

浙江大学



《计算机图形学》 实验报告

实验名称 :	Solar System
姓 名 :	王晓宇
学 号 :	3220104364
电子邮箱 :	3220104364@zju.edu.cn
联系电话 :	19550222634
授课教师 :	吴鸿智
助 教 :	丁华铿

2024 年 12 月 4 日

Dream Car

1 实验内容及简要原理介绍

1.1 实验内容

1.2 简要原理介绍

1.2.1 贝塞尔曲面的定义

2 实验框架思路与代码实现

2.1 实验框架思路

2.1.1 `main` 函数

2.1.2 `framebuffer_size_callback` 函数

2.1.3 `processInput` 函数

2.1.4 `mouse_callback` 函数

2.2 代码实现

2.2.1 `camera.h`

2.2.2 `MyObj.hpp`

2.2.3 `MyBezier.hpp`

2.2.4 `Main.cpp`

3 实验结果与分析

4 思考题

4.1 Obj模型的优缺点

4.1.1 优点

4.1.2 缺点

4.2 贝塞尔曲面的优缺点

4.2.1 优点


4.2.2 缺点

Dream Car

1 实验内容及简要原理介绍

AS#7: Design Your Dream Car

- Program using **2** modeling techniques in today's class to build the shape of your dream car
- Compare the strengths and weaknesses of the techniques you use
- Render your car in the same viewer as in the solar system assignment
- No appearance specification needed, just the shape
- Bonus (10%):
 - Use **3** modeling techniques
- Due: 12/04



1.1 实验内容

本次实验旨在通过OpenGL和相关库实现一个简单的汽车模拟程序。程序将展示一辆汽车，以及我们的视角移动变换下的场景。

此外还要求实现两种建模技术的实现，本文实现了课上所讲的Obj模型导入与贝塞尔曲面实现。

1.2 简要原理介绍

OpenGL(Open Graphics Library)是一个跨语言、跨平台的图形API，用于渲染2D和3D矢量图形。本实验中，我们利用OpenGL的3D图形渲染能力，创建了一个汽车模拟场景。

OBJ (Object) 是一种常见的3D模型文件格式，广泛用于存储几何数据（如顶点、法线、纹理坐标等）。OBJ文件通常包含顶点坐标、法线、纹理坐标和面信息。

贝塞尔曲面是一种广泛应用于计算机图形学中的参数曲面，用于生成平滑的曲面。贝塞尔曲面通常由控制点和贝塞尔曲线生成。贝塞尔曲面广泛应用于计算机图形学中的曲面建模、动画、字体设计等领域。例如，在3D建模软件中，贝塞尔曲面常用于创建平滑的曲面模型。

1.2.1 贝塞尔曲面的定义

贝塞尔曲面由两组贝塞尔曲线生成，通常使用双参数 (u, v) 来表示曲面上的点。对于 $m \times n$ 阶贝塞尔曲面，控制点为 P_{ij} ，曲面上任意一点 $S(u, v)$ 可以表示为：

$$S(u, v) = \sum_{i=0}^m \sum_{j=0}^n P_{ij} \cdot B_i^m(u) \cdot B_j^n(v)$$

其中， $B_i^m(u)$ 和 $B_j^n(v)$ 是Bernstein函数。

2 实验框架思路与代码实现

2.1 实验框架思路

2.1.1 main函数

`main` 函数是整个程序的入口点，负责初始化GLFW、创建窗口、加载OpenGL函数指针、设置视口、加载着色器、创建几何体、加载模型，并在主循环中渲染场景。

实现思路：

1. 初始化GLFW：调用 `glfwInit()` 初始化GLFW库，并设置OpenGL版本为3.3，使用核心模式。
2. 创建窗口：调用 `glfwCreateWindow` 创建一个窗口，调用 `glfwMakeContextCurrent` 将窗口的上下文设置为当前线程的上下文。
3. 调用 `gladLoadGLLoader` 加载OpenGL函数指针。
4. 设置视口：调用 `glViewport` 设置视口大小，启用深度测试、点大小、混合等功能。
5. 设置回调函数：调用 `glfwSetFramebufferSizeCallback` 设置窗口大小变化回调函数，调用 `glfwSetCursorPosCallback` 设置鼠标移动回调函数。
6. 加载着色器：创建 `Shader` 对象，加载顶点和片段着色器。

这里介绍一下引入的vertex.glsl和fragment.glsl，两个文件均使用GLSL（OpenGL着色语言）编写。

顶点着色器的主要作用是处理每个顶点的属性，并将这些属性传递给后续片段着色器的渲染管线阶段，这个顶点着色器的主要作用是将顶点的位置从模型空间转换到裁剪空间，同时计算并传递片段位置、纹理坐标和法线。

这里的片段着色器主要作用是计算每个片段的光照效果。它接收顶点着色器传递的法线和片段位置，以及统一变量传递的观察者位置和光源信息。通过调用不同的光照计算函数，它计算定向光、点光源和聚光灯的光照效果，并将结果累加起来，最终得到片段的颜色，实现了一个基本的光照模型，可以用于渲染具有多种光源的3D场景。

7. 创建 `MyObj` 对象，加载obj模型。
8. 创建 `MyBezier` 对象，加载贝塞尔曲面，紧接着完成参数初始化。
9. 在主循环中，处理输入、计算帧时间、清空颜色和深度缓冲、设置视图和投影矩阵、渲染模型，并交换缓冲区。
10. 在程序结束时，释放资源，终止GLFW。

2.1.2 `framebuffer_size_callback` 函数

`framebuffer_size_callback` 函数在窗口大小变化时被调用，用于调整视口大小以适应新的窗口尺寸。

实现思路：调用 `glViewport` 设置视口大小为新的窗口尺寸。

2.1.3 `processInput` 函数

`processInput` 函数处理键盘输入，包括关闭窗口和控制相机移动。

实现思路：

- 检查 `GLFW_KEY_ESCAPE` 键是否被按下，如果是，则设置窗口关闭标志。
- 检查 `GLFW_KEY_W`、`GLFW_KEY_S`、`GLFW_KEY_A`、`GLFW_KEY_D` 键是否被按下，如果是，则调用 `camera.ProcessKeyboard` 函数控制相机移动。

2.1.4 `mouse_callback` 函数

`mouse_callback` 函数处理鼠标移动事件，用于控制相机视角。

实现思路：

- 计算鼠标当前位置与上一帧位置的偏移量。
- 更新上一帧的鼠标位置。

- 调用 `camera.ProcessMouseMovement` 函数处理鼠标移动，更新相机视角。

2.2 代码实现

部分调用库的头文件设计在`tool`中，这里介绍比较重要的`camera.h`、`MyObj.hpp`和`MyBezier.hpp`

2.2.1 camera.h

```
1  #ifndef CAMERA_H
2  #define CAMERA_H
3
4  #include <glad/glad.h>
5  #include <glm/glm.hpp>
6  #include <glm/gtc/matrix_transform.hpp>
7
8  #include <vector>
9
10 // Defines several possible options for camera movement. Used as
    abstraction to stay away from window-system specific input methods
11 enum Camera_Movement
12 {
13     FORWARD,
14     BACKWARD,
15     LEFT,
16     RIGHT
17 };
18
19 // Default camera values
20 const float YAW = -90.0f;
21 const float PITCH = 0.0f;
22 const float SPEED = 2.5f;
23 const float SENSITIVITY = 0.1f;
24 const float ZOOM = 45.0f;
25
```

```

26 // An abstract camera class that processes input and calculates
    the corresponding Euler Angles, Vectors and Matrices for use in
    OpenGL
27 class Camera
28 {
29 public:
30     // camera Attributes
31     glm::vec3 Position;
32     glm::vec3 Front;
33     glm::vec3 Up;
34     glm::vec3 Right;
35     glm::vec3 WorldUp;
36     // euler Angles
37     float Yaw;
38     float Pitch;
39     // camera options
40     float MovementSpeed;
41     float MouseSensitivity;
42     float Zoom;
43
44     // constructor with vectors
45     Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f),
        glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f), float yaw = YAW, float
        pitch = PITCH) : Front(glm::vec3(0.0f, 0.0f, -1.0f)),
        MovementSpeed(SPEED), MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
46     {
47         Position = position;
48         WorldUp = up;
49         Yaw = yaw;
50         Pitch = pitch;
51         updateCameraVectors();
52     }
53     // constructor with scalar values

```

```

54     Camera(float posX, float posY, float posZ, float upX, float
upY, float upZ, float yaw, float pitch) : Front(glm::vec3(0.0f,
0.0f, -1.0f)), MovementSpeed(SPEED),
MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
55     {
56         Position = glm::vec3(posX, posY, posZ);
57         WorldUp = glm::vec3(upX, upY, upZ);
58         Yaw = yaw;
59         Pitch = pitch;
60         updateCameraVectors();
61     }
62
63     // returns the view matrix calculated using Euler Angles and
the LookAt Matrix
64     glm::mat4 GetViewMatrix()
65     {
66         return glm::lookAt(Position, Position + Front, Up);
67     }
68
69     // processes input received from any keyboard-like input
system. Accepts input parameter in the form of camera defined ENUM
(to abstract it from windowing systems)
70     void ProcessKeyboard(Camera_Movement direction, float
deltaTime)
71     {
72         float velocity = MovementSpeed * deltaTime;
73         if (direction == FORWARD)
74             Position += Front * velocity;
75         if (direction == BACKWARD)
76             Position -= Front * velocity;
77         if (direction == LEFT)
78             Position -= Right * velocity;
79         if (direction == RIGHT)
80             Position += Right * velocity;
81
82         // Position.y = 0.0f;

```



```

83     }
84
85     // processes input received from a mouse input system. Expects
the offset value in both the x and y direction.
86     void ProcessMouseMovement(float xoffset, float yoffset,
GLboolean constrainPitch = true)
87     {
88         xoffset *= MouseSensitivity;
89         yoffset *= MouseSensitivity;
90
91         Yaw += xoffset;
92         Pitch += yoffset;
93
94         // make sure that when pitch is out of bounds, screen
doesn't get flipped
95         if (constrainPitch)
96         {
97             if (Pitch > 89.0f)
98                 Pitch = 89.0f;
99             if (Pitch < -89.0f)
100                 Pitch = -89.0f;
101         }
102
103         // update Front, Right and Up Vectors using the updated
Euler angles
104         updateCameraVectors();
105     }
106
107     // processes input received from a mouse scroll-wheel event.
only requires input on the vertical wheel-axis
108     void ProcessMouseScroll(float yoffset)
109     {
110         Zoom -= (float)yoffset;
111         if (Zoom < 1.0f)
112             Zoom = 1.0f;
113         if (Zoom > 45.0f)

```

```

114         Zoom = 45.0f;
115     }
116
117 private:
118     // calculates the front vector from the Camera's (updated)
    Euler Angles
119     void updateCameraVectors()
120     {
121         // calculate the new Front vector
122         glm::vec3 front = glm::vec3(1.0f);
123         front.x = cos(glm::radians(Yaw)) *
cos(glm::radians(Pitch));
124         front.y = sin(glm::radians(Pitch));
125         front.z = sin(glm::radians(Yaw)) *
cos(glm::radians(Pitch));
126         Front = glm::normalize(front);
127         // also re-calculate the Right and Up vector
128         Right = glm::normalize(glm::cross(Front, worldUp)); //
normalize the vectors, because their length gets closer to 0 the
more you look up or down which results in slower movement.
129         Up = glm::normalize(glm::cross(Right, Front));
130     }
131 };
132 #endif

```

`camera.h` 文件定义了一个名为 `Camera` 的类，用于处理摄像机的移动、视角变换和视图矩阵的计算。以下是对文件内容的详细分析：

- 包含的库：
 - `glad/glad.h`：OpenGL函数加载库。
 - `glm/glm.hpp` 和 `glm/gtc/matrix_transform.hpp`：GLM数学库，用于处理向量、矩阵等数学运算。
 - `vector`：标准库中的向量容器。

- 摄像机移动枚举：定义了摄像机移动的四种方向：`FORWARD`（前进）、`BACKWARD`（后退）、`LEFT`（左移）和 `RIGHT`（右移）。

```
1 enum Camera_Movement
2 {
3     FORWARD,
4     BACKWARD,
5     LEFT,
6     RIGHT
7 };
```

- 默认摄像机参数：
 - `YAW`：偏航角，默认值为 `-90.0f`。
 - `PITCH`：俯仰角，默认值为 `0.0f`。
 - `SPEED`：移动速度，默认值为 `2.5f`。
 - `SENSITIVITY`：鼠标灵敏度，默认值为 `0.1f`。
 - `ZOOM`：缩放级别，默认值为 `45.0f`。

- 类 `Camera`

```
1 class Camera
2 {
3 public:
4     // camera Attributes
5     glm::vec3 Position;
6     glm::vec3 Front;
7     glm::vec3 Up;
8     glm::vec3 Right;
9     glm::vec3 worldUp;
10    // euler Angles
11    float Yaw;
12    float Pitch;
13    // camera options
14    float MovementSpeed;
15    float MouseSensitivity;
16    float Zoom;
```

- 成员变量：
 - `Position`：摄像机位置，类型为 `glm::vec3`。
 - `Front`：摄像机朝向，类型为 `glm::vec3`。
 - `Up`：摄像机上方向，类型为 `glm::vec3`。
 - `Right`：摄像机右方向，类型为 `glm::vec3`。
 - `WorldUp`：世界坐标系的上方向，类型为 `glm::vec3`。
 - `Yaw` 和 `Pitch`：偏航角和俯仰角，类型为 `float`。
 - `MovementSpeed`：移动速度，类型为 `float`。
 - `MouseSensitivity`：鼠标灵敏度，类型为 `float`。
 - `Zoom`：缩放级别，类型为 `float`。
- 构造函数：
 - 提供了两个构造函数，一个接受向量参数，另一个接受标量参数。
 - 初始化摄像机的位置、上方向、偏航角和俯仰角。
 - 调用 `updateCameraVectors` 函数更新摄像机的朝向、右方向和上方向。
- 获取视图矩阵：
 - 使用 `glm::lookAt` 函数计算视图矩阵。
 - `Position` 是摄像机的位置。
 - `Position + Front` 是摄像机的目标位置。
 - `Up` 是摄像机的上方向。
- 处理键盘输入：
 - 根据输入的方向（`FORWARD`、`BACKWARD`、`LEFT`、`RIGHT`）更新摄像机的位置。
 - `deltaTime` 是时间增量，用于控制移动速度。
 - 注释掉的代码 `Position.y = 0.0f;` 用于限制摄像机在Y轴上的移动。

```
1 void ProcessKeyboard(Camera_Movement direction, float
   deltaTime)
```

```

2  {
3      float velocity = MovementSpeed * deltaTime;
4      if (direction == FORWARD)
5          Position += Front * velocity;
6      if (direction == BACKWARD)
7          Position -= Front * velocity;
8      if (direction == LEFT)
9          Position -= Right * velocity;
10     if (direction == RIGHT)
11         Position += Right * velocity;
12
13     // Position.y = 0.0f;
14 }

```

- 处理鼠标移动:

- 根据鼠标的偏移量更新摄像机的偏航角和俯仰角。
- `constrainPitch` 参数用于限制俯仰角的范围, 防止摄像机翻转。
- 调用 `updateCameraVectors` 函数更新摄像机的朝向、右方向和上方向。

```

1  void ProcessMouseMovement(float xoffset, float yoffset,
    GLboolean constrainPitch = true)
2  {
3      xoffset *= MouseSensitivity;
4      yoffset *= MouseSensitivity;
5
6      Yaw += xoffset;
7      Pitch += yoffset;
8
9      if (constrainPitch)
10     {
11         if (Pitch > 89.0f)
12             Pitch = 89.0f;
13         if (Pitch < -89.0f)
14             Pitch = -89.0f;

```

```
15     }
16
17     updateCameraVectors();
18 }
```

- 处理鼠标滚轮：

- 根据鼠标滚轮的偏移量更新摄像机的缩放级别。
- 限制缩放级别的范围在 `1.0f` 到 `45.0f` 之间。

```
1 void ProcessMouseScroll(float yoffset)
2 {
3     Zoom -= (float)yoffset;
4     if (Zoom < 1.0f)
5         Zoom = 1.0f;
6     if (Zoom > 45.0f)
7         Zoom = 45.0f;
8 }
```

- 更新摄像机向量：

- 根据偏航角和俯仰角计算新的朝向向量 `Front`。
- 使用 `glm::cross` 函数计算右方向向量 `Right` 和上方向向量 `Up`。
- 对向量进行归一化处理。

```

1 void updateCameraVectors()
2 {
3     glm::vec3 front = glm::vec3(1.0f);
4     front.x = cos(glm::radians(Yaw)) *
cos(glm::radians(Pitch));
5     front.y = sin(glm::radians(Pitch));
6     front.z = sin(glm::radians(Yaw)) *
cos(glm::radians(Pitch));
7     Front = glm::normalize(front);
8
9     Right = glm::normalize(glm::cross(Front, worldUp));
10    Up = glm::normalize(glm::cross(Right, Front));
11 }

```

camera.h 文件实现了一个简单的摄像机类，用于处理摄像机的移动、视角变换和视图矩阵的计算。该类支持键盘输入、鼠标移动和鼠标滚轮输入，并提供了默认的摄像机参数。通过更新摄像机的位置和方向，可以实现第一人称视角的摄像机控制。

2.2.2 MyObj.hpp

```

1 #ifndef MyObj_HPP
2 #define MyObj_HPP
3
4 #include <glad/glad.h>
5 #include <glm/glm.hpp>
6 #include <glm/gtc/matrix_transform.hpp>
7
8 #include <tool/shader.h>
9
10 #include <string>
11 #include <vector>
12 using namespace std;
13 struct Obj_Vertex
14 {
15     glm::vec3 Position; // 顶点属性
16     glm::vec3 Normal; // 法线
17     glm::vec2 TexCoords; // 纹理坐标

```

```

18     glm::vec4 color;    // 颜色
19 };
20 class MyObj
21 {
22 public:
23     vector<Obj_Vertex> vertex;
24     glm::vec4 color;
25     unsigned int VAO;
26     unsigned int VBO;
27     vector<float> _vert;
28     vector<float> _tex;
29     vector<float> _norm;
30     // mesh Data
31     MyObj(const char *path, glm::vec4 _color = glm::vec4(0.04,
0.9, 0.84, 0.88))
32     {
33         this->color = _color;
34         loadModel(path);
35         // printf("load model success\n");
36         // printf("vertex size: %d\n", vertex.size());
37     }
38     // render the mesh
39     void Draw(Shader &shader)
40     {
41         glBindVertexArray(VAO);
42         glDrawArrays(GL_TRIANGLES, 0, static_cast<GLsizei>
(vertex.size()));
43         glBindVertexArray(0);
44     }
45     void loadModel(const char *path)
46     {
47         float x = 0.f, y = 0.f, z = 0.f;
48         string content;
49         ifstream fileStream(path, ios::in);
50         string str = "";
51

```



```

52     while (!fileStream.eof())
53     {
54         getline(fileStream, str);
55         if (str.compare(0, 2, "v ") == 0)
56         {
57             std::stringstream stream(str.erase(0, 1));
58             stream >> x;
59             _vert.push_back(x);
60             stream >> y;
61             _vert.push_back(y);
62             stream >> z;
63             _vert.push_back(z);
64         }
65         if (str.compare(0, 2, "vt") == 0)
66         {
67             std::stringstream stream(str.erase(0, 2));
68             stream >> x;
69             _tex.push_back(x);
70             stream >> y;
71             _tex.push_back(y);
72         }
73         if (str.compare(0, 2, "vn") == 0)
74         {
75             std::stringstream stream(str.erase(0, 2));
76             stream >> x;
77             _norm.push_back(x);
78             stream >> y;
79             _norm.push_back(y);
80             stream >> z;
81             _norm.push_back(z);
82         }
83         if (str.compare(0, 1, "f") == 0)
84         {
85             string oneCorner, v, t, n;
86             std::stringstream stream(str.erase(0, 2));
87             for (int i = 0; i < 3; i++)

```

```

88         {
89             getline(stream, oneCorner, ' ');
90             stringstream oneCornerSS(oneCorner);
91             getline(oneCornerSS, v, '/');
92             getline(oneCornerSS, t, '/');
93             getline(oneCornerSS, n, '/');
94
95             int Vert_index = (stoi(v) - 1) * 3;
96             int Tex_index = (stoi(t) - 1) * 2;
97             int Normal_index = (stoi(n) - 1) * 3;
98
99             Obj_Vertex _new_vertex;
100             _new_vertex.Position =
glm::vec3(_vert[Vert_index], _vert[Vert_index + 1],
_vert[Vert_index + 2]);
101             _new_vertex.TexCoords =
glm::vec2(_tex[Tex_index], _tex[Tex_index + 1]);
102             _new_vertex.Normal =
glm::vec3(_norm[Normal_index], _norm[Normal_index + 1],
_norm[Normal_index + 2]);
103             _new_vertex.Color = color;
104             vertex.push_back(_new_vertex);
105         }
106     }
107 }
108
109 glGenVertexArrays(1, &VAO);
110 glGenBuffers(1, &VBO);
111 // bind the vertex Array Object first, then bind and set
vertex buffer(s), and then configure vertex attributes(s).
112 glBindVertexArray(VAO);
113
114 glBindBuffer(GL_ARRAY_BUFFER, VBO);
115 glBufferData(GL_ARRAY_BUFFER, vertex.size() *
sizeof(Obj_Vertex), &vertex[0], GL_STATIC_DRAW);
116

```

```

117         glEnableVertexAttribArray(0);
118         glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
sizeof(Obj_Vertex), (void *)0);
119         // vertex normals
120         glEnableVertexAttribArray(1);
121         glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
sizeof(Obj_Vertex), (void *)offsetof(Obj_Vertex, Normal));
122         // vertex texture coords
123         glEnableVertexAttribArray(2);
124         glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
sizeof(Obj_Vertex), (void *)offsetof(Obj_Vertex, TexCoords));
125         // vertex color
126         glEnableVertexAttribArray(3);
127         glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE,
sizeof(Obj_Vertex), (void *)offsetof(Obj_Vertex, Color));
128
129         glBindBuffer(GL_ARRAY_BUFFER, 0);
130         glBindVertexArray(0);
131     }
132 };
133
134 #endif

```

`MyObj.hpp` 文件定义了一个名为 `MyObj` 的类，用于加载和渲染OBJ格式的3D模型。以下是对文件内容的详细分析：

- 包含的库：
 - `glad/glad.h`：OpenGL函数加载库。
 - `glm/glm.hpp` 和 `glm/gtc/matrix_transform.hpp`：GLM数学库，用于处理向量、矩阵等数学运算。
 - `tool/shader.h`：自定义的着色器工具库。
 - `string` 和 `vector`：标准库中的字符串和向量容器。
- 命名空间：`using namespace std;` 简化了标准库的使用。
- 结构体 `Obj_Vertex`

```

1 struct Obj_Vertex
2 {
3     glm::vec3 Position; // 顶点属性
4     glm::vec3 Normal;   // 法线
5     glm::vec2 TexCoords; // 纹理坐标
6     glm::vec4 Color;    // 颜色
7 };

```

◦ 顶点属性：

- `Position`：顶点位置，类型为 `glm::vec3`。
- `Normal`：顶点法线，类型为 `glm::vec3`。
- `TexCoords`：纹理坐标，类型为 `glm::vec2`。
- `Color`：顶点颜色，类型为 `glm::vec4`。

• 类 `MyObj`

```

1 class MyObj
2 {
3 public:
4     vector<Obj_Vertex> vertex;
5     glm::vec4 color;
6     unsigned int VAO;
7     unsigned int VBO;
8     vector<float> _vert;
9     vector<float> _tex;
10    vector<float> _norm;

```

• 成员变量：

- `vertex`：存储顶点数据的向量，类型为 `vector<Obj_Vertex>`。
- `color`：模型的颜色，类型为 `glm::vec4`。
- `VAO` 和 `VBO`：顶点数组对象和顶点缓冲对象的ID。
- `_vert`、`_tex` 和 `_norm`：临时存储顶点位置、纹理坐标和法线数据的向量。

• 构造函数：

- 接受一个OBJ文件路径和一个可选的颜色参数。

- 调用 `loadModel` 函数加载模型数据。
- 渲染函数：
 - 绑定顶点数组对象（VAO）。
 - 使用 `glDrawArrays` 函数绘制三角形网格。
 - 解绑VAO。
- 模型加载函数：
 - 打开OBJ文件并逐行读取内容。
 - 解析顶点位置（`v`）、纹理坐标（`vt`）和法线（`vn`），并存储在临时向量中。
 - 解析面（`f`）数据，将顶点、纹理坐标和法线组合成 `Obj_Vertex` 结构体，并存储在 `vertex` 向量中。
 - 生成并配置VAO和VBO，将顶点数据传递给OpenGL。
 - 启用顶点属性并设置顶点属性指针。

`MyObj.hpp` 文件实现了一个简单的OBJ模型加载和渲染类。它通过解析OBJ文件中的顶点、纹理坐标和法线数据，生成顶点数组，并使用OpenGL进行渲染。该类还支持为模型指定颜色，并在没有指定颜色时使用默认颜色。

2.2.3 MyBezier.hpp

```

1  #ifndef CAMERA_H
2  #define CAMERA_H
3
4  #include <glad/glad.h>
5  #include <glm/glm.hpp>
6  #include <glm/gtc/matrix_transform.hpp>
7
8  #include <vector>
9
10 // Defines several possible options for camera movement. Used as
    abstraction to stay away from window-system specific input methods
11 enum Camera_Movement
12 {
13     FORWARD,
14     BACKWARD,
```

```

15     LEFT,
16     RIGHT
17 };
18
19 // Default camera values
20 const float YAW = -90.0f;
21 const float PITCH = 0.0f;
22 const float SPEED = 2.5f;
23 const float SENSITIVITY = 0.1f;
24 const float ZOOM = 45.0f;
25
26 // An abstract camera class that processes input and calculates
    the corresponding Euler Angles, Vectors and Matrices for use in
    OpenGL
27 class Camera
28 {
29 public:
30     // camera Attributes
31     glm::vec3 Position;
32     glm::vec3 Front;
33     glm::vec3 Up;
34     glm::vec3 Right;
35     glm::vec3 WorldUp;
36     // euler Angles
37     float Yaw;
38     float Pitch;
39     // camera options
40     float MovementSpeed;
41     float MouseSensitivity;
42     float Zoom;
43
44     // constructor with vectors
45     Camera(glm::vec3 position = glm::vec3(0.0f, 0.0f, 0.0f),
glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f), float yaw = YAW, float
pitch = PITCH) : Front(glm::vec3(0.0f, 0.0f, -1.0f)),
MovementSpeed(SPEED), MouseSensitivity(SENSITIVITY), Zoom(ZOOM)

```

```

46     {
47         Position = position;
48         worldUp = up;
49         Yaw = yaw;
50         Pitch = pitch;
51         updateCameraVectors();
52     }
53     // constructor with scalar values
54     Camera(float posX, float posY, float posZ, float upX, float
upY, float upZ, float yaw, float pitch) : Front(glm::vec3(0.0f,
0.0f, -1.0f)), MovementSpeed(SPEED),
MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
55     {
56         Position = glm::vec3(posX, posY, posZ);
57         worldUp = glm::vec3(upX, upY, upZ);
58         Yaw = yaw;
59         Pitch = pitch;
60         updateCameraVectors();
61     }
62
63     // returns the view matrix calculated using Euler Angles and
the LookAt Matrix
64     glm::mat4 GetViewMatrix()
65     {
66         return glm::lookAt(Position, Position + Front, Up);
67     }
68
69     // processes input received from any keyboard-like input
system. Accepts input parameter in the form of camera defined ENUM
(to abstract it from windowing systems)
70     void ProcessKeyboard(Camera_Movement direction, float
deltaTime)
71     {
72         float velocity = MovementSpeed * deltaTime;
73         if (direction == FORWARD)
74             Position += Front * velocity;

```

```

75         if (direction == BACKWARD)
76             Position -= Front * velocity;
77         if (direction == LEFT)
78             Position -= Right * velocity;
79         if (direction == RIGHT)
80             Position += Right * velocity;
81
82         // Position.y = 0.0f;
83     }
84
85     // processes input received from a mouse input system. Expects
the offset value in both the x and y direction.
86     void ProcessMouseMovement(float xoffset, float yoffset,
GLboolean constrainPitch = true)
87     {
88         xoffset *= MouseSensitivity;
89         yoffset *= MouseSensitivity;
90
91         Yaw += xoffset;
92         Pitch += yoffset;
93
94         // make sure that when pitch is out of bounds, screen
doesn't get flipped
95         if (constrainPitch)
96         {
97             if (Pitch > 89.0f)
98                 Pitch = 89.0f;
99             if (Pitch < -89.0f)
100                 Pitch = -89.0f;
101         }
102
103         // update Front, Right and Up Vectors using the updated
Euler angles
104         updateCameraVectors();
105     }
106

```



```

107     // processes input received from a mouse scroll-wheel event.
    only requires input on the vertical wheel-axis
108     void ProcessMouseScroll(float yoffset)
109     {
110         Zoom -= (float)yoffset;
111         if (Zoom < 1.0f)
112             Zoom = 1.0f;
113         if (Zoom > 45.0f)
114             Zoom = 45.0f;
115     }
116
117 private:
118     // calculates the front vector from the Camera's (updated)
    Euler Angles
119     void updateCameraVectors()
120     {
121         // calculate the new Front vector
122         glm::vec3 front = glm::vec3(1.0f);
123         front.x = cos(glm::radians(Yaw)) *
    cos(glm::radians(Pitch));
124         front.y = sin(glm::radians(Pitch));
125         front.z = sin(glm::radians(Yaw)) *
    cos(glm::radians(Pitch));
126         Front = glm::normalize(front);
127         // also re-calculate the Right and Up vector
128         Right = glm::normalize(glm::cross(Front, worldUp)); //
    normalize the vectors, because their length gets closer to 0 the
    more you look up or down which results in slower movement.
129         Up = glm::normalize(glm::cross(Right, Front));
130     }
131 };
132 #endif

```

`MyBezier.hpp` 文件定义了一个名为 `MyBezier` 的类，用于生成和渲染贝塞尔曲面。

- 包含的库：

- `glad/glad.h`: OpenGL函数加载库。
- `GLFW/glfw3.h`: GLFW库, 用于创建窗口和处理输入。
- `iostream`: 标准输入输出流库。
- `vector`: 标准库中的向量容器。
- `glm/glm.hpp`、`glm/gtc/matrix_transform.hpp` 和 `glm/gtc/type_ptr.hpp`: GLM数学库, 用于处理向量、矩阵等数学运算。
- `tool/shader.h`: 自定义的着色器工具库。

- 顶点属性:

- `Position`: 顶点位置, 类型为 `glm::vec3`。
- `Color`: 顶点颜色, 类型为 `glm::vec4`。

```

1 struct Bezier_Vertex
2 {
3     glm::vec3 Position; // 顶点属性
4     glm::vec4 Color;    // 颜色
5 };

```

- 类 `MyBezier`

```

1 class MyBezier
2 {
3 private:
4     int _numGrid;
5     int _num_ControlPoints = 4;
6     glm::vec4 _color;
7     std::vector<glm::vec3> controlPoints;
8     std::vector<Bezier_Vertex> surfacePoints;
9     std::vector<unsigned int> indices;
10
11     unsigned int VAO, VBO, EBO;
12     float height = 0.2f;

```

- 成员变量:

- `_numGrid`: 网格数量, 用于生成曲面点。

- `_num_ControlPoints` : 控制点数量, 默认为4。
- `_color` : 曲面颜色, 类型为 `glm::vec4`。
- `controlPoints` : 控制点, 类型为 `std::vector<glm::vec3>`。
- `surfacePoints` : 曲面点, 类型为 `std::vector<Bezier_Vertex>`。
- `indices` : 索引数组, 类型为 `std::vector<unsigned int>`。
- `VAO`、`VBO` 和 `EBO` : 顶点数组对象、顶点缓冲对象和元素缓冲对象的ID。
- `height` : 曲面高度, 类型为 `float`。

- 构造函数:

- 接受网格数量和颜色参数。
- 初始化控制点数组 `controlPoints`。

- 析构函数:

- 删除VAO和VBO, 释放OpenGL资源。

```

1 ~MyBezier()
2 {
3     glDeleteVertexArrays(1, &VAO);
4     glDeleteBuffers(1, &VBO);
5 }
```

- 初始化函数:

- 调用 `GenerateSurface` 函数生成曲面点。
- 生成并绑定VAO、VBO和EBO。
- 将曲面点数据传递给VBO。
- 启用顶点属性并设置顶点属性指针。
- 将索引数据传递给EBO。

- 渲染函数:

- 绑定VAO。
- 使用 `glDrawElements` 函数绘制三角形网格。
- 解绑VAO。

```
1 void Draw(Shader &shader)
2 {
3     glBindVertexArray(VAO);
4     glDrawElements(GL_TRIANGLES, indices.size(),
5         GL_UNSIGNED_INT, 0);
6     glBindVertexArray(0);
7 }
```

- 生成曲面点函数：

- 清空 `surfacePoints` 向量。
- 遍历网格，计算每个网格点的贝塞尔曲面位置。
- 将计算得到的曲面点存储在 `surfacePoints` 向量中。
- 生成索引数组，用于绘制三角形网格。

```
1 void GenerateSurface()
2 {
3     surfacePoints.clear();
4     for (int i = 0; i <= _numGrid; ++i)
5     {
6         for (int j = 0; j <= _numGrid; ++j)
7         {
8             float u = static_cast<float>(i) / _numGrid;
9             float v = static_cast<float>(j) / _numGrid;
10            glm::vec3 point = BezierPoint(u, v);
11            Bezier_Vertex vertex;
12            vertex.Position = point;
13            vertex.Color = _color;
14            surfacePoints.push_back(vertex);
15        }
16    }
17    for (int i = 0; i < _numGrid; ++i)
18    {
```

```

19     for (int j = 0; j < _numGrid; ++j)
20     {
21         int seq = i * (_numGrid + 1) + j;
22         // No.1 triangle
23         indices.push_back(seq);
24         indices.push_back(seq + 1);
25         indices.push_back(seq + _numGrid + 1);
26         // No.2 triangle
27         indices.push_back(seq + 1);
28         indices.push_back(seq + _numGrid + 2);
29         indices.push_back(seq + _numGrid + 1);
30     }
31 }
32 }

```

- 计算贝塞尔曲面点函数：

- 使用Bernstein函数计算曲面点。
- 遍历所有控制点，计算每个控制点对曲面点的贡献。
- 返回计算得到的曲面点。

```

1  glm::vec3 BezierPoint(float u, float v)
2  {
3      glm::vec3 point(0.0f);
4      for (int i = 0; i < _num_ControlPoints; ++i)
5      {
6          for (int j = 0; j < _num_ControlPoints; ++j)
7          {
8              float basisU = BernsteinBasis(i, u);
9              float basisV = BernsteinBasis(j, v);
10             point += controlPoints[i * _num_ControlPoints + j] *
basisU * basisV;
11         }
12     }
13     return point;
14 }

```

- **Bernstein函数:**

- 计算Bernstein函数的值。
- 根据 `i` 的值返回相应的Bernstein函数值。

```
1 float BernsteinBasis(int i, float t)
2 {
3     if (i < 0 || i > 3)
4         return 0.0f;
5     if (i == 0)
6         return pow(1 - t, 3);
7     if (i == 1)
8         return 3 * t * pow(1 - t, 2);
9     if (i == 2)
10        return 3 * pow(t, 2) * (1 - t);
11    if (i == 3)
12        return pow(t, 3);
13 }
```

`MyBezier.hpp` 文件实现了一个简单的贝塞尔曲面生成和渲染类。该类通过控制点生成曲面点，并使用OpenGL进行渲染。通过设置控制点和网格数量，可以生成不同形状的贝塞尔曲面。该类还支持为曲面指定颜色，并提供了初始化和渲染函数。

2.2.4 Main.cpp

```
1 #include <glad/glad.h>
2 #include <GLFW/glfw3.h>
3 #include <iostream>
4 #include <cmath>
5
6 #include <tool/shader.h>
7 #include <tool/camera.h>
8 #include <tool/MyObj.hpp>
9 #include <tool/MyBezier.hpp>
10
11 void framebuffer_size_callback(GLFWwindow *window, int width, int
    height);
12 void mouse_callback(GLFWwindow *window, double xpos, double ypos);
```

```

13 void processInput(GLFWwindow *window);
14
15 std::string Shader::dirName;
16 int SCREEN_WIDTH = 1600;
17 int SCREEN_HEIGHT = 1200;
18
19 // delta time
20 float deltaTime = 0.0f;
21 float lastTime = 0.0f;
22
23 float lastX = SCREEN_WIDTH / 2.0f; // 鼠标上一帧的位置
24 float lastY = SCREEN_HEIGHT / 2.0f;
25
26 Camera camera(glm::vec3(5.0, 5.0, 5.0));
27
28 using namespace std;
29
30 int main(int argc, char *argv[])
31 {
32     Shader::dirName = argv[1];
33     glfwInit();
34     // 设置主要和次要版本
35     const char *glsl_version = "#version 330";
36
37     // 片段着色器将作用域每一个采样点（采用4倍抗锯齿，则每个像素有4个片
    段（四个采样点））
38     // glfwWindowHint(GLFW_SAMPLES, 4);
39     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
40     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
41     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
42
43     // 窗口
44     GLFWwindow *window = glfwCreateWindow(SCREEN_WIDTH,
    SCREEN_HEIGHT, "Dream_Car", NULL, NULL);
45     if (window == NULL)
46     {

```

```
47         std::cout << "Failed to create GLFW window" << std::endl;
48         glfwTerminate();
49         return -1;
50     }
51     glfwMakeContextCurrent(window);
52
53     if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
54     {
55         std::cout << "Failed to initialize GLAD" << std::endl;
56         return -1;
57     }
58
59     // 设置视口
60     glViewport(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT);
61     glEnable(GL_PROGRAM_POINT_SIZE);
62     glEnable(GL_BLEND);
63     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
64
65     glEnable(GL_DEPTH_TEST);
66
67     // 鼠标键盘
68     // 1.注册窗口变化监听
69     glfwSetFramebufferSizeCallback(window,
framebuffer_size_callback);
70     // 2.鼠标事件
71     glfwSetCursorPosCallback(window, mouse_callback);
72
73     shader shader("./shader/vertex.glsl",
"./shader/fragment.glsl");
74     MyObj car("./src/dream_car.obj");
75     MyBezier bezier(28);
76     MyBezier bezier2(28);
77     bezier.Init();
78     bezier2.Init();
79
80     while (!glfwWindowShouldClose(window))
```



```

81     {
82         processInput(window);
83
84         float currentFrame = glfwGetTime();
85         deltaTime = currentFrame - lastTime;
86         lastTime = currentFrame;
87
88         // 渲染指令
89         glClearColor(1, 1, 1, 1); // 纯白色
90         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
91
92         glm::mat4 view = camera.GetViewMatrix();
93         glm::mat4 projection = glm::mat4(1.0f);
94         float fov = 45.0f; // 视锥体的角度
95         projection = glm::perspective(glm::radians(fov),
96 (float)SCREEN_WIDTH / (float)SCREEN_HEIGHT, 0.1f, 100.0f);
97         shader.use();
98         shader.setMat4("view", view);
99         shader.setMat4("projection", projection);
100
101         glm::mat4 model = glm::mat4(1.0f);
102         model = glm::translate(model, glm::vec3(-5.0f, 0.0f,
103 -5.0f));
104         model = glm::scale(model, glm::vec3(0.1f, 0.1f, 0.1f));
105         shader.setMat4("model", model);
106         car.Draw(shader);
107
108         model = glm::mat4(1.0f);
109         model = glm::translate(model, glm::vec3(5.8f, 2.3f,
110 -2.7f));
111         model = glm::rotate(model, glm::radians(-90.0f),
112 glm::vec3(1.0f, 0.0f, 0.0f));
113         shader.setMat4("model", model);
114         bezier.Draw(shader);
115
116         model = glm::mat4(1.0f);

```

```
113         model = glm::translate(model, glm::vec3(-3.0f, 2.3f,
114 -2.7f));
115         model = glm::rotate(model, glm::radians(-90.0f),
116 glm::vec3(1.0f, 0.0f, 0.0f));
117         shader.setMat4("model", model);
118         bezier2.Draw(shader);
119
120         glfwSwapBuffers(window);
121         glfwPollEvents();
122     }
123     // 释放资源
124     glfwTerminate();
125     return 0;
126 }
127 // 窗口变动监听
128 void framebuffer_size_callback(GLFWwindow *window, int width, int
129 height)
130 {
131     glViewport(0, 0, width, height);
132 }
133 // 键盘输入监听
134 void processInput(GLFWwindow *window)
135 {
136     if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
137     {
138         glfwSetWindowShouldClose(window, true);
139     }
140
141     // 相机按键控制
142     // 相机移动
143     if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
144     {
145         camera.ProcessKeyboard(FORWARD, deltaTime);
146     }
```

```

146     if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
147     {
148         camera.ProcessKeyboard(BACKWARD, deltaTime);
149     }
150     if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
151     {
152         camera.ProcessKeyboard(LEFT, deltaTime);
153     }
154     if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
155     {
156         camera.ProcessKeyboard(RIGHT, deltaTime);
157     }
158 }
159
160 // 鼠标移动监听
161 void mouse_callback(GLFWwindow *window, double xpos, double ypos)
162 {
163
164     float xoffset = xpos - lastX;
165     float yoffset = lastY - ypos;
166
167     lastX = xpos;
168     lastY = ypos;
169
170     camera.ProcessMouseMovement(xoffset, yoffset);
171 }

```

这段代码是一个C++项目的主要头文件和全局变量定义部分，主要用于设置OpenGL和GLFW的环境，并包含了一些必要的库和工具。

首先包含了GLAD和GLFW库的头文件。GLAD是一个用于加载OpenGL函数指针的库，而GLFW是一个用于创建窗口和处理输入的库。

`<tool/shader.h>` 和 `<tool/camera.h>` 分别包含了着色器和相机类的头文件，这些类用于管理着色器程序和相机的视角。`<tool/MyObj.h>` 包含了模型加载和处理的头文件，用于加载和渲染3D模型，`<tool/MyBezier.h>` 包含了贝塞尔曲面处理的头文件，用于加载和渲染贝塞尔曲面。

然后，定义了三个回调函数的声明，这些函数用于处理窗口大小变化、鼠标移动和键盘输入。

`std::string Shader::dirName` 定义了一个静态字符串变量，用于存储着色器文件的目录路径。

再接着定义了窗口的宽度和高度，以及两个全局变量 `deltaTime` 和 `lastTime`，用于计算每帧之间的时间差，以实现平滑的动画效果。`lastx + lasty` 定义了鼠标上一帧的位置，用于处理鼠标移动。`Camera camera(glm::vec3(5.0, 5.0, 5.0));` 创建了一个相机对象，并设置了初始位置。

3 实验结果与分析

运行项目：

在根目录直接执行 `make run` 即可，单独运行 `exe` 是不可以的，由于其依赖的 `obj` 和着色器程序

运行程序后，我们得到了一个静态的模拟窗口。窗口中展示了未贴纹理的汽车，用户可以通过键盘控制摄像机的移动，通过鼠标移动实现视角的旋转。这里使用贝塞尔曲面实现了汽车的引擎盖，我还单独将其平移出来在一侧观看其效果，其实现了较为光滑的过渡：





4 思考题

Compare the strengths and weaknesses of the techniques you use

4.1 Obj模型的优缺点

4.1.1 优点

1. 通用性：

- OBJ格式是一种广泛使用的3D模型文件格式，支持多种3D建模软件（如Blender、Maya、3ds Max等）导出和导入。
- 适用于各种平台和应用场景，易于与其他工具和引擎集成。

2. 简单易用：

- OBJ文件格式简单，易于解析和处理。
- 文件结构清晰，包含顶点、法线、纹理坐标和面信息，便于理解 and 操作。

3. 支持多种数据类型：

- 支持顶点位置、法线、纹理坐标、面索引等多种数据类型。

- 可以存储复杂的几何结构和材质信息。

4. 开源支持：

- 有许多开源库和工具支持OBJ文件的读取和写入，如Assimp、TinyOBJLoader等。

4.1.2 缺点

1. 文件大小：

- 对于复杂模型，OBJ文件可能会变得非常大，导致加载和处理时间较长。
- 文件中可能包含大量重复数据，导致存储和传输效率较低。

2. 不支持压缩：

- OBJ文件通常以纯文本格式存储，不支持压缩，导致文件大小较大。
- 相比之下，其他格式（如FBX、GLTF）支持二进制存储和压缩，文件大小更小。

3. 缺乏版本控制：

- OBJ格式没有版本控制机制，不同版本的文件可能会有不同的解析方式。
- 不同软件导出的OBJ文件可能会有细微差异，导致兼容性问题。

4.2 贝塞尔曲面的优缺点

4.2.1 优点

1. 平滑性：

- 贝塞尔曲面可以生成非常平滑的几何形状，适用于需要高质量曲面的应用场景（如汽车设计、工业设计等）。
- 通过控制点可以精确控制曲面的形状和细节。

2. 灵活性：

- 贝塞尔曲面可以通过调整控制点来改变曲面的形状，具有很高的灵活性。
- 适用于需要动态调整和交互的应用场景（如CAD软件、动画制作等）。

3. 数学基础：

- 贝塞尔曲面基于数学上的伯恩斯坦多项式，具有坚实的数学基础。
- 可以通过数学方法进行分析和优化，适用于需要精确计算的应用场景。

4. 易于实现：

- 贝塞尔曲面的生成和渲染算法相对简单，易于实现。
- 可以通过简单的循环和矩阵运算生成曲面点，适用于实时渲染和交互应用。

4.2.2 缺点

1. 计算复杂度：

- 生成贝塞尔曲面需要计算大量的曲面点，计算复杂度较高。
- 对于高阶贝塞尔曲面，计算量会显著增加，可能导致性能问题。

2. 控制点数量：

- 贝塞尔曲面的形状和细节受控制点数量影响较大。
- 控制点数量较多时，曲面可能会变得过于复杂，难以控制。

3. 局部控制：

- 贝塞尔曲面的局部控制能力较弱，调整一个控制点可能会影响整个曲面的形状。
- 相比之下，**NURBS**（非均匀有理**B**样条）曲面具有更强的局部控制能力。