HW4 report 110550110 林書愷

Main.cpp:

```
float dissolve = 0.0;
```

This global variable is for calculating what the what part the object should dissolve to. Since the way I implement dissolve effect is comparing the vertex is below a x coordinate threshold. As the time pass, this threshold will increase and this variable will be passed into fragment shader of dissolve to decide the amount x threshold need to increase.

```
unsigned int phong_vs, phong_fs;
phong_vs = createShader("shaders/Blinn-Phong.vert", "vert");
phong_fs = createShader("shaders/Blinn-Phong.frag", "frag");
unsigned int gouraud_vs, gouraud_fs;
gouraud_vs = createShader("shaders/Gouraud.vert", "vert");
gouraud_fs = createShader("shaders/Gouraud.frag", "frag");
unsigned int toon_vs, toon_fs;
toon_vs = createShader("shaders/Toon.vert", "vert");
toon_fs = createShader("shaders/Toon.frag", "frag");
unsigned int flat_vs, flat_gm, flat_fs;
flat_vs = createShader("shaders/Flat.vert", "vert");
flat_gm = createShader("shaders/Flat.geom", "geom");
flat_fs = createShader("shaders/Flat.frag", "frag");
unsigned int border_vs, border_fs;
border_vs = createShader("shaders/Border.vert", "vert");
border_fs = createShader("shaders/Border.frag", "frag");
unsigned int dissolve_vs, dissolve_fs;
dissolve_vs = createShader("shaders/Dissolve.vert", "vert");
dissolve_fs = createShader("shaders/Dissolve.frag", "frag");
```

This part is to create shaders: vertex, fragment, geometry for all shadings and record their shader id with variables. The task is done by calling createShader function with the shader files passed in and target shader strings to decide what shader is creating currently.

```
unsigned int phong_program;
phong_program = createProgram(phong_vs, 0,phong_fs);
unsigned int gouraud_program;
gouraud_program = createProgram(gouraud_vs, 0, gouraud_fs);
unsigned int flat_program;
flat_program = createProgram(flat_vs, flat_gm, flat_fs);
unsigned int toon_program;
toon_program = createProgram(toon_vs, 0, toon_fs);
unsigned int border_program;
border_program = createProgram(border_vs, 0 ,border_fs);
unsigned int dissolve_program;
dissolve_program = createProgram(dissolve_vs, 0, dissolve_fs);
```

After the shaders are created this part is to create the shading programs by the created shaders. Using the createProgram function to create shading programs, it will pass in three shaders: vertex, geometry and fragment. Since only flat shading includes geometry shader in this assignment, other programs will have 0(NULL) passed in as they don't possess geometry shader. Finally, the created programs will be memorized with an unsigned int to record their program id.

```
unsigned int current;
unsigned int view_Loc;
if (status == 1) {
    current = phong_program;
}
else if (status == 2) {
    current = gouraud_program;
}
else if (status == 3) {
    current = flat_program;
}
else if (status == 4) {
    current = toon_program;
}
else if (status == 5) {
    current = border_program;
}
else {
    current = dissolve_program;
    if (rotating) {
        dissolve += 0.1;
    }
    int tlocation = glGetUniformLocation(current, "dissolve");
    glUniform1f(tlocation, dissolve);
}
view_Loc = glGetUniformLocation(current, "camera");
```

This part is to decide which shading program is current program. The unsigned int current is to memorize the current shading program's id and view_loc variable is to memorize the uniform varible camera in vertex shader for later pass the camera position to vertex shader. Then the if-else is to judge what shading is currently doing by the status variable. Mostly it will just pass the shading program id to current variable but for the sixth case, dissolve shading it will have to do special task. Since the amount of increasing of x threshold of dissolving is based on the dissolve variable, it needs to increase 0.1 in every iteration of the while loop. However if the object isn't rotating it won't increase so it will be protected by an if. Then the tlocation variable is for memorizing the position of dissolve variable in dissolve fragment shader. Then use glUniformlf to pass dissolve variable to fragement shader. Finally use glGetUniformLocation to get the camera variable position and saves it to view_Loc variable.

```
glUniform3fv(view_Loc, 1, &cameraPos[0]);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, deerTexture);
glBindVertexArray(deerVAO);
glDrawArrays(GL_TRIANGLES, 0, deerModel->positions.size());
glBindTexture(GL_TEXTURE_2D, 0);
```

This part is mainly setting up shaders for drawing out the object and the pass variables to shaders.

- glUniform3fv: this one is for passing the camera position vector3 to vertex shader and position view_Loc.
- glEnable(GL_BLEND) and glBlendFunc: these two function are for initiate the transparent mode of the object(will be used in dissolveshading).glEnable(GL_BLEND);: This line enables blending in OpenGL. When blending is enabled, fragments that are drawn to the framebuffer are combined with theexisting framebuffer contents based on a specified blending equation.glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);: This line sets the blending function. The glBlendFunc function takes two arguments that specify how the source and destination factors are combined. In this case, it specifies the commonly used alpha blending equation:GL_SRC_ALPHA: Specifies that the source fragment's alpha value is used as the blending factor.GL_ONE_MINUS_SRC_ALPHA: Specifies that the destination fragment's

alpha value (1.0 - source alpha) is used as the blending factor. When both of them are set, OpenGL will combine the colors of overlapping fragments based on the specified blending equation, which is particularly useful for rendering semi-transparent objects or achieving effects like smooth transitions between objects.

- glActiveTexture: this function will active GL_Texture0 as texture unit and later all operations related to texture will based on this unit.
- glBindTexture: this function will bind the object's texture(deerTexture) with will texture unit(Texture0) to formalize for later operating in shaders.
- glBindVertexArray: this function will bind the object's vao model with pre-generated vertex array for the shader.
- glDrawArrays: after the object's vertex array is set, this function will based on the set pixel shape(triangle) and draw out the numbers pixels of model's size to draw out the model.
- After drawing is done, call bind texture to detach texture to avoid overriding the object's texture.

```
glUseProgram(current);
unsigned int mLoc, vLoc, pLoc;
mLoc = glGetUniformLocation(current, "M");
vLoc = glGetUniformLocation(current, "V");
pLoc = glGetUniformLocation(current, "P");
glUniformMatrix4fv(mLoc, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(vLoc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(pLoc, 1, GL_FALSE, glm::value_ptr(perspective));

glBindVertexArray(0);
```

This part is for setting the matrix of the shaders by following:

- glUseProgram: set the current shading program for the shaders.
- Unsigned int mLoc, vLoc, pLoc are used for memorizing the position of model, view and perspective matrix position in the shaders and will be fetched by glGetUniformLocation function.
- As the variables' location are set, use glUniformMatrix4fv(they are all mat4 type) to pass in the matrixes to the shaders.
- Finally call glBindVertexArray again and pass in 0(NULL) to detach the model vao with the shaders to avoid overriding the value.

```
if (key == GLFW_KEY_1 && action == GLFW_PRESS) {
    status = 1;
    cout << "KEY 1 PRESSED\n";
}
if (key == GLFW_KEY_2 && action == GLFW_PRESS) {
    status = 2;
    cout << "KEY 2 PRESSED\n";
}
if (key == GLFW_KEY_3 && action == GLFW_PRESS) {
    status = 3;
    cout << "KEY 3 PRESSED\n";
}
if (key == GLFW_KEY_4 && action == GLFW_PRESS) {
    status = 4;
    cout << "KEY 4 PRESSED\n";
}
```

```
if (key == GLFW_KEY_5 && action == GLFW_PRESS) {
    status = 5;
    cout << "KEY 5 PRESSED\n";
}
if (key == GLFW_KEY_6 && action == GLFW_PRESS) {
    status = 6;
    dissolve = 0.0;
    cout << "KEY 6 PRESSED\n";
}
```

This part is in keyCallback function and is used for changing the status to the target shading method based on the key we type. The special part of dissolve shading is that besides changing the status value it will initial the dissolve variable so that the next time doing dissolve shading will be ensured starting from beginning.

Shaders:

Blinn-Phong:

Vertex:

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

uniform mat4 M;
uniform mat4 V;
uniform mat4 P;
uniform vec3 camera;

out vec2 texCoord;
out vec3 normal;
out vec3 view_pos;
out vec4 worldPos;
```

These are the variable of vertex shader, for the layout aPos represent the position of the object is at , aNormal represent the vertexes' normal of the object and the aTexCoord is the object's texture coordinates. Layouts are the input variables into vertex shader besides uniform variables. The four uniform variables: M represents the model's matrix(outcome of rotate, scaling, translate), V represents the view matrix, P represents the perspective matrix and camera represents the position vector of the camera position. The output variables are values for later passing to fragment shader. texCoord is aTexCoord from layout, normal are the calculation outcome of aNormal, vie_pos is camera from uniform variable and worldPos is the world location of object, which is the product of aPos and M matrix.

```
void main()
{

    texCoord = aTexCoord;
    worldPos = M * vec4(aPos, 1.0);
    gl_Position = P * V * M * vec4(aPos, 1.0);
    normal = normalize((transpose(inverse(M))* vec4(aNormal, 0.0)).xyz);
    view_pos = camera;


}
```

The main part mostly do some calculation of vertexes. For the texCoord, it copies aTexCoord's value to pass to fragment shader. worldPos is to calculate the object's world position by timing M and aPos and converting it to vec4 for easier calculation(all matrix are 4*4). Gl_Position is the product of P, V and worldPos matrixes and is used to specify the position of a vertex in clip space, which is a

coordinate space that goes from -1 to 1 in all three dimensions. Normal transforms the input normal vector from object space to world space and ensures that it is normalized. The method is aNormal is the input normal vector in object space.

vec4(aNormal, 0.0) converts the 3D normal vector to a 4D vector by adding a 0.0 as the fourth component.

transpose(inverse(M)) calculates the transpose of the inverse of the model matrix (M). This transformation is used to transform normals because normals don't undergo translation and should be transformed with the inverse transpose of the model matrix to handle non-uniform scaling correctly.

transpose transposes the matrix.

inverse calculates the inverse of the matrix.

transpose(inverse(M)) * vec4(aNormal, 0.0) transforms the 4D normal vector to world space.

.xyz extracts the first three components of the resulting 4D vector, effectively removing the translation part.

normalize normalizes the resulting 3D vector to ensure it has unit length.

Finally , copies camera value to view_pos to pass to fragment shader.

Fragment:

```
in vec2 texCoord;
in vec4 worldPos;
in vec3 normal;
in vec3 view_pos;

uniform sampler2D deer_texture;

out vec4 FragColor;
```

Passed in variables are from vertex shader and the output variable FragColor is a four dimensional vector represent the rgb value of object. Deer_texture is the texture of the object and sampler2D: Specifies the type of the uniform variable. In this case, it's a sampler for 2D textures. It's used to sample colors from a 2D texture image.

```
vec3 Ka = vec3(1.0, 1.0, 1.0);
vec3 Kd = vec3(1.0, 1.0, 1.0);
vec3 Ks = vec3(0.7, 0.7, 0.7);
float shininess = 10.5;
vec3 La = vec3(0.2, 0.2, 0.2);
vec3 Ld = vec3(0.8, 0.8, 0.8);
vec3 Ls = vec3(0.5, 0.5, 0.5);
vec3 lightPos = vec3(10, 10, 10);

vec3 obj_color = vec3(texture(deer_texture, texCoord));
```

This part sets coefficient of shading based on the spec, and obj_color is generated with model texture and mapped by the texCoord to correctly find the right color of vertex.

```
// Ambient
vec3 ambient = La * Ka * obj_color;

// Diffuse
vec3 n_normal = normalize(normal);
vec3 light = normalize(vec4(lightPos, 1.0) - worldPos).xyz;
vec3 diffuse = Ld * Kd * obj_color * max(0.0, dot(light, n_normal));

// Specular (Blinn-Phong)
vec3 view_dir = normalize(view_pos - vec3(worldPos));
vec3 halfway_dir = normalize(light + view_dir);
vec3 specular = Ls * Ks * pow(max(0.0, dot(n_normal, halfway_dir)), shininess);

FragColor = vec4((ambient + diffuse + specular), 1.0);
```

This part is the calculation of ambient, diffuse, specular and is just based on the formulas from the spec and in the end will sum up three outcome to generate FragColor.

Gouraud:

Vertex:

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

uniform mat4 M;
uniform mat4 V;
uniform mat4 P;
uniform vec3 camera;

out vec2 texCoord;
out vec4 worldPos;
out vec3 normal;
out vec3 ambient;
out vec3 diffuse;
out vec3 specular;
```

```
gl_Position = P * V * M * vec4(aPos, 1.0);
texCoord = aTexCoord;
worldPos = M * vec4(aPos, 1.0);
normal = normalize((transpose(inverse(M)) * vec4(aNormal, 0.0)).xyz);
vec3 view_pos = camera;

vec3 Ka = vec3(1.0, 1.0, 1.0);
vec3 Kd = vec3(1.0, 1.0, 1.0);
vec3 Ks = vec3(0.7, 0.7, 0.7);
float a = 10.5;
vec3 La = vec3(0.2, 0.2, 0.2);
vec3 Ld = vec3(0.8, 0.8, 0.8);
vec3 Ls = vec3(0.5, 0.5, 0.5);
vec3 lightPos = vec3(10, 10, 10);
```

```
//ambient
ambient = La * Ka;

//diffuse
vec3 n_normal = normalize(normal);
vec3 light = (normalize(vec4(lightPos, 1.0) - worldPos)).xyz;
diffuse = Ld * Kd* max(0.0, dot(light, n_normal));

//specular
vec3 view_dir = normalize(view_pos - vec3(worldPos));
vec3 reflect_dir = normalize(reflect(-lightPos, normal));
specular = Ls * Ks * pow(max(0.0, dot(view_dir, reflect_dir)), a);
```

Basically is the mix up of vertex and fragment shaders of blinn phong shading and the output therefore will be added with three components of light source.

Fragment:

```
in vec2 texCoord;
in vec3 normal;
in vec3 ambient;
in vec3 diffuse;
in vec3 specular;

uniform sampler2D deer_texture;

out vec4 FragColor;

void main()
{
    vec3 obj_color = vec3(texture(deer_texture, texCoord));
    vec3 a = ambient * obj_color;
    vec3 d = diffuse * obj_color;
    FragColor = vec4((a + d + specular), 1.0);
}
```

Summing up the product of three light sources with texture color.

Flat:

Vertex:

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

out VS_OUT {
    vec3 normal;
    vec2 texCoord;
    vec4 worldPos;
} vs_out;
uniform mat4 M;
uniform mat4 V;
uniform mat4 P;
uniform vec3 camera;
```

```
out vec3 view_pos;
void main()
{
    vs_out.worldPos = M * vec4(aPos, 1.0);
    gl_Position = P * V * M * vec4(aPos, 1.0);
    vs_out.texCoord = aTexCoord;
    view_pos = camera;
    vs_out.normal = normalize((transpose(inverse(M)) * vec4(aNormal, 0.0)).xyz);
}
```

Similar to blinn phong but as there is geometry shader in flat shading, needs to modify the output as vs_out structure.

Geometry:

```
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;
in VS_OUT {
    vec3 normal;
    vec2 texCoord;
    vec4 worldPos;
} gs_in[];
    out vec3 trinormal;
    out vec2 texCoord; D
void main(void) {
    trinormal = normalize(gs_in[0].normal + gs_in[1].normal +gs_in[2].normal);
    for (int i = 0; i < 3; ++i) {
    gl_Position = gl_in[i].gl_Position;
    texCoord = gs_in[i].texCoord;
    EmitVertex();
    }
    EndPrimitive();
}
```

The inputs and outputs: layout(triangles) in;: Specifies that the input to the geometry shader is a set of triangles.

layout(triangle_strip, max_vertices = 3) out;: Specifies that the output from the geometry shader will be a triangle strip, and it will output up to three vertices (max_vertices = 3).

in VS_OUT { ... };: Defines an input block with data passed from the vertex shader (VS_OUT structure).

out vec3 trinormal; out vec2 texCoord;: Defines output variables for the geometry shader.

Main part: trinormal = normalize(gs_in[0].normal + gs_in[1].normal + gs_in[2].normal);: Calculates the normal for the entire triangle by summing the normals of its three vertices and normalizing the result. This normal is then passed to the fragment shader.

for (int i = 0; i < 3; ++i) { ... }: Loops over the three vertices of the input triangle.

gl_Position = gl_in[i].gl_Position;: Sets the position of the current vertex to the position of the corresponding input vertex.

texCoord = gs_in[i].texCoord;: Passes the texture coordinates from the input vertex to the fragment shader.

EmitVertex();: Emits the current vertex to the next pipeline stage.

EndPrimitive();: Indicates the end of the current primitive (triangle strip in this case).

Fragment:

```
in vec2 texCoord;
in vec4 worldPos;
in vec3 trinormal;
in vec3 view_pos;

uniform sampler2D deer_texture;

out vec4 FragColor;
```

```
vec3 Ka = vec3(1.0, 1.0, 1.0);
vec3 Kd = vec3(1.0, 1.0, 1.0);
vec3 Ks = vec3(0.7, 0.7, 0.7);
float shininess = 10.5;
vec3 La = vec3(0.2, 0.2, 0.2);
vec3 Ld = vec3(0.8, 0.8, 0.8);
vec3 Ls = vec3(0.5, 0.5, 0.5);
vec3 lightPos = vec3(10, 10, 10);

vec3 obj_color = vec3(texture(deer_texture, texCoord));
```

```
// Ambient
vec3 ambient = La * Ka * obj_color;

// Diffuse
vec3 n_normal = normalize(trinormal);
vec3 light = normalize(vec4(lightPos, 1.0) - worldPos).xyz;
vec3 diffuse = Ld * Kd * obj_color * max(0.0, dot(light, n_normal));

// Specular (Blinn-Phong)
vec3 view_dir = normalize(view_pos - vec3(worldPos));
vec3 halfway_dir = normalize(light + view_dir);
vec3 specular = Ls * Ks * pow(max(0.0, dot(n_normal, halfway_dir)), shininess);

FragColor = vec4((ambient + diffuse + specular), 1.0);
```

Same as blinn phong shading’s fragment shader.

Toon:

Vertex:

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

uniform mat4 M;
uniform mat4 V;
uniform mat4 P;
uniform vec3 camera;

out vec2 texCoord;
out vec4 worldPos;
out vec3 normal;
out vec3 view_pos;
```

```
worldPos = M * vec4(aPos, 1.0);
gl_Position = P * V * M * vec4(aPos, 1.0);
texCoord = aTexCoord;
view_pos = camera;
normal = normalize((transpose(inverse(M)) * vec4(aNormal, 0.0)).xyz);
```

Same as the vertex shader of blinn phong shading.

Fragment:

```
in vec2 texCoord;
in vec4 worldPos;
in vec3 normal;

uniform sampler2D deer_texture;

out vec4 FragColor;
```

```
vec3 obj_color = vec3(texture(deer_texture, texCoord));
vec4 color;
vec3 lightPos = vec3(10,10,10);
vec3 n_normal = normalize(normal);
vec3 light = (normalize(vec4(lightPos, 1.0) - worldPos)).xyz;
float intensity = max(dot(light, n_normal), 0.0);

if (intensity > 0.95)      color = vec4(1.0, 1.0, 1.0, 1.0);
else if (intensity > 0.50) color = vec4(0.7, 0.7, 0.7, 1.0);
else if (intensity > 0.25) color = vec4(0.3, 0.3, 0.3, 1.0);
else                       color = vec4(0.1, 0.1, 0.1, 1.0);

FragColor = color * vec4(obj_color,1.0);
```

Since there are only single light source in toon shading, so it will be simplify from phong shading. The only difference is to calculate the dot product of the

object normal and the view normal and based on the outcome of this sets threshold to have different color to achieve toon shading effect. vec3 obj_color = vec3(texture(deer_texture, texCoord));: Retrieves the texture color for the fragment.

vec4 color;: Declares a variable to store the final color.

vec3 lightPos = vec3(10, 10, 10);: Defines the position of the light source.

vec3 n_normal = normalize(normal);: Normalizes the interpolated normal vector.

vec3 light = (normalize(vec4(lightPos, 1.0) - worldPos)).xyz;: Calculates the normalized light direction vector.

float intensity = max(dot(light, n_normal), 0.0);: Calculates the lighting intensity using the dot product between the light direction and the normal vector, clamped to a minimum of 0.0.

The if-else statements determine the color of the fragment based on the calculated intensity. The colors are chosen based on different intensity ranges.

FragColor = color * vec4(obj_color, 1.0);: Combines the calculated color with the texture color.

Border:

Vertex:

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

uniform mat4 M;
uniform mat4 V;
uniform mat4 P;
uniform vec3 camera;

out vec2 texCoord;
out vec3 normal;
out vec3 view_pos;
```

```
texCoord = aTexCoord;
gl_Position = P * V * M * vec4(aPos, 1.0);
normal = normalize((transpose(inverse(M))* vec4(aNormal, 0.0)).xyz);
view_pos = camera;
```

Still, same as the blinn phong's vertex shader.

Fragment:

```
in vec2 texCoord;
in vec3 normal;
in vec3 view_pos;

uniform sampler2D deer_texture;

out vec4 FragColor;
```

```
// Sample texture color
vec3 obj_color = vec3(texture(deer_texture, texCoord));

// Calculate the normalized vectors
vec3 normalizedNormal = normalize(normal);
vec3 normalizedView = normalize(view_pos);

// Calculate the dot product between the normalized normal and view vectors
float dotProduct = abs(dot(normalizedNormal, normalizedView));
dotProduct += 0.25;
vec3 white = vec3(1.0,1.0,1.0);
vec3 finalColor = obj_color * dotProduct + white * (1 - dotProduct);
FragColor = vec4(finalColor, 1.0);
```

In this part is very close to toon shading, this is still single light source and finding the dot product of object and view normal. After that add 0.25 to dot product and dot white color vector and the object texture color and do interpolation to them with dot product of normal and view will get the final object color with border effect.

Dissolve:

Vertex:

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoord;

uniform mat4 M;
uniform mat4 V;
uniform mat4 P;
uniform vec3 camera;

out vec2 texCoord;
out vec3 normal;
out vec3 view_pos;
out vec4 worldPos;
```

```
texCoord = aTexCoord;
gl_Position = P * V * M * vec4(aPos, 1.0);
worldPos = vec4(aPos, 1.0);
normal = normalize((transpose(inverse(M))* vec4(aNormal, 0.0)).xyz);
view_pos = camera;
```

This is still same as the blinn phong's vertex shader.

Fragment:

```
in vec2 texCoord;
in vec3 normal;
in vec3 view_pos;
in vec4 worldPos;

uniform sampler2D deer_texture;
uniform float dissolve;

out vec4 FragColor;
```

```
vec3 obj_color = vec3(texture(deer_texture, texCoord));
float todissolve = -25.0 + dissolve;
float nowat = worldPos.x;
if (nowat < todissolve){
    discard;
}
else{
    FragColor = vec4 (obj_color, 1.0);
}
```

This dissolve fragment shader is mostly explained, it will compare the x world position of object with threshold dissolve which will increase by rotate. If the vertex's x position is smaller than threshold discard it, otherwise keep as the texture color of object.