The Virtual Vision Lab

Timothy N. Jones

Columbia University

Department of Computer Science

CONTENTS

1. Introduction

1.1. Motivation

1.1.1. Growth of Computer Vision

Ballard and Brown describe computer vision as "the construction of explicit, meaningful descriptions of physical objects from images....Descriptions are a prerequisite for recognizing, manipulating, and thinking about objects."[1] This is a powerful tool when combined with the symbolic processing capabilities of modern computers, and has led to the advancement of many technologies in the areas of robotics, medical diagnostics, and physics, to name but a few. These technological advances have in turn led to strong economic advances, as in the case of industrial manufacturing, where a wide variety of new tasks including visual inspection for quality control, visual feedback for process control, and automated assembly coordinating hand and eye interaction have become possible. The cost of computer vision systems has been steadily dropping, and is expected to continue to do so for the foreseeable future. This trend, along with the fact that research has been steadily progressing, indicates that the use of computer vision systems in manufacturing will continue to grow. Considering the growth of the field and the benefits to society from its byproducts, such as improved manufacturing systems, it is crucial to have many highly trained scientists that fully understand computer vision and are able to apply its technology and continue its advancement.

1.1.2. Immaturity of Teaching Tools

At the beginning of 1994, an NSF sponsored project was started to develop instructional lab modules for teaching computer vision algorithms to college students. The principal investigators for the project, all faculty members at major teaching and research universities, noticed the need for better teaching tools in their computer vision curricula. Because of the rapidly evolving nature of the field, textbooks are often obsolete as soon as they are published: smaller research papers describing individual algorithms are frequently the primary source of information for leading edge instruction. In addition to textbooks, teachers need timely access to research papers as well as software tools to help students better understand the mathematical and imaging processes which form the basis of computer vision algorithms.

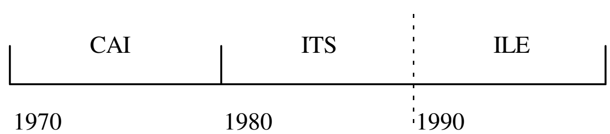
Some generous authors have made source code to implement their algorithms available over the Internet[2], and while this is very useful for experts in the field, it often isn't sufficient for students to be able to pick up and make use of. Tools designed for teaching should augment the implementation of the algorithm with tools to explore input images and output results, tools to interactively adjust the run-time parameters of the algorithm, and hypertext help facilities to provide the student with background and contextual descriptions. Advances in the areas of visualization and multimedia presentation should be fully utilized in state-of-the-art instructional tools to aid students' understanding of the material in any way possible.

1.1.3. Goals for the Project

The ultimate goal of the project is to create hands-on laboratory modules that explain computer vision algorithms in an interactive, realistic environment. Each module will focus on a particular algorithm or class of algorithms, fully described in an on-line copy of a research paper which has been annotated to provide further elucidation. Code that implements the algorithm will also be provided, and will be linked to annotations in the research paper to provide context. Input images and output results will be viewable with image processing tools. Parameters to the algorithm will be tunable in a run-time system executing the code that implements the algorithm. By interacting with the run-time system and integrated tools such as image processing utilities, the student will gain a much deeper understanding of the algorithm's behavior much more quickly than the conventional learning method of simply reading about it.

The first step in the project has been to create the underlying components and interfaces that the higher-level modules will use. This paper describes that work and illustrates its use in the first module for camera calibration algorithms.

1.2. From CAI to ILE — A Brief History of Computer-based Instructional Tools



1.2.1. The First Generation: Computer-Assisted Instruction

Ever since computers first became available to the general public in the timesharing environments of the 1960s, people have been using them for communicating information to one another. Some realized early on that an economic use of this new communications tool was as a replacement for teachers. The first generation of computer-based instructional tools, Computer-Assisted Instruction (CAI) systems, was designed almost as a "mechanical teacher," using the same broadcast-like style of communication of one write-only instructor and many read-only students. This model, characterized by a low amount of interactivity and student control, a high amount of "teaching to," and a fixed order of learning, focused on transferring knowledge to students within a specific domain. The granularity of the instructional material was similar to that of the traditional classroom, i.e., coarse and fixed. An example of this primitive type of system might be a simple database of textual lessons about American history, which students would read on terminals. Students would be presented with multiple-choice questions at the end of sections, of which they would have to answer a certain percentage correctly in order to progress to the next section.

1.2.2. The Second Generation: Intelligent Tutoring Systems

The next stage in the evolution of computer-based instructional tools, Intelligent Tutoring Systems (ITS), coincided with the artificial intelligence boom of the 1980s. These systems attempted to automatically customize lessons for individual students by utilizing models of the students and the knowledge that they were to acquire. Some systems, for example, modeled students as production sets, where individual skills were represented as rules. Once a student acquired the knowledge to fulfill a rule, a skill was earned and the student could progress to higher levels of difficulty. This explicit modeling of students and knowledge was the critical step forward that created the possibility for flexibility in the systems' responses. Many researchers from the computer and cognitive sciences spent years designing and analyzing techniques for collecting knowledge from experts in the domain and structuring it in ITS programs. General recommendations for structuring knowledge include[3]:

- The higher levels of hierarchical information should contain information which is used most frequently or which facilitates retrieval of other related information

- The knowledge unit should contain information that is small enough to be processed, but large enough to make the task of relating the various knowledge units a manageable one by keeping the number of units down.

1.2.3. The Next Generation: Interactive Learning Environments

CAI and ITS systems were successful in meeting their goal of teaching content. The underlying educational philosophy of knowledge transfer, combined with technological models of knowledge representation has been effective in that context and is still used today. The problem is that the knowledge taught to students is rigid and does not allow them to adjust to change in the environment in which the knowledge must be applied. The next step in learning technology is for students to learn how to reason about their environment and adapt their knowledge to fit new situations. Instead of facts and procedures, they need to learn theories and strategies[4]

To meet this goal, Interactive Learning Environments focus on the process of "learning by doing." By immersing the student in active situations, they learn the facts of the domain, but in the context of change from the very beginning. Techniques used to realize this include the facilitation of solo and collaborative work, multimedia presentations, and role-playing. By giving the student maximum freedom of exploration of the knowledge and situations when it is applicable, the non-linearity of true learning can be supported (or at least more closely approximated).

1.3. An Environment for Computer Vision

1.3.1. Importance of Interface Quality

A good human interface is probably the most important criterion for the design of an Interactive Learning Environment — a student must be able to use a program if they are to learn from it. Two things distinguish educational software from general software and highlight the importance of a good human interface[5]. First, the interface provides the entry into the domain for the student; most other software is used by people who are already familiar with the domain being served by the application and can use that to decipher confusing parts of the interface. Second, the interface must provide for variation in the student's skill level. As we have seen, an important component of intelligent teaching tools is a model of the student. Building complete knowledge bases to represent the domain, including the student's misunderstandings about the domain, is too computationally demanding. Menu systems can provide an indirect form of information about the student's version of knowledge, but a well designed system is crucial for this to work. Frye, et. al. discuss some particular interface problems in educational software used to teach arithmetic skills to children[5]:

- Instructional sequences — sometimes it is unclear to the student what to do next
- Miscommunications — sometimes what is shown to the student is misleading or confusing

Systems whose interfaces suffer from problems in these areas can be severely crippled.

1.3.2. Choices for Design

Merrill described the issues that designers have followed in producing instructional systems in the past[6]:

- Selecting the goals — concerning the content, strategic knowledge as well as procedural skills and concepts
- Selecting the strategies — considering the methods that support learning
- Managing the sequence of interaction — based on a decomposition of knowledge that specifies depth and breadth
- Managing the learning — taking into account motivating effects

Some specific analyses of design alternatives for instructional design are available: Psotka, et. al. contribute the following list[8]:

- Didactic presentation of information, versus crafted microworlds that situate the learner in a reactive environment
- Immediate feedback and error correction, versus controlled floundering and supported exploration
- Depth-first versus breadth-first access to knowledge
- Multiple mental models, graduated for the learner, or a single guiding model for all learners

- Relatively abstract perspectives on the knowledge, versus concrete instantiations that can be manipulated directly
- Knowledge structured in simple trees, versus complex networks
- Composition of media: the information loading of graphics, audio, and text

Pirolli and Greeno describe a complete framework for the analysis and study of the design of instruction [7]. They characterize the "problem space" of instructional design by distinguishing nine subspaces, involving different levels and aspects of instruction. There are three levels in their framework:

1. Global — used for designing courses
2. Intermediate — used for designing lessons
3. Local — used for designing presentations and activities

At each level, three types of issues are addressed:

1. Goals and constraints — targets and understood limitations of the instruction
2. Technological resources — provide the means for achieving goals and satisfying constraints
3. Theoretical resources — provide reasons and explanations for decisions that are made in selecting from alternative goals and methods

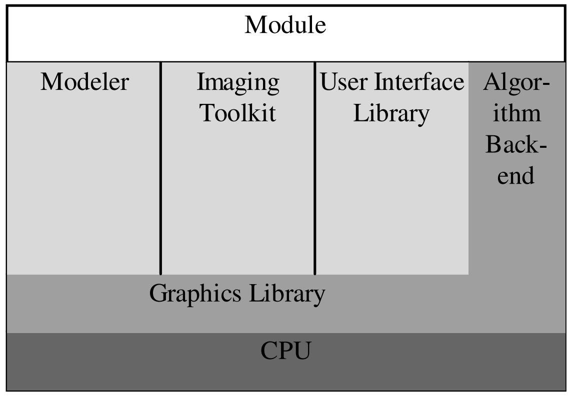
While these collections of checklists tell us what the environment should contain, they do not address the specific attitude of the environment. The power of the Interactive Learning Environment is the interactivity that stimulates the student's creativity and teaches them to apply knowledge under varying conditions — learning by doing generates flexible conceptual models[8]. The diversity of these conditions, many believe, is what provides the student with a full view of all aspects of the domain, and not simply the single view that was used for a particular Instruction. As one of the most influential components of learning is the motivation of the learner, the interactive environment must be compelling.

1.3.3. Environment Components

For the domain of computer vision, an interesting and realistic environment is a robotics manufacturing workcell. Indeed, most computer vision and robotics research takes place in this context in the first place.

The components for the robotics workcell to be represented in the environment are as follows. The main components are the robots, which the student controls to perform some task. The task may be a simple pick-and-place assembly operation (e.g., move widgets from a conveyor belt to a basket). The robots are controlled based upon the output of vision algorithms, which operate on images captured from one or more cameras in the environment. The algorithms operate on a variety of conceptual levels, which allows the student to interact with the system in many different ways and using varying levels of knowledge.

1.4. Conceptual Overview of the Virtual Vision Lab



1.4.1. Description

The structure of the VVL is depicted above. The student interacts directly with a module, which is the glue that ties together the modeler, the imaging toolkit, the user interface library, and an algorithm backend. The modeler, imaging toolkit, and user interface library are themselves built atop a graphics library. The graphics library and algorithm backend interface directly with the underlying system.

The modeler is a geometric CAD modeler which contains physically realistic representations of the objects in the environment (e.g., the robots and the objects to be manipulated). By physically realistic, we mean that the objects have full understanding of their kinematic properties.. For example, the robots are constructed in a hierarchical, object-oriented fashion of links and joints. The physical constraints of the joints are maintained such that the links cannot move to unrealizable locations. The models were built to match the objects in the Robotics Lab at Columbia University.

The imaging toolkit is a set of routines for manipulating images captured from cameras in the real environment. While the student interacts with simulated versions of the robots and other objects through the modeler, they use real images as input to the vision algorithms. This is crucial if the student is going to learn how the algorithms behave under the real-world conditions that they will be subjected to in the field. For example, artifacts such as shadows and specularities are not common in synthetically generated test images, but are quite common in practice. The student must understand the implications of such conditions on the specific algorithms if they are to be able to make intelligent decisions about their use later.

The user interface library contains routines for constructing windows to present and collect information to and from the student. It presents the programmer with high-level graphical user interface (GUI) components to use throughout the module. Examples are pop-up dialog boxes to post error messages, scrolled lists to prompt the student for a choice among many items, and menus.

The algorithm backend is supplied by each module, and implements the algorithm being taught.

1.4.2. Analysis

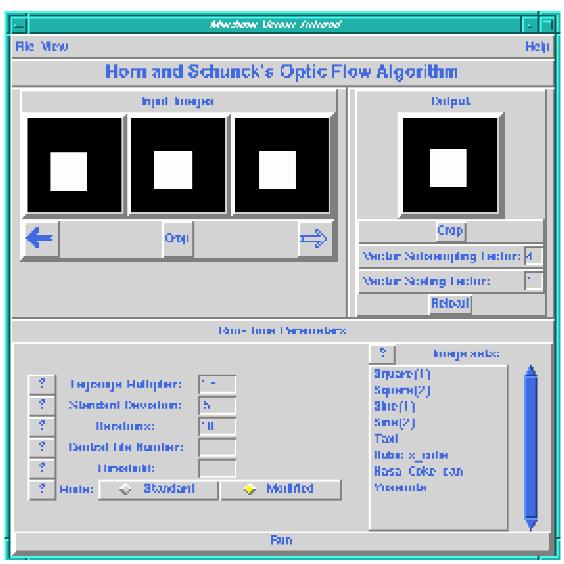
It is useful to consider the characteristics of the VVL in the context of the design points discussed above. The most important criterion, motivation, will be satisfied, we believe, by the framework provided for modules to construct their own scenarios around the environment components such as robots and imaging utilities. By carefully crafting a scenario such that the student is exposed to the proper domain knowledge, the learning process can be structured around a stimulation activity, even playing a game.

The specific goals for content are module specific, and it is up to the individual modules to select the specific strategies for supporting the student's learning. The base system does not provide support for knowledge representation or production systems, though that is something that could be included in the module. We discard the didactic approach to teaching in favor of placing the student in a reactive environment, and the module sets the amount of control imposed on the student's actions. The perspectives on the knowledge applied to the student are practical by the nature of the environment: the robots are distinctly non-abstract and could be noticed immediately in a modern manufacturing environment.

2. History of the Project

2.1. Image-based System

In the beginning, the system was designed to be *image-centric*; that is, the software tools all revolved around the images being processed by the vision algorithm. The student was able to select and examine image sets to feed to the algorithm, adjust the run-time parameters of the algorithm, and note the effect on the output. Image sets were obtained from archive sites, and included real objects such as cars and soda cans as well as synthetic objects such as squares and sine patterns. This design was originally implemented in Tcl/Tk[9], an embeddable, extensible, general purpose language and accompanying library for interfacing with the X window system. Tk contains easy-to-use primitives for constructing graphical user interfaces. Pacco[10], a nice image processing extension for Tk, was soon discovered and integrated into the system.



A module for optic flow algorithms was developed on this platform, based on the freely available implementation of several algorithms by John Barron[11]. Tk was used to provide a forms-like interface for controlling the run-time parameters of the algorithm as well as a menu-based mechanism for selecting images to use as input. A browser was created for the input images and allowed the student to zoom, scale, and crop portions of the image and apply convolution filters, such as edge detectors, to see how the algorithm behaved on the modified image. The browser also overlaid the output from the algorithm (flow vectors) on top of the input imagery to aid the student's evaluation of the output, and allowed the student to subsample the output vectors (useful when there are so many that the output image becomes cluttered) and scale the output vectors (useful when the vectors are very small due to slow movement).

Browsers were also created to allow the student to view the source code of the algorithm implementation and a description of the algorithm containing text, equations, and figures. Finally, a graphical history browser allowed the user to save the state of the tutorial (run-time parameters, input images, output image/vector overlay) for later review. The output data was saved in slots of a grid, so that the user could instantly view the results of the last n tutorial sessions (where n was the size of the grid). Clicking on a grid slot returned the tutorial to the state represented by that slot by loading the proper images and parameter values.

The tutorial contained hooks which allowed it to share information with a running algorithm. This was used to supply additional run-time parameters which depended on some computation by the algorithm of other run-time parameters and report progress information. For example, the Horn algorithm computes a set of spatial intensity gradients in an input image, and the user must select a threshold of these to use in the remaining computation. This was accomplished by the”back-end” algorithm supplying histogram data to the “front-end” GUI, which displayed it to the user in a window and prompted for a threshold selection. Once the selection was made, the GUI passed it to the algorithm and the algorithm continued.

Initial results from this system were encouraging, but performance problems began to appear, particularly with the image operations. At the same time, it became clear that a more exciting learning environment would be necessary to if we wanted the students to understand, in a hands-on manner, how the algorithms really worked and what they were good for. Taking some cues from the systems produced at the Institute for Learning Sciences at Northwestern University[12], we decided that a role-playing scenario would be much more interesting for the student. After a few brainstorming sessions, we decided that a good learning environment would be a simulation of our very own robotics lab; after all, the vast majority of the algorithms we were planning to use were designed to be used in labs like ours.

2.2. Model-based System

The lab contains two Puma 560 robot arms, along with grippers and cameras which may be attached to them. The robots are mounted on each side of a wooden table, which forms the workspace.

2.2.1. Design

The idea behind the simulation is that the student controls the robots through direct commands and the output of a computer vision algorithm to achieve some common task, such as moving a part from one location to another. In order for the student to gain a full understanding of its use and to maximize the learning experience, the vision algorithm should be a central part of the simulation. It should be necessary for the student to perform all of the usual steps, such as data collection and pre-processing, required for the algorithm to operate. The simulation consists of CAD models of the robots, their surrounding environment, such as the table in between them, and the objects that they manipulate.

This simulation was a major addition to the environment, but the images used by the algorithms were still a crucial component. One important design decision was how to integrate the images into the CAD environment. Because we use a sophisticated underlying 3D modeler and renderer, it is possible for us to simulate the *cameras* in the environment as well. This would have the benefit of fitting in smoothly with the other parts of the environment, and would provide a seemless interface to the student. One major problem, however, is that of realism. We stated from the beginning that we wanted the student to be able to observe the performance of the algorithms under adverse conditions, such as shadows and specularities, which they are likely to encounter in the real world. Supporting this in an entirely simulated environment would be difficult, if not impossible, and it was not our goal to do research in the physics-based rendering that would be required for such realistic images. Instead, we decided to use images taken from the cameras in the actual lab and integrate them with the CAD models. To benchmark our integration, we also use simulated images by recording the exact position and orientation of the camera associated with each real image we use, and moving our virtual camera to the same spot in the CAD environment. The rendered image is displayed alongside the real image, and has proven very useful for spotting errors in calibration of our simulator.

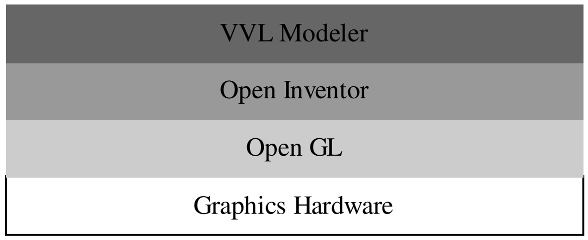
2.2.2. Implementation

This vastly enhanced architecture required us to completely revise the underlying platform, both software and hardware. Tcl/Tk was simply not sufficient for building an interactive, 3D CAD environment. After researching the state-of-the-art platforms for virtual environments[13], we decided that Open Inventor from Silicon Graphics looked like a good choice. Open Inventor is closely integrated with the Motif X toolkit, but is not easily married to Tk, because Tk uses the low-level X library directly. Open Inventor and X each have their own event processing engines, which essentially take over the user's session. If we were to allow both the GUI and the modeler to operate, their event-processing engines had to be combined. It was decided that the easiest way to achieve this was to rewrite the GUI in Motif.

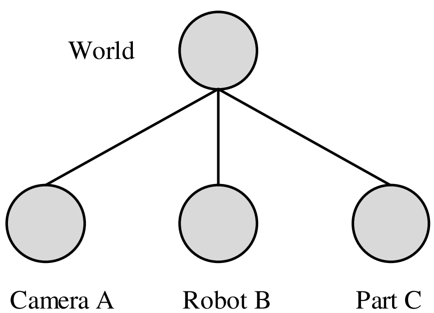
Since we gave up Tcl/Tk and their associated hypermedia tools, the help system and text browsers also had to be rebuilt. It was decided that using the HTML standard[14] would be an efficient way to leverage the functionality of existing hypermedia tools such as world-wide-web browsers. Help documents are written in HTML, and an HTML browser is invoked by the system upon startup. Browser-specific techniques are used to switch the view to specific documents from other parts of the system, such as when the user presses the "help" button in a dialog box.

3. Programmer's Guide

3.1. Modeler



The modeler is a set of C++ objects which represent the physical entities in the lab. It is built on top of the Open Inventor 2.0.1 library from Silicon Graphics[15], which is itself built on top of the Open GL graphics library. The modeler provides an interface to the rest of the system through the header (.h) files



The environment is modeled as a hierarchical organization of objects, rooted at the "world.". Most objects also exist within the Open Inventor layer as nodes in the *scene graph*. The scene graph is a tree structure representing physical objects to be rendered as well as "control" objects that modify the rendering of the other objects (e.g., transforms). There are objects representing robots and cameras, and objects representing the physical entities that the robots interact with (e.g., parts to be manipulated with a gripper). There are also objects representing "virtual" entities that exist to make the simulator easier to use, such as axes which are rendered along side the other objects to provide orientation to the user.

3.1.1. Collection Classes

Objects are housed in “registry” classes, which provide namespaces for referring to the objects. There are predefined registries for robots, cameras, and objects which are neither robots nor cameras.

```
// file: Registry.h
class Registry {
public:
     Registry ();
     ~Registry ();
     void add (void * object, char * name);
     void * get (char * name);
};


// file: Global_Registries.h
extern Registry Robot_Registry;
extern Registry Camera_Registry;
extern Registry Object_Registry;
```

A special class of object is the Environment, which is a “virtual” class* that serves as a container of other objects in a particular environment, say a configuration of the lab for a specific experiment.

* The Environment class is “virtual” in the sense that it does not represent a physical entity; it is not “virtual” in the C++ sense of a purely abstract class.

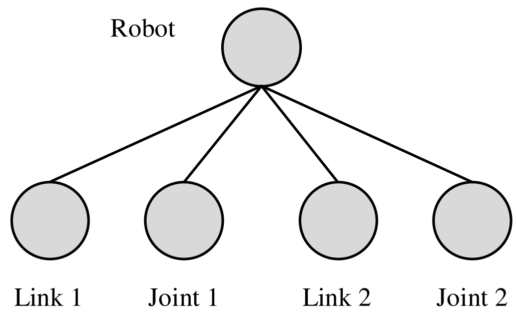
```
// file: Environment.h
class Environment {
public:
     Environment ();
     Environment (const char *datafile);
     Environment (const char *datafile
          const float translation[3],
          const float scaleFactor[3]);
     ~Environment ();
     SoNode *subGraph ();
     void addElement (const char *datafile
          char * name);
     void addElement (SoSeparator *element,
          char * name);
     void transformElement(char * name,
          SoTransformation *transform);
     void hideElement(char * name);
     void showElement(char * name);
     SoGroup * getElement(char * name);
};
```

To create an environment, one would first create an Environment object and add it to a registry (to enable it to be referred to later). The environment would then be populated with objects, and the objects explicitly shown or hidden (to enable or prevent rendering). Objects in the environment are Open Inventor objects, which can be specified either as a pointer to their location in memory or as the name of a file which contains their description. All classes in the Modeler have a **subGraph()** method, which returns a pointer to the location of the corresponding Open Inventor object in memory.

```
#include "Environment.h"
#include "Registry.h"
#include "Global_Registries.h"

Environment *world = new Environment;
Object_Registry.add (world, "world");
world->addElement ("axes.iv", "world axes");
world->showElement ("world axes");
```

3.1.2. Robot Classes



The robots in the environment are represented by C++ classes and serve as a container of sorts for the links and joints which represent the physical components and constraints of the robot.

The RobotLink represents the pieces of the robot — in our case, the various parts of the arm. Its constructor takes a single argument which names the file containing the CAD model representation of the link, specified in the Open Inventor language.

```
// file: RobotLink.h
class RobotLink {
public:
     RobotLink (const char *datafile);
     ~RobotLink ();
     SoSeparator *subGraph ();
};
```

The RobotJoint represents the connection between two links — in our case, a revolute joint (rotation). It is implemented as a set of Open Inventor transformation nodes which are inserted into the rendering queue at the appropriate place. The links which it connects are implied by the order in which it is inserted into the environment. Normally, it will be inserted immediately between two links. The first four arguments to the constructor are the parameters for the Denavit-Hartenberg representation. The "min" and "max" arguments are the minimum and maximum values for theta, which, in our case is the varying "joint angle."

```
// file: RobotJoint.h
class RobotJoint {
public:
     RobotJoint (const float alpha,
          const float theta,
          const float a,
          const float d,
          const float min,
          const float max);
     ~RobotJoint ();
     SoGroup *subGraph ();
     void setTheta(float angle);
     float getTheta();
     SoSFFloat &getThetaRef();
     void monitorTheta(SoSensorCB *func, void *data);
};
```

The Puma560 class ties together the links and joints to create a fully functional kinematic model of the robot.

```
// file: Puma.h
class Puma560 {
public:
     Puma560 (const float theta[NUM_JOINTS],
          const char * name);
     ~Puma560 ();
     SoNode *subGraph ();
     void rotate(float q0, float q1,
          float q2, float q3);
     void rotate(SoTransformation * transform);
     void scale(const float scaleFactor[3]);
     void translate(const float translation[3]);
     void moveCartesian(
          float r11, float r12, float r13,
          float r21, float r22, float r23,
          float r31, float r32, float r33,
          float px, float py, float pz);
     void moveXYZRPY(float x, float y, float z,
          float roll, float pitch, float yaw);
     void moveJointSpace(float j1, float j2,
          float j3, float j4, float j5, float j6);
     void moveRandom();
     void getJointAngles(float &j1, float &j2,
          float &j3, float &j4,float &j5,float &j6);
     SoMFMatrix * getT6();
     void monitorT6();
     void addEndEffector(char *
          end_effector_datafile);
     void addGripper(Gripper * new_gripper);
     Gripper * getGripper();
     void storeIndexed(const CamPosDisplay *
          positions);
     void moveIndexed(const int position);
     const char * get_image_file_name(int position);
     int get_num_camera_stations () const;
};
```

The constructor takes as arguments the initial joint angles and a name for the robot[†].

[†] While the code which creates the Puma560 object makes the decision to add it to a registry (which is the normal case in which a name is needed), the Puma560 class itself needs to be given a unique name to use when registering the dynamic transform representing the position and orientation of the end effector (the T6 transform). This dynamic transform is used to automatically update the end effector attached to the robot whenever the robots configuration changes.

In addition to the joints of the robot, there is a global transform representing the location of the robot within the environment. The **rotate(), scale(),** and **translate()** methods allow that transform to be set.

The **moveCartesian(), moveXYZRPY(),** and **moveJointSpace()** methods allow the joint angles to be set. **moveJointSpace()** sets the angles directly, while the other two methods cause an inverse kinematics routine to be invoked to set the joint angles such that the end effector is placed at a desired location. **moveRandom()** sets the joint angles to random values, and was created for demonstration purposes. To retrieve the current joint angle values, **getJointAngles()** should be used.

Probably the most interesting part of the robot is what it is controlling, be it the location of a camera or the location and status of a manipulator such as a gripper. The T6 transform is the cumulative transform of all joints in the robot, and represents the location of the point at which an end effector attaches to the robot. A 4x4 homogeneous matrix describing the T6 transform may be retrieved at any time using **getT6(),** or, **monitorT6()** can be used once to set up automatic callbacks to be invoked whenever the transform changes (i.e., whenever any joint angle changes). To attach an end effector to the robot, the **addEndEffector()** method can be used. It takes the name of an Open Inventor model file as an argument. A special interface is provided for grippers to be attached to the robot. **addGripper()** attaches a Gripper object, while **getGripper()** retrieves the gripper attached to a robot. These two interfaces distinguish passive effectors (e.g., static models of cameras) from active effectors (e.g., grippers which open and close).

Finally, an interface is provided to store fixed configurations of the robot for later recall. For example, one might want to store several fixed locations of the end effector and switch between them during different stages of a task. This is particularly useful for positioning a camera attached to the robot. Several "camera stations" can be defined by the robot configuration at each, and an image captured from the camera when the robot is in that configuration can be stored. **storeIndexed()** specifies the list of predefined configurations, and **moveIndexed()** commands the robot to put itself in the configuration specified by the indexed element of the list. The number of predefined configurations provided can be retrieved with **get_num_camera_stations()**. The name of a file containing the image associated with a configuration can be retrieved with **get_image_file_name()**. The format of the predefined configurations is described in a later section.

A class is provided to model a simple gripper that may be attached to a robot.

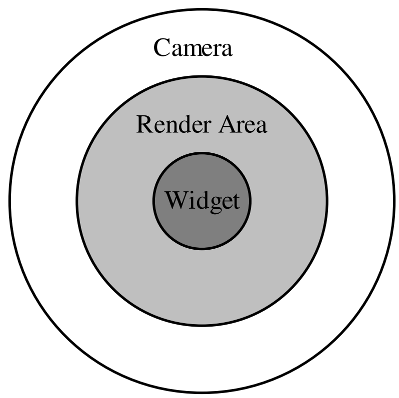
```
// file: Gripper.h
class Gripper {
public:
     Gripper ();
     ~Gripper ();
     SoNode *subGraph ();
     void hold(SoTransformation *transform,
          float width,
          char * file);
     void release();
     void openFinger();
     void closeFinger();
     void setFingerOpening(float width);
     void transform(SoTransformation *transform);
};
```

An object is placed in the gripper with the **hold()** method. A gripper to object transformation is specified, along with the amount of space between the gripper fingers.

The object itself is specified as the name of an Open Inventor data file. To release an object, the **release()** method is used.

The gripper configuration may be set regardless of an object being held with the **openFinger()** and **closeFinger()** methods. An intermediate finger configuration may be specified through the use of the **setFingerOpening()** method, which takes the amount of space between the gripper fingers as an argument.

3.1.3. Camera Classes



Rendering the environment is accomplished through the use of cameras. The rendering takes place inside an Open Inventor render area, which is a wrapper around a Motif/Open GL drawing widget that maps to a window on the screen.

```
// file: Camera.h
class Camera {
public:
     Camera (SoSeparator *scene_arg,
          SoXtRenderArea * ra,
          float headlight_intensity);
     ~Camera ();
     void set_focal(float x, float y, float f);
     void rotate (SoRotationXYZ::Axis axis,
          float angle);
     void translate (float x, float y, float z);
     void transform (const SbMatrix new_transform);
     void transform_track_camera(SoMFMatrix *
          new_transform);
     SoSeparator *get_scene (void);
     Widget getWidget();
     const SbViewportRegion& getViewportRegion();
     SbMatrix *world_to_camera_transform();
};
```

What the camera can see (i.e., what it causes to be rendered), is determined by several factors. First, the camera has a position and orientation within the environment, just like other physical objects. A camera obviously cannot see an object if it is looking in the opposite direction. Second, the camera has a focal length parameter, which is typically fixed and set to match a particular lens being simulated, such as 8.5mm. Objects which are out of the focal range of the camera may still be rendered, but the quality of their representation will suffer. Finally, the underlying Open Inventor camera model uses a tree-based algorithm to determine visibility. If the node representing the camera comes after the node representing an object in the scene graph, the camera will not see the object.

The camera is initialized with a pointer to the root of the environment to be rendered. We always use the world as this value since we want anything physically viewable by the camera to be rendered (i.e., we remove the Open Inventor scene graph ordering constraint). A pointer to an Open Inventor render area is also passed to the constructor. We sometimes use Open Inventor viewer objects (which are derived from render areas) instead, as they allow the user to more easily move the camera and explore the environment. This is particularly useful when debugging the system. Finally, the constructor takes a headlight intensity argument, which is a floating point number between 0 and 1. The camera includes a directional light aimed in the same direction as its orientation to illuminate the objects being rendered. The **set_focal()** method is used to set the focal length of the camera lens.

Once a camera is created, it can be moved with the **translate(), rotate()** and **transform()** methods. To automatically move a camera whenever the robot it is attached to moves, the **transform_track_camera()** method should be used. It is passed a pointer to the transform matrix to track, which may be obtained with the **Puma560->getT6()** method.

Several convenience methods are provided to get pointers to components of the camera for use with other objects. The **get_scene()** method returns a pointer to the root of the environment that the camera is rendering (i.e., the first argument to the constructor). The **getViewportRegion()** method returns a reference to the viewport being used by the render area. It is primarily useful for GUI routines that might want to make a screen dump of a modeler window. Finally, the **world_to_camera_transform()** method returns a pointer to the matrix representing the world to camera transform.

In the section on the robot classes, an interface for using camera stations was described. The stations are implemented as a class which reads a directory for files describing the position and orientation of the camera (i.e., the robot holding the camera) at each station, and files containing images captured at those stations.

```
// file: CamPosDisplay.h
class CamPosDisplay {
public:
     CamPosDisplay (const SbMatrix& T6_to_Lens,
          const char *dirname);
     ~CamPosDisplay ();
     SoSeparator *subGraph ();
     void hide_position(int position) const;
     void show_position(int position) const;
     void show_all() const;
     int get_num_camera_stations () const;
     const double * get_joint(int pos) const;
     const double * get_cartesian(int pos) const;
     const char * get_image_file_name(int pos) const;
};
```

The constructor takes a matrix representing the transform from the robot end effector (T6) to the origin of the camera coordinate system and the name of a directory to search for station files. There are two types of station files: "pose" files and image files. Pose files are named "poseN", where N is the number of the station, and contain the six joint angles and the T6 matrix of the robot in ASCII format. Image files are named "imageN.xpm" and contain the image taken from the camera in XPM format.

Each camera station is represented in the environment as a small icon of a camera correctly positioned and oriented. The **hide_position()** and **show_position()** icons can be used to turn off the icons or turn them back on again. **show_all()** turns on all icons (which is the default state).

The number of stations may be obtained with **get_num_camera_stations().**

The joint angles associated with a camera station can be obtained with the **get_joint()** method The components of the T6 matrix associated with a camera station can be obtained with the **get_cartesian()** method. The image file name associate with a camera station can be obtained with the **get_image_file_name()** method.

3.2. Imaging Toolkit

The imaging toolkit is based upon the freely available XPM library[16], which is a set of routines for reading and writing image files in programs which use the X window system. The main routine of interest is **XpmReadFileToImage(),** which reads a pixmap file in the XPM format into an **XImage** structure in memory. This structure is subsequently used by the **XPutImage()** routine, part of the core X11 library, to display the image on the screen. Individual pixels in the image can be read or written with the **XPutPixel()** and **XGetPixel()** routines, also part of the X11 library. Callbacks can be enabled for the window in which the image is displayed to trap events such as mouse clicks. The **XtAddCallback()** and **XtRemoveCallback()** routines, part of the higher-level Xt toolkit, are used to manage the callback lists.

3.3. User Interface Toolkit

The graphical user interface for the system is based upon the Motif standard. In addition to the main Xm Motif library, the higher-level Xmt library[17] is used as a convenient source of widgets and rapid prototyping tools. Most of the structure of the interface is specified in ASCII X resource files, which are read by the libraries at run-time. Making a change in the interface can usually be accomplished simply by editing the resource files and restarting the program, which avoids the time-consuming recompilation typically required.

In addition to the vast array of routines provided by the X11, Xt, Xm, and Xmt libraries, a few custom routines are provided as an interface to the system's particular design.

```
// file: gui.h
extern void guiMsgPut (const char *msg);
extern void guiMsgAppend (const char *msg);
extern void guiMsgClear (void);
 extern int guiMsgGetInt (int *I);
extern char * guiMsgGetString (char *s, int len);

extern void guiDialogError (const char *s);
```

A one-line message area appears at the bottom of the main window. Text can be placed in it with the **guiMsgPut()** or **guiMsgAppend()** routines. The message area can be cleared with **guiMsgClear().** Integers or character strings can be read from the keyboard (while the mouse focus is in the message area) with **guiMsgGetInt()** and **guiMsgGetString().**

A dialog box displaying a one-line error message can be created with **guiDialogError().** The program will block until the dialog is dismissed by the user.

The help system (considered part of the user interface) has a single routine which is used to load a help document into the external viewer.

```
// file: guiHelp.h
extern void guiHelpLoad (Widget widget,
     XtPointer client, XtPointer call);
```

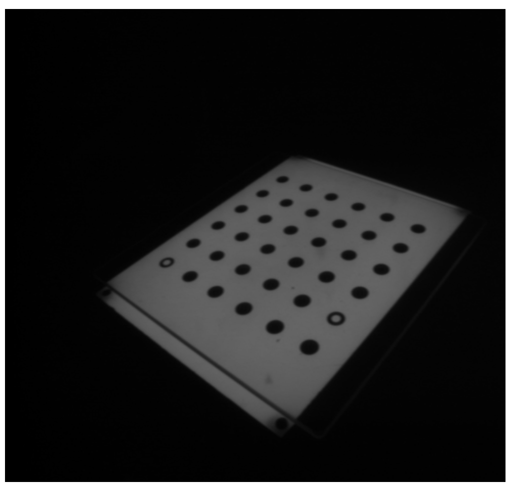
When invoked from a resource file, the widget and call parameters are not used. The client parameter is the name of the help document to be loaded (e.g., "intro.html"). The widget and call parameters are used by the X library, and can be set to 0 (null) when invoked from C/C++ code.

3.4. A Sample Module: Camera Calibration

The tools described above provide the building blocks for tutorial modules, but the "glue" code that ties them all together in a specific order for a particular algorithm must be written by the module author. This section describes the module for camera calibration algorithms in that context.

The module takes the user through the process of calibrating a camera at two camera stations, then using the calibration matrices created by the algorithm to perform a "pick and place" task by identifying two features in each of two 2D images (one taken at each camera station)

A class is added to implement the model for the calibration plate used in the algorithm. The plate is a piece of glass with many precisely placed circles imprinted on it. The centers of these circles (i.e., the centroids), provide the known points in the 3D world which will be matched to the points in the 2D image during the calibration.



```
// file CalibPlate.h
class CalibPlate {
public:
     CalibPlate (int plateSize,
          const char * lightBox_datafile);
     ~CalibPlate ();
     float *centroidCenters();
     SoSeparator *subGraph ();
};
```

The constructor expects a number representing the size of the plate to be modeled. This number is used to calculate the radius of each circle and the space between them. The constructor also expects the name of an Open Inventor data file containing a representation of the light table that the plate rests upon. The light table is important because it provides the illumination necessary to be able to clearly identify the circles.

To obtain the 3D coordinates of the centroids of the circles in the plate, the **centroidCenters()** method is used. It returns an array of floats in the order X1,Y1,Z1,X2,Y2,Z2, etc.

Whne the calibration module is invoked through the "load module" dialog, the calibration-specific objects (i.e., the calibration plate and icons representing the position and orientation of the camera stations) are added to the CAD environment.

```
// add the calibration objects
Environment *world =
     (Environment *) Object_Registry.get ("world");

 CalibPlate * calib_plate =
     new CalibPlate(4,"light.table.iv");
Object_Registry.add(calib_plate,"CalibPlate");
world->addElement(calib_plate->subGraph(),
     "calib_plate");

SoTranslation *plate_transform = new SoTranslation;
plate_transform->translation.setValue(
     -1.3125,0,12.7875);
world->transformElement("calib_plate",plate_transform);

CamPosDisplay * cam_pos_display =
     new CamPosDisplay(Mount_transform_matrix,
     "../../real-data");
world->addElement(cam_pos_display->subGraph(),
     "cam_pos_display");
```

Once the calibration objects are in place, the user is prompted for the number of the first station to use. The robot holding the camera is moved there, and the image of the plate taken at that station is displayed in a dialog box.

```
loop:
do {
  sprintf (s,
     "Please select a camera station (1-%d): ",
     max_station);
  guiMsgPut (s);
} while (guiMsgGetInt (&station) == False ||
     (station < 1 || station > max_station));
...
camera_puma->moveIndexed (station);
```

After the image is displayed, the user is prompted to click the mouse in the center of each circle in the calibration plate (in a specific order). A callback is enabled to trap the mouse clicks, and an Xmt routine is used to block the program until the image dialog is dismissed by the user. The callback handler computes the centroids of the calibration circles and stores them in a temporary file.

```
XtAddCallback(draw, XmNinputCallback, dots_callback,
     &ImagePoints);
XmtBlock (CalibrateImageDialogWidget,
     &CalibrateImageFlag);
XtRemoveCallback(draw, XmNinputCallback, dots_callback,
     &ImagePoints);
```

After the 36 centroids are computed, the backend algorithm itself is run and the resulting data is saved to another temporary file.

```
// initialize calibration routine
initialize_sony_xc77_matrox_parms();

// read centroid data from temp file
FILE *in_fd;
in_fd = fopen(in_name, "r");
load_cd_data (in_fd, &cd);
fclose(in_fd);

// run the algorithm
coplanar_calibration();

// save the result
FILE *out_fd;
out_fd = fopen(out_name, "w");
dump_cp_cc_data(out_fd, &cp, &cc);
fclose (out_fd);
```

The station selection, centroid identification, and calibration computation steps are repeated for the second camera station.

Once the calibration data for two stations has been obtained, the calibration objects are hidden in the environment and the pick and place objects are inserted.

```
world->hideElement ("calib_plate");
world->hideElement ("cam_pos_display");

world->addElement("landscape.iv","landscape");
SoTranslation * paptranslate = new SoTranslation;
paptranslate->translation.setValue(1.0625,0,11.1625);
world->transformElement("landscape",paptranslate);

world->addElement("vehicle.iv","vehicle");
paptranslate->translation.setValue(-5.375,0,15.1);
world->transformElement("vehicle",paptranslate);

world->addElement("rocket.iv","rocket_on_car");
SoRotationXYZ *paprotate = new SoRotationXYZ;
paprotate->axis = SoRotationXYZ::X;
paprotate->angle.setValue(-M_PI/2);
world->transformElement("rocket_on_car",paprotate);
world->transformElement("rocket_on_car",paptranslate);
paptranslate->translation.setValue(0,1.8,2.0625);
world->transformElement("rocket_on_car",paptranslate);
```

Next, the robot holding the camera is moved to the first station, the image of the pick and place objects from that station is displayed, and the user is prompted to click on the pick feature within the image. A routine is invoked to block until the user clicks within the image, and the coordinates of the click are stored in parameters.

```
Puma560 *c_puma =
     (Puma560 *) Robot_Registry.get ("camera_puma");
guiMsgPut ("Moving robot to first camera station...");
c_puma->moveIndexed (CalibrationStations[0]);
...
pickShowImage (name,
     "Please click on the rocket feature point");
pickWaitForClick (pickUserFeatures[0][0],
     pickUserFeatures[0][1]);
```

Image coordinates are similarly obtained for the pick point from the second camera station, and for the place point from both camera stations.

Once the four sets of image coordinates (pick and place points from two stations) have been obtained, they are used along with the two calibration matrices to triangulate to the two points in 3D world coordinates. Finally, the robot holding the manipulator is sent along a path to grasp the object at the pick position, move to the place position, and release the object.

4. Acknowledgments

Pat Crosby and Phil McCoog implemented a large part of the system and also contributed to the design. Professor Peter Allen provided excellent resources for the project, both material and intellectual.

5. References

[1] Ballard, D. H. and Brown, C. M. (1982). *Computer Vision*. Englewood Cliffs, NJ: Prentice-Hall.

[2] http://www.cs.cmu.edu/~cil/v-source.html.

[3] Eylon, B. and Reif, F. (1984). Effects of Knowledge Organization on Task Performance. *Cognition and Instruction*, 1, 5-44.

[4] Soloway, E and Bielaczyc, K. (1994) Interactive Learning Environments: Where They've Come From & Where They're Going. Tutorial presented at CHI 1994, Boston.

[5] Frye, D. Littman, D., and Soloway, E. The Next Wave of Problems in ITS: Confronting the "User Issues" of Interface Design and System Evaluation. In Psotka, J., Massey, L.D., and Mutter, S.A. (eds.), *Intelligent Tutoring Systems: Lessons Learned*. Hillsdale, NJ: Lawrence Erlbaum Associates.

[6] Merrill, M.D. (1987). Prescriptions for an Authoring System. *Journal of Computer-Based Instruction*, 14, 1-10.

[7] Pirolli, P. and Greeno, J. (1988). The Problem Space of Instructional Design. In Psotka, J., Massey, L.D., and Mutter, S.A. (eds.), *Intelligent Tutoring Systems: Lessons Learned*. Hillsdale, NJ: Lawrence Erlbaum Associates.

[8] Psotka, J., Massey, L.D., and Mutter, S.A. (eds.). (1988). *Intelligent Tutoring Systems: Lessons Learned*. Hillsdale, NJ: Lawrence Erlbaum Associates.

[9] ftp://ftp.cs.berkeley.edu/ucb/tcl.

[10] http://iride.unipv.it/pacco/Index.html

[11] ftp://ftp.csd.uwo.ca/pub/vision.

[12] http://www.ils.nwu.edu/.

[13] http://vrml.wired.com/

[14] http://www.ncsa.uiuc.edu/demoweb/html-primer.html

[15] http://www.sgi.com/Technology/Inventor.html

[16] ftp://ftp.x.org/contrib/libraries/xpm-3.4e.tar.gz

[17] Flanagan, D. (1994). *Motif Tools*. Sebastopol, CA. O'Reilly & Associates.