111550006 林庭鳶

Homework 2 – 2024 Computer Graphics

In this computer graphics assignment, I implemented various features to construct and render 3D models with proper lighting and shading techniques. My tasks included loading and parsing `.obj` files for predefined objects, manually generating vertices for complex shapes like a vase using Bezier curves, and setting up Phong lighting calculations for different types of lights (directional, point, and spotlight). I carefully designed each model's geometry, texture mapping, and transformations (scale, rotation, and translation) to ensure they fit seamlessly into the scene. Through these efforts, I achieved accurate and visually appealing 3D renderings while addressing challenges like z-fighting and ensuring proper face orientation for correct lighting effects.

I run the following instruction for running the project:

```
cmake --build build --config Debug --verbose
cd bin
./HW2
```

IMPLEMENTATION DETAILS

1. **assets/models/cube/cube.obj**

   To complete this `.obj` file of a cube, the key task is to ensure that the vertex normals (`vn`) and texture coordinates (`vt`) align correctly with the cube's geometry and the faces defined in the `f` section. According to 8 geven vertices, 6 `vn` represent 6 directions in the space of a cube.

   vn -1.0000 -0.0000 -0.0000: Normal for the left face (pointing in the negative x-direction).
   vn -0.0000 -0.0000 -1.0000: Normal for the back face (pointing in the negative z-direction).
   vn 1.0000 -0.0000 -0.0000: Normal for the right face (pointing in the positive x-direction).

vn -0.0000 -0.0000 1.0000: Normal for the front face (pointing in the positive z-direction).
vn -0.0000 -1.0000 -0.0000: Normal for the bottom face (pointing in the negative y-direction).
vn -0.0000 1.0000 -0.0000: Normal for the top face (pointing in the positive y-direction).

The `vt` lines define **texture coordinates** that map the 2D texture onto the 3D surface of the cube. A cube has six faces, and each face requires a set of texture coordinates to map the texture onto it. Additionally, Some texture coordinates are reused because adjacent faces share edges, and hence, vertices along those edges can have the same texture coordinates.

The `f` lines define **faces** of the cube using vertex indices: f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3, each of its value imply the index of corresponding list. The `f` entries describe how the 3D geometry is constructed using the vertices, normals, and texture coordinates. A cube is composed of 12 triangles (2 per face), and each `f` line represents one triangle.

2. **assets/shaders/light.vert**

The main function in this vertex shader handles transforming vertex data into clip space while also preparing additional information for the fragment shader. Code implementation and explanation is as followed:

```
void main() {
    // Transform position to world space
    FragPos = vec3(ModelMatrix * vec4(position, 1.0));

    // Transform normal to world space using the precomputed
normal matrix
    // TIModelMatrix: The transpose of the inverse of the model
matrix, precomputed to correctly transform normals to world
space.
    Normal = mat3(TIModelMatrix) * normal;

    // Pass texture coordinates to fragment shader
```

```
    TexCoord = texCoord;

    // Transform position to clip space
    gl_Position = Projection * ViewMatrix * vec4(FragPos, 1.0);
}
```

3. assets/shaders/light.frag

This block implements **Phong lighting** for a directional light and point light source by calculating its ambient, diffuse, and specular contributions based on the light's direction, the surface's normal, and the viewer's position. Each term is scaled by the material's properties and the light's color to create realistic shading. Implementation explanation is shown as follows:

```
// 1. Directional Light
    if (dl.enable == 1) {
        // Normalize the direction of the light. The negative sign
is used to invert the direction,
        // as the light direction points towards the surface in
world space.
        vec3 lightDir = normalize(-dl.direction);

        // Calculate the diffuse component using the dot product of
the normalized surface normal
        // and the light direction. This simulates light scattering
and ensures non-negative values
        // (using max) for surfaces facing away from the light.
        float diff = max(dot(normal, lightDir), 0.0);

        // Compute the reflection direction of the light about the
surface normal,
        // used in specular lighting calculations.
        vec3 reflectDir = reflect(-lightDir, normal);

        // Calculate the specular component using the dot product
of the view direction and
        // the reflection direction, raised to the power of the
material's shininess for controlling
        // highlight sharpness. Use max to ensure non-negative
```

```
values.
        float spec = pow(max(dot(viewDir, reflectDir), 0.0),
material.shininess);

        // Compute the ambient term, representing global
illumination,
        // by multiplying the material's ambient reflectivity with
the light's color.
        vec3 ambient = material.ambient * dl.lightColor;

        // Compute the diffuse term, representing the light
scattered across the surface,
        // scaled by the material's diffuse reflectivity and the
light's color.
        vec3 diffuse = material.diffuse * diff * dl.lightColor;

        // Compute the specular term, representing the shiny
highlights on the surface,
        // scaled by the material's specular reflectivity and the
light's color.
        vec3 specular = material.specular * spec * dl.lightColor;

        // resultColor += attenuation * (ambient + diffuse +
specular) for point light and spotlight
        resultColor += ambient + diffuse + specular;
    }
```

The implementation overlaps within three different lights. As for **attenuation**, which represents how the intensity of a point light diminishes with distance, I use the following code to tackle the calculation:

```
float distance = length(pl.position - FragPos);
float attenuation = 1.0 / (pl.constant + pl.linear * distance +
pl.quadratic * distance * distance);
```

Spotlight requires to additionally consider the `theta` value, which is determined by taking the dot product between `lightDir` and the normalized negative direction of

the spotlight (`-sl.direction`). This value represents the cosine of the angle between the fragment and the spotlight's central beam. The condition `theta > sl.cutOff` ensures lighting calculations are performed only for fragments within the spotlight's cone, defined by `sl.cutOff.`

4. **src/model.cpp**

   The task was to parse an `.obj` file and extract the geometric data (`v`, `vt`, `vn`, `f`) into the `Model` structure's `positions`, `texcoords`, `normals`, and `numVertex`. Other fields in the `.obj` file could be ignored. The implementation of the `Model::fromObjectFile` function reads the file line-by-line to process vertex positions (`v`), texture coordinates (`vt`), normals (`vn`), and face definitions (`f`). Temporary vectors store raw data for positions, texture coordinates, and normals, which are later indexed and appended to the `Model` object's `positions`, `texcoords`, and `normals` arrays based on the face definitions. Face indices are converted from 1-based to 0-based indexing to align with C++ array standards. The function ensures the model is complete by setting the `numVertex` attribute based on the total number of vertices and closes the file after processing, readying the model for rendering. This implementation focuses solely on parsing and storing essential geometry data, ignoring unsupported fields for simplicity.

5. **src/main.cpp**

   First, we need to finish all create objects functions. Simply do the following steps:

   1. use `fromObjectFile` to convert object file.
   2. Set num vertex.

3. Push back texture.
4. Scale and rotate.
5. Set up draw mode.

On the other hand, since the object file of vase is not provided, we should generate all the vertices with provided bezier curve. The following code is the implementation:

```
// generate vertices
 for (int h = 0; h <= height_segments; ++h) {
   float t = static_cast<float>(h) / height_segments;
   float y = height * t;
   float radius = bezier(t, p0, p1, p2, p3);

   for (int s = 0; s <= segments; ++s) {
     float theta = static_cast<float>(s) / segments * 2.0f * M_PI;
     float x = radius * cos(theta);
     float z = radius * sin(theta);

     positions.push_back(x);
     positions.push_back(y);
     positions.push_back(z);

     glm::vec3 normal = glm::normalize(glm::vec3(x, 0.0f, z));
     normals.push_back(normal.x);
     normals.push_back(normal.y);
     normals.push_back(normal.z);

     texcoords.push_back(static_cast<float>(s) / segments);
     texcoords.push_back(static_cast<float>(h) / height_segments);
   }
 }

 // construct each triangle in counter-clockwise order
 for (int h = 0; h < height_segments; ++h) {
   for (int s = 0; s < segments; ++s) {
```

```
    int curr = h * (segments + 1) + s;
    int next = curr + segments + 1;

    indices.push_back(curr);
    indices.push_back(next);
    indices.push_back(curr + 1);

    indices.push_back(next);
    indices.push_back(next + 1);
    indices.push_back(curr + 1);
  }
}

// push position, normal, textcoord into model
for (int i : indices) {
  vase->positions.push_back(positions[i * 3 + 0]);
  vase->positions.push_back(positions[i * 3 + 1]);
  vase->positions.push_back(positions[i * 3 + 2]);

  vase->normals.push_back(normals[i * 3 + 0]);
  vase->normals.push_back(normals[i * 3 + 1]);
  vase->normals.push_back(normals[i * 3 + 2]);

  vase->texcoords.push_back(texcoords[i * 2 + 0]);
  vase->texcoords.push_back(texcoords[i * 2 + 1]);
}

GLuint texture = createTexture("../assets/models/Vase/Vase.jpg");
if (!texture) {
  std::cerr << "Failed to load vase texture!" << std::endl;
  delete vase;
  return nullptr;
}
vase->textures.push_back(texture);

vase->numVertex = vase->positions.size() / 3;
vase->drawMode = GL_TRIANGLES;
```

```
 return vase;
}
```

Second, after creating models, use `ctx.models.push_back` to load all models, and `ctx.objects.push_back` for setting up the initial position of all models.

ENCOUNTERED PROBLEMS

1. The bottom of vase was transparent, because I set the normal vector in the wrong direction(if the triangle is constructed counter-clockwise, the side should be the front side) and I also set y = 0.005f to avoid **z-fighting.**