SRGGE - LAB Assignement

Emeline GOT

June 15, 2018

Each session is accessible in the main menu of the window in SRGGE : Exercise 1 *(class ex1)*, Exercise 2 and 3 *(class ex2)*, Exercise 4 *(class ex4)*, Exercise 5 *(class ex5)*, Exercise 6 *(class ex6)*.

1 Session 1

All of the functions for session 1 are implemented in the *ex1* class. The loading and drawing of the models are realized by the OpenGL functions **initializeGL()**, **paintGL()** and **initVertexBuffer()**. The interface is implemented in **controlPanel()**. The NxN copies of the same object are created thanks to a loop in the **paintGL()** function.

2 Session 2

The vertex clustering on a regular grid is implemented in the *ex2* class. The function **cellid** allows to get the cell number in which a vertex is placed depending on the level of details we want. The **computeGrid** function creates the grid and computes the mean of the vertices in each cell, computes the faces and the normals. In **initVertexBuffer**, we send the corresponding information depending on the button that is selected in the interface. Basic vertex clustering corresponds to the ***LOD Basic*** button. We can select the level of details with the slider under the radio buttons.

3 Session 3

3.1 Basic

The quadric error metrics method is implemented in the *ex2* class. The function **computeQ** computes the Q matrix of each vertex and stores it in the QMatrices variable. **computeQuadric** is similar to the **computeGrid** function but instead of computing the mean of all the vertices, it takes the vertex obtained with the quadric error metrics formula ($Q^T * (0, 0, 0, 1)$). We also compute the normals and the faces as before. Quadric error metrics corresponds to the button ***LOD Quadric*** in the interface.

3.2 Advanced

The implementation of the shape preserving algorithm is in *ex2* class. The function **cellnorm** gives the binary number that corresponds to the classification of the normals according to their coordinates' signs. The grid computation is made in **computeGridShape** and is similar to **computeGrid** but we consider 8 additional divisions of the cell depending on the value given by **cellnorm**. This algorithm is activated by the ***LOD Shape Preserving*** button in the interface.

4 Session 4

4.1 Basic

The session 4 is implemented in the *ex4* class. We are using 5 different levels of details that we compute thanks to the code of the exercise 2 and store them with the **computeVFN** function (LOD=8,16,32,64 and complete model). The **totTriangles** function sums up the number of triangles displayed on the screen depending on which LOD is displayed. In **computeIndivCost**, we compute the cost of each model displayed on the screen ($\frac{d}{2^L D}$). The functions **findIDmin** and **findIDmax** get the indices (i,j) of the models that have respectively the lowest cost and the biggest cost [1] if we increase or decrease their level of details. The **computeLevels** function increases or decreases the level of details of one model at each iteration if it is possible depending on the number of triangles displayed. We call this function in **paintGL** so that the **Levels** matrix can be updated each time we change the viewport. Depending on the value of **Levels(i,j)**, we assign a different color in the shader so that the models are printed in different colors if they have different LOD.

This implementation corresponds to the ***Basic*** button of the interface.

4.2 Advanced

The hysteresis computation is made in the *ex4* class with the **computeAdvanced**. The function is very similar to the basic **computeLevels** with one more check for the number of frames. In the **Model_Frames** matrix, we store the number of the frame at which each model had its last change of level. We check if the value stored and the number of frame at which the function is called have a difference greater than 10 frames, in which case we change the level and update the **Model_Frames** matrix.

This implementation corresponds to the ***Advanced*** button of the interface.

[1] Actually, it would make more sense to say they are benefits and not costs. We try to increase the level of the model with the bigger benefit (closest to the viewport) and decrease the level of the model with the lowest one (the one we would the less notice the "bad" LOD).

5 Session 5

For this exercise and the next one, we use these commands for the camera;

- Key up : move forward
- Key down : move backward
- Left key : move to the left
- Right key : move to the right
- Z : move up
- X : move down
- W : move camera up
- S : move camera down
- A : move camera to the left
- D : move camera to the right

5.1 Basic

Session 5 is implemented in the *ex5* class. In the **initVertexBuffer** function, we use VAO for the quads of the ground, the walls and for each model that we store. **computeQuad** compute a ground quad (vertices, faces and normals). **computeWall** computes a vertical quad for the walls. The map we use for the museum are maps of 0 for ground quads, 1 to delimit the rooms and bigger numbers for models to be placed on the map. Thus, 0 can be interior or exterior so we use the **checkIE_indiv** to check if a 0 is surrounded by four 1 (up, down, left, right). We use a second check implemented by **checkIE** that checks if the 0 found inside are not false positive. The function **checkWall** gives the position in which the walls should be on a wall quad depending on the neighbouring quads. In **paintGL**, we display the ground, the walls and then the models if they have been loaded. If the number asked by the map is bigger than the number of model that is asked, the tile will stay white. The functions **loadMap** and **importMap** implement the button loading of the interface and load a txt file as map for the museum. The function **LoadModel** loads model and stores them in a vector. As long as we keep loading, it keeps storing.

5.2 Advanced

It is possible to use a bigger map for the museum such as the file *museum_advanced.txt*.

6 Session 6 - *Not working correctly*

Session 6 is implemented in *ex6* class. **SelectSides** function selects the 2 random sides to create the ray between them. In **ReturnCoords**, we compute the random coordinates depending on which side was selected. The **Rays** function is an implementation of the Bresenham algorithm found online that gives the

cell crossed by the ray. **addPoint** adds the cell to the visibility set computed or creates a new set if we encounter a wall (a cell is refered to by its down left corner). The function **sendSet** is called to update the visibility sets of each quad of the map. We perform an add and check to update the visibility sets of the quads in order to avoid repetition with the **addAndCheck** function. **Visisiblity** creates an empty Visibility set for each quad as initialization and trace r rays. We call this function whenever we load a map to compute the visibility cells of the loaded map. In **paintGL**, we get the position of the camera and display only the quads that are visible from the quad in which the camera is placed.