Tutorial 2: More about shader program

Chenqi Luo

Environment Configuration

```
int main(){

    glfwInit();

    //glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    //glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    //glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

- If you want to use function of glBegin,glEnd， please comment the OpenGL core mode.

Use gluLookAt function

- Windows:

#include <windef.h>

#include <GL/GLU.h>

mac：#include <OpenGL/glu.h>

Ubuntu： #include <GL/glu.h>

```
void initPMV()
{
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(60, 800 / 600, 0.1, 100);

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	gluLookAt
	(
		3, 3, 3,
		0, 0, 0,
		0, 1, 0
	);
}
```

Write your own makefile

- G++ parameter
- -L ：directory your want to link
- -l (lowercase L)：the library name you want
- -I (capital i)：directory you want to include

- Main compiler：
- MSVC、G++、Clang

C++program compile

- g++ -E test.cpp –o test.i
- g++ -S test.cpp –o test.s
- g++ -c test.cpp –o test.o
- g++ test.o –o test

What is .obj file?

- The OBJ file is a file format developed by Wavefront for its workstation-based 3D modeling and animation software, Advanced Visualizer, which can also be read and written through Maya and other tools.

- An OBJ file is a text file that can be opened directly from the txt for viewing and editing.

untitled2 - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
# Blender v2.82 (sub 7) OBJ File: ''
# www.blender.org
mtllib untitled2.mtl
o Cube
v 1.000000 1.000000 -1.000000
v 1.000000 -1.000000 -1.000000
v 1.000000 1.000000 1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 1.000000 -1.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 1.000000 1.000000
v -1.000000 -1.000000 1.000000
vt 0.875000 0.500000
vt 0.625000 0.750000
vt 0.625000 0.500000
vt 0.375000 1.000000
vt 0.375000 0.750000
vt 0.625000 0.000000
vt 0.375000 0.250000
vt 0.375000 0.000000
vt 0.375000 0.500000
vt 0.125000 0.750000
vt 0.125000 0.500000
vt 0.625000 0.250000
vt 0.875000 0.750000
vt 0.625000 1.000000
vn 0.0000 1.0000 0.0000
vn 0.0000 0.0000 1.0000
vn -1.0000 0.0000 0.0000
vn 0.0000 -1.0000 0.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 0.0000 -1.0000
usemtl Material
```

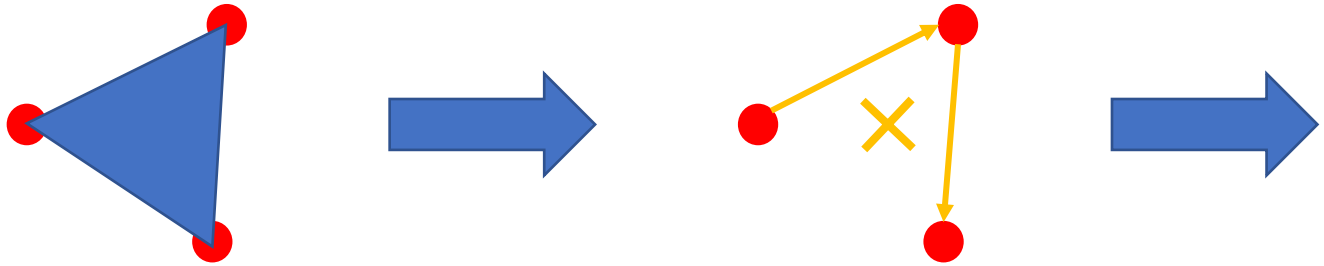
.obj的格式？

- V+：
  - v：位置，
  - vt：纹理坐标，
  - vn：法向量

- f：面片（此处为三角面片）
  - 每个顶点对应v/vt/vn

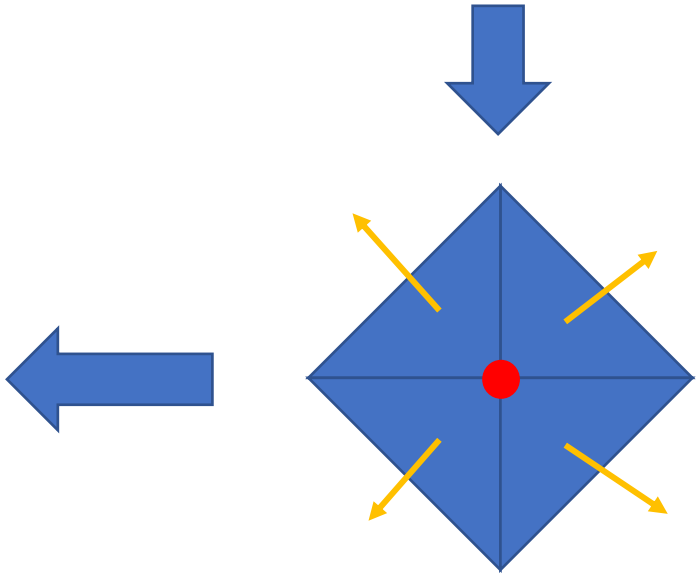
```
1
2  # obj对应的材质文件
3  # mtllib testvt.mtl
4  # 组名称
5  g default
6  # o 对象名称(Object name)
7  o testvt.obj
8  # 顶点
9  v -0.5 -0.5 0.1
10 v -0.5 -0.5 -0.1
11 v 0 0.5 0.1
12 v 0 0.5 -0.1
13 v 0.5 -0.5 0.1
14 v 0.5 -0.5 -0.1
15 # 纹理坐标
16 vt 0 1
17 vt 1 1
18 vt 0.5 0
19 # 顶点法线
20 vn 0 0 1
21 vn 0 0 -1
22 # 当前图元所用材质
23 usemtl Default
24 # s Smooth shading across polygons is enabled by smoothing groups.
25 # Smooth shading can be disabled as well.
26 s off
27 # v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3(索引起始于1)
28 f 1/1/1 5/2/1 3/3/1
29 f 6/2/2 2/1/2 4/3/2
```

计算顶点法向量？



所有面片法向量

Fragment norm 1：…
Fragment norm 2：…
Fragment norm 3：…
…



根据使用顶点的面片的法向量计算顶点法向量（平均）

vn -0.1775 0.8667 -0.4663
vn -0.7331 -0.5304 0.4257
vn -0.7315 -0.5321 0.4264
vn -0.8452 0.4439 0.2977
vn -0.8103 0.0420 0.5845
vn -0.7609 0.4929 0.4220
vn 0.3040 0.2589 0.9168
vn 0.3064 0.2386 0.9215
vn 0.3079 0.2251 0.9244

Read Obj file

Obj

```
if (token[0] == 'v' && isSpace((token[1]))) {
  token += 2;
  float x, y, z;
  parseFloat3(x, y, z, token);
  v.push_back(x);
  v.push_back(y);
  v.push_back(z);
  continue;
}

// normal
if (token[0] == 'v' && token[1] == 'n' && isSpace((token[2]))) {
  token += 3;
  float x, y, z;
  parseFloat3(x, y, z, token);
  vn.push_back(x);
  vn.push_back(y);
  vn.push_back(z);
  continue;
}

// texcoord
if (token[0] == 'v' && token[1] == 't' && isSpace((token[2]))) {
  token += 3;
  float x, y;
  parseFloat2(x, y, token);
  vt.push_back(x);
  vt.push_back(y);
  continue;
}
```

```
// face
if (token[0] == 'f' && isSpace((token[1]))) {
  token += 2;
  token += strspn(token, " \t");

  std::vector<vertex_index> face;
  while (!isNewLine(token[0])) {
    vertex_index vi = parseTriple(token, v.size() / 3, vn.size() / 3, vt.size() / 2);
    face.push_back(vi);
    int n = strspn(token, " \t\r");
    token += n;
  }

  faceGroup.push_back(face);

  continue;
}
```

Graphics Pipeline

- Two main parts
- The first part： transform your data into 2D gl_position

VertexShader->Shape Assembly->geometry shader-> rasterization->fragment shader -> Tests and Blending

The second part：Converts 2D coordinates to actual colored pixels

Decide final pixel color ->test and blending

- lighting
- shadow
- Lighting color

Vertex Shader

- All vertex related attributes can be passed to the vertex shader
- Position
- Normal
- Texture coordinate
- Color
- The vertex memory is managed by defining a VBO, which is sent to the GPU and then executed by the vertex shader.

Example

- Position
- Color

```
float vertices[] = {
    // positions          // colors
     0.5f, -0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   // bottom left
     0.0f,  0.5f, 0.0f,   0.0f, 0.0f, 1.0f    // top

};
```

Create a VAO(Vertex Array Objects)

```
unsigned int VAO;

glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);
```

Define a VBO(Vertex Buffer Objects)

```
//定义一个变量
unsigned int VBO;
// 一个缓冲对象，一个指针ID
glGenBuffers(1, &VBO);
//将我们的VBO绑定到GL_ARRAY_BUFFER这个类型上去
glBindBuffer(GL_ARRAY_BUFFER, VBO);
//把我们自己的数据复制到 当前绑定缓冲 参数1：buffer类型，2：指定传输数据的大小，
//3：数据起始指针，4：如何管理这些数据
glBufferData(GL_ARRAY_BUFFER, sizeof(myVertices), myVertices, GL_STATIC_DRAW);
```

Setting Vertex Attribute Pointer

```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

- p1：the index of attribute,
- p2：size of attribute,
- p3：data type,
- p4：if need normalization,
- p5：stride（next vertex attribute is next data），
- p6：start position pointer

Receive Data from VBO

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;

out vec3 ourColor;

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor;
}
```

- layout (location=N)
- N represents the number of properties that we just defined
- in represents input
- out represents output
- The task of the vertex shader is to perform the conversion of input data to vertices in screen space.

Leave other data to the following shaders.

Rasterization

- Line Rasterization

Bresenham method

- Triangle Rasterization

edgeWalking

By rasterizing, we can get the information after all the interpolated pixels, and then call the fragment shader to draw that pixel.

Fragment Shader

```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;

void main()
{
    FragColor = vec4(ourColor, 1.0f);
}
```

- Draw the pixel
- The pixel color can be computed by lighting ,user-defined color and shadow etc.

Fragment interpolation