Stellarium: Scenery3d Developer Docs

Table of Contents

Versions

- 2011-01-27 First issue of this document
- 2015-02-24 Big rewrite to document current state

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.

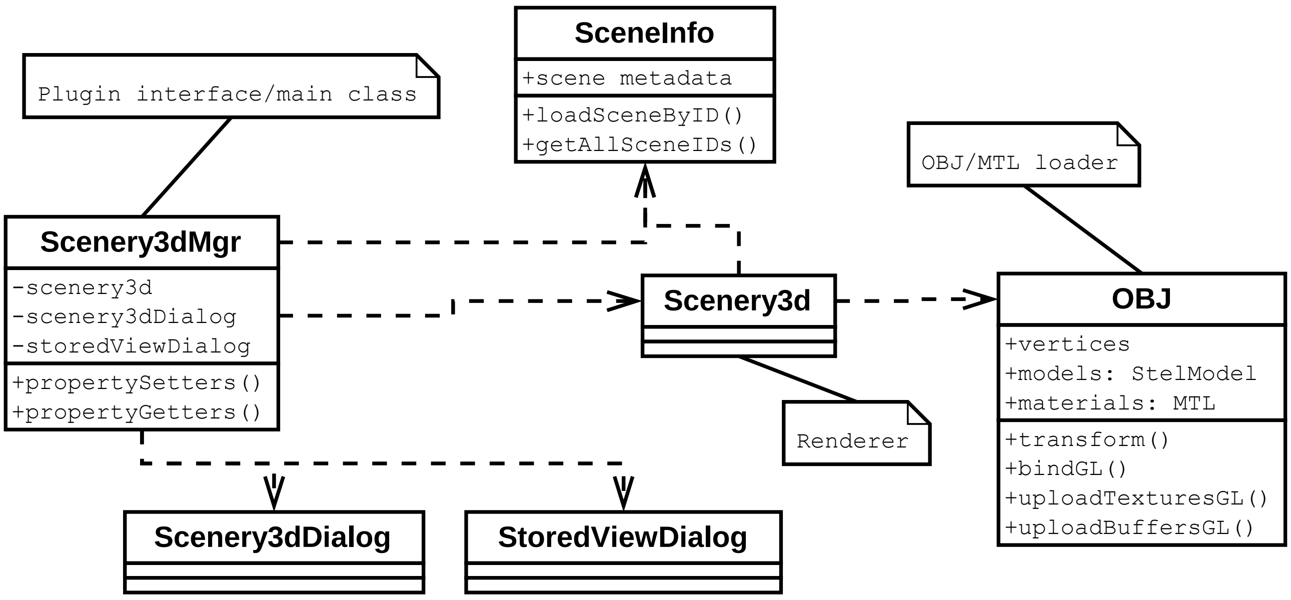
- General information, program architecture
  http://www.stellarium.org/doc/head/index.html
- Coding conventions
  http://www.stellarium.org/doc/head/codingStyle.html

Class structure

- **`Scenery3dMgr`** is the main plug-in class. It is defined as a `QObject` (a `StelModule`) exported as a plug-in module via the `StelPluginInterface`, implemented by `Scenery3dStelPluginInterface`**.** It is responsible for handling user actions through `StelAction` objects, and by creating a few toolbar buttons and exposing some properties as `Q_PROPERTY` that can be changed by the dialog classes. It holds one instance of `Scenery3d` for the actual scenery rendering and instances of `Scenery3dDialog` and `StoredViewDialog` for the GUI.
- **`Scenery3dDialog`** is for displaying the configuration GUI window. It contains all performance/quality options exposed by Scenery3d, and lists the sceneries that are available for loading. It is constructed using the `StelDialog` class and the standard Qt .ui files.
- **`StoredViewDialog`** manages the stored viewpoints of a scene, allowing the user to change to and create predefined views of a scene.
- **`SceneInfo`** describes the metadata of a scene, that is stored in the `scenery3d.ini` file for each scene. It is primarily a plain struct, but contains some static functions for listing scenes and loading metadata from files.
- **`StoredView`** defines a persistent view definition, containing some functions for saving and loading.
- **`Scenery3d`** is the main scene rendering class. It is not a `QObject`. It manages loading, updating and rendering the actual 3D scenery using OpenGL. This uses various helper classes.
- **`ShaderMgr`** is a basic manager for OpenGL shaders. It can load shaders from files defined by a series of bitflags defined by the current rendering state (enabled features such as shadows, bump mapping etc.) and material to render (specularity, transparency...), setting `#define` statements in the shaders to true or false depending on these flags to allow for conditional compilation, allowing for the most performant shaders without depending on branching using shader uniform variables. It caches these shaders, and also accelerates accessing the uniform variables of the shaders by providing access through an enumeration instead of strings.

- **`OBJ`** is an file loader for the simple text-based Wavefront .obj 3D geometry file format. Instances of `OBJ` contain the loaded data of a file, and provide additional functions such as bounding-box calculation, CPU-side (constant) transformations, OpenGL buffer management for their data and material/texture management. The `OBJ` header also contains the definitions for the vertex format (Position, Normals, Texture coordinates, Tangents) and the material format.
- Various other classes, mathematical helpers (AABB, Frustum, Plane, Polyhedron...), Utilities...

In a nutshell:



Filesystem structure

Loading a scenery is similar to loading a landscape. Each scenery is represented by a directory. Available scenes can be found using `SceneInfo::getAllSceneIDs`, which works with the `StelFileMgr`, listing scenes installed under the program directory, and custom scenes in the user's home directory. Basically, it looks for directories in the `scenery3d/` folder relative to the Stellarium main directory as well as the settings folder in the user's home directory, e.g. `~/.stellarium/` on Linux systems. Scene metadata can be loaded using `SceneInfo::loadByID`.

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 the user's documentation in `Scenery3d.pdf`.

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. For best results, the .obj should only contain triangulated geometry. Materials are loaded from the .mtl files referenced in the .obj, for more details on material parameters please consult the user's documentation.

The scene loading started by the user is initiated in `Scenery3dMgr`, which starts an asynchronous loading process to avoid locking up the GUI while loading. When the `SceneInfo` is available, the "real" loading begins in `Scenery3d::loadScene`, which loads the `OBJ` files for the scene and the ground (for collision detection), calculating bounding boxes and transforms each vertex by the value specified through `rot_z` in `scenery3d.ini` to account for a difference between geographical north and grid north. When has been done, `Scenery3d::finalizeLoading` is called in the main thread, uploading the data to OpenGL (using vertex buffers and vertex array objects for the geometry).

The OBJ files are currently loaded in a 2-pass process, which could still be optimized. The loader uses "oldschool" C-IO (fscanf, etc.) extensively, and partly in an unsafe way, and is therefore not very robust against invalid OBJ files and may even present a security risk (buffer overflows) for downloaded scenes. A rewrite using safer Qt IO classes is recommended.

OBJ objects are grouped by their material for better rendering performance and described by `StelModel` objects, but this is currently only possible for objects which follow directly after each other in the OBJ file (i.e. when there are 4 models defined in the OBJ directly after each other with materials 1-1-2-1, 3 StelModels are created, instead of optimal 2 which would require some reordering of the data). Less StelModels lead to a better rendering performance (reduced draw calls, material/shader changes, etc). For best results, the OBJ files should be pre-optimized by grouping all geometry with the same material together (as an OBJ object 'o' or a polygon group 'g').

For transparent objects, the opposite holds – each separate transparent object should be separated in the OBJ file. This ensures correct Z-ordering, which uses the centroid position of each transparent StelModel to determine the order in which to render them.

For rendering, one can simply call `OBJ::bindGL`, which binds the OpenGL buffers/VAO of the model making it ready for drawing. In conjunction with the `ShaderManager`, a consistent vertex/attribute layout is used that avoids having to specify the `glVertexAttribPointer` differently for each shader.

Rendering methods

Older versions of Scenery3d used the `StelPainter` for rendering. This was not an optimal solution, because it required the geometry to be re-uploaded for each draw call (which is bad because our geometry does never change, and potentially consists of a massive amount of triangles, compared to most other geometry used in Stellarium), supports only double-precision (which is unnecessary for our 3D sceneries in almost all cases, because we do not need the accuracy that the astronomical rendering, which potentially has to deal with very large size/distance differences, uses), and supports only a very basic shading model. Because of these reasons, the current version is completely independent of the `StelPainter` (unless for text/debug rendering), providing instead a completely shader-based renderer system. It also no longer uses the fixed-function OpenGL pipeline. Most functionality should be supported on OpenGL 2.1 – level hardware, basic rendering with vertex-based lighting also on OpenGL 2.0. Geometry shader functionality requires OpenGL 3.2. There are some adaptations for OpenGL ES 2.0 support, but this is currently untested.

Stellarium supports various projection methods for displaying the sky. As of version 0.13.3 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.

For this reason, there are 2 basic rendering paths. When Stellarium uses perspective projection, we can use the tradional direct way of OpenGL rendering with a perspective projection matrix. This is done in `Scenery3d::drawDirect`. This requires only 1 rendering pass (unless shadows are used), and the rendering is performed onto the existing contents of the back buffer.

When Stellarium is set to any other projection, the scene rendering is performed in 2 phases, done in `Scenery3d::drawWithCubeMap`. First, we render the scene to a cubemap (using framebuffer objects, `Scenery3d::generateCubeMap`) using perspective 90° FOV projection for each of the six cube faces. Then, we render this cubemap onto a tesselated/subdivided cube that has been projected on the CPU-side using the current projection (`Scenery3d::drawFromCubeMap`). Currently, a 20x20 subdivision is used for each face, resulting in a total of 2646 vertices that have to be projected by the CPU each frame. This is still reasonably fast, while making the subdivision almost unnoticable (the effect is a bit worse when zoomed in and the camera moves). Of course, the whole rendering process is quite a bit slower than the direct rendering setup, because the scene must be rendered 6 times + the cube transformation/rendering also takes a bit of time.

For this reason, some optimizations have been implemented. Firstly, one can change the way the cubemap is generated. The most basic method represents the cubemap by 6 textures (`GL_TEXTURE_2D` objects), this seems to be the fastest way on some integrated Intel GPUs. The second method uses a `GL_TEXTURE_CUBEMAP` object, which seems to be faster on reasonably modern GPUs, and also avoids a possible seam at the edges when rendering the cube. The final method uses OpenGL geometry shaders to render all 6 sides of the cube in a single pass. This may seem like it is the best method, but it depends on the hardware and on the scene if this is true. Especially larger scenes have a risk to be slower than the other methods, probably because of increased GPU memory usage. The initialization in `Scenery3d::initCubemapping` creates the correct OpenGL objects and calculates the cube vertices.

Another optimization avoids performing the slowest step of this rendering method, the creation of the cubemap, in each frame. The user can set an interval (in Stellarium simulation time, not real time), in which the cubemap is not updated. If the time is paused, and the user does not move in the scene (he can still rotate the camera without causing a redraw), the cubemap is never updated again because nothing in the scene changed. Only the cube rendering is performed, using the “static” cubemap.

When the user moves (translation), the cubemap has to be recreated. There is also an optional optimization that only causes 1 or 2 sides of the cubemap to be updated, depending on the viewing direction. This also reduces the cost of rendering.

Collision Detection

The class `Heightmap` represents a data structure to accelerate height queries. Note that this is not a "real" heightmap in the sense of a 2D height texture or similar. An instance of Heightmap is initialized with a reference to an `OBJ` instance. This can be a separate ground model or the scene's model, depending on the `scenery3d.ini`. 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.

This class currently uses a fixed-size grid for all scenes. It works fine, but the performance for initializing the grid and finding the height of a point is bad in larger scenes, and possibly even slower as the whole other rendering process together. For testing of the rendering system performance, the height test should be disabled completely by setting `ground=NULL` in the .ini. A more performant collision detection should be possible by using AABB hierarchies or quadtrees. If required, a general collision detection engine like the open-source Bullet (http://bulletphysics.org) could also be considered.

For best results, the ground OBJ should be optimized, using the minimal amount of triangles to achieve the desired results. A simple optimization is to leave out all vertical structures, because they would have no impact anyway.

Shadow Mapping

The plugin uses an implementation of cascaded shadow mapping (MSDN, Nvidia), originally integrated by Andrei Borza (Thesis).

The view frustum is adapted to the scene's bounding box, and split into 4 subfrusta (`adjustFrustum`). A polyhedron that encompasses the frustum and the relevant shadow casters is calculated for each split (`computePolyhedron`), an orthographic light projection that fits this polyhedron is calculated (`computeCropMatrix`, fit-to-cascade as described in MSDN), and a shadow/depth map is rendered (`renderShadowMaps`) for each split.

For rendering, shadow coordinates are calculated per-vertex, and the cascade to be used is determined using the distance of the fragment in window space (interval-based selection in MSDN). Shadow lookups use the hardware-accelerated `sampler2DShadow`. Shadow filtering is optional, allowing to choose between a combination of OpenGL hardware shadow filtering (fast, but performs filtering constant in UV space, which makes the penumbra size depend on shadow map size and projection – which can vary with camera view), and custom PCF filtering with a 16 or 64-tap poisson disk filter (filter size constant in view space, i.e. always the same penumbra size).

There is also an optional implementation of percentage-closer soft shadows (PCSS, http://developer.download.nvidia.com/shaderlibrary/docs/shadow_PCSS.pdf), which varies the penumbra size depending on the distance between shadow "blocker" and receiver, making shadows sharp near contact points, and blurred further away.

For performance, the user can also disable CSM, so that only 1 shadow cascade is used. There is also the problem of the cubemapping mode. Because the shadowmaps are fit to a perspective view frustum, they have to be recreated for each cubemap side separately for correct shadows. A simplification allows to assume perspective projection for fitting the shadow maps, when the FOV is not too large this is alright, but if it is shadows will be missing. This only requires the usual 4 shadow passes (compared with a total of 6x5=30 passes in the full cubemap case with 4 cascades).