Stellarium: Scenery3d Developer Docs

Table of Contents

Versions

- 2011-01-27 First issue of this document

Introduction

Scenery3d is a plug-in for the open-source software Stellarium. The main functionality is to render 3D meshes in front of the original Stellarium scene, which is composed of a real time sky rendering with a static landscape skybox.

This document should give a quick start for new developers.

Working with Stellarium

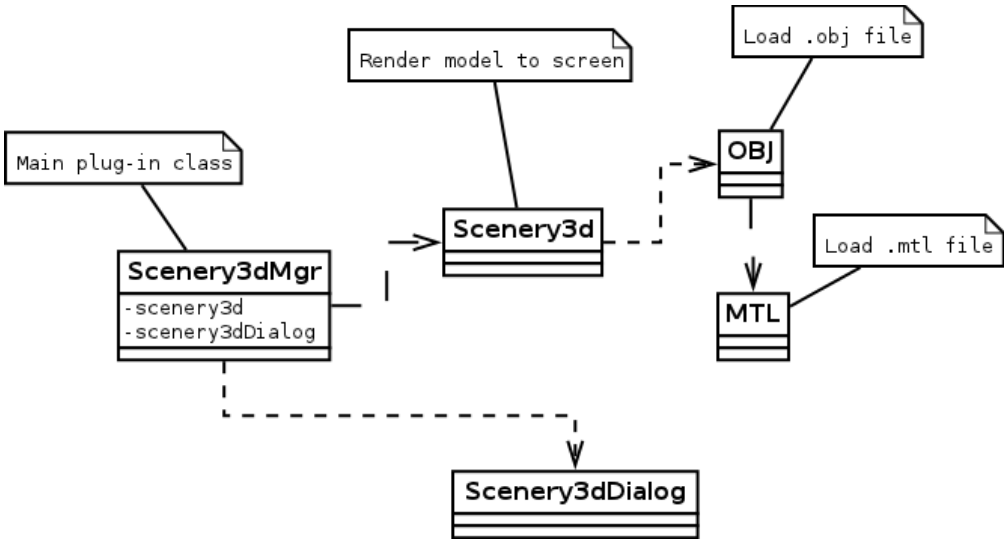
As this plug-in is only part of a bigger project, here are a few pointers that should give you a better understanding of the general Stellarium architecture.

- General information, program architecture
  http://www.stellarium.org/doc/head/index.html
- Coding conventions
  http://www.stellarium.org/doc/head/codingStyle.html
- StelPainter API reference, we use this class to draw vertex arrays
  http://www.stellarium.org/doc/head/classStelPainter.html

Class structure

- **`Scenery3dMgr`** is the main plug-in class. It is exported as a plug-in module via the `StelPluginInterface` interface, implemented by `Scenery3dStelPluginInterface`**.** It creates one instance of `Scenery3d` for the actual scenery rendering and one instance of `Scenery3dDialog` for the GUI.
- **`Scenery3dDialog`** is for displaying the configuration GUI window.
- **`Scenery3d`** is for loading, updating and rendering the actual 3D scenery. This uses various helper classes for loading model files.
- **`OBJ`** represents an .obj model file.

In a nutshell:



Filesystem structure

Loading a scenery is similar to loading a landscape. Each scenery is represented by a directory. `Scenery3dMgr` generates a directory listing using `StelFileMgr::listContents`. The code is in `Scenery3dMgr::getNameToDirMap`. Basically, it looks for directories in `modules/scenery3d/`, relative to the Stellarium main directory as well as the settings folder in the user's home directory, e.g. `~/.stellarium/` on Linux systems.

Each scene must have a `scenery3d.ini` file in the scene's root directory. All file paths referenced in scenery3d.ini are treated as relative to the scene's directory. For more details about the config file format, look at `STELLARIUM-HOWTO.txt` in plugins/Scenery3d/doc/.

Loading model files

Currently we support loading Wavefront .obj files. The class `OBJ` is responsible for this. For more details about the file format, visit http://en.wikipedia.org/wiki/Wavefront_.obj_file.

The .obj format specifies vertices, normals and texture coordinates. Please note that although theoretically you can specify arbitrary polygon faces in an .obj file, we only support loading triangles. Various `asserts` in the program make sure that each face has only three vertices.

The usual way of loading models is to read a scenery configuration using `Scenery3d::loadConfig` and then call `Scenery3d::loadModel`. This not only loads both the scenery and ground models, but also rotates each vertex by the value specified through `rot_z` in `scenery3d.ini` to account for a difference between geographical north and grid north.

For rendering, we convert the data to arrays of `Vec3d` for vertices, `Vec3f` for normals and `Vec2f` for texture coordinates. This is done by calling `OBJ::getStelArrays`, which returns a list of structs containing the arrays for each mesh contained in the .obj file. The resulting data can then be passed to `StelPainter::setArrays` or a similar function to directly draw the meshes.

Rendering methods

Stellarium supports various projection methods for displaying the sky. As of version 0.10.6 this includes Perspective, Equal Area, Stereographic, Fish-eye, Hammer-Aitoff, Cylinder, Mercator and Orthographic projection. With the exception of Perspective, and Orthographic all of those are non-linear and cannot properly be represented by an OpenGL projection matrix.

Direct rendering would be problematic, especially for the nonlinear projection where triangle edges can be contorted, which would require more advanced methods like polygon subdivision and z-buffer correction.

In order to match the 3D scene to the existing sky and landscape rendering, we render the scene to a cubemap using perspective 90° FOV projection for each of the six cube faces. The cube is then distorted by rendering using the `StelPainter::drawSphericalTriangles` method.

In order for this to work, the cube model's faces need to be subdivided into smaller triangles. The cube is generated in the constructor of `Scenery3d`.

Currently, we have two rendering paths: Scenery3d::drawObjModel draws the model using a perspective projection matrix with the FOV configured in Stellarium. This obviously only works for Perspective projection mode.

The second rendering path consists of `Scenery3d::generateCubeMap`, which renders the

scene into six different textures using framebuffer objects, and `Scenery3d::drawFromCubeMap` which renders the tesselated cube as a skybox using the textures generated before. This rendering path is used for all projection modes beside Perspective.

Collision Detection

The class `Heightmap` represents a data structure to accelerate height queries. An instance of Heightmap is initialized with a reference to an `OBJ` instance. Note that in Stellarium, the z-axis represents the height, while the x-y-plane is parallel to the ground.

The triangles of all meshes are organized in a regular grid in the initialization process. Each grid cell stores a list of all triangles that intersect with the grid cell. This is currently done using a simple bounding-box (AABB) test. While this may introduce some false positives, it is generally faster and simpler than similar approeaches.

The method `Heightmap::getHeight` then returns the z (height) value for a given point in the x-y-plane. For this, only the triangles registered in the current grid cell are considered, which should speed up the process considerably.

Shadow Mapping

As of 2011-01-27, we have Shadow Mapping as an experimental feature integrated in the Scenery3d plug-in. This uses the method described in http://www.paulsprojects.net/tutorials/smt/smt.html to render shadows.

It uses three rendering passes. One to generate a shadow map, which is a depth texture rendered from the light's point of view. This happens in Scenery3d::generateShadowMap.

The second pass renders the scene with dimmed lights, while the third pass renders the scene in light using texture comparison in combination with alpha test to render only the parts not covered by shadows over the dimmed scene.

Currently, this doesn't work too well. On one hand, there is the limited resolution of the shadow map, on the other hand we had to disable textures when the shadow map is activated as enabling using multitexturing has caused problems with the rest of Stellarium's rendering process.