Creating an Actuator Part One

The steps covered in part one are:

Creating Your New Class

Setting up a working directory

Before we examine the code, you will need to set up a working directory. This process is very similar to that described in Section 3.2.

- Launch CMake. Set the */CustomActuatorExample* directory as the source code location, and create any directory you wish for the build location.
- Click Configure. Be sure to point the OpenSim installation property to the correct location of your OpenSim 2.0 installation folder. By default, the CMAKE_INSTALL_PREFIX (this flag shows up if you set CMake to show "Advanced View") is set to the same directory as your source code. This will ensure that you will not have to move any associated files to visualize your results in the GUI later on.
- Click Configure again and then click Generate. Then close CMake.

Defining the ControllableSpring class (ControllableSpring.h)

The following instructions will outline ALL the steps for defining the ControllableSpring class. The ControllableSpring class is defined by the file *Controllable Spring.h*. It only contains a partial definition of the class, though. You will need to fill in a few key lines that have been omitted.

At the top of the header file, we include the header for the base class, call the OpenSim namespace, and begin defining the class as a derived class of PistonActuator.

```
#include "PistonActuator.h"

namespace OpenSim {

class ControllableSpring : public PistonActuator {
OpenSim_DECLARE_CONCRETE_OBJECT(ControllableSpring, PistonActuator);
```

Defining properties

Our new actuator will have all of the properties of the PistonActuator class, plus one more for defining the rest length of the spring.

```
protected:
        // Additional Properties specific to a controllable spring need to be defined
        /** rest length of the spring */
        OpenSim_DECLARE_PROPERTY(rest_length, double,
        "rest length of the spring.");
```

The constructors

Next we define the constructor and destructor. The constructor take the same form as the PistonActuator constructors for consistency. The constructor calls the constructProperties method (to be defined later) which initializes some of the basic properties of the class. The default destructor is used.

```
ControllableSpring( std::string aBodyNameA="", std::string aBodyNameB="") :
        PistonActuator(aBodyNameA, aBodyNameB)
{
        constructProperties();
}

/* use the default destructor */
virtual ~ControllableSpring() {};
```

Property Setup

We will define a private member method that are used during construction to initialize the ControllableSpring instance. constructProperties() is used to set up the properties of the ControllableSpring from values read in from an XML file. The only property added in this class is the rest length.

```
/* define private utilities to be used by the constructors. */
private:
void constructProperties()
{
        constructProperty_rest_length(1.0);
}
```

Get and Set methods

Since the rest length was defined as a private member variable, we must define some public methods to get and set its value. The set_... and get_... method calls are automatically generated when you declared the rest_length property above.

```
public:
// REST LENGTH
        void setRestLength(double aLength) { set_rest_length(aLength); };
        double getRestLength() const { return get_rest_length(); };
```

computeForce()

The computeForce() method is the heart of any actuator class. It is called by OpenSim to calculate and apply any loads associated with the actuator. The computeForce() method is defined to be purely virtual in the CustomActuator base class, so any derived classes must define its behavior. PistonActuator has already defined its own implementation of computeForce(), but we will redefine it here so that the ControllableSpring actuator behaves like a spring instead of like an ideal actuator. This method begins by checking that the model and bodies are defined.

```
void computeForce(const SimTK::State& s,
                                              SimTK::Vector_<SimTK::SpatialVec>& bodyForces,
                                              SimTK::Vector& generalizedForces) const
{
  // make sure the model and bodies are instantiated
  if (_model==NULL) return;

  const SimbodyEngine& engine = getModel().getSimbodyEngine();

  if(_bodyA ==NULL || _bodyB ==NULL)
    return;
```

Next, it determines the locations of the application points in both the body and ground frames by doing some transformations. _pointA and _pointB, as well as the bool _pointsAreGlobal, are defined in the PistonActuator base class.

```
  /* store _pointA and _pointB positions in the global frame. If not
  ** alread in the body frame, transform _pointA and _pointB into their
  ** respective body frames. */
  SimTK::Vec3 pointA_inGround, pointB_inGround;
  if (_pointsAreGlobal)
  {
    pointA_inGround = _pointA;
    pointB_inGround = _pointB;
    engine.transformPosition(s, engine.getGroundBody(), _pointA, *_bodyA, _pointA);
    engine.transformPosition(s, engine.getGroundBody(), _pointB, *_bodyB, _pointB);
  }

  else
  {
    engine.transformPosition(s, *_bodyA, _pointA, engine.getGroundBody(), pointA_inGround);
    engine.transformPosition(s, *_bodyB, _pointB, engine.getGroundBody(), pointB_inGround);
  }
```

Now we find the vector pointing from point B to point A expressed in the ground frame and then decompose it into its magnitude and direction.

```
  // find the dirrection along which the actuator applies its force
  SimTK::Vec3 r = pointA_inGround - pointB_inGround;
  SimTK::UnitVec3 direction(r);
  double length = sqrt(~r*r);
```

To compute the magnitude of the force, we first must know the spring stiffness. Since we want stiffness to be the product of optimalForce and the control value, we simply use the computeActuation() method from the base class, which outputs exactly this calculation.

```
  double stiffness = computeActuation(s);
```

Now we find the magnitude of the force from the stiffness and the deflection of the spring. We then form the force vector.

```
  // find the force magnitude and set it. then form the force vector
  double forceMagnitude = (get_rest_length() - length)*stiffness;
  setForce(s, forceMagnitude );
  SimTK::Vec3 force = forceMagnitude*direction;
```

The last operation computeForce() performs is to apply the equal and opposite point forces to the two bodies.

```
  // appy equal and opposite forces to the bodies
  applyForceToPoint(s, *_bodyA, _pointA, force, bodyForces);
  applyForceToPoint(s, *_bodyB, _pointB, -force, bodyForces);
}
```

Finish the class definition and close the namespace

```
//=============================================================================
}; // END of class ControllableSpring
} //Namespace
//=============================================================================
//=============================================================================
```

Next: Creating an Actuator Part Two

Previous: Creating a Customized Actuator

Home: Scripting and Development | Developer's Guide | Adding New Functionality