FPS
paintGL()
iniTimer
endTimer

Universitat Politècnica de Catalunya

Scalable Rendering for Graphics and Game Engines

Master in Innovation and Research in Informatics

Sessions 1 to 3

Project Report

Author:
Miguel Moreno

Teacher:
Marc Comino Trinidad

1 Preamble

The project is designed to work on Linux and QT5. Several extra models are added to ease testing.

2 Session 1: Setting up the viewer

2.1 Load and Draw: buffers

Note: this draws directly from the first delivery of FRR, albeit with slight modifications

First, buffers are initialized:

```
    glGenVertexArrays(1, &VAO[i]);
    glBindVertexArray(VAO[i]);

    // Initialize VBO for vertices
    glGenBuffers(1, &vbo_v_id[i]);
    glBindBuffer(GL_ARRAY_BUFFER, vbo_v_id[i]);
    glBufferData(GL_ARRAY_BUFFER, LOD.vtxPerLOD[i].size() * sizeof(float), &LOD.
    vtxPerLOD[i][0], GL_STATIC_DRAW);
    glVertexAttribPointer(kVertexAttributeIdx, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(kVertexAttributeIdx);


    // Initialize VBO for normals
    glGenBuffers(1, &vbo_n_id[i]);
    glBindBuffer(GL_ARRAY_BUFFER, vbo_n_id[i]);
    glBufferData(GL_ARRAY_BUFFER, LOD.normPerLOD[i].size() * sizeof(float), &LOD.
    normPerLOD[i][0], GL_STATIC_DRAW);
    glVertexAttribPointer(kNormalAttributeIdx, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(kNormalAttributeIdx);

    glBindVertexArray(0);

    // Initialize VBO for faces
    glGenBuffers(1, &faces_id[i]);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, faces_id[i]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, LOD.facesPerLOD[i].size() * sizeof(int), &
    LOD.facesPerLOD[i][0], GL_STATIC_DRAW);
```

As you can see, I have set LOD VAOs and $3 \times LOD$ VBOs: one per each face, normal and vertex data for all levels of detail.

2.2 Load and Draw: Instancing

Note: this draws directly from the first delivery of FRR, albeit with slight modifications

We bind the VAO buffer and useglDrawElementsto render the faces of the mesh. The only difference is that instead of `glDrawElements` I use `glDrawElementsInstanced`, which has another parameter: the number of instances N to use. As the N has to be squared, I set it as N^2.

```
glBindVertexArray(VAO[ myLod ]);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, faces_id[ myLod ]);

//glDrawElements(GL_TRIANGLES, mesh_->faces_.size(), GL_UNSIGNED_INT, 0);
glDrawElementsInstanced(GL_TRIANGLES, LOD.facesPerLOD[ myLod ].size(),
GL_UNSIGNED_INT, 0, num_instances * num_instances);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Now, I have to ensure that I can see all the instances in the scene, in addition to set the distance between them. What I did was set it through the vertex shader by setting the X coordinate offset as *myInstance* div (N^2) to set the row, and the Z coordinate offset as *myInstance* mod (N^2) to set the column, so that the objects would be displayed in a matrix. This offset is summed to the vertex information of the model.

```
//...
uniform int num_instances;
uniform float offset;

smooth out vec3 eye_normal;
smooth out vec3 eye_vertex;

void main(void) {
    int i = gl_InstanceID / num_instances;
    int j = gl_InstanceID % num_instances;
    vec3 posOffset = vec3( offset*i, 0.0f, offset*j );
    vec4 view_vertex = view * model * vec4(vert + posOffset, 1);

    //...

    gl_Position = projection * view_vertex;
}
```

To send the information regarding N and the offset desired (which can be set in the user interface), I send two respective uniforms into the vertex shader.

```
glUniform1i(numinst_location, num_instances );
glUniform1f(offset_location, dist_offset );
```

The camera will pivot around the very first instance. This is a deliberate solution (instead of centering the camera to the center of the matrix), so that anyone can see how objects are seen at a distance. For this, zooming in/out limits have also been increased.

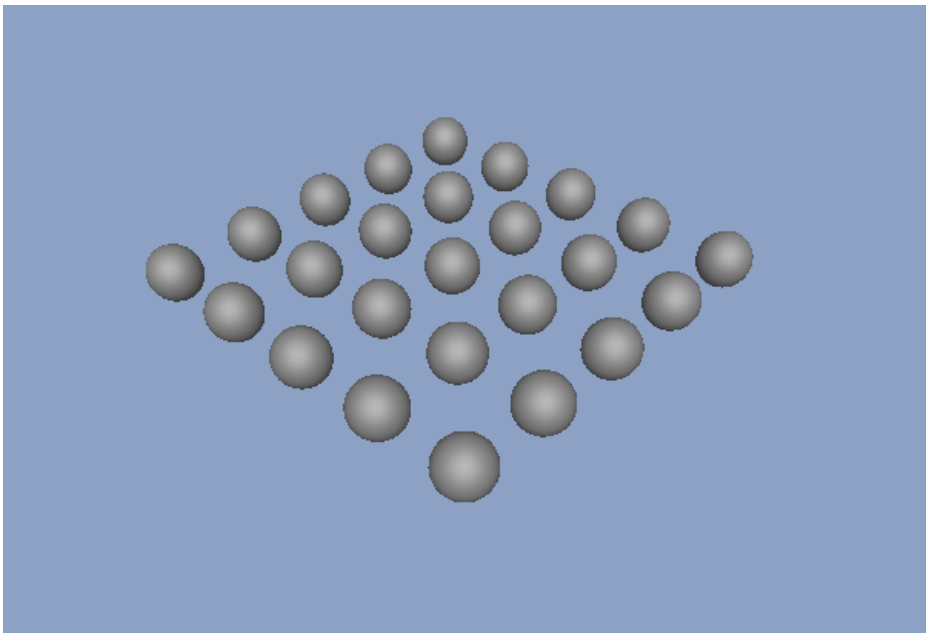

Figure 1: Render of 5x5 `spheres`

2.3 Framerate

Framerate is set as: $FrameRate = \frac{CallsToPaintGL}{endTime - iniTime}$

The function `paintGL()` is called everytime there are new modifications to the scene.

I set an integer function `FPS` which increases for each `paintGL()` call, and two timers `iniTimer` and `endTimer`, which set the starting and finishing time points for the FPS reads This is then converted into a string and sent through a Qt slot, and I restart the iniTimer and FPS count.

3 Session 2: Vertex Clustering

Vertex Clustering is implemented on its own class. The level of detail is determined by the amount of subdivisions each grid will have per dimension from 2..8; for LOD=1, the original model will be used. As an example, for LOD=2 the grid will be have $2 \times 2 \times 2$ cells indexed from 0 to 7.

The cluster building process will build all 8 levels of detail, so when you choose which level you want to see on interaction, instead of rebuilding everything, it will select that level of detail from an indexed set of arrays describing the simplified faces, vertices and normals.

The steps are:

1. Given a 3D grid, for each vertex of the original model: lookup in which cell of the grid the

vertex lies in range $[0 \,..\, LOD^3 - 1]$, and store it on the grid. if the cell to place are the maximal borders in any coordinate, then we subtract 1 to each affected coordinate.

2. Then, it gets the mean vertex for each grid cell and stores it as a new vertex. Careful: if a grid cell has no vertices (I'll call it a "skip" of the created vertex process), the vertex-to-grid array counter should take into consideration the amount of skips and I decrement it with respect to the current cell number.

3. Afterwards, I can find the new edges. The process goes as follows:

   (a) With all the vertices from the original faces, I check which those faces which have vertices on three different grid cells.

   (b) If a face has vertices on three different cells, I can add the cell indices of the grid as a "face"; as there is one vertex per cell, those will map to .

4. Finally, with the vertex and face information I can calculate the normals, by using the algorithm found on `Mesh_io::ComputeFaceNormals`, with slight improvements:

   (a) First I obtain the normals per face (using the cross-product between two of its edges), and then I set the normals to all vertices. For each face I have to get the cosine of the angle of incidence ϕ from each vertex's dot product value.

   (b) When I'm done calculating with the new normals, I normalize them all.

As a note, all methods to obtain the new vertices can be changed from the user interface via its combo box.

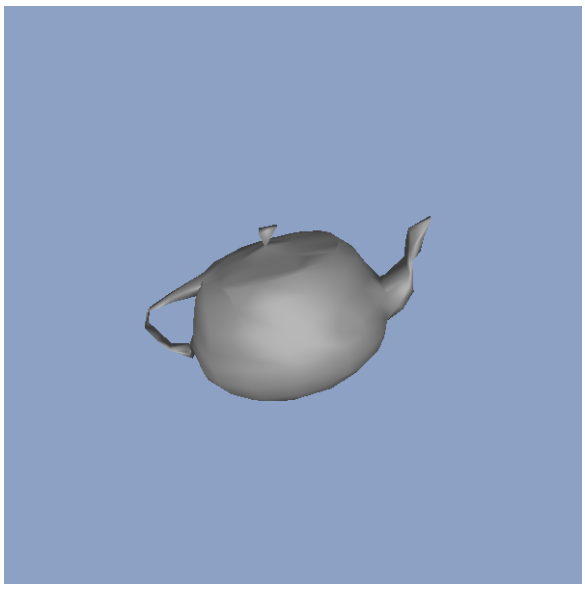

Figure 2: Application of the mean method to the `Teapot` model with a 10x10x10 grid.

4 Session 3: Error Quadrics and Shape-Preserving

4.1 Error quadrics

The process is divided in two steps:

- Pre-processing: first I create an array of matrices `QMatrixPerVert` of size $|V|$:

  $$QMPV_i = Pv * Pv^T : Pv = (n_i.x, n_i.y, n_i.z, -dotProd(n_i, vtx_i))$$

  This is performed previous to the LOD generation

- For each cell per grid, and sum all the matrices in the cell into an auxiliary matrix. Replace the last row of the matrix with (0,0,0,1), compute the inverse, if the matrix was not invertible I set the new vertex as the mean of the sum of all cell vertices, but otherwise I get the vector value (Qinv * (0,0,0,1)) and check if the value is in the bounds of the cell:
  - If that's the case, I push the vector coordinates back to the new vectors array.
  - Otherwise, set the new vertex as the mean of the sum of all cell vertices.

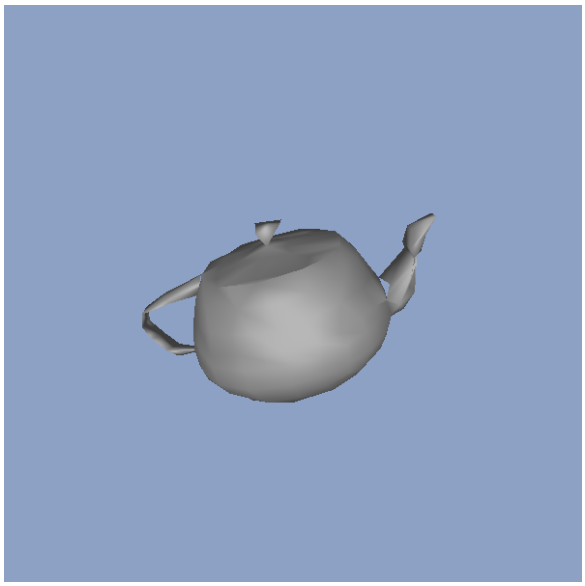

Figure 3: Application of the error quadrics method to the `Teapot` model with a 10x10x10 grid.

4.2 Shape-Preserving Algorithm

The process is divided in these steps:

- When I populate the grid, instead of sizing it with `level3D` rows [1], I resize it with 8*`level3D` rows. The 8 stands for each of the potential X,Y,Z-coordinate signs (2 signs and 3 axis $\rightarrow 2^3 = 8$ positions).
- I get the sign of a given normal as $sign = (\ n.x + 2n.y + 4n.z\ ) \in [0..7]$, n being the normal coordinates of the vector. Grids are now ordered first by the normal coordinates of each cell, and then every cell with the value `8*gridPos + sign` to set which "cell" I am looking at, so cells [0..8) contain the first cell, cells [8..16) belong to second one, and so on...
- I obtain the vertices per each cell I recycle the mean method, but for each of the 8*`level3D` rows. This way, I can pre-compute the faces without having to change its code.

Figure 4: Application of the shape preserving method to the `Teapot` model with a 10x10x10 grid.

[1] equivalent to LOD^3, and LOD are the divisions of the uniform grid per dimension

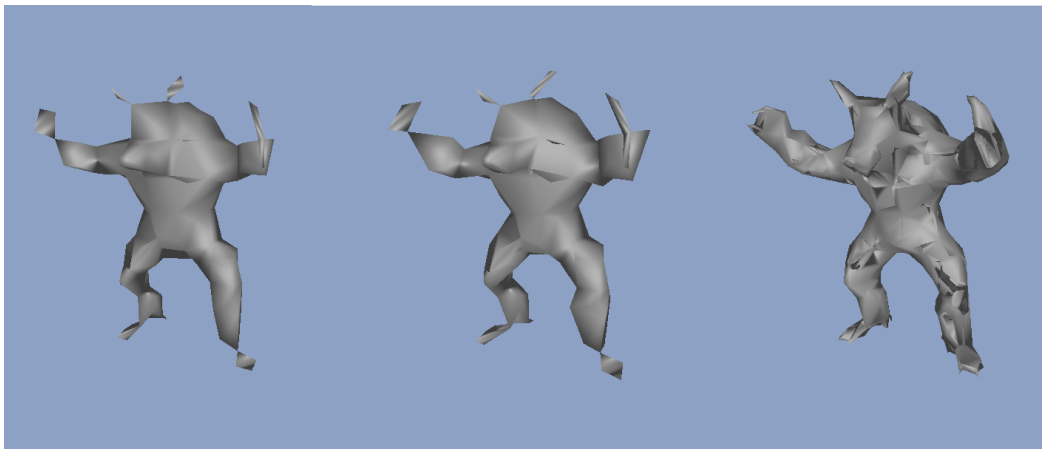

Figure 5: Comparison of methods with the `armadillo` model on a 8x8x8 grid.
From left to right: mean, error quadrics and shape preservation.

5 Extra: voxelization

Another simple metric for generating vertices has been added: voxelize creates each vertex at the center of the cell of each grid. This was made for testing purposes, but I chose to keep it for artistic reasons.

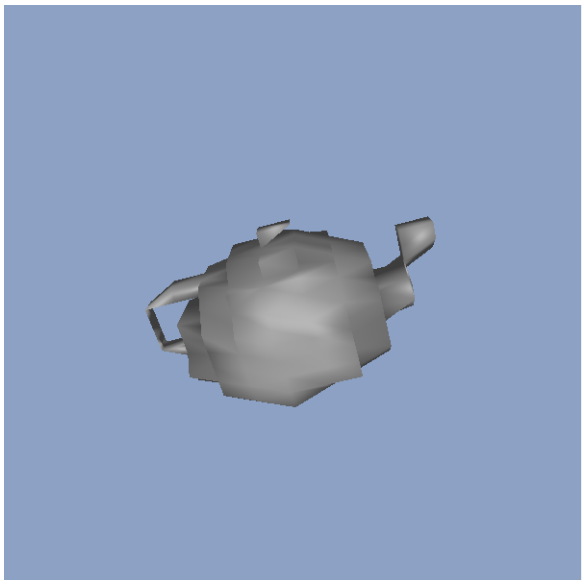

Figure 6: Application of the voxelization method to the `Teapot` model with a 10x10x10 grid.