Lab 9

- Interaction using the mouse and the keyboard
- A simple data structure for storing triangular meshes
- Textures
- Transparency – Blending

1.1 Interaction using the mouse and the keyboard

Analyze the example **WebGL_example_26.html**.

The aim of this example is to illustrate how the keyboard and the mouse can be used to allow for an intuitive interaction with the displayed model(s).

Identify the new interaction functionalities available in this example and analyze the functions that allow to:

- Use the ***cursor keys*** to increment / decrement the rotation speed around the XX axis and the YY axis.

- Use the ***Page Up*** and ***Page Down*** keys to perform ***zoom out / zoom in***.

- Use the **mouse** to change model orientation by selecting a point on the canvas and moving the mouse cursor, while pressing one of the mouse buttons

Task:

- Choose two keyboard keys (e.g., Z and X) and control rotation speed around the ZZ axis.

1.2 A simple data structure for storing triangular meshes

Analyze the example **WebGL_example_27.html**.

The aim of this example is to illustrate how a simple data structure can be used for storing the triangular mesh representing the surface of a model.

The coordinates of the mesh vertices are stored in a **vertices array**; another **colors array** stores the RGB color values associated with each vertex; and an **indices array** stores the indices of the (CCW) vertices defining each one of the mesh triangles.

The use of such a data structure implies some changes to the functions **initBuffers()** and **drawModel()**.

Task:

In file **WebGL_example_27.js** a cube is defined using that simple data structure.

Analyze:

- The vertices array – How are arranged the vertices defining the cube faces? Why does each vertex appear more than once?
- The colors array – Which color is associated with each one of the cube faces?
- The indices array – How is represented each one of the cube faces?

Task:

Analyze the changes done to the **initBuffers()** and **drawModel()** functions.

Note how the mesh triangles are now rendered by calling the function **gl.drawElements()**.

Question:

What will happen if no replicated vertices are allowed in the vertices array?

Task:

Carry out that change in the data structure and verify how the cube is displayed.

1.3 Textures

Analyze the example **WebGL_example_28.html**.

Attention:

Some browsers only allow loading texture image files from a Web server (e.g., Chrome), while others do not place such a restriction (e.g., Microsoft Edge).

The aim of this example is to illustrate how to apply textures to a simple model, which implies significative changes to the code of the previous examples.

Note that there no longer is a color associated with each mesh vertex, since each face will now be rendered using a texture.

Task:

Identify the most important changes made:

- Mapping the texture corner points into the four vertices defining each one of the cube faces.
- Modifications to the ***shaders'*** code.
- Corresponding changes in the **initShaders()** function.
- New functions for loading a texture image and setting the texture rendering features.
- Modifications to the code of functions **initBuffers()** and **drawModel()**.

Task:

Using an image of your choice, load it and check if it is correctly rendered upon each one of the cube faces.

Note that you have to use a **square image**, whose number of rows / columns has to be a power of 2.

Suggestion:

Modify the code so that a different texture image is used for each one of the cube faces.

And, also, different texture images for each one of the cube instances.

1.4 Transparency – Blending

Analyze the example **WebGL_example_29.html**.

Attention:

Some browsers only allow loading texture image files from a Web server (e.g., Chrome), while others do not place such a restriction (e.g., Microsoft Edge).

The aim of this example is to illustrate how to simulate transparent materials.

That can be accomplished by using an alternative to the *Z-Buffer* algorithm to correctly represent pixels in the *frame buffer* – see how that is done.

When using ***blending***, a blend function combines the color information from the various superimposing fragments that contribute to the color of each pixel in the *frame buffer*.

Task:

Identify the most important changes made:

- Modifications to the ***shaders'*** code – Analyze their meaning.
- Corresponding changes in the **initShaders()** function.
- Modifications to the code of function **drawModel()**, to set the blending function and the value of the **alpha** parameter.

Task:

Allow the user to increment / decrement the **alpha** value, between 0.0 and 1.0, using the keyboard.

Verify how the cubes are displayed.