DeepL

Subscribe to DeepL Pro to translate larger documents.
Visit www.DeepL.com/pro for more information.

TH COLOGNE
PROF. DR. FLORIAN NIEBLING
ADVANCED MEDIA INSTITUTE

Technology
Arts Sciences
TH Köln

APRIL 24, 2024
SUBMISSION: NO LATER THAN MAY 10, 2024

COMPUTER GRAPHICS and Animation

SHEET 1 - Geometry

The internship consists of several related tasks in which different parts of a framework are implemented. The aim is to have completed a functioning game and the framework with the necessary tools for your final project at the end.
In this task sheet, you will deal with drawing the geometry of objects. Specifically, knowledge of VertexArrayObjects, VertexBufferObjects and IndexBufferObjects is required here. With this assignment, you have been given a complete project on which the rest of the practical will be based. You should therefore use this project from this practical course onwards.

1.1 3D geometry

You have an array with float values, which are structured according to the following scheme:

$$VBO = [\underbrace{pos1_x, pos1_y, pos1_z, col1_r, col1_g, col1_b}_{\text{Vertex1}}, \underbrace{pos2_x, pos2_y, pos2_z, col2_r, col2_g, col2_b}_{\text{Vertex2}}, ...]$$

1.1.1 VBO

Draw the following points in the following coordinate system and assign the corresponding RGB color to the points.

```
VBO=[ -0.5,-0.5,  0.0,   0.0,   0.0,   1.0,
       0.5,-0.5,  0.0,   0.0,   0.0,   1.0,
       0.5,  0.5, 0.0,   0.0,   1.0,   0.0,
       0.0,  1.0, 0.0,   1.0,   0.0,   0.0,
      -0.5,  0.5, 0.0,   0.0,   1.0,   0.0 ]
```

1.1.2 Stride and offset

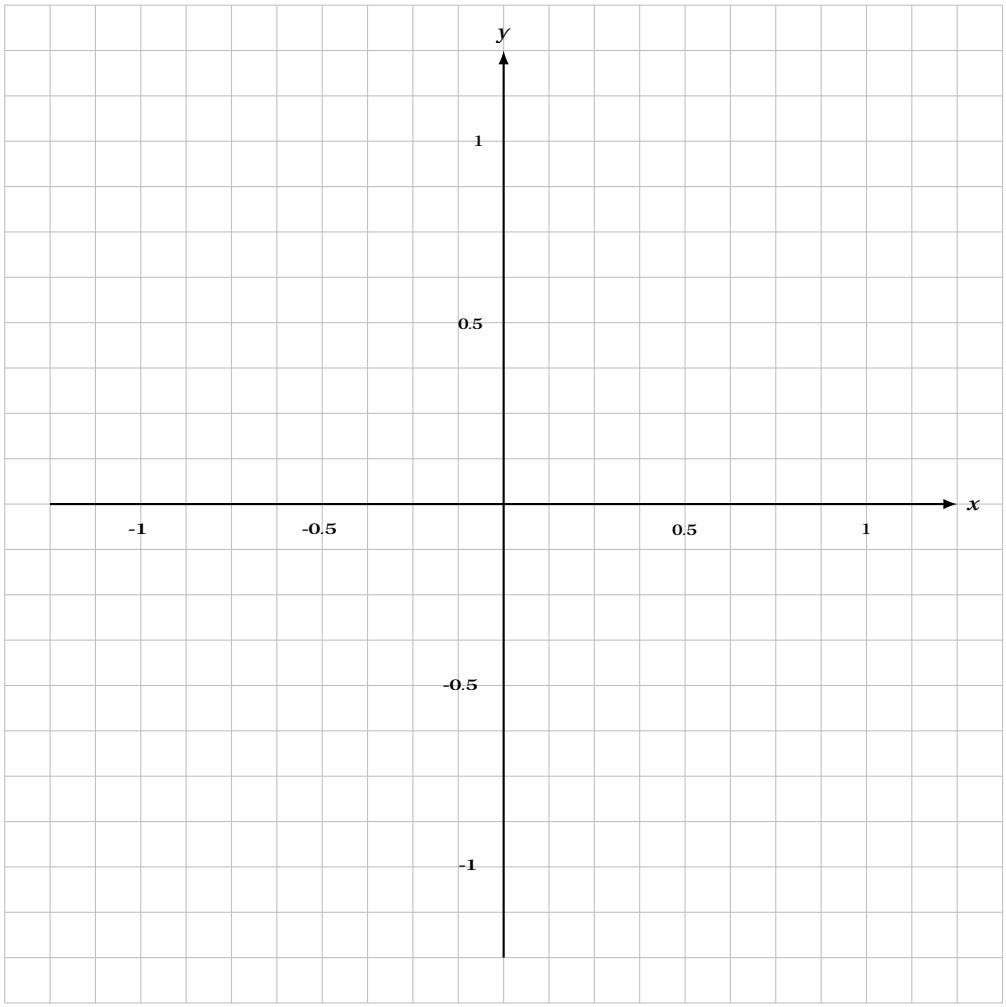
What would the stride and offset look like for both the position data and the color values from task 1.1.1?

1.1.3 IBO

You also have an array that contains the geometry structure. In this case, each line of the IBO is to be interpreted as a triangle.

```
IBO=[ 0 ,  1,  2,
      0,   2,  4,
      4,   2,  3 ]
```

Draw the resulting triangles in the coordinate system on the following page.


y
1
0.5
x
-1
-0.5
0.5
1
-0.5
-1

1.2 Basic geometry

This implementation of the basic geometry consists of three parts and is based on the geometry from task 1.1. You are also required to create and test a mesh yourself.

1.2.1 Scene-init

All objects of the geometry to be displayed are initialized in the Init `block` of the `cga.exercise.game.Scene` class. To begin with, the vertices known from task 1.1.1 should be defined here. The definition also includes the description of the individual vertex attributes so that the shader k n o w s how to interpret the data. The `VertexAttribute` class in the `cga.exercise.components.geometry` package is used to define a VertexAttribute:

```
data class VertexAttributes (
    var n: Int ,        // Number of components of this attribute
    var type : Int ,    // Type of this attribute var
    stride : Int , // Size in bytes of a whole var vertex
    offset : Int        // Offset in bytes from the beginning of the vertex to the
        location of this attribute data
)
```

Then use the indices from the previous task to define the geometry and create a mesh with these three attributes.

1.2.2 Mesh class

Complete the `init block` of the `cga.exercise.components.geometry.Mesh` class to create and manage the necessary buffers (VAO, VBO & IBO). You must also implement the `render()` method so that the geometry of the mesh is redrawn in each frame.
The following procedure should help you a little. Please also use the presentation slides.
During initialization:

1) Create and bind a VAO. Each VBO and IBO that you bind in the sequence is assigned to the current VAO.
2) Create and bind vertex and index buffers.
3) Fill the vertex and index buffers with the corresponding data.
4) Activate and define the respective vertex attributes.
5) Then everything should be untied (unbind) to avoid accidental changes to the buffers and VAO. Remember to unbind the VAO first.

When rendering:

1) Bind the VAO.
2) Draw the elements.
3) Release the binding to avoid accidental changes to the VAO.

1.2.3 Scene-render()

The `render()` method of the `Scene` class is called by the framework in each frame. All objects that are to be displayed must be integrated into the rendering pipeline here. In addition, the shader to be used for rendering must be defined beforehand. To transfer responsibility to the respective shader, please use the `use()` method of the `ShaderProgram`.
If you have done everything correctly, the image in Figure 1 should appear:

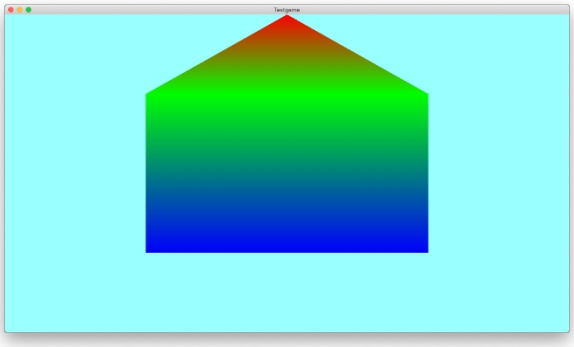
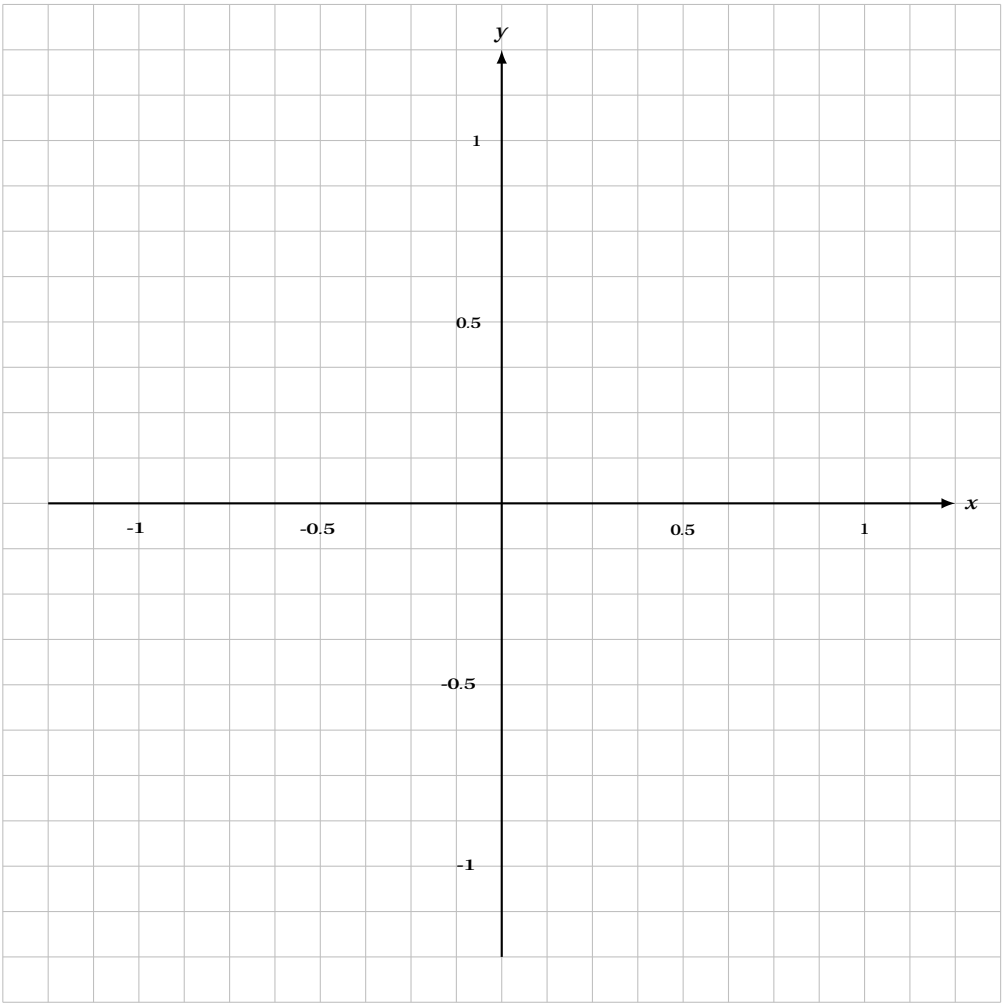


Figure 1: Rendered house

1.2.4 Initials

Change the vertex and index data so that the initials of your name are displayed. The following coordinate system should help you to determine the corresponding coordinates. **Keep the font simple. An O does not have to be round.**



1.2.5 Backface culling

Now activate backface culling by calling the method in the scene with the appropriate parameters. It may also be useful to read the documentation[1]. The triangles created by the vertex data supplied are defined in an anti-clockwise direction. If there are gaps in your initials, please correct the orientation.

1.3 Object handling

The basics of geometry handling should now be understood so that we can go one step further and work with more complex objects. More complex objects can consist of several meshes (in this case, a single mesh models a feature of the object).

1.3.1 OBJLoader

The supplied OBJLoader takes over the reading and interpretation of obj files (manual included). Please load the object `sphere` from the `assets folder` into the scene.

[1] `https://www.khronos.org/opengl/wiki/Face_Culling`

The sequence of actions can be as follows:

a) `loadOBJ(...)` creates an OBJResult

b) The OBJResult contains, among other things, a list of the meshes it contains. These can be accessed as follows, for example:
val objRes: OBJResult
... //get OBJResult by OBJLoader
val objMeshList: MutableList<OBJMesh> = objRes.objects[0].meshes

c) Each OBJMesh has the getter methods for `vertexData` and `indexData`, which you can use to access the mesh data.

d) The vertices store both position (pos), texture coordinates (tex) and the normal (norm) and have the following structure:

$$VBO = [\underbrace{pos1_x, pos1_y, pos1_z, tex1_u, tex1_v, norm1_x, norm1_y, norm1_z}_{\text{Vertex1}}, \underbrace{pos2_x, pos2_y, pos2_z}_{\text{Vertex2}}, ..., ...]$$

Define the `VertexAttributes` and create the mesh `objects`.

e) Render the object(s) in the `Scene.render()` method.

Start the project to ensure that you have implemented everything correctly. Please compare your render result with Figure 2 and interpret it. Then switch on the depth test[2] in the Scene class using the function provided. You should now see the result in Fig. 2. The window has the dimensions -1 to 1 in the x and y directions and 0 in the z direction. The sphere is defined as a unit sphere and therefore takes up the entire window. In this case, the color comes from the vertex attribute of the texture coordinates (vec2), which is interpreted as `vec3` in the shader and for which the
b component is set to 0.

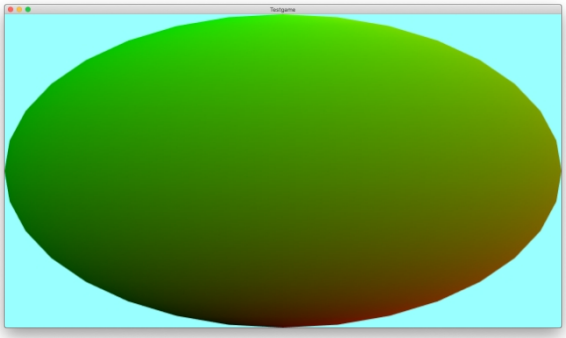


Figure 2: Integration of the sphere

Finally, change the background color to black by changing the following line in the Init block of the scene to the given values.

```
init {
...
        gl Clear Color (0.0  0.0    0.0    1.0 f); GLError . check Throw ()
...     f,                  f,     f,
}
```

2https://registry.khronos.org/OpenGL-Refpages/gl4/html/glDepthFunc.xhtml