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 taking load off the vertex engine in the GPU 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 discretizes 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 and texture semantics can 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 $(2 \times \mathit{size})^3$ 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.

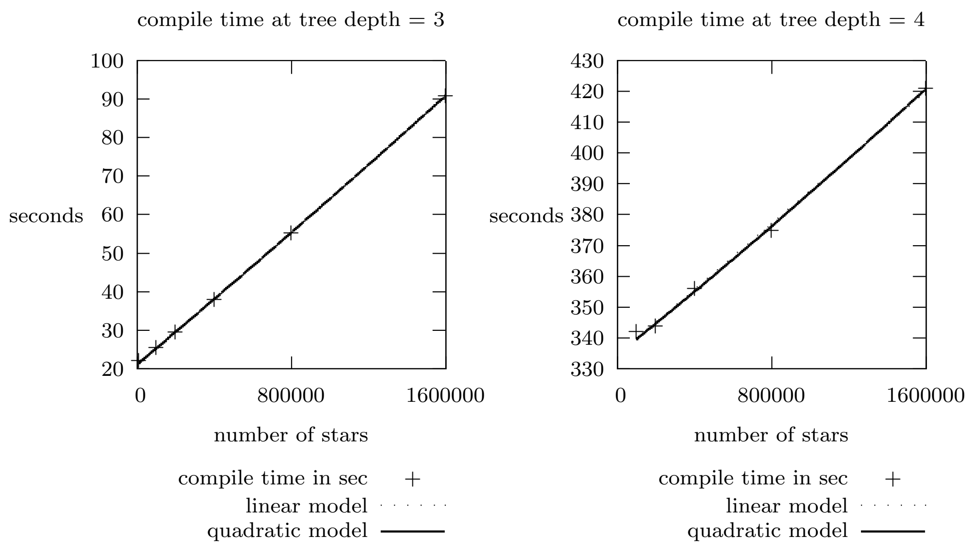

Figure 1: compile times of the octree

2.5 Performance

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

3 Sources

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

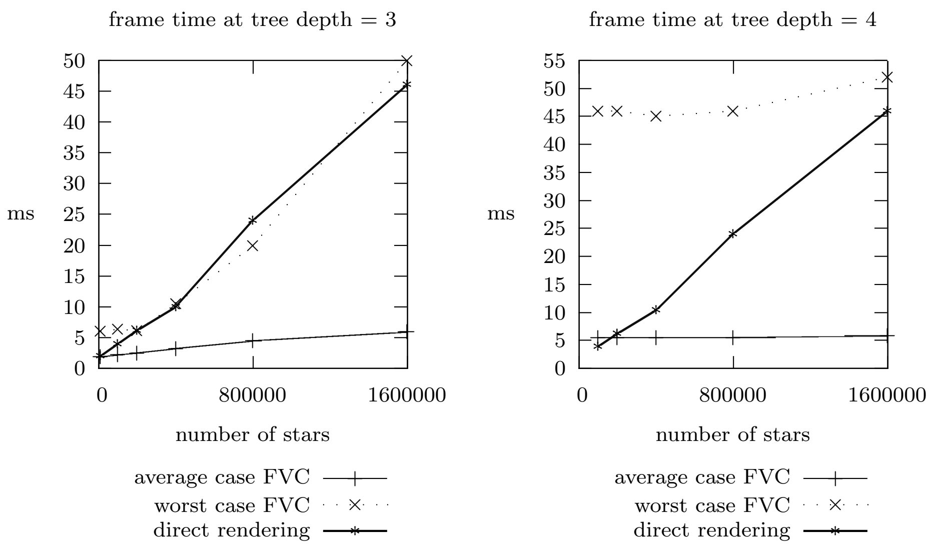

Figure 2: frame times of the octree