Software Environments for Robot Programming

Bruce MacDonald, Geoffrey Biggs, and Toby Collett

Robotics Group, Department of Electrical and Computer Engineering, University of Auckland, New Zealand

1 Introduction

Many challenges must be met if the robot programming process is to be improved for human programmers; challenges involving the environment, robots and tasks. These challenges, listed in Fig. 1, result from the complex interactions robots have in real environments and the complex sensors and actuators that robots use. They are present in most robotic systems, however the emphasis on interaction with human users in a human environment amplifies the difficulties for programmers of robotic assistants. Some of the challenges are also exhibited by some traditional, nonrobotic applications, but have generally not been addressed by mainstream development tools.

A final weakness in robot development systems is that concurrency is not well handled by debugging tools. In fact, concurrency is not well handled in many generic debugging environments. This weakness is more problematic in robot development because concurrency is more prevalent in robot problems.

The result of attempts to deal with these difficulties has been a set of tools in which much of the software infrastructure is proprietary and much is targeted at specific hardware, additionally robot software development kits may be limiting, there is a lack of open standards to promote collaboration, code reuse and integration, and there is a lack of techniques that integrate the human into the robot's programming infrastructure. Robot researchers typically program robot applications using an ad hoc combination of tools and languages selected from both traditional application development tools and proprietary robotic tools. Each robotics lab develops its own tools for debugging robot programs, often with different tools for each robot in the lab.

We believe robot programming systems must be targeted more closely to robotics, paying attention to the needs of the robot developer and therefore the nature of typical robot programs, the typical skills of robot programmers, the interactions between humans and robots, and the programming constructs that prevail in robotic applications. The approach must be oriented towards

D. Brugali (Ed.): Software Eng. for Experimental Robotics. STAR 30, pp. 107–124, 2007.

The nature of the robot environment

- Environments are dynamic, asynchronous and operate in real time.
- Unexpected variations cause nonrepeatable behaviour and unexpected conditions.

The nature of the robot being debugged

- Robots are mobile. The target hardware moves away from the programmer!
- A large number of devices are used for input, output and storage. The combination is dramatically more than a programmer is presented with by a common desktop computer. Although the robot is in the same world as the human programmer, the robot's sensory and motor channels may not match human programmers' familiar senses and effectors, and can be difficult for the human programmer to comprehend.
- There are wide variations in hardware and interfaces for different robots, as opposed to the highly commoditized and standardized desktop.

The nature of mobile robot tasks

- Robot tasks emphasize geometry and 3D space.
- Complex data types must be represented.
- Some robot tasks cannot be interrupted without disrupting the task. For example, one robot cannot be interrupted for programming and debugging while it is helping another robot carry an object.
- There is simultaneous and unrelated activity on many inputs and outputs, requiring a programmer to manage multiple activities in all parts of a robot program.

Fig. 1. Challenges facing human programmers of robots that operate in typical human environments.

the human users with which robots will interact, and the human and robot environments in which both live and work.

If we examine the full process of programming a robot assistant, there are two fundamental aspects; the toolchain a human uses for developing robot programs, and the underlying technologies used to instantiate both the toolchain and the robotic system that is the end product. The toolchain consists of a number of components, including:

- Design Tools
- Code Entry Tools
- Compilers
- Test and Debug Utilities

The technology components include:

- Robot Platform
- Programming Languages

- Frameworks and API's, which can be broken into:
  - Library functionality
  - Interface definitions
- Middleware

There is considerable work in some areas, but others are neglected. Some tools, in particular Player/Stage [PSP2005], are emerging as de facto standards for the middleware, framework and API since they support a large variety of hardware and have a growing user community. Some test and debugging facilities are provided. However, the full process involving human developers and the immersive robot environment is not so well addressed. Fundamental in the challenges above is that it is the *robot's interaction with the environment* that makes robot programming different and challenging. One aspect is the *programmer's lack of understanding of the robot's world view* that makes robot programs difficult to code and debug. Standard debugging tools give programmers access only to program data. This makes debugging robot programs difficult because program data is at best an indirect representation of the robot and environment. The interactive systems must enable a human programmer to create, modify and examine programs and system resources, both on and off line. The human programmer may also interact with the framework and API, to examine, monitor and configure resources, and directly with robots as they perform tasks.

Another fundamental aspect is the *expressiveness of the language* that the human programmer uses to describe the robot's behaviour. Although the trend in recent years is to a few common, general purpose programming languages, such as C and C++, with advanced application libraries and tools, we expect some of the robotic programming challenges cannot be overcome without improving the expressiveness of the programming language itself.

In this chapter we focus on case studies of improvements in two somewhat neglected areas related to the human programmer; in section 2 the expressiveness of the robot programming language, and in section 3 the need for human–oriented robot debugging tools.

2 Programming Languages and Environments

This section discusses the nature of robot programming, and presents a case study where a programming language is enhanced so that it is able to manage dimensioned robot data. Robot systems must be able to:

- reason about geometric, temporal, and statistical data to help interpret the input received from sensors and determine outputs to actuators
- operate in real time; the world will not wait for the robot
- react with specified timing constraints to both internal and external events

- use distributed resources and concurrency to provide redundancy and simultaneous execution of components such as localisation, navigation, planning, and image recognition.

However existing programming tools for humans are lacking:

- geometric, temporal and statistical sensor data are not natively supported in programming languages
- real time support is difficult for programmers and relies on the scheduling in the underlying operating system
- specifying events and responses requires complex exception triggering and handling code
- support for concurrency is limited to simplistic threading models and distributed system programming relies on 3rd party tools that are complex and difficult.

The programming process can be improved by:

- reducing development times and costs for robot systems
- allowing more powerful and expressive robot programs
- allowing less skilled programmers to create robot programs with less training.

2.1 Case Study: Programming with Dimensional Analysis

Consider the need for robots to reason about dimensioned data. This provides a case study for how languages can be adapted with features specifically required when programming robot software.

Robots operate in the real world, which is rich in dimensioned data, including:

- odometry readings for tracking translations and rotations
- IR, sonar and laser scanner range data
- motion control for mobile robots and limbs
- navigation systems that input geometric data, estimate geometric quantities such as obstacles, and give geometric commands for movement
- manipulation systems that measure and control forces and torques as well as movements
- intelligent controllers that often use geometric data, for example in path plans and maps
- time constraints for real time processes
- kinetic equations for robot motion.

Existing systems do not directly support dimensioned data, instead using fixed or floating point numeric types and an implied standard of what dimensionality is used in programs, such as those shown in Table 1. It is potentially unsafe to rely on programmers to maintain dimensional consistency, especially in larger projects where many programmers are involved. It is difficult

Table 1. Current methods for dealing with geometric data in some common robot programming systems.

System	Units used	Representation
Player [VGH2003, CMG2005]	Metres, radians	double
CARMEN [MRT2003]	Metres, radians	double or integer
Marie [CLMV2004]	Millimetres, degrees	integer
Orca [BKMWO06]	Metres, radians	double
Pyro [BMK2003]	Varies by robot	Python number
KUKA [KU2005]	Varies by use	REAL type
ABB [ABB2005]	Varies by use	NUM type

to change the implied standard; the recent Player/Stage unit change from millimetres to metres generated much discussion and considerable reengineering of clients. Another, well known, incident was the failure of NASA's Mars Climate Orbiter mission; the root cause was data with incorrect units [STE1999]. In addition, the lack of direct support for dimensioned data makes debugging difficult as dimensionality is not immediately clear from the code.

To enhance programmability, dimensioned data in robot programs should appear and behave as programmers expect. It should look like dimensioned data in program code, allow new dimensions and units to be defined as necessary by the programmer, support unit algebra (the combination of different units to create more complex units, for example speed being metres divided by seconds), and above all emphasise simplicity.

These goals can be achieved by adding support for dimensional analysis to the programming system. The dimensional analysis technique can help make programs more robust against errors in units by verifying the consistency of the units given. It allows a new class of errors to be caught [GE1977] and enables automatic conversions between compatible dimensionality. Dimensional analysis systems have been proposed for Ada [GE1985, HI1988, RO1988, GP1993], Pascal [GE1977, AG1984, DMM1986, BA1987], C++ [CG1988, UMR1994, TUO2005, BR2001], and Java [ACLM2004], among others, from 1977 to 2005. These efforts vary considerably in their flexibility, usefulness and clarity of program code. However, despite the potential in many common areas of programming and numerous proposals over the years, dimensional analysis has still not become a feature of the mainstream languages commonly used in robotics. Our work applies and extends some of these techniques for robotic programming systems. We believe an application specific design in robotics may be more successful. The limited domain of robotics, with its smaller number of common dimensions and simple units, allows for a simple system.

Dimensional analysis can be implemented by one of four methods:

- a new category of run time dimension information, in addition to type and sign information

- a preprocessor system that performs dimensional analysis and outputs standard program code
- object oriented extensions for managing dimensioned data
- a new data type specifically for dimensioned data.

An object oriented extension allows for dimensioned data support without the need to change the language itself. However, it does not provide any special syntax for dimensioned data and so does not significantly improve the clarity of dimensioned data in program code. On the other hand, a new data type can use a new syntax to specify dimensioned data directly in program code, improving the clarity and making debugging easier. Such a data type is more strongly integrated into the language than an object oriented approach, meaning it may be supported by more of the language's constructs. For example, in C++ user defined types are not supported in the `switch` statement. A new datatype would, however, require modification of the language, which can be a difficult task.

Both methods can be implemented at compile time only, run time only or a mixture of both, simply by deciding when the dimensional analysis and consistency checking are performed. Compile time checking has the advantage that it does not introduce any inefficiencies into the run time program, and allows for earlier program validation. On the other hand retaining the dimension information for use at run time can provide more flexibility, for example by allowing data to be output with the dimension information or allowing input data to be checked rather than requiring it be in the correct dimension already.

The design presented here uses a new primitive data type aimed at supporting dimensional analysis for robotics. The semantics of the data type are designed in such a way as to be implementable in any object oriented language.

The design considers dimensions to be categories into which units are assigned by compatibility. Each dimensioned value is associated with a unit expression, which is made up of one or more units. For example, `1m` is a value of `1` associated with the unit `m` (metres), and `2.5m/s` is a value of `2.5` associated with the unit expression `m/s` (metres per second), which is in turn made up of the unit `m` divided by the unit `s` (seconds) using unit algebra. The `m` unit falls into the `distance` dimension, while the `s` unit falls into the `time` dimension, meaning that values measured in metres cannot be converted to values in seconds. If a programmer attempts to do so, an error will result. However, conversions between values with equivalent dimensionality will be automatic, for example degrees to radians or `m/s` to `km/h`.

Dimensions and units must be defined before they are used, and cannot be modified. Four dimensions commonly used in robotics are predefined (distance, angle, time and mass) along with several units in each. Each dimension is defined as a name. Units require a name, dimension, a ratio relative to other units in the dimension, the numerical representation used (floating or

fixed point), and any physical range limits. Aliases, for example speed aliasing metres per second, are not allowed, to maintain simplicity and clarity.

A simple formalisation of the semantics is shown below, based on the formalised arithmetic of [Ken96]. Assume the following definitions:

$\mathbb{R}$	set of real numbers
r	member of $\mathbb{R}$
$d(v,u)$	a piece of dimensioned data with numeric value v and unit expression, u
u_a, u_b	unit expressions a and b
u_f	unit expression with floating point representation
u_i	unit expression with fixed point representation
$=_D$	dimensional equivalence relation
$C(d,u)$	conversion operator, convert d to units u
$\times$	operation

The conversion operator can be defined as allowing conversions between units of equivalent dimensionality while disallowing conversions between units of inequivalent dimensionality. The conversion factor between units is used to alter the value as necessary when performing the conversion.

$$C(d(v,u_a),u_b) \rightarrow \begin{cases} d(v,u_b) & \text{if } u_a =_D u_b \\ \text{error} & \text{otherwise} \end{cases}$$

The operations between two pieces of dimensioned data can then be defined as follows.

$$\begin{array}{lll} +,- & d(v_1,u_a) \times d(v_2,u_b) \rightarrow & \begin{cases} d(v_1,u_a) \times C(d(v_2,u_b),u_a) & \text{if } u_a =_D u_b \\ \text{error} & \text{otherwise} \end{cases} \\ & d(v_1,u_a) \times d(v_2,u_a) \rightarrow & d(v_1 \pm v_2, u_a) \\ * & d(v_1,u_a) \times d(v_2,u_b) \rightarrow & d(v_1.v_2, u_a.u_b) \\ \div & d(v_1,u_a) \times d(v_2,u_b) \rightarrow & d(v_1.v_2^{-1}, u_a.u_b^{-1}) \\ <,\leq,==,\geq,> & d(v_1,u_a) \times d(v_2,u_b) \rightarrow & \begin{cases} d(v_1,u_a) \times C(d(v_2,u_b),u_a) & \text{if } u_a =_D u_b \\ \text{error} & \text{otherwise} \end{cases} \\ & d(v_1,u_a) \times d(v_2,u_a) \rightarrow & \text{bool} \end{array}$$

The conversion operator is used in some operators to reduce dimensioned data to the units needed for the simpler form of the operation.

Similarly, the operations between dimensioned data and a numeric value can be defined:

$$\begin{array}{ll} +,- & d(v,u_a) \times r \rightarrow \text{error} \\ * & d(v,u_a) \times r \rightarrow d(r.v,u_a) \\ \div & d(v,u_a) \times r \rightarrow d(r^{-1}.v,u_a) \\ <,\leq,==,\geq,> & d(v,u_a) \times r \rightarrow \text{error} \end{array}$$

Finally, the interaction between floating point and fixed point values is defined, governed by the units used for each piece of dimensioned data.

For all except $*, \div$

$$d(v, u_f) \times d(v, u_i) \rightarrow d(v, u_f)$$
$$d(v, u_i) \times d(v, u_f) \rightarrow d(v, u_i)$$

For $*, \div$

$$d(v, u_f) \times d(v, u_i) \rightarrow d(v, u_f)$$
$$d(v, u_i) \times d(v, u_f) \rightarrow d(v, u_f)$$
$$d(v, u_i) \times d(v, u_i) \rightarrow d(v, u_i)$$

The general rule is to convert the right hand operand to the left hand operand's units. So, for example, adding a floating point value and a fixed point value will give a result with floating point units. However, for multiplication and division no conversion is performed, as the units are combined via unit algebra and the unit expression becomes floating point if one of the involved units is floating point.

A prototype, RADAR, has been implemented in the Python language[1] to test the semantics and syntax [BM2005]. While it is arguable that interpreted languages are not suited to robotics, Python was chosen because it has a clean syntax and the interpreter is simple to modify. Since Python is an interpreted language, RADAR uses a run time checking approach to the dimensional analysis. The RADAR syntax for creating dimensioned data is shown below.

```
dimensioned ::= (integer | float)~unit(('*' | '/')unit)*
unit ::= [['-'] digit+] letter+ ['^' ['-'] digit+]
```

Note the overloading of the `*` and `/` operators in unit expressions. This could be considered confusing, for example since `s` might be a variable name. However, we expect most robot programmers will be comfortable with the common mathematical symbols for units.

The *operators* available for the primitive dimensioned types are illustrated in Table 2. Table 3 shows the attributes of dimensioned values and some examples.

The sample code in Fig. 2 illustrates the use of units in the sonar avoidance example from the Player/Stage project.

```
dimensioned ::= (integer | float)~unit(('*' | '/')unit)* unit ::=
[['-'] digit+] letter+ ['^' ['-'] digit+]
```

A built in module `dimension` provides the functionality for managing dimensions and units. As an example, consider a programmer who wishes to use a light intensity sensor. A new dimension, "lumens", would be required, as well as a unit of measure in that dimension. If the programmer wishes to ensure the intensity value is within certain limits, then range limits would be set, as shown in Fig 1.

The primitive data type can be used to create higher level data types representing more complex dimensional structures. For example a tuple of dimensional values can represent a robot pose. A list of these can represent a path. Add a time value to each tuple and the path becomes a motion plan

[1] http://www.python.org

Table 2. Examples of operations on dimensioned data.

Operator	Example	Result
Addition	5m + 2cm	5.02m
	5m + 0.02	Error
Subtraction	5m − 2cm	4.98m
	5m − 2rad	Error
Multiplication	5m × 2	10m
	5m × 2m	$10m^2$
Division	5m ÷ 2	2.5m
	5m ÷ 2s	2.5m/s
	5m ÷ 2m	2.5
Comparison	2m < 3cm	False
	2m > 3cm	True

Table 3. Attributes of dimensioned values and some examples.

Attribute	*Examples*		
Value	43.0	5	1.5
Unit	cm	cell	radians
Representation	float	fixed	float
Unit Range Limits	-10 to 120	0 to 10	0 to 2π
Code	43~cm	5~cell	1.5~rad

Listing 1. New dimensions can be created using the `dimension` module.

```
dimension.DefineDimension ('lumens') dimension.DefineUnit
('lumens',
                        'intensity',
                        UnitPrec_Floating,
                        1.0,
                        UnitRange_Include, 0,
                        UnitRange_Exclude, 10 )
```

that can be iterated over and interpolated based on the time value. Similar to the consistency checking of dimensionality in terms of units of measure, higher level structures could provide support for checking consistency of dimensionality in terms of spatial dimensions. For example, ensuring that a programmer does not inadvertently combine the X and Y dimensions of a robot's pose could be done using a vector structure that uses dimensioned data types for its individual components.

The dimensional analysis system makes the management of real world data in robot software easy and clear, improving the programmability and maintainability of robotic systems. This illustrates just one aspect of how programming languages can be improved specifically for human programmers of robots.

```
if motorsOn:
    position.set_cmd_vel(0~m/s, 0~m/s, 0~rad/s, 1)
    position.enable(1)
else:
    position.set_cmd_vel(0~m/s, 0~m/s, 0~rad/s, 0)
    position.enable(0)

newSpeed = 0~m/s newTurnRate = 0~rad/s while 1:
    robot.read()

    if len (laser.Ranges()) == 0:
        print 'No laser data'
        continue
    leftRanges = laser.Ranges()[:len(laser.Ranges())/2]
    rightRanges = laser.Ranges()[len(laser.Ranges())/2:]
    minLeft = min(leftRanges)
    minRight = min(rightRanges)
    l = (minLeft * 10) / 5 - 1~m
    r = (minRight * 10) / 5 - 1~m
    if l > 1~m:
        l = 1~m
    if r > 1~m:
        r = 1~m
    newSpeed = ((r + l) / 1~s) / 10
    newTurnRate = ((r - l) / 1~m) * 1.57~rad/s
    newTurnRate = min (newTurnRate, 40~deg/s)
    newTurnRate = max (newTurnRate, -40~deg/s)

    position.set_cmd_vel(newSpeed, 0~m/s, newTurnRate, 1)
```

Fig. 2. The Player laser obstacle avoidance example.

3 Robot Debugging Tools

This section presents a case study aimed at improving the process of debugging robot systems. It begins with an analysis of developer–robot interaction and then presents an Augmented Reality (AR) system for visualising robot data.

A shared perceptual space which both the human user and robot understand is a requirement for effective interaction [BEFS2001]. Both the human user and the robot have a perceptual space where communication is meaningful, and it is the overlap of these two spaces that is the interaction space of the human–robot coupling (Fig. 3). The interaction space for input may be different from the output space. For example a robot that knows how to represent an emotion using facial expressions, but cannot recognise the facial expressions of others, will have differing input and output spaces. Any mismatch between the perceptual input and output space may put additional cognitive load on the user, who will have to perform additional translations before communicating with the robot.

This case study focuses on providing a shared perceptual space for the robot–developer interaction. By allowing the developer to understand the robots world view we are effectively enhancing the shared space as we are allowing the developer to view the world in the same way as the robot.

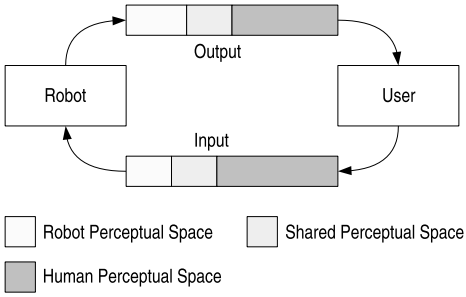


Fig. 3. Shared human–robot perceptual space.

3.1 AR for Developer Interaction

AR provides an ideal presentation of the robot's world view; it displays robot data layered within the real environment. By viewing the data in context with the real world the developer is able to compare the robot's world view against the ground truth of the real world image without needing to know the exact state of the world at all times. This makes clear not only the robot's world view but also the discrepancies between the robot's view and reality. The main challenge in AR is to accurately track the human user so that the overlaid data is accurately positioned when the human viewer changes position and orientation. However this is generally a simpler problem than the alternative, which is to track the entirety of a potentially large and dynamic real world environment and explicitly calculate a comparison between the robot view and the actual world view.

Related Work

Player [PSP2005] has a built in tool, playerv, for visualising its interfaces. However this tool provides the visualisation in isolation. By providing an AR interface for the visualisation we show the data in context with the real objects, making understanding the data more intuitive.

Shikata *et al* [SGNI2003] present an algorithm evaluation system that uses virtual robots in a virtual world to test the performance of avoidance algorithms when human users are involved. The main advantage is that a real human is used rather than a scripted one, thus giving more accurate behaviour while at the same time the virtual robots mean there are no safety issues. In particular the system is used to test avoidance algorithms with no danger to the human user from collisions occurring when the algorithm fails. Milgram *et al* used AR in creating a virtual measuring tape and a virtual tether to assist inserting a peg in a hole [MZDG1993, MRG1995]. Freund *et al* [FSR2001] use AR to more simply display complex 3D state information about autonomous systems. Raghavan *et al* [RMS1999] describe an interactive tool for augmenting the real scene during mechanical assembly. Pettersen *et*

al [PPS2003] present a method to improve the teaching of way points to a painting robot, using AR to show the simulated painting process implied by the taught way points. Brujic-Okretic *et al* [BGHMP2003] describe an AR system that integrates graphical and sensory information for remotely controlling a vehicle.

Daily *et al* [DCMP2003] use AR to present information from a swarm robotics network for search and rescue. Recently KUKA began to investigate the use of AR to visualise data during training [BK2004].

Amstutz and Fagg [AF2002] describe an implementation for representing the combined data of a large number of sensors on multiple mobile platforms, using AR and VR to display the information.

3.2 AR Toolkit Implementation

Given the rapidly changing nature of available AR hardware and software techniques, to remain relevant any system must be flexible and independent of any specific AR implementation.

To be practically effective an AR toolkit must:

- Be modular and flexible in order to make use of core technology advances.
- Be robust, and able to run as a permanent installation in a robotics laboratory.
- Provide sufficient accuracy in the underlying AR system for developer interaction.

All are implemented in our toolkit.

The software architecture has been designed in a highly modular fashion allowing for individual components to be replaced. While the implemented toolkit is Player based, the modular nature of the AR toolkit allows for the easy addition of support for any other device interface system.

The rendering process for the toolkit is broken into four basic stages: capture, preprocessing, rendering and postprocessing. The rendering stage is further broken down into three rendering layers and a ray trace step is optionally performed to enhance stereo representations. Each step is described below.

1. Capture: the background frame, orientation and position of the camera are captured.
2. Preprocessing: such as blob tracking for robot registration.
3. Render - Transformation: the position of the render object is extracted from its associated list of position objects, and appropriate view transforms are applied.
4. Render - Base: invisible models of any known 3D objects are rendered into the depth buffer. This allows for tracked objects such as the robot to obstruct the view of the virtual data behind them. The colour buffers are not touched as the visual representation of the objects was captured by the camera.

5. Render - Solid: the solid virtual elements are drawn.
6. Render - Transparent: transparent render objects are now drawn while writing to the depth buffer is disabled.
7. Ray Trace: to aid in stereo convergence calculation, the distance to the virtual element in the centre of the view is estimated using ray tracing. This is of particular relevance to stereo AR systems with control of convergence, and optical see through stereo systems.
8. Postprocessing: once the frame is rendered any post processing or secondary output modules are called. This allows the completed frame to be read out of the frame buffer and, for example, encoded to a movie stream.

The toolkit architecture is centred around an output device, each of which contains a capture device for grabbing the real world frame, and a camera device for returning the camera parameters, including pose. The capture device could be a null object for optical see through AR, or for a purely virtual environment. Also, the output device maintains three component lists: secondary outputs, which monitor the interface and provide movie capture; preprocessing objects used for image based position tracking; and finally a list of render item pairs. Each *Render Pair* consists of a render object that performs the actual rendering of a virtual element and a chain of position objects that define where it should be rendered.

Fig. 4 shows the software structure and each of the components is summarised in Table 4. The first four items (Capture, Camera, Secondary Output and Preprocessing) are all unique to the output object. The render objects and position objects can be used in multiple combinations, potentially with different output objects. For example a Stereo head mounted display (HMD) needs the same laser data (render object) on both displays (output objects), while for a single output the same origin (position object) could be used to render both laser and sonar data.

To create a permanent laboratory setup it was important to detect the presence of robots. This presence is in two forms, first the system must cope with operational presence, if a robot is switched on or off the system must detect this and if required start (or stop) displaying appropriate data. Secondly, if a robot is physically removed from the system, i.e. is outside the active field of view, or is unable to be tracked then the data for that robot must not be displayed, this is particularly relevant for optically tracked robots.

The toolkit has been tested with two core AR configurations. A magnetically tracked video see–through HMD (Trivisio ARVision3D) provides an immersive, stereo environment for a single user. A fixed overhead camera and an AR display on a large wall mounted screen near the robots provides a communal environment where humans and robots may interact together.

The toolkit has been tested with three different Player compatible robots, the B21r robot from iRobot, the Pioneer 3DX from ActivMedia and our own in house Shuriken robot, representing a rich set of capabilities and a range of scales. Sample output of these systems is shown in Fig. 5. The visualisation

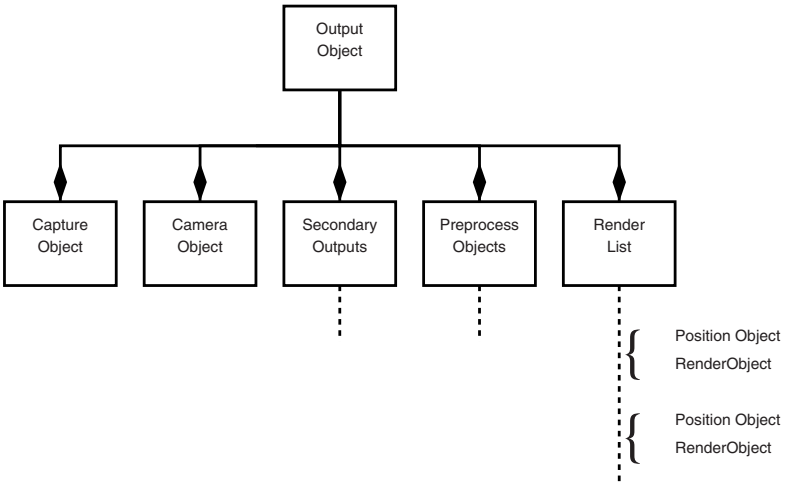


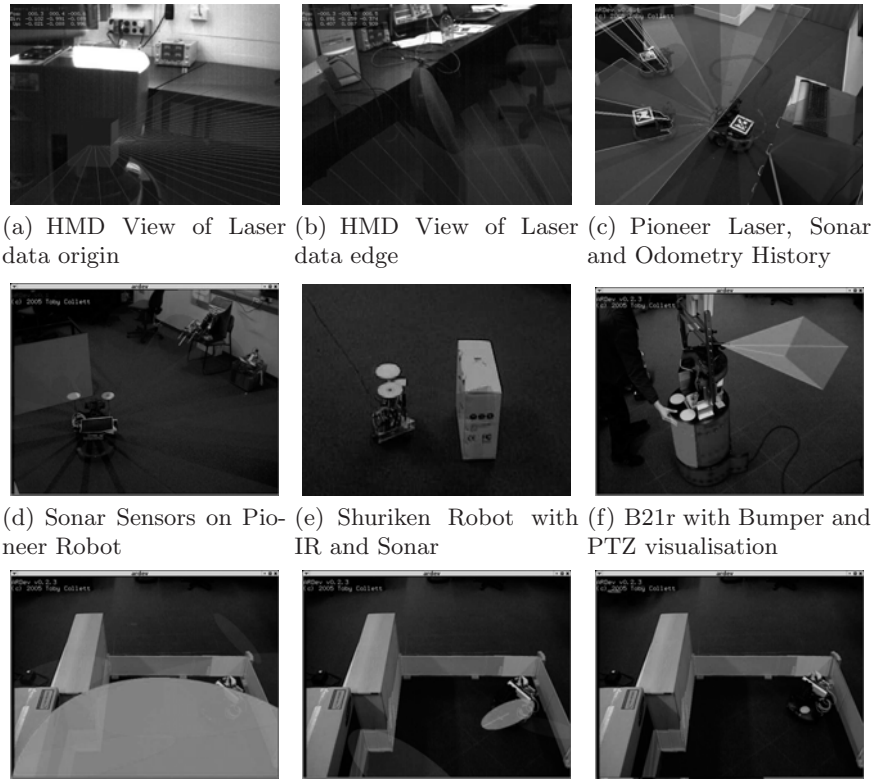
Fig. 4. The ARDev software architecture.

Table 4. Core objects.

ObjectName	Description
OutputObject	Handles the actual rendering of the information and its display
CaptureObject	Supplies the real world input frames, for example from a Video for Linux device, or a blank frame for optical see through AR
CameraObject	Provides the geometry and optical properties of the camera
SecondaryOutputObject	Allows a secondary output stream to exist, such as output to video, or capturing stills
PreProcessObject	Handles processing of a frame before rendering (allows for tracking of markers using image processing)
PositionObject	Returns the position of a tracked object, relative to an arbitrary base
RenderObject	Handles the rendering of a single augmented entity

system has run continuously in the lab for 7 days, at which point it was stopped for an upgrade.

The AR toolkit provides intuitive representations of geometric data from Player interfaces for the robot developer. The use of AR for the visualisations gives the developer an immediate understanding of the robot data in context with the real world baseline, allowing any limitations in the robots world view to be understood, and leading to much faster solutions to related software



(a) HMD View of Laser data origin (b) HMD View of Laser data edge (c) Pioneer Laser, Sonar and Odometry History

(d) Sonar Sensors on Pioneer Robot (e) Shuriken Robot with IR and Sonar (f) B21r with Bumper and PTZ visualisation

(g) AMCL Initial Distribution, High Covariance (h) AMCL Improved Estimate (i) AMCL Converged Result

Fig. 5. AR system output.

issues. Our initial implementation of these concepts has shown useful and promising results for robot software development.

4 Conclusion

The three case studies promote an important goal that designers of robotic systems include the human developer as a significant role. Robot development tools should provide expressive programming languages and frameworks that enhance the opportunity for human developers to describe robot behaviour. Debugging tools should be human–oriented and improve the immediate visualisation of robot data for the human developers.

Acknowledgements

Toby Collett is funded by a top achiever doctoral scholarship from the New Zealand Tertiary Education Commission.

References

[CMG2005] Toby Collett, Bruce MacDonald, and Brian Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proc. Australasian Conference on Robotics and Automation*, Sydney, Australia, December 5–7 2005.

[MRT2003] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: The Carnegie Mellon Navigation (CARMEN) toolkit. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, volume 3, pages 2436–2441, Las Vegas, NV, October 2003.

[CLMV2004] C. Côté, D. Létourneau, F. Michaud, J.-M. Valin, Y. Brosseau, C. Raïevsky, M. Lemay, and V. Tran. Code reusability tools for programming mobile robots. In *Proc IEEE Intl. Conf. on Intelligent Robots and Systems*, volume 2, pages 1820–5, Sendai, Japan, Oct 2004.

[BMK2003] D. Blank, L. Meeden, and D. Kumar. Python robotics: An environment for exploring robotics beyond legos. In *Proceedings of the Thirty-Fourth SIGCSE Technical Symposium on Computer Science Education*, Reno, Nevada, February 2003. ACM Press.

[KU2005] KUKA Automatisering + Robots N.V. http://www.kuka.be/, June 2005.

[GP1993] Dean W. Gonzalez and Tim Peart. Applying dimensional analysis. *Ada Lett.*, XIII(4):77–86, 1993.

[ABB2005] The ABB group. http://www.abb.com/, June 2005.

[GE1985] N. H. Gehani. Ada's derived types and units of measure. *Softw. Pract. Exper.*, 15(6):555–569, 1985.

[GE1977] Narain Gehani. Units of measure as a data attribute. *Computer Languages*, 2(3):93–111, 1977.

[HI1988] Paul N. Hilfinger. An Ada package for dimensional analysis. *ACM Trans. Program. Lang. Syst.*, 10(2):189–203, 1988.

[PPS2003] T. Pettersen, J. Pretlove, C. Skourup, T. Engedal, and T. Lokstad. Augmented reality for programming industrial robots. In *Proceedings of the Second IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 319–20, 7–10 Oct 2003.

[PSP2005] Player/Stage. The player/stage project. http://playerstage.sourceforge.net/, January 2005.

[RMS1999] V. Raghavan, J. Molineros, and R. Sharma. Interactive evaluation of assembly sequences using augmented reality. *Robotics and Automation, IEEE Transactions on*, 15(3):435–449, 1999.

[RO1988] P. Rogers. Dimensional analysis in Ada. *Ada Lett.*, VIII(5):92–100, 1988.

[SGNI2003] R. Shikata, T. Goto, H. Noborio, and H. Ishiguro. Wearable-based evaluation of human-robot interactions in robot path-planning. In *Proc. IEEE International Conference on Robotics and Automation (ICRA 03)*, volume 2, pages 1946–1953, 2003.

[STE1999] A. G. Stephenson and group. Mars Climate Orbiter Mishap Investigation Board Phase I Report. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf, November 10 1999.

[TUO2005] The units of measure library. http://tuoml.sourceforge.net/, 2005.

[UMR1994] Zerksis D. Umrigar. Fully static dimensional analysis with C++. *SIGPLAN Not.*, 29(9):135–139, 1994.

[VGH2003] R.T. Vaughan, B.P. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems (IROS03)*, volume 3, pages 2421–2427, Las Vegas, Nevada, October 2003.

[BHGC06] Antonio C. Domínguez-Brito, Daniel Hernández-Sosa, Josep Isern-González and Jorge Cabrera-Gámez , *CoolBOT: a Component Model and Software Infrastructure for Robotics*, In Brugali D. (Ed.) Software Engineering for Experimental Robotics, Springer STAR series, 2006.

[BKMWO06] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams and Anders Orebäck, *Orca: a Component Model and Repository*, In Brugali D. (Ed.) Software Engineering for Experimental Robotics, Springer STAR series, 2006.

[AG1984] Mukul Babu Agrawal and Vijay Kumar Garg. Dimensional analysis in Pascal. *SIGPLAN Not.*, 19(3):7–11, 1984.

[DMM1986] A Dreiheller, B Mohr, and M Moerschbacher. Programming Pascal with physical units. *SIGPLAN Not.*, 21(12):114–123, 1986.

[BA1987] Geoff Baldwin. Implementation of physical units. *SIGPLAN Not.*, 22(8):45–50, 1987.

[CG1988] R.F. Cmelik and N.H. Gehani. Dimensional analysis with C++. *Software, IEEE*, 5(3):21–27, 1988.

[BR2001] Walter E. Brown. Applied template metaprogramming in siunits: the library of unit-based computation. Technical report, Computational Physics Department, Computing Division, Fermi National Accelerator Laboratory, August 2001.

[ACLM2004] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele, Jr. Object-oriented units of measurement. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 384–403. ACM Press, 2004.

[BM2005] Geoffrey Biggs and Bruce MacDonald. A design for dimensional analysis in robotics. In *Third International Conference on Computational Intelligence, Robotics and Autonomous Systems*, Singapore, December 2005.

[BEFS2001] C. Breazeal, A. Edsinger, P. Fitzpatrick, and B. Scassellati. Active vision for sociable robots. *IEEE Trans. Syst., Man, Cybern. A*, 31(5):443–453, 2001.

[BK2004] Rainer Bischoff and Arif Kazi. Perspectives on augmented reality based human-robot interaction with industrial robots. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 04)*, pages 3226–3231, 2004.

[DCMP2003] M. Daily, Youngkwan Cho, K. Martin, and D. Payton. World embedded interfaces for human-robot interaction. In *Proc. 36th Annual Hawaii International Conference on System Sciences*, pages 125–130, 2003.

[BGHMP2003] V. Brujic-Okretic, J.-Y. Guillemaut, L.J. Hitchin, M. Michielen, and G.A. Parker. Remote vehicle manoeuvring using augmented reality. In *International Conference on Visual Information Engineering. VIE 2003.*, pages 186–9, 7–9 July 2003.

[FSR2001] E. Freund, M. Schluse, and J. Rossmann. State oriented modeling as enabling technology for projective virtual reality. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and System (IROS 01)*, volume 4, pages 1842–1847, 2001.

[MRG1995] P. Milgram, A. Rastogi, and J.J. Grodski. Telerobotic control using augmented reality. In *Proceedings., 4th IEEE International Workshop on Robot and Human Communication. RO-MAN'95*, pages 21–9, Tokyo, 5–7 July 1995.

[MZDG1993] P. Milgram, S. Zhai, D. Drascic, and J. J. Grodski. Applications of augmented reality for human-robot communication. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and System (IROS 93)*, volume 3, pages 1467–1472, 1993.

[AF2002] P. Amstutz and A.H. Fagg. Real time visualization of robot state with mobile virtual reality. In *Proc. IEEE International Conference on Robotics and Automation (ICRA 02)*, volume 1, pages 241–247, 2002.

[Ken96] Andrew John Kennedy, *Programming Languages and Dimensions*, Technical Report St. Catherine's College, March 1996.