Forest Scene Demo Developer Manual

Paul Guerrero

September 24, 2006

1 Compiling the Demo

The Forest Scene Demo is an Ogre (www.Ogre3d.org) application interactively rendering a Forest Scene.

The source code of the Forest Scene Demo is divided into two parts:

- The first part consists of the actual program which constructs the Forest Scene from a text file, handles user input, grows trees, handles collisions, etc.
- The second part is a custom Ogre scene manager (called ProxySceneManager) which handles proxies, static geometry and LOD blending in the scene. It is used as a library by the main program (first part). This custom scene manager can handle all kinds of scenes, not only forest scenes.

The main program and the custom scene manager both have a Visual Studio 2005 project file. The file for the main Program is located at `Landscape/Landscape.sln`. The scene manager is embedded in a slightly modified version of Ogre: `ogreDagon/Ogre_vc8.sln`. Before compiling, check if you have the DirectX SDK installed and configured in Visual Studio 2005 (under Tools—Options—Projects—VC++ Directories). If the DirectX SDK is installed, it should be enough to open these files and compile the applications in 'Release' configuration.

1.1 Library Dependencies

All needed libraries except the DirectX SDK should be included in the distribution, but here is a list of dependencies:

The Forest Scene Demo was compiled using a slightly modified version of Ogre 1.4 (Dagon) and you will need this version of the Ogre libraries (and the standard Ogre dependencies) to compile the Demo. The modified version (including the source files) is included in the distribution. To get the unmodified Ogre libraries including dependencies visit `www.ogre3d.org`. You can modify the standard Ogre library yourself by changing the lines as described in the included `OgreMain/OgreMainChanges.txt` file [1].

The main program additionally needs Ogre Opcode, a collision detection library for Ogre available at
`http://ogreconglo.sourceforge.net/index.php/OgreOpcode_Download_Install`

2 Main Program

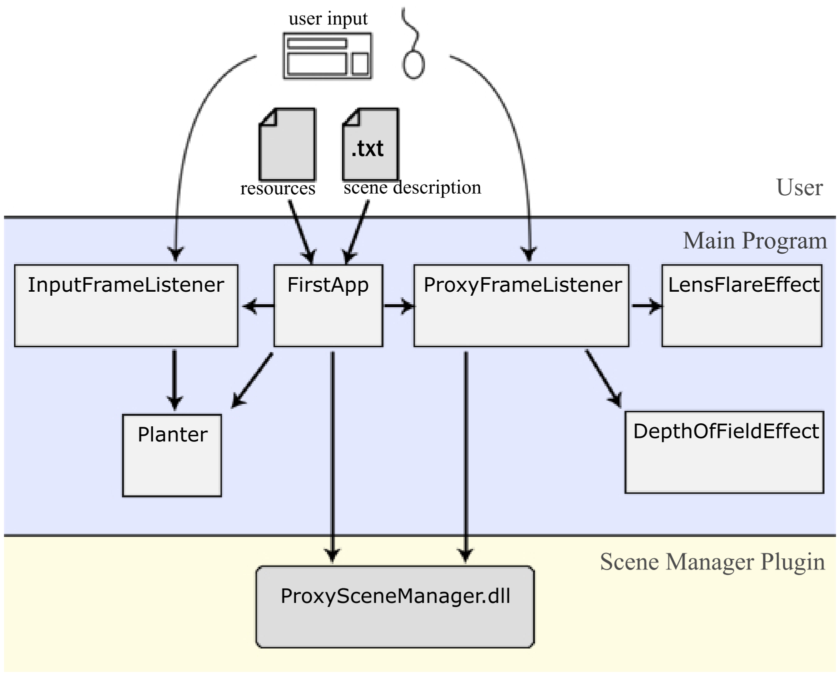

Figure 1: Overview over the classes in the main program.

An overview of the classes in the main program and their interactions is given in Figure 1.

For the remainder of the manual I will assume that the reader is familiar with the main Ogre classes and terms, for more information please read the Ogre documents available in the distribution (OgreDagon/Docs) or on the Ogre website (www.ogre3d.org).

The class `FirstApp` loads resources and constructs the scene using a scene description or scene generation text file (for more information on these files see the user manual). To add objects to the scene it communicates with the

[1]Only 5 lines were changed to grant subclasses more access to the functions of their base classes

`ProxySceneManager` class, a custom scene manager which is available in the `ProxySceneManager.dll` library.

Input handling is done with two frame listeners, the `InputFrameListener`, which handles movement, collisions, the sun animation and tree growth and the `ProxyFrameListener`, which handles Lens Flare, the Depth of Field effect and the ProxySceneManager functionality (Static Geometry and Proxies). In contrast to the `InputFrameListener`, the `ProxyFrameListener` is designed to work with any type of scene using the `ProxySceneManager` and should be easily reusable in other Ogre applications.

The `DepthOfFieldEffect`[2] and `LensFlareEffect` classes both use the Ogre Compositor framework to add their effect in a post-processing pass. They rely heavily on vertex and fragment shaders and while there exist both a HLSL (DirectX) and GLSL (OpenGL) version of the `DepthOfFieldEffect` shaders, the `LensFlareEffect` shaders are only available in HLSL (DirectX) (although it is possible to rewrite them in GLSL).

[2]created by Christian Lindequist Larsen

3 Proxy Scene Manager Plugin

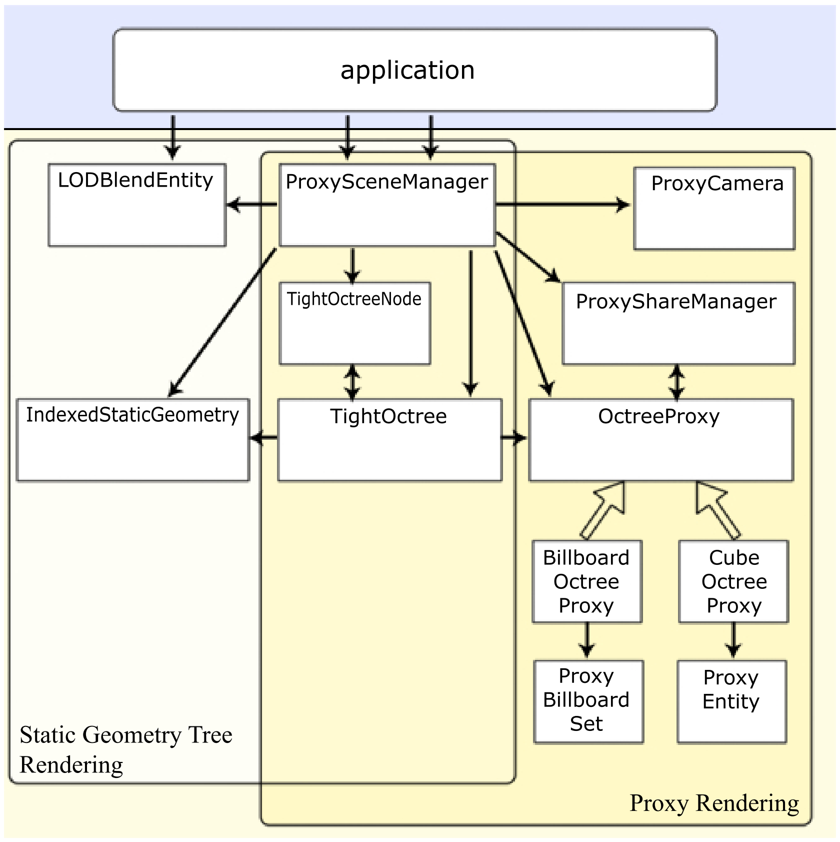

Figure 2: Overview over the classes in the proxy scene manager plugin. The classes used for static geometry tree rendering are on the left, the classes used for proxy rendering on the right. The hollow arrows depict an inheritance relationship.

The Proxy Scene Manager handles static geometry and proxies in a scene. It provides functionality to generate static geometry and proxies, special render modes for proxies / static geometry and methods to generate specialized Entities, Cameras and Nodes. To generate Proxies or Static Geometry for some nodes in a scene you would typically first mark these nodes (`markNode` function), then call the static geometry or proxy generation function (`generateProxies` or `generateStaticGeometryTree`) and finally switch to the appropriate render mode (`setUseImpostors` or `setUseStaticGeom`). A more detailed overview of these functions is given in appendix A.

The Proxy scene manager is implemented as an Ogre plugin (like the standard octree scene manager) and has to be added to `Plugins.cfg` file if it is to be used in an application.

Only the functions of the `ProxySceneManager` and `LODBlendEntity` classes are exported in the library. These functions form the interface to other applications using the Proxy scene manager plugin. A quick overview of the most important functions is given in appendix A.

3.1 The `ProxySceneManager` class

The `ProxySceneManager` is derived from the standard Ogre `OctreeSceneManager`. It creates and manages all other classes in the plugin. The most important additions to the `OctreeSceneManager` functionality are the following:

- Create and manage `LODBlendEntity` objects.
- Extend the loose octree to act as a tight octree.
- Generate Proxies or Static Geometry.
- Functions to render Proxies or Static Geometry (see section 3.10 for more details).

This class also forms the interface to applications that want to use the ProxySceneManager plugin. For an overview of the most important functions see appendix A.

3.2 The `TightOctree` class

The octree scene manager implementation in Ogre uses a loose Octree to accelerate visibility culling. Scene nodes are assigned to exactly one octant (based on position and bounding box extent). When generating Proxies, the standard loose octree has to be adapted to act as a tight octree (for more information on loose and tight octrees see the user manual and the report). For this purpose, the `TightOctree` extends the standard `Octree` class to store the nodes that are only partially inside the tight volume of an octants. If a proxy was created for an octant, a reference to it is also stored in the `TightOctree`. Most of the construction work for the static geometry tree is also done inside the `TightOctree` class and each `TightOctree` stores references to the static geometry region corresponding to its volume.

3.3 The `OctreeProxy`, `BillboardOctreeProxy` and `CubeOctreeProxy` classes

An `OctreeProxy` is a special `SceneNode` (although it is never attached to the scene graph). Each `OctreeProxy` represents a simplified version of objects inside the volume of an octant. Two geometric primitives can be used as a coarse approximation for the objects to be represented; either billboards -

BillboardOctreeProxy - or cubes[3] - CubeOctreeProxy. Both classes are derived from OctreeProxy. Billboards use the ProxyBillboardSet class and Cubes the ProxyEntity class for their representation. These classes are just slimmer versions of the standard BillboardSet and Entity classes. The OctreeProxy class renders an octant from six sides and stores one colour for each side (in coordination with the ProxyShareManager, see section 3.4). The resulting colours are then used by the CubeOctreeProxy or BillboardOctreeProxy to build the proxy.

3.4 The ProxyShareManager class

The ProxyShareManager coordinates the use of shared materials, meshes and textures by the proxies. Proxies are created by the ProxySceneManager and have to be registered in the ProxyShareManager (using the addProxy function). There they are queued until enough proxies have been added or the scene manager notifies that it is done adding proxies (by using the notifyDone function). The ProxyShareManager also contains functionality to store/load proxies (using the ProxyRepository class).

3.5 The ProxyCamera class

The ProxyCamera is a simple OctreeCamera that can calculate the screen area (in pixels) of an AxisAlignedBox (the getScreenArea function).

3.6 The ProxyBillboardSet and ProxyEntity classes

Both of these classes trade some of their functionality for a smaller memory footprint. In their standard Ogre implementation, these classes contain functions that are not needed for proxies.

3.7 The TightOctreeNode class

The TightOctreeNode is derived from the standard Ogre OctreeNode. Each of these nodes can be marked (by calling setProxyless(false)). Only marked nodes are included in proxies or static geometry.

A TightOctreeNode can also be marked as handled (setHandled(true)). This is necessary to avoid adding a node twice when traversing the octree. The handled mark is reset at the end of each frame.

3.8 The IndexedStaticGeometry class

This class is a modified version of the standard Ogre StaticGeometry class. It allows indexing of custom regions inside the static geometry called GeometryBucketLists. To create a GeometryBucketList, first create an empty RegionSubset. The RegionSubset will store the location of your custom region inside the static

[3]for more detail on these representations see the user manual or the report

geometry. Now add all the objects you want in your custom region to the static geometry passing the `RegionSubset` as parameter. Each time you add an object, the `RegionSubset` will be updated. The `RegionSubset` now contains a list of index buffers intervals, corresponding to the locations of the objects you just added to the static geometry. You can modify these index Buffer intervals as you like, as long as they still point to valid locations inside the index buffer (e.g. to optimize a `RegionSubset`, you might merge successive index buffer intervals). When you are done, build the static geometry as usual. Now pass the `RegionsSubset` to the `addCustomGeometry` function, also passing an empty `GeometryBucketList`. The `GeometryBucketList` will be filled with the appropriate `GeometryBucket`s. To render the custom region, call `queue()` on each of the `GeometryBucket`s in the list.

When generating a static geometry tree, the `TightOctree` and `ProxySceneManager` do these steps for you.

3.9 The `LODBlendEntity` class

This class is derived from the standard Ogre `Entity` class. It adds functionality for smooth blending between LODs. Since blending can only be done with materials, you have to adapt the materials of the objects you want to blend. The `LODBlendEntity` calculates the blend factor for each LOD level (ranging from 0.0 to 1.0, where 1.0 means fully blended in) and stores it in the custom GPU parameter number 55 where it can be accessed by a GPU program.

3.10 `ProxySceneManager` Render Modes

The `ProxySceneManager` can operate in three render modes: standard mode, proxy mode and static geometry tree mode. Proxy mode or SG-tree mode are only available if proxies or the SG-tree have been created. Their main difference is the way they traverse the octree and add objects to be rendered along the way. Which octree traversal to use is decided in the `_findVisibleObjects` function. The following three functions define the octree traversals:

- The standard octree traversal is defined inside the `OctreeSceneManager` class, in the function `walkOctree`. In standard mode, the octree is traversed starting at the root, continuing through visible octants and adding visible objects along the way.
- The traversal in proxy mode is defined inside the `ProxySceneManager`, in the function `walkProxyOctree`. In this mode, proxies are queued for rendering instead of real geometry whenever the area of an octant on screen is small enough (the `ProxyCamera` class is used to determine the screen area). In this mode, scene nodes can be referenced by more than one octant. To avoid queuing the same node multiple times, each node can be tagged as queued (using the `_setHandled` function in `TightOctreeNode`).

- The traversal in SG-tree mode is also defined inside the `ProxySceneManager` class, in the function `walkStaticGeomOctree`. In this mode, static geometry buckets are queued for rendering for the more distant octants. Care is taken to add as few buckets as possible to the render queue.

To change render modes use the functions `setUseImpostors` and `setUseStaticGeom`.

A Interface Functions

`ProxySceneManager` class:

```
void markNode(SceneNode* node, bool markChildren=true)
void unmarkNode(SceneNode* node, bool markChildren=true)
MarkedNodeIterator getMarkedNodeIterator()
```

Marks/Unmarks the node pointed to by `node`. Only marked nodes are used in the construction of proxies and static geometry. Unmarked nodes are displayed as usual. If `markChildren` is set to `true`, all child nodes are marked as well.

```
void generateProxies(const String& baseName,
    const String& savefileName="")
void loadProxies(const String& baseName, const String& filename)
void clearProxies()
```

Generates a proxy representation of each octant in the octree. `baseName` shouldn't be used for any other mesh or texture. When using the function `loadProxies`, you have to specify the name of the file that stores the proxies. If a static geometry tree was already created when calling this function, it is cleared before generating proxies. `clearProxies` removes everything associated with proxies. The octree is also truncated to only act as a loose octree.

```
void generateStaticGeometryTree(size_t LODIndex)
void clearStaticGeometryTree()
```

Generates a static geometry tree, based on the scene octree. The LOD with index `LODIndex` of each object is added to the static geometry, or the highest (least detail) LOD, if a LOD with this index does not exist. If proxies were already created when calling this function, they are cleared before generating the static geometry tree.
`clearStaticGeometryTree()` removes everything associated with the static geometry tree.

```
void setUseImpostors(bool b)
bool getUseImpostors(void)
void setUseStaticGeom(bool b)
bool getUseStaticGeom()
```

Use these functions to specify the render mode. If both functions are set to `true`, impostors will be rendered (if available). If both functions are set to `false` or if the chosen mode is not available (because proxies or static geometry were not generated), standard rendering is used.

```
LODBlendEntity* createLODBlendEntity(const String& entityName,
   const String& meshName)
void destroyLODBlendEntity(Entity *e)
void destroyLODBlendEntity(const String& entityName)
```

Creates/destroys a `LODBlendEntity` with name `entityName`. Do not create a `LODBlendEntity` directly, you should always use this function to create a `LODBlendEntity`.

`LODBlendEntity` class:

```
void setBlendMode(LodBlendMode mode)
LodBlendMode getBlendMode()
```

LODBlendEntities automatically blend between LOD levels. These functions set/get the way two LODs are blended. Possible parameters are `LBM_SEQUENTIAL` (default) and `LBM_PARALLEL`. In sequential mode, first one LOD is blended in, then the other LOD is blended out. In parallel mode, both LODs are blended in parallel.