Hardware Abstraction in a Visual Programming Environment

Rick L. Vinyard, Jr. **Joseph J. Pfeiffer, Jr.** **Bernardo Margolis**
Department of Computer Science
New Mexico State University
Las Cruces, NM, USA
+1 505 646 1605
pfeiffer@cs.nmsu.edu

ABSTRACT
Isaac is a visual programming language for geometric reasoning, intended for the control of mobile robots. A major task in mobile robot control, and hence one of the major components of the language, is the handling and fusing of sensor data. A common framework for abstracting the characteristics of a large number of sensor types has been developed, and is being used in the development of the language.

This paper presents two hierarchies for classification of sensors for mobile robots. The first hierarchy presented is based upon intuitive groupings of sensors with similar physical principles. This hierarchy is most useful in organizing an intuitive visual programming environment.

The second hierarchy relates to the mathematical effects of sensors. At the core of this hierarchy are affine transforms. The three transforms of rotation, scaling and translation can be represented through compositions of matrices. This common property forms a superclass in developing a coherent reusable object oriented library.

Keywords
Visual Languages, Robot, Actuator, Transducer, Affine Transform.

1 INTRODUCTION

A key feature in any successful programming environment is the use of a small set of powerful, intuitive concepts which are used consistently throughout the environment. Limiting the number of concepts the programmer is required to learn in order to make use of the environment makes it possible to learn the environment faster, and be more productive. Additionally, implementation is simplified by using a relatively small set of concepts, creating an environment that is both more robust and easier to maintain.

This project uses visual rules expressing geometric transformations to model sensor inputs, perform geometric reasoning, and manage actuator outputs. By expressing these actions using a common framework, we have created a language that is simultaneously quite simple and yet robust.

In this paper, we focus on the use of affine transformations to model sensor inputs, and the development of an editor permitting the programmer to easily define the characteristics of sensors for use in the environment.

The paper is organized as follows: in Section 2 we will briefly introduce Isaac, the language which is currently undergoing our most active development. An overview of the project as a whole is presented in Section 3. Section 4 briefly introduces a design pattern which forms the basis of object oriented sensor class abstraction. Section 5 provides the methodology of using affine transforms in relation to sensor input. Section 6 outlines the sensor abstraction hierarchy. Section 7 outlines some preliminary conclusions, and describes our directions for future research.

2 ISAAC

Our software development concern is a project called Isaac, which has been in progress at New Mexico State University since early 1998[12]. Briefly, Isaac is a visual language for geometric reasoning, for use in programming autonomous mobile robots.

Mobile Robots

Our particular application domain is that of robots navigating in comparatively unconstrained environments. In contrast to the highly constrained environment of a typical assembly–line stationary robot, a mobile robot is called upon constantly to make decisions based on unexpected inputs, to plan its path through unknown environments, and in general to perform a variety of geometric reasoning tasks. Applications of mobile robots are as prosaic as office mail delivery systems and as exotic as planetary exploration.

Visual Programming Languages

Programming language development may be described as an ongoing effort to find representations of appropriate concepts that are both natural to the human programmer, and precise in order that they may be unambiguously interpreted. In the vast majority of programming languages which have been developed to date, these abstractions have been expressed textually.

An alternative mechanism for defining a programming language is to adapt a visual notation, such as dataflow diagrams, for the purpose. Successful visual languages have been developed based on control flow diagrams[7],

dataflow diagrams[4, 8], and several varieties of rule–based languages [1, 11, 13].

One particular notation which is very well suited for use as a programming language is compass–and–straightedge geometric constructions. This notation is intuitive, well–defined, and has a several–millennia history of use for geometric reasoning. In an application such as mobile robot control, this notation is ideal.

3 PROJECT OVERVIEW

The Isaac environment includes two dimensional geometric environment and robot representations, editors for creating geometric models and rules, and a reasoning backend driven by FuzzyCLIPS[9].

The geometric representation is a two dimensional modeling system using triangulated polygons. The system is hierarchical, allowing the representation of articulated objects. An object is made up of an arbitrary number of triangles, and may also have subobjects. Each triangle is made up of three points.

Objects (including subobjects) have transformation matrices associated with them, so they can be translated or rotated by manipulating the matrices. As an object's hierarchy is traversed, the transformation matrices are multiplied so subobjects are located relative to their parents. This supports articulated objects, in which the subobjects are able to move independently of one another.

Rules take the form of a left hand side describing a situation which can occur, and a right hand side. When the left hand side situation is recognized, the rule is fired.

A rule can cause any or all of the following actions to take place:

1. Objects can be added to the environment.
2. Objects can be removed from the environment.
3. Affine transformations can be applied to objects in the environment.

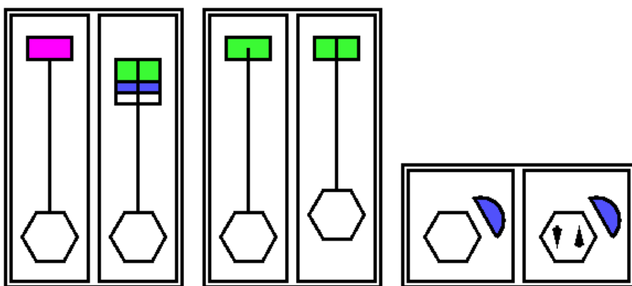

Figure 1: Samples of Isaac Rules

Several examples of Isaac rules are shown in Figure 1. In all rules shown, the hexagon represents a robot, and the boxes represent features in the robot's environment. The first rule represents a rule that might be invoked as the result of a sonar return from an unexpected obstacle. The left side shows the robot and the sonar return; the sonar return is from an area in which no return is expected. As a consequence, a new object is placed in the representation of the environment (these figures are color–coded in the actual environment). In the middle rule, a sonar return from an expected object has an unexpected range. This rule will change the robot's idea of its location. Finally, the last rule in the figure responds to a situation in which the robot finds itself too close to an obstacle. In this case, its motors are set to divert it from the collision.

Isaac rules are parameterized by ''measuring sticks'' inserted in the left and right hand side of the rules. The measuring sticks represent affine transforms to be applied to objects in the rule; for instance, the first two rules shown in Figure 1 are given a translation matrix by the vertical line.

A more complete description of the Isaac language and environment may be found in [12].

Software Organization

In many ways, Isaac is an outgrowth of the earlier Altaira project [11], building on its strengths and attempting to remedy its deficiencies [10]. Early decisions based on lessons learned with Altaira included:

Operating System

The NMSU Department of Computer Science almost exclusively uses Linux. Students are much more likely to have development experience with Linux than other operating systems. Linux had previously been used successfully in Altaira, so it was retained for Isaac.

Language

The nature of the problem, focussing on geometric objects and robot components, lends itself very naturally to an object–oriented approach. Java was regarded as unacceptable due to ongoing performance concerns, so C++ was selected as the language. C++ had also been used successfully in Altaira.

Graphical User Interface

Altaira used Motif, with decidedly mixed results. While the UIL interface description language used by Motif is an excellent, declarative description of the interface, Motif suffers from being excessively rigid in its demands that programmers conform to its recommended look and feel. In addition, Motif's C interface is not appropriate to a program written in C++. Consequently gtkmm (formerly Gtk––) was selected as the user interface. To allow for a change of GUI toolkits at a later date, should that become advisable, a strict model–view–marshaller architecture was implemented for use with the project. Using gtkmm has not been without difficulty, as it is also under development and essentially lacking in documentation.

4 THE MVM (MODEL–VIEW–MARSHALLER) ARCHITECTURE

The Model–View–Marshaller paradigm (MVM) is based upon the concepts of the Model/View/Controller paradigm (MVC) of the SmallTalk80 Graphical User Interface (GUI) system[3].

The Model component is analogous to a data structure. It is responsible for maintaining data values relative to a concrete or theoretical object.

The View component is analogous to a display window. It is responsible for device display of a subset of the data represented by the Model. Most often, the View will comprise a visual display of data on a screen.

The Controller component is responsible for interpretation of user input, and passing of control to the appropriate component. Most modern GUI's (Windows, X, Java) and their associated toolkits(Motif, Gtk+, MFC, JDK, et. al.) provide the functionality of the controller through items such as callbacks or other message handling capabilities. This functionality is closely tied to the View, and is not implemented in the higher abstractions of the MVM.

The focus of the MVM is to provide an abstract layer that may be extended through specialized classes to a variety of specific environments and toolkits.

The MVM architecture is a design pattern that relies heavily upon many other common design patterns. The Model is related to the View through a *subject–observer* pattern [6]. It is implemented in C++ and utilizes the Standard Template Library (STL) for various internal structures. The architecture is shown in Figure 2.

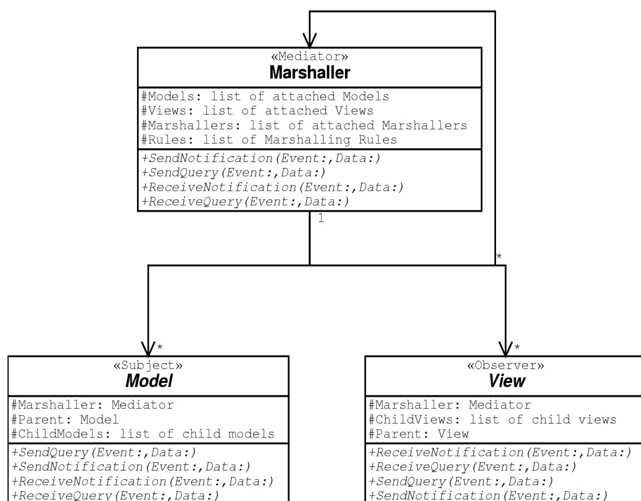

Figure 2: Model–View–Marshaller Design Pattern

A mediator pattern acts as a signal marshaller between a related set of Models and Views. This functionality eliminates the *one to many* relationship between Models and Views and provides a more flexible hierarchy.

Distributed computing is provided through the mediator as well. The mediator is an abstract base class that provides a communication interface to a local Model and View. Since Marshaller–Marshaller communication is also provided, subclasses can be developed for message passing on specific distributed architectures such as CORBA (Common Object Request Broker Architecture) and MPI (Message Passing Interface).

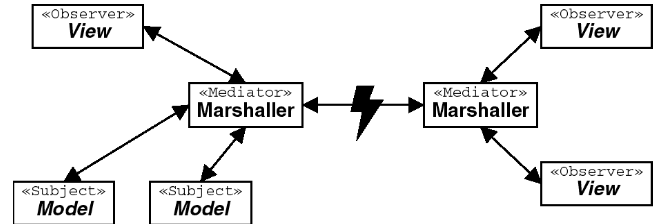

Figure 3: Example of distributed communication

Beyond message passing, the Marshaller provides additional functionality to the MVM architecture. At a base level, it provides a *factory method* for instantiation of Models and Views. Subclasses of Marshaller can provide instantiations of Model and View subclasses.

Another feature the Marshaller provides to the MVM architecture is a *chain of responsibility*. This is most useful when performing queries throughout the hierarchy, or when a selective set of communication rules is desired.

5 SENSOR ABSTRACTIONS IN ISAAC

The most intuitive organization of robotic sensors is related to functional groupings based upon the physical properties of the sensor's operation. Thus, short–range sensors such as local infrared or touch sensors are separated into one group, while other groups may be organized as inertial, rangefinding, interfacing, and so forth. Such a functional grouping, as displayed to the Isaac program, is shown in Figure 4. This is the hierarchy that is presented to a programmer using Isaac for a robotic control task.

For implementation purposes, it is more appropriate to classify sensors according to affine transformation subclasses to increase software re–use.

An affine transform is a geometric transformation which preserves parallelism of lines, though not necessarily lengths or angles[5]. Subclasses of affine transforms include:

- Scaling – which preserves angles
- Rotation – which preserves lengths
- Translation – which preserves both lengths and angles

Affine transforms can be represented as homogeneous matrices, and are easily manipulated through matrix

compositions. Since geometric manipulations constitute a major component of the reasoning performed by Isaac, affine transformations are a logical choice for representing these manipulations. This is true of the reasoning engine present within Isaac, and more specifically of the environmental sensor inputs.

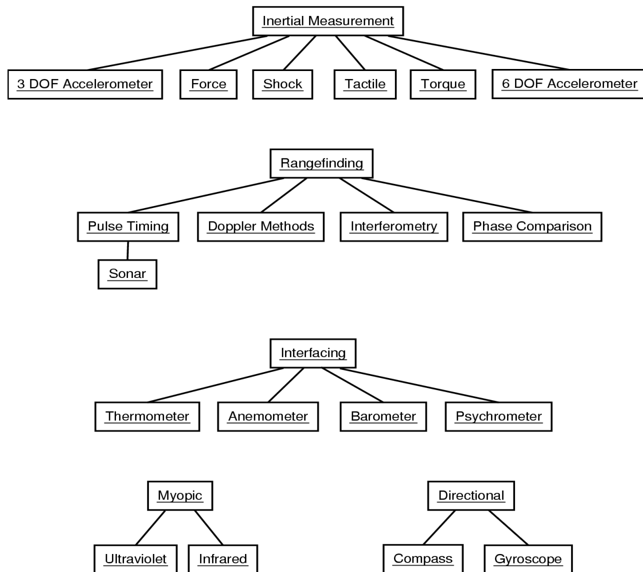

Figure 4: Sensor Classification Based on Functional Groupings

A sensor event is represented as the right–hand side of a rule, placing or modifying objects in the environment and thus enabling rules for firing. The right hand side of a sensor rule is represented by a programmer defined template, and is parameterized by a homogeneous matrix containing entries specified by the sensor inputs..

6 ISAAC SENSOR CLASSES

Isaac maps sensor classes to subclasses of affine transforms based on properties of the sensors. The fundamental sensor classes are:

Identity

An identity sensor is most commonly represented by a simple boolean on–off switch sensor, such as a digital touch sensor. Identity sensors place object(s) at a fixed location in the environment, and do not include any representation of data input strength. No parameterization is required, so an identity transform is applied.

Scalar

Sensors whose output includes a measure of the extent to which a property is present in the environment (such as a temperature sensor) will have a scalar component.

Since Isaac uses the area of an object in its rule enabling to determine the extent to which the corresponding feature is present in the environment, a scalar component is used in the transform to reflect the strength of the sensor's output.

As both boolean and scalar sensors simply identify the presence of an environmental condition, and do not provide information regarding its location in relation to the robot body (except in relation to the fixed sensor location), we frequently refer to sensors in these two classes as ‘‘myopic’’ sensors.

Translation

A sensor whose output is capable of identifying a particular location in relation to the robot, such as a range–finder, will insert an object into the environment at a distance corresponding to the range given by the sensor. This is accomplished by parameterizing a translation matrix.

Rotation

A sensor whose output specifies a particular direction, (such as a compass), will use a rotation matrix to add a fixed–size object a fixed distance from the robot, at an angle to the centerline of the robot determined by the sensor reading.

In addition to the fundamental sensor classes just defined, sensors whose characteristics include aspects of several affine transforms are defined as composite sensor classes. The composite sensor classes are:

Combined Scalar and Translation

Sensors whose outputs express the extent to which some external input is present in the immediate neighborhood of the robot (such as proportional touch sensors or local infra–red sensors) are represented using a combined scalar and translation matrix.

Combined Scalar and Rotation

A combined scalar and rotation transform is used to describe a sensor such as a sound detector using Doppler shift to determine the direction of a sound source. The sound amplitude would be represented using the scalar transform, and the direction using the rotation transform.

Combined Rotation and Translation

A combination of rotation and translation is used to describe sensors such as inertial sensors capable of supplying both rotation and translation information to the robot.

Combined Rotation, Scaling, and Translation

Finally a sensor may have all available components of the transform parameterized.

This classification is summarized in Figure 5.

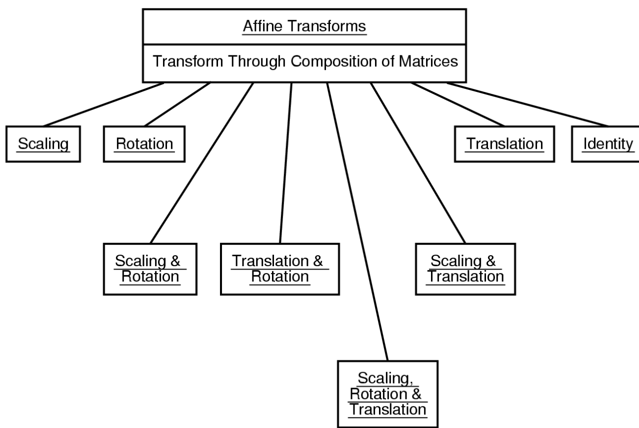

Figure 5: Classification of Sensors by Transformation Type

It must be emphasized that the technology underlying each sensor is abstracted by this classification: a rangefinding sensor may be based on any number of principles, such as pulse timing, parallax, interferometry, or others. All such range–finding sensors are represented in this scheme as either translation or combined scaling/translation sensors, depending on whether the sensor simply reports a range, or also reports a measure of quality or confidence.

7 PRELIMINARY CONCLUSIONS

Visual programming languages provide a very natural mechanism for expressing the geometric reasoning required by mobile robots; however, significant programming challenges are encountered in developing visual environments to perform these reasoning tasks.

Our identification of a small set of geometric transformations which can be used to describe a large number of input sources, and also used in the implementation of the reasoning engine itself, has significantly reduced the amount of code which has needed to be created and allowed for modular support of new sensor types.

Future Work

We are currently investing the representation of a group of robots in terms of the hierarchical geometric model currently defined for a single robot. This will permit us to combine the environment models from all of the robots in the group to provide a single, distributed parallel environment model useable by all of the component robots.

Additionally, the use of a distributed environment provides the robot with the ability to select sensor inputs from devices not physically located on the mobile unit.

Finally, the use of two dimensional representations of affine transforms in Isaac also provide a natural platform for next–generation extensions to the third dimension.

8 REFERENCES

1. Bell, B. and C. Lewis. ‘‘ChemTrains: A Language for Creating Behaving Pictures.’’ in *Proceedings of the 1993 IEEE Symposium on Visual Languages* (1993) 188–195.
2. Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide* (1999).
3. Burbeck, Steve, *Applications Programming in Smalltalk–80: How to use Model–View–Controller (MVC)* (1992)
4. Cox, P. and T. Smedley, ‘‘Prograph: a step towards liberating programming from textual conditioning.’’ in *Proceedings of the 1989 IEEE Workshop on Visual Programming* (1989) 150–156.
5. Foley, James D.; van Dam, Andries; Feiner, Steven K.; Hughes, John F. *Computer Graphics*. Addison–Wesley: Reading, Massachusetts (1997).
6. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley: Reading, Massachusetts (1995).
7. Glinert, E., and S. Tanimoto. ‘‘PICT: An Interactive Graphical Programming Environment.’’ *Computer 17* (November 1984), 7–25.
8. Jagadeesh, J. and Y. Young. ‘‘LabVIEW.’’ Product review, *Computer 26* (February, 1993), 100–101.
9. Orchard, R. A., *FuzzyCLIPS Version 6.04A User's Guide* (1998).
10. Pfeiffer, J. J. Jr., ‘‘Case study: developing a rule–based language for small mobile robots,’’ in *Proceedings of the 1998 IEEE Symposium on Visual Languages* (1998) 144–151.
11. Pfeiffer, J. J. Jr., ‘‘Altaira: a rule–based visual language for small mobile robots,’’ in *Journal of Visual Languages & Computing 9* (1998), 127–150.
12. Pfeiffer, J. J. Jr., ‘‘A language for geometric reasoning in mobile robots,’’ in *Proceedings of the IEEE Symposium on Visual Languages* (Tokyo Japan, September 1999), 164–171.
13. Smith, D. C., A. Cypher, and J. Spohrer. ‘‘KidSim: programming agents without a programming language.’’ in C*ommunications of the ACM 37* (1994) 54–67.