Contents

1 Ronan's OpenMASK Tutorial

This document is a brief hands-on tutorial on installing OpenMASK and programming simple synthetic characters as OpenMASK simulation objects.

2 Installing OpenMASK

Prerequisites

Although it is possible to install and use OpenMASK with gcc-2.95, we will here assume that you have gcc-3.2 installed. We will also assume that your machine runs a Debian GNU/Linux (more precisely, I run a *testing* Debian).

Performer

Download the OpenGL Shader and OpenGL Performer 3.0 (Demo) files from http://www.sgi.com/products/evaluation/Linux_opengl_shader_3.0/ and http://www.sgi.com/products/evaluation/Linux_performer_3.0/ .

You should fetch the following files and save them in a temporary directory:

```
SGIshader_eoe-3.0.i386.rpm
SGIshader_dev-3.0.i386.rpm

performer_eoe-3.0.0_gcc3-0.tgz
performer_dev-3.0.0_gcc3-0.tgz
performer_demos-3.0.0_gcc3-0.tgz
performer_demo_license-3.0.0_gcc3-0.tgz
performer_docs-3.0.0_gcc3-0.tgz
performer_docs-print-3.0.0_gcc3-0.tgz
performer2.4_compat-3.0.0_gcc3-0.tgz
performer2.5_compat-3.0.0_gcc3-0.tgz
```

Change to root, and convert these files to .deb using `alien`:

```
alien SGIshader_eoe-3.0.i386.rpm
alien SGIshader_dev-3.0.i386.rpm

alien performer_eoe-3.0.0_gcc3-0.tgz
alien performer_dev-3.0.0_gcc3-0.tgz
alien performer_demos-3.0.0_gcc3-0.tgz
alien performer_demo_license-3.0.0_gcc3-0.tgz
alien performer_docs-3.0.0_gcc3-0.tgz
alien performer_docs-print-3.0.0_gcc3-0.tgz
alien performer2.4_compat-3.0.0_gcc3-0.tgz
alien performer2.5_compat-3.0.0_gcc3-0.tgz
```

You can then install them with `dpkg`:

```
dpkg -i *performer*.deb *shader*.deb
```

Open Inventor

To avoid problems with Open Inventor, it can be necessary to compile it yourself:

```
apt-get source inventor --compile
dpkg -i *inventor*.deb
```

You can make sure that everything went well for Performer and Open Inventor by running perfly on an .iv file:

```
perfly /usr/share/Performer/data/tux.iv
```

Other Necessary Software

Install PVM, PCCTS and Doxygen:

```
apt-get install pvm pccts doxygen
```

Optionally,if you plan on compiling the QT contributions and examples, install qt-dev:

```
apt-get install qt-dev
```

Compiling OpenMASK

Downloading It

Change back to an ordinary user id, and get the OpenMASK 3.1 for gcc-3.2 release tarball from http://www.openmask.org .

Save it in a directory you have write access to, say /scratch. Uncompress it:

```
cd /scratch
tar xvzf OpenMASK3.1-linux-gcc3.2-performer2.5.2.tar.gz
```

Environment Variables

Before compiling, set the following environment variables (assuming tcsh):

```
# the directory to which OpenMASK got uncompressed
setenv OpenMASKDIR /scratch/linux-gcc3.2
```

```
setenv PCCTSDIR /usr/include/pccts
setenv PCCTSBINDIR /usr/bin

setenv COMPILER gcc-3.2
```

And only if you plan on compiling the QT contributions and examples:

```
setenv QTDIR /usr/share/qt
```

Launching Compilation

Just change to the directory where OpenMASK uncompressed, and launch the `compile` script:

```
cd /scratch/linux-gcc3.2
./compile
```

Possible Problems

You might encounter problems compiling the QT contributions. If they refuse to compile, make sure that in their directory there exists directories named `.moc` and `.ui`. If they don't exist, create them and run `make`.

Also, the `extractDoc` script in /scratch/linux-gcc3.2 (or whatever is the OpenMASK location) might contain a hardcoded path. Correct that path (possibly using the $OpenMASKDIR environment variable), and run ./extractDoc to create the Doxygen documentation.

3 Programming a Character

A Static Character

We try now to have a 3D character show up in an OpenMASK window. For this we need three basic things: a `main` function which will initialise OpenMASK, read the simulation tree and start the simulation, a file describing the said simulation tree, and a class for our character.

main()

We will write our `main` function in a file of its own: *main.cxx*. Its basic structure will be the following:

```
// the simulation tree loader
#include <SimpleOpenMASK3Loader.h>
#include <PsController.h>

#include <PsKernelObjectAbstractFactory.h>
```

```
#include <PsvInteractive3DVis.h>
#include <PsCameraman.h>
#include <PsKeyboardNavigator.h>

#include <PsException.h>

#include "Camille.h"

int main( int argc, char* argv[] )
{
    // construction of a simulation tree for root
    PsObjectDescriptor* simTree;

    if (argc >= 2)
    {
            // the first argument to camille is the simulation tree file
            simTree = (new SimpleOpenMASK3Loader(argv[1]))->getRootObjectDescrip
    }
    else
    {
            cerr << "usage: camille <simtree>" << endl;
            exit(1);
    }

    // we create the simulation controller
    PsController* controller = new PsController ( *simTree, 0 ) ;

    // next come the managers for objects in the simulation tree

    // visualisation
    controller->addInstanceCreator( "PsvInteractive3DVis",
                                    new PsSimpleSimulatedObjectCreator<PsvInterac
    // camera
    controller->addInstanceCreator ("PsCameraman",
                                    new PsSimpleSimulatedObjectCreator<PsCamerama
    // keyboard navigation
    controller->addInstanceCreator ("PsKeyboardNavigator",
                                    new PsSimpleSimulatedObjectCreator<PsKeyboard
    // and our own object: Camille
    controller->addInstanceCreator ("Camille",
                                    new PsSimpleSimulatedObjectCreator<Camille> (

    // finally we initialise and run the simulation
    try
    {
```

```
                controller->init();
                controller->run();
        }
        catch (PsException& e)
        {
                cerr << "Uncaught exception: " << e << endl;
        }

        // and we exit (possibly using Performer's exit function, which frees the gr
        if (Psv3DVis::callPfExit)
                pfExit();
}
```

Character

Let's call our character *Camille*, and give her a class of her own. She has to inherit two classes: PsSimulatedObject, which is the base for any simulated object of OpenMASK, and PsvMechanismPartner, which allows her to be displayed. In *Camille.h*, we write:

```
#include <PsSimulatedObject.h>
#include <PsvMechanismPartner.h>
#include <PsOutput.h>
#include <PsTranslationYRotation.h>

class Camille :
  public PsSimulatedObject,
  public PsvMechanismPartner
{
  public:
     Camille(PsController& ctrl, const PsObjectDescriptor& objectDescriptor)
     virtual ~Camille(void);
};
```

In *Camille.cxx*, our constructor just calls PsSimulatedObject and PsvMechanismPartner's constructors:

```
#include <PsSimulatedObject.h>
#include <PsvMechanismPartner.h>

#include <PsController.h>
#include <PsObjectDescriptor.h>

#include "Camille.h"
```

```
Camille::Camille(PsController& ctrl, const PsObjectDescriptor& objectDescrip
    PsSimulatedObject(ctrl, objectDescriptor),
    PsvMechanismPartner()
{
}

Camille::~Camille(void)
{
}
```

Simulation Tree

The simulation tree describes – in a format special to OpenMASK – all the objects you want to use in the simulation. This includes your own objects (eg, our character), a visualisation object that will allow to display them, and objects to manage interaction with the user (for instance, to allow navigating in the 3D world with the keyboard).

Makefile

There are two concerns when compiling our program. The first is to specify the right path for all the OpenMASK include files and library. The second is to link with the right libraries.

The OpenMASK include files are in the following locations:

```
$(OpenMASKDIR)/kernel/inc
$(OpenMASKDIR)/contrib/3DVis/inc
$(OpenMASKDIR)/contrib/interactive3DVis/inc
$(OpenMASKDIR)/contrib/interactive3DVisUtilities/PsCameraman/inc
$(OpenMASKDIR)/contrib/3DVisPartners/inc
$(OpenMASKDIR)/contrib/userTypes/math/inc
$(OpenMASKDIR)/contrib/userTypes/events/inc
$(OpenMASKDIR)/contrib/3DVisInputHandlers/math/inc
$(OpenMASKDIR)/contrib/loaders/simpleOpenMASK3Loader/inc
$(OpenMASKDIR)/contrib/interactive3DVisUtilities/PsKeyboardNavigator/inc
```

The necessary libraries are in

```
$(OpenMASKDIR)/lib
```

and possibly:

```
/usr/X11R6/lib
```

The complete library list you need to link with is:

```
3DVisPartners
3DVisInputHandlers
3DVis
Interactive3DVis
Interactive3DVisUtilities
UserTypes
SimpleOpenMASK3Loader
OpenMASK
pf
pfdu
pfutil
pfui
Xmu
X11
GLU
GL
Xext
```

Making It Move

To allow Camille to move, we have to do three things:

- Implement init() and compute().

  OpenMASK will call init() when initialising your object, and compute() each time your object is scheduled for activation. We will use these methods to setup our object, and update regularly its position.

- Add an output representing Camille's position.

  An **output** is a special slot on an object, to which other objects can connect one of their **inputs**. OpenMASK then maintains the link between the output and the input, making sure that the value read on the input side is the same as the one written on the output side.

  Here we will create an output in Camille that will represent its position, of type `PsTranslationYRotation` (that is, a translation and a rotation around the Y axis).

- Make a call to visualiseOutput().

  The call to visualiseOutput() will create an input in the visualisation object, and connect it to our position output; therefore, after this call, every change Camille makes in her position output will be reflected on the 3D display.

Wrapping it up, *Camille.h* now looks like:

```
#include <PsSimulatedObject.h>
#include <PsvMechanismPartner.h>
```

```
#include <PsTranslationYRotation.h>
#include <PsOutput.h>

class Camille :
      public PsSimulatedObject,
      public PsvMechanismPartner
{
  public:
        Camille(PsController& ctrl, const PsObjectDescriptor& objectDescriptor);
        virtual ~Camille();

        virtual void init();
        virtual void compute();

  private:
        double theta;
        PsOutput<PsTranslationYRotation>& positionOutput;
};
```

We just added the methods init() and compute(), and the private member `positionOutput`, which is a reference to an output of type `PsTranslationYRotation`. We also added a member named `theta`, which we will use to move Camille around.

Camille.cxx becomes:

```
#include <PsSimulatedObject.h>
#include <PsvMechanismPartner.h>

#include <PsController.h>
#include <PsObjectDescriptor.h>

#include <PsvTranslationYRotationInputHandler.h>

#include <cmath>
using namespace std;

#include "Camille.h"

Camille::Camille(PsController& ctrl, const PsObjectDescriptor& objectDescriptor)
    PsSimulatedObject(ctrl, objectDescriptor),
    PsvMechanismPartner(),
    theta(0.),
    positionOutput(addOutput<PsTranslationYRotation>("position"))
{
    // initialise the position output
    positionOutput.set(PsTranslationYRotation(0.,0.,0.,0.));
```

```
    // and make it visualised
    visualiseOutput<PsvTranslationYRotationInputHandler>(positionOutput, "DCS_pos
}

Camille::~Camille(void)
{
}

void Camille::init()
{
}

void Camille::compute()
{
    // increment the angle
    theta += 0.5;

    // angle in radians
    double theta_rad = theta * M_PI / 180.;

    // update the position
    positionOutput.set
        (PsTranslationYRotation
         (4. * cos(theta_rad),
          0.,
          4. * sin(theta_rad),
          -theta - 90));
}
```

Notice here how we initialised the output. We used the function addOutput(), which creates an output of type `PsTranslationYRotation`, that OpenMASK will know under the name "position". Inputs and outputs all have a name, which can be used to connect them later (we won't use that name in this tutorial).

Then, we set a first value for the output, and call visualiseOutput() on it. The argument "DCS_position" is the name of a transform node in the geometry used to display Camille.

We kept our init() method empty, as we do not have any more initialisation to do (like connecting to other objects). We probably should have put the call to visualiseOutput() in the init() method, but as of OpenMASK 3.1, it makes the application crash, so we put it in the constructor.

Our compute() method updates the position of Camille, so as her to travel around a circle. For this, we use the method set() on our position output. The four arguments to the constructor PsTranslationYRotation() represent the X,Y,Z coordinates and the rotation around the Y axis.

Working With OpenGL Performer

In OpenMASK 3.1, the visualisation is built on top of OpenGL Performer. This means that OpenGL Performer is the one loading your geometry files, and that it keeps a scene graph used to display your objects. It can be useful to access Performer functions that work efficiently on this scene graph.

For instance, we could be willing to use Performer's facilities to calculate intersections of segments with objects in the scene, for the sake of implementing virtual proximeters.

The OpenGL Performer Scene Graph can be accessed in three different ways:

- The PsvPartner interface, ancestor of all visualisable objects keeps a pointer on a subgraph of the OpenGL Performer scene graph (you should note that the partner is usually responsible for creating this scene graph). New descendants of PsvPartner can be created in order to suit the user needs, for example the PsvMechanismPartner used in *Camille* allows to load geometry files. To implement our proximeters, we could create a new descendant of PsvPartner pointing to the scene graph root (this behaviour would be implemented by redefining the initPartnerNode() method), add methods that work on Performer's representations and call them inside the simulated object inheriting of the new descendant of PsvPartner we just created.

- The InputHandlers are small components dedicated to animate a scene graph node by interpreting a simulated output value, they must be instantiated using the method `visualiseOutput<InputHandlerType>()` of a PsvPartner. A number of InputHandlers are distributed in the directory 3DVisInputHandlers of $OpenMASKDIR/contrib. New inputHandlers should be created by the user to implement its own animations or special effects.

- Inheriting from the visualisation simulation object and directly manipulating its pointer on the root node of the scene graph, obviously it works but it will be most painful to integrate these manipulations in third-party applications. In the first way, we will have to add a mechanism for communication between our *proximeters* simulation object inheriting from the new PsvPartner and the objects who need proximeters; this will probably take the form of inputs and outputs, and possibly events or signals.

4 Resources

Official OpenMASK website: http://www.openmask.org .

Official OpenGL Performer home: http://www.sgi.com/software/performer/ .

OpenGL Performer manuals: http://www.sgi.com/software/performer/manuals.html .

Open Inventor home: http://www.sgi.com/software/inventor/ .

Open Inventor manuals: http://www.sgi.com/software/inventor/manuals.html .

PVM home: http://www.csm.ornl.gov/pvm/pvm_home.html .

PCCTS home: http://www.polhode.com/pccts.html .

Doxygen home: http://www.doxygen.org .

5 History

24/02/2003 - Minor corrections.

27/03/2003 - A few simplifications in include dirs and libraries. Last paragraph rewritten by Michal Rouill. Incorporated a few remarks by David Raulo too.

6 Author

Ronan Le Hy (lehy at imag dot fr), INRIA Rhne-Alpes, Cybermove/Laplace project.

This tutorial resides at http://www.inrialpes.fr/sharp/people/lehy/ .

Feel free to send comments and suggestions about this tutorial; however, if you wish to suggest changes, please do so referring to the document in POD format (from which I generate HTML, PDF...). POD is Perl's Plain Old Documentation (see `man perlpod`).

7 Copyright