**Rasterizer documentation**

**Introduction**

For the Prague 2015 project, the template has been extended with a bare-bones software rasterizer. The purpose of this software is to provide a compact and transparent starting point for 3D game development. The software rasterizer implements a basic scene graph, 3D model loading, and accurate rasterization with perspective correct texture mapping, basic shading and clipping.

**Scene graph**

The 3D scene is stored in a simple scene graph. Two node types are available: the first is a mesh-less transform node, the second is a mesh node (which may also contain a local transform). The matrices of the scene graph nodes can be modified using a number of convenience functions that affect translation and/or rotation; see rasterizer.h for details.

**Mesh, texture, material**

The scene is organized in a way that suits state-driven rendering: each mesh can have one material. It is therefore straightforward to render the scene using OpenGL.

**Data access & ownership**

All data in the rasterizer classes is minimal, and ‘public’. This allows for easy manipulation and querying, e.g. when adding ray tracing to the renderer.

The Scene class maintains a list of loaded materials and textures, and serves as the ‘owner’ of these objects. This allows meshes to share materials and textures. Upon deletion of the scene, all data will be correctly disposed of.

**Setting up the engine**

To use the rasterizer in the template, first include its header file:

#include “rasterizer.h”

Next, instantiate the rasterizer at global scope along with a camera, or add it to the game class:

Rasterizer rasterizer;  
Camera camera;

Then, in the init function, initialize the rasterizer, and load some geometry:

rasterizer.Init();  
rasterizer.scene->Add( “assets/unity/unityScene.obj” );

Finally, render the scene in the Tick method:

rasterizer.Render( camera );

This is the bare minimum; from there it’s up to your imagination.

Happy coding,

Jacco Bikker.