NGF - nikki's Game Framework Tutorial

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. Introduction

This tutorial will teach you how to get started with NGF. We won't discuss things such as the FrameListener, SceneNode, Entities, SceneManager, ExampleApplication, or how to use Blender in this tutorial. It is suggested that you have a basic knowledge of how Ogre and Blender work before starting out with this tutorial. Also, you might want to read the NGF manual to get a grip on the basic NGF concepts.

We use the 'Minimal Ogre Collision (MOC)' library from http://www.artifexterra3d.com/moc-minimal-ogre-collision.php. Thanks to the people at Artifex for the great library!

2. Getting Started

Create a copy of the 'ngftutorial/ngftutorialbase' folder in the same directory, probably something like 'ngftutorial/mytutorial'. Once you are done with this tutorial, you would have something like 'ngftutorial/ngftutorialfinal'.

3. Building The Project

If you're on Linux, you can build it easily using premake with a gnu-make target, if you have the required pkg-configs (Ogre, and OIS). You must also edit the premake.lua files to target the correct boost and ExampleApplication include directories.

If you're on Windows, you can use premake to make the required project files. Then, edit the additional include directories list to add the Ogre, OIS, boost and ExampleApplication include directories.
It should build fine. If it doesn't, ask in the forums! :-) Once built, run the application, and you'll hopefully see the

Ogre logo and the green statistics panel.

4. The Base Code

The 'main()' function does the usual starting up and running of an ExampleApplication-based application. We use a cheap 'Globals' namespace hack (shown in Globals.cpp) so that we don't have to pass pointers around. :-)

Let's check out some code in the files in the 'include' directory.

Look at NGFExampleApplication::setupResources() in NGFExampleApplication.h:-

```
//Create the NGF stuff. The level loader needs to be created before the resource parsing,
//because it needs to know about the *.ngf files.
Globals::gom = new NGF::GameObjectManager();
Globals::wom = new NGF::WorldManager();
Globals::load = new NGF::Loading::Loader(&loaderHelperFunction);
```

Here, we create the GameObjectManager, the WorldManager, and the Loading::Loader. If you were to use one one of the existing physics engine wrappers for the GameObject collision events (by uncommenting the appropriate lines at the beginning of Ngf.h), you would have to pass the physics engine's 'World' pointer to the GameObjectManager in the constructor. However, we do not use the physics engines here, so there is nothing we must pass to the GameObjectManager. The WorldManager doesn't require anything either. The Loading::Loader takes a function pointer, of the 'helper function' (lame name, I know, I didn't think of the word 'callback' at that time, and its too late now). We will see this helper function later. The Loading::Loader should be created before you load any resource groups, or the *.ngf scripts (used to define levels) won't get loaded automatically.

Now, look at the end of NGFExampleFrameListener::frameStarted() in NGFExampleFrameListener.h:-

```
//Update NGF stuff. Exit (return false) if F12 key down.
Globals::gom->tick(false, evt);
return (!(mKeyboard->isKeyDown(OIS::KC_F12)));
```

We have to update the WorldManager and GameObjectManager every frame. This is done through their respective 'tick()' functions. We tell the GameObjectManager whether the game is paused or not (it calls the appropriate 'tick' function for each GameObject), and we pass it the FrameEvent, so that it can pass it on to the GameObjects. We give the FrameEvent to the WorldManager for similar reasons. The WorldManager returns 'false' from its tick function if we have to shutdown. Here, we shutdown (by returning 'false' from a 'frameStarted' event) if the F12 key is pressed. We don't update the WorldManager as we're not yet using Worlds, and so have to handle shutting down ourselves.

Lets check out the 'helper function', in 'Helper.h':-

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

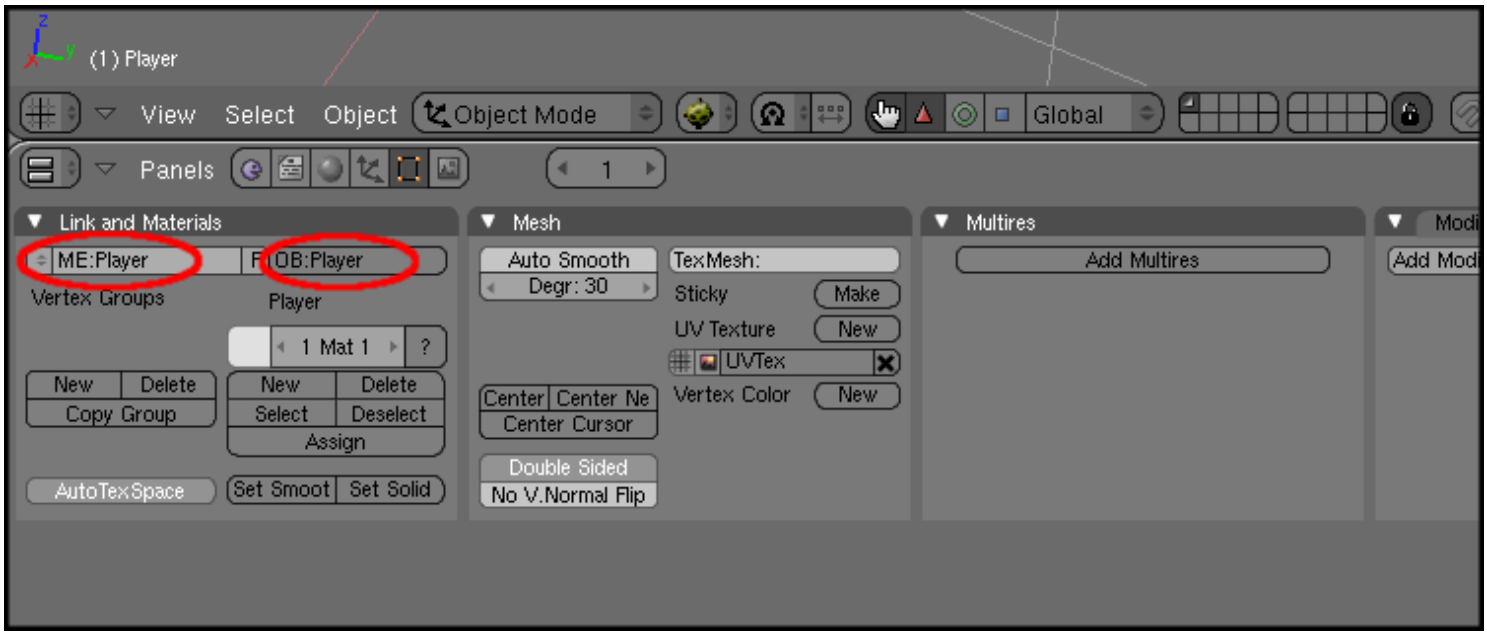
This function 'helps' the Loading::Loader do its job. The Loader simply parses the level file, and calls this function to the the actual work. This function takes the type of object, its name, position, orientation and properties. Generally, you would simply create a GameObject of the appropriate type and pass it the given information. You could also do other things here, may be not even create a GameObject.

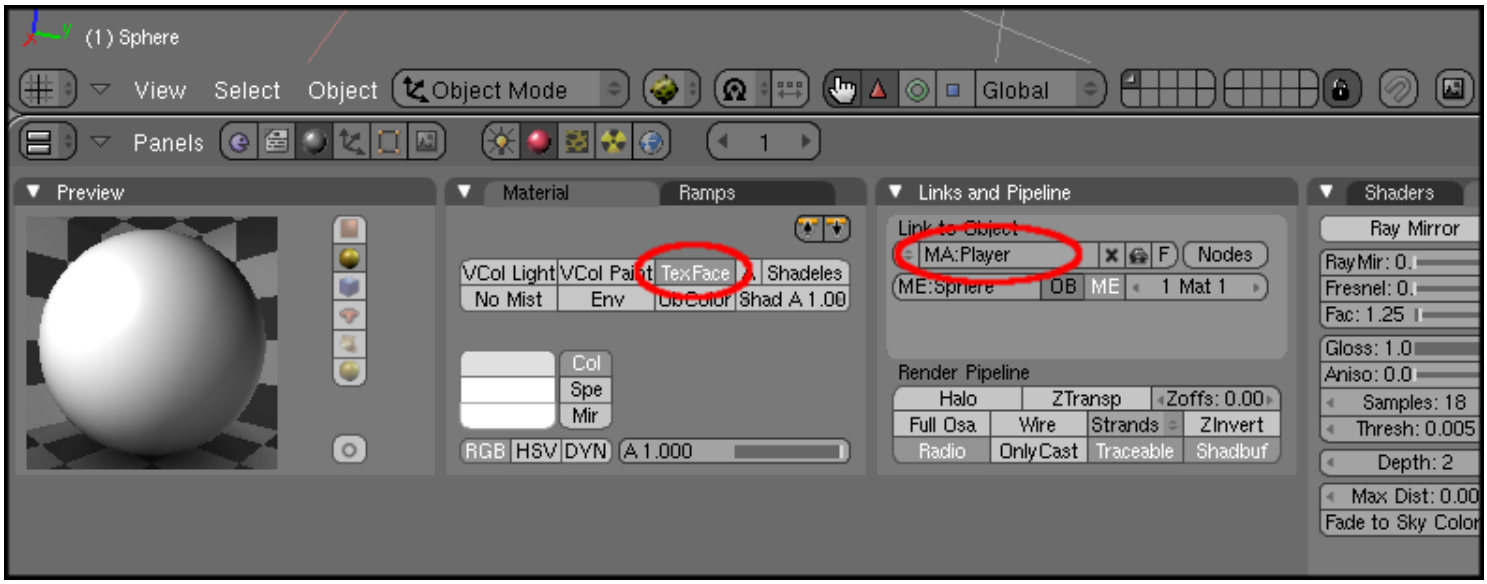
4. Adding A GameObject – The Player

We first have to export the Player.mesh file from Blender. If you don't want to do that, or don't have Blender, then you can just copy the relevant files over from 'ngftutorialfinal'.

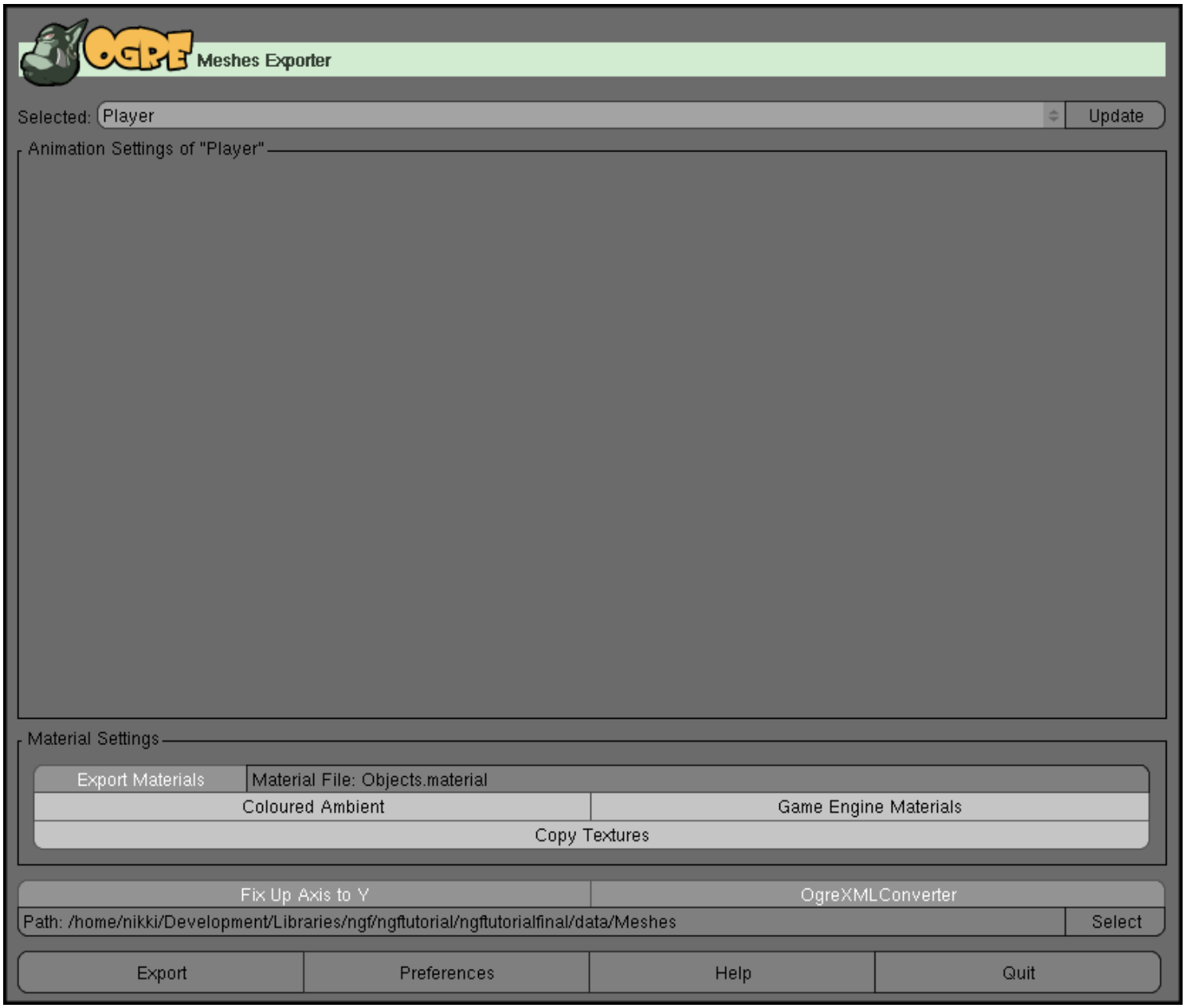
Put the Ogre export script from http://www.ogre3d.org/phpBB2/viewtopic.php?t=45922 into your '.blender/scripts' folder.

Now, open 'etc/ExampleObjects.blend' with Blender. You'll see a spiky object with a yellow and black texture. This is gonna be our Player, say hello! :-) First, we'll give him an appropriate Object and Mesh name, 'Player'. Then, we'll give him a new material (click 'Add New'), named 'Player', and select 'TexFace', so that that it incorporates the texture into the material:-





After doing this, select 'File->Export->OGRE Meshes'. Change the export directory to your 'data/Meshes' directory, and the name of the material file to 'Objects.material'. Choose 'OgreXMLConverter', to get the resultant .mesh.xml files converted to .mesh automatically, and 'Flip Up Axis To Y' to realign it to the Ogre coordinate system.



Lastly, copy 'Player.png' from 'etc' into 'data/Textures'.

Now that we've exported the mesh, we're ready to begin coding!

Create a file, 'include/Player.h'. Copy the following code into it:-

```
class Player : public NGF::GameObject
{
protected:

public:
    Player(Ogre::Vector3 pos, Ogre::Quaternion rot, NGF::ID id, NGF::PropertyList properties,
           String name)
              : NGF::GameObject(pos, rot, id , properties, name)
    {
    }

    ~Player()
    {
    }

    void unpausedTick(const Ogre::FrameEvent &evt)
    {
    }

    void pausedTick(const Ogre::FrameEvent &evt)
    {
    }

    void receiveMessage(NGF::Message msg)
    {
    }
};
```

Basically, we're just deriving from NGF::GameObject, passing it the stuff in the constructor, and overriding some of the events. Lets make it load the mesh file when its created. First, we add two protected variables to the class:-

```
SceneNode *mNode;
Entity *mEntity;
```

This is usual Ogre stuff. Now, in the constructor, we add this:-

```
String idStr = StringConverter::toString(id);
addFlag("Player");

//Create Ogre stuff.
mEntity = Globals::smgr->createEntity(idStr + "-testObjectEntity", "Player.mesh");
mNode = Globals::smgr->getRootSceneNode()->createChildSceneNode(idStr + "testObjectSceneNode", pos,
                                                                rot);
mNode->attachObject(mEntity);
```

First, we convert the ID number (at any point in time, IDs of all GameObjects are unique, and each GameObject has the same ID throughout its lifetime) into a string for later use. Then we add a flag '"Player"', which is useful if another GameObject wants to identify this one's type. Then, we create the Ogre Entity and SceneNode using idStr, with the Player.mesh we created before, and we attach the Entity to the SceneNode. In the destructor, we clean all this up, with this code:-

```
//Destroy Ogre stuff.
mNode->detachAllObjects();
Globals::smgr->destroyEntity(mEntity);
Globals::smgr->destroySceneNode(mNode->getName());
```

Add this line before '`#include "Helper.h"`', in main.cpp:-

```
#include "Player.h"
```

Now, lets create an instance of the GameObject in the scene. Add this code after the existing code in the NGFExampleApplication::createScene() function:-

```
//Create an instance of the Player GameObject.
Globals::gom->createObject<Player>(Vector3::ZERO, Quaternion::IDENTITY);
```

Here, we tell the GameObjectManager to create a GameObject of type Player, at position (0,0,0), and identity orientation.

Compile and run it, and move the camera around (use WASD or the mouse, while holding down the right mouse button). You should see the Player.

Now, lets make him move around. We'll make him move with I and K, and turn with J and L. Put the following code into the 'unpausedTick()' function in Player:-

```
Real move = Globals::keyboard->isKeyDown(OIS::KC_K) - Globals::keyboard->isKeyDown(OIS::KC_I);
Real rot = Globals::keyboard->isKeyDown(OIS::KC_J) - Globals::keyboard->isKeyDown(OIS::KC_L);

mNode->yaw(Degree(rot * 100 * evt.timeSinceLastFrame));
mNode->translate(mNode->getOrientation() * Vector3(0,0,move * 4 * evt.timeSinceLastFrame));
```

I won't explain this, its not NGF-related. Besides, it's pretty self-explanatory. :-)

This is it for our Player now, we'll do the rest after we create the LevelGeometry GameObject.

4. Adding A GameObject – The Level Geometry

Create a new file, 'include/LevelGeometry.h'. Copy the following code into it:-

```
class LevelGeometry : public NGF::GameObject
{
protected:
    SceneNode *mNode;
    Entity *mEntity;

public:
    LevelGeometry(Ogre::Vector3 pos, Ogre::Quaternion rot, NGF::ID id, NGF::PropertyList properties,
                  String name)
                     : NGF::GameObject(pos, rot, id , properties, name)
    {
        String idStr = StringConverter::toString(id);
        addFlag("LevelGeometry");

        //Create Ogre stuff.
        String brushMeshFile = properties.getValue("brushMeshFile", 0, "Player.mesh");
        mEntity = Globals::smgr->createEntity(idStr + "-levelGeometryEntity", brushMeshFile);
        mNode = Globals::smgr->getRootSceneNode()
                              ->createChildSceneNode(idStr + "levelGeometrySceneNode", pos, rot);
        mNode->attachObject(mEntity);
    }
```

```
    ~LevelGeometry()
    {
        //Destroy Ogre stuff.
        mNode->detachAllObjects();
        Globals::smgr->destroyEntity(mEntity);
        Globals::smgr->destroySceneNode(mNode->getName());
    }

    void unpausedTick(const Ogre::FrameEvent &evt)
    {
    }

    void pausedTick(const Ogre::FrameEvent &evt)
    {
    }

    void receiveMessage(NGF::Message msg)
    {
    }
};
```

This is almost exactly like the Player GameObject, the only difference is in the names in a few places, and how the Entity is created. For the Entity, the mesh file to use comes from the properties. We use the value of the 'brushMeshFile' property as our mesh file.

Generally, we see two kinds of GameObjects in a level. One that has the same mesh for every instance of itself, or generally, one who's mesh isn't edited in the level editor while editing the level. These are usually things like the Player, Enemies etc. You generally create these first, and then assemble levels out of them. The other kind contains those GameObjects whose mesh you edit directly in the level editor, while editing the level itself. These generally include walls and floors. I call them 'brushes'. Of course, this distinction was simply what I had in mind while creating NGF, you might think of it a different way.

In Ogre, meshes are usually defined in a mesh file. So, every instance of a brush GameObject would have its own mesh (some may also share a mesh), which you edit while editing the level itself. Each mesh has its own material too. They must, of course, have distinct names. You wouldn't want to name each file individually. You'd just create a brush, give it a texture, and expect it to appear the same way in the game. The Blender exporter has a way of doing this for you.

In Blender, the mesh and material of any object with the property (in Blender, objects can have properties) 'isBrush', will be given a unique name in that .blend file, and a 'brushMeshFile' property is added with the respective mesh name. The 'isBrush' property says that an object is a brush. In a single file, you'd have something like this:-

Level1:
Brush1: Mesh – Level1_b0, Material – Level1_b0_m0
Brush2: Mesh – Level1_b1, Materials – Level1_b1_m0, Level1_b1_m1
Brush2: Mesh – Level1_b2, Material – Level1_b2_m0

Level2:
Brush1: Mesh – Level2_b0, Material – Level1_b0_m0
Brush2: Mesh – Level2_b1, Material – Level1_b2_m0

There is a 1:1 correspondence between Blender '.blend' files and NGF '.ngf' files; between Blender 'scenes' in the .blend file and 'ngflevels' in the .ngf file; between Blender 'meshes' in the .blend file and Ogre '.mesh' files; and

between Blender 'materials' and Ogre 'materials' in the '.material' file. The .mesh and .material files have to be created using the Blender->Ogre mesh exporter.

Bear in mind that all this applies just for the Blender->.ngf exporter. NGF itself doesn't know anything about brushes, it just provides the property system that we use.

Now, back to the LevelGeometry GameObject. Here, we use the 'brushMeshFile' property, so that we use whatever shape we're given in Blender. 'brushMeshFile' is the 'key' of this property, and it can have many 'values' (a colour property, for example, may have several values to express a colour). We tell it to use the first value (the value with index 0). Also, if it can't find this property or value, it defaults to 'Player.mesh'.

Before we get Blender to work with our application, lets run a quick test of the LevelGeometry GameObject. Add the line '`#include "LevelGeometry.h"`' after '`#include "Player.h"`' in main.cpp. Then, put this in the NGFExampleApplication::createScene() function:-

```
//Create a test LevelGeometry instance.
Globals::gom->createObject<LevelGeometry>(Vector3(0,-1,0), Quaternion::IDENTITY,
NGF::PropertyList::create("brushMeshFile", "Player.mesh"));
```

The 'create' static function creates a PropertyList with just one property pair. We don't really have to do this as our LevelGeometry code defaults to the same value, but we do it anyway. If you want to add more properties, you can create a chain of 'addProperty' function calls. Each returns a reference to the PropertyList itself.

Build and run it. You should see a static Player-shaped (and textured) object below the actual player. That's our LevelGeometry. :-) We don't have the collision detection code in yet, we'll add it after the next section.

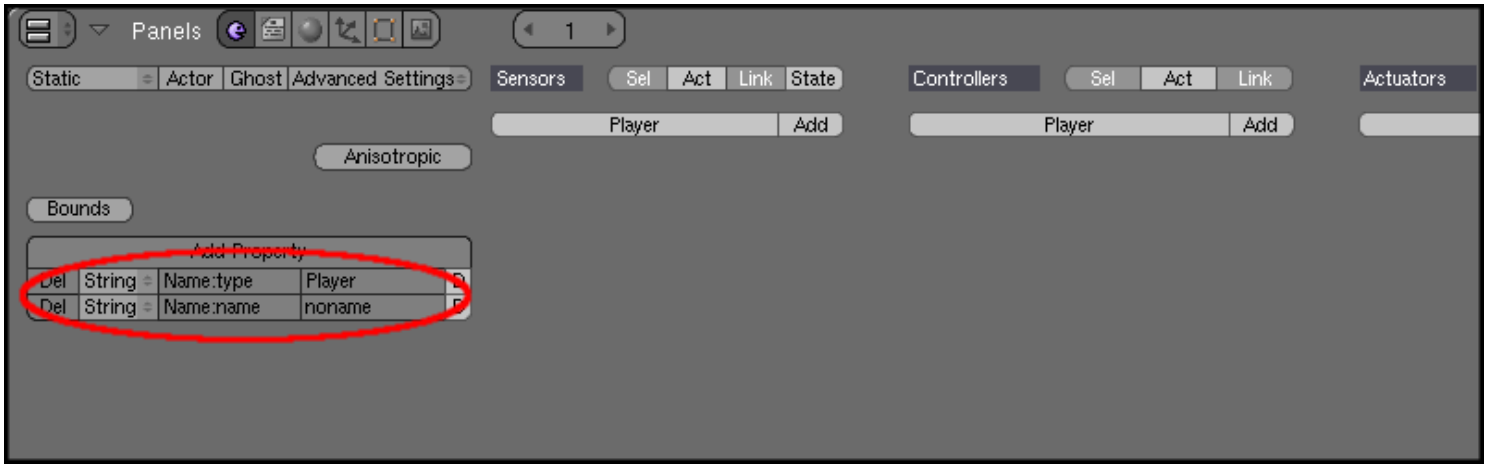
Now, let's try level editing with Blender.

4. Making It Work With Blender

While making levels, we usually first plan out our level geometry (the static stuff, usually brushes), and then start dropping in the logic entities like Monsters, Keys, Doors. So, we need to be able to select from a list of GameObject types and just drop an instance in. Blender's 'Append' function allows us to do this easily. It puts a copy of an object from another file into the file we're editing. So, creating a 'GameObject list' is simply a matter of creating a .blend file with all your objects. You then append from this file. It may change the name of the object, but the properties of the object remain the same (and this is what NGF uses to export a .ngf file). You can also create many files and organise them in a folder hierarchy if your game is really huge and has many types of GameObjects.

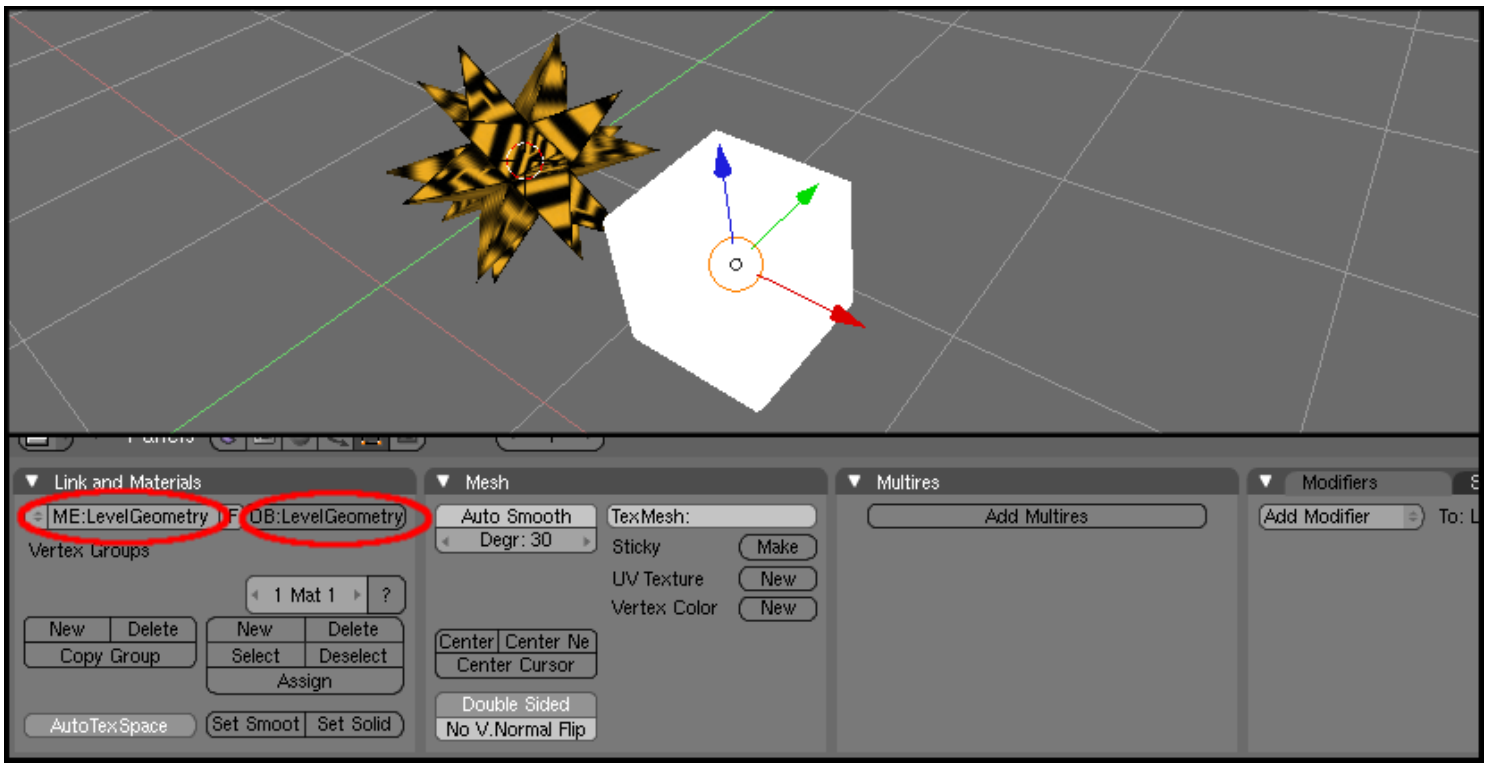
Let's first fill in the required properties for the Player. Open 'etc/ExampleObjects.blend'. In the Buttons Window, open the 'Logic' tab by clicking on the purple pacman icon, or pressing F4. Click 'Add Property' twice, to create two new property pairs. Make both of them of type 'String'. Name the first one 'type', and give it a value of 'Player'. This represents our GameObject type. Name the second one 'name' and give it a value 'noname'. This is the name of this particular instance of theGameObject, which can be used, for example, to get a pointer to the Monster occupying a specific position in a level (GameObjectManager allows you to retrieve pointers to GameObjects by name). The name is meant to be set specifically for each instance of the GameObject you create (that is, for each copy of it in your levels), not here.

Other properties (except 'isBrush', which, as we saw before, proclaims an object as a brush) translate directly into the NGF::PropertyList that is received in the helper function. Examples include a Monster's 'difficulty'.

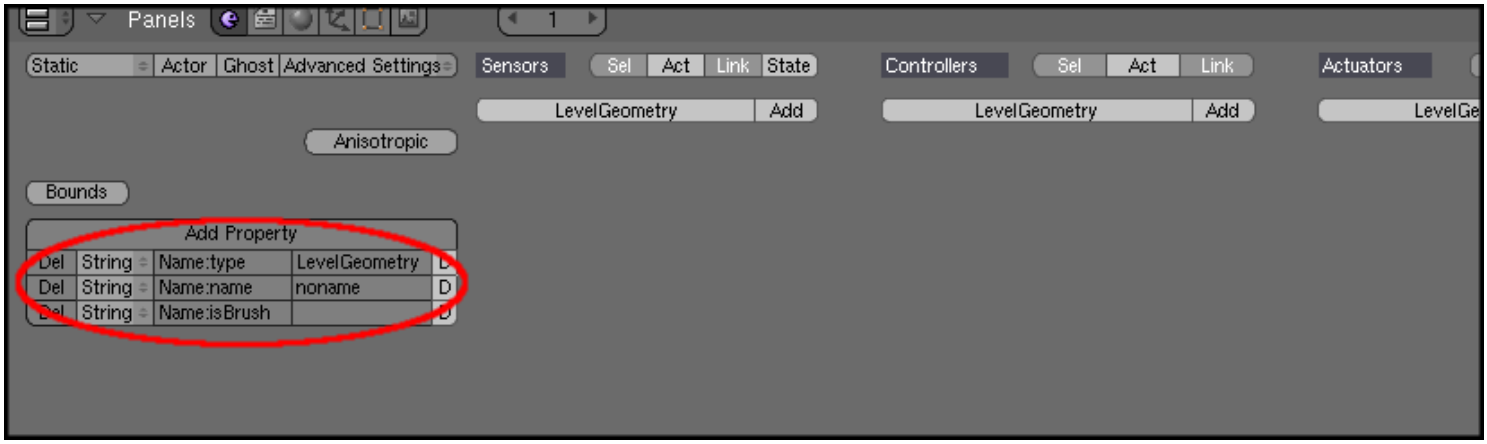


Now, its time to create the LevelGeometry Blender object.

Select the Player, and press Shift+S, and then select 'Cursor → Selection'. Press NUM7 (to go to top view), and then press Space, and choose 'Add → Mesh → Cube'. Press S, and then type 0.25 and press Enter to scale it to 1/4th its size. Press TAB to go into Object mode. Now, press G and move your mouse while holding Ctrl to move it to an empty spot. Name the object and its mesh 'LevelGeometry'.



We'll now fill in the properties for the LevelGeometry object. We do the same thing that we did for the Player, except, this time, we enter 'LevelGeometry' for type, and we add the 'isBrush' property. The 'isBrush' property just tells the exporter, “I'm a brush, put my mesh's name in the .ngf script”.



We don't have to export this cube mesh, because this mesh itself is actually never used. It's just a placeholder, kind of like an 'icon' for LevelGeometries in Blender. In an actual instance, you'd remove this icon and make your own mesh. Save the file with Ctrl+W.

Remember that Loading::Loader doesn't actually create any objects. It only calls the helper function. So, we need to check the type and accordingly create GameObjects in the helper function. Put the following code in the 'loaderHelperFunction()' function:-

```
if(type == "Player")
{
    Globals::gom->createObject<Player>(pos, rot, props, name);
}

if(type == "LevelGeometry")
{
    Globals::gom->createObject<LevelGeometry>(pos, rot, props, name);
}
```

This code is self-explanatory. We just create GameObjects depending upon 'type', and give them their respective positions, orientations and NGF::PropertyLists (the NGF::PropertyList contains the Blender properties other than type, name and 'isBrush').

We haven't made the level yet, but we'll name it 'TestLevel'. Replace the previous two GameObject-creation lines in the NGFExampleApplication::createScene() function with:-

```
//Load the .ngf level.
Globals::load->loadLevel("TestLevel");
```

We can also provide an additional offset and rotation to place and orient the level. This is useful if you load a level without clearing up the scene, thus 'appending' to the existing scene.

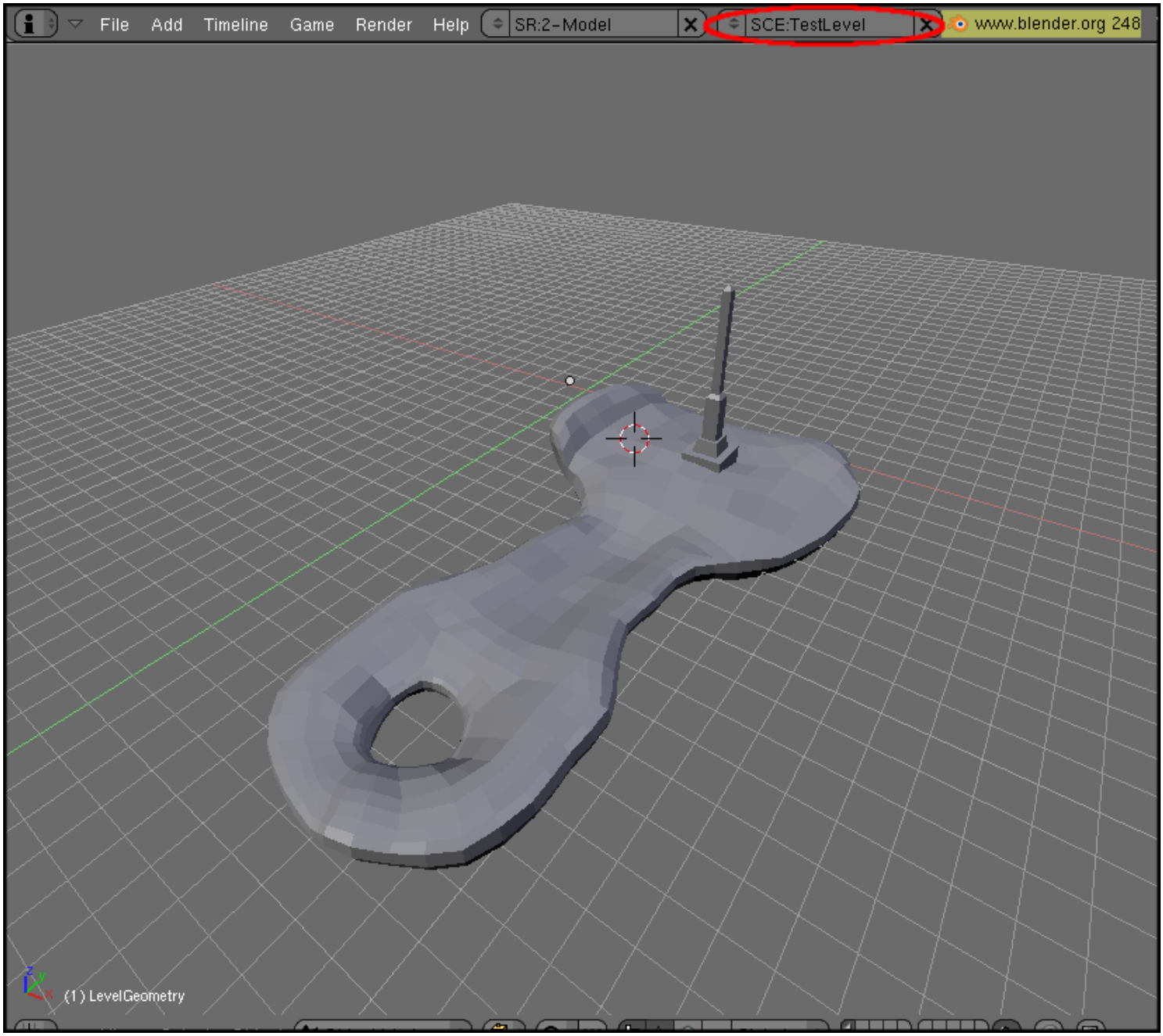
Now, we can proceed with making the actual level.

Open 'etc/ExampleLevels.blend'. First, let's name our level. Near the top of the window, you'll see something like 'SCE:Scene'. Click on it, and rename it to 'TestLevel'.

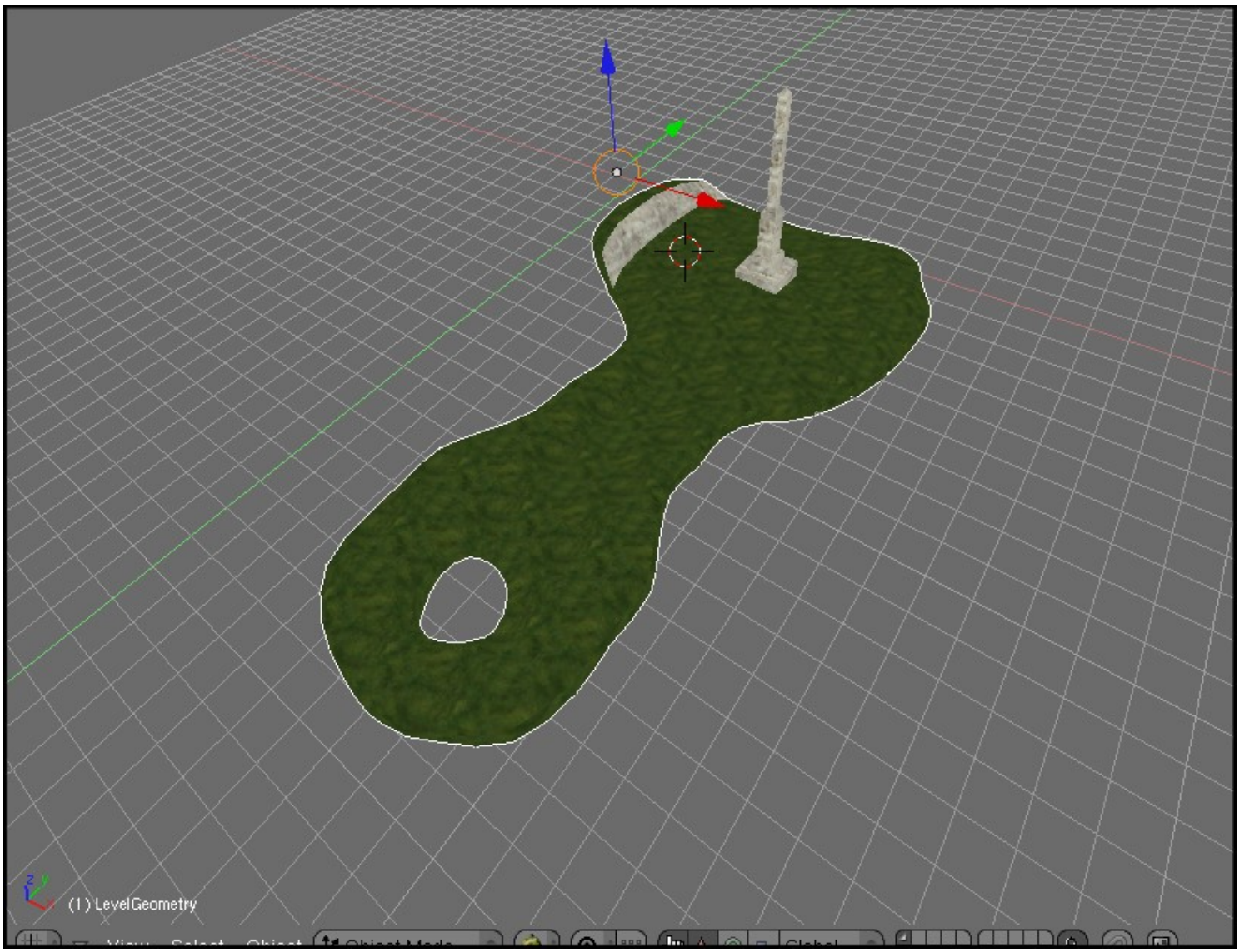
We'll now add a LevelGeometry object. Click 'File → Append or Link', or press Shitft+F1. It shows us

ExampleLevels.blend (this file). Click 'P' to get out of it. Then, browse to and open 'etc/ExampleObjects.blend' (make sure you get the one in your folder, not the one in 'ngftutorialbase' or 'ngftutorialfinal'). Click 'Object', and then 'LevelGeometry'. Click 'Load Library' to copy the LevelGeometry object into ExampleLevels.blend.

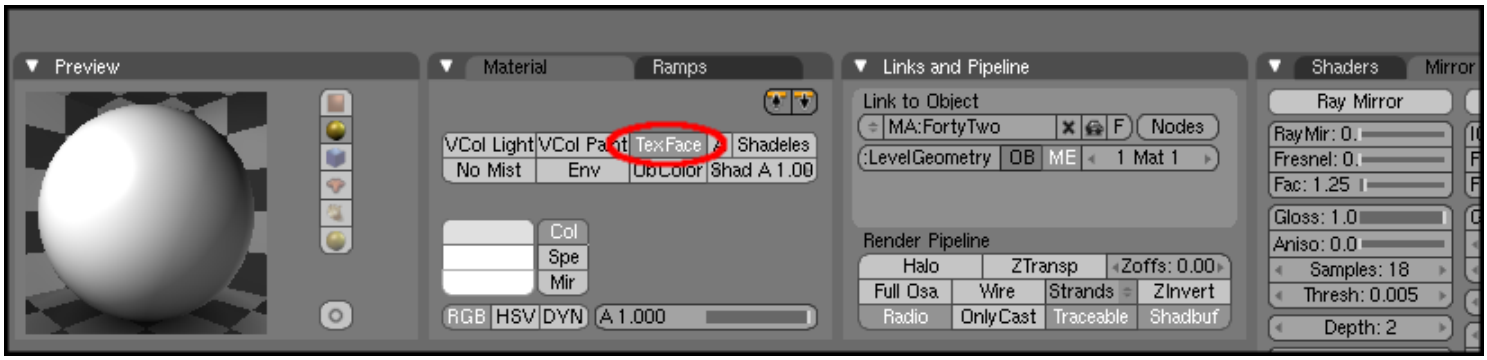
Select the LevelGeometry object, and press TAB to go into edit mode. Delete the existing mesh, and create some level geometry. I'm not going to show you any specifics, but you could probably add a plane, subdivide it a few times and extrude around. You could also start off with a cube and keep extruding. All you need is a closed solid, the Player should remain outside it. Don't use just one LevelGeometry object for the entire scene, use many. That way, it'll be easier to give them textures.



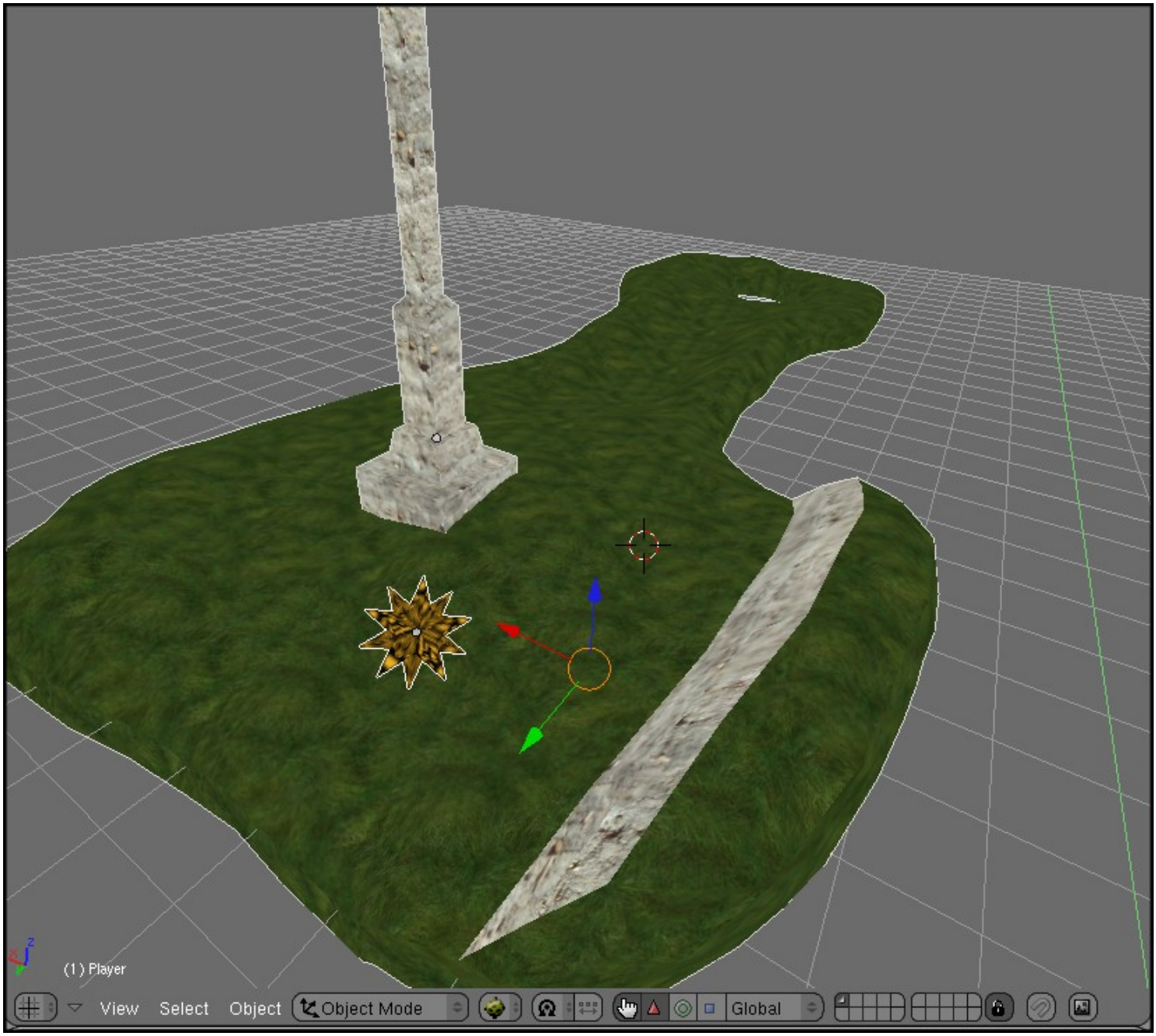
Put a few textures in the 'etc' folder (either make your own or get some from the internet), and texture your model.



Now, we have to create a material for the LevelGeometries. For each object, create a new material, and select TexFace. The name of the material is insignificant, because the .ngf exporter automatically renames them as you read earlier.



Now, put in the Player somewhere on the LevelGeometry by appending him from the ExampleObjects.blend file as you did for the LevelGeometries earlier.



We are now ready to export the level to a .ngf file. First, copy the 'ngf_export.py' from the 'blenderExport' directory where you extracted NGF into your '.blender/scripts' directory, or any directory that Blender searches for scripts. Then, reload the menus in Blender either by restarting it, or by choosing 'Scripts → Update Menus' in the Scripts Window.

Choose 'File → Export → NGF Level (.ngf)', and browse to the data directory, and overwrite 'Levels.ngf'. The exporter makes the brushes get selected, so you can easily export them to .mesh. Now, select

'File → Export → OGRE Meshes'. Choose 'data/Meshes' for the directory, and 'TestLevel.material' for the material filename. Also, click on 'OgreXMLConverter' and 'Flip Up Axis To Y'. Then, click 'Export', and then 'Quit'. Copy your textures into the 'data/Textures' folder.

Just build and run the application, you should see your level loaded, and should be able to move the Player around with IJKL. If you edit the level again, and only change the positions of the objects, or add more objects that are not brushes, or generally do anything that does not affect the meshes, then you don't have to export the meshes again. You will only have to export the .ngf.

You might have noticed that the Player goes right through the LevelGeometry, and also doesn't stick to the floor. Let's use the MOC library to fix this. MOC requires 'query flags' to define the types of objects. Let's define the query flag for the LevelGeometry. Add the following to 'include/Globals.h':-

```
enum QueryFlags
{
    QF_LEVELGEOMETRY = 1<<7
};
```

Now, we have to say that the LevelGeometry Entity is of the 'QF_LEVELGEOMETRY' type. In the LevelGeometry constructor, add the following:-

```
mEntity->addQueryFlags(QF_LEVELGEOMETRY);
```

Lastly, we have to change to Player movement code. Change Player::unpausedTick() to look like this:-

```
void unpausedTick(const Ogre::FrameEvent &evt)
{
    Real move = Globals::keyboard->isKeyDown(OIS::KC_K) - Globals::keyboard->isKeyDown(OIS::KC_I);
    Real rot = Globals::keyboard->isKeyDown(OIS::KC_J) - Globals::keyboard->isKeyDown(OIS::KC_L);

    mNode->yaw(Degree(rot * 100 * evt.timeSinceLastFrame));

    //Remember the old position before moving.
    Vector3 oldPos = mNode->getPosition();
    mNode->translate(mNode->getOrientation() * Vector3(0,0,move * 5 * evt.timeSinceLastFrame));
    Vector3 newPos = mNode->getPosition();

    //Move back if the new place isn't comfortable enough.
    if (Globals::col->collidesWithEntity(oldPos, newPos, 0.7, 0, QF_LEVELGEOMETRY))
        mNode->setPosition(oldPos);

    //We can't fly!
    Globals::col->calculateY(mNode, false, false, 0, QF_LEVELGEOMETRY);
}
```

Basically, we undo a move if we collide with something in our new position. We also make the Player stick to the surface of the LevelGeometry below him.

Build and run, and you should see it work. The collision detection may not be up to the mark, but its enough for our NGF tutorial. You could improve it on your own as an assignment. :-)

Right now, we just have one level. To make an application with more than one level, we can use the NGF Worlds system.

4. Adding Worlds

Create a new file, 'include/Level.h', and copy the following code into it:-

```
class Level : public NGF::World
{
protected:
    String mLevel;

public:
    Level(String levelName)
        : mLevel(levelName)
    {
    }

    ~Level()
    {
    }

    void init()
    {
    }

    void tick(const Ogre::FrameEvent &evt)
    {
    }

    void stop()
    {
    }
};
```

We inherit from World. World is usually used to represent 'scenes' such as the levels in a game, the menu screens etc. It's a pretty simple class, with the 'init()', 'tick()' and 'stop()' events. Worlds are constructed in the beginning and given to the WorldManager. The WorldManager simply runs them in order, calling the 'init', 'tick' and 'stop' events for the Worlds at the appropriate times. For example, when control shifts from one World to another, the World currently running gets a 'stop' event and the next one gets its 'init' event. While its running, it gets the 'tick' event every frame. It runs them in serial order, but you're free to jump from a World to any other World avoiding the order.

The Level takes the name of the level in its constructor, and remembers it. When gets the 'init' event, it will have to load up the level. So, put this in Level::init():-

```
Globals::load->loadLevel(mLevel);
```

Similarly, it'll have to clean up the place in the 'stop' event. So, in Level::stop(), we'd have:-

```
Globals::gom->destroyAll();
```

For this tutorial, we'll have it go to the previous level with the 'P' key, and the next one in with the 'N' key. For this, add the following code before '**return true**;' in 'NGFExampleFrameListener::keyPressed()':-

```
switch (arg.key)
{
case OIS::KC_N:
    Globals::wom->nextWorld();
    break;
```

```
case OIS::KC_P:
    Globals::wom->previousWorld();
    break;
}
```

Since we're now using Worlds and the WorldManager, we can handle the shutting down through them. Modify the last part of 'NGFExampleFrameListener::frameStarted()' to look like this:-

```
//Update NGF stuff. Shutdown if F12 key down.
Globals::gom->tick(false, evt);

if (mKeyboard->isKeyDown(OIS::KC_F12))
    Globals::wom->shutdown();

return Globals::wom->tick(evt);
```

The only thing left now is to actually add the Levels to the WorldManager, and tell it to start running the Worlds. Replace the level-loading lines in 'NGFExampleApplication::createScene()' with the following:-

```
//Add the worlds.
Globals::wom->addWorld(new Level("TestLevel"));

//Start running the Worlds.
Globals::wom->start(0);
```

You can just 'new' the Levels right here, and not need to 'delete' the pointers manually. The WorldManager does that. We tell the WorldManager to run the Worlds, starting with World 0.

Compile and run again, and you won't notice any difference. This is because we still have actually made only one Level. When you try pressing 'N', it'll shutdown cleanly, not crash.

Now, all we you have to do is make more levels in Blender, export them, and add the Levels to the WorldManager, passing each one the name of the level as in the .ngf file, in the constructor. You'll notice that there is no correspondence between levels in the .ngf files, and Worlds. .ngf files are simply scripts describing a scene, loaded with Loading::Loader. Worlds are ways to easily divide the game into sequential 'scenes'. NGF does not assume any connection between them.

To add new levels in Blender, select 'ADD NEW' from the 'SCE:' drop-down box at the top of the window (where you edited the name of the level before). Then, click 'Empty', to make a new, empty level ('scene' in Blender). 'Full Copy' creates a new level that's a copy of the current one.

Edit the name of the level, and create it the same way you made the first level. Overwrite the same 'Levels.ngf' file (the .ngf exporter exports all the scenes, not just the current one, so you can have many levels within a same .blend file, which keeps brush and material names unique), and export the meshes and materials for the brushes. Remember to use a **separate** material file for each scene! Otherwise, you'll overwrite the materials from the other scenes. Only the .ngf file is common for a .blend file, the .mesh and .materials are different (a .mesh for each

mesh, and a .material for each scene). You might be able to work out a different arrangement to suit you though.

If you open the resultant 'Levels.ngf' file in a text editor, you'll see two 'ngflevels'. Do the same for each of your levels, run the game again, and you should be able to move forward and backward through the levels with 'N' and 'P'.

5. Things To Try

Here are some things to try to improve your knowledge of NGF. You may need to check out the NGF code or the manual to see the features NGF provides that are not covered in this tutorial.

- Improve the collision detection mechanism.
- Add a 'finish' object that takes you to the next level when you touch it. Don't call 'WorldManager::nextWorld()' within a method in a GameObject, it'll have to destroy itself.
- Try to integrate a physics engine such as OgreODE or OgreBullet.
- Add a 'Light' object, which creates a point light. Use the properties features to allow setting the lights properties from Blender.