CS130 - LAB - Texture Mapping

Name: Jianeng Yang SID:

Introduction

Texture mapping in GLSL consists of 3 parts

1. Uploading a texture: Handled in the OpenGL program (C/C++) part. In a typical OpenGL program, textures are read from an image file (.png,. tga etc.) and loaded in to OpenGL. The parameters of the texture (such as interpolation methods) are also set in the program.
2. Computing the texture coordinate of a vertex: In GLSL, the texture coordinate `glTexCoord[i]` for a texture i and a vertex is determined in the vertex shader. This is the coordinate of the vertices' corresponding texture positions in the image data of texture i, where i is the index of a texture (in case of multiple textures, $i = 0$ for a single texture).
3. Getting the texture color for a fragment: The texture coordinate, `glTexCoord[i]`, of texture i is readily interpolated to the fragment location by OpenGL. A lookup function such as texture2D is used to get the color from the texture.

Part I: Uploading a Texture

Read the tutorials about uploading a texture file in these links and answer the questions.

1. https://www.gamedev.net/articles/programming/graphics/opengl-texture-mapping-an-introduction-r947
2. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/#how-to-load-texture-with-glfw (Until section "How to load texture with GLFW")

Question 1: Describe briefly with your own words each one of the following functions. Look at the OpenGL documentation for reference. Link: https://www.khronos.org/registry/OpenGL-Refpages/gl4/

Google: "OpenGL 4 references"

`glGenTextures`:

Inputs: receive GLsizei **n** and GLuint * **textures**; n is the number of texture names to be generated. textures is an array that store generated texture names. If n is negative and it is error then it will generated GL_INVALID_VALUE.

`glBindTexture`:

Inputs: receive GLenum **target** and GLuint **texture**; target is the target to which the texture is bound. target must be one of GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_TEXTURE_1D_ARRAY, GL_TEXTURE_2D_ARRAY, GL_TEXTURE_RECTANGLE, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_CUBE_MAP_ARRAY, GL_TEXTURE_BUFFER, GL_TEXTURE_2D_MULTISAMPLE or GL_TEXTURE_2D_MULTISAMPLE_ARRAY. texture is the name of a texture.

`glTexParameter`:

Inputs: receive **target**,**pname**, and **param**. **target** is target to which of texture is bound. **pname** is the symbolic name of a signle-valued texture. **param** is the value of pname and for scalar commands

`glTexImage2D`:

Inputs: receive GLenum **target**, GLint **level**, GLint **internalformat**, GLsizei **width**, GLsizei **height**, GLint **border**, GLenum **format**, GLenum **type**, and const void * **data**. target is specific texture; level is the level-of-fetial number, level 0 is the base image level, level n is the nth mipmap reeduction image,and level must be zero if target is one of texture: GL_TEXTURE_RECTANGLE or GL_PROXY_TEXTURE_RECTANGLE; internalformat is the number of color components in the texture; width is the width of the texture image(at least 1024 texels wide); height is the height of the texture image or if the texture/target is GL_TEXTURE_1D_ARRAY and GL_PROXY_TEXTURE_1D_ARRAY then the height would be the number of layers in a texture array(support 2D texture image for at lease 1024 texels high and arrays for at least 256 layers deep); border is the value must be 0; format is the format of the pixel data. type is the data type of the pixel data(accept values can check in the website.); data is a pointer to the image data in memory.

Question 2: Answer the question below, briefly. Hint: see `glTexParameter`'s reference page.

1. What do minifying and magnifying mean?

🟥Minifying means that the area of the fragment in texture space is larger than one texel. While magnifying means that the area of the fragment in texture space is smaller than one texel.

2. What parameter name should be used in `glTexParameter` function in order to specify minifying function?

   🟥GL_TEXTURE_MIN_FILTER

3. What parameter name should be used in `glTexParameter` function in order to specify magnifying function?

   🟥GL_TEXTURE_MAG_FILTER

4. What are the possible minifying and magnifying functions defined by OpenGL?

   Answer: For minifying: `GL_LINEAR`, `GL_NEAREST`, `GL_NEAREST_MIPMAP_NEAREST`, `GL_LINEAR_MIPMAP_NEAREST`, `GL_NEAREST_MIPMAP_LINEAR`, `GL_LINEAR_MIPMAP_LINEAR`; for magnifiying:`GL_LINEAR` and `GL_NEAREST`;

Question 3: Read the comments and fill out the code accordingly.

```
// ==========================================================================
// Inputs:
//    data: a variable that stores the image data in ``unsigned char*''
// GL_UNSIGNED_BYTE type
//    height: an integer storing the height of the image data
//    width: an integer storing the height of the image data
// Description:
//    A piece of code that uploads the image ``data'' to OpenGL
//
// ==========================================================================
GLuint texture_id = 0;
// generate an OpenGL texture and store in texture_id variable
glGenTextures(1,&texture_id);

// set/``bind'' the active texture to texture_id
glBindTexture(GL_TEXTURE_2D, texture_id);

// Set the magnifying filter parameter of the active texture to linear
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

// Set the minifying filter parameter of the active texture to linear
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

// Set the wrap parameter of "S" coordinate to GL_REPEAT
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);

// Set the wrap parameter of "T" coordinate to GL_REPEAT
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

// Upload the texture data, stored in variable ``data'' in RGBA format
glTexImage2D(GL_TEXTURE_2D, 0,GL_RGBA, width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, data);
```

Part II: Shading with Textures in GLSL

Read the tutorials below and answer the following questions

https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/texturing.php

Introduction section only

http://www.lighthouse3d.com/tutorials/glsl-12-tutorial/simple-texture/

1. Fill out the blanks in the vertex and fragment shaders below to compute the `gl_TexCoord[0]` using `gl_TextureMatrix[0]` and `glMultiTexCoord`.

`vertex.glsl`:

```
varying vec3 N;
varying vec4 position;

// Create a uniform 2D texture sampler variable, with name "tex";
uniform sampler2D tex;

void main()
{
    // compute gl_TexCoord[0] using gl_TextureMatrix[0] and glMultiTexCoord
    gl_TexCoord[0] = (gl_TextureMatrix[0] * gl_MultiTexCoord0);
    N = gl_NormalMatrix * gl_Normal;
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
    position = gl_ModelViewMatrix * gl_Vertex;
}
```

`fragment.glsl`:

```
// create a uniform 2D texture sampler variable, with name "tex", so that
// it will be forwarded from the vertex shader
uniform sampler2D tex;



void main()
{
    // get the texture color from "tex", using a texture lookup function
    // and the s,t coordinates of gl_TexCoord[0].


    vec4 tex_color = (texture2D(tex,gl_TexCoord[0].st));

    // set gl_FragColor to tex_color
    gl_FragColor = tex_color;
}
```

Part III: Texture mapping coding practice

In this part of the lab you will be practicing texture mapping with GLSL with a skeleton code.

The skeleton code has texture files (`.tga` images) `monkey.tga` and `monkey_occlusion.tga`, as well as the base code for creating an OpenGL window, loading a monkey model and drawing it. Follow the steps below and implement texture mapping in the skeleton.

Step 0. Download the skeleton code from the lab webpage to your environment of choice (the ti-05 server works best for this) and unzip/untar it. Open the image files and observe their content.

Step 1. Uploading `monkey.tga` to OpenGL:

- Locate the TODO section towards the end of the `loadTarga` function in `application.cpp`
- Use the code in the answer to Part I Question 3, to upload "data" to OpenGL.

Note 1: the code at the beginning of the function reads the image file in "filename" to the "data" variable.

Note 2: `monkey.tga` is in RGBA format, and the data is a pointer to `UNSIGNED_BYTE` array.

Step 2. Computing texture coordinates

- Locate `vertex.glsl` and compare the code with the vertex shader code in Part II Question 2.
- Note that the code that computes the texture coordinate of the vertex is already computed, and so you have nothing to do and may go to step 3.

Step 3. Computing the color of the fragment using texture color.

- Locate fragment.glsl and compare the code with the phong shader you implemented in a previous lab. This is a phong shader without a specular component.

- Now compute the texture color just like in the fragment shader in Part II Question 3 and store it in a `tex_color` variable. But, do not assign it to `gl_FragColor`.
- Here we would like to use the texture color instead of the material color (`glFrontMaterial.diffuse.rgb`), while keeping the rest of the computation.
- Rewrite the line that computes the `gl_FragColor` to do this.