Implementation Notes and Results of a simple Octree Implementation with View Frustum Culling

Simon Wallner[*]

April 2011

This report provides details on the implementation as well as results of a simple *octree* implementation used for *view frustum culling*. The implementation targets *Windows7 32bit* and was developed against an *ATI Radeon 4830* GPU using the *OpenGL 3.2 core profile*.

1 Motivation

In a typical scene less than 50% of the vertices in the scene contribute to the final rendering. In order to take load off the vertex engine and offloading it to the CPU, view frustum culling can be used. Objects outside the viewing frustum are culled and are not sent to the GPU for rendering.

To accelerate the identification of objects that can be culled a *octree* is used that discretises the scene into blocks on which the culling decision is performed.

2 Implementation

The octree and FVC (view frustum culling) has been integrated in an existing code base that was previously used for the *real time rendering lab course*. A video of the resulting demo can be viewed here: `http://vimeo.com/19202087`. A few details have been changed and other features have been added during this project.

[*]me@simonwallner.at

2.1 Mesh Representation

A *struct-of-arrays* structure is used as the internal mesh representation. The previously hard coded vertex attributes have been replaced with arbitrary vertex attributes and arbitrary 2d textures have been added. Vertex attribute semantics and texture semantics can be specified in the shader and are then combined into a `RenderMesh` for rendering.

2.2 Collada Model Loading

Limited support for loading Collada scenes is provided as long as they adhere to certain constraints (constraints on vertex attributes and textures, general materials and transformations are not supported). During loading the scene is converted into the internal canonical representation. A simple scene graph is used to hold meshes together and simplify rendering.

2.3 Octree

An octree is defined by its origin (relative to its own coordinate system) and a size. It covers a volume of $(2 \times size)^3$ units. The size of the octree must be chosen according to the meshes that are inserted into it. Larger meshes can be inserted but splitting and culling can yield unwanted results.

The octree is created by successively splitting the inserted mesh along the axis aligned faces of its sub cubes. The *Hesse normal form* and a simple clipping algorithm are used to split a mesh along a face. The input mesh is deleted after inserting in order to save memory.

Splitting operates on the vertex positions and supports arbitrary vertex attributes in the input mesh. Shaders and textures are assigned to the resulting meshes after the split.

Currently the insertion depth is controlled by a user defined depth parameter.

2.4 View Frustum Culling

To accelerate rendering view frustum culling (VFC) is used to cull unwanted geometry. VFC operates on the octree cells and performs a simple and conservative visibility test. The edges of a cube are projected into clip space and tested against each side of the viewing cube. A cube is declared invisible if all corner points lie outside of every side of the cube.

The octree is traversed top-down and invisible nodes are ignored. This results in a best case runtime of $\Theta(1)$ (fully invisible) and a worst case runtime of $O(\frac{8^{d+1}-1}{7})$ (fully visible), where d is the maximum recursion depth of the octree. In the worst case scenario the visibility of every sub cube has to be probed.

2.5 Performance

Performance figures have been captured in a single run on the development machine[1].

To test the performance a randomly generated star field has been generated and inserted into the octree. Each star consists of simple regular tetrahedron with 4 triangles and 12 vertices (4 unique vertices). The stars have no other vertex attributes other than position.

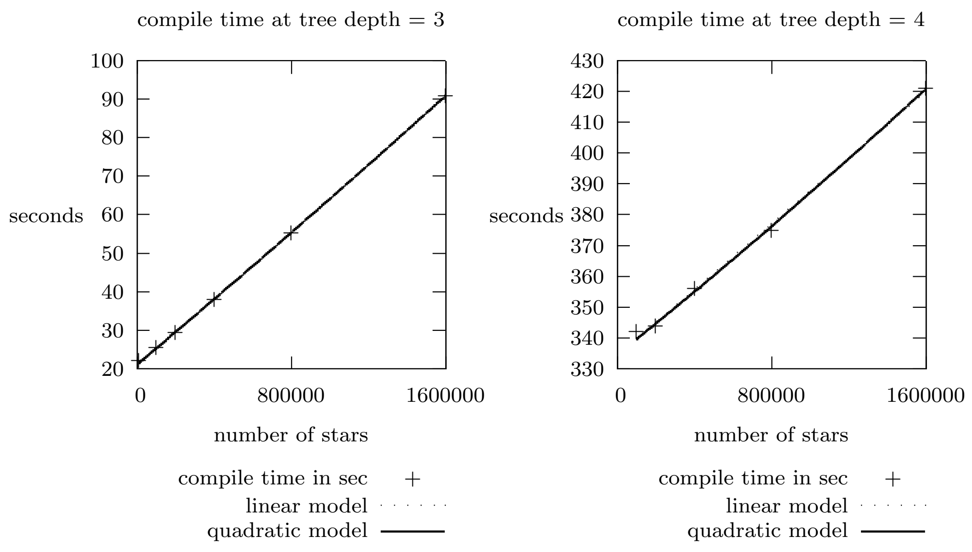

Figure 1: compile times of the octree

Figure 1 shows the compile time of the octree for depths of 3 and 4. From the diagram it can be seen that the time complexity is linear in the number of stars. With increasing insertion depth the time complexity rises sharply.

Figure 2 illustrates the frame time of the view frustum culling and the naive rendering. VFC average case[2] frame times are significantly lower than naive rendering frame times in most of the cases. VFC worst case[3] is roughly similar to the naive frame time for $d = 3$ but significantly higher for $d = 4$.

2.6 Discussion

Octree view frustum culling can greatly reduce frame time especially in cases where there is a lot of stress on the vertex engine of the GPU. Worst case performance however can be significantly higher than naive rendering.

All in all, octree fiew frustum culling is a useful optimization that should perform fine in standard situations.

[1]Win7 64bit, Intel E7400 dual core 2.8GHz, 4k ram, ATI Radeon 4830

[2]Camera centred approximately at the center of the octree.

[3]octree completely visible

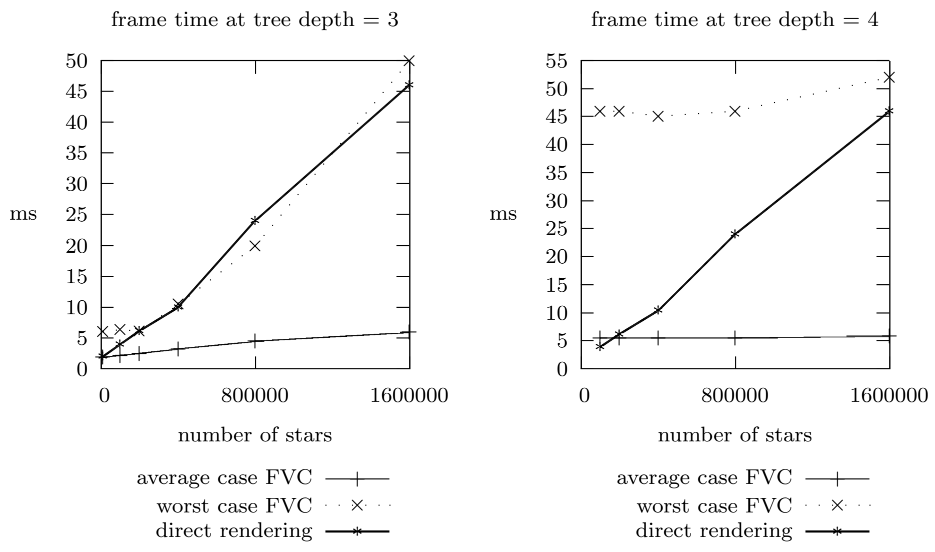

Figure 2: frame times of the octree

3 Future Work/Further improvements

Right now the presented implementation is in an early stage and has a few shortcomings:

- Model matrixes are not applied onto a mesh upon insertion into the tree.
- Insertion depth is either defined at the function call or decided at a per node basis. If a node exceeds a certain threshold it will be subdivided. It would be better to control insertion depth on a per mesh basis, in order to keep batch sizes reasonable.
- Each mesh in in node is drawn in it's own draw call. It would be better to batch these calls together to improve performance. This could be done by determining visibility in a first pass and then batched rendering in a second pass (to reduce draw calls and state changes).

4 Compiling and Running the Demo

All source files are hosted in a public repository at github[4] in the *city* branch. *CMake* is used as a build tool and it should compile with Visual Studio 2010. More detailed instructions on how to build it can be found in the readme file.

[4]`https://github.com/SimonWallner/kocmoc-demo`

4.1 Running

The application should start directly from the supplied package[5]. Application behaviour can be controlled via the `kocmoc.properties` file (resolution, fullscreen mode, debug output, ...).

The camera is either controlled via mouse and WASD or a gamepad (preferably an XBox 360 gamepad). Special function keys include

1 toggle octree rendering (stars may jump do to different random distribution)

2 toggle mesh AABB debug rendering

3 toggle octree debug rendering

4 toggle view frustum culling

F3 toggle wireframe mode

F5 toggle non-planar projection post processing effect

F6 toggle vignetting post processing effect

. take screen shot

5 License

All source code is licensed under the MIT license, all art assets are licensed under the creative commons 3.0 license. More detailed information can be found in the `license.txt` file in the repository.

[5]OpenAL might be required, the redist installer is included in the package