Using PagedGeometry
Tutorial 4: Custom PageLoader

Introduction

This tutorial explains how you can make your own custom PageLoader-derived class (like GrassLoader, TreeLoader2D, and TreeLoader3D) to have complete control over the dynamic loading process. By implementing your own PageLoader, you can load your trees, etc. from literally any source you can imagine, including hard disks, CDs, the Internet, and even procedurally.

Tutorial Requirements

- ✔ A fair understanding of PagedGeometry basics (see tutorials 1 – 3)
- ✔ A basic understanding of Ogre
- ✔ An fair understanding of C++ inheritance and polymorphism

How a PageLoader Works

By now you should know basically what a PageLoader is – it's a abstract class which is used to load all kinds of geometry (trees, grass, etc.) into PagedGeometry, since the PagedGeometry class itself has no built-in functions for the addition of trees, grass, etc. to the scene. This way, you can implement PageLoader – derived classes which allow your geometry to be added to the scene in literally any way you want. As you've seen, PagedGeometry includes several PageLoader classes by default, which can be used to easily display what you want.

As you probably know, PagedGeometry doesn't load everything in the scene all at one – it loads only what it needed to display visible trees, etc. This means entities are loaded dynamically, as your camera moves around in the world. When PagedGeometry needs a certain region of entities to be loaded, it will call on the PageLoader to do so. In other words, implementing a PageLoader is as simple as implementing a loadPage() function that loads entities within a requested boundary.

Defining a PageLoader

To create a custom PageLoader, simply declare a class deriving from PageLoader. For example:

```
class CustomLoader: public PageLoader
{
public:
	void loadPage(PageInfo &page);
};
```

As you can see, the PageLoader structure is very simple; it's basically just a callback. Once you implement the loadPage() function, your new PageLoader will be ready for use!

Here's an example implementation of loadPage(), which will populate the trees randomely:

```
void CustomLoader::loadPage(PageInfo &page)
{
	Vector3 position;
	Quaternion rotation;
	Vector3 scale;
	ColourValue color;
	for (int i = 0; i < 100; i++){
		//Calculate a random rotation around the Y axis
		rotation = Quaternion(Degree(Math::RangeRandom(0, 360)),
Vector3::UNIT_Y);
```

```
            //Calculate a position within the specified boundaries
            position.x = Math::RangeRandom(page.bounds.left, page.bounds.right);
            position.z = Math::RangeRandom(page.bounds.top, page.bounds.bottom);
            position.y = HeightFunction::getTerrainHeight(position.x, position.z);

            //Calculate a scale value (uniformly scaled in all dimensions)
            float uniformScale = Math::RangeRandom(0.5f, 0.6f);
            scale.x = uniformScale;
            scale.y = uniformScale;
            scale.z = uniformScale;

            //All trees will be fully lit in this demo
            color = ColourValue::White;

            //Add a tree to the scene at the desired location
            addEntity(myTree, position, rotation, scale, color);
        }
}
```

This example basically adds 100 trees to PagedGeometry with addEntity(). addEntity() is a function in the base PageLoader class through which all geometry is eventually added to PagedGeometry. To use addEntity(), simply specify the desired entity, postion, rotation, scale, and color.

The most important concept of this function is that the entities which are being added are *all* within the given boundaries. The boundaries, as well as other useful information, is passed to loadPage() through a PageInfo struct. For example, the X/Z position in the above loadPage() implementation is calculated like this:

```
            position.x = Math::RangeRandom(page.bounds.left, page.bounds.right);
            position.z = Math::RangeRandom(page.bounds.top, page.bounds.bottom);
```

So the trees will be between page.bounds.left – page.bounds.right in the X coordinate, and between page.bounds.top – page.bounds.bottom in the Z coordinate. It's important that you don't add any entities outside of these boundaries; when PagedGeometry asks you for a certain region of entities to be loaded, it expects exactly that to be done – attempting to add entities anywhere else may crash the application.

A "Page" of Geometry

As mentioned above, your loadPage() function is simply given a desired rectangular boundary along the X/Z plane, and the task of filling those boundaries with the desired entities. Implementing an effecient PageLoader with this information alone would be extremely difficult in many cases, since retreiving some random rectangular region of entities usually isn't very effecient.

There's one fact that can be extremely helpful when implemeting your PageLoader: The boundaries you are requested to fill out will *always* have the same width and height (both being equivelant to the value given to PagedGeometry::setPageSize()), and will *always* be a single "tile" of an infinitely sized "virtual grid".

It may be helpful to picture the actual layout of your trees as being divided into a huge grid, where each "tile" of that grid (called a "page") can be individually loaded and unloaded. The PageInfo parameter specified to your loadPage() function will always be providing the boundaries, etc. of *one* of those tiles, which you need to load. In fact, you could implement your entire loadPage() function if only given the x and z indexes of the desired tile.

Conclusion

A PageLoader is nothing more than an advanced callback function; when the PagedGeometry engine decides a certain page of geometry needs to be loaded, it calls on the PageLoader to do this. While creating your own custom PageLoader class is an advanced, and sometimes difficult task, it's really very simple in concept. Through custom PageLoader's, your trees, grass, etc can now be loaded from literally any source you can imagine.