SRGGE Project Report

J. Hugo Ballesteros

June 14, 2019

Contents

1 How to Build and Run the Program

This section describes how to build and run the application in a linux environment. It has not been tested in MacOS or Windows, nor is its intention to be run in those systems.

1.1 How to Build

Open a bash terminal at the folder *BaseCodeWithText* and run:

```
mkdir build && cd build
```

afterwards run:

```
cmake ..
```

and

```
make
```

and the binary will be built.

1.2 How to Run

After building the binary, run

```
./BaseCode /path/to/map /path/to/visibility
```

and the application will start.

If no map or visibility is provided, the application will default to parsing a map with more than 10 rooms.

An example run could be:

```
./BaseCode ../maps/3.map ../visibility/results/3.vis
```

2 Resolution of Labs

In this section the different lab sessions are explained and how the implementation for it was done.

2.1 Lab 1

This session started by loading a mesh and rendering $n \times n$ copies of that object. In the current code, it merely is the loading of said object and therefore there is no code to show for it. The calculation of the framerate is also pretty straightforward and done in *main.cpp*.

2.2 Lab 2

This session is one of the more fundamental parts of the whole application. I first start by reading a map that has the following syntax:

- 'w': wall
- ' ': floor
- 'b': bunny model
- 'd': dragon model
- 'a': armadillo model
- 'f': frog model
- 'o': moai model
- 'm': max planck model
- 'p': player position
- 'h': horse model
- 'l': lucy model

To run a variation of your map with a different configuration of walls or floors (i.e. position of meshes is irrelevant) then a new visibility test needs to be run on the map. See Section 6 for details on visibility implementation.

An example map could be for example:

```
wwwwww
w bb w
w    w
w  p w
wwwwww
```

Where a rectangular room with two bunny models would be rendered.

Note that no models are rotated, so there could be cases where the models are facing the walls. This could be solved with a different notation, but would require the storage of individual rotations in the map and therefore would be too costly.

This notation ensures that any number of rooms can be created. The file *3.map* has more than 10 rooms with several models.

Navigation on the application follows a First Person Shooter camera and movement. To move, press *W,S,A,D* to go forwards, backwards, left and right. To move the camera, click the screen and move the mouse around. Vertical movement by default is disabled, to enable it, see Section 6 for Visibility Mode.

2.3 Lab 3

In a first implementation, a uniform grid was used. But this session introduces the second developed advanced function: the octree. The octree follows a usual structure as such:

```
struct node {
  bool is_leaf;
  vector<node*> children;
  vector<Vertex> verts;
};
```

The octree is constructed with respect to the bounding box of the mesh, and so it maintains 3 different lengths for each axis. After the points in the tree, we can call the method *cluster* to get clusters of points of the level that is desired. If we call cluster with ”6” as its argument, then all the nodes in the 6th level of the tree will now be representatives of all its corresponding children. In this phase the average of the points was calculated, but in the next session we changed it to Quadric Error Metrics.

2.4 Lab 4

This session was very simple to implement. For each 3 indexes in the triangles vector that was read from the PLYReader, we can build a vector of Triangle structs, as so:

```
struct Triangle{
  int v1;
  int v2;
  int v3;
  bool is_calc;
  Eigen::Matrix4d K;
};
```

After this we can calculate the K matrices of each triangle and build a V:{F} dictionary in order to know the neighborhood of a vertex. With that each representative Q matrix is the sum of each of its vertices Q matrices, which themselves are a sum of the K matrices of the triangles in their neighborhood.

Using the formula of the original paper by Garland, we obtain the optimal vertex position and use that as the cluster representative, removing all degenerate triangles.

2.5 Lab 5

This session was developed after session 6, and therefore only takes into account the visible objects from the player's point of view (without view frustum culling). Every frame we start with every LOD level at 0 (the lowest level) and I calculate the value of raising the LOD level of an object as follows:

$$value(mesh) = \frac{b(mesh.lod+1) - b(mesh.lod)}{c(mesh.lod+1) - c(mesh.lod)} \; ,$$

where

$$b(level) = \frac{2^{level+5} * 1/distance}{diagonal_b box}$$

and

$$c(level) = size(mesh.level.triangles)$$

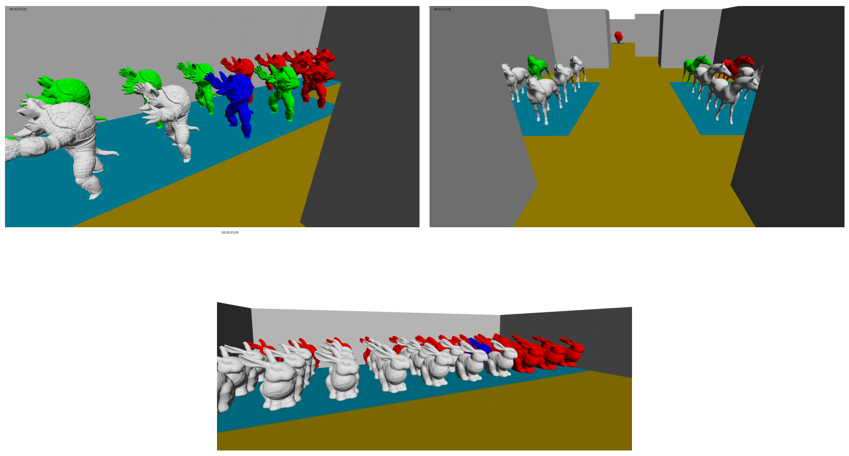
We add 5 to the level because it gives the clustering level of the octree that generated that level.

The triangles per second (TPS) of the machine could not be calculated coherently due to the fact that vertical sync caps the FPS at 60, making a large scene with a lot of triangles that the machine can handle have 60 fps and a scene with a single mesh have 60 fps as well. This makes that the TPS calculation based on past frames be capped when we move to a triangle light room.

To change the value of the TPS, go to line **38** of the file *TimeCritical-Rendering.cpp*.

The meshes in the application will have a solid color representing the LOD that is being rendered. The order of the colors is as follows:

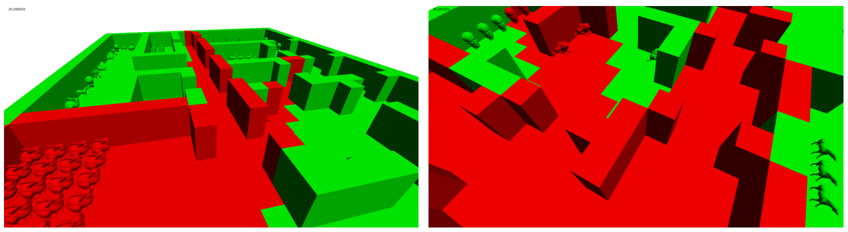
- Red: level 0 (lowest resolution)
- Green: level 1
- Blue: level 2
- White: original mesh



Different rooms with different LOD levels for each mesh.

2.6 Lab 6

This session can be found in the *visibility* folder, where we reuse the *Map* class to store the values of the map and calculate visibility for each cell. In my specific case I used a Bresenham Supercover, introducing the third advanced feature.



Visibility of at two different positions. Red is visible, Green not visible

You can toggle visibility mode by pressing V while the application is running. This will also enable vertical movement so that we can see what is visible and what isn't.
For each cell we create a set number of rays, and for each ray create a position inside the cell at random, and a random angle between 0 and 2π. Then we can calculate which cells that ray intersects using the Bresenham Supercover algorithm, stopping if we hit a wall or if the ray goes out of the bounds of the map. All the cells visited by the rays of the cell we are calculating visibility are the PVS of that cell, and are exactly the cells that will be rendered at the render call when the camera is on that cell.

To run the visibility program, go to the *visibility* folder. Once there:

```
mkdir build && cd build
```

afterwards run:

```
cmake ..
```

and

```
make
```

To run simply:

```
./vis /path/to/map /path/to/output
```

For example:

```
./vis ../../maps/3.map ../results/3.vis
```