Design and Implementation of the Robotic Platform

Markus S. Loffler, Vilas K. Chitrakaran, and Darren M. Dawson

Department of Electrical and Computer Engineering, Clemson University, Clemson, SC29631-0915, USA
E-mail: [loffler, cvilas, ddawson]@ces.clemson.edu

***Abstract* – This paper describes the design and implementation of the Robotic Platform, an object-oriented development platform for robotic applications. The Robotic Platform includes servo control, trajectory generation, 3D simulation, a graphical user interface, and a math library. As opposed to distributed solutions, the Robotic Platform implements all these components on a single hardware platform (a standard PC), with a single programming language (C++), and on a single operating system (the QNX Real-Time Platform) while guaranteeing deterministic real-time performance. This design leads to an open architecture that is less complex, easier to use, and easier to extend.**

1. INTRODUCTION

Robot control systems are very demanding with regard to software and hardware performance because their building blocks cover a wide range of disciplines found in robotics and software development (*e.g.*, real-time programming, 3D graphics, trajectory generation, *etc.*). Hence, it is desirable to create a common platform that can be reused by researchers for different applications. Due to the lack of flexibility and performance of proprietary vendor-supplied robot control languages, previous research focused on building robot control libraries on top of a commonly used programming language (*e.g.*, "C") that was executed on a Unix workstation. RCCL [1] and ARCL [2] are examples of such libraries. Even though this approach leads to a higher flexibility and performance, many robot control platforms developed in the 80's and early 90's were inherently complex due to the limitations of software packages and hardware components of that time. That is, most operating systems did not support real-time programming (fostering projects like RCI [1] and Chimera [3]). In addition, procedural programming languages like "C" tend to reach their limits with regard to reusability for complex projects; furthermore, the limited performance of hardware components forced system developers to utilize distributed architectures that integrated a mix of proprietary hardware and software.

Over the last ten years, many innovations have occurred in the computing area. Specifically, the advent of object-oriented software design [4] facilitated the management of more complex projects while also fostering code reuse and flexibility. For example, robot control libraries like RIPE [5], OSCAR [6], and ZERO++ [7] utilized object-oriented techniques in robot programming. We have also witnessed the proliferation of real-time Unix-like operating systems for the PC [8], which facilitate the replacement of proprietary hardware components for real-time control [9]. In the hardware sector, we have witnessed the advent of high-speed, low-cost PCs, fast 3D graphics video boards, and inexpensive motion control cards. Consequently, the PC platform now provides versatile functionality, and hence, makes complex software architectures and proprietary hardware components superfluous in most cases. The QMotor Robotic Toolkit (QMotor RTK) [10], for example, integrates real-time manipulator control and the graphical user interface (GUI) all on a single PC platform.

Despite the extensive functionality of the PC platform, much of the research in robot control software utilizes distributed architectures [5-7]. Besides the obvious advantages of distributed systems (*e.g.*, more computational power), there are several disadvantages. Specifically, a distributed architecture increases the complexity of the software significantly. Additionally, hard real-time behavior over network connections often requires expensive proprietary hardware. Generally, the overall hardware cost is higher and users have to familiarize themselves with different hardware architectures and operating systems. Even though many platforms developed in the last couple of years attempted to be extensible, these platforms are seldom used and reused. Apparently, engineers consider it easier to develop their applications from scratch. Indeed, from our own experience, the learning curve of installing, learning, and modifying previous robot control platforms is steep.

Given the above remarks, the Robotic Platform is the first platform that has been designed to integrate servo control loops, trajectory generation, task level programs, GUI programs, and 3D simulation in a homogeneous software architecture. That is, only one hardware platform (the PC), one operating system (the QNX Real-Time Platform [8]), and one programming language (C++ [11]) are used. This type of architecture has the following advantages:

Simplicity. A homogeneous non-distributed architecture is much smaller and simpler than a distributed inhomogeneous architecture. It is easier to install, easier to understand, and easier to extend.

Flexibility at all Levels. Every component of the Robotic Platform is open for extensions and modifications. Many past platforms have utilized an open architecture at some levels, but other levels (*e.g.*, the servo control) had been implemented on proprietary hardware such that they could not be modified.

Cost. The Robotic Platform requires fewer hardware components than a distributed platform. Basically, a single PC with one or more input/output (I/O) boards is sufficient. Additionally, PC hardware is very cost effective.

2. POWERFUL TOOLS AND TECHNOLOGIES – THE BASIS FOR THE ROBOTIC PLATFORM

To reduce development effort and complexity, the Robotic Platform is based on general-purpose tools and technologies.

PC Technology. While in the past only expensive UNIX workstations provided the processing power necessary to control robotic systems, the PC has caught up or even exceeded the performance of workstations [9]. Compared to UNIX workstations, a PC based system allows for a greater variety of hardware and software components. Additionally, these components and the PC itself are usually cheaper than their UNIX counterparts.

The QNX Real-Time Platform. The QNX Real-Time Platform (RTP) by QSSL [8] consists of the QNX6/Neutrino operating system and additional components for development and multimedia. QNX6 is an advanced real-time operating system that provides a modern microkernel-based architecture, a POSIX compliant programming interface, self-hosted development, 3D graphics capabilities and an easy device driver architecture. The RTP is also very cost-effective as it is free for non-commercial use and runs on low-cost standard PCs.

Object-Oriented Programming in C++. With regard to developing robot control software, object-oriented programming has several benefits over procedural programming. First, it provides language constructs that allow for a much easier programming interface. For example, a matrix multiplication can be expressed by a simple "`*`", similar to MATLAB programming. Second, object-oriented programming allows for a system architecture that is very flexible but yet simple. That is, the components (classes) of the system can have a built-in default behavior and default settings. The programmer can utilize this default behavior to reduce the code size or override it for specific applications. Finally, object-oriented programming supports generic programming, which facilitates the development of components that are independent from a specific implementation (*e.g.*, a generic class "`Manipulator`" will work with different manipulator types). All of the above benefits are based on the general concepts of object-oriented programming: i) abstraction, ii) encapsulation, iii) polymorphism, and iv) inheritance [4, 10]. The language of choice is C++, as it provides the whole spectrum of object-oriented concepts while maintaining high performance [11].

Open Inventor. Open Inventor [12], developed by Silicon Graphics, is an object-oriented C++ library for creating and animating 3D graphics. Open Inventor minimizes development effort, as it is able to load 3D models that are created in the Virtual Reality Modeling Language (VRML) format. A variety of software packages are available that facilitate the construction of 3D VRML models that represent robotic components. The Robotic Platform also utilizes the functionality of Open Inventor to animate these components.

The QMotor System. Implementation of control strategies requires the capability to establish a deterministic real-time control loop, to log data, to tune control parameters, and to plot signals. For this purpose, the graphical control environment QMotor [13] is used for the Robotic Platform.

3. Overview of the Design

Each component of the Robotic Platform (*e.g.*, manipulators, the trajectory generator, *etc.*) is modeled by a C++ class. A C++ class definition combines the data and the functions related to that component. For example, the class "`Puma560`" contains the data of a Puma 560 robot (*e.g.*, the current joint position) as well as functions related to the Puma (*e.g.*, enabling of the arm power) [10]. Hence, the design of the Robotic Platform results from grouping data and functions in a number of classes in a meaningful and intuitive way. A class can use parts of the functionality and the data of another class (called the *base class*) by deriving this class from the base class. This process is called inheritance, and it attributes heavily to code reuse and eliminates redundancy in the system. To extend the system, the programmer creates new classes. Usually, new classes will be derived from one of the already existing classes to minimize coding effort.

The classes of the Robotic Platform include GUI components and 3D modeling for graphical simulation. These types of components were traditionally found in separate programs (*e.g.*, see the RCCL robot simulator [1]). However, by including them in the same class, we can achieve a tight integration of the GUI, 3D modeling, and other functional parts. Additionally, system extensions automatically include GUI components and 3D modeling.

The main class hierarchy diagram of the Robotic Platform is shown in Figure 1. Each arrow is drawn from the derived class to the parent class; hence, the further to the left a class is listed, the more generic it is. The classes of the Robotic Platform can be separated into the following categories:

The Core Classes. The classes `RoboticObject`, `FunctionalObject`, and `PhysicalObject` build the basis of all robotic objects. The classes `RoboticPlatform` and `ObjectManager` contain functionality for overall management of robot control programs.

Generic Robotic Classes. Derived from the core classes are a number of generic robotic classes. These classes cannot be instantiated. Rather, these classes serve as base classes that implement common functionality while also presenting a generic interface to the programmer (*i.e.*, these classes can be used to create programs that are independent from the specific hardware or the specific algorithm).

Specific Robotic Classes. Derived from the generic robotic classes are classes that implement a specific piece of hardware (*e.g.*, the class `Puma560` implements the Puma 560 robot) or a specific functional component (*e.g.*, the class `DefaultPositionControl` implements a proportional integral derivative (PID) position control).

The ControlProgram Class. This class is part of the QMotor system. It is the basis for all real-time control loops. Classes that require a real-time control loop are derived from the `ControlProgram` class.

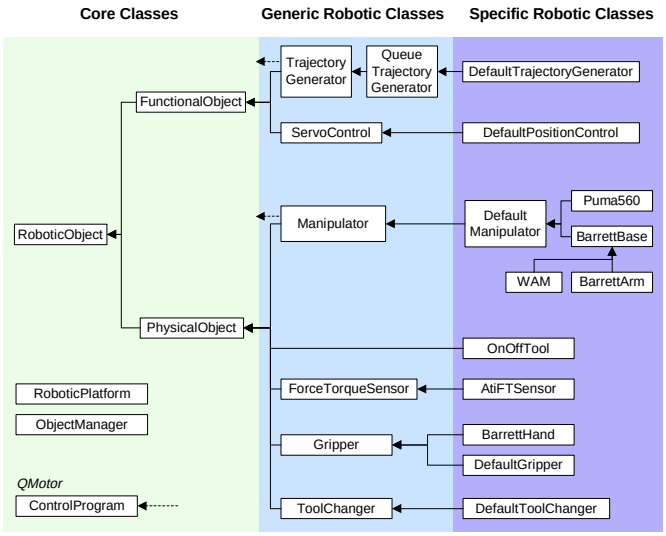


Figure 1. Class Hierarchy of the Robotic Platform

In addition to the classes shown in Figure 1, the Robotic Platform provides the classes of the math library, the manipulator model classes, and several utility classes. These classes and their class hierarchy will be described later in this paper.

In a robot control program, the programmer instantiates objects from classes. The programmer can instantiate as many objects as desired from the same class. For example, it is straightforward to operate two Puma robots by simply creating two objects of the class `Puma560`. As soon as objects are created, the programmer can employ their functionality. The *Object Manager* (see Figure 2) maintains a list of all currently instantiated objects. With the object manager, it is possible to initiate functionality on multiple objects (*e.g.*, to shutdown all objects). The *Scene Viewer* is the default GUI of the Robotic Platform. It contains windows to view the 3D scene of the robotic work cell and a list of all objects. The *QMotor GUI* can be optionally utilized for data logging, plotting and control parameter tuning.

In a robotic system, different components are related to each other. To reflect this fact, object relationships are established between objects. For example, objects can specify their physical connection to each other. Object relationships are implemented by C++ pointers to the related object. The object relationships are indicated by arrows in Figure 2.

The Robotic Platform utilizes a global configuration file to specify the system's configuration. For each object, the configuration file lists the object name in brackets, the class name of the object, and the object settings (see Figure 3).

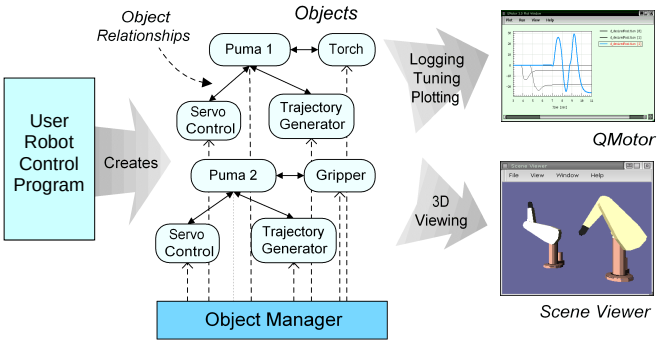


Figure 2. Run-Time Architecture of the Robotic Platform

```
[leader]
class Puma560
position 0 0 0

[follower]
class BarrettArm
simulationMode on

[gripper]
class BarrettHand
port /dev/ser1
```

Figure 3. An Example Global Configuration File

4. The Core Classes

The class `RoboticObject` is the base class for all robotic classes. It defines a generic interface (*i.e.*, a set of functions that can be used with all robotic classes of the Robotic Platform). For example, a program can use the `createControlPanel()` function to tell an object of either the class `Puma560` or the class `Gripper` to display the appropriate control panel. Specifically, the class `RoboticObject` defines i) error handling, ii) object name handling, iii) configuration management, iv) object shutdown, v) the control panel, and vi) thread management. Note that the actual functionality is usually implemented in the derived class. However, the class `RoboticObject` also implements simple default functionality.

The class `PhysicalObject` is derived from the class `RoboticObject`. It is the base class for all classes that represent physical objects (*e.g.*, manipulators, sensors, grippers, *etc.*). Specifically, the class `PhysicalObject` defines the following generic functionality:

- *3D Visualization*. Every physical object can provide its Open Inventor 3D model. The Scene Viewer loops through all physical objects to create the entire 3D scene.
- *Object Connections*. A physical object can specify another object as a mounting location. By using this object relationship, the Scene Viewer is able to draw objects at the right location (*e.g.*, the gripper being mounted on the end-effector of the manipulator).
- *Position and Orientation*. The programmer can set the absolute location of the object in the work cell (or the mounting location, if an object connection is specified).
- *Simulation Mode*. Every physical object can be locked into simulation mode. That is, the object does not perform any hardware I/O; instead, its behavior is simulated.

The class `FunctionalObject` currently does not contain any functionality. It is only added as a symmetric counterpart to the class `PhysicalObject`. Functional robotic classes like the class `TrajectoryGenerator` are derived from the class `FunctionalObject`.

5. Classes Related to Manipulators

The central components of any robotic work cell are manipulators. The class `Manipulator` is a generic class that defines common functionality of manipulators with any number of joints. Derived from the class `Manipulator` is the class `DefaultManipulator`, which contains the default implementation for open-architecture manipulators. Open-architecture manipulators provide access to the current joint position and the control torque/force of the manipulator and hence, allow for custom servo control algorithms. Derived from the class `DefaultManipulator` are the classes that implement specific manipulator types. Currently, two manipulators are supported: the Puma 560 robot and the Barrett Whole Arm Manipulator (WAM) in both the 4-link and 7-link configuration. More information about the specific control implementation of these robot manipulators can be found in [10].

The class `DefaultManipulator` reads the current joint position and outputs the control signal continuously in a QMotor control loop. The actual calculation of the servo control algorithm is contained in a separate servo control object. The class of this object is derived from the class `ServoControl`, which defines the interface of a servo control. The default servo control is defined in the class `DefaultPositionControl`, which implements a PID position control with friction compensation. Manipulator classes like `Puma560` or `WAM` automatically instantiate an object of the class `DefaultPositionControl` for the convenience of the programmer. However, the programmer can switch to a different servo control anytime.

For the simulation of the manipulators, their dynamic model is required. Additionally, for Cartesian motion, forward/inverse kinematics and the calculation of the Jacobian matrix are needed.

All these functions are located in the `ManipulatorModel` classes. The class hierarchy of the `ManipulatorModel` classes is displayed in Figure 4.

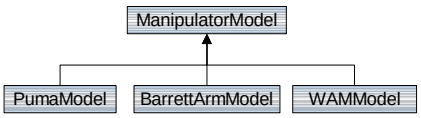


Figure 4. The ManipulatorModel Classes

The trajectory generation is also performed in separate classes. The class `TrajectoryGenerator` defines the interface of a generic trajectory generator. A trajectory generator is any object that creates a continuous stream of setpoints and provides this stream to a manipulator. The manipulator calls the `getCurrentSetpoint()` function of the trajectory generator to determine the current desired position. It is also possible to switch between multiple trajectory generators. The class `QueueTrajectoryGenerator`, which is derived from the class `TrajectoryGenerator`, is a generic interface of a trajectory generator that creates the trajectory along via and target points. The class `DefaultTrajectoryGenerator`, which is derived from `QueueTrajectoryGenerator`, is the specific implementation of a trajectory generator that interpolates both in joint space and Cartesian space, including path blending between two motion segments at the via points.

6. The End-Effector Classes

Several robotic classes refer to end-effectors, as given below:

Gripper Classes. The class `Gripper` is the generic interface class of a gripper. It defines the functions `open()`, `close()`, and `relax()`. The class `DefaultGripper` utilizes two digital output lines to control the gripper, one digital line to open the gripper, and one to close it. The class `BarrettHand` is used to operate the BarrettHand.

Force/Torque Sensor Classes. The generic base class `ForceTorqueSensor` defines the interface of a force/torque sensor. That is, it defines functions to read forces and torques. The class `AtiFTSensor` is the implementation of the ATI Gamma 30/100 Force/Torque sensor.

Toolchanger Classes. The class `ToolChanger` is the generic interface class of a toolchanger. It defines the functions `lock()`, `unlock()`, and `relax()`. The class `DefaultToolChanger` uses two digital output lines to control the lock and unlock function of the toolchanger.

7. The Object Manager

The class `ObjectManager` implements the object manager. Every time a new object is instantiated in the user's robot control program, the object registers itself with the object manager. Similarly, every time an object is destroyed, it is removed from the object list that is maintained by the object manager. The object manager contains functionality to loop through this list to perform operations on multiple objects. For example, the Scene Viewer retrieves a list of all objects that are derived from the class `PhysicalObject` to render each of them, and thereby, is able to render the entire 3D scene.

The functionality of the object manager is also necessary to allow for generic code. Generic code operates any object (*e.g.*, a manipulator object of class `Puma`) through the appropriate interface class (*e.g.*, the class `Manipulator`) by using C++ virtual functions. Hence, generic code does not need to be changed when an object of a different class is used (e.g., the class `WAM`), as long as this object is derived from the same interface class. Generic code is very useful for code-reuse (*e.g.*, only a single generic trajectory generator must be written which can be used with different manipulator types). The following excerpt of generic code is manipulator independent code that works with either the Puma robot, the WAM, or any robot that is added in the future.

```
Manipulator *robot;
ObjectManager om;

robot = om.createDerivedObject<Manipulator>("leader");

cout << "Current End-Effector Coordinate Frame: "
     << robot->getEndEffectorTransform();
```

The above code first calls the function `createDerivedObject()` to create an object of either the classes `Puma560`, `BarrettArm`, or `WAM`. Then, it operates this object via a pointer to the generic base class (*i.e.*, `Manipulator *`). In order to create the desired object, the `createDerivedObject()` function looks for the object name in the global configuration file (see Figure 3). Then, it reads the class name of the object from the configuration file and creates an object of this class. To do so, the framework of the Robotic Platform maintains a list of all classes in the program. Hence, to switch to a different manipulator type, only the class name in the global configuration file has to be changed when using a generic program.

8. The Concurrency Model

While it is often sufficient for many software systems to run as a single task, robotic systems require components like the servo control to be executed concurrently with other components (*e.g.*, the trajectory generator). The Robotic Platform runs all concurrent tasks on the same PC, within a single robot control program. This program spawns threads if concurrent execution is required. Once the program terminates, all threads are automatically terminated. Figure 5 shows an example of how a user program spawns multiple threads.

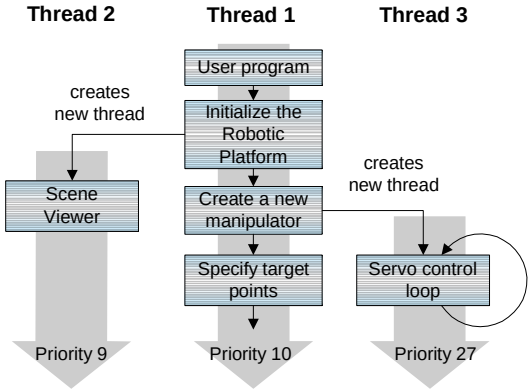


Figure 5. Creating New Threads for Concurrency

At program start, only thread 1 is executing. At the initialization of the Robotic Platform library, a new thread is created that

executes the 3D Scene Viewer. Then, the user's robot control program utilizes a new object of a manipulator class. The creation of this manipulator object automatically spawns a third thread for the servo control loop. Hence, the first thread can go ahead and specify target points for the manipulator, while the servo control loop and the Scene Viewer run in the background. To ensure real-time behavior of time critical tasks, the threads run at different priorities (*e.g.*, the servo control loop runs at the high priority 27). To allow for synchronized communication between the threads, message passing (as provided by the classes `Client` and `Server`) and standard thread synchronization mechanisms are used (as implemented in the classes `Barrier` and `ReaderWriterLock`).

9. Control Implementation with QMotor

QMotor [13] is a complete environment for implementing and tuning control strategies. To implement a real-time control loop, the programmer derives a class from the class `ControlProgram` and reimplements several functions that perform the control calculation and the housekeeping. Once a control program is implemented and compiled, the user can start up the QMotor GUI, load the control program, start it, and tune the control strategy from the control parameter window. Furthermore, the user can open multiple real-time plot windows (see Figure 6) and set logging modes. To utilize QMotor for the Robotic Platform, classes that require a real-time control loop (*e.g.*, `DefaultManipulator`) are derived from QMotor's `ControlProgram` class.

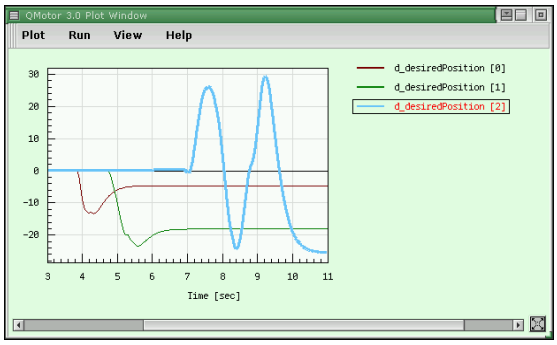


Figure 6. The QMotor Plot Window

10. The Math Library

Previous robot control libraries often introduced their own specific robotic data types. Most of these data types are based on vectors or matrices (*e.g.*, a homogeneous transformation is a 4x4 matrix). Hence, it is more feasible and flexible to use a general C++ matrix library and define robotic types on top of it. Most of the matrix libraries available for C++ use dynamic memory allocation, which risks the loss of deterministic real-time response [10]. To overcome this disadvantage, special real-time matrix classes (see Figure 7) were developed for the Robotic Platform that use templates for the matrix size. Consequently, the matrix size is known at compile time and dynamic memory allocation is not required. The classes `MatrixBase`, `VectorBase`, `Matrix`, `ColumnVector`, `RowVector`, and `Vector` are also parameterized with the data type of the elements. The default element data type is double, which is the standard floating-point data type of the Robotic Platform. The classes `MatrixBase` and `VectorBase` are pure virtual base classes that allow for manipulation of matrices and vectors of an unknown size. Matrices and vectors of an unknown size are required during generic manipulator programming. The class `Transform` implements a homogenous 4x4 matrix, which is typically used to represent coordinate frames in robotic applications. The class `MathException` is used for error-recovery. The example program shown in Figure 8 performs a common task in robotics: Calculating a position equation. This example shows that programming with the math library is very intuitive.

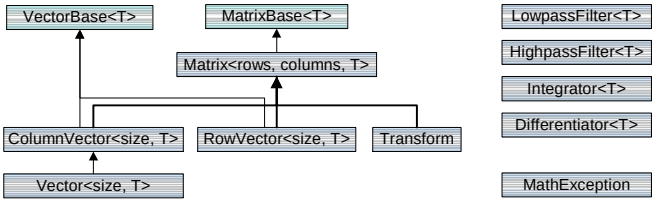
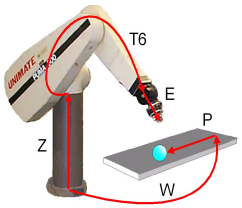


Figure 7. Class Hierarchy of the Math Library




```
Transform Z = translation(0, 0, 0.7);
Transform E = translation(0, 0, 0.1);
Transform W = translation(1, 0.2, 0.3)
              * xRotation(M_PI);
Transform P = translation(-0.5, 0, 0);

// Solve Z*T6*E == W*P
Transform T6;
T6 = inverse(Z) * W * P * inverse(E);
```

Figure 8. Example Program for the Matrix Classes

The math library also provides the classes `LowpassFilter` and `HighpassFilter` for numeric filtering, and the classes `Differentiator` and `Integrator` for numeric differentiation and integration. These classes are parameterized with the data type (*i.e.*, they work with scalars, vectors, and matrices).

11. The Graphical User Interface

Whenever a program of the Robotic Platform is executed, the Scene Viewer window opens up, displaying the 3D scene that contains all objects created in the robot control program (see Figure 9). To assemble the 3D scene, the Scene Viewer loops through all physical objects and obtains their Open Inventor 3D data. Then, the Scene Viewer uses the object connection relationships to display the 3D objects at the right position. Furthermore, the Scene Viewer continuously updates the 3D scene with the current state of all objects (*e.g.*, it uses the current joint position of a manipulator to display the joints in the correct position). Hence, the 3D scene always represents the current state of the hardware (in simulation mode, the simulated state of the hardware is represented). To select the best viewing position, the user can navigate in the 3D scene using the mouse. The user can also open the Object List window, which displays a list of all objects that are currently instantiated by the robot control program, including class name, object name, and object status.

Each object has an individual pop-up menu. This pop-menu appears if the user right clicks on the object in the Scene Viewer rendering area or the Object List window. The pop-up menu has options to select 3D display modes and to open the object's control panel. Additionally, there are menu items that are

specific to the class of the object. For example, a gripper object has additional menu items to open, close, and relax the gripper.

The Robotic Platform provides several GUI utility programs for calibration and testing. Among others, the ***Teachpendant*** allows the user to move the manipulator to desired target position in zero gravity mode, and store these positions in a list. The Teachpendant also utilizes the trajectory generator to move the manipulator back to stored positions.

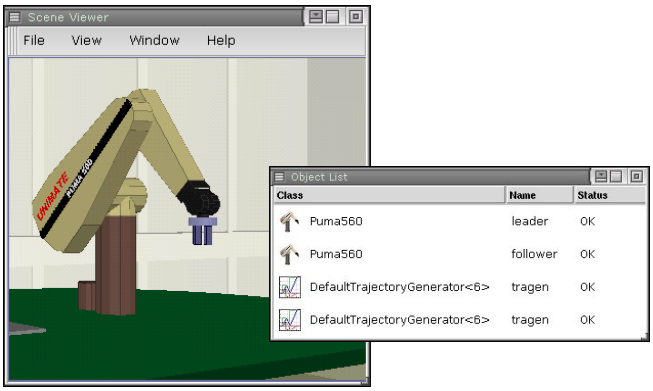


Figure 9. The Scene Viewer and the Object List Window

12. Writing, Compiling, Linking, and Starting Robot Control Programs

A robot control program is first compiled and then linked to the Robotic Platform library. Once the program is compiled and linked, the user can start it from the command line. Figure 10 shows the listing of an example robot control program for a simple pick and place operation.

Every robot control program first calls `RoboticPlatform::init()`. This function initializes the platform and starts up the Scene Viewer. Then, the user's program creates all objects that are required for the robotic task (*i.e.*, a gripper object, a Puma 560 object, and a trajectory generator object are created). The last part of the example program utilizes the trajectory generator object and the gripper object to move the robot to the work piece, close the gripper, pick up the work piece, and drop it at the target position.

```
#include "RoboticPlatform.hpp"

void main(int argc, char *argv[])
{
  RoboticPlatform::init(argc, argv);

  Puma560 puma;
  DefaultGripper gripper;

  DefaultTrajectoryGenerator<6> tragen;
  puma.setTrajectoryGenerator(tragen);
  Transform down = xRotation(M_PI);  // End-effector
                                     // pointing down
  gripper.open();
  tragen.moveTo(translation(0, 0.5, 0) * down);
  tragen.stop(1);
  gripper.close();
  tragen.moveTo(translation(0.5, 0.5, 1) * down);
  tragen.moveTo(translation(1, 1, 0) * down);
  tragen.stop(1);
  gripper.open();
}
```

Figure 10. A Simple Pick and Place Example Program

13. Conclusions

As opposed to past distributed robot control platforms, the Robotic Platform presents a homogeneous, non-distributed object-oriented architecture. That is, based on PC technology and the QNX RTP, all non real-time and real-time components are integrated in a single C++ library. The architecture of the Robotic Platform provides efficient integration and extensibility of devices, control strategies, trajectory generation, and GUI components. The Robotic Platform is built on the QMotor control environment for data logging, control parameter tuning, and real-time plotting. A new, real-time math library simplifies operations and allow for an easy-to-use programming interface. Built-in GUI components like the Scene Viewer and the control panels provide for a comfortable operation of the Robotic Platform and a quick ramp-up-time even for users that are inexperienced in C++ programming.

References

[1] J. Lloyd, M. Parker and R. McClain, "Extending the RCCL Programming Environment to Multiple Robots and Processors", Proc. IEEE Int. Conf. Robotics & Automation, 1988, pp. 465–469.

[2] P. Corke and R. Kirkham, "The ARCL Robot Programming System", Proc. Int. Conf. Robots for Competitive Industries, Brisbane, Australia, pp. 484-493.

[3] D.B. Stewart, D.E. Schmitz, and P.K. Khosla, "CHIMERA II: A Real-Time UNIX-Compatible Multiprocessor Operating System for Sensor-based Control Applications", tech. report CMU-RI-TR-89-24, Robotics Institute, Carnegie Mellon University, September 1989.

[4] B. Stroustrup, "What is 'Object-Oriented Programming'?", Proc. 1st European Software Festival. February, 1991.

[5] D. J. Miller and R. C. Lennox, "An Object-Oriented Environment for Robot System Architectures", IEEE Control Systems February 1991, pp. 14-23.

[6] Chetan Kapoor, "A Reusable Operational Software Architecture for Advanced Robotics", Ph.D. thesis, University of Texas at Austin, December 1996.

[7] C. Pelich & F. M. Wahl, "A Programming Environment for a Multiprocessor-Net Based Robot Control Unit", Proc. 10th Int. Conf. on High Performance Computing, Ottawa, Canada, 1996.

[8] QSSL, Corporate Headquarters, 175 Terence Matthews Crescent, Kanata, Ontario K2M 1W8 Canada, http://qnx.com.

[9] N. Costescu, D. M. Dawson, and M. Loffler, "QMotor 2.0 - A PC Based Real-Time Multitasking Graphical Control Environment", June 1999 IEEE Control Systems Magazine, Vol. 19 Number 3, pp. 68-76.

[10] M. Loffler, D. Dawson, E. Zergeroglu, N. Costescu, "Object-Oriented Techniques in Robot Manipulator Control Software Development", Proc. of the American Control Conference, Arlington, VA, June 2001, to appear.

[11] B. Stroustrup, "An Overview of the C++ Programming Language", Handbook of Object Technology, CRC Press. 1998, ISBN 0-8493-3135-8.

[12] Josie Wernecke, "The Inventor Mentor", Addison-Wesley, ISBN 0-201-62495-8.

[13] N. Costescu, M. Loffler, M. Feemster, and D. Dawson, "QMotor 3.0 – An Object Oriented System for PC Control Program Implementation and Tuning", Proc. of the American Control Conference, Arlington, VA, June 2001, to appear.