展示）

**【个人实现内容 2】**

**多视角下，三维模型的多角度旋转动画**

通过自定义的“体素”建立出完整的三维模型（由另一组员完成）后，我们需要在多个不同的视角下，将三维模型进行多角度旋转\翻转，然后创建并配置动画。

**【代码实现】**

①创建并配置动画：

• 创建 QSequentialAnimationGroup 对象 animGroup，表示一组按顺序播放的动画。

•创建 QPropertyAnimation 对象 zAnim0，对 zValue 属性进行动画。

•添加 zAnim0 到 animGroup，设置其开始值、结束值、持续时间等。

• 同样方式创建 zAnim1 和 zAnim2，分别表示不同的动画阶段。

• 使用 QEasingCurve::OutElastic 设置弹性缓动效果。

• 使用 start 启动整个动画组 animGroup。

②创建和启动循环旋转动画：

• 创建 QPropertyAnimation 对象 rotationAnim，对 rotation 属性进行循环旋转的动画。

• 设置开始值、结束值、持续时间和循环次数。

• 启动旋转动画。

这两部分的代码使用了 Qt 的动画框架，通过创建 QPropertyAnimation 和 QSequentialAnimationGroup 对象，配置不同的属性动画，并通过 start() 或 start() 函数启动这些动画。这些动画

在后台按照设定的参数进行播放，实现了视觉上的动态效果。在 OpenGL 渲染环境中，这些动画效果可以使得渲染的图形具有生动的动态感。动画创建和配置的具体代码如下：

```
32      QSequentialAnimationGroup *animGroup = new QSequentialAnimationGroup(this);
33      animGroup->setLoopCount(-1);
34
35      QPropertyAnimation *zAnim0 = new QPropertyAnimation(this, QByteArrayLiteral("zValue"));
36      zAnim0->setStartValue(1.5f);
37      zAnim0->setEndValue(10.0f);
38      zAnim0->setDuration(2000);
39      animGroup->addAnimation(zAnim0);
40
41      QPropertyAnimation *zAnim1 = new QPropertyAnimation(this, QByteArrayLiteral("zValue"));
42      zAnim1->setStartValue(10.0f);
43      zAnim1->setEndValue(50.0f);
44      zAnim1->setDuration(4000);
45      zAnim1->setEasingCurve(QEasingCurve::OutElastic);
46      animGroup->addAnimation(zAnim1);
47
48      QPropertyAnimation *zAnim2 = new QPropertyAnimation(this, QByteArrayLiteral("zValue"));
49      zAnim2->setStartValue(50.0f);
50      zAnim2->setEndValue(1.5f);
51      zAnim2->setDuration(2000);
52      animGroup->addAnimation(zAnim2);
53
54      animGroup->start();
55
56      QPropertyAnimation* rotationAnim = new QPropertyAnimation(this, QByteArrayLiteral("rotation"));
57      rotationAnim->setStartValue(0.0f);
58      rotationAnim->setEndValue(360.0f);
59      rotationAnim->setDuration(2000);
60      rotationAnim->setLoopCount(-1);
61      rotationAnim->start();
```

（图 8：创建并配置动画）

# 四、小组协作实现

## 【合作实现内容】

### 合作完成三维模型的全局光线追踪算法

在我们的项目中，将二维图像构建成三维模型、并用自定义着色器进行初步上色后，我们还加入了全局光线追踪算法，对三维模型的每一个像素进行光线追踪，实现颜色的最终更新。

光线追踪（Ray tracing）是一种用于渲染图像的计算机图形技术。它模拟了光线在场景中的传播和相互作用，以生成逼真的图像。光线追踪通过跟踪从摄像机发出的光线，并模拟它们在场景中的反射、折

射和散射等现象来计算每个像素的颜色值。

光线追踪的基本原理是从视点发射光线，当光线与场景中的物体相交时，根据材质属性计算出交点处的颜色。为了获取逼真的结果，光线会继续向前追踪并与其他物体相交，直到达到最大反射次数或遇到光源为止。

相对于传统的渲染方法，光线追踪能够更准确地模拟光线的传播和反射行为，因此可以生成逼真的阴影、反射、折射和光照效果。

**【实现思路】**

①发射光线：

• 通过计算像素位置和相机参数，确定从相机出发穿过像素的光线。

• 生成主光线，它从相机位置穿过图像平面的特定像素。

②光线与物体相交：

• 对场景中的每个物体执行碰撞检测。

• 使用光线和物体的几何形状进行相交测试，找到与光线最近相交的物体表面，并记录相交点的位置和法线等信息。

③光线与物体表面相交后的处理：

• 获取相交点处的表面属性，例如法线方向、表面颜色、材质属性等。

• 根据光线的方向和相交点的表面属性，计算反射、折射或吸收。

• 根据材质属性和光线入射角计算反射和折射光线的方向。

④递归追踪：

• 当计算出反射或折射光线后，以相交点为起点，继续发射新的光线。这些新的光线按照反射方向或折射方向在场景中传播，再次进行相交

测试。递归地进行此过程，直到达到最大递归深度或满足终止条件。这一过程可能会经过多次反射和折射，直到达到最大递归深度或能量衰减到一定阈值。

⑤追踪深度控制：

• 设置最大递归深度，以防止无限递归。光线追踪通常有一个最大的递归深度限制阈值。可以根据场景的复杂性和需求进行调整，以平衡渲染质量和性能。

⑥采样与积累：

• 为了减少图像中的噪点，对于每个像素进行多次采样。每次采样略微改变光线的方向或相机位置。

•将多次采样得到的颜色值进行累积平均，以获得最终像素的颜色值。

⑦生成图像：

• 当所有像素都进行了光线追踪和采样后，将得到的颜色值合成为最终的图像。可以应用色彩校正、色调映射等技术，增强图像的视觉效果和真实感。

**【代码实现】**

光线追踪算法在 trace 函数中实现，具体实现过程如下：

①初始化变量：初始化一些变量，包括光线与物体相交的距离、递归深度、相交的物体（aim）、相交点的位置（point）、法线（n）、光线方向（l）、观察方向（v）以及最终的颜色值（hit_color）。

```
184 QColor GLWindow::trace(Ray ray, int depth)
185 {
186     float distance = INFINITY;
187     int dep = depth;
188     Object* aim = nullptr;
189     QVector3D point, n, l, v;
190     QColor hit_color = m_scn->get_ambient_light();
191     int res_hit = 0;
```

②物体相交检测：

循环遍历场景中的每个物体，调用 is_intersected 方法检测光线是否与物体相交，并记录最近相交的物体和相交的距离。

```
193     for (int k = 0; k < m_scn->get_object_count(); k++)
194     {
195         Object* obj = m_scn->get_object(k);
196         int res;
197         if (res = obj->is_intersected(ray, distance))
198         {
199             aim = obj;
200             res_hit = res;
201         }
202     }
```

③相交点的处理：

如果存在相交点，则获取相交点处的表面属性，例如法线方向、表面颜色、材质属性等。根据光线的方向和相交点的表面属性，计算反射、折射或吸收。根据材质属性和光线入射角计算反射和折射光线的方向。

```
203
204     if (distance != INFINITY)
205     {
206         point = ray.get_point(distance);
207         n = aim->get_normal(point);
208         n.normalize();
209
210         if (res_hit == INTERSECTED_IN)
211         {
212             n = -n;
213         }
214
215         v = m_eye - point;
216         v.normalize();
217
218         for (int k = 0; k < m_scn->get_light_count(); k++)
219         {
220             PointLight pl = m_scn->get_light(k);
221             l = pl.get_position() - point;
222             l.normalize();
223
224             Ray l_ray = Ray(point + l * SMALL, l);
225
226             float shade = 1.0f;
227             float distance = INFINITY;
228
229             for (int k = 0; k < m_scn->get_object_count(); k++)
230             {
231                 Object* obj = m_scn->get_object(k);
232                 if (obj->is_intersected(l_ray, distance))
233                 {
234                     shade = 0.0f;
235                     break;
236                 }
237             }
238
```

④光照计算：

对于每个光源，计算光照效果，包括漫反射和镜面反射。

- 根据物体的漫反射系数和光照方向计算漫反射光照的贡献。
- 根据物体的镜面反射系数和观察方向计算镜面反射光照的贡献。

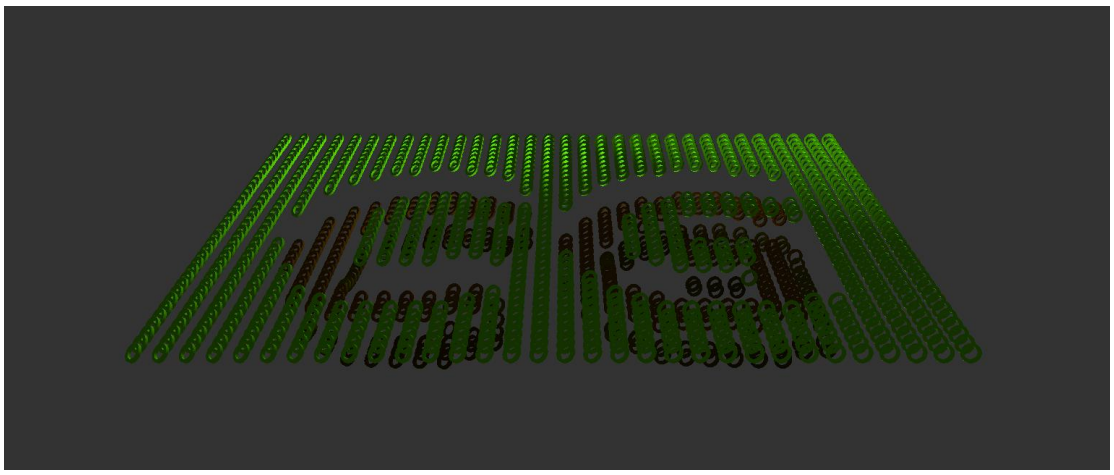
```
239             if (aim->get_diffuse() > 0)
240             {
241                 float cos = QVector3D::dotProduct(l, n);
242                 if (cos > 0)
243                 {
244                     float diffuse = cos * aim->get_diffuse() * shade;
245                     diffuse++;
246 //                    hit_color = hit_color + diffuse * pl.get_color() * aim->get_color(point);
247                 }
248             }
249
250             if (aim->get_spec() > 0)
251             {
252                 QVector3D h = 2 * QVector3D::dotProduct(n, l) * n - l;
253                 float cos = QVector3D::dotProduct(h, v);
254                 if (cos > 0)
255                 {
256                     float specular = powf(cos, 20) * aim->get_spec() * shade;
257                     specular++;
258 //                    hit_color = hit_color + specular * pl.get_color();
259                 }
260             }
261         }
```

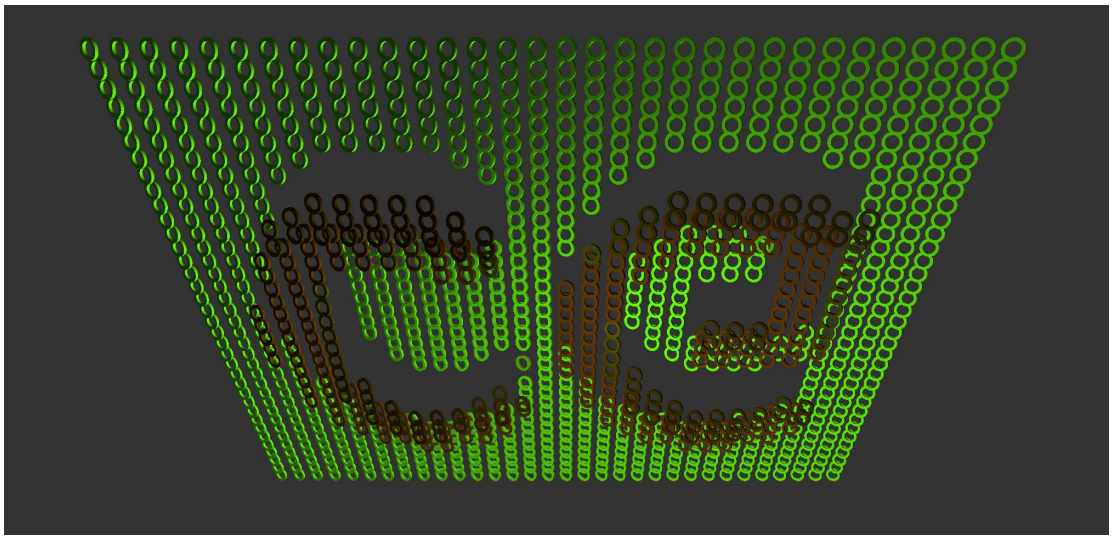
⑤递归追踪：如果物体有反射属性且递归深度未达到设定的最大深

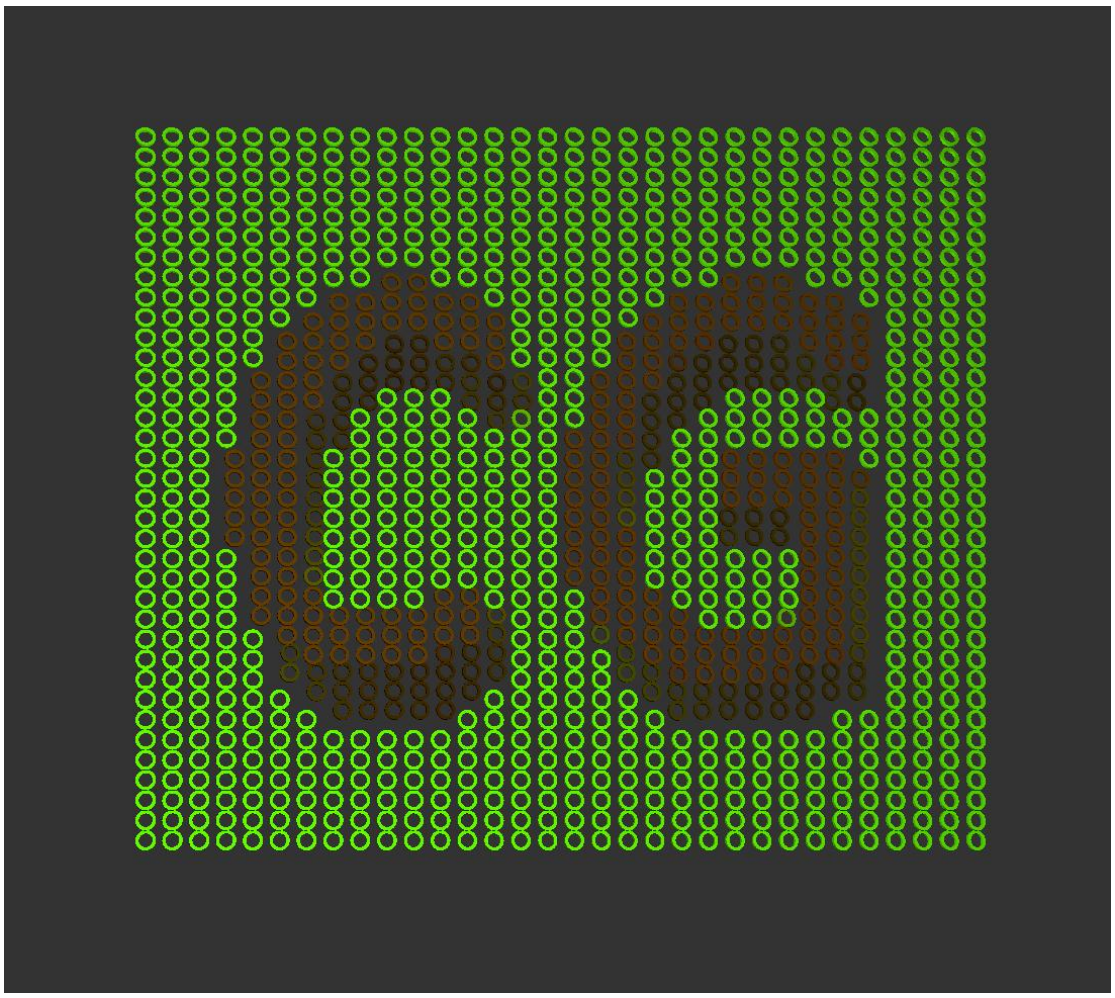
度，则递归追踪反射光线。最终返回计算得到的颜色值即可。

```
263         if (aim->get_refl() > 0 && dep < TOTALDEPTH)
264         {
265             QVector3D refl = 2 * QVector3D::dotProduct(n, l) * n - l;
266             refl = 2*refl;
267 //            hit_color = hit_color + trace(Ray(point + refl * 0.0001f, refl), ++dep);
268         }
269     }
270     return hit_color;
271 }
272
```

【实现效果】







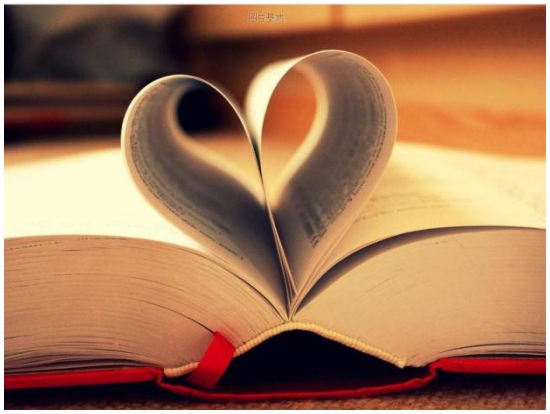
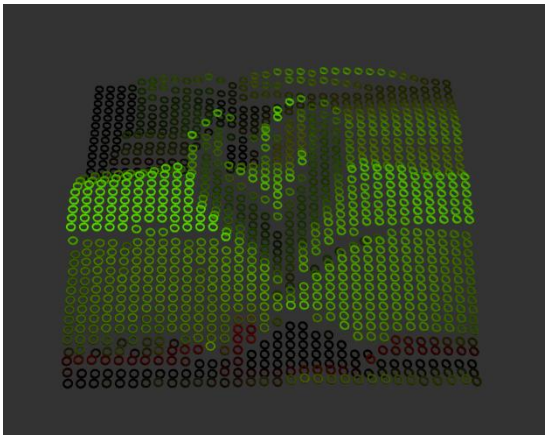
可以很明显的看到，当三维模型以不同角度旋转、光线发生变化时，三维模型显示出更逼真的光照和阴影效果，更加自然的明暗分割。

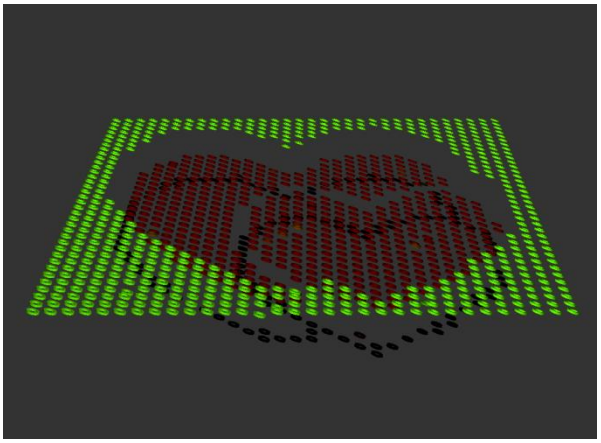
## 五、效果展示

整个项目以动画播放和交互的形式进行，总体效果见 video.mp4

这里，我们展示几个三维模型动画的截图。





（注：左侧为二维图片，右侧为我们的项目的运行截图：二维图片的三维重建动画）