OpenSceneGraph & VRPN : Tips and Tricks

- Developments -

Date de mise en ligne : Saturday 1 November 2008

bhbn.free.fr

Abstract

This short document explains how to program with OpenSceneGraph in order to control objects or the camera from external tracking. These are however general guidelines which can be used for programming many things. I present how to create an OSG node callback, how to connect to a remove VRPN tracker, and how to create a custom camera manipulator.

Visiting a node during update

Given that you have a ref to a node in the scenegraph;

```
 osg::Node *mynode;
```

You want to move it at every update. For this, you create a NodeCallback which is dedicated to your needs. So let's define the class;

```
 class MyUpdateCallback : public osg::NodeCallback
{
    double last_visit, deltat;
    int last_traversal_number;

  public:

    // Constructor
    PhonemonUpdateCallback(): last_visit (0.0), deltat (0.0),last_traversal_number (0)
    {}

    // Operator () called during update
    virtual void operator() ( osg::Node* node, osg::NodeVisitor* nv );
};
```

The constructor is emtpy (default NodeCallback constructor) but the '()' operator will be defined later. You can now (somewhere where your node is available) associate an instance of this callback to the node;

```
 mynode ->setUpdateCallback ( new MyUpdateCallback() );
```

Well, now, make sure your callback does something;

```
 void MyUpdateCallback::operator() (osg::Node* node, osg::NodeVisitor* nv)
{
    // compute deltat ; only once per traversal!
    if ( nv->getTraversalNumber() != last_traversal_number && nv->getFrameStamp() )
    {
        // update remaining time
        double tick = nv->getFrameStamp()->getSimulationTime();

        // skip the first update,
        // (it could be too big, breaking the simulation)
        if ( last_visit > 0 )
            deltat = tick - last_visit;

        //
        // DO SOMETHING HERE
        //

        // update time for next itteration
        last_visit = tick;
        last_traversal_number = nv->getTraversalNumber();
    }

    // the callback could be associated to more nodes, let it continue its way
    traverse(node,nv);
}
```

Creating a VRPN client to get data from a tracker

First, you need to define a client for a remote tracker;

```
#include <vrpn/vrpn_Tracker.h> vrpn_Tracker_Remote *_tkr;
```

For later, you will need to have a some user data allocated somewhere to put the tracker info into it. We want to use it into OSG so let's create a transformation matrix.

```
osg::Matrixd _mat;
```

The VRPN update will be done by callback function, so you also need to declare this function. Notice that the function is generic, and takes a pointer to void instead of a pointer to your matrix (it is the C way to implement polymorphism). The other argument is passed by VRPN with the actual sensor data.

```
void handle_tracker(void *userdata, const vrpn_TRACKERCB t);
```

Now, you can create your tracker and assign it the matrix and callback function. To instanciate a new vrpn remote tracker, you just give it in argument the string describing the tracker; the string device could be "Tracker0@localhost" if a local server runs with a tracker called "Tracker0" (defined in vrpn.cfg), or in IMI case "TI1@tcp://10.59.64.96" [1] for a remote tracker "TI1" running on the machine whose IP is 10.59.64.96. Multiple clients (and machines) can simultaneously access to the tracker data.

```
 _tkr = new vrpn_Tracker_Remote(device.c_str());
_tkr->register_change_handler(&_mat, handle_tracker);
```

You can now initialise your matrix by reading data from VRPN

```
 // Call the VRPN main loop:
// the callback gets called
_tkr->mainloop();
```

Of course, we have to define what the handle_tracker callback does:

```
 void handle_tracker(void *userdata, const vrpn_TRACKERCB t) {

    osg::Matrixd rmatrix,tmatrix;
    // get the rotation of the tracker into a tmp matrix
    rmatrix.makeRotate(osg::Quat(t.quat[0], t.quat[1], t.quat[2], t.quat[3]));

    // get the translation of the tracker into a tmp matrix
    tmatrix.makeTranslate(t.pos[0], t.pos[1], t.pos[2]);

    // fill in the user data matrix (it is in fact our _mat)
    // with the combination of rotation and translation
    *((osg::Matrixd *)(userdata)) = rmatrix * tmatrix;

}
```

So, now, every time you call _tkr-»mainloop();, the matrix _mat is updated with VRPN sensor data.

For example, you may want to set the matrix of Transform node from a VRPN tracker data: this can be done with an update visitor (see above) applied on a osg::MatrixTransform node; it is in the update visitor that the VRPN mainloop shall be called.

Setting up a custom camera manipulator

In OSG, the camera is not part of the scene graph. Therefore, it cannot be manipulated like a node with a visitor. But of course, there is another and dedicated way to do it.

First, let's check how the camera manipulators are created and assigned to a viewer (in a main file of your OSG program);

```
 // given your viewer is there:
osgViewer::Viewer viewer(arguments);

// Create a 'meta' manipulator which handles key press to change manipulator
osg::ref_ptr<osgGA::KeySwitchMatrixManipulator> ksw = new osgGA::KeySwitchMatrixManipulator;

// Create several camera manipulators.
ksw->addMatrixManipulator( '1', "Trackball", new osgGA::TrackballManipulator() );
ksw->addMatrixManipulator( '2', "Flight", new osgGA::FlightManipulator() );
ksw->addMatrixManipulator( '3', "Drive", new osgGA::DriveManipulator() );
ksw->addMatrixManipulator( '4', "Terrain", new osgGA::TerrainManipulator() );

// attach to the viewer
viewer.setCameraManipulator( ksw.get() );
```

These are the predefined manipulators of OSG. What we need is a custom implementation of one.

Here is an example which I call 'follow manipulator' which can be used to follow a given object in the scene. Whatever happens to the transform node followed is applied to the camera. This is done by inheriting from a matrix manipulator, where we have to implement a set of virtual methods.

```
 #include class FollowManipulator: public osgGA::MatrixManipulator {

    class UnknownTransformType {};

    public:

        FollowManipulator(MatrixTransform *node, osg::Matrixd& offset):
               MatrixManipulator::MatrixManipulator(), _mat(node), _offset(offset)
        { }

        virtual bool handle(const osgGA::GUIEventAdapter&, osgGA::GUIActionAdapter&);
        virtual void setByMatrix(const osg::Matrixd& m);
        virtual osg::Matrixd getMatrix() const;
        virtual void setByInverseMatrix(const osg::Matrixd &m);
        virtual osg::Matrixd getInverseMatrix() const;

    private:

        osg::MatrixTransform *_mat;
        osg::Matrixd _offset;
};
```

As you can see, the FollowManipulator class keeps a reference to a transform node and to local offset matrix; this matrix will give the relative placement of the camera to the object (e.g. translation on the -Z axis to be abble to see the object). The set of methods is to be implemented to allow OSG to use the manipulator.

Now, the more important is the 'getMatrix' method of our matrix manipulator; this is the function called when the camera is checked in OSG.

```
 osg::Matrixd FollowManipulator::getMatrix() const {

    osg::Matrix mat;

    // get the matrix of the object node
    mat = *(_mat->getWorldMatrices().begin());

    // apply the offset
    mat = osg::Matrixd::inverse(_offset) * mat;

    // this is where the camera should be
    return mat;
}
```

The function 'getWorldMatrices' (notice the plurial form) returns a vector of matrices, so we just take the first one. The other functions are pretty straightforward, and often just empty;

```
 // we don't want the user to be able to set the matrix
void FollowManipulator::setByMatrix(const osg::Matrixd& m) {}

void FollowManipulator::setByInverseMatrix(const osg::Matrixd& m) {}

// getting the inverse is easy
osg::Matrixd FollowManipulator::getInverseMatrix() const {
    return osg::Matrix::inverse(getMatrix());
}

// we have nothing to handle here (no device)
bool FollowManipulator::handle(const osgGA::GUIEventAdapter&, osgGA::GUIActionAdapter&) {
    return false;
}
```

So, now, considering you have a pointer to an object node (e.g. osg::MatrixTransform *mynode) you can use you manipulator:

```
 osg::Matrixd m;

m.setTrans(0.0, 0.0, -5.0);

ksw->addMatrixManipulator( '5', "Follow", new osgGA::FollowManipulator(mynode, m) );
```

In this case, it will be activated when you hit the key '5'. To make the last one added the active one, just do:

```
ksw->selectMatrixManipulator(ksw->getNumMatrixManipulators() - 1);
```

---

[1] The address of the server is given with the "tcp://" prefix in order to guarantee that VRPN uses TCP to establish the connection; this is necessary sometimes in order to work inside protected networks (schools, universities), as the usual default approach is fast but unsafe.