Mobile Robotics 1997

Jessica Bayliss, Chris Brown, Rodrigo Carceroni
Christopher Eveland, Craig Harman, Amit Singhal, Mike Van Wie

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 661

August 1997

Abstract

After abandoning an attempt to build our own gasoline-powered automated outdoor vehicle in 1995, we purchased two M68332-controlled wheelchairs for indoor and outdoor mobile robotics research. Much of the first year has been spent on various infrastructure projects, several of which are described here. At this writing we are beginning to be in a position to do nontrivial applications and research using these platforms. This compendium of facts and experiences is meant to be useful in getting to know the organization and capabilities of our mobile robots. We first cover the basic hardware and the serial protocol used to communicate between the main computing engine and the microcontroller responsible for sensor management, motor control, and low-level sensori-motor control loops. We describe the interface to the video digitizer, a low-level obstacle avoidance routine, and a general software organization for a control architecture based on video streams. Dynamic nonholonomic models and a virtual environment for debugging and experimenting with them are described next, followed up by a visual servoing application that uses "engineered vision" and special assumptions.

This material is based on work supported by the Luso–American Foundation, Calouste Gulbenkian Foundation, JNICT, CAPES process BEX 0591/95-5, NSF IIP grant CDA-94-01142, DARPA DURIP MDA972-92-J-1012, NSF Research Instrumentation IRI-9202816, and DARPA VSAM contract DAAB07-97-C-J027.

Contents

1 Hardware **3**

2 The Serial Protocol **5**

2.1 Goals . . . 5

2.2 ARC . . . 5

Parallel programming . . . 6

Multi-streamed Serial I/O . . . 6

2.3 Architecture: ARC . . . 7

Processing I/O . . . 8

Sensor readings . . . 8

Starting the Serial Protocol . . . 11

2.4 Architecture: Linux . . . 13

Initialization . . . 14

Motor Library Connection Attempts . . . 14

2.5 The C Programming Interface . . . 17

The motor library . . . 17

The sensor library . . . 18

2.6 Timing and Bandwidth Issues . . . 19

Asynchronous Command Flow Control . . . 19

Monitoring serial line bandwidth . . . 21

2.7 Extending the serial protocol . . . 21

3 Obstacle Avoidance **23**

4 A Software Infrastructure for Mobile Robotics **25**

4.1 Architecture . . . 25

VideoBuffers . . . 25

The Scene Description Bus . . . 26

Putting it all Together . . . 26

4.2 User's Manual . . . 26

VideoBuffer API . . . 27

SDBus Client API . . . 29

The SDBus Server . . . 31

The SDBus in LISP . . . 32

4.3 A Sample Application . . . 32

5 Low Level Control to Enable Visual Servoing **35**

5.1 Hardware . . . 35

5.2 Software . . . 36

5.3 Limitations . . . 38

6 A Virtual Environment Testbed for Driving a Wheelchair **40**
6.1 The Possibilities of Simulation 40
6.2 Simulator Dynamics 41
6.3 *PerSim*: A program for Virtual Reality Simulation 41
6.4 Plans for Using the Virtual Wheelchair Environment 44

7 A Frame-Grabber Abstraction **45**
7.1 URCS Users 45
7.2 Installation 46
7.3 The Basic Programming Paradigm 47
Using Command Line Arguments 48
7.4 Configuration Options 49
Capture Modes 49
Grabber Output Geometry 51
Grabber Input Source 53
Frame Acquisition Frequency 53
7.5 Interacting with the Frame Grabber 54
Using the Instrumentation Data 54
Changing the Grabber Configuration During Execution 55

8 A Visual Servoing Application: Chair-Following **57**
8.1 Introduction 57
8.2 Background 57
8.3 Tracking the Target with the Use of 3-D Information 61
8.4 Efficient Low Level Image Processing 66
8.5 Visual Control and Experimental Performance 68

9 Conclusions **70**

10 Acknowledgements **70**

A Code Fragments for Wheelchair Dynamic Simulation **74**

B Data Sheets for Shaft Encoders and Electronics **84**

Figure 1: One of the automated wheelchairs.

1 Hardware

Our twin Vector Mobility wheelchairs are modified by the Kiss Institute of Practical Robotics (see `http://www.kipr.org/`) with an Onset M68332 micro-controller, a sonar range sensor, bump sensor, odometry on each wheel, and a large complement of IR proximity sensors. Commands can change the velocity or the steering direction of the chair, though the individual wheels are not individually controllable (the Onset's output is further interpreted by a proprietary controller that generates PCM signals to the two wheel motors given output either from the microcontroller or from a joystick manual control) (Fig. 1).

Among other hardware modifications, we added a twin-Pentium computer from Real World Interfaces (RWI) (see `http://www.rwii.com`), to run Linux, fitted heavier batteries (DieHard Marine Starting Deep Cycle), enhanced RWI's DC-to-DC converter with multiple 5V and 12V output jacks, added a Matrox Meteor digitizer and two sorts of wireless ethernet connectivity (Fig. 2). We have upgraded the odometry of one chair to use a bi-directional shaft encoder (Model 230 from Encoder Products Co., with a US Digital Corp LS7083 encoder-to-counter interface chip).

Two Indoor-Outdoor Automated Wheelchairs

Figure 2: Block diagram of wheelchair hardware complement. A Cognachrome system from Newton Labs (see `http://www.newtonlabs.com`) will be included in the near future, and both the wheelchairs will have Aironet wireless ethernet bridges, with one chair being connected to the department network.

2 The Serial Protocol

2.1 Goals

Our autonomous wheelchairs have a powerful twin Pentium Linux processor for high-level reasoning, vision processing, communications, and generally, computation that does not involve wheelchair control or sensor reading. The serial protocol provides a standard C library interface for control of the wheelchairs from the Linux computer. Each wheelchair is equipped with a TinMan supplementary wheelchair controller from the KISS Institute of Practical Robotics (KIPR). The TinMan controller uses a 32-bit Motorola MC68332 processor for motor control and sensor monitoring. This controller is suitable for simple navigation tasks such as obstacle avoidance. More complex tasks such as visual navigation, however, require more computational power and access to a larger suite of sensors. This environment is provided by the dual-Pentium Linux box strapped to each wheelchair. The serial protocol is an interface that gives complex computational tasks running on Linux access to the low level motor functions and sensor readings brokered by the TinMan controller.

The serial protocol is meant to support various forms of control functionality. ARC allows multiplexing of several abstract serial "streams" on one serial line, and we envisioned streams with the following characteristics. One should flow "down" from Linux to Microcontroller carrying commands and inquiries. Commands should be of two varieties. First, for feedback control and fast response, we need a *synchronous* stream providing basically remote procedure calls. The caller (Linux process) is suspended until the callee responds. Second, since it is possible to put rather serious functionality into the ARC microcomputer, we envisioned commands of the "Go do this complicated job while I think about something else. Tell me how it came out" variety. Possibly the programs running on ARC would have something interesting to impart, or to ask of the Linux side: an "upcall". Thus we need an *asynchronous* stream for arbitrary communication from autonomous ARC processes. An *error* stream is like the asynchronous stream but gets higher priority and, and *sensor readings* are messages that presumably always have the same format and should be sent at some constant rate.

These abstractions were in our minds as we developed the serial protocal that is described in more detail below.

2.2 ARC

The TinMan controller came with a copy of the ARC C development environment for programming the MC68332 processor. ARC was created by Newton Research Labs for programming 32-bit embedded systems. It consists of a suite of compilation tools, including the C compiler `arcc` and a serial interaction program, `arc`, for downloading programs to the MC68332.

ARC can be installed on Linux, SunOS, Solaris or Irix workstations. Currently, ARC is installed on the Linux box on each wheelchair. Through the remainder of the ARC discussion, the Linux box will be referred to as the *host*. The MC68332 processor will be referred to as the *target*.

The serial protocol is implemented on the target using ARC. ARC supports multi-tasking and multi-streamed serial I/O, making it a powerful and flexible environment for implementing a subsumption architecture. `arcc` uses the GNU C compilation backend, so it is not only a powerful but also a familiar environment for the members of our research team.

More information about ARC, including the User's manual, is available online:

`http://www.newtonlabs.com/arc/`

Parallel programming

ARC provides end users with a fine degree of control over the time given to each process. The environment supports up to 32 simultaneous processes. Processes are scheduled "round robin" — the multitasker iterates through the process table and context switches when a process's allocated time expires.

Processes are created using the `start_process()` function with the syntax

```
pid start_process(char *process_name, void (*)function,
                  int stack_size, int time_slice)
```

where `time_slice` is measured in scheduler ticks. The granularity of each tick is determined by the programmable scheduling interrupt period. The interrupt period is currently set to 976,000 nanoseconds, which yields 1024 scheduler ticks per second.

The serial protocol code relies heavily on two functions to control process scheduling. The `defer()` function causes the current process to give up the rest of its allocated scheduler ticks. The `hog_processor()` function gives the current process 256 more scheduler ticks. The `hog_processor()` function should be used with care. If a process uses all 256 allocated scheduler ticks, the frequency of each cycle through the process table is reduced to 4 Hz. This frequency may be too low for real time control of the wheelchairs.

ARC also provides semaphores, or locks, to guarantee mutual exclusion for critical sections of code. The `lock()` function has the syntax `void lock(lock_t *flag)`. This function will defer, or spin, until the lock is released by another process using the `unlock()` function with the syntax `void unlock(lock_t *flag)`.

Multi-streamed Serial I/O

Our method for handling multi-streamed I/O is based on a design and sample code supplied by Anne Wright of Newton Research Labs. The protocol can be parsed by a two-state finite state machine, shown in Figure 3). The protocol uses [0xfe][1] as the stream break character.

All characters received in State 0 are passed on to the current input stream. A stream break, [Oxfe], results in a transition to State 1.

[1] Hexadecimal numbers in brackets represent bytes with non printable values

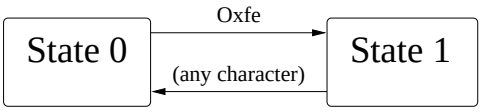

Figure 3: Multi-streamed Serial Protocol FSM

process	*description*
`dispatch`	Serial I/O, process commands from host
`driver`	Motor control, obstacle avoidance
`read_sensors`	Update infrared & bump sensor readings
`encoders`	Update shaft encoder readings
`sonars`	Update sonar readings
`send_sensor_packets`	Send sensor packets to host

Table 1: Description of process functionality

In State 1, the last received character was a stream break, but not the character before that. This state transitions back to State 0 on any input, and the character just received is interpreted as follows:

- The number of the new input stream if it is in range
- A literal with the same value as the stream break
- An error (out of range, not a stream break)

Data can be sent to a particular stream by prefacing it with a stream break followed by the stream number. For example, to send "foo" to stream 5, you would send:

```
[0xfe][0x05]foo
```

2.3 Architecture: ARC

The foundation for the target-side serial protocol is a collection of functions that KIPR included with the TinMan robots. These functions provide a clean interface to the wheelchair sensors, as well as some rudimentary navigational functions. The TinMan source code is distributed across the files `dio.c, diotpu.c` (digital I/O and Timer Processor Unit functions), `encoders.c, sonar.c`, all of which are `#include`'d from `tm.c` The TinMan code can be accessed by placing a `#include "tm.c"` statement in a source file, or by linking `tm.o` into a program.

When running, the protocol uses six separate processes. The functionality of each process is summarized in Table 1, and the location of the source code for the corresponding function is listed in Table 2.

process	*location*	
`dispatch`	`dispatch.c:`	`main()`
`driver`	`dispatch.c:`	`driver()`
`read_sensors`	`tm.c:`	`TM_read_sensors()`
`encoders`	`encoders.c:`	`encoders_main()`
`sonars`	`sonar.c:`	`sonars_main()`
`send_sensor_packets`	`dispatch.c:`	`send_sensor_packets()`

Table 2: Location of source for each process

Processing I/O

The serial protocol currently multiplexes five streams on the serial line. One stream (input) is reserved for data sent from host to target, the remaining four streams (sensor, error, synch and asynch) are reserved for sending data from target to host.

The input stream relays turning, driving and state control commands. Sensor packets are sent at fixed intervals across the sensor stream, and debugging information is sent across the error stream. The synch and asynch streams are used to send responses from synchronous and asynchronous commands, respectively. Figure 4 illustrates which processes access each of the streams.

The `dispatch` process is responsible for receiving and processing commands from the host. A valid command consists of a one byte identifier, possibly followed by additional arguments. See section 2.7 for details about creating new commands. The file `messages.h` is used by the serial protocol implementation on both the host and the target. The pseudocode in Figure 5 illustrates how `dispatch` processes commands. A command read from the serial line is used to update a global variable. The global variable is monitored and acted upon by another process, such as `driver`.

While it would be possible for `dispatch` to execute all of the commands itself, delegating this task to other processes gives the programmer more control over the amount of time given to each component of the serial protocol. This degree of control is important if the protocol is to be used for real time control of the wheelchair.

Sensor readings

The wheelchair sensor readings are continuously updated by three separate ARC processes - `encoders`, `sonars` and `read_sensors`. The latter process updates the infrared sensors, bump sensors, and the position of the joystick. Each of these processes writes the updated sensor readings to global variables that can be read by other processes.

The updated readings are sent as sensor packets across the sensor stream. These readings can be accessed by C programs running on Linux by using the sensor library, described in section 2.5. The sensor packet format is shown in Table 3.

Sensor packets are sent at a fixed frequency, which can be modified by changing the `#define`'d constant

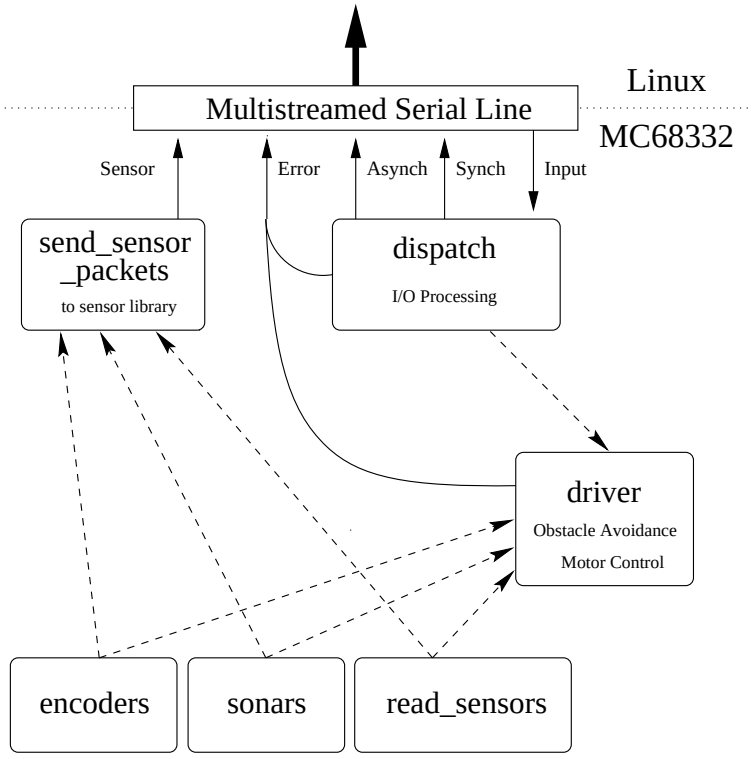

Figure 4: ARC serial protocol process view. Dotted lines indicate that information is passed between processes using global variables

```
/* ----- Global variables ----- */
int drive_speed, turn_speed, avoidance;
lock_t serial_line;

read_cmd:

  command = read(command stream, character);

  lock(&serial_line);

  switch (command) {
    case DRIVE_TACIT: case DRIVE_ACK:
      drive_speed = read(command stream, character);
      if (command == DRIVE_ACK)
        write(synch stream, ATTN);  /* --- Synchronous response */
      goto end_command_interp;

    case TURN_TACIT: case TURN_ACK:
      turn_speed = read(command stream, character);
      ...

    case AVOID_ON:
      write(error stream, debugging message);
      avoidance = 0;
      goto end_command_interp;
  }

end_command_interp:
  unlock(&serial_line);
  goto read_cmd;
```

Figure 5: Pseudocode for the ARC process, `dispatch`

Sensor Packet Format

bits	0-31	32-63 (int)	64-95 (int)	96-127 (int)	128-159 (int)
data	sensorbits	sonar	left_encoder	right_encoder	thetime

Composition of *sensorbits*

bits	0-15	16-27	28	29	30-31
data	*Unused*	Infrared	Left bump	Right bump	*Unused*

Table 3: Sensor Packet Format

SENSOR_PACKET_FREQUENCY in the file `dispatch.c` Pseudocode for the flow control implemented in `send_sensor_packets` is shown in Figure 6.

Starting the Serial Protocol

What follows are some brief instructions and examples illustrating how to use ARC to compile and run the serial protocol. It is not intended as a comprehensive guide to the serial interaction program, `arc`, or the ARC C compiler, `arcc`. Please refer to Chapters 3 and 4 of the ARC User's manual for detailed information on these two programs.

The serial interaction program is started by typing `arc` at the host's command line. The `named_socket_server` program will be started if it is not already running and no other program is offering equivalent services. A copyright notice and license agreement will be displayed, followed by output similar to this:

```
Port /dev/cua0, baud 38400
    Terminal port # 8788
Term socket = <fd 4>
arc>
```

The `ramload` command can be used from the `arc` prompt to download programs to the target. This command can accept either a source file or a precompiled binary as an argument. If `ramload` is invoked on a source file, it will compile this file and then download the resultant binary to the target processor. The file `dispatch.c` merges in all of the source for the serial protocol, thanks to liberal usage of the `#include` pragma to load C source files.

The following session output illustrates this process:

```
arc> ramload dispatch.c
Running arcc dispatch.c
arcc version 1.4
Input file: dispatch.c, output file: dispatch (kernel=
 ttaleboot.976, addr 0x1c8000)
set_checksum version 1.4
```

```
/* --- Compute sensor_packet_delay from the constant
   --- SENSOR_PACKET_FREQUENCY and the duration of
   --- each scheduler tick.  */

while (1) {

wait_for_delay_period:

  dt = time() - thetime;
  if (dt < sensor_packet_delay) {
    defer();
    goto wait_for_delay_period;
  }

  lock(&serial_line);

  thetime = time();
  compute sensorbits;

  write(sensor, ATTN);
  write(sensor, sensorbits);
  write(sensor, sonar);
  write(sensor, left_encoder_ticks);
  write{sensor, right_encoder_ticks);
  write(sensor, thetime);

  unlock(&serial_line);
}
```

Figure 6: Pseudocode for the ARC process, `send_sensor_packets`

```
Adding dependencies: dispatch 0, ttaleboot.976 54d3862a
dispatch checksum changed from 0 to b40449d0
Done
```

At this point in the process, the binary file `dispatch` has just been compiled from the `dispatch.c` source file. It is also possible to compile `dispatch` from a UNIX command prompt, using the syntax `arcc dispatch.c` The output below shows the binary file being downloaded to the target. If the binary file already exists and you want to avoid recompilation time, simply type `ramload dispatch` at the `arc` prompt. The rest of the output from the `ramload` command is listed below.

```
in download_aout_rest
Download dispatch (a.out file, gdb protocol)
Reading symbol data from dispatch ...    (36 symbols not
 classified)
Read 25118 symbols (of 25118 symbols total)
Downloading code (0x1c8000-0x1cac8f).....................
Downloading global initializers (0x1caf3c-0x1cafbb).
arc>
```

Once a new program has been downloaded to the target, all user processes stop. To start the serial protocol, type `run dispatch` at the `arc` prompt. Alternatively, you can push the red RESET button. The `main()` function in `dispatch.c` will start all of the target side serial protocol processes. After exiting the serial interaction program, the host and target can start communication. [2]

2.4 Architecture: Linux

The Linux-side serial protocol is started by running three programs - `serial`, `command` and `dispatch`. The source files used to create these programs are shown in Table 4. These programs can be thought of as daemons. They are intended to run continually in the background and interface with programs that use the motor or sensor library. `serial` is responsible for I/O across the serial line. Once this program has initialized, it will write all output from the `command` program to the serial line, and will send all output from the serial line to the `dispatch` program. `serial` does not examine any of the data it is shuffling back and forth. The `command` program is responsible for connecting to programs that utilize the motor library, and relaying the commands that they send to `serial`. `dispatch` reads the data that `serial` sends it, checks the stream number, and routes the data to the appropriate stream handler process.

[2] Newton Labs uses some code internally that permits the `arc` serial interaction program and user programs to share the serial line. According to Anne Wright, this code has a "messy interface," and using both programs simultaneously increases latency.

source file(s)	*purpose*
`arc_io.c,h`	Multi-streamed I/O
`command.c`	Communicates with motor library programs, `serial` and `dispatch`
`dispatch.c`	Routes streamed I/O to appropriate handler
`handlers.c,h`	Process output from `dispatch`, write to shared memory if necessary
`messages.h`	Definitions for serial protocol commands
`motorlib.c,h`	Motor library implementation
`sensorlib.c,h`	Sensor library implementation
`serial.c`	Reads, writes to serial line

Table 4: Linux source files

Initialization

1. Start `serial` from a prompt. It will open the serial line for communication, and then listens for a connection from `command` on the SERIALLISTENSOCK socket. (SERIALLISTENSOCK and the other named sockets are `#define`'d in `messages.h`

2. Start `command` from a prompt. It will connect to `serial`, and then listen for a connection from the synchronous stream handler on SYNCHSOCK. (stream handlers are explained in greater detail below)

3. After `serial` has connected to `command`, it will listen for a connection from `dispatch` on SERIALTALKSOCK.

4. Start `dispatch` from a prompt. It will connect to `serial`, and then `fork()` off the four stream handlers. Data sent across the serial line by ARC, such as sensor packets, is now being processed.

These steps are only executed the first time that the serial protocol is initialized.

Motor Library Connection Attempts

The following steps are executed each time a motor library program attempts to connect to the serial program. In the explanation below, references are made to the numbered steps illustrated in Figure 7.

1. The `main()` function in `command` waits for a motor library program to attempt to connect to MOTORSOCK (1).

2. When a connection is made (2), `command:main()` checks to see if the semaphore controlling access to the motors is set. If the semaphore is set, the connection attempt is rejected. If the semaphore was free, `command:main()` sets it and `fork()`'s. The

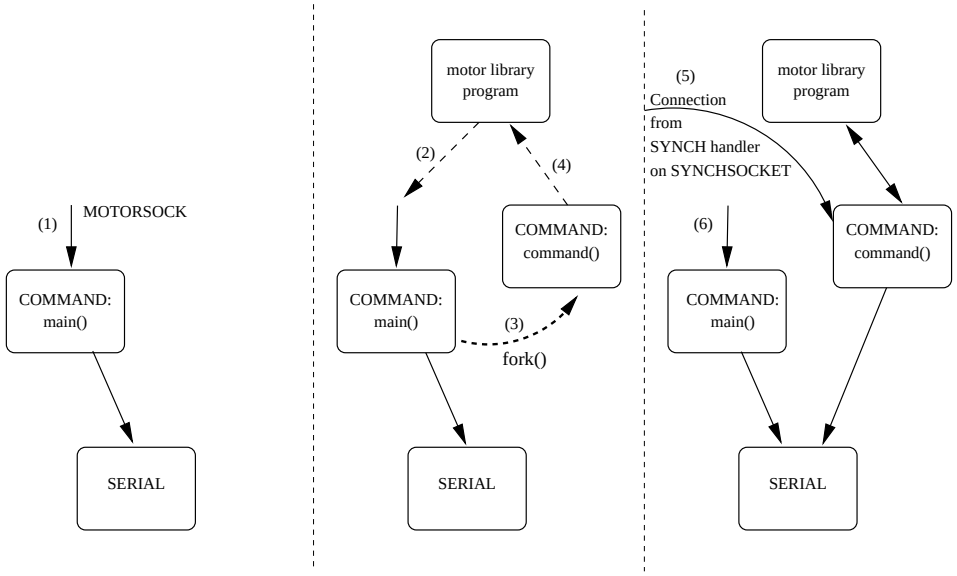

Figure 7: Command connection process

child process will execute the `command()` function (3), and will be referred to as `command:command()`. It is responsible for communicating with the motor library program (4). The parent process, `command:main()`, continues to listen for connection attempts on MOTORSOCK (6).

3. `command:command()` connects to the synchronous stream handler on the SYNCH-SOCK socket.

4. `command:command()` tests its connection to the synchronous stream handler by writing the synchronous command INIT_CMD (`#define`'d in `messages.h`) to `serial`. `command:command()` then listens for an INIT_CMD on SYNCHSOCK.

5. `serial` sends the INIT_CMD across the serial line to the ARC-side serial protocol. The ARC-side serial protocol writes an INIT_CMD to the synchronous stream of the serial line. `serial` receives this character, and sends it to `dispatch`, which in turn sends it to the synchronous stream handler. This stream handler writes the INIT_CMD to SYNCHSOCK, where it is read by `command`. This pathway between processes is used by all synchronous commands.

6. The motor library program is now successfully connected to the serial protocol. This state is illustrated in Figure 8.

7. Commands are relayed until the motor library program disconnects from the serial protocol.

The Serial Protocol Architecture: Process View

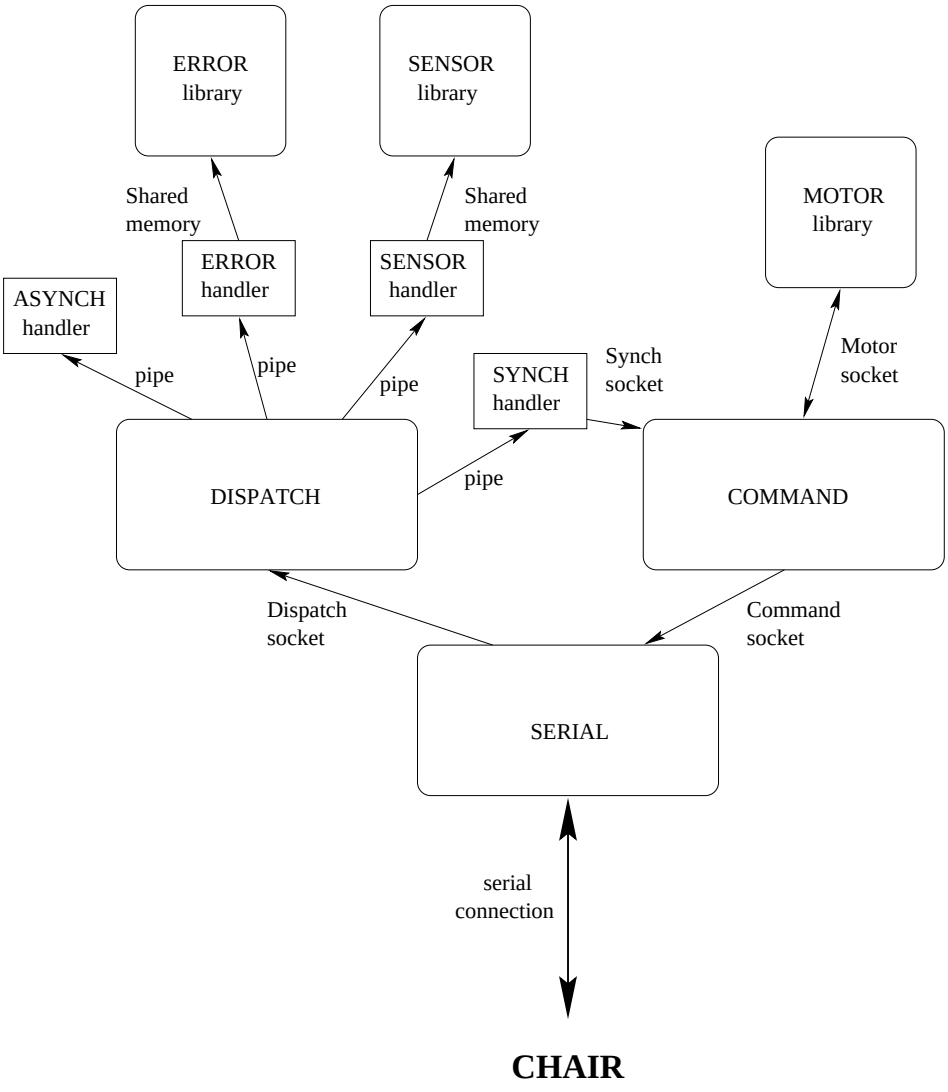

Figure 8: Serial protocol innards: Linux side

2.5 The C Programming Interface

The C programming interface to the serial protocol is implemented as a pair of libraries: one to control the chair's motors and one to read its sensors. Each of these libraries, in turn, communicates with the collection of daemons that run behind the scenes; the daemons do the dirty work of controlling the serial line, maintaining the shared memory segment for sensor readings, and so on. The purpose of the libraries is to hide implementation details from the programmer and allow a simple, controllable C interface to the wheelchairs.

To use the serial protocol library (which includes both the motor and sensor libraries), link with `libchair.a`. Include files `motor.h` for the motor library and `sensor.h` for the sensor library. (Include and library files reside in the /u/mobile directory).

All library functions work in basically the same way: first, the library is initiated via a call to the relevant attach() function; then, functions in that library are called by the user; and finally, the library is closed via the relevant detach() command. Only one user process may connect to the motor library at a given time. As many users as desired may attach to the sensor library.

The motor library

`int motor_attach()`: Initiates the motor-handling part of the library, and prepares the wheelchair motors to accept drive and turn commands. On success returns 0; on failure, -1.

`int TM_drive(int speed)`: Sets the motors to a speed in the range $[10, -10]$. If speed is out of that range, `TM_drive()` truncates it automatically. `TM_drive()` returns the speed to which the motors are finally set, or ERROR on error.

`int TM_drive_asynch(int speed)`: An asynchronous version of the above command. Speed is still in the range $[10, -10]$. `TM_drive_asynch`, however, returns no value.

`int TM_turn(int speed)`: Sets the turn speed to a value in the range $[10, -10]$. If speed is out of that range, `TM_turn()` truncates it automatically. `TM_turn` returns the value to which the turn speed is finally set, or ERROR on error.

`int TM_turn_asynch(int speed)`: An asynchronous version of the above command. Speed is still in the range $[10, -10]$. `TM_turn_asynch`, however, returns no value.

`int TM_avoid(short f)`: Controls obstacle avoidance on the chair. When obstacle avoidance is active, the chair filters all motor commands through Amit Singhal's routine designed to keep the chair from colliding with objects. [FOLLOWING PARAGRAPH SHOULD PROBABLY BE REPLACED WITH A REFERENCE TO AMIT'S SECTION OF THE PAPER] If an obstacle is in front of the chair, a `TM_drive` command will be translated into `TM_turn` commands until the obstacle is cleared.

The obstacle avoidance routine can not be counted on to keep the chair from colliding with all of the objects in the software lab. Because it relies on the IR sensors, which are

placed low on the chair chassis, it can not detect the presence of desks, chairs, and other high obstacles. The IR is also fooled by dark metal surfaces like the supplies cabinet in the mail room.

Passing a non-zero value to `TM_avoid` turns on obstacle avoidance; a zero turns it off.

`int TM_control(short f)`: Controls feedback-driven course correction for drive commands. When course correction is active, the chair filters all motor commands through Chris Eveland's routine designed to keep it as close as possible to the last course set. This routine checks the value of the encoder on each wheel of the wheelchair to determine whether the chair is moving in the correct direction; if not, it modifies the motor commands to better reflect the requested command.

Passing a non-zero value to `TM_control` turns on course correction; a zero turns it off.

`int TM_cycle_time(void)`: Returns the value of the ARC function `swap_cycle_time()`. This value, according to the ARC manual, is "the average number of scheduler ticks that it is taking to completely cycle through the process table. This number will range from 0 (if everyone is deferring) to the sum of the number of ticks allotted to all the processes (if no one is deferring). This number allows you to have an idea of the processor load."

`int motor_detach()`: Disconnect from motor library, deactivate motors and reset communications. Returns 0 on success, -1 on failure.

The sensor library

`int sensor_attach()`: Initiates the sensor-handling part of the library. This function must be called before any of the sensor-accessing functions are called. Once sensors are initialized, they will be updated asynchronously several times per second. The function `sensor_attach()` returns 0 on success; -1 on failure.

`int *TM_ir()`: Reads the IR sensors. Returns a 12-element array of ints, one for each IR sensor, in the order in which their inputs are numbered. So the IR sensor in input jack 1 maps to element 0 of the return array, IR 2 maps to element 1, and so on. The call should follow the following form:

```
int *irs = TM_ir();
```

Memory for the return array is allocated by the `TM_ir` function, but must be freed by the calling function.

`int *TM_bump()`: Reads the bump sensors. Returns a 2-element array of ints, one for each bump sensor. The left bump sensor maps to array element 0; the right to array element 1. The call should follow the following form:

```
int *bump = TM_bump();
```

Memory for the return array is allocated by the `TM_bump` function, but must be freed by the calling function.

`int *TM_encoder();`: Reads the shaft encoders. Returns a 2-element array of ints, one for each wheel. The left encoder maps to array element 0; the right to array element 1. The call should follow the following form:

```
int *enc = TM_encoder();
```

Memory for the return array is allocated by the `TM_encoder` function, but must be freed by the calling function.

`int TM_timestamp()`: Returns the value of the ARC function `time()`, which is the number of scheduler ticks since the microcontroller was last power cycled.

`int sensor_detach()`: Disconnect from the sensor library. Returns 0 on success, -1 on failure.

2.6 Timing and Bandwidth Issues

In an earlier version of the serial protocol, there were problems with real time control of the chair from the Linux box. The problem arose when more control signals were generated than the serial line and the message decoding on the 332 could handle. This mismatch caused a buffer to fill up, introducing latency into the command path.

This was a serious problem when several seconds worth of commands had been stored in the buffer. At this point, if a stop command was given, all of the drive commands issued before it that were still in the buffer would be executed before this stop request was processed. Usually this had catastrophic results.

Asynchronous Command Flow Control

To address this, software flow control for asynchronous turn and drive commands was added to the file `command.c` The same method is used for both types of commands, however only the asynchronous turn command case is described here. Pseudocode for the flow control is provided in Figure 9.

The frequency at which asynchronous motor commands can be sent is specified in Hertz by the constant TACIT_COMMAND_FREQUENCY `#define`'d in `command.c` A delay period between motor commands is calculated from this constant. When an asynchronous motor command is received, `command` modifies a global turn variable. Then, it checks to see if an amount of time greater than the delay period has passed since the last command. If the delay period has been exceeded, the turn command will be sent to `serial`. If not, `command` will continue to read commands until the delay period has been exceeded, and then only the most recent turn command will be issued. An asynchronous turn command will never be issued unless the turning speed has changed since the last command that was issued.

```
while (1) {

  /* --- Non-blocking I/O --- */
  command = read(MOTORSOCK, character);

  if (command) {

    /* --- Process commands --- */
    switch(command) {

    case TURN_TACIT:
      speed = read(MOTORSOCK, character);
      pending_turn = TRUE;
      pending_turn_speed = speed;
      break;

    case DRIVE_TACIT:
      ...
    }
  }

  current_time = time();

  if (pending_turn &&
      (current_time > next_packet_time) &&
      (pending_turn_speed != current_turn_speed)) {
    pending_turn = FALSE;
    next_packet_time = current_time + delay;
    current_turn_speed = pending_turn_speed;

    /* --- Execute motor command --- */
  }
}
```

Figure 9: Pseudocode for `command()` function defined in `command.c`

Monitoring serial line bandwidth

The `serial` program can monitor the serial line's data rate. Simply edit the source file `serial.c`, and change the line `#undef TIMING_TESTS` to `#define TIMING_TESTS`. Now, `serial` will print out the data rate each second in bits per second.

By modifying TACIT_COMMAND_FREQUENCY, as well as the constant SENSOR_PACKET_FREQUENCY in the ARC file `dispatch.c`, it is possible to monitor and tweak the rate at which data is sent across the serial line.

The motor library function `TM_cycle_time()` is also a useful tool. Dividing the number of timer ticks per second (currently 1024) by the average cycle time yields the average frequency at which the serial line is checked for commands and the sensors are read. This frequency is, theoretically, the maximum rate ate which sensor packets could be sent and commands processed. The practical maximum is constrained by various latencies within the serial protocol as well as the serial line's bandwidth.

2.7 Extending the serial protocol

The serial protocol is designed to be flexible and extendable. Synchronous and asynchronous commands can be added without making any changes to the architecture of the protocol or changing the syntax of currently implemented commands.

Commands sent from Linux to ARC consist of a one byte command ID followed by a variable number of arguments. The end user executes commands by making a call to a motor library function. This function sends the ID and arguments to the `command` process, which in turn sends them to the `serial` process, and then down the serial line to the ARC process `dispatch`. If the command is synchronous, ARC will write the response back to the serial line using the synchronous stream. This response will be read by the `serial` process, and will then be passed on to `dispatch`, routed to `synch_handler`, and the synchronous return value will be read by `command`. `command` will then return this value to the motor library program. The synchronous command pathway can be seen in Figure 8.

Valid command ID's are specified in the `messages.h` file. This file is used by both the ARC and Linux sides of the serial protocol. New commands are created by adding entries of the form

```
#define COMMAND_NAME [bytevalue]
```

Bytevalue must fall between the constants CMD_LOW and CMD_HI, which are also `#define`'d in `messages.h`

New commands can be implemented by executing the following steps:

1. Add a motor library function to the files `motorlib.c` and `motorlib.h` This function should write the command ID and any arguments to MOTORSOCK.

2. Add a case statement to the function `command()` in the `command.c` file to handle the new command. (See Figure 9) The code in the case statement should read all of the

command arguments, and then write ATTN, the ID and the arguments to SERIALLISTENSOCK. All commands sent to ARC must be preceded by an ATTN character. The `serial` program will read these characters from SERIALLISTENSOCK, and then write to them to the serial line. The ARC process `dispatch` will read the ID and arguments from the serial line. If necessary, `command` should wait for the synchronous return value to be written to SYNCHSOCK.

3. Add a case statement to the `main()` function of the ARC file `dispatch.c` to handle the new command. If necessary, write a return value to the synchronous stream. (See Figure 5) The `dispatch` process may either execute the command itself, or modify a global variable which will cause another process to execute the command. The latter method gives the programmer more control over the time granted to each process.

The serial protocol is undergoing constant scrutiny and we already see possibilities for improvement. The current "error" stream is used solely for sending debugging information, and should probably be renamed "debug." A real error stream, which returns some form of error values, could be implemented. Error messages could be timestamped, or they could be stamped with some unique command ID. Implementing command ID's (for asynchronous commands) could be tricky, especially because some of the commands will be ignored by the flow control routines. Sending command ID's down the serial line could also reduce the frequency of commands.

The interaction of explicit commands from the user at the Linux level with lower-level code (such as obstacle avoidance or other "reflexive" operations) in the microcontroller is interesting. Motor library programs currently have no way of knowing whether or not the obstacle avoidance routines have overridden a user turn or drive command - this information would be *very* useful to have. Should user programs have to check for this information, or would an interrupt handler work better? At a higher level, how would a program decide to "give up" and try another path when the obstacle avoidance routines prevent it from going in a certain direction?

3 Obstacle Avoidance

The goal of the obstacle avoidance research effort was to implement a reactive obstacle avoidance strategy for allowing the wheelchair to perform collision free navigation in tight and cramped spaces. We have developed an ad-hoc rule-based reasoning strategy that uses the various sensors (bump sensors, infra-red proximity sensors and sonar sensor) on the wheelchair to get information about obstacles in the neighborhood of the robot. The obstacle avoidance algorithm then takes the motor control commands received from the navigation system (human using joystick or autonomous driving program) and runs them through the rule database, adjusting their values depending on the locations of the obstacles before sending them on to the onboard motor controller. The current obstacle avoidance algorithm has the following rule base:

```
if obstacle detected in front of robot then
  if back is clear and both corners are not
     then reverse at low speed
  else if one corner is clear
     then move forward at low speed while turning moderately
          towards clear corner
  else if one side is clear
     then move forward at low speed while turning sharply
          towards clear side
  else
     turn sharply towards one side
else
  if sonar reading < 0.7 m
     then slow to 20% of maximum speed
  else if sonar reading < 1.2 m
     then slow to 50% of maximum speed
  if one corner is not clear
     then turn sharply away from that corner
  else if one side is not clear
     then turn slightly away from that side
  else if drive command is to turn towards a side
     then if corner is clear
        turn slightly away from that side
     else turn sharply away from that side
  else if drive command is to go in reverse
     then if back is not clear
        then if side is clear
           turn in place to clear side
     else turn in place to any side
  else if driving command higher than maximum speed
     limit driving command to maximum speed
```

The above obstacle avoidance algorithm was implemented using ARC, a software development system for the wheelchair robot, and integrated into a remote operation demonstration system with the help of Chris Eveland. Multiple tests were performed on this system for navigation through cramped doorways, computer lab environments, among moving people and in hallways. The tests showed that the system is extremely robust when the size of the obstacles is large enough to be detected fairly easily by the infrared proximity detectors. Our obstacle avoidance algorithm performs very well on very hard tasks such as going through a small doorway but fails to follow walls well. Part of this is due to a control problem that is causing the robot to drift to the right.

Another aspect of this research work involved obtaining a better configuration of the Infra Red proximity sensors on the robot. The original configuration was found to be lacking and left major gaps and holes where obstacles were not being detected. We modified the placement of these sensors based on experimental results and obtained a better coverage of the wheelchair's environment. Our next step is to model the sensors in the simulator and find their optimal spatial configuration for obstacle avoidance.

Timing tests were run with the help of Craig Harmon on the wheelchairs when it was performing obstacle avoidance. It was determined that the obstacle avoidance program was generating drive commands to the motor controllers at a rate of 60-80 Hz per second. This rate is very good and reflects our extremely simplistic approach to the problem. Some of the limitations of the current system include the inability to detect thin vertical objects such as chair and table legs, and dark color objects due to sensor limitations. Also, the approach is completely ad-hoc and all refinement until now has been performed on the basis of empirical results.

We propose to continue research in obstacle avoidance and formalize the above system into a fuzzy rule based logic controller. This formalization would help us better understand the operation of our system, while making it modular and easily maintainable. We plan to use this obstacle avoidance algorithm in conjunction with a path planner to perform motion planning for the wheelchair.

4 A Software Infrastructure for Mobile Robotics

The goal of this software architecture is to allow quick production of visual systems for robot control. There are several important considerations in the design of such a system. One is to provide an interface for communication between low level vision routines, and high level reasoning. Of course efficient use of computational resources is also important, as well as issues of reliability and portability.

This architecture addresses all of these issues. In the following sections, the architecture will be explained in the context of these issues. At the end a user's manual is provided, as well as the discussion of the design of a sample application using this architecture.

4.1 Architecture

There are two main components to the architecture. First a means of image based communication based communication between vision routines is provided using shared memory. The library takes care of all of the synchronization issues, so that the programmer may concern themselves with issues more closely related to their application. This communication is provided using "video buffers".

The second component is a lower bandwidth but more loosely coupled means of communication. This allows processes running on different machines an easy way of communicating with each-other, again without burdening the programmer with the details of TCP/IP sockets. This communication is provided by the "scene description bus".

VideoBuffers

VideoBuffers are the application of the readers/writers problem to frames in an image sequence. There is assumed to be one source which is generating images, and several readers that may read from this stream.

These images are simply stored in a contiguous block of memory, along with a certain amount of book keeping information. In addition, a writer function is registered to go along with it. When the buffer is created, a thread that writes onto it is also created. The programmer simply needs to specify a function that will do the writing.

Once the buffer has been created, readers may read from it, using a pair of functions that open and close a frame for reading. Because there is a temporal ordering on the images, there are two modes in which a reader may want to read images from the buffer. It may want the current image (or some fixed number of images in the past), or it may want to block and wait for the next image which it has not read yet. In order to handle both of these situations, sequence numbers are kept. When a read request is given, the sequence number must be given with it. If the number is less than 1, then it is taken to be an offset from the current image. The call to the start read function will take care of making sure no one is writing the requested frame, and if the frame has not yet been generated, then suspending the thread until it is.

`openBuffer`	Opens a new buffer with the default values.
`attachWriter`	Attaches a writer thread to a buffer.
`closeBuffer`	Closes a buffer.
`start_write`	Get the writer lock for the next frame n the sequence.
`finish_write`	Release the writer lock.
`start_read`	Get the reader lock for a frame.
`finish_read`	Release the reader lock.

Table 5: VideoBuffer functions

`connectSDBus`	Connect to a server.
`addSDBusFilter`	Request messages matching a filter.
`writeSDBus`	Broadcast a message.

Table 6: Scene Description Bus Functions

The Scene Description Bus

The scene description bus provides a broadcast medium (hence bus) over which information about a scene may be shared. Clients attached to the bus are called filters. There is a server that runs on one machine and distributes the information. The server passes information to be broadcast to a specific client only if the message matches a pattern, or filter. In this way the message only needs to go to the group of clients that are interested in that particular type of message.

Putting it all Together

In building an application, several modules will be made for specific visual tasks. These can then be used a stock building blocks and glued together via video buffers. From these building blocks, a higher level representation of the scene will be built, which therefore has a more concise description and is then suitable for broadcast over the scene description bus. This processing will be done on fairly tightly couple processors. Then, the higher level reasoning routines can listen to their output, and possibly put back information onto the bus, as well as do whatever robot control is needed. This can all be done more remotely and in a more distributed manner, because the amount of information that needs to be moved around is less.

4.2 User's Manual

There are relatively few function in the VideoBuffer and SDBus APIs. Their names and purposes are outlined in tables 5 and 6. In the next two section they are described in more detail.

VideoBuffer API

The use of a video buffer is similar to the manner in which one would use a file. First it is opened, then it is read or written to, and finally it is closed. The main difference is that there are two function for both reading and writing: one to start, and one to finish. In between one has complete access to the data so that it may be manipulated in the most efficient manner possible.

These function are defined in the following sections.

openBuffer

Synopsis `VideoBuffer *openBuffer()`

Discussion This function returns a buffer with all of the default settings. See the file `vbuf.h` for a definition of the structure.

As one writes modules that use this API, it is natural to extend these functions. For instance, a frame differencing module may want to define a function openDiffBuffer which would be a specialized version of openBuffer. In fact that is exactly what is done in the sample application (see below).

attachWriter

Synopsis

```
void
attachWriter(buf, func, args)
VideoBuffer *buf;
void *(*)(void *)func;
void *args;
```

Parameters

`buf` The buffer that the writer thread will belong to.

`func` The function that will run as the thread. It should not exit under normal conditions, continuously writing new data onto the buffer as it becomes availible.

`args` The argument to the function. Usually this will be an array of VideoBuffer pointers, however anything that is desired may be passed. Part of this data must be the buffer onto which to write, otherwise there will be no way for the thread to access its own buffer.

Discussion This function attaches a writer to a buffer. It is nothing more than a wrapper function for the thread creation function, but this wrapper allows the possibility of changing thread packages, or other implementation details, without having to effect the API.

closeBuffer

Synopsis `void closeBuffer(VideoBuffer *buf)`

Parameters

`buf` The buffer to be closed.

Discussion Simply close the buffer. It may not be used after it is closed, or results will be undefined.

start_write

Synopsis `int start_write(VideoBuffer *buf)`

Parameters

`buf` The buffer to be written to.

Discussion Start writing on the next frame. If all frames in the buffer are currently being read, then this will block until a space becomes free for the new frame. The contents of the frame before the write are undefined. The return value is the frame number on which the write should take place. This is not to be confused with the sequence number. The sequence number is a unique number to every frame (at least until all of the 32 bit integers are used up), while the number that it returned is just an index into the image array. The only argument is a pointer to the video buffer to which the write is to take place.

finish_write

Synopsis `void finish_write(VideoBuffer *buf, int frame)`

Parameters

`buf` The buffer that was written to.

`frame` The frame number that was written.

Discussion The first argument is a video buffer to which the write has been done, and the integer is the frame index on which it was done. After this is called, the writing locks will be released, and the process is no longer allowed to write to the frame.

start_read

Synopsis `int start_read(VideoBuffer *buf, int frame)`

Parameters

`buf` The buffer that will be read from.

`frame` The frame number that will be read from. It is either an offset from the current frame, in which case the offset must be negative, or a sequence number, in which case it will be positive.

Discussion The two arguments describe the buffer and sequence number from which to read. The sequence number can be wither absolute or relative. If it is positive, it is taken to be absolute, in which case permission will be granted to read from that sequence number, or if that frame has been overwritten, the frame closest to it. In the case that the number is negative or zero, the frame with that offset from the current frame will be given. In the second case, the function may block until the requested sequence number is created. The return value is the index into the image array from which to read that frame.

finish_read

Synopsis `void finish_read(VideoBuffer *buf, int frame)`

Parameters

`buf` The buffer that was read from.

`frame` The frame number that was read.

Description The two arguments are the buffer and image index that a read has been complete on. The locks associated with it will be released.

SDBus Client API

The SDBus's API has only three functions. The basic sequence of actions for a SDBus client is to first make a connection to the SDBus using `connectSDBus`, then to (optionally) add filters using `addSDBusFilter`. After that setup is done, data may be written to the bus using `writeSDBus`. What exactly it means to connect to the server and write to it should be fairly clear without further explanation. Adding filters, however, may not be so clear. When data is written to the SDBus, not all of the clients that are listening to it may need that data, so having the server send the data to all of the clients would not be very efficient. As a result, each client may define filters for different types of data that it wants

to receive. There are basically regular expressions which the server can match against to decide who to send the data to. Therefore, when a client wants to get a certain type of data, after connecting to the server, which allows it to write to the bus, the client must then send a message requesting a specific filter to be added. This message consists of three parts: the regular expression, a function address, and an ID. The regular expression is what the server matches against to decide if the client should get that message. It should be in the form of a PERL regular expression, without the "/"s on either end. The function address is a function that will be called when this message is received. To implement this, there is actually a thread created when the bus is connected to. For the most part this thread sits blocked waiting to be able to read from the server. When the server sends a message to the client it also sends a function address and id along with the message, so that this thread may call the function with the correct arguments. To the programmer, this may remain transparent, and the effect is that the function that is added with the filter will be called each time that a message matching the filter is received. Its arguments are the text of the message, plus an id number. This ID is the last bit of information that is needed to set up a filter. It is simply stored by the server and sent back whenever a match is made. This allows the same function to handle multiple filters, and know which filter it was that matched the message, simply by looking at the ID that is sent back. For the most part, this is not needed, and the ID may be set to 0.

The actual definition of each of these functions follows:

connectSDBus

Synopsis `SDBus *connectSDBus(char *hostname, int port)`

Parameters

`hostname` The host to connect to. The server should be running on this host, otherwise the connect will fail.

`port` The port on which the server is running.

Discussion The default port for the server is 8157, so that is most likely the value to use for this argument. The return value is a SDBus pointer which is used when referencing this bus later. See below for more information on starting the server.

addSDBusFilter

Synopsis

```
void
addSDBusFilter(bus, filter, func, ID)
SDBus *bus;
char *filter;
```

```
void(* func)(int, char *);
int ID;
```

Parameters

`bus` The bus to add the filter to.

`filter` A PERL regular expression with the "/" removed from either end.

`func` The function to call in the event of a match.

`ID` An itenger that will be passed, along with the broadcast message, to the function handling matches.

Discussion This function adds a filter to the bus. In order to reduce wasted bandwidth, a message is only sent to a client if the client expresses interest in it, by having it match a filter. When the connection to the bus is made, a thread is started to read from it. When the server detects a filter match, a message is sent to the client, and read by this thread. The message includes the function that will handle the message, an ID to identify which filter matched the message, and the text of the message itself. The client thread then dispatches the specified function with the text and ID parts of the message.

writeSDBus

Synopsis `void writeSDBus(SDBus *bus, char *str, int len)`

Parameters

`bus` The bus to write to.

`str` The string to write.

`len` The length of the string.

Discussion This function simply writes a string onto the specified bus.

The SDBus Server

The SDBus requires a master server to be running that keeps track of all clients that are connected to the bus, as well as what requests they want to listen to. If a new message is sent over the bus the message is forwarded only to those hosts that have requested to get that type of message, and it will be wrapped appropriately.

There are no arguments required to start the server, simply run the program `sdbserver`, and it will be ready. Debugging information is sent to the standard out, this may be redirected if it is not needed.

The SDBus in LISP

The nature of information that is likely to be going across the SDBus, and the manner in which it is most likely to be used make is desirable to be access the bus from within a LISP function. For that purpose, I have made a non-threaded version of the sdbus functions. They reside in the `sdbus_nt.o` object file.

The file `filters.lisp` contains function definitions for a LISP version of the API, which can be used by Allegro CommonLISP. The simplified LISP API is:

(lconnectSDBus `[` *hostname* `[` *port* `] ]` **)** This function returns a number that is the file descriptor of the bus. Hostname, if provided, is a string containing the host on which the server is running. Port, if provided is the port to connect to. The defaults are fox.cs.rochester.edu and 8157.

(laddSDBusFilter *bus filter func id***)** This function has no meaningful return value. Bus should be a bus file descriptor, as returned by `lconnectSDBus`. Filter should be a PERL regular expression which will match any message to be received by this callee. Func should be a LISP function that has been declared c-callable. Its arguments should be a signed-word (integer) which is the id to attach to this instance of this filter. This allows the same function to keep track of different filters by giving a different ID to each filter. This ID will be passed to the function along with the message. The second argument to the function is a pointer to a character. Finally, ID is the ID to pass to the function.

(lwriteSDBus *bus message***)** This function has no meaningful return value. Bus should be a bus file descriptor, as returned by `lconnectSDBus`. Message is the message to send to the bus.

(lreadSDBus *bus***)** This function has no meaningful return value. Bus should be a bus file descriptor, as returned by `lconnectSDBus`. When called, execution will block until there is a pending message sent across the bus if there is no pending message when called. This message will then be read, for its text, ID, and function address. The function address will then be called with the text and the ID as arguments.

(lcloseSDBus *bus***)** This function has no meaningful return value. Bus should be a bus file descriptor, as returned by `lconnectSDBus`. Upon being called, this bus' file descriptor will be closed, causing the SDBus server to remove any filters that are owned by that connection to the bus.

4.3 A Sample Application

For the purpose of an example of how the system could be used, a system for analyzing security images is described. The input to the system is a video stream coming from a video camera, and the output is a collection of pictures of suspects and the objects they moved.

To do this, a number of reusable visual modules are created for tasks such as frame differencing, flow computation, and so on. They are joined together using VideoBuffers, and output important information such as actor positions and object movements to the SDBus. Two (possibly remote) LISP processes then process this information to keep track of the position of actors and objects, as well as detect which actors are suspects. When a suspect is found, a message is sent back over the SDBus to the visual part of the system so that the image of the suspect may be saved. These interconnections are explained in figure 10.

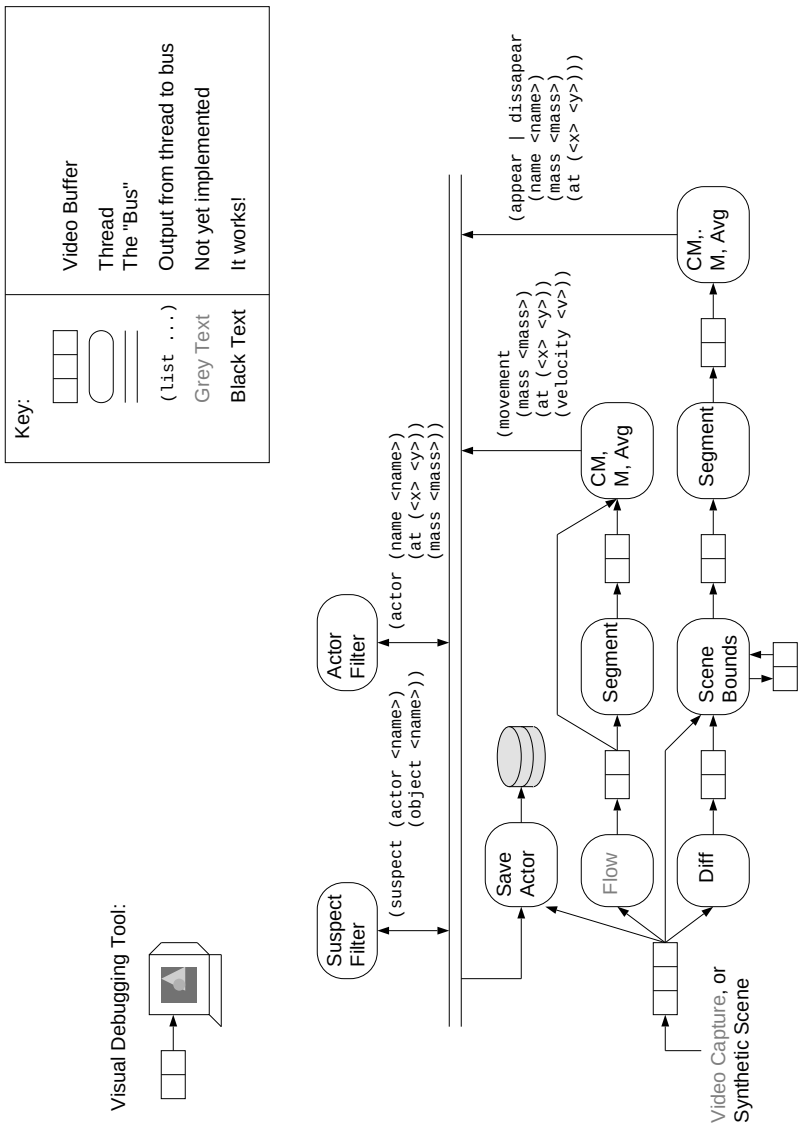

Figure 10: The design of a sample security application.

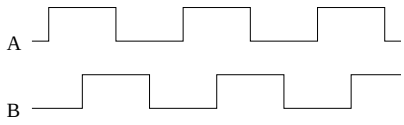

Figure 11: Sample output of a quadrature encoder. Here A leads B. If the shaft were to rotate in the opposite direction, A would follow B.

5 Low Level Control to Enable Visual Servoing

One of the primary goals of our wheelchair robot is the ability to navigate via visual control. In this section we discuss some of the foundations needed in order to perform this task.

One of the first things that comes to mind to try out when starting with visual control on the wheelchair is to try it oneself, using teleoperation. After hooking up the cameras, and writing an X based control program that allows teleoperation, we quickly discovered that controlling the wheelchair well is nearly impossible. Differences in calibration of the motor torques, differences in tire pressure, and the angles at which the small wheels are pointing all play a significant role in making the wheelchair go in a direction other than the one you tell it to go.

If humans are unable to drive the wheelchair in a straight line, it seems unfair to ask the visual behavior program to be able to do the same task, so a lower level stabilization is needed. A simple PID controller getting feedback from optical encoders mounted on the motor shafts can achieve this.

5.1 Hardware

Unfortunately, the encoders that come with the wheelchair are not suitable for this control application, because they do not give direction information. One would think that a state-vector this basic would have no need of a sensor: if we command forward we should *go* forward. However, if starting up when the front and rear casters in arbitrary positions, (as occurs after tight turns), the initial driving wheel velocities are in fact not determined by the command, and wheels can actually go backwards. We have replaced the original encoders with bi-directional hollow shaft encoders. Our controllability improved dramatically – we believe that this upgrade is a fine investment and recommend it highly.

The encoders give two outputs, A and B, which are square waves 90 degrees out of phase, as shown if figure 11. Although the 332 is capable of decoding this signal, the ARC system does not provide any way to do so in software, so a decoder is needed which converts this input into increment/decrement signals. This circuitry must then be incorporated into the tupperware box containing the 332 micro controller.

To this end, a sub-board with the circuit of figure 12 is made. It is then patched onto the Tin Man supplementary controller board. There are only 14 TPU channels provided by the Tattletale prototyping board that this is based on, which means the extra two channels must be taken from another function. Unfortunately, there are only two 5V TPU channel

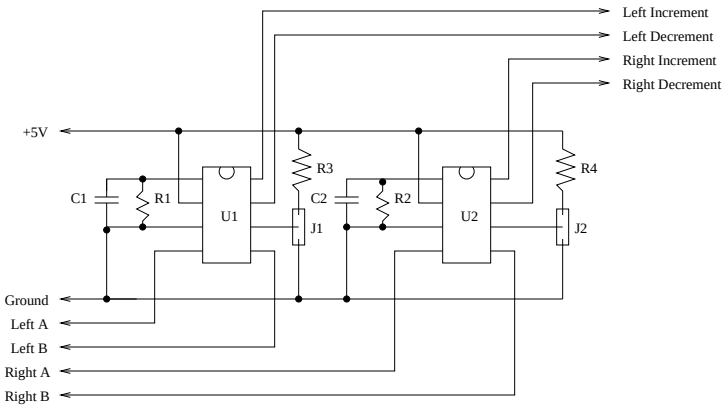

Figure 12: Decoder circuit for the wheel encoders. Parts are as follows: R1, R2 are 100kΩ. R3, R4 are 1.5kΩ. C1, C2 are 0.1μF. U1, U2 are US Digital LS7083. J1 and J2 control the output frequency, and should both be in the same position. In the up position, there will be an output signal for each transition of the square wave input, while in the down position there will be an output only for every complete period (every 4 transitions).

slots available on the TinMan version of this board, so they must be taken from either a 9 or 12V source. The two best candidates are connectors 14 and 15, which are used for LED 2 and 3 with TPU channels 6 and 7. To use these two slots, take the power from connectors 20 and 21, but run the signal over to the inputs for connectors 14 and 15. Of course this means that unless the reverse is done for these connectors, they will not be useful anymore.

One more complication in getting the decoder board patched in is to hook up the LEDs. They must be buffered, or the LED will keep the input to the chip low. There are several choices on how to do this. One is to disconnect the LEDs, while another is to add an LED buffer chip. Our choice was to use two of the channels of the existing LED buffer that are no longer needed for channels 14 and 15. They drive a pair of LEDs so the A and B inputs from an encoder will work in unison.

5.2 Software

After there is hardware support for this control, some software must be written to take advantage of it. This falls into two categories. First the provisions for reading from the encoders must be modified, and second a PID controller must be written.

Modifying the library is a simple matter of initializing the two extra channels (TPU channels 6 and 7), and in the function that updates the shared variables holding the counters, subtract the decrements in addition to adding the increments to the left and right counters.

That done, it is a simple matter to write the controller. First, there should be a control process, that will try to keep the chair at a given speed and heading. This information is

Connector	Sensor	Port	Volts
0	IR0	E0	12
1	IR1	E1	12
2	IR2	E2	12
3	IR3	E3	12
4	IR4	E4	12
5	IR5	E5	12
6	IR6	E6	12
7	IR7	E7	12
8	IR8	E8	12
9	IR9	E9	12
10	IR10	E10	12
11	IR11	E11	12
12	Unused	TPU4	9
13	Sonar1	TPU5	9
14	*Disabled*		
15	*Disabled*		
16	R_BUMP	AD3	5
17	L_BUMP	AD2	5
18	Unused	AD4	5
19	Unused	AD6	5
20	Left Inc	TPU6	5
21	Left Dec	TPU7	5
22	Right Inc	TPU8	5
23	Right Dec	TPU15	5

Figure 13: Connection for the modified TinMan Tattletale Model 8 board. Connectors 14 and 15 could be connected to AD5 and AD6, but are not used in our application.

stored in a global variable, and when drive and turn commands are received from the serial line, they are updated.

The control is done in a loop. At the start of the loop, the current errors for the drive and turn outputs are determined. Things would be easier if we had direct control over each wheel, because our sensors and controls would be directly matched. As it is we must determine our drive speed and turn speed based on what we see from the two wheels motion. To calculate the measured drive speed, d_m, take the average of the two wheel's speeds. The measured turn speed, t_m, is the difference between the left and right wheels. Let $\dot{x}_l$ and $\dot{x}_r$ be the left and right rates of change of wheel position. Thus

$$\begin{aligned} d_m &= \frac{\dot{x}_l + \dot{x}_r}{2} \\ t_m &= \dot{x}_l - \dot{x}_r \end{aligned}$$

Next the drive and turn speeds we would like to set (called d_s and t_s) can be defined as some constant times our inputs. This in turn allows us to define our driving and turning error as $d_e = d_s - d_m$ and $t_e = t_s - t_m$ respectively. The rate of change of error can be approximated by taking differences between successive iterations of the loop, and the integral of the error can be approximated by the following:

$$\begin{aligned} \int_0^t d_e dt &\approx \alpha \int_0^{t-1} d_e dt + (1-\alpha) d_e \\ \int_0^t t_e dt &\approx \alpha \int_0^{t-1} t_e dt + (1-\alpha) t_e \end{aligned}$$

The command to be set to the motors is then a weighted sum of the error, integral of error, and the derivative of the error.

There are some tricks that can make the controller work better. One is to have it defer its time slice a number of times before it does the control in each loop. This allows the estimate of velocity to be less prone to variation due to the speed at which it can sample the encoders. With our setup, it is capable of updating the control outputs at about 100Hz. Cutting this down to 25Hz still gives quite good response, but allows the encoders to see more ticks per time unit. One way to get the best of both worlds would be to get more ticks per revolution of the encoder. Our encoders have 40, which is close to what the original had, while it is possible to get up to 100. This coupled with the 4 times multiplication possible with the decoder board, would give 400 ticks per revolution.

A second trick is to note that the motors will not move at all with less than a 50% duty cycle. Therefore, it is a good idea to add 50, or -50, to the motor commands issued to the chair. This will significantly reduce oscillations that would be caused by this otherwise.

Another place to take careful attention is balancing the gains between turning and driving. Because the hardware motor controller has different magnitudes of output for turns and drives, the gain needs to be different to get good behavior.

5.3 Limitations

This controller is intended to be an aid for higher level control, and as such is not intended to solve all of the problems associated with controlling the chair. In fact, it can not hope

to with only the two wheel position sensors that it has as inputs.

One problem that it is that it will have a hard time going straight for a long period of time. Although it is good at overcoming the problems associated with the smaller wheels forcing it to one direction or the other at start up, once things get going it cannot entirely correct for at least one problem. This is that due to differences in tire pressure, the wheel diameters are not exactly the same, which means although its sensors may be telling it that it is going straight, it is not.

This can be overcome with a trim, but it does point out another shortcoming of the controller. These differences in diameter along with possible wheel slippage on some surfaces makes navigation based on these encoders prone to error. Thus a position control, rather than the described velocity control, is not really feasible.

Regardless of these problems, this is exactly what the higher level systems hope to solve using vision or other techniques. Eliminating the effects of the casters and differences in motor torques is still a great help for these systems, and what this controller was designed for.

6 A Virtual Environment Testbed for Driving a Wheelchair

6.1 The Possibilities of Simulation

Why and when should a simulator be used instead of the "real thing"? Factors for use in such considerations include the ease through which the real task may be performed, the time it takes to perform the real task, how dangerous the real task is to perform, the ease of simulation of the factors involved, and how easily the sensory input to the system may be modelled. How much each of these factors is weighed depends on the goals of the system.

For instance, consider the goal of training people how to use power wheelchairs as in [5]. This task is easy to perform in a real environment (just put the trainee in the wheelchair) and it takes an amount of time dependent on the trainee. The motivating factor of simulating the power wheelchair in a virtual environment is primarily because letting a trainee loose in a wheelchair might cause the trainee to have a dangerous accident and might also cause harm to the wheelchair. Of course, the main factor against simulation is that the wheelchair simulator may only serve a limited amount of individuals and developing a realistic simulator may prove very expensive.

Currently, wheelchair simulators are mainly used in the design of wheelchair systems [30; 29] and development in semi-intelligent wheelchair systems [25]. As Jari Ojala discusses, the use of simulators for these purposes contain advantages and disadvantages. The main advantages of such a system are:

1. Simulators may aid humans in finding fundamental design flaws.
2. Algorithms may be tried over and over on a simulator and their performance may be fine tuned.
3. Simulators are safe and predictable.
4. A simulator's environment may be slowed down in order to see problems in the system more clearly.

Of course this doesn't mean the simulator doesn't have associated problems. The biggest problem with any simulator is that it can not simulate every aspect of reality. For instance, a room in virtual reality lacks all room acoustics. Real sensors may be hard to model or the environment itself may prove difficult to model in a simulator.

While the exact nature of a sensor or environment may be near impossible to model, simpler, "mostly" correct models may be achievable. As an example, a ccd camera is modelled easily for a virtual reality simulator as it simply "takes a picture" of a scene from part of the virtual environment. Simulating ultrasonic sensors is much more difficult.

The technique discussed in [25] creates a computer graphics ultrasonic sensor. The time response $Y(t)$ in a space P (sensor's local coordinate system) may be calculated as follows when the input is an acoustical pulse:

$$Y(t) = A\frac{D_r(\theta)}{(2\pi r)^2}h(t - \frac{2r}{c}$$

where r is the distance of P and transducer center O. The D_r term is the directional characteristic of the sensor used, θ is the azimuth of P to the transducer center axis, A is a constant, and $h(t)$ is the impulse response of the sensor. The echo response $Z(t)$ is obtained through integrating this equation over the surface of the relevant objects.

$$Z(t) = \int Y(t)dS + N(t)$$

where $N(t)$ is a random noise generator. In order to shorten the integration the $h(t)$ term may be calculated once and stored in a static matrix. D_r may also be approximated using a simplification based on Bessel functions and the integral in $Z(t)$ may be calculated using the areas of the polygons of which the objects are formed. An even simpler approximation would be to shoot out a two dimensional "ray" from a graphical object in a certain direction and whatever the ray intersects is an obstacle.

6.2 Simulator Dynamics

I guess I never understood the big deal about this stuff simply because most of it is sort of "built into" the original SGI perfly simulator. Basically, I had to hack Roger's code down to 2nd order Runge Kutta (so it didn't mess up the speed of the simulation and go unstable). The SGI has a very nice clock function which is used by the original simulator in several places and which gets called whenever a drawing update happens, as that's when the car/wheelchair position gets updated.

Of course, none of this *guarantees* that you get real-time simulation. In fact, if other intensive apps are running in the background you don't get real-time sim. When nothing else is running and you use super user privileges to hog the processors you get from 30-60 Hz depending on the complexity of the world you're in (the lab is pretty simple and tends to run around 60 Hz while Garbis' town runs around 30 Hz with the dynamics code hooked in).

Some of the relevant simulator code, which gives a flavor of the parameters that are at issue and the complexity of the dynamic model, appears in Appendix A.

6.3 *PerSim*: A program for Virtual Reality Simulation

The origins of *PerSim* exist in a demonstration Silicon Graphics program called *Perfly*. The Perfly program allows a graphical database or group of scenes to be loaded and enables flying or driving around this scene with a mouse [11]. This program has subsequently been modified for Garbis Salgian's driving world [27] which has added automated object behavior as well as all items needed for a full virtual reality simulator. Garbis Salgian is currently in the process of interfacing this program to two Phantom robots in order for full haptic finger sensing in virtual reality environments.

The PerSim program currently contains an easily modifiable car dynamics module, the ability to add dynamic objects that move on a set path with a path file, the ability to add static objects to the graphical world, and contains a module for collision detection. The car dynamic initialization file contains modifiable parameters such as the maximum car speed and maximum acceleration while a car description file contains information about what the car will look like and the path of the car if it is loaded as a dynamic object. A dynamic model for a wheelchair is in the process of being implemented.

In order to avoid the problems associated with "works in progress", a working version of the PerSim program was copied to the directory /u/bayliss/sim/persim. In the future, all necessary modifications to this program will be communicated to other users of the program (when there are other users) in a timely manner and different versions of the program are already being kept track of through the use of RCS. In this manner, it is hoped that other individuals will not have to each keep and maintain their own version of PerSim (PerSim is not a small program), but will only need a copy of the part of the program that needs to be modified for their needs.

In virtual reality simulation, often one of the most important aspects of the simulation is how realistic the world appears. For instance, compare the pictures in Figure 14. In order to achieve this degree of realism it is necessary to model objects in the environment. This may be a time consuming task for complex objects such as computers and requires special software. The software used for the computer lab environment model is called *Showcase*. The Showcase tool not only allows the creation of three-dimensional models with a drag-and-drop method, but allows texture mapping and shadow creation for the models.

Only a few drawbacks exist when using the Showcase tool. The first is that Showcase does not easily allow the creation of models that need to be a fixed height, depth, and width. The "container" where models are created contains a grid where each square is 0.5 meters (or possibly 0.5', depending on what metric system the container uses). Unfortunately, when an object exceeds a certain size in one dimension, the program reduces the total amount of grid squares (by some multiple of 5). Thus, when a model for a software lab floor was extended to 50 meters, the 100 grid squares were cut down to 10 grid squares (each of size 10 meters). According to the on-line documentation, the grid squares exist only for "relative" size modeling between different objects.

While Showcase allows the creation of shadows for modelled objects, it does not allow the creation of different light sources. A tool called *SceneViewer* enables this and also allows light sources to be different colors. For both SceneViewer and PerSim, the Showcase model must be saved in *Inventor* file format. This particular file format is available from showcase, which saves Inventor files in binary Inventor format (rather than ASCII Inventor format).

After completing a model of the necessary environment for simulation, persim may be used to perform simulation tasks. The persim program allows easy run-to-run modifications through the use of initialization files and an extensive group of command line options. In this way, both static and dynamic objects may be added to an environment and the speed and behavior of the main object of simulation may be changed.

Command line options control the number of channels to be used, the color of the sky, the kind of driving (or flying) to be used, and tell various statistics about the program while it is running. Command line options are located in the cmdline.c file and keyboard options

(a)

(b)

Figure 14: (a) The software lab at the University of Rochester Computer Science Department. (b) A three dimensional model of the software lab.

(options available while running a program) are located in keybd.c. Note that all letters of the alphabet are in current use for command line options, so it is difficult to add this type of option to the persim program.

Three types of initialization files currently exist. Object description files describe static and dynamic object properties including the file containing object graphics as well as the position of the object. Dynamic objects also need the file name of a path file. All dynamic objects currently move on a set path and the object's path file contains the coordinates, angles, and speed of an object on this path. A demonstration object description file may be found in /u/bayliss/sim/persim/descr/objdescr. This file also contains miscellaneous information that Garbis Salgian uses for controlling the speed, steering, and braking of a go kart simulator.

A third type of initialization file is used by the wheelchair dynamics module and tells about the different properties of a wheelchair, such as the maximum allowable acceleration and the wheel radius. This initialization file allows the dynamics of the wheelchair to change as the physical makeup of the wheelchair changes. For information on the individual elements of the file, please see Roger Gans. A sample initialization file may be found in /u/bayliss/sim/persim/descr/init_file.

6.4 Plans for Using the Virtual Wheelchair Environment

One of the main goals of real-time EEG analysis is to allow handicapped people to drive a wheelchair using EEG information. In order to achieve this goal, several procedural steps must be accomplished. Artifact must be reduced or eliminated from the signal in order to maintain control. Higher dimensional control must be exhibited through pattern recognition techniques and experimentation. While past systems have shown only 1-dimensional control, higher dimensional control is needed in order to drive a wheelchair as the wheelchair must have the ability to go, stop, and turn in different directions. Moving a cursor on a screen is a simple task for use in experimentation.

After higher dimensional control has been achieved, it will then be necessary to experiment with wheelchair driving. In the early stages of trying to drive, people will most likely make many mistakes and will need a safe environment in which to make these mistakes. A virtual wheelchair simulator provides such an environment. As an added benefit, the virtual environment is completely controllable and obstacles may be added or removed in order to test the ability of a trained individual to move the wheelchair in the environment of the software lab. This will provide a measure of how well the system works before people are asked to try the system in the real world.

7 A Frame-Grabber Abstraction

The Matrox Meteor is a PCI frame grabber that allows one to capture color or monochromatic images at rates of up to 30 Hz and store them directly in the memory of a personal computer. For a detailed description of this device, we refer the reader to its WWW page: http://www.matrox.com/imgweb/meteorm.htm. A driver for it, to be used with the FreeBSD operating system, was originally developed by Jim Lowe and Mark Tinguely in 1995. In 1996, Jim Bray converted this driver to the Linux operating system. The resulting Linux version, currently on release 1.4, has been maintained by Ian Reid and is available free of charge. It works with kernel versions not older than 1.3.72, offering a lot of different configuration options to the user, but it is not exactly straightforward to use.

One problem is that it requires the preallocation of a big area of physical memory during the installation of the Linux kernel. The size of this area, which is reserved exclusively for the frame grabber internal data structures and for the grabbed images, determines the limits on the dimensions of the images that can be grabbed. However, this information is not directly available for the user programs, making the control of the image size a difficult issue. In real-time applications, it is interesting to divide this area in multiple buffers so that frames already grabbed don't need to be copied before the process of grabbing the next frame starts. However, this requires the user to manage certain internal data structures of the current version of the driver directly, not to mention the fact that in this case the maximum size of the images depends on the number of buffers used. In addition, there are certain restrictions in the sizes allowed for the images which are not very well documented. For instance, if the input for the frame grabber is in NTSC mode, then the number of rows must be a multiple of thirty and the number of columns must be a multiple of forty. And finally, it seems to us that some of the configuration options are not very useful for most users and the additional complexity needed to take care of all of them makes the driver not very user–friendly.

In order to simplify the usage of this driver, we developed a programmable interface, which hides most of its complexity, greatly reducing the application development time. We eliminate some of the options originally available (the most drastic restriction that we impose is the fact that the original driver supports input signals in NTSC, PAL and SECAM formats, but our interface was developed exclusively for NTSC), but we believe that for many real–time applications, the resulting simplicity more than compensates this loss of generality. From the point of view of the user, our interface is composed by two main elements: a configuration file that can be used to set the frame grabber *a priori* and a library with routines and global variables that provide some basic user–level instrumentation and allow the configuration of the frame grabber to be modified in real–time.

7.1 URCS Users

If you are a user at the University of Rochester Computer Science Department, then you may skip the installation (Section 7.2). Just log on the machine firestone.cs.rochester.edu and the INTERF_ROOT directory (which we mention in the following sections) is located at /home/carceron/meteor_int_1.4.1.

7.2 Installation

If you wish to use this interface but it is not available at your institution, you can obtain it free of charge via anonymous ftp. Notice that this code comes with no guarantee at all and its authors are not liable for any damage that its utilization may cause. Of course, a prerequisite for the utilization of this interface is the proper installation of the Matrox Meteor driver for Linux, version 1.4. So, assuming that the driver is properly installed, issue an anonymous ftp at ftp.cs.rochester.edu. After sending your complete e-mail address as password, follow the script below:

```
ftp> bin
ftp> cd pub/packages
ftp> get meteor_int_1.4.1.tar.gz
ftp> quit
linux % gunzip meteor_int_1.4.1.tar.gz
linux % tar xvf meteor_int_1.4.1.tar
linux % cd meteor_int_1.4.1/include
```

This will take you to a directory containing two header files. Now, you must edit the file meteorInstal.h, in order to inform the interface about the particularities in the Meteor's installation at your machine. More specifically, you will be required to provide four pieces of information:

- The path to the driver's header file called ioctl_meteor.h (according to the previous installation);
- The number of pages of physical memory (with 4 KBytes each) that were reserved to the frame grabber when the Linux kernel was compiled (according to the previous installation);
- The path to the meteor device (according to the previous installation);
- The default path to a file that is going to be used to define the initial configuration of the Meteor frame grabber (in this case you are free to choose any path that you want, but we suggest that you provide the path for the .meteor file located at the directory meteor_int_1.4.1 that you just created).

After the file meteorInstal.h is properly modified, you need to compile the interface, which can be done with the following commands:

```
linux % cd ..
linux % make
linux % pwd
```

At this point, the interface is ready to be used. Notice that after executing the commands above, your current directory should be the meteor_int_1.4.1 that was created during the installation process. From now on, we refer to the complete path for this directory (displayed by pwd) as INTERF_ROOT. You can check whether the installed interface works properly or not by executing the following sequence of commands:

```
linux % cd examples
linux % make
linux % make demo
```

This should create a small window that exhibits the images digitized from the first Composite Video input channel of the Meteor board (there may be a small initialization delay, and the colormap used to display the images may seem a bit strange). After you see the images in the screen, type ⟨Control–C⟩. The window should then disappear and some performance statistics should be displayed.

7.3 The Basic Programming Paradigm

The interface that we introduce here is primarily aimed at real–time applications. So, we adopt an event–driven programming paradigm, in which the role of the final user is to provide handling functions for the main events that can take place in the system. In practice, these handling functions are not actually used as handlers for events such as interrupts generated by the digitization of new frames. This job is done by the interface itself. When the interface then verifies that it is safe to start the processing of a new frame, it calls the proper handling function and provides a time stamp indicating the exact moment in which the frame in question arrived. This way, new frames can be received by the interface, in case an invocation of a handling routine eventually takes more time than the digitization period. But from the point of view of the user, the handling routines are executed atomically, in the sense that each one of them is guaranteed to finish before the next one is called.

More specifically, a typical user program written with this interface must include at least one call to a function called imgProcLoop, whose prototype is declared in the file INTERF_ROOT/include/grabber.h. This function consists of a real–time image processing loop that keeps track of the arrival of new digitized frames and invokes certain user–defined handling routines that it receives as parameters, whenever it is save to do so. imgProcLoop takes five arguments: the first two are the command–line variables argc and argv, and the last three are the user–defined handling routines (if the application does not need one or more of these, the user can simply pass NULL pointers instead).

Immediately after this function is called, the frame grabber is properly configured and started in a mode that grabs a single image. The function passed as the third argument of imgProcLoop (if any) is then called, receiving a pointer to the first byte of this initial image as argument. This function is intended to be an initialization function for the real–time image processing application. For instance, in a tracking system, it could be a function that recognizes a certain predefined target conveniently located in front of the camera. Then, the grabber is switched to a continuous capture mode and the function corresponding to the fourth argument of imgProcLoop (if any) is called every time a new image is available, also receiving a pointer to the first position of the image to be processed. The user can read and write directly to the buffers where the digitized images are stored by the frame grabber, because the interface guarantees that each buffer is not modified until the execution of its handling routine is over. Finally, whenever the image processing loop is terminated

(through a keyboard–generated interrupt, for instance), the fifth argument of imgProcLoop (if any) is invoked with no argument.

So, any application developed with our interface must include the header file grabber.h, located at INTERF_ROOT/include. This file, on turn, includes meteorInstal.h, located at the same directory. Thus, we recommend the utilization of the directive -I (with the path INTERF_ROOT/include as argument) when compiling. The resulting object files must be linked with the library INTERF_ROOT/lib/grab.a, which contains the implementation of the frame grabber interface. For debugging purposes, the user can link code with grabdb.a instead of grab.a.

To illustrate the description presented so far, we include the user code for a simple application that just grabs images and displays them remotely, using a set of X–Windows routines (some of which were implemented by other authors) declared in the file INTERF_ROOT/examples/src/xdisplay.c:

```
#include "grabber.h"
#include "xdisplay.h"

/* Trivial usage example: just grab the frames and display them using
   the functions defined in xdisplay.h */

int main (int argc, char * argv[])

{
  (void) imgProcLoop (argc, argv, setupDisplay, displayFrame, finishupDisplay);
}
```

Since the only argument to any user–defined handling routine (if any) is a pointer to the first byte of the frame to be processed, we use a global variable called geo (also declared in the file grabber.h) in order to provide the user with information about the dimensions of the incoming frames and the format in which their pixels are encoded. The variable geo is intended to be a read–only variable and the values of its fields **can not be changed directly be the user under any circumstances**, or unpredictable errors (such as the application crashing or hanging) may happen. It contains two fields called rows and columns which store the dimensions of the incoming frames and a field called oformat which is always equal to one of the following four constants: METEOR_GEO_RGB24, METEOR_GEO_RGB16, METEOR_GEO_YUV_PACKED or METEOR_GEO_YUV_PLANAR. For a detailed explanation of the meaning of the pixel formats represented by each of these constants, we refer the user to Section 7.4.

Using Command Line Arguments

The function imgProcLoop uses getopt to parse the command line arguments stored in argc and argv. This means that the user can define command line arguments in a completely independent way, but must necessarily parse these arguments with getopt too, or otherwise

they will be interpreted as arguments to imgProcLoop. If you are not familiar with this parsing routine for optional arguments, try man 3 getopt for details.

In general, the usage of an application built with our interface is:

⟨name⟩ [⟨userDefinedOptions⟩ −−] [⟨interfaceOptions⟩]

Notice that the user–defined options must appear always before the interface options and these two types of options must always be separated by the delimiter −− (double minus sign). The interface options may be any subset of the following list:

- –d⟨path⟩: ⟨path⟩ specifies the path to the device corresponding to the frame grabber. The default is the path specified during the installation (/dev/meteor0 for URCS users).
- –f⟨path⟩: ⟨path⟩ specifies the path to the file that contains the configuration options for the grabber interface. For URCS users, the default is INTERF_ROOT/.meteor
- –h⟨host⟩: ⟨host⟩ specifies the name of the machine in whose console the images will be displayed, if an X–Windows interface is used. The default is the current value of the environment variable DISPLAY. Actually, this environment variable must exist previously (possibly with a different or even undefined value) in order for this option to work.
- –s: do not display any statistics about the application when its execution is finished.

7.4 Configuration Options

So far, we described the steps needed to use the frame grabber in the simplest way possible. But of course, one of the main purposes of this interface is to allow the user to configure the frame grabber to meet the specific requirements of the intended application in a simple way. If this configuration is not going to be changed throughout the execution of the real–time image processing loop, then it can be performed with a configuration file, whose name and location are specified by the command line option –f (as shown above).

Each non–blank line on this file should contain the name of an option to be configured followed by the value to which the option must be set, and possibly some comments afterwards. In the default configuration file INTERF_ROOT/.meteor we show all the options available. Now we explain each one in detail.

Capture Modes

The original driver for the Matrox Meteor frame grabber (without this interface) can capture images in multiple modes of operation:

- **Single–Frame Capture**: The user explicitly issues a capture command to the frame grabber and the frame grabber outputs a single image to the beginning of the preallocated memory area. Whenever another image is needed, another capture command

must be issued. This mode has little programming overhead if compared to the other options, but it is not very efficient.

- **Asynchronous Multiple–Frame Capture**: The user issues a single command to start the capture and the frame grabber outputs new images to the beginning of the preallocated memory area as fast as it can. Whenever the image processing is over, the user issues another command to stop the frame grabber. This mode is also relatively simple to use and it is much faster than the previous one. However, the user has no control over the frequency of capture. Furthermore, successive images are written in the same memory location and the application may read different parts of multiple frames as if they had been digitized at a unique instant. So, this mode is ideal for applications that can get by with free–streaming data such as the simple grab–and–display usage example that we showed in the previous section.

- **Synchronous Multiple–Frame Capture**: Finally, there is this mode in which the user again issues a single command to start the capture, but the images are digitized at a fixed rate (30 Hz by default) and are stored at different buffers inside of the preallocated memory area. Whenever a new frame is digitized, a user–defined interrupt is generated by the driver. The user application is then responsible for catching these interrupts and calling the proper handling routines. This mode is perfect for real–time computer vision applications, but if the driver is used without our interface, it requires a considerable programming overhead. Initially the user must determine the number of buffers to be used. This number can never be smaller than three, but on the other hand the total amount of memory required by all buffers, with the selected image size, can not be greater than the size of the preallocated memory area less the space needed to keep the internal data structures of the driver. Among other internal data structures, the driver uses a bitmap to keep track of which buffers are free. Whenever the digitization of a new image is completed, the driver marks a selected output buffer as busy. However, it is up to the user to choose the order in which the stored frames will be processed. While this results in greater flexibility, it also adds the complexity of forcing the user to keep track of the states of the bitmap to select which frame must be processed next. In addition, the user is also required to determine the initial address in which the selected frame is stored and he must access the internal bitmap in the driver directly, to mark the buffers as idle whenever the processing of their frames is finished. Finally, the frame grabber is stalled whenever the number of busy buffers exceeds a certain maximum limit and the continuous capture is resumed whenever the number of busy buffers drops below a certain minimum limit. Unless our interface is used, it is also up to the user to determine these limits properly and to set them by modifying certain internal data structures of the driver.

Thus, one of the main sources of complexity in the usage of the original driver is that the user must write a completely different program, depending on which of these modes is desired. So, one of the main goals of our interface is to make the existence of these multiple modes transparent to the applications. Assuming that the initialization of many applications takes a time much longer than the average time needed to process an ordinary

frame, the interface always grabs the first image using the Single–Frame Capture mode. Then we give the user a choice of any of the two Multiple–Frame Capture modes for the real–time image processing loop. This choice is performed by inserting a single line with the keyword UseInterrupts followed by either On (Synchronous mode) or Off (Asynchronous mode), in the configuration file. If no line beginning with UseInterrupts is found, then the default option (DEFAULT_USE_INTERRUPTS) defined in the header file grabber.h is used.

If the user selects the Synchronous mode, then there is also the issue of how to select the next frame to be processed, in case there are multiple buffers available. Currently we offer two policies: process the frames in First In First Out order (FIFO) or use the Most Recently digitized Frame only (MRF) at any step and ignore all the old frames. In order to express this option, the user must insert a line in the configuration file containing the keyword RetrievePolicy followed by either FIFO or MRF. If no such line is present in the configuration file, then the default option, DEFAULT_POLICY (also defined in grabber.h) is used.

The FIFO policy is especially appropriate for real–time applications whose handling routines execution times have a relatively large variance, but still have an average inferior to the digitization period. In these cases, the use of the FIFO policy will allow eventual delays introduced by exceptionally slow executions of the handling routines to be compensated with the extra time not used when these routines can be executed relatively fast. This way, the application will be able to service all the incoming frames in a timely fashion, even if some of them take more time than the digitization period. However, if the average execution time of the handling routines is bigger than the digitization period, this policy will introduce bigger and bigger time lags until all the available buffers for incoming frames get filled with images awaiting service. In this extreme event, the frame grabber will be stalled and no further frames will be digitized until the user can service at least one of the pending images. So, in the cases in which the average execution time is bigger than the digitization rate (typically, non–real–time applications), the best option is to use MRF, which will discard incoming frames more frequently, but will guarantee that a delay bigger than the digitization period never occurs between the instant in which a (non–discarded) frame is grabbed and the instant in which it is processed.

Grabber Output Geometry

Another complex aspect in the usage of the frame grabber's driver is the proper selection of the geometry of the output generated by the grabber, which consists of the number of rows and columns in each frame, the number of buffers in the preallocated memory and the format used to represent each pixel. If the Matrox Meteor driver version 1.4 is used alone, then the user must make sure that there is enough preallocated memory to fit the selection of output geometry plus the internal data structures of the driver. Furthermore, the user must also respect other restrictions, such as the fact that with NTSC signal the number of rows and columns must be a multiple of thirty and forty, respectively, and the fact that the number of buffers must be at least three if the Synchronous Multiple–Frame Capture mode is selected.

On the other hand, if our interface is used, all these constraints are managed automatically. Initially, the user selects a desired image size (rows and columns) and pixel format. In order to set the number of rows in the image, the configuration file must contain a line starting with the keyword ImageRows, followed by an integer number. Similarly, in order to set the number of image columns, the keyword is ImageCols. If any (or both) of these keywords is not present, then the default values DEFAULT_ROWS and DEFAULT_COLS, defined in grabber.h, are used.

The pixel format is defined with the keyword OutputFormat, followed by one of the following four options: Rgb24Bits, Rgb15Bits, 422Yuv16BitsPacked or 422Yuv16BitsPlanar. Unfortunately, the exact meaning of each of these formats (with the exception of 422Yuv16BitsPlanar) seems to depend on hardware and software details such as model of the Meteor board and version of the Linux kernel. The difficulty is that, depending on these factors, the driver may use either a little endian encoding (in which the first byte of a word is the most significant), or a *big endian* encoding (in which the last byte of a word is the most significant). If you are a URCS user, then you can build your applications assuming that big endian is used. Otherwise, if you don't know which encoding is used by the driver installed in your machine, you can verify this by placing a (light) red pattern in front a color camera connected to the input of the frame grabber and then performing the capture with the Rgb24Bits format. Divide each resulting digitized image into four–byte words. If the average value of the second byte on each word is greater than the average value of the third byte on each word, then the encoding is little endian, otherwise the encoding is big endian. Having this in mind, the different output formats available can be described as follows:

- Rgb24Bits: This is a four–byte–per-pixel format. Each frame is an array with $4 \times$ ImageRows $\times$ ImageCols bytes, where the word composed by bytes $4i$ to $4i+3$ is used to represent pixel i. For each such four–byte word, the second, third and fourth most significant bytes represent the intensities of the red, green and blue bands, respectively. The most significant byte is always meaningless.

- Rgb15Bits: This is a two–byte–per-pixel format. Each frame is an array with $2 \times$ ImageRows $\times$ ImageCols bytes, where the word composed by bytes $2i$ and $2i+1$ is used to represent pixel i. The intensities of red, green and blue are encoded with 5 bits each. The most significant bit of each 2–byte word is meaningless, the next five most significant bits encode the intensity of red, the following five bits encode the intensity of green and the five least significant bits encode the intensity of blue.

- 422Yuv16BitsPacked: This is a four–byte–per–pair–of–pixels format. Each frame is an array with $2 \times$ ImageRows $\times$ ImageCols bytes. In this format, the word composed by bytes $4i$ to $4i+3$ is used to represent pixels $2i$ and $2i+1$. Within each such word, the most significant byte encodes the U components of the two pixels, the second most significant byte encodes the intensity (Y component) of pixel $2i$, the third most significant byte encodes the V components of the two pixels and the least significant byte encodes the intensity of pixel $2i+1$. So, for each pixel, the intensity is represented with 8 bits and the U and V components are represented with only 4 bits each.

- 422Yuv16BitsPlanar: This is a two–byte–per-pixel format, also with 8 bits for intensity and 4 bits for each of U and V. However, each frame is divided into five independent

arrays, laid in consecutive memory chunks: the first array has ImageRows × ImageCols bytes and encodes the intensity levels, so that the i–th pixel is represented by the byte i; the following four arrays have (ImageRows × ImageCols) / 4 bytes each and encode the U components of the even field, the V components of the even field, the U components of the odd field and the V components of the odd field, in this order. The main advantages of this format are that it is the only one whose meaning does not depend on the endian encoding, and that the intensity levels can be extracted directly if the incoming images are to be treated as being monochromatic.

After the geometry options described so far are read from the configuration file, our interface checks whether the numbers of rows and columns agree with the maximum and minimum limits allowed, expressed in the constants MAX_IMG_ROWS, MAX_IMG_COLS, MIN_ROW_INCREM and MIN_COL_INCREM (in grabber.h). Whenever these constraints are violated, the numbers of rows and columns are set to be equal to the appropriate limits and warnings tell the user that the initial selection was not feasible. Then, the interface makes sure that the number of rows and columns are multiples of MIN_ROW_INCREM and MIN_COL_INCREM, respectively. In case a violation is verified, each improper value is truncated down to the closest feasible value and a warning is printed. Finally, the interface allocates as many buffers as possible with the resulting geometry. If the resulting number of buffers is smaller than the minimum limit required by the selected capture mode, then the frame format (rows and columns) is scaled down in an approximately uniform way across the two dimensions (subject to the restrictions on the allowed values) until the amount of memory required by the resulting geometry with a minimum number of buffers is smaller than the amount of memory reserved for the frame grabber. Again, a warning is printed to make the user aware of the required resizing.

Grabber Input Source

The Matrox Meteor board has four Composite Video (CV) input channels and one S–Video input. If you are a URCS user, the CV inputs numbered zero to three correspond to the red, green, blue and black wires (in this order) in the cable plugged to the Meteor board of firestone. To select one of these input sources, the user must insert a line starting with the keyword InputSource, followed by either Camera0 or Camera1 or Camera2 or Camera3 or Rgb or Svideo. The first four options will result in the digitization of one of the four CV channels, which may contain either a monochromatic signal or a composite polychromatic signal. Another possibility is to treat the first three input sources as the red, green and blue bands of a RGB signal by using the Rgb option. Notice that in this case, three independent monochromatic signals obtained by three different cameras can be used. Then, if either Rgb24Bits or Rgb15Bits is used as the output format, the Matrox Meteor board will actually digitize stereo images (it is probably necessary to use the same Synch signal in all the cameras too). Finally, the Svideo option grabs images from the S–Video input. If no InputSource option is found in the configuration file, then a default option defined by the constant DEFAULT_INPUT_SOURCE (file grabber.h) is used.

Frame Acquisition Frequency

Another useful feature of the driver that our interface kept is the ability to change the frequency in which new frames are digitized by the meteor board. This can be accomplished simply by inserting a line in the configuration file with the keyword FramesPerSecond, followed by an integer not smaller than one and not bigger than thirty, which is interpreted as the desired frame rate, in Hertz. The default frame rate is defined by the constant DEFAULT_FPS (file grabber.h).

7.5 Interacting with the Frame Grabber

So far, we have described how to set all the configuration options available before the execution of the intended application begins. But in many cases, it is interesting to monitor the performance of the system periodically and modify the configuration of the frame grabber on–the–fly, in order to adapt to changing conditions either in the computational environment (for instance, CPU load peaks) or in the real world (for instance, the occurrence of multiple targets in the visual field, in the context of tracking).

As we mentioned in the previous section the choice of an ideal retrieval policy for the Synchronous Multiple–Frame Capture mode, for instance, depends on whether the average execution time of the frame–arrival handling routines is smaller than the frame grabbing period. So, the user might want to keep an estimate of this average time and then modify either the retrieval policy or the frame acquisition frequency accordingly. In the next subsection we describe the support that our interface provides for performance monitoring and in the following subsection we describe how to change the frame grabber configuration in real time, as required by the specific needs of the application at hand.

Using the Instrumentation Data

The instrumentation data collected by the interface upon the arrival of each new input frame can be viewed by the user through the global variable state, defined in the file grabber.h. This struct is supposed to be a read–only variable (like geo) and we strongly recommend (but not really require) that the users do not modify it in any way. Its most important field, which is called thisArrivedAt is a time stamp corresponding to the moment when the frame currently being serviced was received by the interface. In order to check the average delay between the reception of a frame and the invocation of its handling routine, the user can call the function local_time (also declared in grabber.h) in the beginning of the handling routine. The difference between the value returned by this routine and the value stored in state.thisArrivedAt will be equal to the delay before the processing of the current frame was initiated. If this delay remains bigger than the digitization frequency for several frames, this can mean that the application is not able to catch up with the frequency of the digitizer and some appropriate action (such as reducing the frequency or changing the policy to MRF) is needed.

Other fields of great interest in the variable state are the counters frameNum, totalReceived and totalQueueFull. The first two represent the number of frames completely processed so

far and the total number of frames received by the interface, respectively. So, if the difference between these two counters increases suddenly, this is an indication that either several frames were discarded or several frames were queued in the interface, creating a big time lag in the processing. In either case, some appropriate action must be taken. The counter totalQueueFull indicates the total number of times that the pool of buffers reserved for incoming frames became completely full, causing the frame grabber to stall. If this counter increases between two successive invocations of a user–defined handling routine which is supposed to run in real time, then some action must be taken immediately in order to increase the bandwidth of the system (for instance a reduction of the resolution of the images through a frame resizing). This will never happen if the MRF policy is being used, because new incoming old frames awaiting servicing will be immediately discarded upon arrival of new frames.

Finally, the interface also provides a rough categorization of the total time elapsed so far. This time is divided in three classes: the busyTime, spent in the execution of user–defined routines; the idleTime, in which the CPU was not used at all; and the ovhdTime, spent in the execution of the interface itself. A predominance of idleTime in general indicates that some processing capacity is being wasted and it may be a good idea to increase either the resolution or the digitization frequency (if possible). Notice that this will never happen if the Asynchronous capture mode is being used, because in this case the frequency of the capture–and–process cycle will be as high as possible.

Changing the Grabber Configuration During Execution

In order to change the configuration of the frame grabber in real time, the user must use two global variables declared in the file grabber.h: grabberAction and gActPars. The former is used to encode one or more commands to be performed by the grabber and the latter is used to pass the parameters of these commands. Because certain configuration changes in the frame grabber imply the deallocation of the frames currently stored in memory, the commands issued by the user do not take effect until the current handling routine invocation is over. After that, the commands are processed as soon as a new frame arrival interrupt is issued by the meteor device and no other handling routine invocation is performed meanwhile (even if there are old frames queued).

As shown in the file grabber.h, in the current version of the interface, there are six different commands available, each one encoded by a constant with the prefix G_ACT_. Each command (with exception of G_ACT_CH_MODE, which just flips back and forth between the two capture modes), requires appropriate fields of the variable gActPars to be set in advance. For instance, the command G_ACT_CH_SRC, which changes the geometry of the incoming frames, requires gActPars.rows to be set to the desired number of rows, gActPars.columns to be set to the desired number of columns and gActPars.oformat to be set to the desired pixel format, as explained in Section 7.4. Similarly, the commands G_ACT_CH_SRC, G_ACT_CH_POLICY and G_ACT_FPS require, respectively, the fields source, policy and fps to be set to appropriate values.

Since each of these commands is represented by a separate bit in the encoding space, the user can issue multiple commands at the same time by concatenating them with the |

(binary or) operator. For instance, suppose that the user wants to change the input source to the CV input number one and divide the current number of rows and columns by two at the same time. This can be accomplished by inserting the following segment of code in the application:

```
gActPars.source = METEOR_INPUT_DEV1;
gActPars.rows = geo.rows / 2;
gActPars.columns = geo.columns / 2;
grabberAction = G_ACT_CH_SRC | G_ACT_CH_GEO;
```

Since it is not always possible to execute the user commands exactly as issued (due to restrictions in the possible values of the parameters, as explained in the previous section), the interface sets the fields of gActPars to their actual current values upon completion of any command. It also resets grabberAction to the default value G_ACT_NONE, to avoid that a same command is accidentally repeated over and over. However, we strongly discourage the users from employing the values stored in gActPars to infer anything about the current frame being handled. In particular, the number of rows and columns of the current frame must be always obtained from the global variable geo, as explained in Section 7.3, and never from gActPars, especially in applications with X–Windows user interfaces. The reason for this is that, in our intended programming model, the event handlers in a X–Windows user interface directly set the variables gActPars and grabberAction in response to events such as the resizing of a display window. But due to the reasons explained above, these actions do not take effect from the point of view of the grabber until the next frame–arrival interrupt is generated. So, accessing the current image based on the values stored in gActPars can cause segmentation faults (in case a display window is resized to a bigger size, for instance).

A last word must be said about the special command G_ACT_STOP. As we mentioned in Section 7.3, the imgProcLoop routine, which is the basis for the development of user applications, consists of a loop that continues forever unless the user explicitly takes some action to stop it. One possible such action is to press ⟨Control–C⟩, generating a keyboard interrupt that results in the termination of the application and the exhibition of some general performance statistics (if DEFAULT_VERBOSE is set to 1 in the grabber.h file). However, in many cases, the user wants a given application to execute only for a certain predefined number of frames, or more generally, only until a certain predefined condition that can be verified automatically in execution time is met. For this reason, we provide the command G_ACT_STOP. This command results in the immediate interruption of the current call to imgProcLoop, as soon as the current user–defined handling routine is finished. By default, G_ACT_STOP deallocates all the internal data structures needed by the interface, closes the Meteor device, and prevents any further calls to imgProcLoop from having any effect (they will just return 0, while the usual return value is 1). However, if the application potentially needs to perform additional calls to imgProcLoop, the user can set gActPars.halt to 0. This will stop the continuous capture, as usual, but will keep the Meteor device allocated exclusively to the user, allowing future calls to imgProcLoop. For safety reasons, gActPars.halt is reset to 1 at every iteration. In any case, the user is advised to have gActPars.halt set to 1 when the last G_ACT_STOP command is issued by the application.

A good example of the programming paradigm presented in this section can be found in the file autoresize.c, which is located in the directory INTERF_ROOT/examples/src.

8 A Visual Servoing Application: Chair-Following

8.1 Introduction

One of the main obstacles to the practical feasibility of many computer vision techniques has been the necessity of using expensive specialized hardware for low–level image processing, in order to achieve real–time performance. However, gradual improvements in the architectural design and in the manufacturing technology of general–purpose microprocessors have made their usage for low–level vision more and more attractive. In this report, we demonstrate the real–time feasibility of a tracking system for smart vehicle convoying, implemented entirely on a dual Pentium processor. The task at hand consists of enabling an autonomous mobile robot with a camera to follow a target placed on the posterior part of another mobile platform controlled manually.

The key ideas used to achieve efficiency are quite traditional concepts in computer vision. In the low–level image processing front, we use multi-resolution techniques to allow the system to locate quickly the regions of interest on each scene and then to focus its attention exclusively on them, in order to obtain accurate geometrical information at relatively low computational cost. In the higher–level processes of geometrical analysis and tracking, the key idea is to use as much information available *a priori* about the target as possible, in order to develop routines that combine maximum efficiency with high precision, provided that its specialized geometry and dynamics assumptions are met. One serial connection between the vision computer and the wheelchair's microcontroller supports five logical streams of data for control purposes.

So, while we do introduce some novel formulations, especially in the context of geometrical analysis of the scenes, the main goal of the present work is clearly to demonstrate that by carefully putting together several well–established concepts and techniques in the area of Computer Vision, it is possible to tackle the challenging problem of smart vehicle convoying with low–cost equipment.

In Section 8.2, we discuss some of the related work in the areas of tracking and pose estimation. In Section 8.3 we discuss our approach for recovering three–dimensional information and exploiting the temporal coherence of sequences of images. In Section 8.4 we discuss the aspects related to the problem of low level image processing: how to extract and correctly identify useful features in the input images. In Section 8.5 we present our approach to deal with the problem of visual control, that is, how to control the speed and steering of the trailing robot so that it will not loose track of the leading mobile platform. Section 1 briefly describes the hardware, and Section 2 describes the serial protocol.

8.2 Background

The task of tracking a single target can be divided in two parts: *acquisition* and *tracking proper* (below, simply *tracking*) [9]. Acquisition involves the identification and localization (possibly via motion detection and segmentation) of the target, as well as a rough initial estimation of its pose (position and orientation), velocities and possibly some other *state–variables* of interest. This phase is in some aspects quite similar to the problem of object

recognition. Usually, generality is more important at this point than in the subsequent tracking phase, because in several practical applications, many different targets of interest may appear in the field–of–view of the tracking system and thus it is not possible to use techniques that work only for one particular type of target.

The information obtained in the acquisition phase is then used to initiate the tracking phase, in which the target's state–variables are refined and updated at a relatively high frequency. In this report we argue that in this phase, after the target has been identified and its initial state has been properly initialized, all the specific information available about its geometry and dynamics should be exploited in the development of specialized routines that are appropriate for real–time usage and, still, require only inexpensive general–purpose hardware.

As suggested by Donald Gennery [9], the tracking phase, which is the major problem studied here, can be divided into four major subtasks:

1. **Prediction:** Given a multi-valued time–series with the history of (noisy) measurements of the target state–variables performed so far, it is necessary to predict the values of these variables in the next sampling instant, so that the tracking system can always restrict its search for the target to a relatively small part of the scene. Traditionally, this extrapolation of the values of the state–variables is done recursively, for efficiency reasons. In other words, at any point in time, the tracker keeps only a vector of current estimates for the *true* (as opposed to *measured*) values of the state–variables and some of the higher–order moments of the multi-valued time–series (typically the covariance matrix). Then, given the new measurements, all these variables are extrapolated for some future instant and the whole process can be repeated as soon as the another set of measurements is available.

2. **Projection:** Then, given the predicted pose of the target and a certain model for the 3-D–to–2-D transformation performed by the camera, it is necessary to determine the appearance of the target in the scene. Typically, this task is formulated as the determination of positions, orientations or apparent angles, and the visibility analysis, for a set of distinctive target features.

3. **Measurement:** The next step is to search for the expected visible features in the image. The problem of identifying which image features correspond to each model features, known as the *correspondence problem*, is the hardest aspect of this task. However, this problem can be avoided (or at least ameliorated) if the features are distinctive enough to be uniquely distinguished regardless of the pose of the target (via color information, for instance). Another useful trick is to make the other three steps of the tracking phase tolerant to false matches in the solution of the correspondence problem, via the use of robust statistics.

4. **Back-projection:** Finally, it is necessary to compute the discrepancy between the actual image measurements and the image measurements that would be expected given the analysis performed in the Projection step. Of course, it is also necessary to determine how this discrepancy affects the current estimate for the state of the target.

Ultimately, some sort of back-projection from the 2-D image plane to the 3-D scene space is needed to perform this task.

In our tracking system, one of the steps in which we exploit most heavily the availability of *a priori* information about the target in order to improve efficiency and accuracy is Back-projection. More specifically, at this point (as well as in the Projection step) we make use of the fact that our target is a rigid object composed by points whose relative 3-D positions are known *a priori*. The problem of recovering the pose of a three–dimensional object from a single monocular image, given a geometrical *model* for this object, has been heavily studied in the last two decades or so. The solutions proposed in the literature can be classified, according to the nature of the imaging models and mathematical techniques employed, as: analytical perspective, affine and numerical perspective.

The general idea of the analytical solutions is to work with a fixed number of known correspondences between model and image features. Then, they express image properties such as feature positions [10; 13; 1; 8; 17], orientations [7; 23] or apparent angles [33; 28; 22] as a function of a predefined set of pose parameters. By matching the resulting expressions against the corresponding actual image measurements, one can derive a set of polynomial equations involving the pose parameters. Finally, if the number of correspondences is big enough, these equations can be combined algebraically, yielding the desired pose.

The problem with this approach is that, if the imaging process is modeled as a perspective transformation, then the equations that relate the image measurements to the three–dimensional geometrical information known *a priori* are non–linear. Because of this, all the solutions cited above work only for up to four features and can not be efficiently extended to deal with more complex geometrical shapes. Furthermore, it is known that any analytical solution based on fewer than six point correspondences is necessarily ambiguous [8], yielding multiple plausible answers for certain problem instances. In addition, due to the use of multiple non–linear constraints, most of the techniques mentioned above rely on finding the roots of polynomial equations with degree higher than four. Unfortunately, there is provably no closed–form solution for this problem itself. Finally, many of these techniques have very poor error propagation properties. In a survey performed by Haralick and Lee [10], all the techniques tested were found to produce large errors (at least 0.1% in the actual 3-D positions of the object features,) just as a result of the propagation of rounding errors with single precision arithmetic. Of course, this problem becomes much more serious if we take into account the effect of quantization noise in the imaging process, for instance.

So, the source of most of these problems is the intrinsic non–linearity of the geometrical constraints that arise when the imaging process is modeled as a perspective transformation. Under certain special conditions, an imaging transformation that is actually perspective (or even more complex, if we take into account lens distortion, for instance), can be reasonably approximated with much simpler models. Techniques based on linearized camera models [2; 15; 14] are also simple, efficient, and, contrary to the analytical solutions, they work for scenes with arbitrarily many features. However, they are not able to cope with significant perspective distortion and, unfortunately, since we use a camera with a relatively wide field–of–view to avoid losing track of the target in our application, we have to face this type of complication.

So, an ideal pose recovery algorithm should combine the generality of a perspective camera model with the robustness of the schemes based on affine approximations. Indeed, it is possible to satisfy this requirement in practice by casting the problem of pose estimation into an equivalent multivariate numerical parameter estimation problem. There are at least two distinct ways in which this can be done.

The most traditional, straightforward, and widely used numeric approach consists of defining a measure of the discrepancy between the actual image measurements and the measurements that would be expected given a perspective camera model and an arbitrary estimate for the unknown pose. Then, by replacing the chosen error measure (which is a non–linear function of the pose parameters) with a local linear approximation around the point corresponding to the current pose estimate, one can compute a correction that in general yields a better pose estimate. This process can be iterated until (ideally) the error function is locally minimized and the current pose estimate converges to the actual pose, within a predefined desired precision.

David Lowe [21; 20; 19] proposed a classic solution along this line. Given a certain pose estimate, his algorithm computes the expected values for a vector of measurements (positions or orientations) in the resulting image, using a non–linear projective model. Then, Newton's iterative gradient method is employed to minimize this error vector in a least–squares sense. Lowe's algorithm was later improved by other researchers through the use of more accurate projective models [3; 32; 16] and optimization techniques with better convergence properties [26]. Similar solutions were also proposed for the specific case of independent orientation recovery from line correspondences [26; 18; 34].

A more recent approach, suggested by DeMenthon and Davis [6], consists of computing an initial estimate for the pose with a weak perspective camera model and then refining this model numerically, in order to account for the perspective effects in the image. The key idea is to isolate the non–linearity of the perspective projection equations with a set of parameters that explicitly quantify the degree of perspective distortion in different parts of the scene. By artificially setting these parameters to zero, one can then generate an affine estimate for the pose. Then, the resulting pose parameters can be used to estimate the distortion parameters and this process can be iterated until the resulting camera model (presumably) converges to full perspective. Oberkampf *et al* [24] extend DeMenthon–Davis's original algorithm to deal with planar objects (the original formulation is not able to handle that particular case) and Horaud *et al* [12] propose a similar approach that starts with a paraperspective rather than a weak perspective camera model.

The main advantage of this kind of approach is its efficiency. Like the optimization–based techniques, each iteration of the algorithms based on initial affine approximations demands the resolution of a possibly over-constrained system of linear equations. However, in the latter methods, the coefficient matrix of this system depends only on the scene model and thus its (pseudo) inverse can be computed off–line, while the optimization–based techniques must necessarily perform this expensive operation at every single iteration. [6].

However, all the solutions mentioned so far are much more general than the application that we have in mind, namely: track and follow a target placed on the posterior position of a non–holonomic vehicle whose motion is roughly constrained to a plane perpendicular to the image plane of the (single) camera used. Most of the model–based pose recovery

algorithms available in the literature do not impose any restriction on the possible motions of the target and thus use camera models with at least six degrees of freedom, such as the *perspective*, the *weak perspective* and the *paraperspective* models.

Wiles and Brady [31] propose some simpler camera models for the important problem of smart vehicle convoying on highways. In their analysis, they assume that a camera rigidly attached to a certain trailing vehicle is used to estimate the structure of a leading vehicle, in such a way that the paths traversed by these two vehicles are composed exclusively by a series of translations and rotations parallel to a unique "ground plane". In spite of the focus of their research being the recovery of structure from motion, many of their observations and suggestions can be generalized to the analogous problem of pose estimation.

Clearly, the application–specific constraints reduce the number of DOF in the relative pose of the leading vehicle to three. Because the camera does not undergo any rotation about its optical axis, the $\mathbf{x}$ axis of the camera frame can be defined so as to be parallel to the ground plane. Furthermore, the tilt angle between the camera's optical axis and the ground plane (α) is fixed and can be measured in advance. In this situation, the general perspective camera can be specialized to a model called perspective *Ground Plane Motion* (GPM) camera, whose *extrinsic parameter matrix* is much simpler than that of a six–DOF perspective model.

In our application we take this idea to an extreme. We not only simplify even more the model proposed by Wiles and Brady with the assumption that the image plane is normal to the ground plane ($\alpha = 0$), but also use a specially engineered symmetrical pyramidal target, in order to make the problem of inverting the perspective transformation performed by the camera as simple as possible. In addition, inspired by the work of DeMenthon and Davis [6], we adopt a solution based on the numerical refinement of an initial weak perspective pose estimate, in order to obtain accuracy at low computational cost. But rather than starting from scratch and iterating our numerical solution until it converges for each individual frame, we interleave this numerical optimization with the recursive estimation of the time series equivalent to the state of the target, as suggested by Donald Gennery [9]. So, only one iteration of the numerical pose recovery is performed per frame and the temporal coherence of the visual input stream is used in order to keep the errors in the estimates for the target state down to an sufficiently precise level.

8.3 Tracking the Target with the Use of 3-D Information

To some extent all computer vision is "engineered" to remove irrelevant variation that is difficult to deal with. Still, our engineering takes a fairly extreme form in that we track a specially-constructed three-dimensional calibration object, or target, attached to the lead robot. Our special target, placed on the back of the leading mobile robot, consists of two planes parallel with respect to each other and orthogonal to the ground plane, kept in fixed positions with respect to the mobile robot by a rigid rod, as shown in Fig. 15(a). The plane closer to the center of the leading robot (typically further away from the camera) contains a rectangle composed by four identical circles with a diameter of 4.0 inches each, so that two of the virtual edges defined by these points are parallel to the ground plane, as shown in Fig. 15(b). The other target plane (more distant from the leading robot) contains a unique

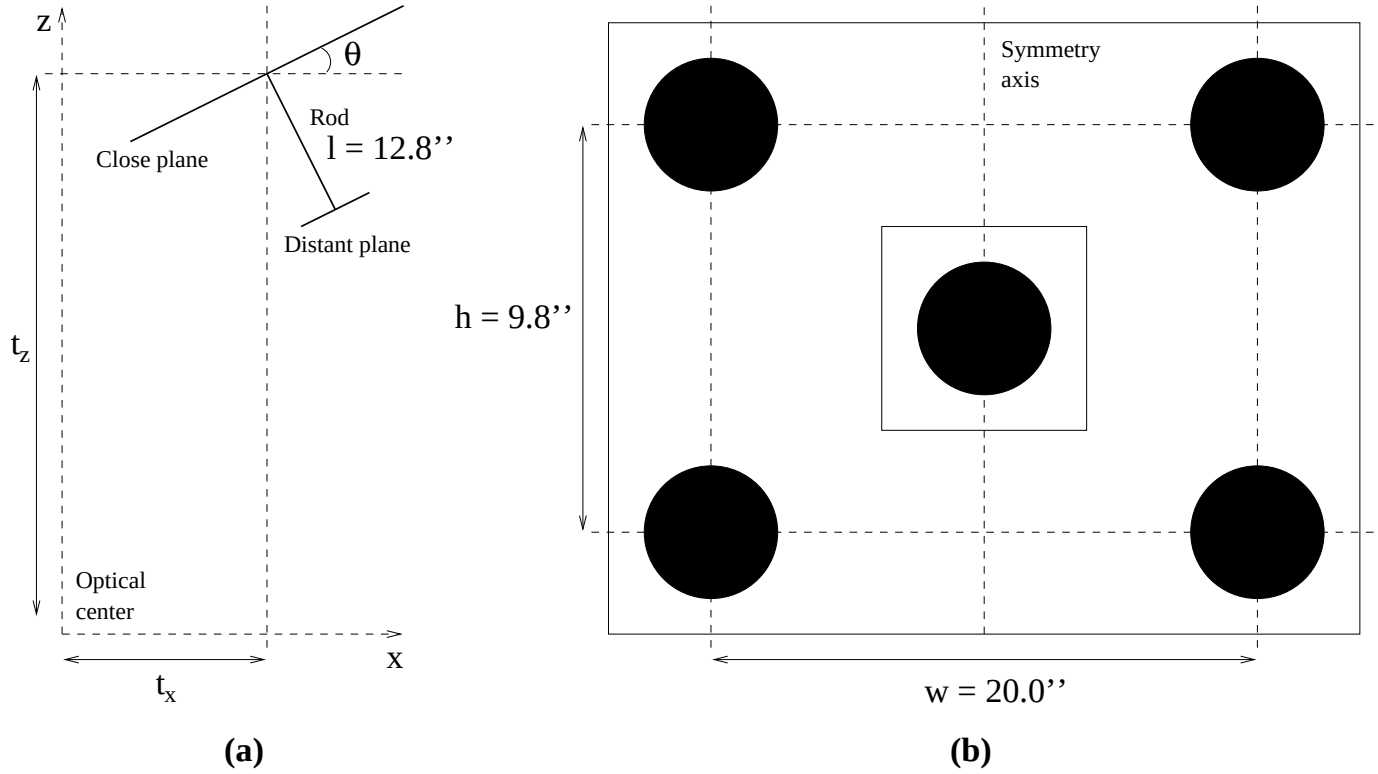

Figure 15: Geometry of the target placed on the posterior part of the leading vehicle: (a) top view; and (b) frontal view.

circle with a diameter of 3.5 inches. These two planes are arranged so that the orthogonal projection of the this 3.5–inch circle on the plane closer to the leading robot lies on the axis of vertical symmetry of the rectangle composed by the other four points (Fig. 15(b)).

From the point of view of our tracking algorithm, the state of this target is described with respect to a coordinate system attached to the camera, whose **x** and **y** axes correspond to the horizontal (rightward) and vertical (downward) directions on the image plane, respectively, and whose **z** axis corresponds to the optical axis of the camera (forward). Due to the ground–motion constraint, the target has only 3 DOF with respect to the camera. The state–variables used to encode these DOF are: the distances between the camera's optical center and the centroid of the target's rectangle along the **x** and **z** axes, denoted by t_x and t_z, respectively, and the counterclockwise (as seen from the top) angle between the **x** axis and the plane that contains the rectangle, denoted by θ.

At each step of the tracking phase, the tracker initially performs a *a priori Prediction* of the state of the target, based uniquely on the history of the values for the state–variables. More specifically, since our mobile robots can stop and turn quite sharply, we perform this prediction with a simple velocity extrapolation for each state–variable, because under these circumstances of a highly-maneuverable target and rather slow update rates, more complex filtering is impractical and destabilizing. Let $\hat{v}^{(i)}$ and $v^{(i)}$ denote the *estimated* and *measured* values for state–variable v at step i, respectively, where v is one of t_x, t_z and θ. Then the

Circle	x_i	y_i	z_i
Top left	$-w/2$	$-h/2$	0
Top right	$+w/2$	$-h/2$	0
Bottom left	$-w/2$	$+h/2$	0
Bottom right	$+w/2$	$+h/2$	0
Central	0	h_c	$-l$

Table 7: Coordinates of the target centroids in the model reference frame. Here w denotes the centroid–to–centroid horizontal width of the target's rectangle, h denotes the centroid–to–centroid vertical height of the target's rectangle and l denotes the orthogonal distance between the two parallel planes that compose the target.

predictions performed by the tracker are:

$$\begin{cases} \hat{t}_x^{(i)} &= 2\,t_x^{(i-1)} - t_x^{(i-2)}, \\ \hat{t}_z^{(i)} &= 2\,t_z^{(i-1)} - t_z^{(i-2)}, \\ \hat{\theta}^{(i)} &= 2\,\theta^{(i-1)} - \theta^{(i-2)}. \end{cases} \tag{1}$$

The predicted values for the state–variables are used to compute the appearances, on the image plane, expected for the five circles that compose the target. This corresponds to the *Projection* step, according to the outline presented in Section 8.2, and amounts to projecting the known geometry of the target, according to our simplified perspective GPM camera model. The model reference frame is defined so that its origin is the centroid of the target's rectangle, its **y** axis is aligned with the **y** axis of the camera frame and its **z** axis, when placed at the origin, points in the opposite direction of the centroid of the central circle. So, using the fact that the target has vertical symmetry, one can express the coordinates of the circle centroids in this frame according to the Table 8.3.

According to our imaging model, the projection equation that yields the image coordinates of an arbitrary point i, $[u_i, v_i]^T$, as a function of its coordinates on the model reference frame, $[x_i, y_i, z_i]^T$, is:

$$[u_i, v_i, 1]^T = \lambda\, M_{int}\, M_{ext}\, [x_i, y_i, z_i, 1]^T, \tag{2}$$

where the matrix of *intrinsic camera parameters*, M_{int}, (calibrated *a priori*) and the matrix of *extrinsic camera parameters*, M_{ext}, (estimated by the tracker) are given by:

$$M_{int} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3}$$

$$M_{ext} = \begin{bmatrix} \cos\hat{\theta} & 0 & -\sin\hat{\theta} & \hat{t}_x \\ 0 & 1 & 0 & h_0 \\ \sin\hat{\theta} & 0 & \cos\hat{\theta} & \hat{t}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4}$$

These predicted appearances are then used in the *Measurement* phase, which corresponds to the low level processing of the current input image, as we describe in the Section 8.4.

Finally, the low–level image processing module returns the positions, measured in the image plane, for the apparent centroids of the target circles, which are used in the *Reprojection*, yielding the *measured* state of the target in the current step of the tracking phase. Let the apparent centroids of the top left, top right, bottom left, bottom right and central circles be denoted by $[u_{tl}, v_{tl}]^T$, $[u_{tr}, v_{tr}]^T$, $[u_{bl}, v_{bl}]^T$, $[u_{br}, v_{br}]^T$ and $[u_c, v_c]^T$, respectively. In order to simplify the derivation of the equations that yield the measured state–variables t_x, t_z and θ, we define the *image measurements* m_x, m_z and m_θ, as follows:

$$m_x = \frac{u_{bl} + u_{tl} + u_{br} + u_{tr}}{4} - u_0, \tag{5}$$

$$m_z = \frac{v_{bl} - u_{tl} + u_{br} - u_{tr}}{2}, \tag{6}$$

$$m_\theta = u_c - u_0. \tag{7}$$

By replacing the predicted state–variables in Eqs. (2) to (4) with their measured counterparts and substituting the resulting expressions (as well as the centroid coordinates given in Table 8.3) into Eqs. (5) to (7), one can express the image measurements above as a function of the state–variables:

$$m_x = \frac{f_x}{2}\left(\frac{t_x - w\,\cos\theta}{t_z - w\,\sin\theta} + \frac{t_x + w\,\cos\theta}{t_z + w\,\sin\theta}\right), \tag{8}$$

$$m_z = \frac{f_y}{2}\left(\frac{h}{t_z - w\,\sin\theta} + \frac{h}{t_z + w\,\sin\theta}\right), \tag{9}$$

$$m_\theta = f_x\left(\frac{t_x + l\,\sin\theta}{t_z - l\,\cos\theta}\right). \tag{10}$$

In order to perform the *Back-projection*, we need to solve the system above for the unknown pose parameters t_x, t_z and θ. A possible approach would be to try to combine these equations analytically, but due to the nonlinearity of the camera model, this approach is likely to result in a solution with ambiguity problems and poor error propagation properties. Instead, we exploit the temporal coherence of the sequence of images through a numerical algorithm that is iterated for successive input images, in order to recover precise values of the pose parameters. We start with Eq. (9), since that is the only equation in the system above that involves only two of the three unknowns: t_z and θ. Instead of trying to solve for both unknowns at the same time, we use the measured value of θ from the previous step of the tracking process in order to get the estimated value for t_z at the current step:

$$t_z^{(i)} = \frac{f_y\,h + \sqrt{(f_y\,h)^2 + \left(2\,m_z\,w\,\sin\theta^{(i-1)}\right)^2}}{2\,m_z}. \tag{11}$$

Now we solve Eq. (8) for the unknown t_x. Of course, the resulting expression still depends on both t_z and θ. The value of t_z in the current tracking step has just been computed according to Eq. (11) and can be used in the recovery of t_x. But the value of θ is still unknown in the current step and thus, it is obtained from the previous step:

$$t_x^{(i)} = \frac{m_x}{f_x}\left(t_z^{(i)} - \frac{w^2\sin^2\theta^{(i-1)}}{t_z^{(i)}}\right) - \frac{w^2\sin\theta^{(i-1)}\,\cos\theta^{(i-1)}}{t_z^{(i)}} \tag{12}$$

Finally, Eq. (10) can be solved directly for θ, after the values of t_z and t_x are both known. Initially, we rewrite it as:

$$\sin\theta + \frac{f_x}{m_\theta}\,\cos\theta + \frac{f_x\,t_z}{m_\theta\,l} - \frac{t_x}{l}$$

Notice that this expression is on the form:

$$\sin\theta + c_1\,\cos\theta + c_2 = 0, \quad \text{with:} \quad c_1 = \frac{f_x}{m_\theta}, \quad c_2 = \frac{f_x\,t_z}{m_\theta\,l} - \frac{t_x}{l}.$$

So, we rename $\sin\theta$ as a new variable and use the trigonometric identity $\cos\theta = \sqrt{1-\sin^2\theta}$ to reduce the equation above to a quadratic form. By checking the two roots of the transformed equation for consistency with the original form, we can determine a unique solution for θ:

$$\theta = \sin^{-1}\left(\frac{-c_2 - c_1\sqrt{c_1^2 - c_2^2 + 1}}{c_1^2 + 1}\right).$$

Substituting back the original expressions renamed as c_1 and c_2 and simplifying the resulting equation, we obtain the final formula for θ:

$$\theta^{(i)} = \sin^{-1}\left(\frac{f_x\,k_2 - m_\theta\sqrt{k_1 - k_2^2}}{k_1}\right), \quad \text{where:} \quad k_1 = f_x^2 + m_\theta^2, \quad k_2 = \frac{m_\theta\,t_z^{(i)} - f_x\,t_x^{(i)}}{l}. \tag{13}$$

Eqs. (11) to (13) allow one to perform pose recovery recursively, using the solution found in the previous step as an initial guess for the unknown pose at the current step. However, we still need an initial guess for θ at the first time that Eqs. (11) and (12) are to be used. Our choice, in this case, is to set $\theta^{(0)} = 0$, reducing the equations mentioned above to:

$$t_z^{(1)} = \frac{f_y\,h}{m_z}, \tag{14}$$

$$t_x^{(1)} = \frac{m_x\,t_z^{(1)}}{f_x}. \tag{15}$$

Notice that this amounts to a weak perspective approximation, since $\theta = 0$ implies that all the four vertices of the target rectangle that is used to recover t_z and t_x are at the same depth with respect to the camera. So, in this sense, our pose recovery algorithm is inspired in the scheme proposed by DeMenthon and Davis [6], because it starts with a weak perspective approximation and then refines the projective model iteratively, in order to recover a fully perspective pose. As we mentioned in Section 8.2, the basic differences are that we use a much more specialized camera model, with only three DOF (as opposed to six in DeMenthon–Davis's algorithm), and we embed the refinement of the projective model in successive steps of the tracking phase, rather than starting all over from scratch and iterating our algorithm until it converges for each frame. This is a way of exploiting the temporal coherence of the input images to achieve relatively precise pose estimates at low computational cost. Finally, one last difference between our work and DeMenthon–Davis's

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 16: Discrete Gaussian kernel used in the subsampling process; $\sigma = 0.8493 \pm 0.0001$.

method is that our initial solution is not completely affine, since the equation for θ takes into account the distortion caused by the fact that the central circle in the target is located at a different plane than the other four circles. In fact the three-dimensionality of the target is very important: For small variations of heading, the three-dimensional target exhibits first order effects (proportional to the sine of the angle near zero) but a two-dimensional target exhibits only second-order effects proportional to the cosine of the angle near zero.

8.4 Efficient Low Level Image Processing

The image acquisition is performed with a Matrox Meteor frame grabber. In order to achieve maximum efficiency, this device is used in a mode that reads the images directly to the memory physically addressed by the Pentium microprocessor, using multiple preallocated buffers to store successive frames. This way, the digitized images can be processed directly in the memory location where they are originally stored, while the following frames are written to different locations.

The initial step of the low–level image processing is the construction of a multi-resolution pyramid. In the current implementation, we start with digitized images of size 180×280. On each of the lower resolution levels, each image is obtained by convolving the corresponding image in the immediately higher resolution level with a Gaussian kernel and subsampling by a factor of two. This operation was implemented in a very careful way, in order to guarantee the desired real–time feasibility. Instead of using some general convolution routine that works with arbitrary kernels, we implemented a hand–optimized function that convolves images with a specific 3×3 blurring kernel, corresponding to a bivariate Gaussian distribution with standard deviation equal to 0.8493 ± 0.0001, on both axes. The use of a single predefined kernel eliminates the need to keep its elements either in specially allocated registers or in memory, speeding up the critical inner loop of the convolution. The criterion used in the choice of the particular kernel shown in Fig. 16 is the fact that its elements are all powers of two, and thus it can be implemented with integer additions only. With a careful subexpression factorization, the resulting convolution and subsampling can be implemented with only 1.5 memory accesses with pointer increment, 2.25 integer additions, 0.25 pointer comparisons and 0.25 shift–right operations per element of the original image, on average.

The next step is the segmentation of the target in the image. In order to obtain some robustness with respect to variations in the illumination and in the background of the scene, we use a target composed of black circles printed on white paper and perform a histogram

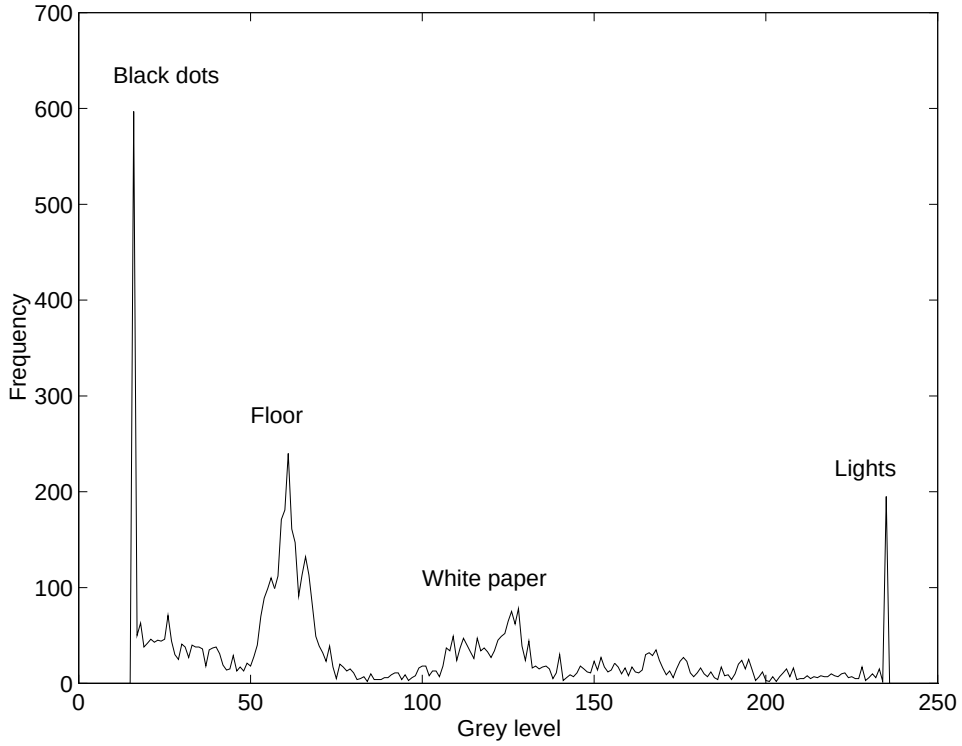

Figure 17: Grey level histogram in a typical scene with the target at about 10 feet from the camera.

analysis to determine an ideal threshold to binarize the monochromatic images grabbed by the Matrox Meteor in the trailing robot, so that the black dots can be told apart from the white paper. For efficiency purpose the grey–level frequency information needed to generate the histograms is gathered on–the–fly, during the subsampling process, at the extra cost of 1 memory accesses with pointer increment, per pixel.

A typical low–resolution grey–level histogram for an image grabbed in the environment used in the tests, with the chair relatively far away from the camera (about 10 feet), is shown in Fig. 17. It was verified that there is a satisfactory contrast between the black dots (corresponding to the darkest end of the histogram) and the white paper (corresponding to a peak of intermediate intensity), for a wide range of illumination conditions. Unfortunately, it was also verified that in general there are many different parts of the scene background that have roughly the same intensity than either one of these two parts of the target.

In order to determine the ideal thresholding point, we initially smooth the histogram, by convolving it with an unidimensional discrete low–pass (*average*) filter with a kernel of size eight. Then we scan the color histogram from the darkest to the lightest grey level, looking for the first valley that appears after a peak, in order to try to separate the black dots from anything else but a few black spots in the scene. A peak is defined as a grey level that has frequency strictly higher than the following eight levels in the histogram, at least, and a valley is defined as a grey level that is at least eight levels apart from the previous peak and has frequency strictly lower than the following eight levels, at least. Actually, for

improved efficiency, the smoothing is performed on–the–fly, during the scanning process, and it is interrupted as soon as the desired valley is found. This valley is then used as the binarization threshold.

The next step is to detect and label all the connected regions of low intensity (according to the selected threshold) in the image. This is done using the local blob coloring algorithm described in Ballard and Brown [4]. Again, the criterion that determined the selection of this technique was its efficiency, since it scans the entire input image just once, from the left to the right and from the top to the bottom.

Initially, this algorithm is used to detect all the dark regions in a level of low resolution in the pyramid. In this phase, in addition to labeling all the connected components in the image, we also compute, on–the–fly, their bounding boxes, centroids and masses. The dark regions detected in the image are compared against the appearances predicted for the target's black circles by the tracker that we describe in Section 8.3. For each predicted appearance (converted to the appropriate level of resolution), we initially label as matching candidates all the detected regions with similar mass and aspect ratio. Among these, the detected region whose centroid is closest to the position predicted by the tracker is selected as the final match for the corresponding circle in the target.

The selected bounding boxes are then converted to a level of high resolution, and the blob coloring algorithm is used on each resulting window, in order to refine the precision of the estimates for the centroid positions in the image. The resulting image positions are used as inputs to the tracker, that recovers the 3-D pose of the target, predicts how this pose will evolve over time, and then reprojects the 3-D predictions into the 2-D image plane, in order to calculate new predicted appearances for the black dots, which are used on the next step of the low level digital image processing.

8.5 Visual Control and Experimental Performance

In addition to tracking the leading robot in the field–of–view of the camera placed at the trailing robot, the problem of smart convoying also requires the motion of the trailing robot to be properly controlled, so that the target to be followed never disappears from its visual field (or alternatively, it is reacquired whenever it disappears). In our system, this control is based entirely on the 30 Hz error signal corresponding to the values recovered for t_x and t_z (θ is used only in the prediction of the appearance of the target on the next frame).

We use a simple PID controller, whose *proportional*, *integral* and *derivative* gains are developed empirically, in order to convert the t_x and t_z signals into steering angle and linear speed commands for the robot. The goal of this controller is to keep t_x equal to zero and t_z equal to a convenient predefined value (currently about 5 feet). Due to a limitation in the throughput of the steering–speed command interpreter in our mobile robots, the t_x signal is subsampled to a 6 Hz rate and the t_z signal is subsampled to a 3 Hz rate. The turning angle and linear speed signals are then discretized to a -10 to 10 scale and passed down as commands to the trailing robot, in such a way that the turn and speed commands are never issued concurrently, at a single instant in time.

This first attempt at a controller that makes maximum use of a specially-engineered tracking target performs rather well, in that it manages to keep the lead chair in view,

keeps distance constant, and can reliably track turns of 30 to 45 degrees without losing target features. The work is in early stages, however, and we have not performed extensive testing or evaluation. In fact, the issue of basic controller design for this task is potentially interesting. Our current controller assumes "off road" conditions: it is permissible always to head directly at the lead vehicle, thus not necessarily following its path. The error metric could simply be to keep the target centered in the field of view and correctly sized. If vehicles must stay "on road", then what is desired is that

$$\mathbf{s}_f^{(t)} = \mathbf{s}_l^{(t-d)}, \tag{16}$$

where $\mathbf{s}_f$ and $\mathbf{s}_l$ are the state vectors of the follower and leader, respectively, and d is some desired time delay. In words, the follower should re-trace the trajectory of the leader precisely. The criterion that the following vehicle maintain a constant distance from the lead vehicle and stay on its path can be disastrous since it could, for example, induce the follower to take a sharp corner too fast to maintain distance with a leader who speeds up once through the corner. Eq. (16) raises a number of interesting issues: state estimation of the leader's heading (local steering angle, say) as well as speed (or accelerations) are ultimately needed, to be duplicated for local control. Vision becomes harder since the follower cannot always aim itself at the leader. The desired trajectory is known, which turns the problem into one that can perhaps more usefully be related to optimal control than to simple feedback control.

9 Conclusions

Several improvements and additions to the wheelchairs have turned them into flexible research instruments. For instance, the improved odometry instrumentation allows a controller that overcomes the notorious directional inaccuracies when the chair is commanded to start moving in some direction when its three passive casters are in random orientations. We are able to close a control loop for a convoying task using non-trivial computer vision techniques. Our controller takes advantage of a special-purpose three-dimensional target for which pose-estimation algorithms can be specialized. A serial link protocol supports various abstractions of control, allowing various levels of open- and closed-loop paradigms. Our current controller makes "off-road" assumptions, *viz.* that there are no constraints on the follower's path. On-road following strategies are easy to state but will require some potentially interesting research.

10 Acknowledgements

Thanks to Helder Araujo, Dana Ballard, Josh Drake, Roger Gans, David Miller, Randal Nelson, Rajesh Rao, Garbis Salgian, Randy Sargent, Jim Vallino, Justin Vliestra, and Dave Koberstein and Paul Chinn of Proxim.

References

[1] M. A. Abidi and T. Chandra. A new efficient and direct solution for pose estimation using quadrangular targets: Algorithm and evaluation. *IEEE Trans. PAMI*, 17(5):534–538, 1995.

[2] T. D. Alter. 3-D pose from 3 points using weak–perspective. *IEEE Trans. PAMI*, 16(8):802–808, 1994.

[3] H. Araujo, R. L. Carceroni, and C. M. Brown. A fully projective formulation for Lowe's tracking algorithm. Technical Report 641, U. Rochester Comp. Sci. Dept., 1996.

[4] D. H. Ballard and C. M. Brown. *Computer Vision.* Prentice–Hall, 1982.

[5] W.E. Carlson, D.L. Stredney, R. Jackson, R. Batley, and S. Simon. The determination of environmental accessibility and wheelchair user proficiency through virtual simulation. *http://pft5xx36.ft90.upmc.edu/RTP/RERC_Restasks.html*, 1997.

[6] D. F. DeMenthon and L. S. Davis. Model–based object pose in 25 lines of code. *Int. J. of Comp. Vis.*, 15:123–141, 1995.

[7] M. Dhome, M. Richetin, J-T. Lapresté, and G. Rives. 3-D pose from 3 points using weak–perspective. *IEEE Trans. PAMI*, 11(12):1265–1278, 1989.

[8] M. A. Fischler and R. C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analisys and automated cartography. *Comm. ACM*, 24(6):381–395, 1981.

[9] D. B. Gennery. Visual tracking of known three–dimensional objects. *Int. J. Comp. Vis.*, 7(3):243–270, 1992.

[10] R. M. Haralick and C. Lee. Analysis and solutions of the three point perspective pose estimation problem. In *Proc. IEEE Conf. CVPR*, pages 592–598, 1991.

[11] J. Hartman and P. Creek. *IRIS Performer Programming Guide.* Silicon Graphics Computer Systems, 1994.

[12] R. Horaud, S. Christy, F. Dornaika, and B. Lamiroy. Object pose: Links between paraperspective and perspective. In *Proc. Int. Conf. Comp. Vis.*, pages 426–433, 1995.

[13] R. Horaud, B. Conio, O. Leboulleux, and B. Lacolle. An analytic solution for the perspective 4-point problem. *CVGIP*, 47:33–44, 1989.

[14] D. P. Huttenlocher and S. Ullman. Object recognition using alignment. In *Proc. Int. Conf. on Comp. Vis.*, pages 102–111, 1987.

[15] D. P. Huttenlocher and S. Ullman. Recognizing solid objects by alignment with an image. *Int. J. of Comp. Vis.*, 5(2):195–212, 1990.

[16] M. Ishii, S. Sakane, M. Kakikura, and Y. Mikami. A 3-D sensor system for teaching robot paths and environments. *Int. J. Rob. Res.*, 6(2):45–59, 1987.

[17] S. Linnainmaa, D. Harwood, and L. S. Davis. Pose determination of a three–dimensional object using triangle pairs. *IEEE Trans. PAMI*, 10(5):634–647, 1988.

[18] Y. Liu, S. Huang T, and O. D. Faugeras. Determination of camera location from 2-D to 3-D line and point correspondences. *IEEE Trans. PAMI*, 12(1):28–37, 1990.

[19] D. G. Lowe. Solving for the parameters of object models from image descriptions. In *Proc. ARPA IU Workshop*, pages 121–127, 1980.

[20] D. G. Lowe. Three–dimensional object recognition from single two–dimensional images. *Artif. Intell.*, 31(3):355–395, 1987.

[21] D. G. Lowe. Fitting parameterized three–dimensional models to images. *IEEE Trans. PAMI*, 13(5):441–450, 1991.

[22] C. B. Madsen. Viewpoint variation in the noise sensitivity of pose estimation. In *Proc. IEEE Conf. CVPR*, pages 41–46, 1996.

[23] N. Navab and O. Faugeras. Monocular pose determination from lines: Critical sets and maximum number of solutions. In *Proc. IEEE Conf. CVPR*, pages 254–260, 1993.

[24] D. Oberkampf, D. F. DeMenthon, and L. S. Davis. Iterative pose estimation using coplanar feature points. *Comp. Vis. Image Understanding*, 63(3):495–511, 1996.

[25] J. Ojala, K. Inoue, and M. Takano. Development of an intelligent wheelchair using computer graphics animation and simulation. *Computer Graphics*, 10:285–295, 1991.

[26] T. Q. Phong, R. Horaud, and P. D. Tao. Object pose from 2-D to 3-D point and line correspondences. *Int. J. Comp. Vis.*, 15:225–243, 1995.

[27] G. Salgian and D. Ballard. Developing autonomous naviagation algorithms using photorealistic simulation. *Submitted to IEEE Conference on Intelligent Transportation Systems*, 1997.

[28] T. Shakunaga and H. Kaneko. Perspective angle transform: Principle of shape from angles. *Int. J. Comp. Vis.*, 3:239–254, 1989.

[29] J.B. Shung, G. Stout, M. Tomizuka, and D.M. Auslander. Dynamic modeling of a wheelchair on a slope. *J. of Dynamic Systems, Measurement, and Control*, 105:101–106, 1983.

[30] J.B. Shung, M. Tomizuka, D.M. Auslander, and G. Stout. Feedback control and simulation of a wheelchair. *J. of Dynamic Systems, Measurement, and Control*, 105:96–100, 1983.

[31] C. Wiles and M. Brady. Ground plane motion camera models. In *Proc. European Conf. on Comp. Vis.*, volume 2, pages 238–247, 1996.

[32] A. D. Worrall, K. D. Baker, and G. D. Sullivan. Model based perspective inversion. *Image Vis. Comp.*, 7(1):17–23, 1989.

[33] Y. Wu, S. S. Iyengar, R. Jain, and S. Bose. A new generalized computational framework for finding object orientation using perspective trihedral angle constraint. *IEEE Trans. PAMI*, 16(10):961–975, 1994.

[34] J. S.-C. Yuan. A general photogrammetric method for determining object position and orientation. *IEEE Trans. Rob. Aut.*, 5(2):129–142, 1989.

A Code Fragments for Wheelchair Dynamic Simulation

Below a typical use in perfly.c

```
  {
          float sin_head, cos_head;

          PFCOPY_VEC3(ViewState->viewCoord.hpr, ViewState->prevCoord.hpr);
          PFCOPY_VEC3(ViewState->viewCoord.xyz, ViewState->prevCoord.xyz);
          your_favorite_car_dynamics(frame_time,
                                     0.0f * ViewState->SteeringRate,
                                     0.0f * ViewState->AccRate,
                                     1.0f * ViewState->BrakeRate);
          pfSinCos(ViewState->viewCoord.hpr[0] +90.0f, &sin_head, &cos_head);
          ViewState->viewCoord.xyz[0] -= 0.1 * cos_head;
          ViewState->viewCoord.xyz[1] -= 0.1 * sin_head;

          PFCOPY_VEC3(ViewState->prevCoord.hpr, ViewState->viewCoord.hpr);
          PFCOPY_VEC3(ViewState->prevCoord.xyz, ViewState->viewCoord.xyz);
          addHeadGaze();
        }
```

Where the car dynamix are doine like this:

```
#include "perfly.h"
#include "cardynamics.h"
#include "dynamics.h"

/***** local global variables ***************************/
struct dynparams dparam;
struct initial_conditions initconds;
static double eps=1.0e-4, *ystart, *last_y;
static double  t1,t2,dt, *dydx;
static matrix_t yp;
static double hmin;
/*****************************************************/


/***** global variables used by the bikedynamics.c *******/
float last_steering;
float steering;
float gas;
float brake;
```

```
float delta_seconds;
extern Boolean         STOPPED;
int badlast,nbad,nok;
/*******************************************************/

void
update_kart_init()
{
    yp = mat_new(BUFFER_LENGTH, STATE_VARS+LEAD_VARS+1) ;
    ystart = vector(1,NVARS);
    last_y = vector(1,NVARS);
    dydx = vector(1,NVARS);
    init_parms(INIT_FILE);   /* read and interpret the initialization file */
    init_wchr(ystart);  /* set car into initial state */
    init_wchr(last_y);  /* set car into initial state */
    t1 = 0.0;
    t2 = delta_seconds;
}

void your_favorite_car_dynamics(float frame_time,
                                float str,
                                float gs,
                                float brk)
{
  static int isInitial = 1;
  float delta_dist=0, steer_angle;
  static float distance=0;
  float sin_head, cos_head;

  steering = str;
  gas = gs;
  brake = brk;
  delta_seconds = frame_time;

  /*  printf("steering %f gas %f brake %f\n", steering, gas, brake); */


  if(isInitial)
  {
    update_kart_init();
    isInitial = 0; /* for one time initialization */
    toggleTexture();
  }

    rk2simple(ystart, NVARS, (double) t1, (double) t2, 1, whchr_derivs);
```

```
/*      printf("y1: %f, y2: %f, y3: %f, y4: %f, y5:%f, y6:%f, y7:%f\n", ystart[1],
           ystart[2], ystart[3], ystart[4],
           ystart[5], ystart[6], ystart[7]);

*/

  if (steering != 0)
  {
    delta_dist = (float)(sqrt(ystart[1]*ystart[1] +
                              ystart[2]*ystart[2])) - distance;
    if (delta_dist == 0.0)
      steer_angle = 0.0;
    else steer_angle = RAD2DEG(ystart[7]);

  }
  else
  {
    steer_angle = 0.0;
    delta_dist = (float)(sqrt(ystart[1]*ystart[1] +
                              ystart[2]*ystart[2])) - distance;
  }

  /* Change the car's viewstate */
 /* ViewState->viewCoord.xyz[1] += ABS(delta_dist);*/

  ViewState->viewCoord.hpr[0] += STOPPED == 0 ? steer_angle : 0.0f;
  pfSinCos(  ViewState->viewCoord.hpr[0] +90.0f, &sin_head, &cos_head);
  ViewState->viewCoord.xyz[0] += delta_dist * cos_head;
  ViewState->viewCoord.xyz[1] += delta_dist * sin_head;

  /* update time deltas */
  t1=t2;
  t2=t2+delta_seconds;

  /* update last_y */
  distance = (float)(sqrt(ystart[1]*ystart[1] + ystart[2]*ystart[2]));

  /* update the last steeing value */
  last_steering = steering;

}
```

And the tough stuff is here, in the dynamics file...

```
#include        <stdio.h>
#include        <math.h>
#include        <cbtypes.h>
#include        <cbmath.h>
#include        "dynamics.h"

/********* globals/defines used in derivs      *****************/
#define ABS(x) (((x) > 0) ? (x) : -(x))
#define PI 3.14159265359
#define GRAVITY 9.81
#define RHO 1.23
Boolean STOPPED; /* for when the car is stopped */
extern  float steering;              /* kart steering value */
extern  float last_steering;         /* last kart steering value */
extern  float gas;                   /* kart gas value */
extern  float brake;                 /* kart brake value */
double  e0;
double  dxsav,step_scale;
int     debug, debug1;


/* definitions to make the code shorter */
#define M dparam.M             /*vehicle mass*/
#define J dparam.J             /*body inertia*/
#define L dparam.L             /*wheel base*/
#define j dparam.j             /*wheel inertia*/
#define r dparam.r             /*wheel radius*/
#define A dparam.A             /*frontal area*/
#define CD dparam.CD           /*drag coefficient*/
#define nu1 dparam.nuTorque    /*linear vehicle friction*/
#define nu2 dparam.nuSteer  /*for steering function (unused)*/
#define nu dparam.friction    /*linear vehicle friction*/
#define tau dparam.qfriction /*for steering function (unused)*/
#define maxadot dparam.adot   /*steering rate*/
#define amax dparam.amax     /*maximum steering angle*/
#define VM dparam.VM           /*maximum params vehicle speed*/
#define fraction dparam.gas   /*braking on centripetal problems*/
#define brakes dparam.brake   /*maximum braking acceleration*/
#define kappa dparam.dmatch   /*maximum braking acceleration*/
#define motorV dparam.volts   /*maximum braking acceleration*/
#define motorR dparam.resistance   /*maximum braking acceleration*/
#define wref 4.0 /* max wheel speed for the wheelchair */
```

```
/*************************************************************
 * get_torque:
 *
 * torque = m * r * (accel - brake) = I * (accel - brake)/r
 *
 * Note: For stopping we need to apply negative torque.
 * Since most of us stop faster than we accelerate, there's
 * a "scale" term for stopping. When the car is stopped there
 * is no torque and the STOPPED variable is set so that
 * the steering wheel rate won't change. Also, the accel and
 * brake values are supplied by the go kart.
 *************************************************************/
/* basic torque routine: no backup, acceleration proportional to grad Vq */

double get_torque(double y[])
{
    double speed, torque = (VM*M * r * (gas - brake)), test;
    static double maxacc = -1, bsens = 2;

    speed = r*y[6]/cos(y[7]);


    if ((speed <= 0) && (torque <= 0.0))
    {
        STOPPED = TRUE;
        return(0.0);
    }

    /* initialize maximum allowable torque */
    if (maxacc == -1)
        maxacc=0.5*RHO*A*CD*r*VM*VM*
            (1.0+0.5*sin(speed*PI/VM)*sin(speed*PI/VM));

    if(maxacc<0.0)
        maxacc=0.0;

    /* Limiting conditions */
    if( torque > maxacc )
        torque = maxacc;

    if(torque < ( test=-r*M*brakes )) /* brakes helps us stop faster */
        torque = test * bsens;

    /* Cannot back up, need to stop continuously */
    if(torque < 0.0)
```

```
        torque *= H(speed,step_scale);

    /* Take your foot off the gas if the centripetal force is too large */

    if( (sin(y[7])*speed*speed/L) > (0.5*GRAVITY) )
        torque = fraction*test;

    if( (-sin(y[7])*speed*speed/L) < (-0.5*GRAVITY) )
        torque = fraction*test;

    STOPPED = FALSE;
    return torque;

}

/*************************************************************
 * get_steering:
 *
 * When the kart steering rate is given to the model directly
 * the y position values of the kart will oscillate. For some
 * this effect appears to go away when the range of the
 * steering values is restricted. Note that when the steering
 * value goes from negative to positive (or vice versa) we must
 * add the values to get the steering difference.
 *************************************************************/
double get_steering(double y[])
{
    double delta_steer, sensitivity=1.0;

    if (STOPPED) /* hold the wheel still when stopped */
        return(0.0);
    else if ((last_steering < 0) && (steering > 0))
        delta_steer = (double)(ABS(steering) + ABS(last_steering));
    else if((last_steering > 0) && (steering < 0))
        delta_steer = -1*(double)(ABS(last_steering) + ABS(steering));
    else
        delta_steer = (double)(steering - last_steering);

    /* A set of limiting conditions */
/*    if( delta_steer > maxadot )
        delta_steer = maxadot;

    if( delta_steer < -maxadot )
        delta_steer = -maxadot;
```

```
    if( ( y[7] >= amax ) && ( delta_steer > 0.0) )
        delta_steer = 0.0;
    else if ( ( y[7] <= -amax ) && ( delta_steer < 0.0) )
        delta_steer = 0.0;
*/
    return (delta_steer * sensitivity);

}



/* Might have to pass in real world position here */
void init_car(double y[])
{

        y[1]=initconds.x;          /*x position*/
        y[2]=initconds.y;          /*y position*/
        y[3]=initconds.q;          /*azimuth of pursuit vehicle*/
        y[4]=0.0;                  /*initial rear wheel orientation*/
        y[5]=0.0;                  /*initial front wheel orientation*/
        y[6]=initconds.ffdot;      /*front wheel rotation rate*/
        y[7]=initconds.a;          /*steering angle*/
        STOPPED = TRUE;            /* we are initially stopped */
}

void init_wchr(double y[])
{
        y[1]=initconds.x;          /*x position*/
        y[2]=initconds.y;          /*y position*/
        y[3]=initconds.q;          /*azimuth of pursuit vehicle*/
        y[4]=initconds.wl;         /* lft wheel rotation rate */
        y[5]=initconds.wr;         /* rt wheel rotation rate */
        y[6]=0.0;                  /* initial torque rate */
        y[7]=0.0;                  /* initial steering rate */
        STOPPED = TRUE;            /* we are initially stopped */
}

/*********************************************************
   motorTorque: 4/24/96 : C code for torque (motor friction deducted)
   code segment calculates ideal torque given speed and resistance
 *********************************************************/

double motorTorque(double wheeln, double resistance, double V)
{
  double i,T,n,gearing=1.0;
```

```
  n = gearing * wheeln;
  i = (V - 0.6*n) / resistance;
  T = 0.3*i/PI - 0.141;

  return T;
}


 void whchr_derivs(double t,double yin[], double dydx[])
{
  double drag, *fq, speed;
  int i;
  double x, y, wr, wl, q;
  double cq,sq;
  double vsq;
  double nurolling = 1.0, wroffset = 2.0;
  /* note: wroffset is because the right wheel goes faster
     than the left under normal conditions */

  /* initialization of variables */
  fq = vector(3,5);
  for (i=3; i<=5; i++) fq[i] = 0.0; /* zero out the torques */

  x=yin[1];
  y=yin[2];
  q=yin[3];
  wl=yin[4];
  wr=yin[5];
  speed = 0.5 * r * (wl+wr);
  vsq = speed * speed;
  cq=cos(q);
  sq=sin(q);

  /* 1st half of car dynamics */
  wl = get_torque(yin);
  wr = wl + wroffset  ;
  fq[4] = motorTorque(wl / (2.0*PI),motorR,motorV);
  fq[4] -= 0.5 * nurolling * M * GRAVITY * r * (wl/wref);
  fq[5] = motorTorque(wr / (2.0*PI),motorR,motorV);
  fq[5] -= 0.5 * nurolling * M * GRAVITY * r * (wr/wref);
  yin[6] = fq[4] + fq[5];

  yin[7] = get_steering(yin);
  fq[4] -= 0.5 * yin[7];
```

```
  fq[5] += 0.5 * yin[7];

  drag = 0.5 * CD * A * RHO * vsq; /* standard air drag term (unimportant for wheelchair

  /* the following C code comes from maple */
  dydx[1] = (wl/2 + wr/2) * r * cos(q);
  dydx[2] = (wl/2 + wr/2) * r * sin(q);
  dydx[3] = -r * (wl-wr);
  dydx[4] = -r/(M*r*r+2.0*j) * drag * r/(2.0*J*r*r+j) *
            fq[3] + (4.0*j+r*r*M+4.0*J*r*r)/(M*r*r+2.0*j) /
            (2.0*J*r*r+j) * fq[4]/2 + r*r*
            (-M+4.0*J)/(M*r*r+2.0*j) /(2.0*J*r*r+j) *
            fq[5]/2;
  dydx[5] = -r/(M*r*r+2.0*j) * drag + r/(2.0*J*r*r+j) * fq[3] +
           r*r*(-M + 4.0*J) / (M*r*r+2.0*j) / (2.0*J*r*r+j) *
           fq[4]/2 + (4.0*j + r*r*M +4.0*J*r*r) / (M*r*r+2.0*j) /
           (2.0*r*r*M + 2.0*J*r*r + j) * fq[5]/2;

}


 void car_derivs(double t,double yin[], double dydx[])
{
        double x,y,xdot,ydot,q,qdot,fr,frdot,ff,ffdot,a,adot;
        double  ca,sa,cq,sq;
        double t1,t2,rhs,inertia,vsq;
        double  FD,Tfriction;
        double  d=0.5*L;        /*location of center of mass*/


    x=yin[1];y=yin[2];q=yin[3];fr=yin[4];ff=yin[5];ffdot=yin[6];a=yin[7];
    ca=cos(a);sa=sin(a);cq=cos(q);sq=sin(q);
    qdot=r/L*sa*ffdot;frdot=ca*ffdot;vsq=pow(r*frdot,2)+pow(d*qdot,2);
    xdot=r*cq*frdot-d*sq*qdot;ydot=r*sq*frdot+d*cq*qdot;

/* 1st half of car dynamics */

    dydx[1] = xdot;
     dydx[2] = ydot;
    dydx[3] = qdot;
    dydx[4] = frdot;
    dydx[5] = ffdot;

    FD=0.5*RHO*A*CD;
```

```
    inertia=j;
    inertia+=(r/L)*(r/L)*sa*sa*(J+M*d*d);
    inertia+=ca*ca*(j+M*r*r);

/* << here it is!  change these two to ask the user and he can drive.
   also need to change delta t up in nuexecutive.c >>*/

/*
Note that one could call these routines, and maybe should, outside
this inner loop, up in nuexecutive.c (q.v.).
a) it might take a long time to convert inputs,
I don't know...if the numbers are living in some registers, then not.
b) they may not change very fast. */

    rhs=get_torque(yin);/* Engine driving torque */

    adot=get_steering(yin);/* Steering wheel rate */
if(debug)
        printf("\n in derivs, robx,y, rhs (torque), steer:\n %f %f %f %f", x,y, rhs, ado

    Tfriction=-dparam.friction*FD*r*r*VM*(1.0+ca*ca)*ffdot;
        /* 'viscous' friction*/
    t1=-FD*r*ca*r*r*(ca*ca+(d/L)*(d/L)*sa*sa)*ffdot*ffdot;
        /* quadratic drag terms*/
    t2=sa*ca*(j+M*r*r-(r/L)*(J+M*d*d))*ffdot*adot;
                        /*steering drags*/
        /*    rhs+=Tfr(t,yin,u,v)+Tfriction+t1+t2; */  /* adds torque twice.*/
     rhs +=Tfriction+t1+t2;
    dydx[6] = rhs/inertia;

    dydx[7] = adot;

    return;
}
```

B Data Sheets for Shaft Encoders and Electronics