NGF - nikki's Game Framework Manual

NOTE: Preliminary version, not done yet.

NGF (nikki's Game Framework) is a set of framework classes in C++ that can help in the development of software, especially games, using Ogre. It has been used in GraLL (http://www.grall.uni.cc).

If you find any fault in the documentation, feel free to tell me about it. You can send me an email at s.nikhilesh@gmail.com.

0. Contents

1. Concepts

GameObjects and the GameObjectManager

While making a game, or at least, one that is more complex than pong or pacman, it is really difficult to put everything in just a single 'main()' function. It could be divided up into different functions, but that would not fully exploit the object-orientedness of C++. So, we use the old technique of 'divide and conquer', and try to divide the game into objects, that have variables (properties and attributes that should be 'remembered'), and functions (methods or events, in which the object should 'do' something).

An object could be, for example, the 'player', an 'enemy' or even a 'sound source'. If you think of your game as a play, then, the objects are the actors, that communicate, do things and influence each other. They enter and are introduced at some point in the play, and then they leave at some point in the play.

We could directly make different classes based on this, but, we can see that objects have certain similarities, some of which are given below:-

- Most of them need to do something when they are created.
- Most of them need to do something when they are destroyed.
- Most of them need to do something 'everytime' (search for enemies, play sounds)

- Most of them need to send messages to others.
- Most of them need to receive messages and act according to the information received.
- Most of them need to do something when they collide with another object.
- Many of them need to create or destroy other objects.
- Many of them need to destroy themselves.
- Objects can be grouped based on further similarities such as 'enemies', 'collidables', 'dangerous' etc.

There may also be other similarites you can think of.

All of these objects need to be 'managed' (kept track of, updated at each time interval etc.). For this, a list of all objects must be maintained. This fact, along with the similarites above, will lead to the conclusion we should have a class from which all objects derive. This class should have certain methods used by most objects such as 'create', 'destroy', 'update', 'receiveMessage' etc.

This is what the NGF class, 'GameObject' is. And, the NGF class 'GameObjectManager' manages these objects. It keeps track of them, updates them at each time interval, and provides methods to create them, destroy them and send messages between them. If you use OgreODE, or OgreBullet, it comes with a method to inform objects about collisions. If not, you can rewrite this method according to the physics library you use.

Worlds and the WorldManager

In most games, there are different 'scenes', for example, a menu screen, an options screen and the different levels. Usually, these scenes create objects at the beginning, and destroy them all at the end of the scene. Also, most games have a pre-defined 'order' of scenes, and these scenes are run in order.

You can think of it as a play, with different acts. The play has a pre-defined order of acts, and these acts are acted out in order. However, what happens in a game is not fixed like in a play. In many cases, we jump from one scene to another scene not following the order.

Something has to be done at the beginning of the scene, at the end, and while it is being run. So, each scene has to have an 'initialise' event, an 'update' event, and an 'end' event. There also has to be an entity that remembers the order of the scenes and runs them in order.

In NGF, these scenes are called 'Worlds'. And, the NGF class 'WorldManager' remembers their order and runs them. The WorldManager has methods to express the order of the Worlds, start running the Worlds, go to the next World, go to the previous World, or jump directly to any World in the list. When it is told to run a different World (the next one, the previous one, or any other World), it tells the currently running World to carry out its 'end' event, and the next World to carry out its 'intialise' event. At each time interval, it updates the World that is running currently.

Loading and the NGF file format

Although GameObjects can be used for anything, they are usually used to represent physical objects, with a position and orientation. It is difficult to hard code the position and orientation of every object in a scene. It is much easier to edit the positions and orientations of the objects in a graphical editor, export it to a file, and load up that file in the game. For this, the '.ngf' file format is provided. A '.ngf' file is just a simple text file, like this:-

```
ngflevel Lvl1
{
      object
```

```
	{
		type PhysicsObj
		name noname
		position 0.500000 0.500000 36.500000
		rotation 1.000000 0.000000 0.000000 0.000000

		properties
		{
			objType Box
		}
	}

	object
	{
		type PhysicsObj
		name Key
		position 0.500000 0.500000 72.500000
		rotation 1.000000 0.000000 0.000000 0.000000

		properties
		{
			objType Sphere
			speed 200
		}
	}
}

ngflevel Lvl2
{
	object
	{
		type Brush
		name noname
		position -0.500000 1.500000 25.500000
		rotation 1.000000 0.000000 0.000000 0.000000

		properties
		{
			brushMeshFile Lvl2_b0.mesh
		}
	}
}
```

It is a very simple format, and an exporter is already provided for Blender. The usage of the Blender exporter is documented in the file '<fixme>' in the 'docs' folder.

In each file, more than one level can be specified, using the 'ngflevel' keyword. These files are loaded automatically when the Resources are initialised in Ogre. Each level has many objects, and each object specifies its type (such as 'Player', 'Enemy' etc.), its name (names of specific instances of a GameObject, such as 'MainEnemy', 'BossLevel2' etc.), its position and rotation, and its properties.

The property list has many 'keys' (name of the property), and 'values' (values of the property). More than one value can be used for one key, for example, to hold RGB colour information. These translate directly into the NGF::PropertyList type. Properties can be easily obtained in code using:

```
propertyList.getValue(Ogre::String key, unsigned int index, Ogre::String defaultVal)
```

The file is then loaded using the 'Loader' class under namespace 'Loader' in NGF. The actual objects are not created by the Loader. Instead, the Loader class calls a callback function that is defined by the user. It calls this function for each line that it encounters in the file, and provides it with the type, name, position, rotation and properties. The callback function is free to do whatever it wants. This is because there is no easy way (that I know of) to convert a string to a class name in C++.

This callback system also makes the loading system very flexible. This type of callback function should have the following structure:-

```
void func(String type, String name, Vector3 pos, Quaternion rot, NGF::PropertyList props)
```

Here is an example that is sure to explain everything:-

```
void loaderHelperFunction(String type, String name, Vector3 pos, Quaternion rot,
                          NGF::PropertyList props)
{
	if (type == "TestObj")
	{
		NGF::GameObject *obj = Globals::gom->createObject<TestObj>(pos, rot, props,
		                                                           name);
	}

	if (type == "Enemy")
	{
		//You can also do other things while creating a GameObject. This shows the
		//flexibility of the loader mechanism (this is actually a rather stupid example,
		//because a thing like incrementing an enemy count would be done in the creation
		//even of the object itself, not in a loader callback like this).
		NGF::GameObject *obj = Globals::gom->createObject<Enemy>(pos, rot, props, name);

		++enemyCount;
	}

	if (type == "Song")
	{
		//The level loader mechanism doesn't always have to be used to load just
		//GameObjects. Here is an example that doesn't create any objects, but plays
		//music. Other possible uses are level settings, such as gravity, ambient light
		//etc.
		musicPlayer->playSong(props.getValue("file", 0, "HappyBirthday.ogg");
	}
}
```

Thus, the loading mechanism has nothing to do with GameObjects as such. It simply parses the file, and tells you about it. It is for you to do whatever you want with the information given. So, 'type', 'name', 'pos' etc., don't mean really mean anything until you use them as you see fit.

2. Installation

Just extract the zip file to any location on your hard drive. To use the Blender to .ngf exporter, put the file 'ngf_export.py' (located in the 'blenderExport' folder) in your Blender scripts folder (by default, <blender installation folder>/.blender/scripts, or ~/.blender/scripts under Linux). NGF requires boost.

3. Usage

Include the file 'Ngf.h' (found in 'ngf/include') in your project's source code (make sure that it lies in the include path). You would also have to compile 'ConfigScript.cpp' and 'Ngf.cpp', and link them to your project (this is usually done just by copying them into the project folder, or symlinking to them, and compiling them along with the other project source code files).

If you want to use OgreOde or OgreBullet with NGF, you will have to comment or uncomment the first few lines of 'Ngf.h' accordingly.

4. Explanation

Not so fast! :-)