Tutorial 07

Harsha Vardhan Kode

October 7, 2015

---

1 About

In this tutorial, hierarchical modelling is demonstrated. We have used the elongated version of the colorcube to represent an arm in the hierarchy. There are three arms in total, connected to one another. Use the keys 1,2 and 3 to switch between arms. Use Arrow keys, Pgup and Pgdn to rotate the arms.

2 Running the Code

If you have a driver supporting OpenGL 4.1 then running executable will do the trick for you. But if your system supports OpenGL 3.2+. Then in order to run the code the following changes need to be made in the 07_xyz.cpp. In the line number 136 and 137,

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
```

you need to change to

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
```

Also in the 02_fshader.glsl and 02_vshader.glsl you need to modify the first line to

```
#version 330
```

Once you make the above changes you can compile and run the executable.

3 Understanding the code

The understanding for the code of basics is documented in the previous tutorials. This tutorial would not go into much detail about those again, but explain all the new stuff.

3.1 Hierarchy node

First, we'll look at the new class in the file **hierarchy_node.cpp**.

```
class HNode {
  //glm::vec4 * vertices;
  //glm::vec4 * colors;
  GLfloat tx,ty,tz,rx,ry,rz;

  std::size_t vertex_buffer_size;
  std::size_t color_buffer_size;

  GLuint num_vertices;
  GLuint vao,vbo;

  glm::mat4 rotation;
  glm::mat4 translation;

  std::vector<HNode*> children;
  HNode* parent;

  void update_matrices();

  public:
  HNode (HNode*, GLuint, glm::vec4*,  glm::vec4*, std::
     size_t, std::size_t);
  //HNode (HNode* , glm::vec4*,  glm::vec4*,GLfloat,GLfloat
     ,GLfloat,GLfloat,GLfloat,GLfloat);

  void add_child(HNode*);
  void render();
  void change_parameters(GLfloat,GLfloat,GLfloat,GLfloat,
     GLfloat,GLfloat);
  void render_tree();
  void inc_rx();
  void inc_ry();
```

```
    void inc_rz();
    void dec_rx();
    void dec_ry();
    void dec_rz();
  };
```

This class is pretty simple, it stores the handles to the VAO and VBO of the object and also its translation and rotation parameters. There are two main functions : render and render_tree. render function renders just the object with the current matrix stack (we will discuss how the matrix stack is updated later). render_tree function renders the object along with its all children recursivey. Apart from that there are functions to increment and decrement the rotation parameters. Next we look at its constructor:

```
void initBuffersGL(void)
{

HNode::HNode(HNode* a_parent, GLuint num_v, glm::vec4*
   a_vertices, glm::vec4* a_colours, std::size_t v_size, std
   ::size_t c_size){

    num_vertices = num_v;
    vertex_buffer_size = v_size;
    color_buffer_size = c_size;
    // initialize vao and vbo of the object;


    //Ask GL for a Vertex Attribute Objects (vao)
    glGenVertexArrays (1, &vao);
    //Ask GL for aVertex Buffer Object (vbo)
    glGenBuffers (1, &vbo);

    //bind them
    glBindVertexArray (vao);
    glBindBuffer (GL_ARRAY_BUFFER, vbo);


    glBufferData (GL_ARRAY_BUFFER, vertex_buffer_size +
       color_buffer_size, NULL, GL_STATIC_DRAW);
    glBufferSubData( GL_ARRAY_BUFFER, 0, vertex_buffer_size,
       a_vertices );
    glBufferSubData( GL_ARRAY_BUFFER, vertex_buffer_size,
       color_buffer_size, a_colours );
```

```
    //setup the vertex array as per the shader
    glEnableVertexAttribArray( vPosition );
    glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE,
       0, BUFFER_OFFSET(0) );

    glEnableVertexAttribArray( vColor );
    glVertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0,
       BUFFER_OFFSET(vertex_buffer_size));


    // set parent

    if(a_parent == NULL){
      parent = NULL;
    }
    else{
      parent = a_parent;
      parent->add_child(this);
    }

    //initial parameters are set to 0;

    tx=ty=tz=rx=ry=rz=0;

    update_matrices();
  }
```

It takes as arguments: its parent, the vertex arrays, and their respective sizes. The allocation and initialization of the vao and vbo corresponding to the object happens in the constructor itself. The initial translation and rotation parameters are set to 0. The parent node's child list is automatically updated here. Now we look at the main functions of the class

```
  void HNode::render(){

    //matrixStack multiply
    glm::mat4* ms_mult = multiply_stack(matrixStack);

    glUniformMatrix4fv(uModelViewMatrix, 1, GL_FALSE, glm::
       value_ptr(*ms_mult));
    glBindVertexArray (vao);
    glDrawArrays(GL_TRIANGLES, 0, num_vertices);

    // for memory
    delete ms_mult;
```

```
}
```

You'll see that here, all the matrices in the matrix stack, which is a global variable, are multiplied to one another and then passed to the vertex shader. The basic code for rendering is the same as in all the previous tutorials.

```
void HNode::render_tree(){

  matrixStack.push_back(translation);
  matrixStack.push_back(rotation);

  render();
  for(int i=0;i<children.size();i++){
    children[i]->render_tree();
  }
  matrixStack.pop_back();
  matrixStack.pop_back();

}
```

You will see that in the render_tree function the translation and the rotation matrices are pushed into the matrix stack. The recursion happens in a depth-first fashion. Once the current node and all its children are rendered, the matrices are popped of the stack.

3.2 Initialisation

Now, we come to the changes in the main file.

```
void initBuffersGL(void)
{

  // Load shaders and use the resulting shader program
  std::string vertex_shader_file("07_vshader.glsl");
  std::string fragment_shader_file("07_fshader.glsl");

  std::vector<GLuint> shaderList;
  shaderList.push_back(csX75::LoadShaderGL(GL_VERTEX_SHADER,
     vertex_shader_file));
  shaderList.push_back(csX75::LoadShaderGL(GL_FRAGMENT_SHADER
     , fragment_shader_file));

  shaderProgram = csX75::CreateProgramGL(shaderList);
```

```
  glUseProgram( shaderProgram );

  // getting the attributes from the shader program
  vPosition = glGetAttribLocation( shaderProgram, "vPosition"
      );
  vColor = glGetAttribLocation( shaderProgram, "vColor" );
  uModelViewMatrix = glGetUniformLocation( shaderProgram, "
     uModelViewMatrix");

  // Creating the hierarchy:
  // We are using the original colorcube function to generate
      the vertices of the cuboid
  colorcube();

  //note that the buffers are initialized in the respective
     constructors

  node1 = new csX75::HNode(NULL,num_vertices,v_positions,
     v_colors,sizeof(v_positions),sizeof(v_colors));
  node2 = new csX75::HNode(node1,num_vertices,v_positions,
     v_colors,sizeof(v_positions),sizeof(v_colors));
  node2->change_parameters(2.0,0.0,0.0,0.0,0.0,0.0);
  node3 = new csX75::HNode(node2,num_vertices,v_positions,
     v_colors,sizeof(v_positions),sizeof(v_colors));
  node3->change_parameters(2.0,0.0,0.0,0.0,0.0,0.0);
  root_node = curr_node = node3;

}
```

Here, the shaders are initialised, and three nodes are created, and are linked according to their hierarchy. As mentioned before, the colorcube with modified vertices is used for rendering.

```
void renderGL(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  matrixStack.clear();

  //Creating the lookat and the up vectors for the camera
  c_rotation_matrix = glm::rotate(glm::mat4(1.0f), glm::
     radians(c_xrot), glm::vec3(1.0f,0.0f,0.0f));
  c_rotation_matrix = glm::rotate(c_rotation_matrix, glm::
     radians(c_yrot), glm::vec3(0.0f,1.0f,0.0f));
  c_rotation_matrix = glm::rotate(c_rotation_matrix, glm::
     radians(c_zrot), glm::vec3(0.0f,0.0f,1.0f));
```

```
  glm::vec4 c_pos = glm::vec4(c_xpos,c_ypos,c_zpos, 1.0)*
     c_rotation_matrix;
  glm::vec4 c_up = glm::vec4(c_up_x,c_up_y,c_up_z, 1.0)*
     c_rotation_matrix;
  //Creating the lookat matrix
  lookat_matrix = glm::lookAt(glm::vec3(c_pos),glm::vec3(0.0)
     ,glm::vec3(c_up));

  //creating the projection matrix
  if(enable_perspective)
    projection_matrix = glm::frustum(-7.0, 7.0, -7.0, 7.0,
       1.0, 7.0);
    //projection_matrix = glm::perspective(glm::radians(90.0)
       ,1.0,0.1,5.0);
  else
    projection_matrix = glm::ortho(-7.0, 7.0, -7.0, 7.0,
       -5.0, 5.0);

  view_matrix = projection_matrix*lookat_matrix;

  matrixStack.push_back(view_matrix);

  node1->render_tree();

}
```

In the rendering function, The matrix stack is cleared, and the viewing matrix is pushed in first. the creation of the viewing matrix is explained in previous tutorials. After this, the render_tree function of the root node in the hierarchy is called to render the full hierarchy.

4 Shaders

4.1 Vertex Shader

```
attribute vec4 vPosition;
attribute vec4 vColor;
varying vec4 color;
```

The vertex shader is similar to the shader in the tutorial 4. It just multiplies the modelview matrix to the vertex position.

4.2 Fragment Shader

The fragment shader is quite similar to the code previous tutorial and you can easily see that we are assigning the color of the fragment, from our input from vertex shader.

5 Keyboard Input

In the key_callback() function in gl_framework.cpp we deal with various keyboard inputs. Various additional inputs have been added to handle the movement of the arms and also to switch between arms.

```
else if (key == GLFW_KEY_1 && action == GLFW_PRESS)
   curr_node = node1;
 else if (key == GLFW_KEY_2 && action == GLFW_PRESS)
   curr_node = node2;
 else if (key == GLFW_KEY_3 && action == GLFW_PRESS)
   curr_node = node3;
 else if (key == GLFW_KEY_LEFT && action == GLFW_PRESS)
   curr_node->dec_ry();
 else if (key == GLFW_KEY_RIGHT && action == GLFW_PRESS)
   curr_node->inc_ry();
 else if (key == GLFW_KEY_UP && action == GLFW_PRESS)
   curr_node->dec_rx();
 else if (key == GLFW_KEY_DOWN && action == GLFW_PRESS)
   curr_node->inc_rx();
 else if (key == GLFW_KEY_PAGE_UP && action == GLFW_PRESS)
   curr_node->dec_rz();
 else if (key == GLFW_KEY_PAGE_DOWN && action ==
    GLFW_PRESS)
   curr_node->inc_rz();
```