The Active Vision Shell

Jeffrey A. Fayman 1
Ehud Rivlin 1
Henrik I. Christensen 2

1Department of Computer Science
Israel Institute of Technology – Technion
Haifa, Israel
2Laboratory of Image Analysis, Aalborg University
Aalborg East, Denmark

Abstract

In this paper, we present an agent architecture/active vision research tool called the Active Vision Shell (AV-shell). The AV-shell can be viewed as a programming framework for expressing perception and action routines in the context of situated robotics. The AV-shell is a powerful interactive C-shell style interface providing many capabilities important in an agent architecture such as the ability to combine perceptive capabilities of active vision with active capabilities provided by other robotic devices, the ability to interact with a wide variety of active vision devices and a set of active image routines. We present the user interface to these capabilities as well as an appropriate architecture for handling the complexities of an agent architecture.

1 Introduction

In recent years, active vision has become an intensive area of research. Active vision, which was first introduced in [4] and later explored in [2], is defined as the explicit control of the sensory system to improve robustness and eliminate ill-posed conditions. It has been shown that by actively controlling the vision system, many classic computer vision problems become easier to solve [2]. To this end, a wide variety of mechanisms such as robotic heads, single or multiple vision sensors mounted on robotic arms etc have been developed as active vision research tools [25, 15, 21, 8, 27, 22, 16].

In [28], several important open areas in active vision research are pointed out. Among these areas, the importance of agent architectures in which active vision is integrated with robotic architectures is emphasized: “one view of a robot architecture is a programming framework for expressing perception and action routines, and the algorithms, data structures and techniques/methodology for combining them in a robot agent to achieve a task”.

As an aid to research, the agent architecture should be general enough to interact with existing active vision and robotic devices. Recent attempts at providing this generality
focus on robotic heads alone [12]. Clearly, more general mechanisms are needed. These mechanisms should be applicable not only to various robotic head designs, but to other active vision paradigms such as “Eye-in-Hand” and should be flexible enough to adapt to new paradigms. An effective agent architecture must also provide a vocabulary of visual routines to be used as the building blocks for vision-based tasks in situated robotics. Additionally, appropriate tools for combining these routines to realize the perceptive capabilities used in a perception-action cycle are required.

In this paper, we present an agent architecture/active vision research tool called the Active Vision Shell (AV-shell) which focuses on these areas of active vision research. In particular, the AV-shell can be viewed as a programming framework for expressing perception and action routines in the context of situated robotics. The AV-shell is a powerful interactive C-shell style interface which provides many capabilities important in an agent architecture such as the ability to combine perceptive capabilities of active vision with active capabilities provided by other robotic devices, the ability to interact with a wide variety of active vision devices and a vocabulary of active vision routines.

The architecture upon which the AV-shell is built is also presented. This architecture provides features for handling the reflexive nature of basic active vision processes as well as integrating active vision devices into more complex systems. At the end of the paper, we show how the AV-shell can be used to easily implement complex interactive behaviors involving active vision devices.

The remainder of this paper is organized as follows: In section 2, we review related work. Next, in section 3, we present the design goals and methods used in the development of the AV-shell. In section 4 we present the AV-shell and its user interface. In section 5 we discuss the architecture upon which the AV-shell is based, and in section 6 we discuss AV-shell implementation issues. In section 7 we give an example of how the AV-shell can be used to implement complex perception-action tasks and we conclude with section 8.

2 Related Work

The work presented in this paper is concerned with providing an agent architecture/active vision research tool. Our system provides the ability to integrate active vision and robotics to achieve various tasks, to interact with a wide variety of active vision devices and a vocabulary of active image routines.

Many researchers have looked into problems of combining perception and action at the micro level of the active vision device [24, 18, 32, 6, 11, 3, 23]. However, relatively little work has been done in integrating active vision into more complex systems. In [13], Crowley and Christensen discuss a sophisticated architecture and system called “VAP/SAVA” for the integration and control of a real-time active vision system. Their system provides a comprehensive agent architecture which was designed as a continuously operating process. An application of the system in the context of visual navigation is presented. While VAP/SAVA has many features present in our system, we are also interested in providing an active vision research tool. Therefore, our system provides a full programming language with robotics related operators and the ability to perform simulations. Additionally we provide mechanisms for experimenting with many active vision device configurations. Our image processing rou-
tines differ as well in that we are trying to provide a set of general active image routines whereas routines in VAP/SAV/, are made up of three specific classes of procedures: (1) edge segment extraction; (2) edge chain extraction and; (3) procedures for providing measures for ocular reflexes. Several other active vision environments are described in [10].

Crowley et-al. [12] discussed layered control of a binocular camera head in which a fixation point may be commanded by an external process or driven by simple measurements from the scene. However, the relationship between the head and other devices was not addressed. This can be seen as a first step in creating an AV-shell like interface.

Research in active vision has lead to the design of a variety of paradigms for controlling the sensory system such as “eye-in-hand” and “robot heads”. In most of this work, the control and interaction mechanisms described were specific to the particular hardware being discussed. Crowley et-al. [12] discussed a head control mechanism called the “virtual head” which provides a protocol general enough to map onto the kinematic structure of many physical heads. In their controller, all device specific parameters are encoded in a low-level module called the translator. This work is similar in nature to ours in that both are trying to provide a general mechanism for controlling a variety of devices, however, Crowley et-al. are concerned only with stereo robot heads whereas we are interested in more general methods applicable to a wide variety of active vision devices.

The idea of a vocabulary of primitive processes is an important one. In [29], Ullman discusses a set of five operations that he feels are important for extracting shape properties and spatial relations among objects and object parts. In that paper, he discusses the importance of a fixed set of powerful basic operations together with the means for combining them into different routines. Similarly, we define a set of routines, however we approach the definition of these routines from an active-vision/purposive point of view. We also provide the tools necessary for combining these routines into higher level processes and to achieve specific goals.

3 AV-Shell Goals and Methods

We believe that the full realization of the benefits of active vision will not occur until active vision is studied in broader contexts. In biological systems the visual system and the apparatuses used for its movement perform in a way that can be described as reflexive, providing the perceptive capabilities used in a perception-action cycle. The actions in this cycle are in turn carried out by other parts of the system. This implies that perception and action must be combined to provide capabilities similar to those found in complex systems. It also necessitates powerful and flexible architectures. In this section, we discuss the design goals behind the agent architecture/active vision research tool AV-Shell which was designed to provide these capabilities.

The AV-shell can be viewed as playing dual roles: (1) as suggesting an agent architecture framework; and (2) as an active vision research tool. As an agent architecture framework, the AV-shell provides tools necessary for efficient integration of active vision into complex systems. As an active vision research tool, the AV-shell provides an intuitive interface to an extensive set of active vision related commands, the ability to perform simulations and a full programming language with robotics related operators.
Several goals guided the design of the AV-shell. Firstly, it should be applicable to a wide variety of active vision devices. This helps to make it portable and gives us the ability to experiment with various active vision configurations using the same software. Secondly, it should provide a modular vocabulary of visual routines including high-level activity routines, medium-level basic process routines and low-level primitive routines. It should also provide the architectural flexibility to experiment with various combinations of these visual modules. Thirdly, the AV-shell should provide a convenient platform for performing experiments in these areas. Lastly, it should be based on an architecture which both enables the various components of active vision to be integrated, and allows active vision to be incorporated into more complex systems.

In the following subsections, we discuss the way in which the first two goals have been realized in the AV-shell. In particular, how the AV-shell enables control of various active vision devices and the visual routines provided. In the next two sections, the experimental platform, user interface and architecture are discussed.

3.1 Active-vision devices

We explained earlier that the primary function of an active vision device is to allow for active movement of the visual sensors thereby making previously ill-posed problems well defined. This implies that the device is simply a manipulator used for moving the visual sensors. There are various paradigms that have been used to do this. A partial list of these paradigms is given below:

Movable-Eye
Several new device designs for camera movement have appeared recently in the literature. These devices are small parallel linkage manipulators used for moving single cameras. Examples of moveable eyes can be found in [17, 5].

Eye-in-Hand
In this configuration, a single camera is mounted on a robotic arm which provides movement capabilities. Examples of this configuration can be found in [1, 26].

Robotic Head
Robotic heads are special purpose manipulators used to provide visual system capabilities similar to those found in biological systems. Most heads are binocular or trinocular. Examples of robotic heads can be found in [15, 25, 21, 8].

Head-in-Hand
In this configuration, a robotic head is mounted on a robotic arm. The movement of the position and orientation of the head frame is provided by the arm and the head provides foveation capabilities. An example of a head-in-hand can be found in [27].

Camera or Head Mounted on a Mobile Platform
In this configuration, a fixed camera or robotic head is mounted on a mobile platform. Examples of this configuration can be found in [9, 19, 13].
It is desirable to provide an abstraction for a general interface to this wide range of devices. At the same time, we would like encapsulation to enable transparent use of existing control software. This naturally leads to an object oriented approach and because we conceptually have difficulties supporting both perception and action in one device, we believe that it is necessary to view the vision sensors and the mechanisms used to move them separately. We define an “Active-vision device” (AVD) to include two parts: the vision sensor, which we refer to as “camera” and the device used for moving the camera, which we refer to as “head”. By doing this, several benefits are realized. Firstly, the mechanism will be applicable to stationary vision systems as well; these are simply cameras without the ability to move. Secondly, by decoupling cameras and heads, the “vision” problems and the “active” problems of active vision can be studied individually or together.

Another benefit we gain by this view of AVD’s is that we can define their operational space in terms of the cameras alone without concern for the underlying movement mechanism. This leads to a set of medium-level commands that are compatible with many active vision devices (these commands will be discussed in section 4).

Operational spaces are used as a method to reduce the complexity of control problems of robotic devices and as an aid to human robot interaction. Examples of operational spaces are Cartesian space for manipulators and shape space [14] for dextrous robotic hands. In most previous work the operational space of robotic heads is defined as the 3-D position of a fixated object relative to some fixed system such as the cyclopean system. The user commands the head to fixate on a point with a command such as look-at <3-D Point> [12].

This definition is useful in cases of binocular or trinocular heads where optical axes intersect (i.e. 0 < θ < π/2 and π/2 < θ, < π). However, the potential of these heads is not fully realized as it does not take advantage of their ability to look in more than one direction at a time and it is not applicable to other AVD’s such as eye-in-hand where there is only one camera.

We propose a more general definition of operational space for AVD’s called “Visual Operation Space” (VOS). VOS is a mapping from Ω to O where Ω is a set of vision sensors and O a set of particular operational space mappings. VOS distinguishes between cameras whose optical axes intersect and those that do not. In particular if Ω is some set of vision sensors, then in theory, the optical axes of some of the sensors in Ω will intersect. We denote these as Λ. For some subset of Λ, there may be defined gaze space transformations g(θ) and f(p) (i.e. forward and inverse kinematics of a head) of which we are interested in using. We will denote these sensors as Υ. The remaining cameras form another set called Ψ (i.e. Ψ = {Ω – {ςₙₙₙₙ}_ς∈Υ}). Finally, the defined operational space mappings form a set O. VOS is then the mapping Ω → O such that

\[
VOS(x) = \begin{cases} 
  g(x), f(x) & \text{if } x \in Υ \\
  d(x) & \text{if } x \in Ψ
\end{cases}
\]

The functions g and f are the defined gaze space transformations for the particular cameras and the function d is the transformation required to bring the optical axis of a camera to the appropriate direction.
3.2 AVD class hierarchy

VOS gives us an operational space which is not restricted to a particular active vision device. For example, we can just as easily specify a viewpoint for a camera mounted on a head as a camera mounted on a manipulator. The physical mechanism for providing this generality is given by a set of data structures. We took an object oriented approach in the definition of these data structures. In this approach, the particular features possessed by an AVD define its class. Examples of features are independent pan, independent tilt and cyclotorsion. Data specific to a given class define its type, and an instance contains data specific to an instance of a device type. Using this methodology, any number of instances of various types can be dynamically created and controlled as each level in the hierarchy inherits information from its ancestors. The same class:type:instance hierarchy applies to manipulators, cameras and robotic heads so that various AVD’s can be dynamically created and manipulated. An example of typical information contained in these data structures is given in figure 1. In the example, a subset of the head class, type and instance structures are presented for the Aalborg head class. A complete listing of data structures for cameras and heads is given in appendix A. All values specified in these data structures are converted to device coordinates at the low-level.

3.3 Active-vision control

Now that we have a definition of AVD’s, it is instructive to understand what the purpose of these devices are. This will lead to the definition of a set of active vision routines.

On the image level, we view the purpose of active vision devices as performing three primary activities:

- **Fixation**
  
  Bringing the primary attention of the active vision device to a new point of interest. Fixation includes a saccadic motion and continuous vergence control which is driven by disparity and accommodation cues.

- **Pursuit**
  
  Pursuit involves the tracking of a moving object and is driven by target velocity estimates and dynamic accommodation.

- **Stabilization**
  
  Stabilization is the compensation of perceived target motion due to egomotion.

It is clear that these primary activities are built up of a set of basic processes including accommodation, disparity detection, motion detection and saccadic motion. Looking at the primary activities from the point of view of these basic processes, we see that they can be defined as active compositions of the basic processes in continuously operating perception action loops. We will discuss this composition in more detail later.
/* Head Class Table */
/* In this example of the head class table we are looking at head class 1 which is an AALBORG */
/* class head. This means that class 1 represents those heads designed with coupled pan, */
/* coupled tilt and two vergence motors for left and right cameras. */

typedef struct {
  char NAME[16]; /* Head class name : AALBORG */
  int NCAMERAS; /* Number of Cameras for the class : 2 */
  int ATTR[NATTRIBUTES] /* Various head attributes that define a head : PAN, TILT, V1, V2 */
};

/* Head type table */
/* In this example of the head type table, we are looking at a head type called TRH (Technion */
/* Robot Head) which is a class 1 head. We can see that the TRH type head has minimal joint */
/* values of -90 (pan), -45 (tilt) and -90 (each vergence motor). */

typedef struct {
  char NAME[16]; /* Head type name : TRH */
  int USE; /* USE >= 1 indicates number of head ... : 1 */
  int CLASS; /* head class. One of 1..9 : 1 */
  int NJ; /* number of head joints : 4 */
  int NCAMERAS; /* number of cameras supported by head : 2 */
  double a[JPH_MAX+1] /* Denevit-Hartenberg joint distances */
  double d[JPH_MAX+1] /* Denevit-Hartenberg link offsets */
  double QMIN[JPH_MAX+1] /* Minimal values of joint angles : -90, -45, -90, -90 */
};

/* Head table (contains head instances) */
/* In this example of the head table, we are looking at an instance called inst1 of type TRH. */
/* This instance is not open for attachments (cameras 1 and 4 are already attached) and it has */
/* been located. The data structures tell us that the head is attached to a PUMA manipulator */
/* and gives us the current joint values for the pan and tilt motors. */

typedef struct {
  char NAME[16]; /* Head name : inst1 */
  int OPEN; /* OPEN>=1 indicates open head (for attachment!) : 0 */
  int L0; /* L0=1 head located : 1 */
  int PORT; /* port that head is connected to, 0 if not : 6 */
  int MMP; /* Pointer to manipulator that supports the head : 1 (PUMA) */
  int TYPE; /* Head type (index of head type table) : 3 */
  int HDDCAM[CPH_MAX]; /* Pointer to camera instance(s) mounted on head : 1, 4 */
  double PAN; /* pan angle : 30.0 */
  double TILT; /* tilt angle : -8.0 */
};

Figure 1: Data Structure Example
3.4 Active vision routines

Much of the past work in active vision has tried to simulate on a mechanical device the capabilities of the human visual system. The main objective of this system is to insure that the object of interest (or changing objects of interest) is continuously projected onto the area of the eye with the highest resolution (fovea centralis), and that the projected images are of high quality.

In [29], Ullman discusses the idea that different image processes should share elemental operations implying that new combinations of basic operations can be assembled to meet new computational goals. Swain et-al. [28] advocate this idea when they say that an important area of active vision research is in developing a vocabulary of visual routines to be used as the building blocks for vision-based tasks in situated robotics. These views lead to a layered, hierarchical structure for active vision image routines where higher level routines are made up of lower level routines.

The active vision routines provided by the AV-shell are made up of three “levels”; (1) high-level activity routines supporting activities such as fixation and pursuit; (2) medium-level basic process routines such as accommodation and motion detection; and (3) low-level primitive routines such as convolution and correlation. In this configuration, processes at higher levels are made up of processes at lower levels. Below, we will discuss in more detail these levels of active image routines. The syntax of these routines will be given in section 4 and architectural issues of active image processing will be discussed in section 5.

At the highest level of the active vision routine hierarchy we have the activity routines. The AV-shell provides routines supporting the following activities:

- **Fixation**
  The process by which the primary attention of the active vision device is brought to a new point of interest in the world. Fixation includes the saccadic motion used to initially align the optical axis with the fixation point, followed by continuous vergence control which is driven by disparity and accommodation cues.

- **Smooth pursuit**
  Track a moving object with smooth camera motions. Smooth pursuit is the process by which we track a moving object. The goal is to keep the retinal position of the moving object at (0,0). Smooth pursuit is driven by velocity mismatch and/or position error.

- **Stabilization**
  Stabilization is similar to pursuit in that both are attempting to keep a moving target fixated. However, in stabilization, the perceived target motion is caused by egomotion and the stabilization mechanism is used to compensate the egomotion.

At the medium-level, the AV-shell includes a set of basic process routines which are used to implement high-level activities.

- **Dynamic accommodation**
  Provides sharp high quality images by focus adjustments and is a depth cue. Unfocused images contain less information than focused images. In an active system where
control is closed by vision, it is extremely important to consistently provide sharp high quality images. The better the image quality, the more accurate will be the results. Accommodation is typically controlled using maximization of image gradients.

- **Aperture control**
  Aperture control provides us with two important abilities. Firstly, by controlling the aperture, we are able to adjust for changes in lighting. This will help to give high contrast images over a range of lighting conditions. Secondly, the aperture setting is related to the depth of field of view. By adjusting the aperture, we are able to adjust the depth at which objects are in focus. The size of the depth of field is given by:

\[
\Delta Z = \frac{2 \cdot Z \cdot F \cdot \lambda \cdot s_c \cdot (Z - \lambda)}{F^2 \cdot \lambda^2 - s_c^2 \cdot (Z - \lambda)^2}
\]  

(1)

A point not in focus is projected onto the image plane as a circle called the “circle of confusion” which is denoted \( s_c \) in equation 1. Aperture, \( F \) refers to the diameter of the lens opening and can also be expressed as relative aperture \( \frac{F}{r} \). Aperture can be controlled using simple statistical descriptors derived from the image histogram.

- **Saccade**
  Rapid eye movement of both eyes to a new gaze point. A saccade is a rapid movement of both eyes used to change the gaze point as quickly as possible. A saccade is usually invoked when a new area of interest in the scene is identified. The eyes are rapidly moved to the new area for further exploration. Saccades are typically open-loop controlled in the sense that no visual processing is carried out during the saccade.

- **Vergence/disparity estimation**
  Keep the fixation distance consistent with the target of fixation. In binocular heads, the optical axes of the two cameras sharing a common tilt plane will intersect at some point in space (gaze point) provided \( 0 < \theta_l < \frac{\pi}{2} \) and \( \frac{\pi}{2} < \theta_r < \pi \). By adjusting \( \theta_l \) and \( \theta_r \), we can control the distance from the cameras to the gaze point. The main goal of the vergence process is to keep the fixation distance consistent with the target of fixation. Research has shown that vergence is driven both by disparity and accommodation cues [25, 7].

- **Peripheral motion detection**
  Detect motion in some unknown area of the scene. One of the ways that active vision systems simplify vision problems is by centering an object of interest on the image plane (foveation). An important question is “what is this object of interest and how do we find it”. One answer to this question is that potential objects of interest are moving objects. Of course we do not know where this motion is a-priori so a mechanism for detecting motion in some unknown part of the image is needed. This type of motion detection is called peripheral motion detection.

- **Foveal motion detection**
  Detect motion in a small region centered on the image plane. In foveal motion detection, we are interested in detecting motion in a small area centered on the image plane.
Algorithms performing foveal motion detection can be implemented in real time as the size of the processing window is relatively small. Foveal motion detection forms the basis of smooth pursuit.

- **Cepstral filtering**

  Measure disparity between images in the left and right cameras. In order to perform the vergence function, we must be able to measure the disparity between images in the left and right cameras. Cepstral filtering provides us an efficient means of doing this. The cepstral filter is computed as follows:

  \[
  C(x', y') = F \left( \log \left( |F(u,v)| \right) \right)
  \]  

  Equation 2 results in peaks at positions corresponding to dominating disparity.

  At the low-level, the AV-shell provides a set of functions which are typically used in image processing. These low-level routines will be discussed in more detail in section 4.

### 3.5 Integration of active vision routines

While the routines presented previously constitute a rich set of active vision routines, the task of composing them into continuously running perception action processes has not been specified. In this work, we adopt a model proposed in [20] called the Robot Schemas (RS) model. Table 1 summarizes the RS composition operators. In the RS model, communication channels between concurrent processes are called “ports”. Messages are written to, and read from ports. A port to port connection relation can be specified as an optional third parameter in concurrent composition. This connection relation specifies a set of couples \(op \mapsto ip\) indicating that port \(ip\) and \(op\) are connected.

Using this notation, we can readily see how the high-level activities of fixation and, pursuit can be specified (stabilization is similar to pursuit): Fixation is initialized with a saccade to the fixation point followed by continuous vergence control driven by disparity and accommodation cues.

\[
\text{fixation} = \text{saccade}; ((\text{disparity} \mid \text{accomodation}) :: \text{(vergence control)})
\]

In pursuit, vergence, foveal motion detection and dynamic accommodation are used to continuously drive motion of the vision sensor.

\[
\text{pursuit} = (\text{vergence} \# \text{foveal motion estimation} \# \text{dynamic accommodation}) :: \text{move}.
\]

This notation allows us to specify concurrency but not the interplay between processes that we would like to support as in the biologically motivated vergence architecture shown in figure 2.

---

1 Notation in table 1 is consistent with those of [20], however, in our implementation, the operators were changed to avoid conflicts with existing operators. These changes are given in section 4.
1. **Sequential Composition**: $T = P; Q$. The process $T$ behaves like the process $P$ until that terminates, and then behaves like the process $Q$ (regardless of $P$’s termination status).

2. **Concurrent Composition**: $> T = (P|Q)$. The process $T$ behaves like $P$ and $Q$ running in parallel and with the input ports of one connected to the output ports of the other as indicated by the port-to-port connection map $c$. This can also be written as $t = (\{i \in I : P_i\})$ for a set of processes indexed by $I$.

3. **Conditional Composition**: $T = P \langle v \rangle : Q_v$. The process $T$ behaves like the process $P$ until that terminates. If $P$ aborts, then $T$ aborts. If $P$ terminates normally, then the value $v$ calculated by $P$ is used to initialize the process $Q$, and $T$ then behaves like $Q_v$.

4. **Disabling Composition**: $> T = P \# Q$. The process $T$ behaves like the concurrent composition of $P$ and $Q$ until either terminates, then the other is aborted and $T$ terminates. At most one process can stop; the remainder are aborted.

5. **Synchronous Recurrent Composition**: $T = P \langle v \rangle :: Q_v$. This is recursively defined as $P :: Q = P : (Q ; P :: Q)$.

6. **Asynchronous Recurrent Composition**: $T = P \langle v \rangle :: Q_v$. This is recursively defined as $P :: Q = P : (Q | (P :: Q))$.

**Table 1**: Summary of RS Composition Operators
4 Implementation of the AV Shell

In this section we present the AV-shell. The AV-shell is an agent architecture and tool for conducting research in active vision. The active vision components of the system provide the ability to control a wide variety of AVD’s while integrating active vision into complex systems is accomplished by combining the AV-shell with another system called the Robot Shell (R-shell) [31, 30, 14]. The R-shell provides the tools necessary for controlling and coordinating various robotic devices such as manipulators and dextrous robotic hands. However, it does not possess active vision capabilities. The AV-shell was designed to fill this gap. Together, the AV-shell and the R-shell provide a powerful active vision research tool. We begin with a brief overview of the R-shell, then discuss the AV-shell.

4.1 Robot Shell

The R-shell is an interactive program written in C under the UNIX operating system. It interprets and executes robotics related commands called “R-shell commands” in much the same way that a shell such as c-shell interprets and executes system related commands. Robotic data in the R-shell environment are entered, manipulated and displayed through the use of these commands. Included in the set of R-shell commands are an extensive set of algebraic operations such as matrix and vector addition, multiplication, matrix inversion and transposition, computation of determinants, traces and norms as well as various arithmetic operations and standard mathematic functions for scalars all of which are used extensively in robotics work.

R-shell commands can be entered interactively or placed in R-shell scripts and executed by simply typing the script file name followed by its actual parameters. Thus scripts provide procedural abstraction in R-shell. Scripts can call other scripts and can call themselves providing nesting and recursion. The R-shell script language provides a full set of flow control statements such as if-then-else, while, loop and case. Additionally, R-shell supports both call-by-name and call-by-value methods for passing parameters.

The R-shell has built in capabilities for handling three main entities: arms, hands and objects. The R-shell maintains an extensive set of data structures used for dynamic creation and manipulation of these entities.

Two main modes for running R-shell currently exist: off-line and on-line. Using the
off-line mode, users can enter and manipulate R-shell data, create manipulators, hands and objects and write, test and debug scripts without endangering any physical devices (i.e. crashing the robot into the wall because of a programming error). This in effect provides a robotics simulator. After the scripts have been fully debugged, the user can invoke the on-line version of R-shell which permits physical connection to real devices.

### 4.2 AV-shell

The AV-shell provides capabilities missing in R-shell for handling AVD’s. These capabilities are provided in much the same way the R-shell provides capabilities for handling manipulators. The interface consists of four groups of commands: camera, head, active image routine and general. Camera commands are used to control cameras and camera lenses, head commands are used to control heads, active image routines provide the levels of active image functions provided by the AV-shell and general commands are R-shell commands which have been extended to handle AVD’s. Here we will discuss camera, head and active image routine commands. A review of important R-shell commands is given in appendix B.

#### 4.2.1 Camera Commands

**camera option value**

Camera is used to control the currently selected camera `cSEL`. If `cSEL` is connected to a target controller, the command may affect the real camera, depending on `option`. Note that this interface is controller specific. Valid options are:

- `-r` - Read lens controller data
- `-w` - Write data to the lens controller
- `-d` - Overwrite lens controller variables with their default values
- `-u` - Update data of `cSEL` with values of global AV-shell vectors
- `-i` - Move focus, zoom and aperture of `cSEL` to their home position.
- `-I` - Initialize the lens controller of `cSEL`
- `-a` - Adjust the aperture of `cSEL` incrementally or absolutely
- `-f` - Adjust the focus of `cSEL` incrementally or absolutely
- `-z` - Adjust the zoom of `cSEL` incrementally or absolutely

**find [-c] objectname**

The coordinates of the center of gravity of object `objectname` is determined and placed in the standard R-shell variable `a`. `Objectname` must be one of the canonical objects known to AV-shell (currently ellipse and square) or must be taught to AV-shell via the `memorize` command.

**memorize objectname**

Object `objectname` is presented to the AV-shell to be memorized. This memorization takes the form of a correlation mask which can later be used in the `find` command.

**c2s**

Map a Cartesian space coordinate to a screen space coordinate using the calibration
matrix in the standard AV-shell variable \( C \). The resulting screen space coordinate will be placed in the standard AV-shell variable \( ss \). Note that there is not an inverse to this function (i.e. \( s2c \)) as we are dealing with only one camera.

### 4.2.2 Head Commands

```
head option value
```

Head is used to control the currently selected head \( hdsel \). If \( hdsel \) is connected to a target controller, the command may affect the real head, depending on option. Note that the interface is controller specific. Valid options are:

- `-r` - Read head controller data
- `-w` - Write data to the head controller
- `-d` - Overwrite head controller variables with their default values
- `-u` - Update data of \( hdsel \) with values of global AV-shell vectors
- `-i` - Move joints of \( hdsel \) to their home position.
- `-l` - Initialize the head controller of \( hdsel \)
- `-j[n][v]` - Modify joint \( n \) of \( hdsel \) incrementally or absolutely

```
g2j [-v]
```

Map gaze space variable to joint space variable for a specific head. This command is the inverse of \( j2g \). Joint angles will be computed and placed into the standard AV-shell variable \( q \). They will be placed according to the given position of the foveated object with respect to the cyclopean frame of the head which is the current value of the standard R-shell variable \( p \).

```
j2g [-v]
```

Map joint space variable to gaze space variable for a specific head. This command is the inverse of \( g2j \). If options are not supplied, the command will set the standard R-shell variable \( p \) to the value of the position of the gaze point with respect to the cyclopean frame of the head.

### 4.2.3 Active image routine commands

As we discussed earlier, active image processes are made up of three levels of routines: high-level activities, medium-level basic processes and low-level routines. A summary of the changes in composition operators given in table 1 is first given in table 2. Then we will give the syntax for the high-level activity and medium-level basic processes as well as a partial list of low-level commands. Note that the structure of these commands is hierarchical in that commands at higher levels are built up of commands from lower levels.

#### High-Level Commands

```
fixate \[ x, y, z \]
```

If head \( hsel \) is not attached to a manipulator, the fixate command will cause the optical axis of the cameras attached to head \( hsel \) to intersect at the 3D point given in \( x, y, z \) if they are given or the value given in the standard R-shell variable \( a \) if they are not
Table 2: Composition Operator Modifications

<table>
<thead>
<tr>
<th>Operator</th>
<th>Original</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Composition</td>
<td></td>
<td>sec</td>
</tr>
<tr>
<td>Concurrent Composition</td>
<td></td>
<td>cnc</td>
</tr>
<tr>
<td>Conditional Composition</td>
<td></td>
<td>cdc</td>
</tr>
<tr>
<td>Disabling Composition</td>
<td></td>
<td>dsc</td>
</tr>
<tr>
<td>Synchronous Recurrent Composition</td>
<td></td>
<td>src</td>
</tr>
<tr>
<td>Asynchronous Recurrent Composition</td>
<td></td>
<td>arc</td>
</tr>
</tbody>
</table>

given. If $hsel$ is attached to a manipulator, the motion necessary for fixation will be divided between the head and the manipulator such that the total joint travel for the two devices is minimized. Fixation is continuously maintained.

Fixate is implemented as the following composition of medium-level commands:

$$\text{fixate} = \text{saccade}\ \text{sec}\ ((\text{disparity}\ \text{cnc}\ \text{accomodate})\ \text{src}\ (\text{verge}))$$

$$\text{pursuit}\ [-c\ \text{cname}]\ [-h\ \text{hname}]\ [-o\ \text{objectname}]$$

Pursuit invokes a continuous cycle whereby the screen coordinates of object $\text{objectname}$ are driven to $(0,0)$. If $\text{objectname}$ is not given, the currently selected object $\text{osel}$ is used. If the -c option is used and camera $\text{cname}$ is attached to a manipulator (i.e. eye-in-hand), pursuit will instruct the manipulator to make the motions necessary to cause the optical axis of camera $\text{cname}$ to intersect with the changing position of the object. If $\text{cname}$ is attached to a head, the head axes will be changed in order to cause the intersection. If the -h option is used, and the head $\text{hname}$ is not attached to a manipulator, the pursuit command will cause the head to track the object. If $\text{hname}$ is attached to a manipulator, the motions required for pursuit will be divided between the head and the manipulator such that manipulability of the manipulator is maximized and pursuit is accomplished.

For heads, pursuit is implemented as the following composition of medium-level commands:

$$\text{pursuit} = (\text{verge}\ \text{cnc}\ \text{motion}\ \text{cnc}\ \text{accomodate})\ \text{src}\ \text{move}.$$  

For single cameras, pursuit is implemented as the following composition of medium-level commands:

$$\text{pursuit} = (\text{motion}\ \text{cnc}\ \text{accomodate})\ \text{src}\ \text{move}.$$
stabilize [-c cname] [-h hname] [-o objectname]
Stabilize behaves like pursuit with the difference being that the perceived motion of
the object is being generated by motions of the platform to which the head is attached
rather than actual object motion. A mobile robot is a typical platform necessitates
the stabilize command.

Medium-Level Commands

accomodate [-c cname]
Accomodate provides a quality measure as to the quality of focus obtained with cam-
era Mname. Accomodate can be used in synchronous recurrent composition with the
camera -f command to implement continuous focus control.

aperture [-c cname]
Aperture provides a quality measure as to the quality of image lighting obtained with
camera Mname. Aperture can be used in synchronous recurrent composition with the
camera -a command to implement continuous iris control.

disparitv
Disparity computes the disparity between two images using the low-level cepstral com-
mand.

motion [-p] [-f size]
The motion command implements both peripheral and foveal motion detection. If
the -p option is used, motion causes a continuous peripheral monitoring for peripheral
motion. If the -f option is used, motion causes a continuous foveal motion detection in
a fovea of size size · size.

saccade [-c cname] [-h hname] [x, y, z]
A saccade is invoked to the 3D point given in x, y, z if they are given or the point given
in the standard R-shell variable a if not. If the -c option is used and the camera Mname
is attached to a manipulator, the saccade command causes the manipulator to make
the motions necessary to cause the optical axis of camera Mname to intersect at the
point. The manipulator will do this with maximum velocity and acceleration. If the
-h option is used and the head is independant or is attached to a manipulator, only
the head joints will move (also with maximum velocity and acceleration).

verge [-o objectname]
Verge causes adjustment of the vergence angles of the cameras with the goal of keeping
the fixation distance consistent with the target of fixation objectname if it is given or
the currently selected object ose1 if not.

Low-Level Commands
cepstral
  Compute the cepstral filter of two images

convolution
  Perform a general convolution. This can be a 2D spatial convolution or a 3D spa-
tiotemporal convolution.

epipole
  Construct epipolar line corresponding to a feature

histogram
  Generate a histogram of an area

laplacian
  Perform Laplacian convolution

median
  Perform general median filter

normalflow
  Compute normal flow field of an area

segmentation
  Perform thresholding of an image region

sobel
  Perform a Sobel edge detection

The AV-shell commands given above provide a transparent interface to a wide range of
active vision devices. In order to install a new active vision device, it is necessary only to
write a single module consisting of device specific mappings.

5 Combining Perception and Action

Combining the perceptive capabilities of AVD’s with the action capabilities of other robotic
devices in an agent architecture poses some interesting problems. On one hand, we must view
perception and action on a micro-level looking only at the AVD as it operates independently
in an perception-action loop. On the other hand, at the macro-level, the perceptions provided
by the AVD are used to drive actions of other robotic devices. In this section we will look at
architectural issues related to the combination of perception and action at both the micro
and macro levels.

The control of robotic devices can be conveniently divided into three levels: high, medium and low. Figure 3 shows this relationship.

High-level control usually refers to task-level planning. At this level, reasoning is incor-
porated into the control system with methods such as knowledge-based systems, fuzzy logic
and artificial neural networks. An example is the reasoning behind chess playing algorithms.
Typical output from the high-level includes Cartesian information such as the position and orientation of various pieces on the chess board and to where they should be moved.

The medium-level takes output from the high-level and converts it into a form that is suitable to the joint controllers. In the case of the chess playing robot, the medium-level will convert the Cartesian position and orientation information of the pieces provided by the high-level into the joint values needed by the robot at the low-level to move the pieces. As an aid to research, the medium-level should provide an interactive interface so that researchers can bypass the high-level task planners.

Low-level controllers deal with trajectory generation, sensor integration and servo joint control. This is the level where connection to physical devices is made.

### 5.1 Active-Vision integration and Active-Image processing

While this hierarchy is appropriate for devices such as manipulators, it is insufficient for active image processing and the various compositions of active vision routines, both of which are required in AVD’s. Active vision processing is needed to compose lower level active image routines into higher level activities. These higher level activities in turn must be composed according to changing tasks and and effectively invoked. Effective invocation requires system delays to be compensated for by the controller (using a Smith-controller or other techniques).

In order to modularize these various levels of processing, we feel that in the case of AVD’s, two additional levels are needed in the control hierarchy of figure 3. The new levels are called the “Active Vision Integration Level” (AVIL) and the “Active Image Processing
The hierarchy of figure 3 has been augmented with the new levels. The augmented hierarchy is shown in figure 4.

According to the diagram, the low-level is responsible for providing pre-attentive image processes. The AIPL performs filtering and and fusion to provide data to the set of attentive routines found in the AVIL. The AVIL in turn is responsible for composing the higher level primitives according to changing requirements and compensating system delays. All three of these levels are accessible from the command interface at the medium-level facilitating development and simulation.

6 Implementation

By the nature of the different speed requirements necessary for interactive and real-time processing at the medium and AVIL, AVPL and low levels, these components should be decoupled. The interactive and developmental components of the AV-shell should make use of well known development tools. The C programming language and Unix operating system provides just such tools, therefore, these components of the system are implemented in C under Unix.² Unix, however, is not a real-time operating system and is not applicable for the time requirements of the AVIL, AVPL and low-level subsystems. Therefore, a separate

²The use of C and Unix provide portability and are accepted in the vision and robotics communities as standard.
AVII, AVPI and low-level module is employed which uses hardware/software applicable to real-time activities. This module is called the “Target Machine”. Figure 5 shows the logical system configuration.

![Logical System Architecture](image)

**Figure 5: Logical System Architecture**

In this configuration, the Host machine runs the following processes:

- **R** Process executing R-shell/AV-shell
- **H** Host server

and the Target machine runs the following processes:

- **T** Target server
- **AVI** Active vision integration process
- **AIP** Active image processing process
- **CNT** Control processes
- **S** Sensor processes
- **IO** IO process

The R-shell/AV-shell process **R** interacts with the user and generates messages whenever a command relating to a camera or head is issued. The host server **H** receives messages from the target machine and updates shared host machine local variables.

On the target machine, the target server **T** receives messages from the host and uses the messages to update shared target machine local variables which are then interpreted by other target processes. The Active Image Processing process **AIP** is responsible for composing the primitive pre-attentive active vision routines into data usable by the higher level attentive routines and performing data reduction. The AV-integration process **AVI** is responsible for
compensating system delays and determining the appropriate composition of higher level attentive AV processes given changing requirements. The control process CNT provides servo control. Sensor processes S, monitor selected data and periodically inform the host machine, AIP and AVI processes about changes. The input-output process IO is used for communicating with physical devices being controlled.

The physical system being developed at the Technion consists of a 90 MHz Pentium PC running the DOS operating system. Installed in the PC are special purpose cards for doing image processing, lens control and servo control. A drawback to DOS is its lack of multi-tasking capabilities. The logical design described above lists several processes which must run in parallel on the target machine. This cannot be done under DOS. We solve the problem by using special purpose boards for several of the processes. These boards run independently of the CPU and are in essence independent processes. Only the T, AIP and AVI processes run on the CPU, however, the speed of the 90 MHz Pentium should overcome any problems related to this lack of parallelism. Figure 6 shows the physical system configuration being developed at the Technion.

![Figure 6: Physical System Architecture](image)

7 Example AV-shell Usage

In this section, we will demonstrate how the AV-shell can be used to easily implement a complex perception-action task. In the example, the AV-shell will be used to coordinate the activities of three robotic devices: a manipulator, a dextrous robot hand and an AVD (head). The scenario, which is illustrated in figures 7 and 8, consists of an AdeptOne manipulator with an attached Belgrade/USC dextrous hand [14]. Additionally, there is a table in the workspace of the AdeptOne on which a cylindrical toy car is moving. Observing the movement of the car is an Aalborg robotic head [8]. The goal of the example is to have
the head track the car and signal when the car is about to fall off an edge of the table. The arm waits for the signal at which point it is commanded to bring the hand to the car so that it can be grasped before it falls.

Figure 7: Example Environment

Figure 8: Example Environment

The AV-shell script necessary to perform this task is made up of two parts. In the first part, the internal representations of the environment and devices are created. This includes creating and locating the manipulator, hand, table, cameras and head. In the second part, we activate the head and perform the task.

Creating the internal representation of the environment in the AV-shell includes defining parameters of the various components present. Each robotic device has a set of parameters required for its control which must be defined. This includes things such as minimum and
maximum joint values etc... Additionally, the components must be initially located relative to the universe and devices attached to other devices (i.e. hand on manipulator and cameras on head) in reality must also be attached in the internal representation. Also, the description of the car must be memorized by the AV-shell.

Once the environment has been defined, we are ready to begin the task. The first step is to connect the internal description of the devices to their physical counterparts. This is accomplished with the connect command. Once connected, commands causing changes in the internal configuration will effect the physical devices as well. If we execute the task without connection to the physical devices, we have simulation. Next, using the AV-shell commands find and saccade, we locate the car in the environment, invoke a saccade to its position and begin smooth pursuit tracking (this is an example of where perception-action takes place at the micro-level of the active vision device). When the car approaches an edge of the table, the event e1 is set signaling the arm to bring the hand to the estimated falling point of the car (this is an example of where perception-action takes place at the macro-level of the integrated system). At the same time, the hand is preshaped for grasping the car. When the arm has brought the hand to the grasping point and the hand is preshaped, the car is grasped.

In this example, specific robotic devices were used. The same code is applicable to any other devices known to the AV-shell. For instance, we could have used a PUMA manipulator and a KTH head instead. The AV-shell script for performing this task is given below.

```
# Create environment for grasp task
# Create and locate manipulator:
# Here we define parameters for the adeptone manipulator type, create an
# instance called M and locate it relative to the universe.

cop -a
# default category option is -a (arm)
type adeptone
# create arm type name
par a 325.0 375.0 0.0
# define link parameters

inst adeptone M
# create an arm instance called M
loc
# locate arm M at Universe
tran 0 100 150
# translate arm base
rot = 90
# rotate arm base about z-axis by 90 degrees

# Create and locate dextrous hand:
# Here we define parameters for the USCBGD hand type, create an instance
# called H and attach H to M.

cop -h
# change default category option to -h (hand)
type USCBGD
# create hand type name
par gmin -30, -45, -90, -90
# define hand parameters

inst USCBGD H
# create a hand instance called H
open M
# prepare manipulator for attachment
attach H M
# mount hand onto arm
close M
# disable further attachments
```
### Create and locate table:

- change default category option to -o (object)
- create table name
- locate table at Universe
- translate table base about z-axis by 90 degrees

### Create and locate head:

- change default category option to -d (head)
- create head type name

### Create and locate cameras:

- change default category option to -c (camera)
- create camera type name
- create camera instances called CL and CR
- prepare head for camera attachment
- select camera CL
- mount CL onto HD
- translate camera CL to final position on HD
- select camera CR
- mount CR onto HD
- translate camera LR to final position on HD
- disable further attachments on HD

### Memorize description of the car

- AV-shell memorizes the image of the car
- make car the selected object

---

**Tracking and grasping the car**

- create event names
- connect manipulator M with real arm (port 1)
- connect head HD with real head (port 2)
- connect hand H with real hand (port 3)
- change default category option to -d (head)
- locate the car in the environment
- saccade to the car and begin immediate tracking
- set e1 when car approaches table edge
- this includes estimate of falling location
- wait for event e1 to occur
8 Conclusion

In this paper, we addressed issues related to agent architectures in active vision research. We presented the agent architecture/active vision research tool called the AV-shell. The AV-shell provides a programming framework for expressing perception and action routines as well as the data structures and techniques necessary to integrate a wide range of active vision devices into complex robotic architectures. We showed that in order to provide a general mechanism for controlling a variety of active vision devices it is convenient to view the vision sensor and its movement mechanism separately. This lead to a definition of an operational space and the definition of data structures appropriate for controlling a wide range of active vision devices. We then discussed the importance of a hierarchical set of active image routines and showed how they can be composed at various levels to implement complex visual behaviors. The syntax for AV-shell commands was then presented. Finally we discussed implementation issues and showed, through an example, how the AV-shell can be used to implement complex perception-action tasks.

Future work on the AV-shell includes research into effective composition of attentive routines given changing tasks, effective compensation of system delays both at the micro-level of the active vision device and at the macro-level of the integrated system. Additionally, we will continue research into robust implementations of the active image processing routines.
Appendices
# define CAMC_MAX 2
    /* Maximal number of camera classes */
# define CAMT_MAX 5
    /* Maximal number of camera types */
# define CAMI_MAX 5
    /* Maximal number of camera instances */
# define HEADC_MAX 2
    /* Maximal number of head classes */
# define HEADT_MAX 2
    /* Maximal number of head types */
# define HEAD_MAX 2
    /* Maximal number of head instances */

# define HEAD_ID 99
    /* Head identifier */
# define HEAD_ID 30
    /* Head type identifier */
# define CAMERA_ID 31
    /* Camera identifier */
# define CAMERAT_ID 32
    /* Camera type identifier */

# define CAM1 0
# define CAM2 1

# define RPAN 2
    /* called these RPAN, RTILT and RVERGENCE */
# define RTILT 3
    /* because PAN, TILT and VERGENCE conflict with */
# define RVERGENCE 4
    /* head table members */
# define RVERGENCE 5

# define NATTRIBUTES 11
# define FOCUS 0
    /* various head attributes */
# define ZOOM 1
# define APERTURE 2
# define CYCLOTORSION 3
# define UNCOUPLEDVERG 4
# define COUPLEDVERG 5
# define INDEPEYE_TILT 6
# define COUPLED_EYE_TILT 7
# define INDEP_PAN 8
# define COUPLED_PAN 9
# define ADJUSTABLE_BASE 10

typedef struct {
    char NAME[16];  /* Head class name */
    int USE;       /* USE=1 indicates class existence */
    int N;        /* Number of Joints for the class */
    int NCAMERAS; /* Number of Cameras for the class */
    int JT[JPH_MAX]; /* Joint types (0 - rotary, 1 - prismatic) */
    int ATTR[NATTRIBUTES]; /* Various head attributes that define a head */
    double BASE_DISP; /* Baseline displacement (m) */
    double TILTDISP;  /* Tilt displacement (m) */
} headClasstable;

extern headClasstable HEADCT[HEADC_MAX+1]; /* HeadClassTable[HEADCT[0] not used */
/* Head type table */
extern int HEADTT_LAST; /* Index of the last entry in HEADTT table */
extern int HEADTYPE_SEL; /* Index of the selected head type */

typedef struct {
  char NAME[16]; /* head type name */
  int FREE; /* FREE=1 indicates free entry of HDT-table */
  int USE; /* USE >= 1 indicates number of head instances of that head type, zero if no instances */
  int CLASS; /* head class. One of 1..9 */
  int NJOINTS; /* number of head joints */
  int NCAMERAS; /* number of cameras supported by head */
  double qMIN[JPH_MAX+1]; /* Minimal values (rad) of joint angles */
  double qHOM[JPH_MAX+1]; /* Nominal values (rad) of joint angles */
  double qMAX[JPH_MAX+1]; /* Maximal values (rad) of joint angles */
  double WMAX[JPH_MAX+1]; /* Maximal joint velocity (Controller) */
  double KP[JPH_MAX+1]; /* Proportional feedback gain (Controller) */
  double KV[JPH_MAX+1]; /* Velocity feedback gain (Controller) */
  double KI[JPH_MAX+1]; /* Integral feedback gain (Controller) */
  double DB[JPH_MAX+1]; /* Dead band (Controller) */
  double BIAS[JPH_MAX+1]; /* Bias - to fight friction (Controller) */
  double TAUMAX[JPH_MAX+1]; /* Maximal torque applied by the motor (Controller) */
  double ACCT[JPH_MAX+1]; /* Acceleration time (Controller) */
  double ERRLIMIT[JPH_MAX+1]; /* Error limit (Controller) */
  int DD[JPH_MAX+1]; /* Drive mode (interpolated, numeric) (Controller) */
  int MM[JPH_MAX+1]; /* Motor mode (force, posit-rel, posit-abs) (Controller) */
} headTypetable;
extern headTypetable HEADTT[HEADT_MAX+1]; /* Headtype table (HEADTT[0] not used) */

/* Head table (contains head instances) */
extern int HEAD_LAST; /* Index of the last entry in HEAD-table */
extern int HEAD_SEL; /* Index of the selected head instance */
extern int HEAD_TYPE; /* Head type (index of HDT-table) */
extern int HEAD_CLASS; /* Head class (index of head class) */

int HEADCAM[CPH_MAX]; /* Pointer to camera instance(s) mounted on head */
double BASELINE; /* baseline between two cameras on head */
double PAN; /* pan angle */
double TILT; /* tilt angle */
double VERGENCE[CPH_MAX]; /* vergence angle for each camera on head */
double VERSION; /* version angle */
vector3 GAZE_POINT; /* point of interest is located at intersection of the optical axis. */

homo TB; /* Homogeneous transform of the head base */
  (relative to universal coordinate system) */
homo CW[,CPH_MAX]; /* Homogeneous transform of each camera's focal */
double Q[,JPH_MAX+1]; /* Joint angles */
/* Parameters that follow are controller param. */
/* Which are saved/stored from/to the controller */
double QHOM[,JPH_MAX+1]; /* Nominal joint values */
double WMAX[,JPH_MAX+1]; /* Maximal joint velocity (Controller) */
double KP[,JPH_MAX+1]; /* Proportional feedback gain (Controller) */
double KV[,JPH_MAX+1]; /* Velocity feedback gain (Controller) */
double KI[,JPH_MAX+1]; /* Integral feedback gain (Controller) */
double DB[,JPH_MAX+1]; /* Dead band (Controller) */
double BIAS[,JPH_MAX+1]; /* Bias - to fight friction (Controller) */
double TAUMAX[,JPH_MAX+1]; /* Maximal torque applied by the motor (Controller) */
double ACCT[,JPH_MAX+1]; /* Acceleration time (Controller) */
double ERLIM[,JPH_MAX+1]; /* Error limit (Controller) */
int DD[,JPH_MAX+1]; /* Drive mode (interpolated, numeric) (Controller) */
int MM[,JPH_MAX+1]; /* Motor mode (force, posit-rel, posit-abs) (Controller) */
} headTable;
extern headTable HEADT[,HEAD_MAX+1]; /* Head table */

/* Camera Class Table (cameras include lens) */
typedef struct {
    char NAME[16]; /* Hand class name */
    int USE; /* USE = 1 indicates class existence */
} cameraClasstable;
extern cameraClasstable CCT[,CAMC_MAX+1]; /* Camera class table */

/* Camera Type Table (cameras include lens) */
typedef struct {
    char NAME[16]; /* Camera type name */
    int FREE; /* FREE=1 indicates free entry of CT-table */
    int USE; /* USE >= 1 indicates number of camera instance */
    /* ... instances of that camera type, zero if none */
    int CLASS; /* camera class. One of 1..9 */
    double FOCAL; /* focal length of camera - 0 if controllable */
    /* whereby FOCAL is interpreted as zoom. */
    double XPARM; /* pixels/mm in x direction */
    double YPPM; /* pixels/mm in y direction */
} cameraTypetable;
extern cameraTypetable CAMTT[,CAMT_MAX+1]; /* Camera type table */

/* Camera Table (contains camera instances) */
typedef struct {
    char NAME[16]; /* Camera name */
    int FREE; /* FREE=1 indicates free entry of CM-table */
    int LOC; /* LOC=1 camera located */
    int PORT; /* port that camera is connected to, 0 if not */
} cameraClasstable;
extern cameraClasstable CAM[,CAM_MAX+1]; /* Camera Table */

extern int CT_LAST; /* Index of the last entry in CT table */
extern int CT_SEL; /* Index of the selected camera type */

extern int CAMERA_LAST; /* Index of the last entry in CM-table */
extern int CAMERA_SEL; /* Index of the selected camera instance */
extern int CAMERA_TYPE;
extern int CAMERA_CLASS;

/* Camera Classtable */
extern cameraClasstable CCT[,CAMC_MAX+1]; /* Camera class table */
/* Camera Type Table */
extern cameraTypetable CAMTT[,CAMT_MAX+1]; /* Camera type table */
/* Camera Table */
extern cameraClasstable CAM[,CAM_MAX+1]; /* Camera Table */
int MNP;  // Pointer to manipulator that supports camera
//  (index of MNP-table)
//  (MNP = 0 - hand is not mounted onto manip.)
int HEADPTR;  // Pointer to head that supports the camera
//  (index of HEAD-table)
int TYPE;  // Camera type (index of CT-table)
double FOCAL;  // Focal length (mm)
double FOCUS;  // Focus point (m)
double APERTURE;  // Diameter of lens opening (% open)
homo TH;  // Homogeneous transform of the camera base with
//  respect to the universal coordinate system
homo TW;  // Homogeneous transform of focal frame with
//  respect to the camera base.
homo CALIBRATION;  // Calibration matrix with respect to universe
) cameraTable;
extern cameraTable CAMT[CAMT_MAX+1];  // camera table (CAMT[0] not used)
A description of R-shell commands is provided below. These commands are made of five groups; (1) basic R-shell commands; (2) manipulator commands; (3) hand commands; (4) physical connection commands; and (5) device integration commands.

### B.1 Basic R-shell commands

Basic R-shell commands are used to define and manipulate data and variables. Some of these commands have an optional parameter called “Category Option” (cop). If cop is not given, a default value will be taken from the current category option which is set with the cop command. Cop indicates which type of entity will be effected by the command. Valid category options are -a (arm), -h (hand) and -o (object). The following is a list of basic R-shell commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>aa2r</strong> [aa[1] aa[2] aa[3] angle]</td>
<td>The orientation matrix which corresponds to equivalent axis aa and angle will be computed and placed in the standard R-shell variable R. If arguments are not supplied, the current values of the of the standard R-shell variable a and s will be used instead.</td>
</tr>
<tr>
<td><strong>align</strong> [cop] [ref]</td>
<td>The coordinate axes of sel are made parallel to those of the reference manipulator, hand, camera, head or object ref, where sel is the currently selected manipulator, hand, camera, head or object as specified by the category option cop. The position of sel is not changed. Ref may be any located manipulator, hand, camera, head or object and must not be in the same tree as sel. If the reference is omitted or is U, sel is aligned with the universal frame.</td>
</tr>
<tr>
<td><strong>attach</strong> [cop] [name] pname</td>
<td>This command will rigidly attach the selected arm, hand, camera, head or object if cop is given, or the specified arm, hand, camera, head or object if name is given to the arm, hand or object pname. Attachment will occur in a frame-to-frame manner and pname will become the parent of the arm, hand, camera, head or object to be attached. The parent must be opened before attachment using the open command.</td>
</tr>
<tr>
<td><strong>axis</strong> name</td>
<td>Axis will compute the magnitude and the unit vector which correspond to the specified vector name and place the results into the standard R-shell variables s and a respectively.</td>
</tr>
<tr>
<td><strong>calibrate</strong> name</td>
<td>The calibration routine for the manipulator, hand, camera or head given in name is invoked. In the case of a camera, the transformation matrix relating camera name if name is given or the currently selected camera csel to the universal frame is determined and placed in the standard AV-shell variable C.</td>
</tr>
</tbody>
</table>
The manipulator, hand, head or object \((name)\), which may be primitive or composite, is closed to attachments and detachments. If \(name\) is a composite object, operations such as trans, rot and align will effect the object as a whole, and internal parts cannot be relocated with respect to each other.

**cop category-option**
Category options \(-a\) (arm), \(-h\) (hand), \(-c\) (camera), \(-d\) (head), \(-o\) (object)) can be defined and omitted later in commands type, par, loc, trans, rot and align. The current default category option can not be altered by these commands.

**def name expression ...**
The variable name is set to the value of the expression given by \(expression\). The target can be a standard or a user-declared variable. \(Expression\) is an R-shell expression, written in infix notation, using operands, operators and parentheses. Operands can be scalar constants, variables or function designators.

**delete names**
The corresponding names are removed from the R-shell environment provided certain conditions are met. For example, if \(name\) is a manipulator, it cannot be holding a hand, camera, head or object.

**detach name**
This command allows detachment of arms, hands, cameras, heads and objects. The parent of \(name\) must be open. If \(name\) is an object, the subtree \(name\) will be detached from the object \(o(name)\). the detached part will now become a free object and the shape of the tree will be restored to what it was before it was attached.

**disp [option] expression \([ >> filename]\)**
This command is used to display values for R-shell expressions. \(Expression\) may be a standard R-shell variable, a user-defined variable or any valid R-shell expression. Multiple values for vectors and arrays are separated by spaces.

**enter [-ascii] variable \([- < filename]\]**
Enter is used to interactively input values for R-shell variables. \(Variable\) may be a standard R-shell variable or a user-defined variable. Input is taken from the keyboard by default, but can be taken from a file if the redirection option is used. If the -ascii option is used, the ascii representation of the characters entered will be stored in \(variable\).

**fun name fexpression**
The user-defined function with the name \(name\) is created. The function body is specified by \(fexpression\) which is an R-shell expression in which operands can also be parameters. Function parameters are denoted by \(\$i\) where \(i\) specifies the position of the corresponding actual parameter in the function designator.
get variable
This command is used with the -n option of R-shell to send variable values to a remote machine. Variable may be any standard R-shell variable or a user-defined variable. R-shell responds to this command by sending an integer which represents the number of values to be sent followed by the values themselves.

inst type-name instance-name(s)
Instances of type type-name are created. Upon instantiation, type type-name becomes the selected type and the last instance of instance-name becomes the selected arm, hand, camera or head. The type data of type-name is transferred to the instance data of instance instance-name.

loc [cop] [ref]
The coordinate frame of sel is placed such that it coincides with that of the reference frame ref, where sel is the currently selected manipulator, hand, or object, as specified by the category option cop.

obj [name ...]
Obj introduces primitive objects into the R-shell environment. Newly created objects are initially closed, unlocated and not held by any manipulator or hand.

open pname
The manipulator, hand, head or object (pname), which may be primitive or composite, is opened enabling attachment of new parts or detachment or relocation of existing parts. Composite objects must be opened before internal parts can be relocated.

par [cop] parspec value
The parameter parspec is set to value for the selected manipulator or hand type.

prt [-n] [-p] message - prt -c
In the first form, message is printed on the screen followed by a newline. The newline may be suppressed with the -n option. If -p is given, R-shell will pause for input from the user after printing message. The second form is used to clear the terminal screen.

r2aa
The equivalent axis and angle of rotation are computed for the orientation matrix, the current value of R-shell variable R. The equivalent axis of rotation (vector) is placed in standard R-shell variable a and angle is placed in standard R-shell variable s.

r2rpy
The roll-pitch-yaw angles are computed for the orientation matrix, the current value of the standard R-shell variable R, and placed into the standard R-shell variable a. In the degenerate case (pitch = ± or -90 degrees), the yaw angle is set to zero.

r2xyz
The x-y-z Euler angles are computed for the orientation matrix, the current value of the standard R-shell variable R, and placed into the standard R-shell variable a.
The z-y-z Euler angles are computed for the orientation matrix, the current value of the standard R-shell variable $\mathbf{R}$, and placed into the standard R-shell variable $\mathbf{a}$. In the degenerate case ($\beta = 0$ or 180 degrees), the first Euler angle is set to zero.

**repeat**

The last R-shell command is repeated. This command is suitable for manual incremental motions.

**restore prefix**

An R-shell state is explicitly restored from the file *prefix.d* if a file name is specified, otherwise, the state of R-shell will be restored from the current data file. R-shell data environment is always implicitly restored at each R-shell login.

**rot [cop] [ref] [axis phi]**

Sel is rotated about the axis by $\phi$, in terms of the coordinate system $\text{ref}$, where $\text{sel}$ is the currently selected manipulator, hand or object as specified by the category option $\text{cop}$.

**rpy2r [alpha beta gamma]**

The orientation matrix which corresponds to roll-pitch-yaw angles $\alpha$, $\beta$, and $\gamma$ will be computed and placed in the standard R-shell variable $\mathbf{R}$. If arguments are not supplied, the current values of the standard R-shell variable $\mathbf{a}$ will be used instead.

**save prefix**

The current state of R-shell is explicitly saved to the file *prefix.d* if a file name is specified, otherwise the state of R-shell will be saved to the current data file.

**sel name**

The manipulator, hand, camera, head, object or type referenced by *name* becomes the “selected” manipulator, hand, camera, head, object or type respectively until a different one is selected by using the sel command again.

**show name or spec**

Show *name* will display the R-shell entity *name*, which can be a variable, function, event, arm or hand instance, arm type or hand type. Show *spec* will display various R-shell information, depending on the specifier *spec*.

**stop [inst ...]**

The current motion of the named arm and/or hand and/or head instances *inst* is halted immediately. If no arguments are supplied, the current motion of all connected arms, hands and heads is stopped.

**tran [cop] [ref] [axis delta]**

The selected manipulator, hand, camera, head or object *sel*, as specified by the category option *cop*, is translated according to the given arguments.
**type [cop] class tname**

This command creates a new arm, hand, camera or head type. The type is given the name \textit{tname}, and is of class \textit{class}. The class parameter determines the type of manipulator, hand, camera or head. Valid classes are in the range 1-9. Type name is an R-shell identifier.

**var type names**

A single variable, or several variables with names specified in the list \textit{names} are created. The variable will be of type \textit{type}.

**wait**

Wait delays interpretation of the next R-shell command until completion of the current motion of all connected arms and hands.

**where [cop] [ref]**

The homogeneous transform matrix of \textit{sel} with respect to the reference manipulator, hand or object \textit{ref} is computed and displayed on the screen.

**xyz2r [α β γ]**

The orientation matrix which corresponds to x-y-z Euler angles α, β and γ will be computed and placed in the standard R-shell variable \textbf{R}. If arguments are not supplied, the current values of the standard R-shell variable \textbf{a} will be used instead.

**zyz2r [α β γ]**

The orientation matrix which corresponds to z-y-z Euler angles α, β and γ will be computed and placed in the standard R-shell variable \textbf{R}. If arguments are not supplied, the current values of the standard R-shell variable \textbf{a} will be used instead.

### B.2 Manipulator Commands

The R-shell user creates and manipulates images of real-world manipulators and environments by manipulating instances of these items with R-shell commands. The following commands are used for the manipulation of manipulators.

**arm option**

Arm is used to control the currently selected manipulator \textit{msel}. If \textit{msel} is connected to a target controller, the command may affect the real manipulator, depending on \textit{option}. Valid options are:

- **-r** - Read arm controller data
- **-w** - Write data to the arm controller
- **-d** - Overwrite controller variables with their default values
- **-u** - Update internal data of \textit{msel} with values of the global R-shell vectors
- **-i** - Move joints of \textit{msel} to their home position
- **-l** - Initialize the controller of \textit{msel}
- **-j** - Move joint of \textit{msel} incrementally of absolutely
c2j [-v][-F][-r][-l][-a][-b][-f][-n]
Map Cartesian space variable to joint space variable for a specific manipulator. This command is the inverse of j2c. If options -v and -F are not supplied, only the joint angles will be computed and placed into the standard R-shell variable q. They will be placed according to the given position and orientation of the manipulator’s wrist with respect to the manipulator’s base, which are the current values of the standard R-shell variables p and R respectively.

grip value
Grip opens and closes the gripper of the selected arm instance. Value represents the desired aperture of the gripper in millimeters and may be from 0 to 90 inclusive. The selected manipulator must be connected to use this command.

j2c [-v][-F]
Map joint space variable to Cartesian space variable for a specific manipulator. This command is the inverse of c2j. If options are not supplied, the command will set the standard R-shell variables p and R to the values of the position and orientation of the manipulator’s wrist, both with respect to the manipulator’s base, for given values of the link parameters which correspond to that type.

B.3 Hand Commands
The following commands are used for the manipulation of hands.

hand option
Hand is used to control the currently selected hand hsel. If hsel is connected to a target controller, the command may affect the real hand, depending on option. Valid options are:
- -r - Read hand controller data
- -w - Write data to the hand controller
- -d - Overwrite controller variables with their default values
- -u - Update internal data of hsel with values of the global R-shell vectors
- -i - Move joints of hsel to their home position given in GHOM
- -l - Initialize the controller of hsel
- -j - Move joint of hsel incrementally of absolutely
- +s - Establish continuous sensor data reading
- -s - Terminate continuous sensor data reading

shape [-gk][[\^] \delta]
The -g parameter specifies a new GGM to follow. If it is not supplied, the previous value of GGM is used (the default GGM “0” is used if no previous value exists.) If \delta is prefixed by “\^” the value is taken to be an increment to the current value. If “\^” is omitted, the value is taken to be absolute.
grasp \([\wedge] \ f\]

If \( f\) is prefixed by “\(\wedge\)”, the value of \( f\) will be taken to be an increment to the current desired reflex forces. This can be used to tighten or loosen the grasp. If the prefix “\(\wedge\)” is omitted, the value is taken to be absolute. The ungrasp command is used to terminate force control and expand the hand until all forces become 0.

B.4 Connection to Real Devices

Manipulator and hand instances created in R-shell are just images of the real world. The commands previously described create and modify this image of the real manipulator or hand. In order to cause changes to an actual device, this image must be connected to the real device. When devices are physically connected to R-shell in this way, R-shell commands which affect the configuration and position of the manipulator or hand internally will now also affect the configuration of the physical device. Once connection is established, data in the manipulator or hand controller and in R-shell can be shared. In this way, R-shell has access to the real values generated in the controller and can use them for its own purposes.

\[
\text{connect}[-e,-s,-b] \text{ instance-name [server-name]}
\]

This command establishes a physical link from R-shell to the device via software called the target server. With this physical link established, commands that affect the configuration of the manipulator or hand internally will now also cause motion of the physical device. Supported communications methods are: \(-e\) (ethernet sockets), \(-s\) (serial link), \(-b\) (backplane - shared memory). If the \(-e\) option is specified, the parameter \text{server-name} must also be specified.

\[
\text{disconnect} \text{ instance-name}
\]

This command causes the connected manipulator or hand to be disconnected. Subsequent commands will not cause changes to the physical device.

B.5 Multiple Device Integration

\[
\text{event name }\ldots
\]

Event introduces event flags into the R-shell environment. Newly created event flags are initially cleared.

\[
\text{delete event-names}
\]

The corresponding event names are removed from the R-shell environment.

\[
\text{set name }\ldots
\]

The named event flags are set. The status of an event flag can be displayed by show (RSHELL).

\[
\text{clear name }\ldots
\]

The named event flags are cleared. The status of an event flag can be displayed by the show command.
wor name...
   Wor delays interpretation of the next command until any of the named event flags in list name... have been set.

wand name...
   Wand delays interpretation of the next command until all of the named event flags in list name... have been set.

An event can be automatically set after the R-shell commands (hand, shape, grasp, ungrasp, arm, loc, rot, tran or align) by adding the command line suffix "e," which specifies the event flag to be set by the controller when the motion indicated by the command is completed.

References


