Investigating robot navigation in health care with the Giraff telepresence robot

Ove Kåven

FYS-3921 Master's Thesis in Electrical Engineering

June 2013

Abstract

The Norwegian public healthcare system will not have the manpower to care for the elderly at the same level as now, unless technological solutions are found to make the most of the available manpower. This thesis investigates potential technologies for allowing the Giraff, a telepresence robot, to navigate and patrol an eldercare center autonomously, thus letting caregivers save time when checking on the care recipients. It describes the design and implementation of a platform to interface with the Giraff's hardware, and demonstrates that the developed system is a useful platform for developing such navigation systems.

Acknowledgements

I'd like to thank my advisors, Robert Jenssen (Department of Physics and Technology, University of Tromsø), Per Hasvold, and Stein Olav Skrøvseth (both at Norwegian Centre for Integrated Care and Telemedicine), for the help they've given me despite the difficulties of writing this thesis. Also thanks to Lars Ailo Bongo (Department of Computer Science, University of Tromsø), for the extra assistance. None of us had much prior experience in this particular field of research, but everyone contributed what they could.

Contents

1. Introduction

1.1. Motivation

As a consequence of the increasing life expectancy in Norway, the number of seniors needing care from the Norwegian public welfare system will continue to grow in the coming years. It is estimated that in 2025, 16% of the population (900 000 people) will be above 67 years old, and 250 000 will be above 80 years, while the number of healthy, young people available to provide that care will decrease correspondingly. To uphold the current standards for elderly care, the healthcare sector would have to recruit at least every 4th youth in the nation in order to satisfy the needs for 2025, and every 3rd in order to satisfy the needs for 2035. This is neither realistic nor desirable [1].

Instead, the healthcare system needs to use the manpower it has more efficiently. One way to do this is by developing new technology to assist caregivers. For example, it would be useful to be able to deploy robots at care centers that can be used to rounds and check in on the elderly, without needing a caregiver to always be physically present. Remote-controlled telepresence robots for such purposes already exist, allowing the caregiver to make rounds in multiple locations without leaving his/her own office, though these can be tedious to use, as their every move needs to be controlled manually. For seniors living in their own homes, there are also projects underway to provide robotic personal assistants [2, 3], though they are still under heavy development, and not yet ready for the public.

1.2. Objectives

For this thesis, I have explored the possibility of relieving caregivers further by automating the navigation of telepresence robots used at care centers. In particular, I have been working with the Giraff telepresence robot, currently being tested at a local care center (Kroken sykehjem), and exploring ways to make it navigate such a center without explicit assistance from the caregiver. This robot was chosen because it is already commercially available for a reasonable price (while still having sufficient capabilities for such use), and a unit was available for research use at the time of writing.

While other, more powerful robotic platforms exist, such as the Willow Garage PR2 (http://www.willowgarage.com/), they are expensive, still considered experimental, and usually not designed as telepresence robots. This may possibly make them useful assistants, but they're less useful for health personnel that wishes to talk to patients remotely. The Giraff may be better for this purpose.

The idea of automating the navigation of a telepresence robot is that the caregiver should be able to request a particular room with a single command, rather than manually commanding every step necessary to get there, thus allowing the caregiver to focus on more important tasks. Further down the line, the robot should also be able to do fully autonomous daily or nightly patrols, looking for anomalies, and only alerting the caregiver if it finds any.

Creating such a system is a large undertaking, requiring the use of algorithms and techniques that's still subject to much research. A complete navigation system for a particular robot would need at least these components:

- A system for getting input from available sensors (in this case, the camera).
- A system for controlling the actuators (in this case, the wheel motors).
- A system for extracting features (landmarks) from the sensor input.
- A system for estimating the robot's location using the observed landmarks, in combination with the current speed of the motors.
- A system for detecting obstacles using the sensor input.
- A system for choosing the robot's destination.
- A system for planning a route to the robot's desired destination.
- A system for following that route, while avoiding obstacles.

Other systems (such as remote control and teleconferencing) may also be desirable. The first two listed systems are the ones that are responsible for communicating with the particular robot's hardware. The remaining systems are of a more general nature and can implement any applicable technique found in the research literature, though for this particular robot, some will be more suitable than others.

Various implementations of these more general systems already exist. However, systems that allow them to communicate with the Giraff are not yet available, and the Giraff's standard controller software is proprietary and difficult to extend for this purpose. Thus, a completely new software stack is needed for supporting autonomous navigation on the Giraff. The first step is to communicate with the robot hardware.

1.3. Contributions

This thesis makes the following contributions:

- The design and implementation of a platform with components to control (and simulate) the Giraff's motors, and capture frames from its camera (Chapter 3, Appendix B, and on CD-ROM)
- An investigation of the techniques and technologies that can be used to implement the remaining systems (Chapter 2)
- Documentation of low-level details of the Giraff motor controller (Appendix A)

The developed platform functions as a framework with components to control (and simulate) the Giraff's motors, and capture frames from its camera. For evaluation purposes, I've also made a proof-of-concept of a feature extractor component.

A notable feature of the developed framework is that it can also record data for playback later, so that any components built on this platform can be prototyped and offline-tested without the actual Giraff. This is useful if such units are not permanently available to developers, and may speed up development and testing.

Detailed knowledge of the motor controller is needed for properly operating the controller, but these details were either not documented, or incorrectly documented, at the time of writing. Many of these details were learned through reverse engineering, and are now documented here.

1.4. Chapter list

This thesis is structured as follows:

Chapter 1: The introduction.

Chapter 2: Description of some of the techniques and technologies that may be used in the Giraff.

Chapter 3: Description of the actual platform developed for investigating the above technologies,

Chapter 4: Description of the Giraff robot itself.

Chapter 5: Evaluation of the usefulness of the developed platform.

Chapter 6: Conclusion.

Appendix A: Low-level documentation of the Giraff motor controller.

Appendix B: Listings of the source code of the developed platform.

Appendix C: Description of the contents of the CD-ROM.

2. Navigation technologies

This chapter describes some of the navigation techniques and technologies that may be used in the Giraff to make it more autonomous. To form a complete system, they may be combined as shown in Figure 2.1. The actual platform I've built to allow exploring such techniques is described in Chapter 3.

2.1. General

The Giraff is designed to operate in an indoor environment without significant obstacles. It does not need to function everywhere, and making the environment accommodate the robot is acceptable, if necessary. However, even if the environment is suitable for the robot, the robot will still need to become familiar with it, in order to successfully navigate it. That is, the robot needs to acquire an internal map of the environment that it can refer to when trying to figure out where it is and where it needs to go, and that map should reflect what the robot's sensors can see.

Ideally, the robot should be able to build the map itself, based on what it sees as it moves around. This problem, Simultaneous Localization And Mapping (SLAM), is a complex problem and still the subject of much research, but is difficult to avoid in this kind of setting. In principle, it might be possible to enter the building's blueprints into the robot instead, but this would be tedious, and such blueprints would probably not include obstacles such as furniture or people. Hence, the robot still needs to be able to analyze and map out its surroundings, in order to navigate safely.

2.2. Feature (landmark) extraction

The first step of any SLAM approach is to use sensors to identify and locate landmarks that can later be used to estimate the current position. The method used to identify landmarks should be as noise-resistant, unambiguous, and accurate as possible, yet not too computationally demanding, due to the finite power of the robot's onboard computer.

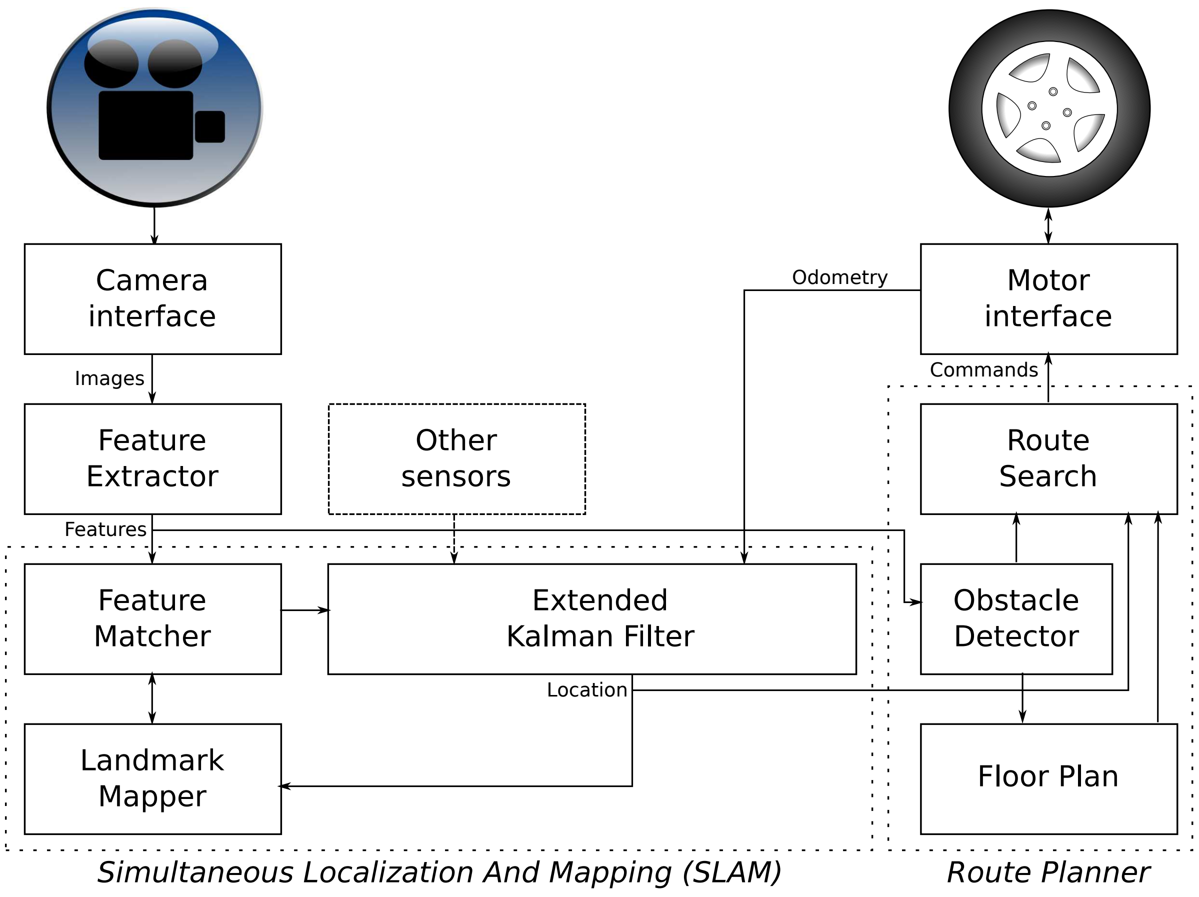


Figure 2.1.: A possible navigation system

Fortunately, it doesn't have to be perfect, as the occasional misidentified landmark can be rejected later by a good SLAM algorithm.

Some SLAM implementations are designed to function with a particular type of sensor, such as range finders, while others are more universal. Also, some SLAM implementation already include a feature extractor, and thus don't need a separate component to do this, but many don't.

2.2.1. Using a single camera

This is the only sensing approach which does not require augmenting the Giraff's hardware. However, for localization, it may also be one of the more computationally demanding approaches, since a single image from the camera is not enough to find the distance to an object. Instead, as the robot is moving, different images (from different positions) must be compared, and visual features matched. With enough data, the 3D position of the feature (and of the robot observing it) can be estimated within a reasonable margin of error. The feature then becomes part of the robot's «map» and can be used as a landmark later. The estimation of the 3D position is, however, usually left to the SLAM algorithm, not to the feature extractor. Thus, the chosen SLAM algorithm must be among those which can process monocular input.

Visible landmarks can be extracted from camera images using feature extractors of the type commonly used in computer vision. Typically, they attempt to find corners of objects, since their positions are relatively clearly defined and they can be tracked fairly reliably. Many corner detectors are available in the OpenCV library's «Image Processing» and «2D Features Framework» modules. Various implementations can also be found in other free and open source libraries, such as the CVD library (http://www.edwardrosten.com/cvd/cvd/html/index.html).

A common choice of feature extractor is the Harris corner detector [4], but using larger image patches may be more reliable in some cases [5]. There are more powerful extractors, such as the Scale-Invariant Features Transform (SIFT) [6]. However, because of the limited CPU power of the Giraff, I expect that it's better to stick with a conventional corner detector. One that offers a very good balance of speed and reliability and has gained some popularity recently is the FAST corner detector [7].

2.2.2. Using stereo cameras

If a robot is equipped with two cameras, separated by a fixed distance and a known angle, and with known calibration parameters, then stereo vision can be used to instantly find the distance to objects in view, much like humans do it. Since the relative orientations of the two cameras are always known exactly, depth information can be extracted more reliably and with less computation than with a single camera. This improves the accuracy and robustness of SLAM, and reduces the number of ambiguities [8].

2.2.3. Using radio beacons

A robot could estimate its position based on receiving radio signals from beacons installed at known positions in a building. A sensor that could receive such signals could be connected to one of the Giraff's USB ports. Unfortunately, GPS-style distance measurements are not practical with ordinary beacons, but as long as the directions to the beacons can be estimated by the sensor, the robot's position could still be triangulated using SLAM techniques [9].

2.2.4. Using ceiling landmarks

Another camera could be added to the robot that would be pointed directly upwards, tracking the ceiling. This could be quite usable for SLAM [10, 11], especially for rooms with ceiling lights. The extra camera could be connected to one of the Giraff's USB ports. (The Giraff's primary camera should probably not be used for this purpose, as it could then no longer see what's in front of it, which would defeat the purpose of patrolling.)

2.2.5. Using laser range finders

Unlike a regular camera, laser range finders can measure the distance to an object directly, and thus find the 3D position of any visible object with minimal computation. When cameras are used to locate interesting features and range finders are used to pinpoint their position, features can be tracked quite accurately [12].

For robot navigation, LIDARs (Light Detection and Ranging) are often used. A laser pulse is emitted, and a mirror deflects it in a particular direction. When the pulse hits an object, it is reflected back to the LIDAR, which measures the time between emission and reflection, and thus the distance to the object. By turning the mirror appropriately, the LIDAR can scan everything in front of it pixel by pixel, creating a depth image. Such a ranging module could be connected to one of the USB ports, and used for SLAM.

A cheap alternative is the Microsoft® Kinect™ sensor. An infrared laser illuminates the scene with random patterns, and the reflections are captured with an infrared camera. The sensor can use the reflected patterns to estimate distances [13].

2.2.6. Using ultrasound sonar

Like lasers, sonars can measure the distance to objects and find the 3D position of objects directly. Sonar modules could be connected to one of the USB ports, and used for SLAM [14, 15].

2.3. Position estimation

Once sensor data from the environment is available, it can be used to estimate the robot's position in various ways, depending on the type and quality of sensor data.

2.3.1. Dead reckoning

Pure dead reckoning is probably the simplest approach - just use the robot's odometry directly. This would probably be combined with a recorded route, which the robot would then follow every time, since it won't be aware of obstacles in the way. However, while this may work for small apartments, estimates from dead reckoning are prone to accumulating errors over time and distance. For navigating larger buildings, the position estimate would need to be regularly corrected using other position estimates, making dead reckoning unsuitable.

2.3.2. The Kalman filter

Most position estimation approaches use some variant of the Kalman filter [16] to combine odometry with position estimates calculated from the robot's sensors. The Kalman filter can be summarized as follows: Given a hidden multivariate time series (such as the true coordinates of a robot over time), with a known but noisy transition model, and an observable time series that is a linear transformation of the unknown time series plus noise (such as measurements from the robot's sensors), the Kalman filter is a statistically optimal way of combining the estimate of the previous hidden state (the previous position) with a new observation (sensor measurements) to produce an estimate of the current hidden state (the current position). It is a recursive estimator (it does not need to recalculate previous observations for every new observation), and thus quite suitable for real-time applications.

Note that the basic Kalman filter (KF) is only meaningful when state changes (position changes) can be expressed as a linear transition matrix. For physical systems, this is often not the case. However, the nonlinear transition model can be linearized by taking the Jacobian matrix, evaluate it based on the current state, and use this as the transition matrix [17]. While this only gives a first-order approximation, it often works quite well, provided the state doesn't change too much between updates. To compensate for the approximation error, some «stabilizing noise» should also be added to the covariance matrix after each update. This method is called the Extended Kalman Filter (EKF). Most of the SLAM papers referenced below use the EKF (but this doesn't preclude using more recent KF variants instead, such as the Unscented Kalman Filter [18]).

A complication arises from the possibility that the robot might crash into something and not move in the expected direction at all. In this case, the motor odometry would be completely wrong, but there is no direct way to model such failure conditions in a basic (or extended) Kalman filter. One way of handling this might be to maintain several Kalman filters (one for standard operation, and the others for failure conditions) and assume that the filter that gives the best predictions is more likely to be correct. That way, if the sensors report that the robot isn't moving, the robot can deduce that since the KF that models a crash matches the data best, there's a high probability that it has indeed crashed into something, and should initiate recovery procedures. However, since crashing into things is not meant to be part of standard operation procedure, simpler solution might be acceptable. For example, it might suffice to say that if the robot is supposed to be moving, but the sensors report less movement than some predefined threshold for some predefined time, then initiate emergency procedures.

For convenience, OpenCV's Video Analysis contains an implementation of the Kalman filter. By default, its KalmanFilter class implements only the basic Kalman filter, but by modifying the matrices it uses, it can also implement the extended Kalman filter. It could be used if a given SLAM implementation doesn't supply its own Kalman filter.

2.3.3. Visual odometry

Visual odometry improves on the dead reckoning approach by adding a second source of odometry, which may reduce the error of the position estimate. Features from successive images from the camera can be compared, and the apparent motion patterns, the «optical flow», can be estimated.

If the robot is moving forward, everything it sees will seem to move away from the center of the image. The speed at which things move may allow the robot to estimate how fast it is moving forward. However, ambiguities exist since this speed is dependent on how far away the objects are, which is initially unknown. Fortunately, with enough observations (and using the speed reported by the motor when necessary), these distances can be estimated, and a useful 3D model of what's in front of the robot can be computed, which can then be used to calculate the robot's velocity [19, 20].

Once visual odometry is available, it can be combined with the robot's regular odometry through the Kalman filter or similar. This could produce good results, but will not be as powerful or robust as a full SLAM approach, because once an object leaves the robot's field of view, the robot forgets about it. Without maintaining a map, the robot cannot use landmarks for more robust localization.

2.3.4. SLAM

SLAM techniques are based on building and updating an internal map of the environment, using statistical methods to minimize uncertainly. Once landmarks have been found, they must be checked against the robot's internal map. If they are thought to be new landmarks, they are added based on the current estimated position. If they are already known, their known position can be used to update the current position estimate. In most cases, both the landmark position and the current position is uncertain, so that both must be continually updated, and preferably as robustly as possible. The final estimate should be based on both the visible landmarks and the motor odometry, and if no known landmarks are in sight, the odometry might be the only available source of position information.

Because the system should ideally run in real time, the number of tracked features needs to be bounded. Since no feature is statistically independent of any other feature (their position estimates are all related through the error of the robot's estimate of its own position, at the very least), a big covariance matrix has to be maintained, and used for updating every tracked feature after every new measurement. Some scheme for keeping the covariance matrix manageable is required, or at least minimize the effort of updating it [21]. The number of tracked features can be reduced by throwing away unimportant features (e.g., features close enough to each other that it isn't useful to track all of them), but to be able to handle a large map, the map needs to be broken down into sections. Fortunately, it seems it is possible to maintain conditionally independent covariance matrices for each local map, if each local map is considered a node in a Bayesian network [22].

Many SLAM implementations can be found on OpenSLAM, http://www.openslam.org/. OpenSLAM is not a project in itself, but a hosting and portal site that allows SLAM researchers to publish their own open source SLAM implementations. Several interesting projects are listed here, e.g. the RobotVision project for single-camera SLAM [23]. However, many of the projects don't support Windows, and thus would not work on the Giraff. RobotVision is designed to be cross-platform, though, so it may work, though its authors have only tested it on Linux. Another option is to take some promising Matlab project, such as EKFMonoSLAM [24, 25], and convert it to C++ (probably with the help of some C++ matrix library, e.g. the TooN library also used by RobotVision, http://www.edwardrosten.com/cvd/toon.html).

Not all open source SLAM implementations of interest are listed on OpenSLAM, unfortunately. For example, the author of [5] (Prof. A. Davison) has created a SceneLib that implements many of the techniques described in his papers. (It appears to be a powerful single-camera SLAM implementation, but unfortunately, it is also only for Linux.)

2.3.5. Satellite navigation

Traditionally, GPS doesn't work indoors. However, given the recent surge in interest in indoor positioning by cell phones, chips are apparently now being developed that can combine signals from USA's GPS, Russia's GLONASS, China's Compass, and EU's Galileo, and thus possibly work indoors. (See http://www.computer.org/portal/web/computingno
location-and-navigation-technology-indoors) If such a chip is made available as a USB adapter, it could be installed in the Giraff's USB port to provide position estimates.

2.4. Obstacle detection

Obstacle detection needs to use the same sensor data that the position estimation does, just for a different purpose. The main challenge is is that detecting solid objects need more information than the sparse set of features typically tracked by SLAM. However, the extra information does not necessarily need to be explicitly tracked in detail, they just need to be detected when they are right in front of the robot. Then the robot just needs to know that there's something there, maybe add it to its floor plan, and find some way around it, or some other route to its destination. If the robot has some sort of range finder, obstacles are typically not too hard to detect. Otherwise, it may need to use pattern recognition or maybe optical flow to detect whether it's dangerously close to something.

2.5. Destination selection

Typically, the destination is selected by the user, either interactively, or by preprogramming some patrol route. Selection a destination results in a set of target coordinates being given to the route planner.

2.6. Route planning

Once the robot knows where it is and where to go, it must figure out how to get there. Since there may be walls and other obstacles in the way, this has challenges of its own.

Some of the planning approaches that might be possible to implement on the Giraff are:

2.6.1. Recorded route

This is probably the simplest approach. A human can train the robot by manually steering it where it needs to go. The robot remembers the route, and replays the recorded actions of the human whenever the robot needs to. If the robot can have multiple destinations, the robot could remember waypoints and the routes between certain pairs of then. Then finding a route to somewhere distant becomes a standard graph search problem, with each edge in the graph being a recorded route. (Even a cost heuristic is available, since the waypoint coordinates are known and the Euclidean distance between them can easily be calculated. Thus, an A* graph search could be used if there was any chance that the number of known routes would be too large for a standard graph search to handle effectively.)

An obvious problem with this approach is that if obstacles (including people) move into the robot's path, the robot won't know how to avoid them.

2.6.2. Providing a floor plan

A floor plan of the building could be given to the robot, naming each room and the available doorways between them. Internally, the robot would store this floor plan in graph form, with each node in the graph being the name and coordinates of a room, and each edge being doorways and their coordinates. When the robot is asked to go to a particular room, it can use an A* graph search to find which doors it has to go through to get there. Between the doors, the robot may try to go the shortest route, but must try to avoid obstacles along the way using other algorithms (see below).

2.6.3. Teaching a floor plan

The robot could be steered by a human (or even being instructed to try to follow a human) between rooms. In each room, the robot would be told the name of the room the robot is in. The robot may then associate that name with its current position, and try to get there again whenever it is instructed to go to that room again. It can use some obstacle detection method to find walls and other obstacles, and use the resulting map to plan routes. This map can be represented using either vectors or bitmaps (where bitmaps make for the simplest path planning algorithms, but usually needs more memory).

2.6.4. Obstacle avoidance

In the event the robot was instructed to go to a particular destination unassisted, and it is trying to find the shortest path while mapping obstacles along the way, then the robot should probably use the D* graph search instead of the A* graph search to plan the route, to minimize time wasted replanning the route whenever an obstacle is detected [26]. In order to apply D* search, each room could be internally represented as a bitmap (grid), where each pixel (grid square) is «colored» according to whether it is thought to contain an obstacle, thought to be traversable, or not yet explored. This grid is updated as the robot moves around, and D* used to replan the route after each update.

2.7. Getting there

Once a route has been decided upon, the robot's motors need to be told where to go. This may, on its own, involve some algorithms and maths, since the Giraff's motors have ramp-up and ramp-down times that may need to be taken into account. Turning while moving has some interesting mathematical properties (the curves the robot follow are apparently clothoid segments [27]), the parameters of which need to be computed before sending the command to the motors.

3. The developed system

3.1. General

The developed system has four main modules:

- GiraffNav, the main program and user interface. It starts and controls the other systems, and handles user input.
- DisplayWindow, which displays the current camera image (and other information) on the screen. It allows monitoring, measuring, and debugging of the other systems.
- GiraffCamera, which can capture, record, and play back video. The video frames captured here can be used for localization and mapping.
- GiraffMotor, which can give motor commands, and capture, record, play back, and simulate their responses. The route planner can send its command here for execution.

The system is meant as a platform for the development of other navigation modules, as shown in Figure 3.1. Thus, for testing and evaluation purposes, there's also a fifth module, FeatureExtract, which demonstrates a feature extractor.

3.2. The GiraffMotor module

The GiraffMotor module's primary function is to accept commands for the Giraff's various motors and controls, and transmit them to the Giraff's AVR microcontroller for execution (or, if not running on a real Giraff, simulate them). It also regularly reads back odometry from the microcontroller, which the navigation modules can use to determine the robot's movement. For testing and evaluation, a dead-reckoning position estimate is computed from this odometry.

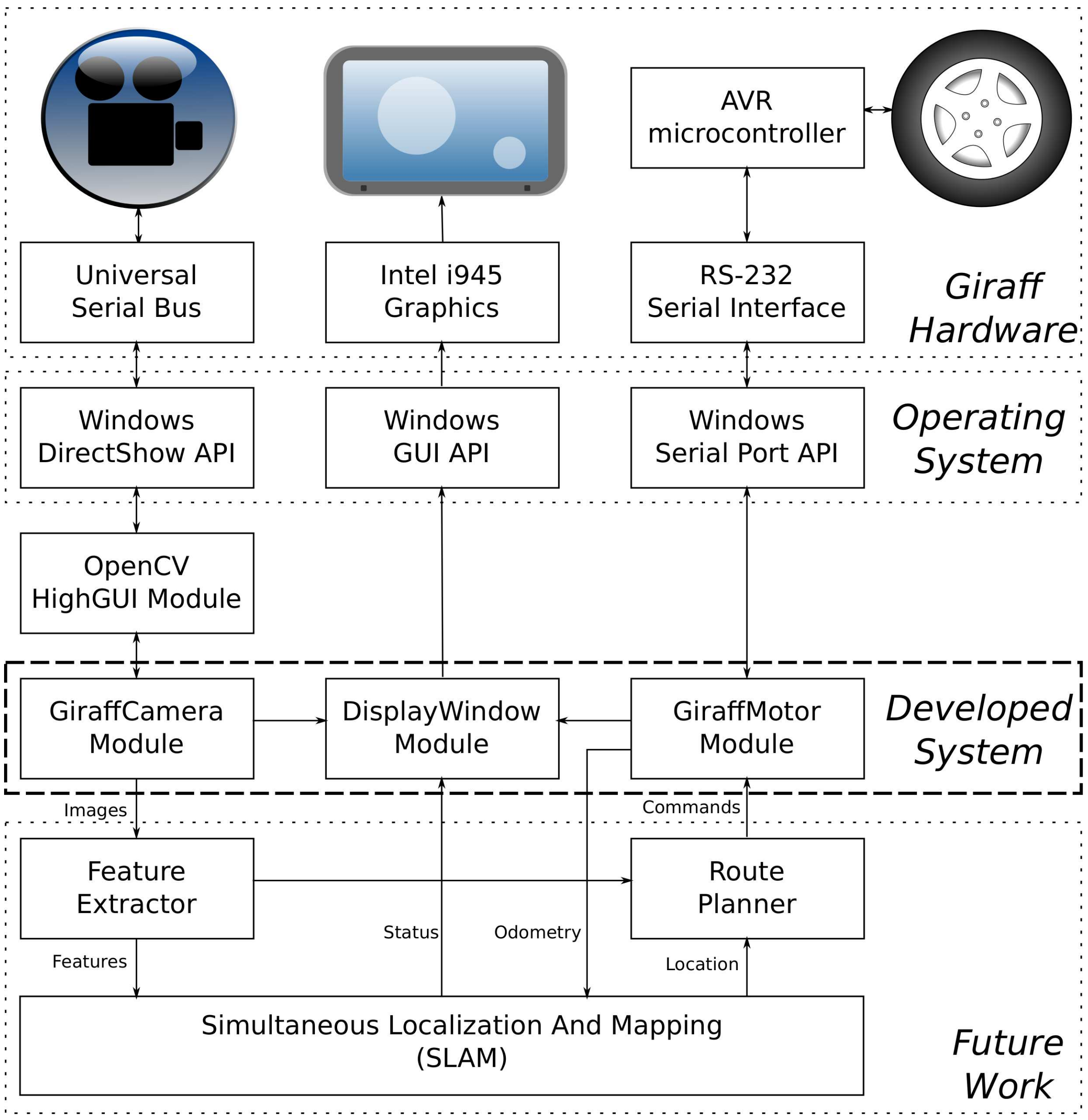


Figure 3.1.: Big-picture view of system (GiraffNav module not shown)

The GiraffMotor module contains two separate motor-related subsystems, one controller (the GiraffMotor class) and one simulator (the GiraffMotorSim class). On startup, the GiraffMotor class will try to connect to the microcontroller board, which is wired to the main computer's primary serial port (called «COM1» in Windows). If the microcontroller is not found, the system will fall back to using the simulator, allowing various features to be tested without the actual Giraff. This can be useful for checking whether navigation commands make sense before risking trying them on the real Giraff, but more importantly, it allows much of the system to be developed without always having access to the Giraff (as its limited availability was a major issue during this project).

3.2.1. The Motor Controller

The GiraffMotor class handles all communication with the microcontroller (real or simulated). If a real microcontroller is present, GiraffMotor powers it up and opens a communication link, sends commands, and receives responses.

3.2.1.1. Recording

When recording, all commands sent to the microcontroller (or simulator), and their responses, are saved to a text file, prefixed by the time elapsed since the start of recording. When playing back a recording, these commands and their responses are interpreted as if they were sent. The recorded commands are not sent to the microcontroller or simulator, but the recorded responses are interpreted as normal motor odometry, and used to estimate the current position. The recorded time is used to ensure that the recording is played back at the same speed as it was recorded at. (This also affects video playback, since the camera and motor systems run in the same thread. In order to stay synchronized, flags in the motor record files are used to mark when to allow a new frame to be loaded from recorded video.)

3.2.1.2. Handling user movement commands

The commands that GiraffMotor is allowed to send to the microcontroller is listed in Appendix A. These commands are designed for moving specific distances and stopping at specific points. However, since the system hasn't implemented autonomous navigation yet, currently the Giraff is primarily moved by pressing the arrow keys on the keyboard, and in this case it is not known beforehand how far the user wants the Giraff to move. To handle this, the Giraff's ability to preempt previous commands is used. When a key

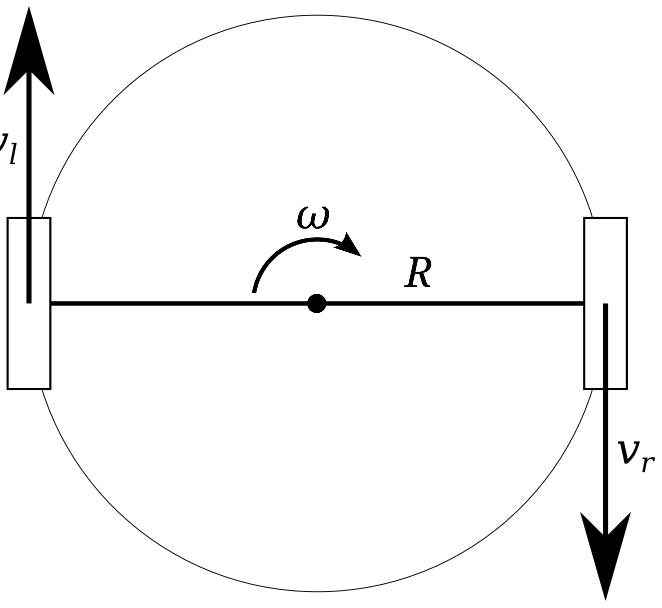


Figure 3.2.: Kinematics of turning

is pressed, the movement command given specifies some distance ahead of the current position (specifically, the full-speed-to-zero deceleration distance is multiplied by the AHEAD_FACTOR defined at the top of GiraffMotor.cpp, and the result is used as the movement distance). As long as a key is held down, new movement commands are issued periodically (specifically, whenever the distance left of the previous movement command is less than twice the deceleration distance). When a key is released, a final movement command is issued, requesting the minimum distance needed to decelerate from the current speed, plus a 10ms «reaction time» margin (i.e., the distance that would be traveled if the current speed was maintained for 10ms), to account for the time it takes to transmit the command to the microcontroller, and other potential delays.

3.2.1.3. Calculating turns

When setting up and tracking turns, some of the calculations require converting between wheel speed and angular speed. To find the conversion factor, refer to Figure 3.2, which shows rotating in place. According to material provided by Giraff Technologies, the distance between the two drive wheels is 499mm. Thus, the radius of the circle followed by the wheels is $R = 499\text{mm}/2 = 249.5\text{mm}$. To convert from wheel speed to angular speed in radians, note that $\omega = v_l/R = v_r/R$. To convert to degrees, multiply with a factor $360/2\pi = 180/\pi$. The final factor, $180/(R \cdot \pi)$, is in the source code denoted the TURN_FACTOR. The same factor also applies when converting between wheel distance and angular distance.

Actually, this factor also applies to curved motion, not just rotating in place. Given a frame of reference that follows the center of the Giraff (the middle dot in Figure 3.2), then at any given instant, the wheels can be thought of as moving the same way around this center as in the rotating-in-place case. It only remains to find the wheel speed in this frame of reference. From the formulas described in the Appendix (if correct),

Left Wheel Velocity = Overall Velocity * (1+vg)
Right Wheel Velocity = Overall Velocity * (1-vg)

Denote the overall velocity v and the virtual gear ratio g. Then it is apparent that, after canceling out the overall velocity, $v_l = v_r = vg$. Hence, if the current speed v and the current gear ratio g are both known, simply multiply them to get the wheel speed. Then use TURN_FACTOR to convert to angular speed in degrees. (Or, if a particular angular speed is desired, simply divide by the overall speed and TURN_FACTOR to get the desired gear ratio.) This can then be used as input for a location estimation algorithm.

3.2.1.4. Curved motion issues

Curved motion is the most challenging kind of motion to get right. Not only because of the computations involved, but also because of quirks and bugs in the motor controller.

The current speed of the wheels can be read from the motor controller as the «gvr» parameter. However, according to the manufacturer, this parameter does not give the overall velocity, but the velocity of the left wheel. Moreover, testing seems to show that this velocity is not computed using the formulas above, but using the incorrect formulas found in the documentation, i.e. Left Wheel Velocity = Overall Velocity * (1/(1-vg)). Hence, to find the overall velocity, you must compensate for this by multiplying gvr with (1-vg). From there, you can then find the actual wheel velocities if needed.

Even this kind of compensation wouldn't be possible if vg=1, since this would result in a division by zero, which probably results in the Giraff returning infinity for «gvr» (though I haven't tested this). The simplest way to avoid this singularity is to just never let the virtual gear ratio be as high as 1. (In the current system, it should only get to 0.51, bugs in the motor controller notwithstanding.) But if vg ever becomes 1 anyway, the code will, just in case, attempt to fall back to estimating the current speed by dividing distance travelled by time elapsed since the last odometry update.

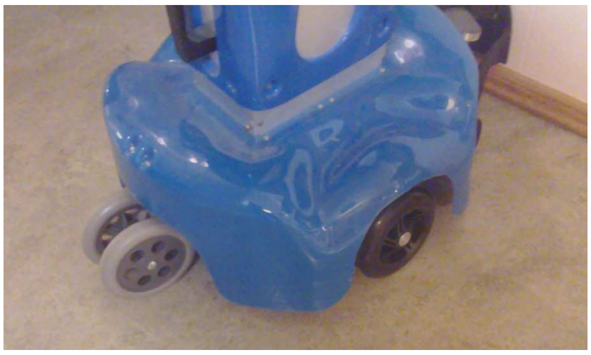


Figure 3.3.: Bottom of chassis. rear swivel caster, and right drive wheel.

According to the manufacturer, it's likely that a future version of the Giraff's software will change «gvr»'s behaviour so that it reports overall speed directly. Once this happens, the system may need to be recompiled to remove the compensation factor. (This can be done by commenting out the GVR_IS_LEFT definition at the top of GiraffMotor.cpp.)

Another problem, which I have not found a way to compensate for, is the way that the «cdp» parameter works, which is supposed to tell the controller when to start decreasing the virtual gear ratio back towards zero. In practice, it's not very useful, as the ramp-down profile used in practice is based on the distance left, not on the value of «cdp». In the end, I could only find two ways to exit curved motion: either come to a full stop, or force «vgr» to zero, thus converting the ramp-down into a flat, horizontal line. This has the effect of making the virtual gear ratio instantly zero, which causes a noticeable jerk. However, since this behaviour, unfortunate as it is, is at least predictable and makes it possible to move the Giraff around with the keyboard without too much trouble, I decided to use this method until Giraff Technologies addresses the problem. Also, some future autonomous navigation solution (that doesn't rely on input from the keyboard) might be able to plan its moves in such a way that it could avoid this issue.

3.2.1.5. Position estimation issues

Even if the odometry from the motor controller were perfect, the motor controller only knows about the two drive wheels on the sides of the Giraff. There are also two swivel casters (undriven wheels), one in front and one in back, as seen in Figures 3.3 and 4.2. When the Giraff turns or moves, these casters must turn to follow, and since the Giraff needs to move some distance before they've fully aligned themselves, they have a significant effect on how the Giraff travels. Worst case, if you turn in place for a bit, and then try to start moving forward, these casters may cause the Giraff to turn up to about

45 degrees extra before they've finally reoriented themselves. This effect isn't known to the motor controller, so for dead reckoning to be accurate, a model of the casters and their effect on movement may need to be devised and implemented. Fortunately, the problem can be mitigated by making sure to never turn in place, and only allow the Giraff to turn while also moving forward (assuming curved motion works satisfactorily). Assuming the direction estimate is also corrected using the camera, this issue might then even be something that could be neglected, though experimentation is the only way to make sure.

Position estimation also gets computationally tricky when moving in an arc, either due to explicit curved motion, or due to the effect of the casters. The motion profile need to be calculated, and integration techniques be used to determine what the new position would be. However, given that the resulting position would not be correct even if I implemented this (because of the casters and other issues), in my system I've only approximated it. I find the mean speed and the mean turning rate since the last update, and use this to calculate a first-order approximation of the new position. It is expected that a future localization system would use the camera image to correct this estimate anyway.

3.2.1.6. Implementation details

For communication through the serial port, the GiraffMotor class uses standard Windows API routines. After the serial port device is opened, SetCommState is used to set the important parameters (115200bps, 8 data bits, no parity). Since the microcontroller uses a line-based protocol, commands and responses do not have a fixed size. To handle this, SetCommTimeouts is used to set the read timeouts to zero (so that ReadFile always immediately returns whatever has been received, if anything), and the aforementioned SetCommState is also used to set the event character to the end-of-line character. That way, WaitCommEvent can be used to wait for the end-of-line character, then ReadFile can be used to read the complete line received. Some extra buffering logic (base on the C++ string class) is used for cases where ReadFile happens to read more than one line.

Because no timeout is applied to WaitCommEvent, there's a chance that this technique may cause the system to hang indefinitely if the microcontroller doesn't work, but this has not been an issue. (It could be addressed by opening the serial port device in overlapped I/O mode, which is less convenient to program, but would allow WaitCommEvent to be cancelled in response to some timeout or user action.)

When the GiraffMotor class needs to measure time, it uses the high-precision timers known in Windows as performance counters. These are typically hardware clocks built into CPUs or motherboards. In Windows, QueryPerformanceCounter can be used to read out the number of ticks since some arbitrary starting time (typically the time the computer was booted up). QueryPerformanceFrequency tells you how many ticks are per second. Thus, taking the difference between two QueryPerformanceCounter readings, and dividing it with the QueryPerformanceFrequency result, gives you the number of seconds between the two readings, with accuracy on the order of microseconds or nanoseconds. Timing information is currently only really needed for recording and playing back motor data, however.

3.2.2. The Motor Simulator

The GiraffMotorSim class attempts to simulate what the real microcontroller is supposed to do, i.e., it attempts to conform closely to the behaviour described in Appendix A, though only for features actually needed by GiraffMotor. Only the motor odometry that the microcontroller would report is computed, not the Giraff's resulting position in space. But since the simulator does not control anything physical, its simulated motions are far more precise than the real Giraff's motion would be.

Curved motion is implemented as documented in Appendix A, though the real controller may behave differently. For example, setting «vg» to zero causes the simulator to simulate a straight line motion (without turning) no matter what «vgr» is, but this does not seem to be the case for the real controller. However, I still implemented the simulator the way things are documented to work, rather than how they actually seem to work, in case such deviations are just bugs that will be fixed by the manufacturer at some point. (Also, for some of these deviations, it's just not clear what's going on in the real controller, and it would take too much time to figure out.)

3.2.2.1. Implementation details

For timing, GiraffMotorSim uses the same QueryPerformanceCounter technique that GiraffMotor uses, except when simulating the transmission delay that would occur when sending and receiving data strings through the serial port. This is done by calling Sleep, which only has a millisecond resolution (and often waits longer than requested).

When the simulator is asked to start a motion, the motion profile (times, distances) is calculated, using the kinematic equations of motion where needed. Care is taken to handle various corner cases, including speed changes and direction reversal (handled as a ramp-up from negative velocity to the target velocity). The calculated profile, along with the times for transitions (changes in acceleration) are stored in the class. Then, every time a new command/request is received from GiraffMotor, the current time is compared with the stored times, and new motor state and odometry is calculated, ready to be reported back to GiraffMotor when needed. Straight-line motion profiles and rotate-in-place motion profiles are kept separate (in retrospect, this would not have been necessary, though it did make the design slightly cleaner).

3.3. The GiraffNav module

This is the main module, responsible for starting, running, and shutting down the system. It measures the system's performance, and also interprets keyboard input from the user. The velocities used when the arrow keys are used are defined in this module (KBD_TURN_SPEED and KBD_MOVE_SPEED).

When the user starts a recording, this module constructs the file names based on the current system time, then passes the request on the GiraffCamera and GiraffMotor modules. When the user requests playback, this module also chooses the files to play back. Currently, the file name is specified in the source code (the PLAY_PATH and PLAY_FILE definitions) and compiled in, it cannot be changed at runtime, though adding a file selector for this could be a useful feature to add at some point.

The default camera resolution is also chosen here (the DEF_WIDTH and DEF_HEIGHT definitions). By default, an 800x600 camera resolution is set, because the camera appears to not always work if higher resolutions are used. With 800x600, the camera appears to be able to deliver about 10 frames per second.

3.3.1. Implementation details

Currently, the system is mostly single-threaded (though if video recording is enabled, video encoding is done in a separate thread). The primary reason for running the camera and motor in the same thread is to get a reliable association between a camera image and the corresponding motor odometry. As soon as a new image is retrieved from the camera (typically every 100 ms or so), new motor odometry is almost immediately retrieved from the Giraff's controller (this typically only takes a couple of milliseconds). Because the time to transfer images from the camera to the main computer through USB is probably much longer than the camera's exposure time, I expect this odometry to most closely match the next frame rather than the previous one, but I have not investigated this further.

3.4. The GiraffCamera module

The GiraffCamera module's primary function is to communicate with the Giraff's camera, and capture video frames in a way that is useful for navigation.

The camera can be accessed like a regular USB webcam (for example, through Video For Windows or DirectShow). In the implemented GiraffCamera class, OpenCV's HighGUI module is used. Its capture interface works as a convenient wrapper for DirectShow. In addition to camera access, HighGUI also provides video decoding and encoding (by using the open source FFmpeg library, which is included in the OpenCV distribution), which the GiraffCamera class can use to record and play back video. For recording video, I chose to use the DivX (i.e., MPEG-4 Part 2) format, as testing showed it to have decent encoding performance, in addition to good compression.

3.4.1. Implementation details

When recording, video encoding is done in a separate thread (synchronized using standard Windows primitives, like event and semaphore objects), so that things like retrieving motor odometry don't need to wait for encoding. A buffering system is also added to try to reduce lag spikes when saving large amounts of data (I used a USB flash drive, and these don't always have constant write speeds), though this didn't completely eliminate such problems. (It's possible the GiraffMotor module would need to do something similar in order to reduce these problem further.)

If no camera is connected, GiraffCamera can fall back to playing back a predefined video (and endlessly repeating it), which allows the system to be tested on a computer without a camera. This is the TEST_INPUT definition at the top of GiraffCamera.cpp, and I've just used one of the OpenCV sample videos.

The image grabbed from the camera (or played back from video) is returned to the main program as an OpenCV matrix.

3.5. The DisplayWindow module

The DisplayWindow module displays information from the other modules on the Giraff's LCD monitor, so that the system can be monitored, measured, and debugged.

3.5.1. Implementation details

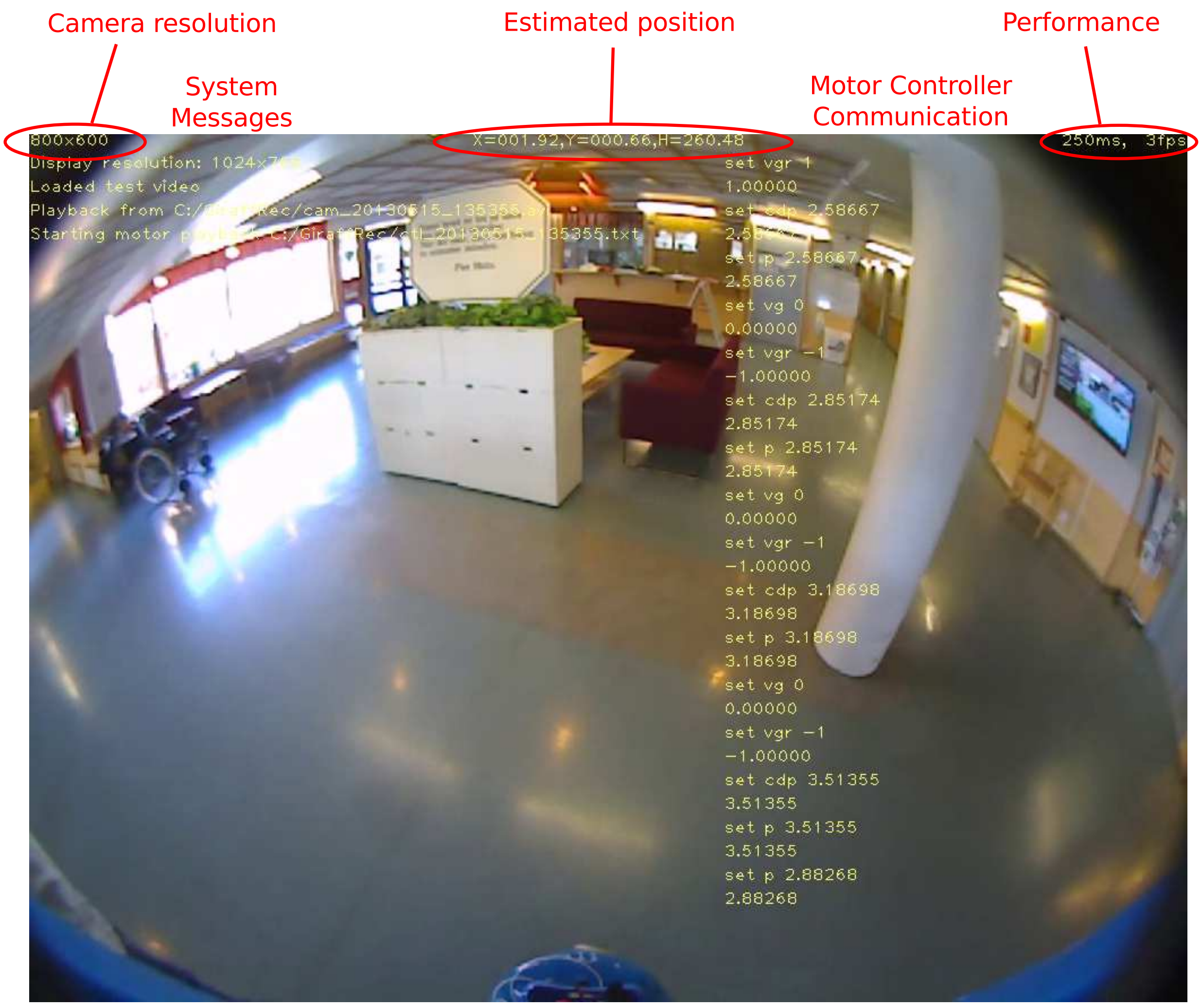
The user interface display is implemented using a combination of the standard Windows API and OpenCV.

When the main program calls the DisplayWindow's Start() method, a fullscreen window is created using the standard Windows API. Since the Windows API is a C interface, and DisplayWindow is a C++ class, usual techniques for bridging the C and C++ interfaces are used, including storing the C++ instance pointer into the window structure (using APIs such as SetWindowLongPtr). The standard Windows message loop is implemented in the ProcessInput() method.

Using the SetInputHandler() method, the main program can provide a callback for processing user input. When Windows calls the window procedure with a keyboard message, the message is sent on to the input callback, allowing the main program (the GiraffNav module) to process it.

Other modules can also call the DisplayWindow's SetCameraInfo, SetPositionInfo, SetPerformanceInfo, PrintLeft, and PrintRight methods when they have information to show to the user. The DisplayWindow class then stores these strings internally. PrintLeft and PrintRight implement a scrolling buffer by using a C++ «deque» container type, and limiting its size by deleting the topmost strings when its size exceeds a predefined threshold (the BUFFER_SIZE definition at the top of DisplayWindow.cpp).

Most of the real work happens when DisplayWindow's Show() method is called to show a camera image. The image is provided as an OpenCV matrix. This image is copied and resized to fit the display using OpenCV's resize function, and then any stored information (from SetCameraInfo etc) is rendered on top of this using OpenCV's putText function. Using OpenCV is faster than using equivalent Windows functions. Windows functions are only needed for showing the finished image. This is done by wrapping the image data in a Windows Device-Independent Bitmap (DIB) and blitting it onto the fullscreen window using SetDIBitsToDevice. (Alternatively, DirectDraw could perhaps be used for a theoretically more efficient display solution, but given that the display update only happens a few times per second, any improvements would probably be marginal.)



```
800x600                              X=001.92,Y=000.66,H=260.48                     250ms,  3fps
Display resolution: 1024x768
Loaded test video
Playback from C:/GiraffRec/cam_20130515_135355.avi
Starting motor playback C:/GiraffRec/ctl_20130515_135355.txt
```

```
set vgr 1
1.00000
set cdp 2.58667
2.58667
set p 2.58667
2.58667
set vg 0
0.00000
set vgr −1
−1.00000
set cdp 2.85174
2.85174
set p 2.85174
2.85174
set vg 0
0.00000
set vgr −1
−1.00000
set cdp 3.18698
3.18698
set p 3.18698
3.18698
set vg 0
0.00000
set vgr −1
−1.00000
set cdp 3.51355
3.51355
set p 3.51355
3.51355
set p 2.88268
2.88268
```

Figure 3.4.: Screenshot of a playback on a regular laptop, with UI elements marked

3.6. The FeatureExtract module

This module is a proof-of-concept to show how features could be extracted from images captured by GiraffCamera. It currently uses the FAST corner detector [7]. For visualization of the detected corners, it renders pink circles around them on the camera image shown by DisplayWindow. See Section 5.2.

3.7. The User Interface

The view provided by DisplayWindow has several parts, as shown in Figure 3.4. The current camera image is in the background, scaled to fit the screen. On the top left, the current camera resolution is shown. The top right is for keeping track of the system's performance. Currently, it shows the rate at which camera images (frames) are processed (milliseconds per frame, and frames per second). The top center is for displaying the current estimated position. Currently, it shows a dead reckoning estimate (and typically not a very accurate one since, while the motors are modeled, the effect of the casters (front and back swiveled wheels) are not).

On the left is a scrolling text area that can be used to show system state. Currently it mostly shows whether recording or playback is active, and what file is being recorded to or played from. On the right is a scrolling text area that shows communication with the motor controller.

To interact with the system, the following keyboard commands are available.

Key	Action
Escape	Exits program
Left/Right Arrow	Makes the Giraff turn as long as the keys are held down
Up/Down Arrow	Makes the Giraff move as long as the keys are held down
Numbers (1 to 5)	Tries to change camera resolution
Enter	Allows typing in your own commands for the motor controller
A	Toggles automatic retrieval of motor odometry
B	Sends a «get bulk_data» command (shows motor state)
H	Sends a «home» command (starts head homing sequence)
P	Toggles playback
R	Toggles recording
T	Tilts head to vertical position
U	Sends an «undock» command (backs and turns 180 degrees)

3.8. Software used

This section describes the software used in the developed system.

3.8.1. Development environment

The system is written using C++. As a fully compiled language, this gives better performance and needs less memory than interpreted languages like Python or Matlab. On an embedded system like the Giraff's onboard computer, making the most of the available resources is often important.

As the base development environment, I chose to use MinGW (www.mingw.org) with the MSYS option. MinGW is based on the open-source and cross-platform GNU Compiler Collection (GCC). Since most open-source navigation software is written using GCC (and usually on Linux), it seemed that using GCC for this project might make it easier to get such navigation software working later on. For the IDE (Integrated Development Environment), I chose to use Code::Blocks (www.codeblocks.org), but this isn't important, as editors and IDEs are just a matter of taste.

3.8.2. OpenCV

OpenCV (Open Source Computer Vision Library), at http://www.opencv.org/, is an extensive library of computer vision and machine learning algorithms. It implements both classic and state-of-the-art algorithms, all highly optimized and easy to use. It is released under the BSD license, making it free for all. Some of the modules of interest are:

- OpenCV's HighGUI library provides easy to use routines for creating GUIs and capturing images from cameras. This library is used for accessing the Giraff's camera.
- OpenCV's Image Processing library provides a host of image processing and analysis routines. Of particular interest here are the feature extractors.
- OpenCV's Video Analysis library provides routines for motion analysis. Among other things, it has routines to calculate optical flow, and even an implementation of the Kalman filter.
- OpenCV's 3D Reconstruction library provides routies to calibrate cameras, compare stereo images, and calculate projections and backprojections. It could be used to compensate for the fisheye effect of the wide angle lens.
- OpenCV's 2D Features Framework library provides more advanced feature extractors and pattern matchers.
- OpenCV's Object Detection and Machine Learning libraries provides many advanced machine learning algorithms.

Several books have been written about OpenCV [28, 29, 30]. This library is the backbone of many interesting projects, and so I chose it for this project as well.

4. The Giraff

4.1. Introduction

The Giraff is a mobile telepresence robot developed by Giraff Technologies AB, Sweden (http://www.giraff.org/). It is designed to be remote controlled by caregivers, allowing them to check up on care recipients without physically being there. Caregivers may use their own computers to connect to any recipient's Giraff robot, move it around using their computer's mouse, and see its environment and talk to people through the robot.

The Giraff is already involved in several other research projects. The unit I've had access to is operated by NST (Norwegian Centre for Integrated Care and Telemedicine) and primarily involved in the EU's VictoryaHome project, a project for putting robots in the homes of care recipients to act as proxies for human caregivers when they're not present, automatically alerting them whenever needed. For information about the project, see, for example, http://www.itfunk.org/docs/prosjekter/AAL-VictoryaHome.htm. Some more information about how the Giraff, in particular, is used in this project is available at http://www.robotdalen.se/en/News/Press-releases/2013/Giraff-key-player-in-new-EU-project-VictoryaHome-/

It is hoped that the Giraff can be used to fill roles such as

- Provide social interaction opportunities for people who live isolated or that don't get out of their houses much for health reasons, such as old age, COPD, or disability. Caregivers, family, and friends can simply log on to their computers to talk, without having to drive there. For caregivers, this saves valuable time and allows them to efficiently care for more people, which may live all over a wide area. Although this can't completely replace the human touch, and personal visits will still be important from time to time, this can supplement them and greatly increase the effectiveness of resource-starved health care departments, as the need for health care continues to grow faster than the resources to provide them.

- Allow physicians to check up on patients under their care that aren't in their

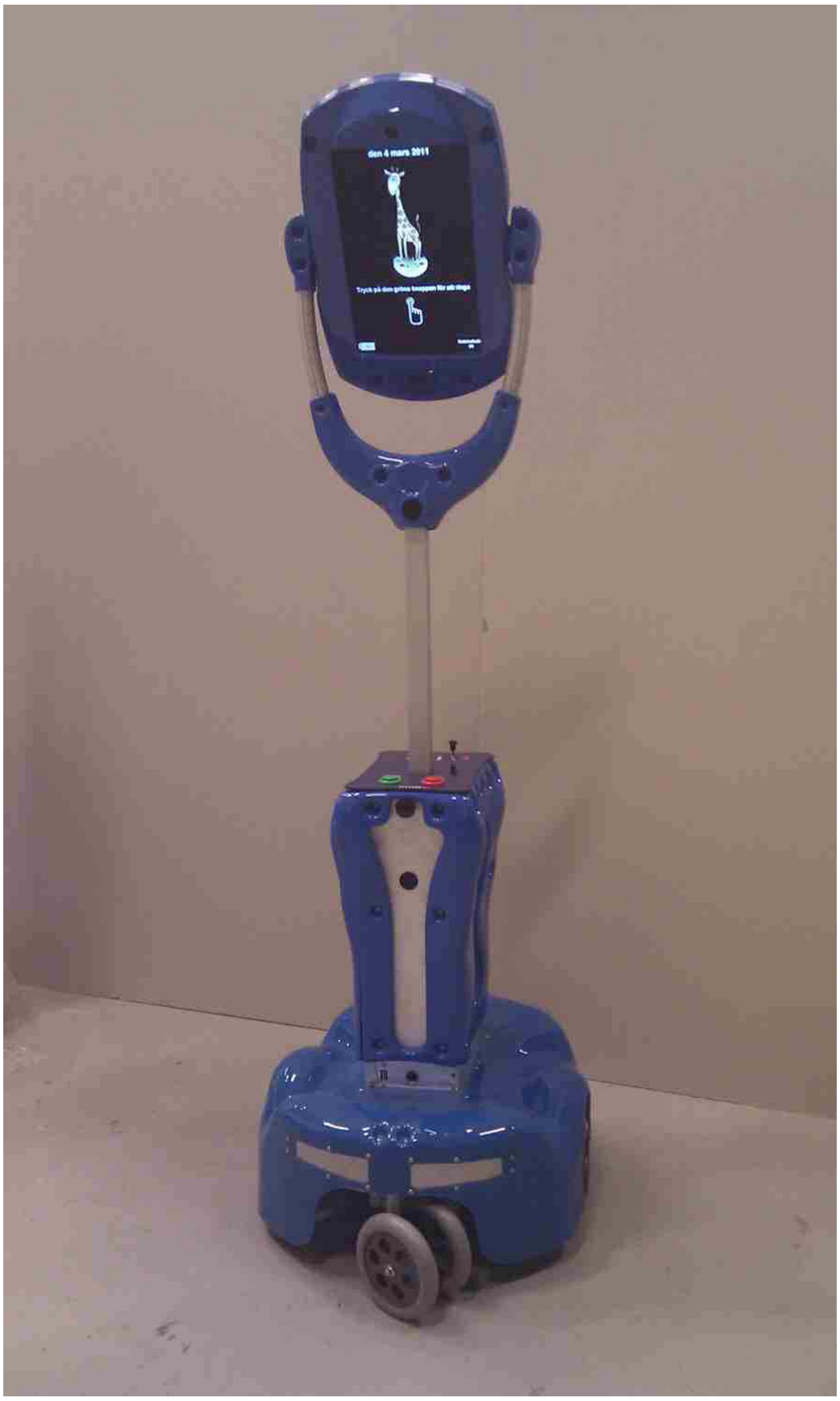


Figure 4.1.: Photo of the Giraff (from material provided by Giraff Technologies)

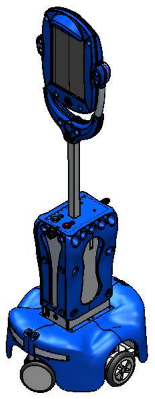


Figure 4.2.: Drawing of the Giraff (from material provided by Giraff Technologies)

> hospitals, such as in elder care centers. To supplement the regular visits to the care centers, the physician may use the robot to talk to people and solve simple problems without needing to drive there every time.

Unfortunately, the Giraff's standard software provides little automation and can be tedious to use, because every movement it can do needs to be explicitly commanded. It is hoped that adding more automation and autonomy to the Giraff can make its use simpler, allowing the users to focus more on the tasks they want to accomplish, and less on the fine details of steering the Giraff around. It might even help save lives if it could autonomously respond to persons in distress and report the situation to emergency personnel.

4.2. Design

As can be seen in Figures 4.1 and 4.2, the Giraff has a base unit, a long neck, and a head. The base unit houses a computer, control buttons, and motors for the 4 wheels. The head is connected to a tiltable panel with a monitor and a camera. The total height of the Giraff is a little over 1.6m. When a caregiver is communicating with another person through the robot, this allows comfortable interaction. The tiltable panel allows the caregiver to look up or down as needed. The control buttons on the chassis allow the care recipient to call the caregiver, accept and disconnect calls, and adjust the volume level. These functions are also available through a remote control. When the robot is not in use, it stays in its docking station, facing the wall.

4.3. Computer Specifications

The exact specifications of the Giraff's main computer were not available, but by accessing the operating system's Control Panel, it was possible to extract the following relevant information.

CPU	Intel Core 2 T7200, 2 GHz
GPU	Intel i945 Express
RAM	1 GB
Storage Type	Patriot Memory USB device
Storage Capacity	Primary partition 3.5GB (1.5GB free)
Operating System	Microsoft Windows Embedded Standard

The Giraff also has two USB ports. The rear port is meant to hold a wireless network adapter, and the front port can be used for connecting input devices like keyboards and mice, when necessary for administration [27].

These specifications suggest that the Giraff might be powerful enough to allow reasonably advanced applications to run on the device itself. A sufficiently efficient navigation application could run on it directly; remote-control solutions may not be necessary. This would be an advantage, as a remote-control solution for autonomous navigation would require more hardware and be less robust.

4.4. Camera

According to Giraff Technologies, the sensor chip is a Cynove USB device with a listed sensor size of 1/3.2" and a resolution of 1600x1200. It is fitted with a 1.8mm wide angle lens. For digital image sensors, the listed sensor size is usually about 1.5 times the actual sensor size, so the actual diagonal of the sensor would be about 5.68mm. Thus, the diagonal field of view is approximately $2 \arctan \frac{d}{2f} = 2 \arctan \frac{5.68\text{mm}}{2 \cdot 1.8\text{mm}} \approx 115°$. The horizontal field of view is approximately $2 \arctan \frac{4.54\text{mm}}{2 \cdot 1.8\text{mm}} \approx 103°$.

Because of the camera's wide angle, it would seem like a good idea to capture video at high resolution, in order to detect relatively distant landmarks with reasonable accuracy, though this may need to be balanced with the slower transfer speed and higher computational workload of a higher resolution. Testing suggests that the highest video resolution the camera is able to deliver at a practical rate is 800x600, at about 10 frames per second.

When using the camera for navigation, it is necessary to correct for the distortion (fisheye effect) caused by the lens. An advantage of the wide angle is that the robot can more easily keep landmarks and obstacles in view while turning and moving.

4.5. Motor Controller

The Giraff's wheels are controlled by an AVR microcontroller running custom software. It communicates with the main computer through a RS232-type serial port interface, using a line-based ASCII protocol [31]. The controller accepts operations like moving a specific distance, turning a specific angle, or a combination of both (curved motion). In buffered mode, up to four such operations can be placed in queue. All operations have ramp-up and ramp-down times, so that jerky motions cannot happen. The microcontroller also controls the tilt of the Giraff's head, and gives access to the buttons on the chassis.

With some caveats, the microcontroller can help estimating the robot's position by keeping track of the distance travelled by its drive wheels. The controller can provide this information to the main computer on request. For navigational purposes, this is usually known as odometry, and can be used for dead reckoning, which is necessary when no other position estimate is available (i.e., no known landmarks are in sight). However, testing shows that this is, unfortunately, not reliable enough to be used on its own.

Another issue is that the Giraff's default remote control software gain exclusive control over communication with the motor controller while it is running. Thus, the default software would need to be shut down before other navigation software can control the motor, or some way of multiplexing the motor controller port needs to be developed. One way to do this may be to create a virtual motor controller port that both pieces of software can connect to. Then the software behind the virtual controller communicates with the real controller, and routes commands and responses to whichever piece of software needs it. Another option might be to make a new navigation system a fully functional substitute for the default software, so that running the default software will just never be necessary.

5. Evaluation

5.1. Functionality

The implemented system works as described, and can be used to steer the Giraff through the care center, and record the journey for later playback, if a keyboard is connected. The images in Figures 3.4, 5.1 and 5.2 are from such a recorded journey.

5.2. Extensibility

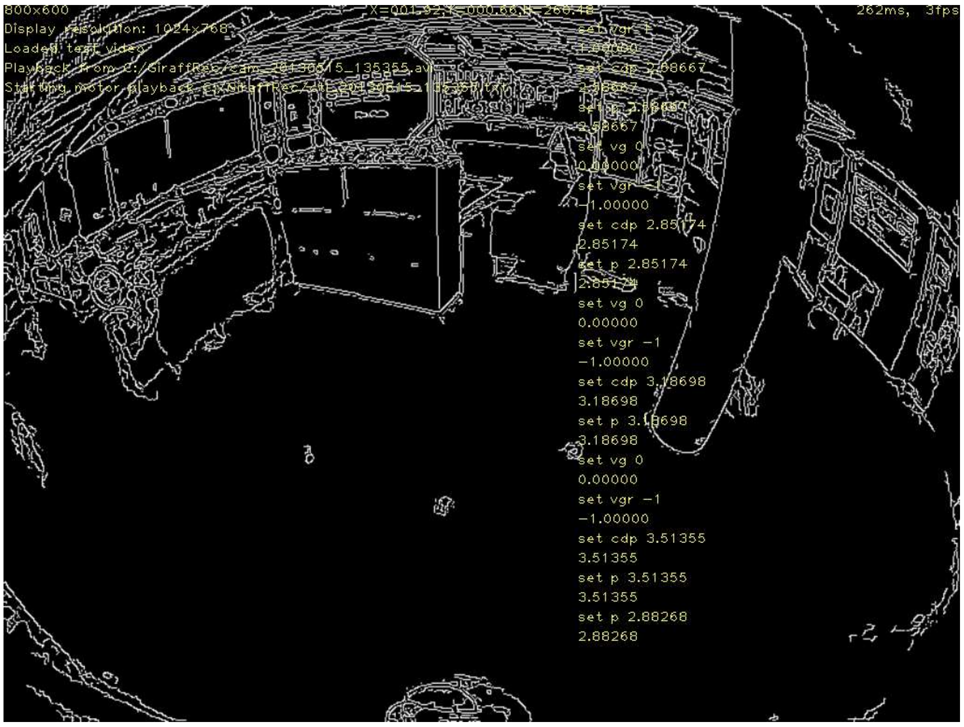
Requirement: As the developed system is meant to be a platform on as which a larger system could be built, it should be possible to implement other components on top of it.

Figure 5.1 shows the results of adding an image processing algorithm (the Canny edge detector [32], available in OpenCV), as an example of how such algorithms can be added. (Also, the ability to detect moving edges might be useful for obstacle avoidance.) Figure 5.2 demonstrates a particular type of feature extractor (the FAST corner detector [7], also available in OpenCV) that may be used as part of a navigation system. The features shown in the figure (pink circles) could be matched with previously known features, and their coordinates given to a SLAM implementation, which could then use them to determine the robot's current position.

5.3. Recording and playback

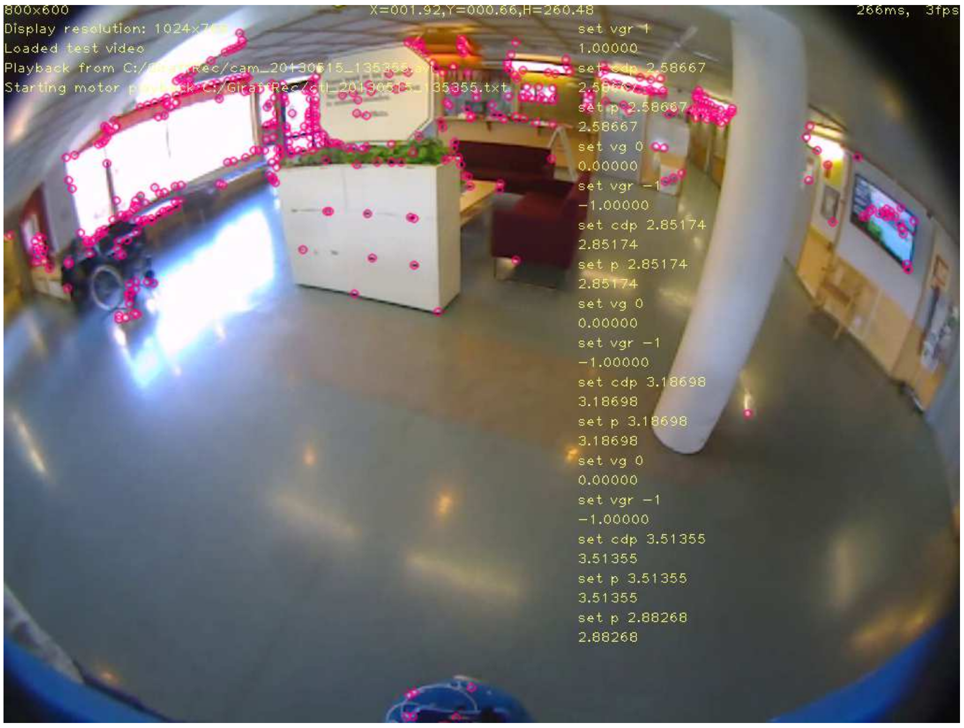
Requirement: When playing back a recording, the resulting visuals and motor odometry should be identical to what was seen when the recording was first created.

Some sample records are available on the attached CD-ROM. While testing shows that they do appear to be the same, there are still some lag spikes while recording, meaning that the Giraff does not work fast enough to do a smooth recording. Adding multithreading to the motor recording component might mitigate this. But since the



```
800x600                     X=001.92,Y=000.66,H=260.48                262ms,  3fps
Display resolution: 1024x768                    set vgr 1
Loaded test video                               1.00000
Playback from C:/GiraffRec/cam_20130515_135355.avi  set cdp 2.58667
Starting motor playback C:/GiraffRec/ctl_20130515_135355.txt  2.58667
                                                set p 2.58667
                                                2.58667
                                                set vg 0
                                                0.00000
                                                set vgr -1
                                                -1.00000
                                                set cdp 2.85174
                                                2.85174
                                                set p 2.85174
                                                2.85174
                                                set vg 0
                                                0.00000
                                                set vgr -1
                                                -1.00000
                                                set cdp 3.18698
                                                3.18698
                                                set p 3.18698
                                                3.18698
                                                set vg 0
                                                0.00000
                                                set vgr -1
                                                -1.00000
                                                set cdp 3.51355
                                                3.51355
                                                set p 3.51355
                                                3.51355
                                                set p 2.88268
                                                2.88268
```

Figure 5.1.: Playback with Canny edge detector



```
800x600                     X=001.92,Y=000.66,H=260.48                266ms,  3fps
Display resolution: 1024x768                    set vgr 1
Loaded test video                               1.00000
Playback from C:/GiraffRec/cam_20130515_135355.avi  set cdp 2.58667
Starting motor playback C:/GiraffRec/ctl_20130515_135355.txt  2.58667
                                                set p 2.58667
                                                2.58667
                                                set vg 0
                                                0.00000
                                                set vgr -1
                                                -1.00000
                                                set cdp 2.85174
                                                2.85174
                                                set p 2.85174
                                                2.85174
                                                set vg 0
                                                0.00000
                                                set vgr -1
                                                -1.00000
                                                set cdp 3.18698
                                                3.18698
                                                set p 3.18698
                                                3.18698
                                                set vg 0
                                                0.00000
                                                set vgr -1
                                                -1.00000
                                                set cdp 3.51355
                                                3.51355
                                                set p 3.51355
                                                3.51355
                                                set p 2.88268
                                                2.88268
```

Figure 5.2.: Playback with FAST corner detector (corners highlighted with pink)

captured video frames have timestamps in the motor record, this problem does not cause any drift in the timing of the playback.

5.4. Motor control

Requirement: Movement commands from the user should be properly interpreted and cause the Giraff to move in the desired way.

Commands can currently only be given using a connected keyboard, but this should suffice for evaluation. The sample records, available on the CD-ROM, shows that moving the Giraff around this way works. However, because of the problems with curved motion described in Section 3.2.1.4, turns are somewhat difficult to predict, and some movement jerks often happen when ending them. Possibly a future navigation solution would be able to plan moves in advance in such a way that these jerks can be avoided.

5.5. Motor simulation

Requirement: The motor simulator should emulate the actual motor controller as faithfully as possible.

Testing shows that the simulator is close to the real thing, with a few caveats. Unlike the real thing, the simulator does not make errors. For example, for mechanical reasons, the real motor controller is usually not able to hit the exact distance requested. If you request a certain distance, it will usually report an odometry that's off by a few millimeters. The simulator, however, will always report the exact requested distance in its odometry. Also, there are certain bugs in the real controller that's not replicated faithfully in the simulator, such as the quirky behaviour of the «Clothoid Deceleration Point» command used for curved motion. And some minor features, such as changing the head tilt angle, reporting presses of the chassis buttons, and checking the battery status, are not simulated at all. These are fairly minor issues, however, and the simulator works fine for its intended purpose of simulating the result of navigation commands.

5.6. Discussion

Clearly, many more things could have been explored or implemented in this project. In particular, it would have been very interesting to try an actual SLAM implementation on the Giraff. Unfortunately, because of the Giraff's limited availability, and the motor control took much more time than expected, in part because the original documentation was missing some vital information. However, using the platform described in this thesis, and the documentation provided in Appendix A, I believe implementing and evaluation navigation algorithms on the Giraff can now be done more efficiently.

In retrospect, it might have been a good idea to prioritize differently. For example, spending less time on tuning the motor controller and simulator would mean more time for trying out navigation algorithms, and for describing what has been done. It would also have been interesting to set up a few experiments, such as trying to do a simple pre-programmed patrol using dead reckoning. Since dead reckoning is unreliable, especially given the effects of the casters, the Giraff would probably not go exactly where it should, but it would be a good demonstration of the functionality of the motor controller.

6. Conclusion

For this thesis, I've built a platform for developing navigation solutions for the Giraff, a telepresence robot. I've designed and implemented a system to interface with its hardware, and also investigated many of the challenges involved in making it able to navigate a large building without human assistance, including localization and route planning. I examined some of the algorithms and technologies that could be used to solve those problems – some that require adding more sensors to the Giraff, and some that don't.

The implemented platform shows camera images and motor odometry on the screen, and allows the user use the keyboard to control the Giraff's motors and move it around. It can record and play back video and motor data, and when run on a regular computer, it can simulate the Giraff's motors. This allows offline development and evaluation of localization and navigation solutions, facilitating future work on the Giraff.

Based on a literature study of localization approaches, it appears that adding extra sensors may allow more robust algorithms to be used, but given the controlled environment the Giraff is meant to operate in, adding sensors is by no means necessary. A single-camera SLAM approach could work quite well. In particular, it might be interesting to try converting the EKFMonoSLAM source code (found on http://www.openslam.org/) from Matlab to C++ for use on the Giraff. Since this approach also allows the Giraff to be used unmodified, which is cheaper and more convenient for the users, this seems like the preferred approach. If we were to add a sensor, however, an infrared laser range finder would probably be most useful, in order to minimize the risk of crashing into things.

Bibliography

[1] Kunnskapsdepartementet, “Meld. St. 13 (2011-2012): Utdanning for velferd,” 17 Feb. 2012.

[2] S. S. Srinivasa, D. Ferguson, C. J. Helfrich, D. Berenson, A. Collet, R. Diankov, G. Gallagher, G. Hollinger, J. Kuffner, and M. V. Weghe, “HERB: a home exploring robotic butler,” *Autonomous Robots*, vol. 28, pp. 5–20, Jan. 2010.

[3] K. Yamazaki, R. Ueda, S. Nozawa, M. Kojima, K. Okada, K. Matsumoto, M. Ishikawa, I. Shimoyama, and M. Inaba, “Home-assistant robot for an aging society,” *Proceedings of the IEEE*, vol. 100, no. 8, pp. 2429–2441, 2012.

[4] M. L. Benmessaoud, A. Lamrani, K. Nemra, and A. Souici, “Single-camera EKF-vSLAM,” *Proceedings of World Academy of Science: Engineering & Technology*, vol. 42, pp. 924 – 929, June 2008.

[5] A. Davison, “Real-time simultaneous localisation and mapping with a single camera,” in *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pp. 1403 –1410 vol.2, Oct. 2003.

[6] A. Ali and M. Nordin, “Sift based monocular slam with multi-clouds features for indoor navigation,” in *TENCON 2010 - 2010 IEEE Region 10 Conference*, pp. 2326 –2331, Nov. 2010.

[7] E. Rosten and T. Drummond, “Machine learning for high-speed corner detection,” in *In European Conference on Computer Vision*, pp. 430–443, 2006.

[8] L.-F. Gao, Y.-X. Gai, and S. Fu, “Simultaneous localization and mapping for autonomous mobile robots using binocular stereo vision system,” in *Mechatronics and Automation, 2007. ICMA 2007. International Conference on*, pp. 326 –330, Aug. 2007.

[9] X. Kuai, K. Yang, S. Fu, R. Zheng, and G. Yang, "Simultaneous localization and mapping (SLAM) for indoor autonomous mobile robot navigation in wireless sensor networks," in *Networking, Sensing and Control (ICNSC), 2010 International Conference on*, pp. 128 –132, Apr. 2010.

[10] W. Jeong and K. M. Lee, "CV-SLAM: a new ceiling vision-based SLAM technique," in *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pp. 3195 – 3200, Aug. 2005.

[11] C.-J. Wu and W.-H. Tsai, "Location estimation for indoor autonomous vehicle navigation by omni-directional vision using circular landmarks on ceilings," *Robotics and Autonomous Systems*, vol. 57, pp. 546 – 555, May 2009.

[12] S. Fu, H. ying Liu, L.-F. Gao, and Y.-X. Gai, "Slam for mobile robots using laser range finder and monocular vision," in *Mechatronics and Machine Vision in Practice, 2007. M2VIP 2007. 14th International Conference on*, pp. 91 –96, Dec. 2007.

[13] Z. Zalevsky, A. Shpunt, A. Maizels, and J. Garcia, "Method and system for object reconstruction." Patent WO2007043036, Apr. 2007.

[14] T. Yap and C. Shelton, "SLAM in large indoor environments with low-cost, noisy, and sparse sonars," in *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pp. 1395 –1401, May 2009.

[15] S.-Y. Hwang, J.-T. Park, and J.-B. Song, "Autonomous navigation of a mobile robot using an upward-looking camera and sonar sensors," in *Advanced Robotics and its Social Impacts (ARSO), 2010 IEEE Workshop on*, pp. 40 –45, Oct. 2010.

[16] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering*, vol. 82, pp. 35–45, Mar. 1960.

[17] A. H. Jazwinski, *Stochastic Processes and Filtering Theory.* Academic Press, Apr. 1970.

[18] S. J. Julier and J. K. Uhlmann, "Unscented filtering and nonlinear estimation," *Proceedings of the IEEE*, vol. 92, pp. 401 – 422, Mar. 2004.

[19] J. Campbell, R. Sukthankar, I. Nourbakhsh, and A. Pahwa, "A robust visual odometry and precipice detection system using consumer-grade monocular vision," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pp. 3421 – 3427, Apr. 2005.

[20] D. Nister, O. Naroditsky, and J. Bergen, "Visual odometry," in *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, vol. 1, pp. I–652 – I–659 Vol.1, June 2004.

[21] A. J. Davison and N. Kita, "Sequential localisation and map-building for real-time computer vision and robotics," *Robotics and Autonomous Systems*, vol. 36, pp. 171 – 183, Sept. 2001.

[22] P. Pinies and J. Tardos, "Scalable SLAM building conditionally independent local maps," in *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pp. 3466 –3471, Oct. 2007.

[23] H. Strasdat, J. M. M. Montiel, and A. Davison, "Scale drift-aware large scale monocular slam," in *Proceedings of Robotics: Science and Systems*, (Zaragoza, Spain), June 2010.

[24] J. Civera, O. G. Grasa, A. J. Davison, and J. M. M. Montiel, "1-point ransac for extended kalman filtering: Application to real-time structure from motion and visual odometry," *J. Field Robot.*, vol. 27, pp. 609–631, Sept. 2010.

[25] J. Civera, A. Davison, and J. Montiel, "Inverse depth parametrization for monocular slam," *Robotics, IEEE Transactions on*, vol. 24, no. 5, pp. 932–945, 2008.

[26] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pp. 3310 –3317 vol.4, May 1994.

[27] Giraff Technologies AB, *Advanced Operational Guide For Giraff Version 3.1*, June 2011.

[28] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library.* O'Reilly Media, Oct. 2008.

[29] R. Laganière, *OpenCV 2 Computer Vision Application Programming Cookbook.* Packt Publishing, May 2011.

[30] D. L. Baggio, S. Emami, D. M. Escrivá, K. Ievgen, N. Mahmood, J. Saragih, and R. Shilkrot, *Mastering OpenCV with Practical Computer Vision Projects.* Packt Publishing, Dec. 2012.

[31] Giraff Technologies AB, *Giraff Motor Controller Board Serial Interface*, May 2012.

[32] J. Canny, “A computational approach to edge detection,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 8, pp. 679–698, June 1986.

Appendix

A. The Motor Controller Interface

This appendix is intended to expand on the manufacturer's own documentation, «Giraff Motor Controller Board Serial Interface» [31]. It notes and corrects errors and omissions in their documentation, and attempts to explain a few things that may be unclear. However, you won't need to have the manufacturer's documentation in order for this appendix to be useful to you.

A.1. Overview

The Giraff's motor controller is mounted near the bottom of the Giraff's chassis. Its brain is an AVR microcontroller. The controller's primary functions are to control the two side wheels, to control the head's tilt angle, and to report the state of the two buttons and the dial on the Giraff's chassis. It can also report the charge level of the Giraff's battery, but this is not covered in this appendix.

The controller responds to commands sent to it via a RS232-style serial interface. For making the Giraff move, these commands don't control the wheels directly, but sets parameters such as acceleration, maximum speed, and distance, which the board then uses to calculate a motion profile. This profile is followed until either the motion is complete, or another command changes the motion profile. Since instantaneous changes in speed aren't physically possible (and trying it may cause damage to the Giraff), a

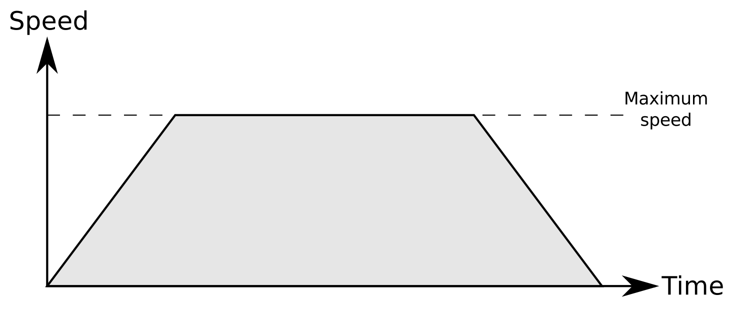


Figure A.1.: Motion profile

standard motion profile has a «ramp-up» with constant acceleration until the requested maximum speed is reached, followed by a period of constant speed, then a final «ramp-down» with constant deceleration until the Giraff reaches a full stop at the requested final position (or at least close to it). This results in a trapezoidal speed profile, as seen in Figure A.1. (If the requested distance is very short, the maximum speed may not be reached, resulting in a triangular speed profile.)

A.2. Movement styles

Any of the following styles can selected with the «set r» command.

A.2.1. Straight line motion

The Giraff can move forwards or backwards in straight lines. In this style, «set p» specifies the distance in meters (which can be negative to move backwards), «set v» specifies the maximum speed (in meters per second), and «set a» specifies the acceleration. The speed and acceleration should be positive numbers, regardless of direction. The distance travelled and current speed is reported as «cdis» and «gvr», respectively. «cang» is always reported as zero.

A.2.2. Rotating in place

The Giraff can rotate in place by driving its wheels in opposite directions. In this style, the distance given to «set p» is in degrees, not meters. Positive turns right, negative turns left. The angle travelled and current angular speed is reported as «cang» and «gvr», respectively, though both of these have the opposite sign of what they should. Note that, like for straight line motion, the speed taken by «set v» (and acceleration given to «set a») is specified in meters, not in degrees. A program must convert accordingly if it wants a specific turning speed. «cdis» is always reported as zero.

A.2.3. Curved motion

The Giraff can turn while moving by driving its wheels with different speeds. This style is a superset of straight line motion, and is built around a concept called a «virtual gear ratio». When starting or ending a turn, the gear ratio is changed gradually from the initial to the final gear ratio, much like acceleration does for velocity (though in this case the rate of change is per meter, not per second).

In addition to the straight-line parameters, «set vg» specifies the maximum gear ratio to use, and «set vgr» specifies the gear ratio rate of change (per meter). The latter should be positive or negative depending on whether to turn right or left, respectively. Due to bugs, I don't know for sure whether the former should also be negative when turning left, though it seems to work that way. Finally, «set cdp» specifies the position (in meters) at which to start changing the gear ratio back towards zero, ultimately ending the turn. The gear ratio will also automatically be reset to zero if the Giraff completes its motion and stops.

In the manufacturer's documentation [31], the wheel speeds are given as

Left Wheel Velocity = Overall Velocity * (1/(1-vg))
Right Wheel Velocity = Overall Velocity * (1/(1+vg))

It also says a gear ratio of 1.0 is the ratio where the Giraff will pivot around its own wheels. Since the above formulas don't actually achieve this (it would cause a division by zero), or even maintain the overall velocity, I believe the correct formulas to use are actually

Left Wheel Velocity = Overall Velocity * (1+vg)
Right Wheel Velocity = Overall Velocity * (1-vg)

(This is acknowledged by an engineer at Giraff Technologies.)

When executing curved motion, «cdis» still reports the distance travelled along the curve, but according to the manufacturer, «gvr» reports the velocity of the left wheel, not the overall velocity. Moreover, testing shows that «gvr» is calculated using the (probably incorrect) formulas from the documentation, so a program needs to take these things into account when trying to determine the actual speed. The current virtual gear ratio is reported as «cvg». The angle that has been covered is not reported, as «cang» is still always zero. A program would need to calculate such things on its own, based on distance travelled and such.

Also note that attempting to preempt a curved motion command in the current version of the microcontroller may cause unexpected behaviour. Depending on the circumstances, the virtual gear ratio may jump instantly to an undesirably high ratio. The only way I found to avoid this is to set «vgr» to zero when preempting, but this will cause the virtual gear ratio to jump instantly to zero instead. This is, of course, also undesirable due to the physical stress it causes to the Giraff's hardware. The problem has been reported to Giraff Technologies and will hopefully be fixed in a future revision of the microcontroller's software.

A.3. Connection details

The controller's serial interface is wired to the main computer's serial port. To communicate with the controller, the parameters should be set to

- Baud Rate: 115200
- Data Bits: 8
- Stop Bit: 1
- Parity: None

When a computer opens the serial port, it is expected to set the DTR (Data Terminal Ready) signal high. When the controller board detects the DTR signal, it will power up and identify itself by transmitting a line like the following:

`# Giraf` *version*`,`*date*

(followed by carriage return and line feed characters). From testing on an actual Giraff, however, it appears that before this line, another line may appear, saying just `Ca>`. It may be an artifact of the board's initialization process and should probably be ignored.

After the version line, an `OK>` prompt will appear (followed by carriage return and line feed) when the board is ready to receive commands. Commands should not be sent before this. When a command has been sent to the board (followed by a carriage return only), the board will generate an appropriate response, followed by a new `OK>` prompt. Again, a new command should not be sent before the new OK prompt has been seen.

All commands and responses are made up of regular ASCII strings. In the documentation, parameters are regular human-readable decimal numbers (in ASCII encoding). However, testing showed that while commands can be sent using this format, responses do not seem to work like this. This was not in the documentation, but some detective work suggested that, depending on the type of the parameter, the responses are encoded as follows:

Type	Transfer format	Binary interpretation
Integer	`I`$\star$*aabbccdd*	32-bit two's complement integer
Floating-point	`F`$\star$*aabbccdd*	IEEE 754 single-precision floating point

The transfer formats encode the binary value as hexadecimal numbers, in little-endian byte order. That is, each pair of hexadecimal digits (i.e., each 8-bit byte) has the most significant digit to the left, but on the other hand, the most significant (aka highest order) byte is to the right (i.e., *aa* is least significant, and *dd* is most significant). Thus, some care needs to be taken to keep things ordered correctly when decoding the value.

Some commands («get button_data» and «get bulk_data») return more than one parameter. In this case, the parameters are returned as a comma-separated name-value list. For example, the response from «get button_data» looks like

`but0:`*value*`,but1:`*value*`,dial:`*value*
where each individual value is encoded as described above.

A.4. Commands

All listed «set» commands have a corresponding «get» command which returns the last set value. Commands that start neither with «set» nor «get» do not return any values (only the `OK>` prompt).

Where not otherwise specified, command parameters are floating-point values.

A.4.1. set v

Sets the maximum speed (velocity), in meters per second.

A.4.2. set r

Selects the movement style, according to the following table.

r	mode
$r = 0$	Rotating in place
$0 < r \leq 50$	Straight line motion
$50 < r$	Curved motion

(The documentation do not mention the $r > 50$ requirement for curved motion.)

A.4.3. set a

Sets the acceleration, in meters per second per second.

A.4.4. set p

Starts a move. All other motion parameters must be set before issuing this command.

When rotating in place, specifies number of degrees to rotate. When moving in a straight or curved line, specifies number of meters to move. (See A.2 for details.)

If another move is already in progress, the previous move may be preempted, or the new move queued until the previous move is complete, depending on what mode is set with «set mode». (See A.4.8.)

A.4.5. get cang

Gets the current angle.

When rotating in place, returns degrees rotated so far. When moving in a straight or curved line, always zero.

A.4.6. get cdis

Gets the current distance.

When moving in a straight or curved line, returns distance travelled so far. When rotating in place, always zero.

A.4.7. get gvr

Gets the current (instantaneous) velocity.

When rotating in place, returns degrees per second. When moving in a straight or curved line, returns meter per second. If moving in a curved line, special care must be taken when interpreting this value. (See A.2.3.)

A.4.8. set mode

An integer. Sets the movement mode. This is a bitmask. The following bits can be set (but can not be read back):

Bit	Value	Description
0	1	Absolute movement mode
2	4	Buffer next move

The following bits can be read:

Bit	Value	Description
3	8	ESTOP
7	128	Currently moving

In relative mode, all moves are relative to the current position. When absolute mode is enabled, the Giraff begins tracking distances since the moment absolute mode is enabled. All moves, including the «set p» parameters and the reported «cang» and «cdis», becomes relative to this position. Note that this only tracks distance travelled, and is dependent on the current movement style. Changing the style will reset the absolute mode position to the current position.

In unbuffered mode, new moves issued with «set p» preempt the current move, and starts immediately. When buffering is enabled, a new move gets queued and only starts when the previous move completes (i.e., when the Giraff comes to a full stop). Up to 4 moves can be buffered.

If the ESTOP bit is set, it means something with the wheels is not working correctly. Details may be available from the manufacturer.

A.4.9. set undock

Starts an undock sequence. Queues two moves: one to back out the specified distance, and one to rotate 180 degrees.

A.4.10. home

No parameter. Starts the head homing sequence. The head slowly tilts, searching for its «home» position. This is the position the head is in when the Giraff is «sleeping», about 45 degrees off vertical.

Note that the homing sequence appears to start automatically when the microcontroller is activated, so issuing this command is usually not needed.

A.4.11. get tilt_homing_state

An integer. Returns the homing status.

Value	Description
0	Homing not started
1	Homing started
2	Homing failed
3	Homing succeeded

A.4.12. set tilt_angle_from_home

Tilts the head to the given angle, in radians, relative to the home position. (The documentation says relative to vertical, but that's not the case.)

If the head homing sequence has not been completed, this command will preempt the homing sequence and usually tilt the head to the wrong angle.

A.4.13. set vg

Sets the maximum virtual gear ratio. See A.2.3.

A.4.14. set vgr

Sets the virtual gear ratio rate of change (per meter). See A.2.3.

A.4.15. set cdp

Sets the Clothoid Deceleration Point, the point in the move where the virtual gear ratio starts decelerating to its final value. See A.2.3.

A.4.16. get cvg

Gets the current virtual gear ratio. See A.2.3.

A.4.17. get but0

Gets number of button 0 presses since microcontroller startup.

A.4.18. get but1

Gets number of button 1 presses since microcontroller startup.

A.4.19. get dial

Gets rotation of dial since microcontroller startup.

A.4.20. get button_data

Gets «but0», «but1», and «dial» with a single command. Returns the parameters as a list.

A.4.21. get bulk_data

Gets «cang», «cdis», «gvr», «tilt_angle_from_home», «imdl», «imdr», «cvg», and «mode» with a single command. Returns the parameters as a list. Very useful for regular retrieval of motor odometry.

B. Source code listings

This appendix has been added for the convenience of those reading this thesis, so that they don't have to get a copy of the CD-ROM to see the source code. Instead, they can peruse it here.

B.1. GiraffMotor.hpp

```
#ifndef GIRAFFMOTOR_HPP
#define GIRAFFMOTOR_HPP

#include "DisplayWindow.hpp"

#include <windef.h>
#include <fstream>

class GiraffMotorSim;

class GiraffMotor
{
public:
    enum ReplyType {
        NoReply,
        SimpleReply,
        BulkReply
    };

    GiraffMotor(DisplayWindow* win);
    ~GiraffMotor();
    bool Start();
    void Stop();
    bool Process();
    bool StartRecord(const std::string& name);
    void StopRecord();
    bool StartPlayback(const std::string& name);
    void StopPlayback();

    // manual/interactive commands triggered by user
    void Undock();
    void Home();
    void SetTilt(double angle);
    void GetBulkData();
    void SetMotion(double speed);
    void SetTurn(double speed);

    // special functions
    bool SendCommand(const std::string& cmd,
                     bool silent=false);
    void AddReply(const std::string& reply);
    std::string GetParameter(const std::string& param,
                             ReplyType type,
                             bool silent=false);
    std::string SetParameter(const std::string& param,
                             double value,
                             bool silent=false);
    bool SendUserCommand(const std::string& cmd);

    // Hack to check impact of get_bulk_data per-frame,
    // should otherwise always be left on.
    // This field should be removed, especially if motor
    // control is moved into a separate thread
    bool m_autoupdate;

private:
    static const double turn_factor;
    DisplayWindow* m_win;
    HANDLE m_port;
    LONGLONG m_freq;
    std::ofstream m_mrec;
    std::ifstream m_mplay;
    GiraffMotorSim* m_sim;
    bool m_rec, m_play;
    LONGLONG m_rectime, m_playtime;
    std::string m_readbuf;
    bool m_bufchecked;
    double m_accel, m_vgaccel;
    // current position estimate
    double m_curx, m_cury, m_curdir, m_curspd, m_currot;
    // current user request
    double m_usrmotionspd, m_usrturnspd;
    double m_curmotionspd, m_curturnspd;
    // current motor command
    int m_turnmode;
    unsigned m_absmode;
    double m_speed, m_gear, m_gearrate, m_gearpos;
    double m_nextdis, m_nextpos, m_brkdist;
    // current motor status
    double m_cang, m_cdis, m_gvr, m_cvg;
    double m_lcang, m_lcdis, m_lgvr, m_lcvg;
    unsigned m_cmode, m_lmode;
    LONGLONG m_cstamp, m_lstamp;
    double m_timedelta;
    bool m_not_first;

    void RecordParameter(const std::string& param,
                         const std::string& reply,
                         const std::string& orig,
                         ReplyType type,
                         char flag);
    bool PlaybackData();
    bool InitPort();
    bool InitSimulator();
    void WaitForLine();
    bool ReadLine(std::string& line,
                  ReplyType type=NoReply,
                  bool silent=false);
    bool ReadReply(std::string& reply,
                   ReplyType type,
                   bool silent=false);
    std::string WriteCommand(const std::string& out,
                             ReplyType type,
                             char flag,
                             bool silent);
    bool ReadVersion();
    std::string FormatReply(const std::string& reply,
                            ReplyType type);
    void FormatField(std::ostream& ost,
                     std::istream& ist);
    unsigned ToInt(unsigned u);
    float ToFloat(unsigned u);
    unsigned ToInt(const std::string& data);
    float ToFloat(const std::string& data);
    void ParseBulkData(const std::string& data);
    void RunMotor();
    void CalcMove();
    void CalcRotate();
    void CalcMoveStep(double dis);
    void CalcRotateStep(double ang);
    double CalcMoveBrakeDist(double spd);
    double CalcRotateBrakeDist(double rot);
    void UpdatePosition();
    void ShowPosition();
};

#define GIRAFF_BUFFERS 4

class GiraffMotorSim
{
public:
    GiraffMotorSim(GiraffMotor* ctl);
    ~GiraffMotorSim();
    void SimulateCommand(const std::string& cmd);

private:
    GiraffMotor* m_ctl;

    static const double turn_factor;
    static const double default_tilt;

    struct Move
    {
        unsigned mode;
        // parameters used
        double v, r, a, p;
        double vg, vgr, cdp;
    };

    // wheel moves
    unsigned m_bufcount;
    Move m_buf[GIRAFF_BUFFERS+1];
    // head tilts
    unsigned m_homing;
    double m_tilt;
    // for timing
    // (c = counter value, equivalent to time)
    LONGLONG m_freq, m_lastc;
    // Current Giraff state
    double m_cang, m_cdis, m_cvg;
    double m_vang, m_vdis, m_gvr;

    // Current motion profile
    LONGLONG m_startc, m_stopc;
    // distance part (for moving around)
    LONGLONG m_updc, m_downdc;
    double m_startdv, m_peakdv, m_rampda;
    double m_refdp, m_updp, m_downdp, m_stopdp;
    // angular part (for turning)
    LONGLONG m_upac, m_downac;
    double m_startav, m_peakav, m_rampaa;
    double m_refap, m_upap, m_downap, m_stopap;
    // gear ratio part
    LONGLONG m_upgc, m_downgc, m_stopgc;
    double m_startgr, m_peakgr, m_rupgr, m_rdowngr;
    double m_stopgr, m_downgd;

    void StartStraight(double dist,
                       double start_pos,
                       double start_spd,
                       double cdp,
                       double start_vg);
    void StartRotate(double degrees,
                     double start_angle,
                     double start_spd);
    double TimeFromPosition(double pos,
                            double ramp_up_time,
                            double cruise_time,
                            double ramp_down_time,
                          double ramp_up_dist,
                          double cruise_dist,
                          double ramp_down_dist,
                          double accel,
                          double start_speed,
                          double peak_speed);
    void UpdateMotion();
    void StartMotion();
    void EndMotion();
    bool QueueMotion();
    bool QueueUndock(double dist);
    void SimulateLag(unsigned bytes);
    void SimulateReply(const std::string& reply);
    void InputFloat();
    void Output(std::ostream& out, double val);
    void Output(std::ostream& out, unsigned val);
};

#endif // GIRAFFMOTOR_HPP
```

B.2. GiraffMotor.cpp

```
#include "GiraffMotor.hpp"

#include <windows.h>

#include <sstream>
#include <iomanip>

#define PORT_NAME "COM1"

using namespace std;

// The acceleration the GiraffMotor class uses by default.
// (Not necessarily the same as what the motor
// controller board itself uses by default.)
#define DEF_ACCEL 0.5

// The virtual gear ratio rate of change the GiraffMotor
// class uses by default.
#define DEF_VGACCEL 1.0

// The distance between the Giraff's wheels are 499mm,
// so when rotating in place, their turn radius is 249.5mm.
#define TURN_RADIUS 0.2495
// Conversion factor between degrees and
// circle arc covered by wheels.
#define TURN_FACTOR (180 / (TURN_RADIUS * M_PI))

// This is used when the user is controlling the motor
// manually, so the distance to go isn't known in
// advance. To calculate the distance we tell the
// motor to go, we multiply the "braking distance"
// with this factor. (Every time the distance left falls
// below a factor of 2, a new command is automatically
// sent to the controller in order to make it keep going.
// Hence, this factor must be more than 2.)
#define AHEAD_FACTOR 10

// Use absolute mode, which makes position tracking
// a little more accurate in some cases.
// Unfortunately, curved motion may be troublesome in
// this mode, because of bugs in the controller.
//#define USE_ABSOLUTE_MODE

// Whether the simulated controller will reverse back if
// its braking distance is too long to stop at the requested
// position (applicable when the destination position is
// suddenly changed while traveling at full speed).
// The real Giraff seems to do this in absolute mode.
#define SIM_OVERSHOOT_FIX

// Enable simulation of curved motion.
#define SIM_CURVED

// gvr is (incorrectly) speed of left wheel
// instead of overall speed.
#define GVR_IS_LEFT

// Show the "OK >" prompt on the display.
//#define SHOW_PROMPT

enum ModeBit {
    MODE_ABSOLUTE = 1,
    MODE_BUFFERED = 4,
    MODE_ESTOP = 8,
    MODE_MOVING = 128
};

static double fix_degrees(double angle)
{
    while (angle < 0)
    {
        angle += 360;
    }
    while (angle >= 360)
    {
        angle -= 360;
    }
    // returned angle is between 0 and 360
    return angle;
}

#if 0
static double ctr_degrees(double angle)
{
    // returned angle is between -180 and 180
    return fix_degrees(angle+180)-180;
}
#endif

GiraffMotor* motorControl;

const double GiraffMotor::turn_factor = TURN_FACTOR;

GiraffMotor::GiraffMotor(DisplayWindow* win) :
    m_autoupdate(true),
    m_win(win), m_port(INVALID_HANDLE_VALUE),
    m_sim(NULL), m_rec(false), m_play(false),
    m_rectime(0), m_playtime(0),
    m_bufchecked(false),
    m_accel(DEF_ACCEL), m_vgaccel(DEF_VGACCEL),
    m_curx(0), m_cury(0), m_curdir(0), m_curspd(0),
    m_usrmotionspd(0), m_usrturnspd(0),
    m_curmotionspd(0), m_curturnspd(0),
    m_turnmode(0), m_absmode(0),
    m_cang(0), m_cdis(0), m_gvr(0), m_cvg(0),
    m_lcang(0), m_lcdis(0), m_lgvr(0), m_lcvg(0),
    m_cmode(0), m_lmode(0),
    m_cstamp(0), m_lstamp(0), m_timedelta(0),
    m_not_first(true)
{
    // get timer frequency
    LARGE_INTEGER freq;
    QueryPerformanceFrequency(&freq);
    m_freq = freq.QuadPart;
}

GiraffMotor::~GiraffMotor()
```

```
{
    StopRecord();
    StopPlayback();
    Stop();
}

bool GiraffMotor::Start()
{
    if (!InitPort())
    {
        // Could not initialize serial port
        Stop();
        return false;
    }
    if (!ReadVersion())
    {
        // Did not detect Giraff board
        //Stop();
        return true;
    }
    return true;
}

void GiraffMotor::Stop()
{
    // Stop any running simulation
    if (m_sim)
    {
        delete m_sim;
        m_sim = NULL;
    }
    // If the serial port is open, close it
    if (m_port != INVALID_HANDLE_VALUE)
    {
        CloseHandle(m_port);
        m_port = INVALID_HANDLE_VALUE;
    }
}

bool GiraffMotor::Process()
{
    std::string line;
    // Check for unexpected responses,
    // maybe resulting from user commands
    while (ReadLine(line))
    {
        // unexpected
    }
    if (!m_autoupdate)
    {
        return true;
    }
    if (m_play)
    {
        if (!PlaybackData())
        {
            return false;
        }
        UpdatePosition();
    }
    else
    {
        // Transfer current motor state
        // (this is a few ms of just waiting,
        // maybe consider creating a separate
        // thread for these things, though that
        // may make it harder to synchronize
        // readings from the camera and motor)
        string data = GetParameter("bulk_data", BulkReply, true);
        ParseBulkData(data);
        // Send motor commands as needed
        RunMotor();
    }
    ShowPosition();
    return true;
}

bool GiraffMotor::StartRecord(const string& name)
{
    if (m_rec)
    {
        StopRecord();
    }
    string fn = name + ".txt";
    // start motor recording
    m_mrec.open(fn.c_str(), ios_base::out | ios_base::trunc);
    if (m_mrec.is_open())
    {
        // get reference time for recording
        LARGE_INTEGER current;
        QueryPerformanceCounter(&current);
        m_rectime = current.QuadPart;
        // set recording state
        m_rec = true;
        m_win->PrintLeft("Starting motor record " + fn);
        return true;
    }
    else
    {
        m_win->PrintLeft("Couldn't start motor record");
        return false;
    }
}

void GiraffMotor::StopRecord()
{
    if (m_rec)
    {
        m_mrec.close();
        m_rec = false;
        m_win->PrintLeft("Motor record stopped");
    }
}

bool GiraffMotor::StartPlayback(const string& name)
{
    string fn = name + ".txt";
    // start motor recording
    m_mplay.open(fn.c_str(), ios_base::in);
    if (m_mplay.is_open())
    {
        // get reference time for playback
        LARGE_INTEGER current;
        QueryPerformanceCounter(&current);
        m_playtime = current.QuadPart;
        // set recording state
        m_play = true;
        m_win->PrintLeft("Starting motor playback " + fn);
        return true;
    }
    else
    {
        m_win->PrintLeft("Couldn't start motor playback");
        return false;
    }
}

void GiraffMotor::StopPlayback()
{
    if (m_play)
    {
        m_mplay.close();
        m_play = false;
        m_win->PrintLeft("Motor playback stopped");
    }
}

void GiraffMotor::RecordParameter(const std::string& param,
                                  const std::string& reply,
                                  const std::string& orig,
                                  ReplyType type,
                                  char flag)
{
    if (!m_rec)
    {
        return;
    }
    LARGE_INTEGER current;
    QueryPerformanceCounter(&current);
    LONGLONG diff = current.QuadPart - m_rectime;
    double ofs = (double)diff / m_freq;
    m_mrec << setprecision(3) << fixed
           << setw(8) << ofs
           << ":[" << flag << "] "
           << param << ": "
           << FormatReply(reply, type);
    if (!orig.empty())
    {
        m_mrec << " <= " << orig;
    }
    m_mrec << endl;
}

bool GiraffMotor::PlaybackData()
{
    if (m_mplay.eof())
    {
        // already complete
        return false;
    }
    for (;;)
    {
        double ofs;
        string line;
        m_mplay >> ofs;
        if (m_mplay.eof())
        {
            // playback complete
            return false;
        }
        getline(m_mplay, line);

        // delay as appropriate to force the
        // playback to have about the same speed
        // as the original recording did
        LARGE_INTEGER current;
        QueryPerformanceCounter(&current);
        LONGLONG target = m_playtime + ofs * m_freq;
        LONGLONG diff = target - current.QuadPart;
        if (diff > 0)
        {
            unsigned msec = (diff * 1000) / m_freq;
            if (msec > 0)
            {
                Sleep(msec);
            }
        }

        if (line.length() < 4)
        {
            return false;
        }

        char flag = line[2];
        size_t colpos = line.find(": ", 5);
        if (colpos == string::npos)
```

```
{
    return false;
}
string param = line.substr(5, colpos-5);
size_t replypos = colpos+2;
size_t seppos = line.find(" <= ", replypos);
string reply, orig;
if (seppos == string::npos)
{
    reply = line.substr(replypos);
}
else
{
    reply = line.substr(replypos, seppos-replypos);
    orig = line.substr(seppos+4);
}

if (flag == ' ' && param == "bulk_data")
{
    ParseBulkData(reply);
    if (m_brkdist != 0)
    {
        // update state as needed to estimate
        // the original movements
        if (!(m_cmode & MODE_MOVING))
        {
            m_brkdist = 0;
        }
    }
    // this kind of record happens after we get a video frame,
    // so exit loop here in order to display the recorded frame
    break;
}
if (flag == 'S')
{
    // recorded a SetParameter
    string cmd = "set " + param + " " + orig;
    m_win->PrintRight(cmd);
    m_win->PrintRight(reply);
    // parse what we need to estimate
    // the original movements
    istringstream ist(reply);
    if (param == "r")
    {
        double r;
        ist >> r;
        m_turnmode = (r > 0) ? 1 : -1;
    }
    else if (param == "mode")
    {
        // For mode, the reply is generally incorrect,
        // so take the mode from the original request.
        istringstream ist2(orig);
        unsigned mode;
        ist2 >> mode;
        m_absmode = mode & MODE_ABSOLUTE;
    }
    else if (param == "v")
    {
        ist >> m_speed;
    }
    else if (param == "vg")
    {
        ist >> m_gear;
    }
    else if (param == "vgr")
    {
        ist >> m_gearrate;
    }
    else if (param == "p")
    {
        double pos;
```

```
                ist >> pos;
                if (m_turnmode > 0)
                {
                    double dist = m_absmode ? pos - m_cang : pos;
                    if (dist != 0)
                    {
                        int sign = (dist > 0) ? 1 : -1;
                        m_brkdist = CalcMoveBrakeDist(sign * m_speed);
                    }
                }
                else if (m_turnmode < 0)
                {
                    double dist = m_absmode ? pos - m_cdis : pos;
                    if (dist != 0)
                    {
                        int sign = (dist > 0) ? 1 : -1;
                        m_brkdist = CalcRotateBrakeDist(sign * m_speed * turn_factor);
                    }
                }
            }
        }
        else if (flag == 'G')
        {
            // recorded a GetParameter
            string cmd = "get " + param;
            m_win->PrintRight(cmd);
            m_win->PrintRight(reply);
        }
        else if (flag == 'C')
        {
            // recorded a SendCommand
            m_win->PrintRight(param);
        }
    }
    return true;
}

void GiraffMotor::Undock()
{
    SetParameter("undock", -0.5);
}

void GiraffMotor::Home()
{
    SendCommand("home");
}

void GiraffMotor::SetTilt(double angle)
{
    SetParameter("tilt_angle_from_home", angle);
}

void GiraffMotor::GetBulkData()
{
    GetParameter("bulk_data", BulkReply);
}

void GiraffMotor::SetMotion(double speed)
{
    m_usrmotionspd = speed;
}

void GiraffMotor::SetTurn(double speed)
{
    m_usrturnspd = speed;
}

bool GiraffMotor::InitPort()
{
#ifdef PORT_NAME
    // Try to open the serial port that is
    // connected to the motor control board.
    m_port = CreateFile(PORT_NAME,
                        GENERIC_READ | GENERIC_WRITE,
                        0,
                        NULL,
                        OPEN_EXISTING,
                        0,
                        NULL);
    if (m_port == INVALID_HANDLE_VALUE)
    {
        // Could not open real serial port,
        // initialize simulator instead,
        // so the rest of the program
        // can still be used.
        return InitSimulator();
    }
    // Get current serial port settings
    DCB dcb;
    memset(&dcb, 0, sizeof(dcb));
    dcb.DCBlength = sizeof(dcb);
    if (!GetCommState(m_port, &dcb))
    {
        // Could not get state from serial port
        return false;
    }
    // Configure serial port
    // Set 115200 bps, 8 data bits, no parity, 1 stop bit
    dcb.BaudRate = CBR_115200;
    dcb.ByteSize = 8;
    dcb.Parity = NOPARITY;
    dcb.StopBits = ONESTOPBIT;
    // Set event char to the end of line character,
    // so that we can use WaitCommEvent to wait
    // for the arrival of a complete line
    dcb.EvtChar = '\n';
    if (!SetCommState(m_port, &dcb))
    {
        // Could not configure serial port
        return false;
    }
    // Set the events that WaitCommEvent should wait for.
    if (!SetCommMask(m_port, EV_ERR | EV_RXFLAG))
    {
        // Could not configure serial port
        return false;
    }
    // Set appropriate timeouts to make sure ReadFile
    // always returns immediately without waiting
    // (possibly returning an error if no data is
    // available). This is necessary since we
    // don't know in advance how long a reply is
    // going to be. So if we need to wait for one,
    // we'd rather use WaitCommEvent, then use
    // ReadFile to read whatever we got, without
    // waiting any longer than that.
    COMMTIMEOUTS tos;
    tos.ReadIntervalTimeout = MAXDWORD;
    tos.ReadTotalTimeoutMultiplier = 0;
    tos.ReadTotalTimeoutConstant = 0;
    tos.WriteTotalTimeoutMultiplier = 0;
    tos.WriteTotalTimeoutConstant = 0;
    if (!SetCommTimeouts(m_port, &tos))
    {
        // Could not configure serial port
        return false;
    }
    return true;
#else
    // No serial port, initialize simulator
    return InitSimulator();
#endif
}
```

```
bool GiraffMotor::InitSimulator()
{
    m_sim = new GiraffMotorSim(this);
    return true;
}

void GiraffMotor::WaitForLine()
{
    if (!m_port)
    {
        return;
    }
    DWORD mask = 0;
    // There's a risk that this could wait forever
    // if the motor board is failing, perhaps we
    // should use the overlapped I/O mode so that
    // we can limit the waiting time.
    WaitCommEvent(m_port, &mask, NULL);
}

bool GiraffMotor::ReadLine(string& line,
                           ReplyType type,
                           bool silent)
{
    // see if there's already a complete line in the buffer
    if (!m_bufchecked &&
        !m_readbuf.empty())
    {
        size_t n = m_readbuf.find('\n');
        if (n != string::npos)
        {
            // found one, return it
            n++; // end line after the \n
            line = m_readbuf.substr(0, n);
            m_readbuf.erase(0, n);
            if (!silent)
            {
                m_win->PrintRight(FormatReply(line, type));
            }
            return true;
        }
        else
        {
            m_bufchecked = true;
        }
    }
    // no such luck, try to read more from the serial port
    char buf[256];
    DWORD bytesRead;
    if (!m_port ||
        !ReadFile(m_port, buf, sizeof(buf), &bytesRead, NULL))
    {
        // read failure
        return false;
    }
    // read successful, see if we now have a complete line
    char* eol = (char*)memchr(buf, '\n', bytesRead);
    if (eol)
    {
        // we have one, return it
        eol++; // end line after the \n
        line = m_readbuf;
        line.append(buf, eol-buf);
        // store remainder of buffer for later
        m_readbuf.assign(eol, buf+bytesRead-eol);
        m_bufchecked = false;
        if (!silent)
        {
            m_win->PrintRight(FormatReply(line, type));
        }
        return true;
    }
    else
    {
        // incomplete line, store buffer for later
        m_readbuf.append(buf, bytesRead);
        m_bufchecked = true;
        return false;
    }
}

bool GiraffMotor::ReadReply(string& reply,
                            ReplyType type,
                            bool silent)
{
    string line1, line2;
    if (type != NoReply)
    {
        while (!ReadLine(line1, type, silent))
        {
            WaitForLine();
        }
    }
#ifndef SHOW_PROMPT
    silent = true;
#endif // SHOW_PROMPT
    while (!ReadLine(line2, NoReply, silent))
    {
        WaitForLine();
    }
    // Remove the \r\n from the reply.
    reply = line1.substr(0, line1.length()-2);
    // Ignore line2 for now, it is always supposed
    // to be "OK >\r\n", and in the event that it isn't,
    // I'm not yet sure what to do about it.
    return true;
}

void GiraffMotor::AddReply(const string& reply)
{
    m_readbuf += reply;
    m_bufchecked = false;
}

string GiraffMotor::WriteCommand(const string& out,
                                 ReplyType type,
                                 char flag,
                                 bool silent)
{
    string reply;
    if (!silent)
    {
        m_win->PrintRight(out);
    }
    if (m_sim)
    {
        m_sim->SimulateCommand(out);
        ReadReply(reply, type, silent);
        return reply;
    }
    DWORD written = 0;
    if (!m_port ||
        !WriteFile(m_port, out.data(), out.length(),
                   &written, NULL) ||
        written != out.length() ||
        !ReadReply(reply, type, silent))
    {
        return string();
    }
    return reply;
}

bool GiraffMotor::SendCommand(const string& cmd,
                              bool silent)
```

```
{
    string reply;
    reply = WriteCommand(cmd + "\r", NoReply, 'C', silent);
    RecordParameter(cmd, reply, "", NoReply, 'C');
    return true;
}

string GiraffMotor::GetParameter(const string& param,
                                 ReplyType type,
                                 bool silent)
{
    string reply;
    char flag = silent ? ' ' : 'G';
    ostringstream ost;
    ost << "get " << param << "\r";
    reply = WriteCommand(ost.str(), type,
                         flag, silent);
    RecordParameter(param, reply, "", type, flag);
    return reply;
}

string GiraffMotor::SetParameter(const string& param,
                                 double value,
                                 bool silent)
{
    string valstr, reply;
    ostringstream ost;
    ost << value;
    valstr = ost.str();
    reply = WriteCommand("set " + param +
                         " " + valstr + "\r", SimpleReply,
                         'S', silent);
    RecordParameter(param, reply, valstr, SimpleReply, 'S');
    return reply;
}

bool GiraffMotor::SendUserCommand(const string& cmd)
{
    string reply;
    if (cmd.compare(0, 4, "get ") == 0)
    {
        if (cmd == "get bulk_data")
        {
            reply = WriteCommand(cmd + "\r", BulkReply, 'G', false);
            RecordParameter(cmd.substr(4), reply,
                            "", BulkReply, 'G');
        }
        else
        {
            reply = WriteCommand(cmd + "\r", SimpleReply, 'G', false);
            RecordParameter(cmd.substr(4), reply,
                            "", SimpleReply, 'G');
        }
    }
    else if (cmd.compare(0, 4, "set ") == 0)
    {
        size_t n = cmd.find(' ', 4);
        reply = WriteCommand(cmd + "\r", SimpleReply, 'S', false);
        if (n != string::npos)
        {
            RecordParameter(cmd.substr(4, n-4), reply,
                            cmd.substr(n+1), SimpleReply, 'S');
        }
        else
        {
            RecordParameter(cmd.substr(4), reply,
                            "", SimpleReply, 'S');
        }
    }
    else
    {
        reply = WriteCommand(cmd + "\r", NoReply, 'C', false);
        RecordParameter(cmd, reply, "", NoReply, 'C');
    }
    return true;
}

bool GiraffMotor::ReadVersion()
{
    string line;
    // read initial line
    while (!ReadLine(line))
    {
        WaitForLine();
    }
    if (line.at(0) != '#')
    {
        // Seems the controller might send an extra line
        // (saying "Ca>") before it sends the version
        // line. If this happens, try reading again.
        while (!ReadLine(line))
        {
            WaitForLine();
        }
    }
    if (line.at(0) != '#')
    {
        // If we still haven't got a version, give up.
        return false;
    }
    // We have the version line, wait for OK line.
    if (!ReadReply(line, NoReply))
    {
        return false;
    }
    // All done.
    return true;
}

string GiraffMotor::FormatReply(const string& reply,
                                ReplyType type)
{
    istringstream ist(reply);
    ostringstream ost;
    ost << setfill('0') << setprecision(5) << fixed;
    if (type == BulkReply)
    {
        // Format bulk_data, which is a comma-separated
        // list of variables, where the name is separated
        // from the value by a colon.
        int next = ist.get();
        while (next != EOF)
        {
            // copy names and commas verbatim
            ost.put(next);
            if (next == ':')
            {
                // convert value
                FormatField(ost, ist);
            }
            next = ist.get();
        }
    }
    else if (type == SimpleReply)
    {
        // Format a normal single-value reply.
        FormatField(ost, ist);
    }
    else
    {
        // copy reply verbatim
        int next = ist.get();
        while (next != EOF)
        {
            ost.put(next);
            next = ist.get();
        }
    }
    return ost.str();
}

void GiraffMotor::FormatField(ostream& ost,
                              istream& ist)
{
    int next = ist.peek();
    char ch;
    unsigned u;

    switch (next)
    {
    case 'I':
        // hex-encoded 32-bit integer
        ist.get(ch); // 'I'
        ist.get(ch); // '*'
        ist >> hex >> u;
        ost << ToInt(u);
        break;
    case 'F':
        // hex-encoded 32-bit floating point
        ist.get(ch); // 'F'
        ist.get(ch); // '*'
        ist >> hex >> u;
        ost << ToFloat(u);
        break;
    default:
        // assume ASCII-encoded floating point,
        // copy unmodified
        while (next != EOF &&
               next != '\r' &&
               next != ',')
        {
            ch = ist.get();
            ost.put(ch);
            next = ist.peek();
        }
        break;
    }
}

unsigned GiraffMotor::ToInt(unsigned u)
{
    union {
        unsigned val;
        unsigned char d[4];
    } v;
    // convert byte order
    v.d[0] = u >> 24;
    v.d[1] = u >> 16;
    v.d[2] = u >> 8;
    v.d[3] = u;
    return v.val;
}

float GiraffMotor::ToFloat(unsigned u)
{
    union {
        float val;
        unsigned char d[4];
    } v;
    // convert byte order
    v.d[0] = u >> 24;
    v.d[1] = u >> 16;
    v.d[2] = u >> 8;
    v.d[3] = u;
    return v.val;
}
```

```
unsigned GiraffMotor::ToInt(const string& data)
{
    istringstream ist(data);
    char ch;
    unsigned u, i;

    switch (ist.peek())
    {
    case 'I':
        // hex-encoded 32-bit integer
        ist.get(ch); // 'I'
        ist.get(ch); // '*'
        ist >> hex >> i;
        u = ToInt(i);
        break;
    case 'F':
        // hex-encoded 32-bit floating point
        // (wrong type for this routine,
        // shouldn't happen)
        u = 0;
        break;
    default:
        // assume ASCII-encoded integer
        ist >> u;
        break;
    }
    return u;
}

float GiraffMotor::ToFloat(const string& data)
{
    istringstream ist(data);
    char ch;
    unsigned u;
    float f;

    switch (ist.peek())
    {
    case 'I':
        // hex-encoded 32-bit integer
        ist.get(ch); // 'I'
        ist.get(ch); // '*'
        ist >> hex >> u;
        f = ToInt(u);
        break;
    case 'F':
        // hex-encoded 32-bit floating point
        ist.get(ch); // 'F'
        ist.get(ch); // '*'
        ist >> hex >> u;
        f = ToFloat(u);
        break;
    default:
        // assume ASCII-encoded floating point
        ist >> f;
        break;
    }
    return f;
}

void GiraffMotor::ParseBulkData(const string& data)
{
     // save previous state
    m_lcang = m_cang;
    m_lcdis = m_cdis;
    m_lgvr = m_gvr;
    m_lcvg = m_cvg;
    m_lmode = m_cmode;
    m_lstamp = m_cstamp;
    // estimate time delta for new state
    LARGE_INTEGER current;
    QueryPerformanceCounter(&current);
    m_cstamp = current.QuadPart;
    if (m_not_first)
    {
        if (m_cstamp == m_lstamp)
        {
            // if no time has passed since last update
            // for some reason, do not update state yet,
            // as doing so could cause problems later
            // (should never happen, but just in case)
            m_timedelta = 0;
            return;
        }
        m_timedelta = (double)(m_cstamp - m_lstamp) / m_freq;
    }
    else
    {
        m_timedelta = 0;
        m_not_first = true;
    }
    // iterate through each name:value combination
    size_t cur_pos = 0;
    while (cur_pos < data.length())
    {
        // get the name
        size_t colon = data.find(':', cur_pos);
        if (colon == string::npos)
        {
            // not a valid entry, abort
            break;
        }
        string name = data.substr(cur_pos, colon-cur_pos);
        // get the value
        size_t comma = data.find(',', colon+1);
        if (comma != string::npos)
        {
            // comma found, more entries follow
            cur_pos = comma+1;
        }
        else
        {
            // no more commas, this is the last entry
            cur_pos = data.length();
            comma = data.length();
        }
        string value = data.substr(colon+1, comma-colon-1);

        // parse the entry
        if (name == "cang")
        {
            m_cang = -ToFloat(value);
        }
        else if (name == "cdis")
        {
            m_cdis = ToFloat(value);
        }
        else if (name == "gvr")
        {
            m_gvr = ToFloat(value);
        }
        else if (name == "cvg")
        {
            m_cvg = ToFloat(value);
        }
        else if (name == "mode")
        {
            m_cmode = ToInt(value);
        }
    }
}

void GiraffMotor::RunMotor()
```

```
{
    UpdatePosition();

    if (m_turnmode > 0 && m_brkdist != 0)
    {
        // Motor is currently moving...
        if (m_usrmotionspd == 0)
        {
            // ...and we're waiting for it to stop...
            if (m_cmode & MODE_MOVING)
            {
                // ...and it hasn't stopped yet.
                // Calculate braking distance from current speed,
                // plus 0.01s "reaction time" for sending
                // commands to the controller.
                double brake_dist = CalcMoveBrakeDist(m_curspd) +
                                    m_curspd * 0.01;
                if ((m_nextdis > 0 &&
                     m_nextpos > (m_cdis + brake_dist)) ||
                    (m_nextdis < 0 &&
                     m_nextpos < (m_cdis + brake_dist)))
                {
                    // The last command asked the controller
                    // to move too far. Preempt last command
                    // to make it stop ASAP.
                    CalcMoveStep(brake_dist);
                    SetParameter("p", m_nextpos);
                }
                return;
            }
            else
            {
                // ...and it has stopped.
                m_brkdist = 0;
            }
        }
        else
        {
            // ...and we want it to keep moving.
            if (m_nextdis == 0 ||
                m_usrmotionspd != m_curmotionspd ||
                m_usrturnspd != m_curturnspd)
            {
                // Got new command from user.
                CalcMove();
            }
            else if ((m_nextdis > 0 &&
                      m_nextpos < m_cdis + 2*m_brkdist) ||
                     (m_nextdis < 0 &&
                      m_nextpos > m_cdis + 2*m_brkdist))
            {
                // Renew move command to keep moving.
                CalcMoveStep(AHEAD_FACTOR*m_brkdist);
            }
            else
            {
                return;
            }
            SetParameter("cdp", m_gearpos);
            SetParameter("vgr", m_gearrate);
            SetParameter("vg", m_gear);
            SetParameter("p", m_nextpos);
            return;
        }
    }
    else if (m_turnmode < 0 && m_brkdist != 0)
    {
        // Motor is currently turning in place...
        if (m_usrturnspd == 0 ||
            m_usrmotionspd != 0)
        {
            // ...and we're waiting for it to stop.
            if (m_cmode & MODE_MOVING)
            {
                // ...and it hasn't stopped yet.
                // Calculate braking distance from current speed,
                // plus 0.01s "reaction time" for sending
                // commands to the controller.
                double brake_dist = CalcRotateBrakeDist(m_currot) +
                                    m_currot * 0.01;
                if ((m_nextdis > 0 &&
                     m_nextpos > (m_cang + brake_dist)) ||
                    (m_nextdis < 0 &&
                     m_nextpos < (m_cang + brake_dist)))
                {
                    // The last command asked the controller
                    // to move too far. Preempt last command
                    // to make it stop ASAP.
                    CalcRotateStep(brake_dist);
                    SetParameter("p", m_nextpos);
                }
                return;
            }
            else
            {
                // The motor has stopped.
                m_brkdist = 0;
            }
        }
        else
        {
            // ...and we want it to keep turning.
            if (m_nextdis == 0 ||
                m_usrturnspd != m_curturnspd)
            {
                // Got new command from user.
                CalcRotate();
            }
            else if ((m_nextdis > 0 &&
                      (m_nextpos - (m_cang + 2*m_brkdist)) < 0) ||
                     (m_nextdis < 0 &&
                      (m_nextpos - (m_cang + 2*m_brkdist)) > 0))
            {
                // Renew turn command to keep moving.
                CalcRotateStep(AHEAD_FACTOR*m_brkdist);
            }
            else
            {
                return;
            }
            SetParameter("p", m_nextpos);
            return;
        }
    }

    // If we get here, then the motor is idle.

    if (m_usrmotionspd != 0)
    {
        // Request to move.
        CalcMove();
        SetParameter("r", 1000);
        SetParameter("mode", m_absmode);
        SetParameter("a", m_accel);
        SetParameter("v", m_speed);
        SetParameter("cdp", m_gearpos);
        SetParameter("vgr", m_gearrate);
        SetParameter("vg", m_gear);
        SetParameter("p", m_nextpos);
    }
    else if (m_usrturnspd != 0)
    {
        // Request to turn in place.
        CalcRotate();
        SetParameter("r", 0);
        SetParameter("mode", m_absmode);
        SetParameter("a", m_accel);
        SetParameter("v", m_speed);
        SetParameter("p", m_nextpos);
    }
}

void GiraffMotor::CalcMove()
{
    m_curmotionspd = m_usrmotionspd;
    m_curturnspd = m_usrturnspd;

    // Set wheel speed
    m_speed = fabs(m_curmotionspd);
    if (m_speed == 0 || m_curturnspd == 0)
    {
        // Moving straight ahead.
        m_gear = 0;

        // Due to bugs in the motor controller,
        // just suddenly telling the controller to
        // take the gear ratio to zero using "cdp"
        // is problematic (the gear ratio jumps
        // and causes the motor to turn faster for
        // a while). Setting "vg" to zero makes no
        // appreciable difference. Setting "vgr" to
        // zero forces the ratio to zero instantenously,
        // with a horrible jerk that's probably not
        // good for the motors.

        m_gearrate = 0;
    }
    else
    {
        // Calculate the virtual gear ratio needed to
        // turn with the requested angular speed,
        // if we'll also be traveling forward
        // at the requrested overall speed.
        m_gear = m_curturnspd / (m_speed * turn_factor);
        if (m_gear < 0)
        {
            m_gearrate = -m_vgaccel;
        }
        else if (m_gear > 0)
        {
            m_gearrate = m_vgaccel;
        }
        else
        {
            m_gearrate = 0;
        }
    }
    // Calculate braking distance.
    m_brkdist = CalcMoveBrakeDist(m_curmotionspd);
    // Initiate motion.
    m_turnmode = 1;
#ifdef USE_ABSOLUTE_MODE
    m_absmode = MODE_ABSOLUTE;
#else
    m_absmode = 0;
#endif // USE_ABSOLUTE_MODE
    CalcMoveStep(AHEAD_FACTOR*m_brkdist);
}

double GiraffMotor::CalcMoveBrakeDist(double spd)
{
    double brake_time = fabs(spd) / m_accel;
    return spd * brake_time / 2;
}

void GiraffMotor::CalcMoveStep(double dis)
```

```
{
    m_nextdis = dis;
    if (m_absmode)
    {
        m_nextpos = m_cdis + m_nextdis;
    }
    else
    {
        m_nextpos = m_nextdis;
        m_cdis = 0;
    }
#if 0
    if (m_gear != 0)
    {
        m_gearpos = m_nextpos;
    }
    else
    {
        //m_gearpos = m_cdis + m_curspd * 0.1;
    }
#else
    m_gearpos = m_nextpos;
#endif
}

void GiraffMotor::CalcRotate()
{
    m_curmotionspd = m_usrmotionspd;
    m_curturnspd = m_usrturnspd;

    // Calculate wheel speed.
    m_speed = fabs(m_curturnspd) / turn_factor;
    // Calculate braking distance.
    m_brkdist = CalcRotateBrakeDist(m_curturnspd);
    // Initiate motion.
    m_turnmode = -1;
#ifdef USE_ABSOLUTE_MODE
    m_absmode = MODE_ABSOLUTE;
#else
    m_absmode = 0;
#endif // USE_ABSOLUTE_MODE
    CalcRotateStep(AHEAD_FACTOR*m_brkdist);
}

double GiraffMotor::CalcRotateBrakeDist(double rot)
{
    double brake_time = fabs(rot) / (m_accel * turn_factor);
    return rot * brake_time / 2;
}

void GiraffMotor::CalcRotateStep(double ang)
{
    m_nextdis = ang;
    if (m_absmode)
    {
        m_nextpos = m_cang + m_nextdis;
    }
    else
    {
        m_nextpos = m_nextdis;
        m_cang = 0;
    }
}

void GiraffMotor::UpdatePosition()
{
    if (m_turnmode == 0 ||
        m_brkdist == 0)
    {
        // standing still
        return;
    }
    if (m_turnmode < 0)
    {
        // turning in place
        double turndelta = m_cang - m_lcang;
        m_currot = -m_gvr;
        m_curdir += turndelta;
        return;
    }

    if (m_timedelta == 0)
    {
        // no time has passed since last update
        return;
    }

    // moving in a straight or curved line

    double distdelta = m_cdis - m_lcdis;
#ifdef GVR_IS_LEFT
    if (m_cvg == 0)
    {
        // straight ahead
        m_curspd = m_gvr;
    }
    else if (m_cvg != 1)
    {
        // if gvr is the speed of the left wheel,
        // calculate the overall speed given the
        // current gear ratio
        m_curspd = m_gvr * (1 - m_cvg);
    }
    else
    {
        // ideally we should never let the gear
        // ratio become as large as 1 or -1,
        // but if it happens, make an estimate
        m_curspd = distdelta / m_timedelta;
    }
#else
    m_curspd = m_gvr;
#endif // GVR_IS_LEFT
    m_currot = m_cvg * turn_factor;

    // In principle, we should use integration techniques
    // to calculate the current position, given the motor
    // feedback and the known behaviour of our commands,
    // including the expected motion envelope.
    // However, even then the results would probably not
    // match physical reality very well, so these
    // approximations are probably good enough, as long
    // as the motor state is updated often enough.
    double avgvg = (m_cvg + m_lcvg) / 2;
    double avgrot = avgvg * turn_factor;
    double turndelta = avgrot * distdelta;
    double avgdir = m_curdir + (turndelta / 2);
    m_curdir += turndelta;
    m_curx += distdelta * cos(avgdir * M_PI / 180);
    m_cury += distdelta * sin(avgdir * M_PI / 180);
}

void GiraffMotor::ShowPosition()
{
    ostringstream ost;
    ost << fixed << setprecision(2)
        << internal << setfill('0');
    ost << "X=" << setw(6) << m_curx
        << ",Y=" << setw(6) << m_cury
        << ",H=" << setw(6) << fix_degrees(m_curdir);
    m_win->SetPositionInfo(ost.str());
}
```

```
const double GiraffMotorSim::turn_factor = TURN_FACTOR;
// The standard tilt angle returned to when homing the head.
const double GiraffMotorSim::default_tilt = 0.0872664;

GiraffMotorSim::GiraffMotorSim(GiraffMotor* ctl) :
    m_ctl(ctl), m_bufcount(0),
    m_homing(0), m_tilt(default_tilt),
    m_cang(0.0), m_cdis(0.0), m_cvg(0.0),
    m_vang(0.0), m_vdis(0.0), m_gvr(0.0),
    m_startc(0), m_stopc(0),
    m_updc(0), m_downdc(0),
    m_startdv(0.0), m_peakdv(0.0), m_rampda(0.0),
    m_refdp(0.0), m_updp(0.0), m_downdp(0.0), m_stopdp(0.0),
    m_refap(0.0), m_upap(0.0), m_downap(0.0), m_stopap(0.0),
    m_upgc(0), m_downgc(0), m_stopgc(0),
    m_startgr(0.0), m_peakgr(0.0), m_rupgr(0.0), m_rdowngr(0.0),
    m_stopgr(0.0), m_downgd(0.0)
{
    // clear states
    memset(&m_buf, 0, sizeof(m_buf));
    // set defaults
    m_buf[0].v = 0.6; // 0.6 m/s
    m_buf[0].a = 0.6; // 0.6 m/s^2
    m_buf[0].vg = 1;
    m_buf[0].vgr = 0.4; // FIXME, what's the actual default?
    // get timer frequency
    LARGE_INTEGER freq;
    QueryPerformanceFrequency(&freq);
    m_freq = freq.QuadPart;
    // signal readiness
    SimulateReply("# Giraf Simulator");
}

GiraffMotorSim::~GiraffMotorSim()
{
}

void GiraffMotorSim::StartStraight(double dist,
                                   double start_pos,
                                   double start_spd,
                                   double cdp,
                                   double start_vg)
{
    Move& cur = m_buf[0];
    if (cur.mode & MODE_ABSOLUTE)
    {
        dist -= start_pos;
        cdp -= start_pos;
    }
    else
    {
        start_pos = 0;
        m_cdis = 0;
        m_cang = 0;
    }
    if (start_spd == 0)
    {
        // when starting from a full stop, assume the initial
        // virtual gear ratio to be zero
        start_vg = 0;
    }
    // Get the sign of the desired motion here,
    // so we can calculate the motion profile
    // using positive numbers, then apply the
    // proper sign afterwards.
    // (Interpret dist == 0 as "stop ASAP",
    // taking sign from current velocity instead.)
    int sign = (dist != 0) ?
                ((dist >= 0) ? 1 : -1) :
                ((start_spd >= 0) ? 1 : -1);
    // but let start_speed be negative if we're
    // initially moving in the wrong direction.
    double start_speed = sign * start_spd;
    double total_dist = fabs(dist);
    double peak_speed = cur.v;
    double accel = cur.a;
    double ramp_up_time = (peak_speed - start_speed) / accel;
    double ramp_up_dist = (peak_speed + start_speed) * ramp_up_time / 2;
    double ramp_down_time = peak_speed / accel;
    double ramp_down_dist = peak_speed * ramp_down_time / 2;
    double cruise_dist = total_dist - ramp_up_dist - ramp_down_dist;
    double cruise_time = cruise_dist / peak_speed;
    bool overshoot = false;
    if (cruise_dist < 0)
    {
        // Short move, won't get to maximum speed.
        // In case we're already moving, calculate the
        // ramp-up distance we would have needed to get
        // up to the current velocity from standing still.
        double prev_dist = start_speed * start_speed / accel / 2;
        // Since the total ramp-up and ramp-down must
        // be of equal lengths, find how long the ramps
        // must be to cover the required distance
        ramp_down_dist = (prev_dist + total_dist) / 2;
        if (ramp_down_dist >= total_dist &&
            start_speed >= 0)
        {
            // Current speed is too high to stop within
            // the specified distance.
#ifdef SIM_OVERSHOOT_FIX
            // We'll try to stop as soon as possible
            // and reverse back. Find how long the ramps
            // need to be in this case. Also invert
            // signs since we want to reverse.
            ramp_down_dist = (total_dist - prev_dist) / 2;
            accel = -accel;
            // The ramp-up includes the reversal of direction.
            peak_speed = -sqrt(2*accel*ramp_down_dist);
            ramp_up_dist = total_dist - ramp_down_dist;
            ramp_up_time = (peak_speed - start_speed) / accel;
            ramp_down_time = peak_speed / accel;
#else
            // Alternatively, just stop as soon as possible,
            // don't bother to reverse back.
            peak_speed = start_speed;
            ramp_up_dist = 0;
            ramp_up_time = 0;
            ramp_down_dist = prev_dist;
            ramp_down_time = peak_speed / accel;
            overshoot = true;
#endif // SIM_OVERSHOOT_FIX
        }
        else
        {
            // If start_speed is negative, the ramp-up
            // will include the reversal of direction.
            peak_speed = sqrt(2*accel*ramp_down_dist);
            ramp_up_dist = total_dist - ramp_down_dist;
            ramp_up_time = (peak_speed - start_speed) / accel;
            ramp_down_time = peak_speed / accel;
        }
        cruise_dist = 0;
        cruise_time = 0;
    }
    // set velocities and accelerations
    m_startdv = sign * start_speed;
    m_peakdv = sign * peak_speed;
    m_rampda = sign * accel;
    // calculate timestamps for velocity envelope
    double time_step = ramp_up_time;
    m_startc = m_lastc;
    m_updc = m_startc + time_step * m_freq;
    time_step += cruise_time;
    m_downdc = m_startc + time_step * m_freq;
    time_step += ramp_down_time;
    m_stopc = m_startc + time_step * m_freq;
    // calculate positions for velocity envelope
    m_refdp = start_pos;
    m_updp = sign * ramp_up_dist;
    m_downdp = m_updp + sign * cruise_dist;
    m_stopdp = (!overshoot) ? dist : (m_downdp + sign * ramp_down_dist);

    // zero out angle envelope
    m_upac = m_startc;
    m_downac = m_stopc;
    m_startav = 0;
    m_peakav = 0;
    m_rampaa = 0;
    m_refap = 0; // m_cang?
    m_upap = 0;
    m_downap = 0;
    m_stopap = 0;

    // Calculate virtual gear ratio envelope
    // (This calculates the "ideal" envelope, the way it's
    // documented. However, the real controller doesn't seem
    // to be quite as ideal, due to bugs.)
#ifdef SIM_CURVED
    if ((cur.r >= 50 && cdp != 0 && cur.vg != 0) ||
        start_vg != 0)
    {
        double ramp_down_start = ramp_up_time + cruise_time;
        double total_time = ramp_down_start + ramp_down_time;
        int vg_sign = (cur.vgr != 0) ?
                      ((cur.vgr >= 0) ? 1 : -1) :
                      ((start_vg >= 0) ? 1 : -1);
        double vg_start = vg_sign * sign * start_vg;
        double vg_peak = vg_sign * cur.vg;
        double vg_accel = vg_sign * cur.vgr;
        int vg_up_sign = 1, vg_down_sign = -1;
        double vg_ramp_down_pos = sign * cdp;
        if (vg_ramp_down_pos > total_dist)
        {
            vg_ramp_down_pos = total_dist;
        }
        double vg_ramp_up_dist;
        if (vg_ramp_down_pos <= 0 ||
            vg_peak == 0 ||
            accel < 0)
        {
            // No ramp-up, start ramping down immediately.
            vg_ramp_up_dist = 0;
            vg_peak = vg_start;
            vg_ramp_down_pos = 0;
        }
        else if (vg_peak < vg_start)
        {
            // Ramp-up is actually a ramp-down
            // to a lower gear ratio.
            vg_ramp_up_dist = (vg_start - vg_peak) / vg_accel;
            vg_up_sign = -1;
        }
        else
        {
            // Normal ramp-up.
            vg_ramp_up_dist = (vg_peak - vg_start) / vg_accel;
        }
        if (vg_ramp_up_dist > vg_ramp_down_pos)
        {
            // Short move, won't get to maximum gear ratio.
            // Calculate how far we would get.
            vg_ramp_up_dist = vg_ramp_down_pos;
            vg_peak = vg_start + vg_up_sign * vg_ramp_down_pos * vg_accel;
        }
        // Calculate ramp-up time
        double vg_ramp_up_time;
```

```
if (vg_ramp_up_dist > total_dist)
{
    // won't reach peak gear ratio in this move
    vg_ramp_up_time = total_time;
    vg_peak = vg_start + vg_up_sign * total_dist * vg_accel;
}
else
{
    vg_ramp_up_time = TimeFromPosition(vg_ramp_up_dist,
                                      ramp_up_time,
                                      cruise_time,
                                      ramp_down_time,
                                      ramp_up_dist,
                                      cruise_dist,
                                      ramp_down_dist,
                                      accel,
                                      start_speed,
                                      peak_speed);
}
// Calculate ramp-down start time
double vg_ramp_down_start;
vg_ramp_down_start = TimeFromPosition(vg_ramp_down_pos,
                                     ramp_up_time,
                                     cruise_time,
                                     ramp_down_time,
                                     ramp_up_dist,
                                     cruise_dist,
                                     ramp_down_dist,
                                     accel,
                                     start_speed,
                                     peak_speed);
// Calculate ramp-down time
double vg_ramp_down_dist;
if (vg_peak < 0)
{
    // Ramp-down is actually a ramp-up
    // from a negative gear ratio.
    vg_ramp_down_dist = -vg_peak / vg_accel;
    vg_down_sign = 1;
}
else
{
    // Normal ramp-down.
    vg_ramp_down_dist = vg_peak / vg_accel;
}
double vg_stop = 0, vg_stop_pos;
vg_stop_pos = vg_ramp_down_pos + vg_ramp_down_dist;
double vg_ramp_down_time;
if (accel < 0)
{
    // there's a momentary stop during reversal
    double prev_dist = total_dist - 2 * ramp_up_dist;
    double stop_dist = vg_stop_pos - vg_ramp_down_pos;
    if (prev_dist < stop_dist)
    {
        // force gear ratio to zero when we reverse
        vg_ramp_down_dist = -prev_dist;
        vg_ramp_down_time = -start_speed / accel;
        vg_stop_pos = vg_ramp_down_pos + vg_ramp_down_dist;
    }
    else
    {
        // gear ratio reaches zero before the stop
        double speed = sqrt(start_speed*start_speed +
                            2*accel*vg_ramp_down_dist);
        vg_ramp_down_time = (speed - start_speed) / accel;
    }
}
else if (vg_ramp_up_dist >= total_dist)
{
    // won't start ramp-down while moving
    vg_ramp_down_dist = 0;
```

```
            vg_ramp_down_time = 0;
            vg_stop_pos = total_dist;
        }
        else if (vg_stop_pos > total_dist)
        {
            // won't reach zero gear ratio while moving
            vg_ramp_down_dist = total_dist - vg_ramp_down_pos;
            vg_ramp_down_time = total_time - vg_ramp_down_start;
            vg_stop_pos = total_dist;
        }
        else
        {
            vg_ramp_down_time = TimeFromPosition(vg_stop_pos,
                                                 ramp_up_time,
                                                 cruise_time,
                                                 ramp_down_time,
                                                 ramp_up_dist,
                                                 cruise_dist,
                                                 ramp_down_dist,
                                                 accel,
                                                 start_speed,
                                                 peak_speed)
                                - vg_ramp_down_start;
        }

        // calculate timestamps for gear ratio envelope
        double vg_ramp_stop = vg_ramp_down_start + vg_ramp_down_time;
        m_upgc = m_startc + vg_ramp_up_time * m_freq;
        m_downgc = m_startc + vg_ramp_down_start * m_freq;
        m_stopgc = m_startc + vg_ramp_stop * m_freq;
        // calculate gear ratios
        m_startgr = start_vg;
        m_peakgr = vg_sign * sign * vg_peak;
        m_rupgr = vg_up_sign * vg_sign * sign * vg_accel;
        m_rdowngr = vg_down_sign * vg_sign * sign * vg_accel;
        m_stopgr = vg_sign * sign * vg_stop;
        m_downgd = vg_ramp_down_pos;
    }
    else
#endif // SIM_CURVED
    {
        m_upgc = m_startc;
        m_downgc = m_startc;
        m_stopgc = m_startc;
        m_startgr = 0;
        m_peakgr = 0;
        m_rupgr = 0;
        m_rdowngr = 0;
        m_stopgr = 0;
        m_downgd = 0;
    }
}

void GiraffMotorSim::StartRotate(double degrees,
                                 double start_angle,
                                 double start_spd)
{
    Move& cur = m_buf[0];
    if (cur.mode & MODE_ABSOLUTE)
    {
        degrees -= start_angle;
    }
    else
    {
        start_angle = 0;
        m_cdis = 0;
        m_cang = 0;
    }
    // Calculate the distance the wheels must travel.
    double dist = degrees / turn_factor;
    // The wheels accelerate using the same parameters
    // as in a straight-line move. Calculate motion.
    StartStraight(dist, 0, start_spd / turn_factor, 0, 0);
    if (cur.r < 0)
    {
        // If r is negative, the motion don't seem to
        // get converted back to angles, but are reported
        // to the app as distances traveled by the wheels.
        return;
    }
    bool overshoot = (m_stopdp != dist);
    // Convert calculated positions to angles.
    m_upac = m_updc;
    m_downac = m_downdc;
    m_startav = m_startdv * turn_factor;
    m_peakav = m_peakdv * turn_factor;
    m_rampaa = m_rampda * turn_factor;
    m_refap = start_angle;
    m_upap = m_updp * turn_factor;
    m_downap = m_downdp * turn_factor;
    m_stopap = (!overshoot) ? degrees : (m_stopdp * turn_factor);
    // Clear positions and velocities,
    // since we're staying in place.
    m_updc = m_startc;
    m_downdc = m_stopc;
    m_startdv = 0;
    m_peakdv = 0;
    m_rampda = 0;
    m_refdp = 0; // m_cdis?
    m_updp = 0;
    m_downdp = 0;
    m_stopdp = 0;
}

double GiraffMotorSim::TimeFromPosition(double pos,
                                        double ramp_up_time,
                                        double cruise_time,
                                        double ramp_down_time,
                                        double ramp_up_dist,
                                        double cruise_dist,
                                        double ramp_down_dist,
                                        double accel,
                                        double start_speed,
                                        double peak_speed)
{
    double cruise_start = ramp_up_time;
    double ramp_down_start = cruise_start + cruise_time;
    double total_time = ramp_down_start + ramp_down_time;
    double cruise_pos = ramp_up_dist;
    double ramp_down_pos = cruise_pos + cruise_dist;
    double stop_pos = ramp_down_pos + ramp_down_dist;
    if (pos <= 0)
    {
        // immediately
        return 0;
    }
    else if (pos < cruise_pos)
    {
        // during ramp-up
        double dist = pos;
        double speed = sqrt(start_speed*start_speed +
                            2*accel*dist);
        return (speed - start_speed) / accel;
    }
    else if (pos < ramp_down_pos)
    {
        // during cruise
        double dist = pos - cruise_pos;
        return cruise_start + dist / peak_speed;
    }
    else if (pos < stop_pos)
    {
        // during ramp-down
        double dist = pos - ramp_down_pos;
        double speed = sqrt(peak_speed*peak_speed -
                            2*accel*dist);
        return ramp_down_start + (peak_speed - speed) / accel;
    }
    else
    {
        // never
        return total_time;
    }
}

void GiraffMotorSim::UpdateMotion()
{
    LARGE_INTEGER current;
    QueryPerformanceCounter(&current);
    LONGLONG now = current.QuadPart;
    while (m_bufcount != 0)
    {
        if (now >= m_stopc)
        {
            // Move complete.
            m_cdis = m_refdp + m_stopdp;
            m_cang = m_refap + m_stopap;
            m_cvg = m_stopgr;
            m_vdis = 0;
            m_vang = 0;
            m_gvr = 0;
            m_lastc = m_stopc;
            EndMotion();
            continue;
        }
        // Interpolate distance part of profile
        if (now >= m_downdc)
        {
            // ramping down.
            double time_delta = (double)(now - m_downdc) / m_freq;
            double velocity = m_peakdv - m_rampda * time_delta;
            double dist = (m_peakdv + velocity) * time_delta / 2;
            m_cdis = m_refdp + m_downdp + dist;
            m_vdis = velocity;
        }
        else if (now >= m_updc)
        {
            // cruising.
            double time_delta = (double)(now - m_updc) / m_freq;
            double velocity = m_peakdv;
            double dist = m_peakdv * time_delta;
            m_cdis = m_refdp + m_updp + dist;
            m_vdis = velocity;
        }
        else
        {
            // ramping up.
            double time_delta = (double)(now - m_startc) / m_freq;
            double velocity = m_startdv + m_rampda * time_delta;
            double dist = (m_startdv + velocity) * time_delta / 2;
            m_cdis = m_refdp + dist;
            m_vdis = velocity;
        }
        // Interpolate gear ratio part of profile
        if (now >= m_stopgc)
        {
            m_cvg = m_stopgr;
        }
        else if (now >= m_downgc)
        {
            // ramping down.
            double dist_delta = abs(m_cdis - m_refdp) - m_downgd;
            m_cvg = m_peakgr + m_rdowngr * dist_delta;
        }
        else if (now >= m_upgc)
        {
            // cruising.
            m_cvg = m_peakgr;
        }
        else
        {
            // ramping up.
            double dist_delta = abs(m_cdis - m_refdp);
            m_cvg = m_startgr + m_rupgr * dist_delta;
        }

        // Interpolate angular part of profile
        if (now >= m_downac)
        {
            // ramping down.
            double time_delta = (double)(now - m_downac) / m_freq;
            double velocity = m_peakav - m_rampaa * time_delta;
            double dist = (m_peakav + velocity) * time_delta / 2;
            m_cang = m_refap + m_downap + dist;
            m_vang = velocity;
        }
        else if (now >= m_upac)
        {
            // cruising.
            double time_delta = (double)(now - m_upac) / m_freq;
            double velocity = m_peakav;
            double dist = m_peakav * time_delta;
            m_cang = m_refap + m_upap + dist;
            m_vang = velocity;
        }
        else
        {
            // ramping up.
            double time_delta = (double)(now - m_startc) / m_freq;
            double velocity = m_startav + m_rampaa * time_delta;
            double dist = (m_startav + velocity) * time_delta / 2;
            m_cang = m_refap + dist;
            m_vang = velocity;
        }
        if (m_buf[0].r != 0)
        {
#ifdef GVR_IS_LEFT
            m_gvr = m_vdis / (1 - m_cvg);
#else
            m_gvr = m_vdis;
#endif // GVR_IS_LEFT
        }
        else
        {
            m_gvr = -m_vang;
        }
        break;
    }
    m_lastc = now;
}

void GiraffMotorSim::StartMotion()
{
    if (m_bufcount == 0)
    {
        // nothing to do
        return;
    }
    Move& cur = m_buf[0];
    if (cur.mode & MODE_MOVING)
    {
        // already started
        return;
    }
    cur.mode |= MODE_MOVING;
    if (cur.r > 0)
    {
        StartStraight(cur.p, m_cdis, m_vdis,
                      cur.cdp, m_cvg);
    }
    else
    {
        StartRotate(cur.p, m_cang, m_vang);
    }
}

void GiraffMotorSim::EndMotion()
{
    unsigned n;
    if (m_bufcount == 0)
    {
        // nothing to do
        return;
    }
    // shift next requests into place,
    // replacing completed request
    for (n=0; n<m_bufcount; n++)
    {
        m_buf[n] = m_buf[n+1];
    }
    m_bufcount--;
    // start next request, if any
    StartMotion();
}

bool GiraffMotorSim::QueueMotion()
{
    if (m_bufcount >= GIRAFF_BUFFERS)
    {
        // out of buffers
        return false;
    }
    unsigned mask = MODE_ABSOLUTE;
    unsigned mode = m_buf[m_bufcount].mode;
    m_buf[m_bufcount].mode = mode & mask;
    if ((m_bufcount == 0) ||
        (mode & MODE_BUFFERED))
    {
        // initial state for next request
        m_buf[m_bufcount+1] = m_buf[m_bufcount];
        // start current request
        m_bufcount++;
    }
    else
    {
        // remove buffered requests
        if (m_bufcount > 1)
        {
            // initial state for next request
            m_buf[1] = m_buf[m_bufcount];
            m_bufcount = 1;
        }
        // preempt current move
        m_buf[0] = m_buf[m_bufcount];
    }
    // start sequence
    StartMotion();
    return true;
}

bool GiraffMotorSim::QueueUndock(double dist)
{
    if (m_bufcount >= GIRAFF_BUFFERS-1)
    {
        // out of buffers
        return false;
    }
    // initial state for next request
    m_buf[m_bufcount+1] = m_buf[m_bufcount];
    m_buf[m_bufcount+2] = m_buf[m_bufcount];
    // request reversing (in straight line)
    m_buf[m_bufcount].mode = 0;
    m_buf[m_bufcount].r = 1;
    m_buf[m_bufcount].p = -dist;
    m_buf[m_bufcount].vg = 0;
    m_buf[m_bufcount].vgr = 0;
    m_buf[m_bufcount].cdp = 0;
    m_bufcount++;
    // request rotating in-place
    m_buf[m_bufcount].mode = 0;
    m_buf[m_bufcount].r = 0;
    m_buf[m_bufcount].p = 180;
    m_buf[m_bufcount].vg = 0;
    m_buf[m_bufcount].vgr = 0;
    m_buf[m_bufcount].cdp = 0;
    m_bufcount++;
    // start sequence
    StartMotion();
    return true;
}

void GiraffMotorSim::SimulateLag(unsigned bytes)
{
    // Since the serial port is configured for
    // 115200 bps, and each character takes 10 bits
    // (1 start bit, 8 data bits, and 1 stop bit),
    // it can only transfer 11520 characters/second.
    // Since each command transfers something like
    // 30-40 characters, this lag could affect timing
    // by several milliseconds, so we'll simulate it
    // here, just in case.
    // Calculate microsecond wait.
    DWORD us = (bytes * 1000000) / 11520;
    // Convert to milliseconds.
    DWORD ms = us / 1000;
    // Round up.
    if ((us % 1000) >= 500)
    {
        ms++;
    }
    // Do the wait.
    if (ms)
    {
        Sleep(ms);
    }
}

void GiraffMotorSim::SimulateReply(const string& reply)
{
    string out;
    if (!reply.empty())
    {
        out = reply + "\r\nOK >\r\n";
    }
    else
    {
        out = "OK >\r\n";
    }
    // wait the milliseconds it would take to
    // receive the reply (including the "OK" line)
    SimulateLag(out.length());
    // simulate the reply
    m_ctl->AddReply(out);
}

void GiraffMotorSim::SimulateCommand(const string& cmd)
{
    ostringstream rst;
    istringstream ist(cmd);
    string op;

    // wait the milliseconds it would take to
    // transmit the command
    //SimulateLag(cmd.length());
    // update simulation state
    UpdateMotion();

    // set default reply format
    rst << setfill('0') << setprecision(5) << fixed;
    // parse command
    ist >> op;
    if (op == "set")
    {
        // parse Set command
        Move& next = m_buf[m_bufcount];
        string par;
        ist >> par;
        if (par == "v")
        {
            ist >> next.v;
            Output(rst, next.v);
        }
        else if (par == "r")
        {
            ist >> next.r;
            Output(rst, next.r);
        }
        else if (par == "a")
        {
            ist >> next.a;
            Output(rst, next.a);
        }
        else if (par == "p")
        {
            ist >> next.p;
            if (QueueMotion())
            {
                Output(rst, next.p);
            }
            else
            {
                rst << "ERROR: Queue rollover";
            }
        }
        else if (par == "vg")
        {
            ist >> next.vg;
            Output(rst, next.vg);
        }
        else if (par == "vgr")
        {
            ist >> next.vgr;
            Output(rst, next.vgr);
        }
        else if (par == "cdp")
        {
            ist >> next.cdp;
            Output(rst, next.cdp);
        }
        else if (par == "mode")
        {
            // Only the lower 4 bits can be set.
            unsigned mask = 0xf;
            unsigned mode;
            ist >> mode;
            next.mode = (next.mode & ~mask) |
                        (mode & mask);
            Output(rst, next.mode);
        }
        else if (par == "undock")
        {
            double dist;
            ist >> dist;
            if (QueueUndock(dist))
```

```
            {
                Output(rst, dist);
            }
            else
            {
                rst << "ERROR: Queue rollover";
            }
        }
        else if (par == "tilt_angle_from_home")
        {
            ist >> m_tilt;
            Output(rst, m_tilt);
        }
        else
        {
            rst << "Unknown name: " << par;
        }
    }
    else if (op == "get")
    {
        // parse Get command
        Move& cur = m_buf[0];
        Move& next = m_buf[m_bufcount];
        string par;
        ist >> par;
        if (par == "v")
        {
            Output(rst, next.v);
        }
        else if (par == "r")
        {
            Output(rst, next.r);
        }
        else if (par == "a")
        {
            Output(rst, next.a);
        }
        else if (par == "p")
        {
            Output(rst, next.p);
        }
        else if (par == "vg")
        {
            Output(rst, next.vg);
        }
        else if (par == "vgr")
        {
            Output(rst, next.vgr);
        }
        else if (par == "cdp")
        {
            Output(rst, next.cdp);
        }
        else if (par == "cvg")
        {
            Output(rst, m_cvg);
        }
        else if (par == "mode")
        {
            unsigned c_mask = MODE_ESTOP | MODE_MOVING;
            unsigned n_mask = MODE_ABSOLUTE;
            unsigned mode = (cur.mode & c_mask) |
                            (next.mode & n_mask);
            Output(rst, mode);
        }
        else if (par == "tilt_homing_state")
        {
            Output(rst, m_homing);
        }
        else if (par == "tilt_angle_from_home")
        {
            Output(rst, m_tilt);
        }
        else if (par == "but0")
        {
            rst << "0";
        }
        else if (par == "but1")
        {
            rst << "0";
        }
        else if (par == "dial")
        {
            rst << "0";
        }
        else if (par == "button_data")
        {
            rst << "but0:0,but1:0,dial:0";
        }
        else if (par == "bulk_data")
        {
            rst << "cang:";
            Output(rst, -m_cang);
            rst << ",cdis:";
            Output(rst, m_cdis);
            rst << ",gvr:";
            Output(rst, m_gvr);
//                << ",tilt_angle_from_home:" << m_tilt
//                << ",imdl:0"
//                << ",imdr:0"
            rst << ",cvg:";
            Output(rst, m_cvg);
            rst << ",mode:";
            Output(rst, cur.mode);
        }
        else
        {
            rst << "Unknown name: " << par;
        }
    }
    else if (op == "home")
    {
        // no reply
    }
    else
    {
        rst << "Unknown name: " << op;
    }
    SimulateReply(rst.str());
}

void GiraffMotorSim::Output(ostream& out, double val)
{
#if 1
    // The controller seems to send floats using a hex
    // encoding of the binary representation of a 32-bit
    // floating-point register. Reproduce it here.
    union {
        float val;
        unsigned char d[4];
    } v;
    v.val = val;
    out << "F*" << hex;
    for (unsigned n=0; n<4; n++)
    {
        unsigned u = v.d[n];
        out << setw(2) << u;
    }
#else
    out << val;
#endif
}

void GiraffMotorSim::Output(ostream& out, unsigned val)
```

```
{
#if 1
    // The controller seems to send integers using a hex
    // encoding that has the least-significant byte first.
    // Reproduce it here.
    union {
        unsigned val;
        unsigned char d[4];
    } v;
    v.val = val;
    out << "I*" << hex;
    for (unsigned n=0; n<4; n++)
    {
        unsigned u = v.d[n];
        out << setw(2) << u;
    }
#else
    out << val;
#endif
}
```

B.3. GiraffCamera.hpp

```
#ifndef GIRAFFCAMERA_HPP
#define GIRAFFCAMERA_HPP

#include "DisplayWindow.hpp"

#include <opencv2/highgui/highgui.hpp>

#define CAM_REC_BUFFERS 8

class GiraffCamera
{
public:
    GiraffCamera(DisplayWindow* win);
    ~GiraffCamera();
    bool Start(int width=0, int height=0);
    void Stop();
    bool Grab(cv::Mat& frame);
    bool StartRecord(const std::string& name);
    void StopRecord();
    bool StartPlayback(const std::string& name);
    void StopPlayback();

private:
    DisplayWindow* m_win;
    cv::VideoWriter m_vrec;
    cv::VideoCapture m_vplay;
    bool m_sim, m_rec, m_play, m_eof;
    cv::Mat m_frame;
    // for recording thread
    cv::VideoCapture m_vcap;
    HANDLE m_recthread;
    HANDLE m_recfstart, m_recfdone;
#ifdef CAM_REC_BUFFERS
    cv::Mat m_recbuf[CAM_REC_BUFFERS];
    unsigned m_recpos;
#endif
    void SetCameraInfo();
    static DWORD WINAPI RecThread(LPVOID param);
};

#endif // GIRAFFCAMERA_HPP
```

B.4. GiraffCamera.cpp

```
#include "GiraffCamera.hpp"

#include <windows.h>

#include <sstream>
#include <iomanip>

#define CAM_DEVICE 0

#define TEST_INPUT "D:/Giraff/OpenCV/Source/samples/gpu/768x576.avi"

using namespace std;
using namespace cv;

GiraffCamera::GiraffCamera(DisplayWindow* win) :
    m_win(win), m_sim(false), m_rec(false),
    m_play(false), m_eof(false)
{
}

GiraffCamera::~GiraffCamera()
{
    StopRecord();
    StopPlayback();
    Stop();
}

bool GiraffCamera::Start(int width, int height)
{
#ifdef CAM_DEVICE
    m_vcap.open(CAM_DEVICE);
    if (!m_vcap.isOpened())
#endif
    {
        // Could not open real camera,
        // load prerecorded video instead,
        // so the rest of the program
        // can still be used.
        m_vcap.open(TEST_INPUT);
        if (!m_vcap.isOpened())
        {
            return false;
        }
        if (!m_sim)
        {
            m_win->PrintLeft("Loaded test video");
            m_sim = true;
        }
    }

    // request resolution
    if (!m_sim && width && height)
    {
        m_vcap.set(CV_CAP_PROP_FRAME_WIDTH,  width);
        m_vcap.set(CV_CAP_PROP_FRAME_HEIGHT, height);
    }

    // show actual resolution on display
    SetCameraInfo();

    return true;
}

void GiraffCamera::Stop()
{
    m_vcap.release();
}

bool GiraffCamera::Grab(Mat& frame)
```

```
{
    if (m_rec)
    {
        // if we're recording, wait for recording
        // thread to finish encoding previous frame
        WaitForSingleObject(m_recfdone, INFINITE);
    }
    if (m_play)
    {
        if (m_eof)
        {
            // playback already complete
            return false;
        }
        // get next frame from playback
        else if (!m_vplay.read(m_frame))
        {
            // playback complete
            m_eof = true;
            return false;
        }
    }
    // get next frame from camera or video
    else if (!m_vcap.read(m_frame))
    {
        if (!m_sim)
        {
            // camera failure
            return false;
        }
        // end of video, rewind
        Stop();
        Start();
        if (!m_vcap.read(m_frame))
        {
            // give up
            return false;
        }
    }
    if (m_rec)
    {
        // if we're recording, tell the recording
        // thread that we have a new frame
#ifdef CAM_REC_BUFFERS
        m_recbuf[m_recpos] = m_frame.clone();
        m_recpos = (m_recpos + 1) % CAM_REC_BUFFERS;
        LONG sem_count = CAM_REC_BUFFERS;
        // clear event before ReleaseSemaphore
        // to avoid race conditions (we can
        // set it again afterwards)
        ResetEvent(m_recfdone);
        if (ReleaseSemaphore(m_recfstart, 1, &sem_count))
        {
            sem_count += 1;
            if (sem_count < CAM_REC_BUFFERS)
            {
                // still room for more frames,
                // so set event again
                SetEvent(m_recfdone);
            }
        }
        else
        {
            // if the synchronization stuff works,
            // we should never get here
            m_win->PrintLeft("Semaphore release failed");
        }
#else
        SetEvent(m_recfstart);
#endif
    }
    // could display frame here,
    // but we'll leave it to GiraffNav
    //m_win->Show(m_frame);
    // return captured frame
    frame = m_frame;
    return true;
}

bool GiraffCamera::StartRecord(const string& name)
{
    if (m_rec)
    {
        StopRecord();
    }
    //int fourcc = CV_FOURCC_PROMPT;
    // Lossy codecs listed at
    // http://opencv.willowgarage.com/wiki/documentation/cpp/highgui/VideoWriter
    //int fourcc = CV_FOURCC('P','I','M','1'); // 22 fps
    //int fourcc = CV_FOURCC('M','J','P','G'); // 20 fps
    //int fourcc = CV_FOURCC('M','P','4','2'); // 25 fps
    //int fourcc = CV_FOURCC('D','I','V','3'); // 20 fps
    int fourcc = CV_FOURCC('D','I','V','X'); // 26 fps
    //int fourcc = CV_FOURCC('U','2','6','3'); // 26 fps
    //int fourcc = CV_FOURCC('F','L','V','1'); // 26 fps
    // Uncompressed
    //int fourcc = CV_FOURCC('I','4','2','0'); // 32 fps

    double fps = 10;
    string fn = name + ".avi";
    // initialize video recording
    Size sz(m_vcap.get(CV_CAP_PROP_FRAME_WIDTH),
            m_vcap.get(CV_CAP_PROP_FRAME_HEIGHT));
    m_vrec.open(fn, fourcc, fps, sz, true);
    if (m_vrec.isOpened())
    {
        // turn on recording
        m_rec = true;
#ifdef CAM_REC_BUFFERS
        m_recpos = 0;
#endif
        m_win->PrintLeft("Recording to " + fn);
        // start recording thread
#ifdef CAM_REC_BUFFERS
        m_recfstart = CreateSemaphore(NULL, 0, CAM_REC_BUFFERS, NULL);
        m_recfdone = CreateEvent(NULL, TRUE, TRUE, NULL);
#else
        m_recfstart = CreateEvent(NULL, FALSE, FALSE, NULL);
        m_recfdone = CreateEvent(NULL, FALSE, TRUE, NULL);
#endif
        m_recthread = CreateThread(NULL, 0, RecThread,
                                   this, 0, NULL);
        return true;
    }
    else
    {
        m_win->PrintLeft("Couldn't start recording");
        return false;
    }
}

void GiraffCamera::StopRecord()
{
    if (m_rec)
    {
        // turn off recording
        m_rec = false;
        // wake recording thread, so it notices
        // that m_rec is now false
#ifdef CAM_REC_BUFFERS
        // no need to check if ReleaseSemaphore
        // fails here, since if it does, the
        // recording thread is already awake
        ReleaseSemaphore(m_recfstart, 1, NULL);
#else
        SetEvent(m_recfstart);
#endif
        // wait for it to complete
        WaitForSingleObject(m_recthread, INFINITE);
        // shut down
        CloseHandle(m_recthread);
        CloseHandle(m_recfdone);
        CloseHandle(m_recfstart);
        m_vrec.release();
        m_win->PrintLeft("Recording stopped");
    }
}

bool GiraffCamera::StartPlayback(const string& name)
{
    string fn = name + ".avi";
    m_vplay.open(fn);
    if (m_vplay.isOpened())
    {
        // turn on playback
        m_play = true;
        m_eof = false;
        m_win->PrintLeft("Playback from " + fn);
        // show playback resolution on display
        // (don't bother showing fps, as we don't
        // put the real fps into our recordings)
        ostringstream ost;
        ost << m_vplay.get(CV_CAP_PROP_FRAME_WIDTH) << "x"
            << m_vplay.get(CV_CAP_PROP_FRAME_HEIGHT);
        m_win->SetCameraInfo(ost.str());
        return true;
    }
    else
    {
        m_win->PrintLeft("Couldn't start playback");
        return false;
    }
}

void GiraffCamera::StopPlayback()
{
    if (m_play)
    {
        // turn off playback
        m_play = false;
        m_eof = false;
        m_vplay.release();
        m_win->PrintLeft("Playback stopped");
        // restore original camera resolution
        SetCameraInfo();
    }
}

void GiraffCamera::SetCameraInfo()
{
    // show camera resolution on display
    ostringstream ost;
    ost << m_vcap.get(CV_CAP_PROP_FRAME_WIDTH) << "x"
        << m_vcap.get(CV_CAP_PROP_FRAME_HEIGHT);
    double fps = m_vcap.get(CV_CAP_PROP_FPS);
    if (fps)
    {
        // if FPS is available, show it too
        ost << ", "
            << m_vcap.get(CV_CAP_PROP_FPS) << "fps";
    }
    m_win->SetCameraInfo(ost.str());
}

DWORD WINAPI GiraffCamera::RecThread(LPVOID param)
{
    GiraffCamera *obj = (GiraffCamera*)param;
    unsigned nextpos = 0;
    while (true)
    {
        // wait for captured frame
        WaitForSingleObject(obj->m_recfstart, INFINITE);
        if (!obj->m_rec)
        {
            // recording has been turned off, exit
            break;
        }
        // encode frame
#ifdef CAM_REC_BUFFERS
        obj->m_vrec.write(obj->m_recbuf[nextpos]);
        obj->m_recbuf[nextpos].release();
        nextpos = (nextpos + 1) % CAM_REC_BUFFERS;
#else
        obj->m_vrec.write(obj->m_frame);
#endif
        // signal completion
        SetEvent(obj->m_recfdone);
    }
    return 0;
}
```

B.5. DisplayWindow.hpp

```
#ifndef DISPLAYWINDOW_HPP
#define DISPLAYWINDOW_HPP

#include <opencv2/core/core.hpp>
#include <windef.h>
#include <string>
#include <deque>

typedef void (*InputProc)(int code, int type);
typedef std::deque<std::string> DisplayBuffer;

class DisplayWindow
{
public:
    DisplayWindow(HINSTANCE hInst,
                  HINSTANCE hPrevInst);
    ~DisplayWindow();
    void SetInputHandler(InputProc proc);
    bool Start();
    void Stop();
    void ShowError(LPCSTR pMsg);
    void ShowError(LPCSTR pMsg, DWORD code);
    void Show(const cv::Mat& frame);
    bool ProcessInput();
    void SetCameraInfo(const std::string& info);
    void SetPositionInfo(const std::string& info);
    void SetPerformanceInfo(const std::string& info);
    void PrintLeft(const std::string& info);
    void PrintRight(const std::string& info);
    void InputLeft(const std::string& info);
    void InputRight(const std::string& info);

private:
    HINSTANCE m_hinst;
    HWND m_hwnd;
    InputProc m_proc;
    std::string m_caminfo, m_posinfo, m_perfinfo;
    DisplayBuffer m_leftbuf, m_rightbuf;
    std::string m_leftinput, m_rightinput;
    bool InitApp();
    bool InitWindow();
    void CloseWindow();
    LRESULT WndProc(HWND hwnd,
                    UINT uMsg,
                    WPARAM wParam,
                    LPARAM lParam);
    static
    LRESULT CALLBACK CWndProc(HWND hwnd,
                              UINT uMsg,
                              WPARAM wParam,
                              LPARAM lParam);
};

#endif // DISPLAYWINDOW_HPP
```

B.6. DisplayWindow.cpp

```
#include "DisplayWindow.hpp"

#include <opencv2/imgproc/imgproc.hpp>
#include <windows.h>
#include <sstream>

#define BUFFER_SIZE 32

using namespace std;
using namespace cv;

static const char *app_name = "GiraffNav";

DisplayWindow::DisplayWindow(HINSTANCE hInst,
                             HINSTANCE hPrevInst) :
    m_hinst(hInst), m_hwnd(NULL)
{
    if (!hPrevInst)
    {
        if (!InitApp())
        {
            // couldn't register window class
            ShowError("Couldn't register window class: ",
                      GetLastError());
            return;
        }
    }
}

DisplayWindow::~DisplayWindow()
{
    Stop();
}

void DisplayWindow::SetInputHandler(InputProc proc)
{
    m_proc = proc;
}

bool DisplayWindow::Start()
{
    if (!InitWindow())
    {
        ShowError("Couldn't create window: ",
                  GetLastError());
        return false;
    }
    // show the resolution of the Giraff's monitor
    ostringstream ost;
    ost << "Display resolution: "
        << GetSystemMetrics(SM_CXSCREEN) << "x"
        << GetSystemMetrics(SM_CYSCREEN);
    PrintLeft(ost.str());
    return true;
}

void DisplayWindow::Stop()
{
    CloseWindow();
}

void DisplayWindow::ShowError(LPCSTR pMsg)
{
    MessageBox(m_hwnd, pMsg, app_name,
               MB_OK | MB_ICONERROR);
}

void DisplayWindow::ShowError(LPCSTR pMsg, DWORD code)
{
    ostringstream ost;
    ost << pMsg << code;
    ShowError(ost.str().c_str());
}

// This function is a bridge between the Win32 API
// (which is plain C) and the C++ class DisplayWindow.
LRESULT CALLBACK DisplayWindow::CWndProc(HWND hwnd,
                                         UINT uMsg,
                                         WPARAM wParam,
                                         LPARAM lParam)
{
    DisplayWindow *win;
    if (uMsg == WM_NCCREATE)
    {
        CREATESTRUCT* cs = (CREATESTRUCT*)lParam;
        // This is supposed to be the first message the
        // window receives. (In reality, it isn't,
        // but it's close enough for our purposes.)
        // lpCreateParams is the DisplayWindow pointer provided
        // to CreateWindowEx.
        win = (DisplayWindow*)cs->lpCreateParams;
        // Save it in the window structure.
        SetWindowLongPtr(hwnd, 0, (LONG_PTR) win);
    }
    else
    {
        // Get the DisplayWindow pointer previously stored
        // in the window structure.
        win = (DisplayWindow*)GetWindowLongPtr(hwnd, 0);
    }
    if (win)
    {
        // Dispatch message to DisplayWindow, if possible.
        return win->WndProc(hwnd, uMsg, wParam, lParam);
    }
    else
    {
        // Otherwise (i.e., it's one of the messages that
        // arrive before WM_NCCREATE), do default processing.
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
}

// Register window class for main window
bool DisplayWindow::InitApp()
{
    WNDCLASSEX wcx;

    wcx.cbSize = sizeof(wcx);
    wcx.style = CS_HREDRAW | CS_VREDRAW;
    wcx.lpfnWndProc = CWndProc;
    wcx.cbClsExtra = 0;
    wcx.cbWndExtra = sizeof(DisplayWindow*);
    wcx.hInstance = m_hinst;
    wcx.hIcon = NULL; // no icon yet
    wcx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wcx.hbrBackground = (HBRUSH) GetStockObject(BLACK_BRUSH);
    wcx.lpszMenuName = NULL;
    wcx.lpszClassName = "GiraffNavClass";
    wcx.hIconSm = NULL;

    return RegisterClassEx(&wcx);
}

// Create main window
bool DisplayWindow::InitWindow()
{
    m_hwnd = CreateWindowEx(
        0,
        "GiraffNavClass",
        app_name,
        //WS_OVERLAPPEDWINDOW, // regular window
        WS_POPUP, // fullscreen (no caption or border)
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        (HWND) NULL,
        (HMENU) NULL,
        m_hinst,
        this);
    if (!m_hwnd)
    {
        return false;
    }
    ShowWindow(m_hwnd, SW_SHOWMAXIMIZED);
    return true;
}

// Destroy main window
void DisplayWindow::CloseWindow()
{
    if (m_hwnd)
    {
        DestroyWindow(m_hwnd);
    }
}

bool DisplayWindow::ProcessInput()
{
    MSG msg;
    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        if (msg.message == WM_QUIT)
        {
            // Terminate application
            return false;
        }
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return true;
}

static int RenderInfo(Mat& out, int x, int y, const string& info,
                      int align=-1)
{
    const Scalar color(128,255,255); // yellow
    int fontFace = FONT_HERSHEY_PLAIN;
    double fontScale = 1;
    int thickness = 1;
    int baseLine = 0;
    Size sz = getTextSize(info, fontFace, fontScale,
                          thickness, &baseLine);
    Point org(x, y + sz.height);
    if (align > 0)
    {
        org.x -= sz.width;
    }
    else if (align == 0)
    {
        org.x -= sz.width/2;
    }
    putText(out, info, org, fontFace, fontScale,
            color, thickness);
    return sz.height + baseLine;
}

static void RenderBuffer(Mat& out, int x, int y,
                         DisplayBuffer& buf,
                         string& input)
{
    DisplayBuffer::iterator it = buf.begin();
    while (it != buf.end())
    {
        y += RenderInfo(out, x, y, *it) + 5;
        it++;
    }
    if (!input.empty())
    {
        RenderInfo(out, x, y, input);
    }
}

// Show camera image in main window
void DisplayWindow::Show(const Mat& frame)
{
    // Get size of window drawing area,
    // so we can scale the image to fit it.
    RECT rect;
    GetClientRect(m_hwnd, &rect);
    int width = rect.right;
    int height = rect.bottom;
    // To enforce the alignment required by
    // SetDIBitsToDevice, round the width
    // down to the nearest multiple of 4.
    width = width&~3;
    // Scale image (without interpolation,
    // in order to save CPU).
    Mat out;
    resize(frame, out, Size(width, height),
           0, 0, INTER_LINEAR);
    // Overlay some information from the subsystems
    RenderInfo(out, 0, 0, m_caminfo, -1);
    RenderInfo(out, width/2, 0, m_posinfo, 0);
    RenderInfo(out, width, 0, m_perfinfo, 1);
    RenderBuffer(out, 0, 20, m_leftbuf, m_leftinput);
    RenderBuffer(out, width*3/5, 20, m_rightbuf, m_rightinput);
    // Create bitmap info needed by SetDIBitsToDevice
    BITMAPINFOHEADER bmih;
    bmih.biSize = sizeof(bmih);
    bmih.biWidth = out.cols;
    bmih.biHeight = -out.rows; // negative = top-down DIB
    bmih.biPlanes = 1;
    bmih.biBitCount = 24;
    bmih.biCompression = BI_RGB;
    bmih.biSizeImage = 0;
    bmih.biXPelsPerMeter = 0;
    bmih.biYPelsPerMeter = 0;
    bmih.biClrUsed = 0;
    bmih.biClrImportant = 0;
    // Draw video frame in window
    HDC hdc = GetDC(m_hwnd);
    SetDIBitsToDevice(hdc, 0, 0,
                      width, height,
                      0, 0,
                      0, out.rows,
                      out.data,
                      (BITMAPINFO*)&bmih,
                      DIB_RGB_COLORS);
    ReleaseDC(m_hwnd, hdc);
}

LRESULT DisplayWindow::WndProc(HWND hwnd,
                               UINT uMsg,
                               WPARAM wParam,
                               LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_NCCREATE:
        // The window now exists.
        m_hwnd = hwnd;
        return TRUE;
    case WM_NCDESTROY:
        // The window no longer exists.
        m_hwnd = NULL;
        return 0;
    case WM_KEYDOWN:
    case WM_KEYUP:
    case WM_CHAR:
        if (m_proc)
        {
            m_proc(wParam, uMsg);
        }
        return 0;
    case WM_CLOSE:
        // The user pressed the Close button
        // (or its keyboard shortcut, Alt-F4).
        DestroyWindow(hwnd);
        return 0;
    case WM_DESTROY:
        // The main window is being closed, so make
        // sure the app itself also terminates.
        PostQuitMessage(0);
        return 0;
    default:
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
}

void DisplayWindow::SetCameraInfo(const std::string& info)
{
    m_caminfo = info;
}

void DisplayWindow::SetPositionInfo(const std::string& info)
{
    m_posinfo = info;
}

void DisplayWindow::SetPerformanceInfo(const std::string& info)
{
    m_perfinfo = info;
}

// to remove end-of-line characters from end of string
static size_t chomped(const string& info)
{
    size_t n = info.find_last_not_of("\r\n");
    if (n != string::npos)
    {
        return n+1;
    }
    else
    {
        return 0;
    }
}

static string chomp(const string& info)
{
    size_t n = chomped(info);
    return info.substr(0, n);
}

// add line to left pane
void DisplayWindow::PrintLeft(const string& info)
{
    m_leftbuf.push_back(chomp(info));
    while (m_leftbuf.size() > BUFFER_SIZE)
    {
        m_leftbuf.pop_front();
    }
}

// add line to right pane
void DisplayWindow::PrintRight(const string& info)
{
    unsigned span = 32;
    size_t len = chomped(info);
    // split string into lines of 32 characters each
    for (size_t n=0; n<len; n+=span)
    {
        size_t end = n+span;
        if (end > len)
        {
            end = len;
        }
        m_rightbuf.push_back(info.substr(n,end-n));
    }
    // if buffer is now full, scroll up by removing
    // lines from the top
    while (m_rightbuf.size() > BUFFER_SIZE)
    {
        m_rightbuf.pop_front();
    }
}

// show user input for left pane
void DisplayWindow::InputLeft(const string& info)
{
    m_leftinput = info;
}

// show user input for right pane
void DisplayWindow::InputRight(const string& info)
{
    m_rightinput = info;
}
```

B.7. GiraffNav.cpp

```
#include "DisplayWindow.hpp"
#include "GiraffCamera.hpp"
#include "GiraffMotor.hpp"
#include "FeatureExtract.hpp"

#include <opencv2/highgui/highgui.hpp>
#include <windows.h>
#include <sstream>
#include <iomanip>

#define DEF_WIDTH 800
```

```
#define DEF_HEIGHT 600

#define KBD_TURN_SPEED 45
#define KBD_MOVE_SPEED 0.4

// Disp. res. 800x1280
// ca 10fps at 800x600 capture
// Notes from testing:
// Default tilt angle #bab8b23d = 0.0872664005

#define PLAY_PATH "C:/GiraffRec/"
#define PLAY_FILE "20130515_135355"

using namespace std;
using namespace cv;

static DisplayWindow* mainWindow;
static GiraffCamera* mainCamera;
static GiraffMotor* mainMotor;

static FeatureExtractor* extractor;

enum InputMode
{
    INPUT_NONE = 0,
    INPUT_LEFT = 1,
    INPUT_RIGHT = 2
};

static InputMode inputMode = INPUT_NONE;
static bool returnPressed = false;
static string inputLine;
static bool isRecording = false;
static bool isPlaying = false;

void SetResolution(int width, int height)
{
    mainCamera->Stop();
    mainCamera->Start(width, height);
}

void ToggleRecording()
{
    if (!isRecording)
    {
        // decide on a file name
        SYSTEMTIME tm;
        GetLocalTime(&tm);
        ostringstream ost;
        ost << tm.wYear
            << setfill('0')
            << setw(2) << tm.wMonth
            << setw(2) << tm.wDay
            << "_"
            << setw(2) << tm.wHour
            << setw(2) << tm.wMinute
            << setw(2) << tm.wSecond;
        string name = ost.str();
        if (!mainCamera->StartRecord("/GiraffRec/cam_" + name))
        {
            return;
        }
        if (!mainMotor->StartRecord("/GiraffRec/ctl_" + name))
        {
            mainCamera->StopRecord();
            return;
        }
        isRecording = true;
    }
    else
    {
        mainMotor->StopRecord();
        mainCamera->StopRecord();
        isRecording = false;
    }
}

void TogglePlayback()
{
    if (!isPlaying)
    {
        string name = PLAY_FILE;
        if (!mainCamera->StartPlayback(PLAY_PATH "cam_" + name))
        {
            return;
        }
        if (!mainMotor->StartPlayback(PLAY_PATH "ctl_" + name))
        {
            mainCamera->StopPlayback();
            return;
        }
        isPlaying = true;
    }
    else
    {
        mainMotor->StopPlayback();
        mainCamera->StopPlayback();
        isPlaying = false;
    }
}

void InputHandler(int code, int type)
{
    if (type == WM_KEYUP &&
        code == VK_RETURN)
    {
        returnPressed = false;
    }

    if (inputMode)
    {
        if (type != WM_CHAR)
        {
            return;
        }
        switch (code)
        {
        case '\b': // Backspace
            if (!inputLine.empty())
            {
                inputLine.erase(inputLine.length()-1);
            }
            break;
        case '\e': // Esc
            inputMode = INPUT_NONE;
            inputLine.clear();
            mainWindow->InputRight(inputLine);
            return;
        case '\r': // Enter
            if (returnPressed)
            {
                // keypress already handled separately
                return;
            }
            if (!inputLine.empty())
            {
                mainMotor->SendUserCommand(inputLine);
            }
            inputMode = INPUT_NONE;
            inputLine.clear();
            mainWindow->InputRight(inputLine);
            return;
        default:
            if (code >= 32 && code <= 126)
            {
                // Regular ASCII character
                inputLine.push_back(code);
            }
            break;
        }
        mainWindow->InputRight(inputLine + "_");
        return;
    }

    if (type == WM_KEYUP)
    {
        switch (code)
        {
        case VK_DOWN:
        case VK_UP:
            mainMotor->SetMotion(0);
            break;
        case VK_LEFT:
        case VK_RIGHT:
            mainMotor->SetTurn(0);
            break;
        }
    }

    if (type != WM_KEYDOWN)
    {
        return;
    }
    switch (code)
    {
    case VK_ESCAPE:
        // Initiate system shutdown
        mainWindow->Stop();
        break;
    // Manual movement
    case VK_LEFT:
        mainMotor->SetTurn(-KBD_TURN_SPEED);
        break;
    case VK_RIGHT:
        mainMotor->SetTurn(KBD_TURN_SPEED);
        break;
    case VK_UP:
        mainMotor->SetMotion(KBD_MOVE_SPEED);
        break;
    case VK_DOWN:
        mainMotor->SetMotion(-KBD_MOVE_SPEED);
        break;
    // Keys to try out various resolutions.
    case '1':
        SetResolution(1600, 1200);
        break;
    case '2':
        SetResolution(1280, 960);
        break;
    case '3':
        SetResolution(1024, 768);
        break;
    case '4':
        SetResolution(800, 600);
        break;
    case '5':
        SetResolution(640, 480);
        break;
    // Misc keys
    case VK_RETURN:
        // input motor command
        inputMode = INPUT_RIGHT;
        returnPressed = true;
        mainWindow->InputRight("_");
        break;
    case 'A':
        // this is a hack to check
        mainMotor->m_autoupdate = !mainMotor->m_autoupdate;
        if (mainMotor->m_autoupdate)
        {
            mainWindow->PrintLeft("Motor autoupdate on");
        }
        else
        {
            mainWindow->PrintLeft("Motor autoupdate off");
        }
        break;
    case 'B':
        mainMotor->GetBulkData();
        break;
    case 'H':
        mainMotor->Home();
        break;
    case 'P':
        TogglePlayback();
        break;
    case 'R':
        ToggleRecording();
        break;
    case 'T':
        mainMotor->SetTilt(1);
        break;
    case 'U':
        mainMotor->Undock();
        break;
    }
}

void MainLoop()
{
    LARGE_INTEGER freq, period;
    LARGE_INTEGER last_count;
    DWORD frame_count = 0;
    DWORD fms = 0;
    DWORD fps = 0;
    Mat frame;

    QueryPerformanceFrequency(&freq);
    // recalculate performance data every 250ms.
    period.QuadPart = freq.QuadPart / 4;

    QueryPerformanceCounter(&last_count);
    while (mainWindow->ProcessInput())
    {
        bool ok = mainMotor->Process();
        if (!ok && isPlaying)
        {
            TogglePlayback();
            mainMotor->Process();
        }
        mainCamera->Grab(frame);
        extractor->Process(frame);
        mainWindow->Show(frame);
        frame_count++;

        // check performance measures
        LARGE_INTEGER cur_count, diff_count;
        QueryPerformanceCounter(&cur_count);
        diff_count.QuadPart = cur_count.QuadPart - last_count.QuadPart;
        if (diff_count.QuadPart >= period.QuadPart)
        {
            // recalculate performance data
            LONGLONG factor = frame_count * freq.QuadPart;
            fms = (diff_count.QuadPart*1000) / factor;
            fps = factor / diff_count.QuadPart;
            frame_count = 0;
            last_count.QuadPart = cur_count.QuadPart;
        }
        ostringstream ost;
        ost << fms << "ms, "
            << setw(2) << fps << "fps";
        mainWindow->SetPerformanceInfo(ost.str());
    }
}

int WINAPI WinMain(HINSTANCE hInst,
                   HINSTANCE hPrevInst,
                   LPSTR pCmdLine,
                   int nCmdShow)
{
    mainWindow = new DisplayWindow(hInst, hPrevInst);
    mainWindow->SetInputHandler(InputHandler);
    if (!mainWindow->Start())
    {
        return 0;
    }
    mainCamera = new GiraffCamera(mainWindow);
    if (!mainCamera->Start(DEF_WIDTH, DEF_HEIGHT))
    {
        mainWindow->ShowError("Could not connect to camera!");
        return 0;
    }
    mainMotor = new GiraffMotor(mainWindow);
    if (!mainMotor->Start())
    {
        mainWindow->ShowError("Could not connect to motor controller!");
        return 0;
    }
    extractor = new FeatureExtractor(mainWindow);
    MainLoop();
    delete extractor;
    delete mainMotor;
    delete mainCamera;
    delete mainWindow;
    return 0;
}
```

B.8. FeatureExtract.hpp

```
#ifndef FEATUREEXTRACT_HPP
#define FEATUREEXTRACT_HPP

#include "DisplayWindow.hpp"

#include <opencv2/highgui/highgui.hpp>

#define CAM_REC_BUFFERS 8

class FeatureExtractor
{
public:
    FeatureExtractor(DisplayWindow* win);
    ~FeatureExtractor();
    void Process(cv::Mat& frame);

private:
    DisplayWindow* m_win;
};

#endif // FEATUREEXTRACT_HPP
```

B.9. FeatureExtract.cpp

```
#include "FeatureExtract.hpp"

// Sample feature extractor

#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/features2d/features2d.hpp>

using namespace std;
using namespace cv;

FeatureExtractor::FeatureExtractor(DisplayWindow* win) :
    m_win(win)
{
}

FeatureExtractor::~FeatureExtractor()
{
}

void FeatureExtractor::Process(cv::Mat& frame)
{
    Mat grayframe;

    // Convert to grayscale
    cvtColor(frame, grayframe, CV_BGR2GRAY);

#if 0 // Canny edge detector (just for demonstration)
    Mat cannyframe(grayframe.size(), grayframe.type());
    Canny(grayframe, cannyframe, 20, 50);
    cvtColor(cannyframe, frame, CV_GRAY2BGR);
#endif // 1

#if 1 // "FAST" corner detector
    vector<KeyPoint> keypoints;
    FAST(grayframe, keypoints, 50);

    // draw pink circles around detected corners
    drawKeypoints(frame, keypoints, frame,
                  Scalar(128,0,255),
                  DrawMatchesFlags::DRAW_OVER_OUTIMG);

    // These eypoints could be given to some
    // SLAM implementation.
#endif
}
```

C. Contents of the CD-ROM

The CD-ROM contains these directories:

- `Bin`: This directory contains the binaries needed to run the system. They can be copied to a USB memory stick, which can then be inserted into one of the Giraff's USB ports, along with a computer mouse. When browsing the contents of the memory stick, doubleclick `GiraffNav.exe`. (If you plan to do any recording, make sure that a `GiraffRec` directory exists on the memory stick, otherwise recording may fail.)
- `GiraffNav`: This is the source code of the developed system, along with the Code::Blocks project file, and MinGW-compiled binaries.
- `OpenCV`: This is the source code of OpenCV version 2.4.9, and MinGW-compiled binaries of it. These binaries are needed for building GiraffNav.
- `GiraffRec`: This directory contains a couple of recordings of the Giraff moving around the care center using the developed system.

UNIVERSITY OF TROMSØ