12-2011

cytonGrasp: Cyton Alpha Controller via GraspIt! Simulation

Nicholas Wayne Overfield
noverfie@utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Nicholas Wayne Overfield entitled "cytonGrasp: Cyton Alpha Controller via GraspIt! Simulation." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Lynne Parker, Major Professor

We have read this thesis and recommend its acceptance:

Michael Berry, Bruce MacLennan

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

cytonGrasp: Cyton Alpha Controller via GraspIt! Simulation

A Thesis Presented for

The Master of Science

Degree

The University of Tennessee, Knoxville

Nicholas Wayne Overfield

December 2011

to Ken and Linda Overfield for all the support, food, shelter, and comic books

Acknowledgments

This thesis is the product of countless people beyond myself, and could not have been completed without the support of many peers, advisors, and friends.

I would like to acknowledge my advisor, Dr. Lynne Parker, for being among the hardest working people I know. Her kindness, dedication, and genuine love for not only her work but also her students are a constant source of inspiration. Her willingness to put up with my mistakes and faults, as well as to work with me and help me be a better person by guiding me through my studies, encouraging and supporting me throughout, is but one reason I am in her debt.

I also need to thank the rest of my thesis committee, Dr. Michael Berry and Dr. Bruce MacLennan, for all their additional guidance, their time and support, and for easing scheduling conflicts. I greatly appreciate all the effort they put into helping me with this project.

None of this work would be possible without all the hard work of Dr. Matei Ciocarlie who works on the development of the Graspit! Simulator. Not only has Dr. Ciocarlie developed amazing software that propelled my work forward to new heights and loftier goals, but he gave numerous suggestions and assistance via email throughout the life of the project. Never before have I encountered someone willing to provide assistance in as much detail to a colleague he has never met. Thank you, Dr. Ciocarlie for all your help and kindness.

In that same vein, I need to recognize John Kelly without whom I would still be modeling the face sets of dice in the basement of the computer science building. Without John's

dedication to his friends and endless hours of support many engineering concepts would still elude me and the modeling for my GraspIt! simulation would have been nearly impossible.

My sincerest thanks and many applogies to the lab staff of our Electrical Engineering and Computer Science Department, in particular Andrew Morgan Brackett, for his many hours of support installing and linking not only Robot Operating System (ROS), but helping me get the GraspIt! simulator running and teaching me numerous new Linux techniques.

I need to thank Scott Wells, program manager for the Center for Intelligent Systems and Machine Learning (CISML), for assisting me in preparing my final drafts of this thesis.

I need to acknowledge my colleagues in the Distributed Intelligence (DI) Lab Chris Reardon, Michael Bailey, Richard Edwards, Dr. YuanYuan Li, Tony Zhang, John Hoare, Hao Zhang, Sudarshan Srinivasan, Chi Zhang, and Robert Lowe, for supporting me throughout this project and all their helpful suggestions.

Thank you to the rest of my wonderful friends, David and Ben Boersma, Matt and Elizabeth Brown, Clinton and Matt Nolan, Katie Schuman, and William Pierce. Your friendships mean more to me than I can ever express and constantly made this process feel less uphill. Finally, I need to thank Kerry O'Donnell for her boundless patience, generosity, and love which was always a source of strength for me.

The mind has exactly the same power as the hands; not merely to grasp the world, but to change it - Colin Wilson

Abstract

This thesis addresses an expansion of the control programs for the Cyton Alpha 7D 1G arm. The original control system made use of configurable software, which exploited the arm's seven degrees of freedom and kinematic redundancy to control the arm based on desired behaviors that were configured off-line. The inclusions of the GraspIt! grasp planning simulator and toolkit enable the Cyton Alpha to be used in more proactive on-line grasping problems as well as presenting many additional tools for on-line learning applications. In short, GraspIt! expands what is possible with the Cyton Alpha to include many machine learning tools and opportunities for future research.

Noteworthy features of GraspIt!:

- A 3D user interface allowing the user to see and interact with virtual objects, obstacles, and robots, in addition to a 3D representation of the Cyton Alpha
- A collision detection and contact determination system within simulation
- On-line grasp analysis routines
- Visualization methods for determining the weak points within a grasp, as well as creating projections of grasp quality and the ability to resist dynamic forces.
- Computation of numerical grasp quality metrics and visualization methods for proposed grasps
- Dynamics engine
- Support for lower-dimensional, hand posture subspaces

- Interaction with sensors (Flock of Birds tracker) and hardware (Pioneer robot) within simulation
- GraspIt! can generate huge databases of labeled grasp data, which can be used for data-driven grasp-planning algorithms and it has built-in support for the Columbia Grasp Database

By making use of the GraspIt! simulator, it is possible to test algorithms for grasp manipulation, grasp planning, or grasp synthesis more quickly and with greater repeatability than would be possible on the real robot.
Contributions of this system include:

1. A joint based 3D rendering of the Cyton Alpha 7D 1G arm
2. Simulated bodies for several objects in the DI Lab
3. Support for multiple representations of joint data within three-dimensional space:
    - Euler Angles
    - Quaternions
    - Denavit-Hartenberg Parameters
4. A framework for future work in grasp-planning, grasp synthesis, cooperative grasping tasks, and transfer learning applications with the Cyton Alpha arm.

Contents

List of Figures

Chapter 1

Introduction

The cytonGrasp software represents the merger of two bodies of distinct software for controlling and modeling the Cyton Alpha 7D 1G arm – the Cyton Control API and the Graspit! grasp planning software.

1.1 Cyton Control API

The Cyton Alpha 7D 1G is a seven degree of freedom manipulator arm with a gripper end effector of two prismatic joint fingers capable of moving 35 cm apart. The Cyton Alpha is released by Robai and is accompanied with its control software which handles the arm's kinematic redundancy and interfaces with its input devices. The Cyton Alpha Control API has two major components, a calculationAPI used for inverse-kinematic calculations and a hardware API used to control the Cyton arm directly. The control API can output a set of joint positions that pass directly to the hardware [1].

1.2 CytonViewer

In addition to the control API Robai also released the Cyton Viewer software, as shown in Figure 1.1, which can simulate the robot as well as offer a means of teleoperation. It

allows for direct end-effector or joint manipulation within the Cyton Alpha's full range of movement.

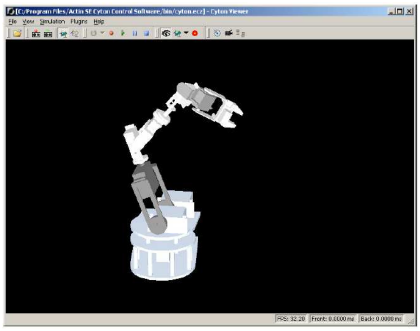

Figure 1.1: CytonViewer

At the time of this writing, version 1.2 of the CytonViewer and accompanying control software are available on Mobile Robots's webpage for Linux along with some required packages. The software has been updated to version 2.1 but only for Windows based platforms.

The software relies upon the libraries listed in Appendix 1 and is split into two sections – controlInterface and hardwareInterface. The former does the inverse-kinematic equations described above as well as provides visualization, while the latter connects to and issues commands to the hardware. The two pieces communicate by means of several unique predefined types, in particular, sets of vectors and Quaternions for each joint position specified in the include file cytonTypes.h.

1.3 GraspIt! Software

GraspIt! is a system of software tools under development at Willow Garage. It is a full simulator that can accommodate any arbitrary hand, arm, object, or obstacle. It also contains a library of such objects, which is still growing. It can load objects and obstacles directly and populate a complete simulation world [2].

The GraspIt! engine includes rapid collision detection, contact determination, and grasp planning systems. Once a grasp is created, a set of grasp quality metrics can be created and

visualization methods allowing the user to see weak points or potential issues with the grasp as well as arbitrary 3D projections of the grasp's 6D wrench space [2].

1.4 Motivation

The purpose of this system is to provide tools to simply use the Cyton Alpha arm, but additionally to allow access to GraspIt!'s functionality and the many machine learning areas and applications it presents. There are several positive and negative aspects to both the Cyton Control API and GraspIt!, primarly how daunting and inaccessible each piece of software is. The primary motivition for this project is to provide tools to fellow researchers to ease the use of GraspIt! and many other tools to enable those examining grasp techniques, with or without the Cyton Alpha arm, to do so easily and efficiently and in that way contributes to machine learning as a whole.

The libraries bundled with the CytonViewer and the control software have gone many years without Linux support. The lib and other binaries used to compile the control software only exist in 32-bit form and are linked against specific Linux libraries (libjasper-1.701-1 among others), which are so old that obtaining them is difficult. Newer versions of most of these libraries cause the control software to behave in aberrant ways.

The installation instructions for the Cyton libs involve a modification to the LD_LIBRARY_PATH, which can cause these out of date libraries to get preferential treatment during execution of other programs, causing many runtime issues. OpenInventor, gtk-recordMyDesktop, and Devede were a few of the programs that had conflicts, although anything dependent upon libraries listed in Appendix 1 could be faulty.

Additionally, using the cytonViewer as an application or attempting to visualize the Cyton arm proved faulty at best. Loading the included cyton.ecp would often cause the simulation to crash, even if there was no movement plan or path file for the arm. This was likely related to the age of the Linux release and its incompatibility with newer libs installed on the test OS. With the release of Cyton II for Windows, the 1.2 version on Linux seemed to get little support from the Cyton community. Additionally, the Actin Viewer version of

the model had no obvious methods for making use of the simulated arm beyond display and teleoperation. There was little support for any kind of physical interaction with simulated objects or other mechanics where simulation would be very useful in operation alongside the physical arm. While many of these features were expanded in the Cyton II software, two years have passed without a Linux release.

GraspIt! provided a means of simulating the arm in the manner desired. The Pioneer robot, the Cyton Alpha Arm, and any objects or obstacles could be simulated in a full 3D environment with a physics engine in place, which provided, among many other features, realistic friction and gravity. Any object could be added directly to the simulation so long as the required files were provided. The arm, obstacles, and objects were stored in simulation worlds, defined in a simple XML document.

The Barrett Arm, one of the first arms added to GraspIt!, provides an example of many of GraspIt!'s stronger features, including the calculation of eigengrasp data, pose estimation, and coordination between the real and simulated world by means of flock of birds tracking. Among its many features, the ability to calculate strong grasps, their quality, and effectiveness made GraspIt! stand out as a tool that would provide many useful features beyond simulation. While implementing these features in a machine learning or planning task are outside the scope of my project, support for them is enabled, and they were a large factor in deciding to build a simulated version of the Cyton Alpha arm in GraspIt!.

GraspIt! is a part of Robot Operating System (ROS) with a strong and active development community. New obstacles, robots, and features are constantly being added. While this project was underway, GraspIt! underwent its official release and is now directly in the ROS repository and can easily be installed and accessed as a part of ROS. In that same vein, many examples of a messaging protocol between GraspIt! and other programs as a ROS node exist, meaning the program has a strong focus on modularity and the project could potentially be updated as a subset of a larger project easily.

1.5 Problem Statment and Challenges

As stated in the introduction, before work on the CytonGrasp API could begin, the two pieces of software needed to be merged in a way that they would not conflict. The complexity of these combined pieces of software made compiling them with a VirtualBox an obvious choice. In this way, instead of utilizing make and Qmake files to figure out how to link and compile an example, any users of the system could be provided with a guaranteed working version that was highly portable.

Beyond simply building the two projects, the Cyton API and GraspIt! use very different mathematical structures to maintain the same information; thus, there was a need to create a system that would support both Quaternion geometry and Denavit-Hartenberg Parameters while still being able to present the data in familiar Euler angles.

1.5.1 Cyton 32-bit binaries

There is no way to guarantee that a user will be working on a 32-bit system, thus many issues arose involving the 32-bit Cyton control binaries as the source files to compile them were unavailable. In early development, any attempt to compile with the m32 flag seemed to solve this issue. The integration with GraspIt!, however, due to the many programs upon which GraspIt! depended, made this option unfeasible, as it would require a massive recompilation of various programs as well as changes to more than 30 make and Qmake files. Unfortunately, the cytonGrasp project is not platform independent at this time.

1.5.2 Graspit Compilation

GraspIt! makes extensive use of the Qt toolkit for its visualization and as such is compiled by a combination of rosmake and qmake files used to install the programs and generate makefiles. This leads to some interesting complications when linking against the GraspIt! binaries, as their installation and compilation is automatic.

First, there is a long list of ROS packages, listed in Appendix 2, upon which GraspIt! depends, in order to link with the Cyton Control API's 32 bit versions of each of these packages that need to be installed. To build GraspIt!, one needs to use ROS's build structure. Simply invoke rosmake graspit once the ROS_PACKAGE_PATH has been set up according to ROS's wiki.

The folder graspit_source contains all of the source, object, and model files used by GraspIt!. This is what cytonGrasp links against. The object files are located in a hidden folder graspit_source/.obj and built by the makefile generated in graspit_source when rosmake is invoked. The makefile in cytonGrasp's source has variables that represent the location of GraspIt!'s install and the graspit_source subdirectory, if the build path changes.

1.5.3 The QT Libraries

A major source of error, carefully isolated in the makefiles for cytonGrasp, was the overlap between the Cyton control API's visualization and Graspit!. Both use the Qt toolkit, but are far enough apart to depend on different versions of the software due to their development times.

The specific libraries in question are bold in Appendix I. CytonGrasp does not make use of CytonViewer's visualization features within its control or calculation aspects; due to this, it can be compiled with the later versions of the QT libraries needed by GraspIt!. If one were to attempt to visualize the Cyton Alpha in Actin Viewer using the combined libraries, however, the project becomes unstable. The libraries are clearly marked and sorted in the makefile for cytonGrasp as both are needed on the system, but only the latter libraries are used in the final linking of the project.

1.5.4 Open Inventor

Open Inventor development toolkit is the recommended way to create objects for GraspIt! as both make extensive use of the Coin 3D graphics toolkit. As with the Qt library, a conflict exists where GraspIt! uses coin40, while Open Inventor makes use of Coin60. A

much simpler solution is to make the objects in Open Inventor on one system, export them as IV files, move them to the GraspIt! Installation, and load them into GraspIt!, as there is no major change with the format of IV files. This allows for objects to be quickly built for use in a simulated world.

Open Inventor is an object-oriented, 3D rendering toolkit. It is a library of objects and methods used to created interactive 3D graphical applications and scenes. Open Inventor is written in C++ with C bindings and has a framework based on OpenGL. Its objects include database primitives: shapes, properties, groups, and engine objects as well as interactors [3].

1.6 Approach

The cytonGrasp system can be thought of as two major pieces of software working in harmony. As described in the introduction, the first major hurdle was simply compiling and linking these two piece of arm simulation and control. Next came the design of a box world, which could be represented both in simulation and reality and would demonstrate the strengths of this unique system. A design for the combined systems which harnessed the strengths of the arm's built-in control structure while being able to make the most of GraspIt!'s planning capabilities would require a system that could share data in multiple forms yet still be simple to follow, as well as, being able to freely switch itself between controlling the real and simulated arm.

1.7 Contributions

In addition to the cytonGrasp framework and source code and the unique approach to handling the Cyton control API it presents, several GraspIt! simulated objects were prepared as part of the cytonGrasp box world. Included with each of these objects is the source code which generates the scene graphs used to render each object in 3D. The end result is that GraspIt! is application ready, and a 1:1 scale replica has been made, which can replicate

most of the current research being done as part of GraspIt! with the unique Cyton Alpha arm.

Objects in the test world include five kinds of die:

- Cube (d6)
- Octahedron (d8)
- Heptagonal bipyramid (d10)
- Icosahedron (d20)

Each of these objects are a unique shape and differently angled surfaces making grasping them a unique problem, especially for a gripper like the Cyton Alpha. The white table model and the box Pioneer are also included, as both were used to make the simulation exactly like the physical world. There are four versions of the cytonAlpha modeled for GraspIt! (including the IV and XML files). The primary model has colored joints so that each section is a unique color, one which is a solid metallic gray and black, one composed of the orginal STL objects used to help build the other models, and one which has fixed mobility and treats the arm as a single, solid object. These arm models are in turn made up of their own sets of joint models (both IV and XML files) with a larger XML file that specifies the relationships of the joints and their connections, as detailed in the section 3.3.

Chapter 2

Literature Review

The cytonGrasp project is meant to provide an expansive toolset for future work, with the Cyton Alpha arm which is practical, modular, and accessible. As mentioned in the introduction, the key aim of the project is to increase opportunities for future research with the arm by utilizing the expansive set of features offered by GraspIt! Therefore, in order to better provide an understanding of the depth of features now available for future work an overview of the work done with GraspIt! – both past and present – will makes up the bulk of this chapter. The research and results of preliminary GraspIt! work are reviewed and seen through the lens of how they can relate specifically to peer-to-peer robot teaming, grasp mechanics, and capabilities of the Cyton Alpha arm. Section 2.1 provides a basic view of what GraspIt! was built for and how it can be used. Section 2.2 introduces the concept of the grasp wrench space. This leads into a discussion of eigengraps, the principle components of a grasp wrench space, in Section 2.3. Section 2.4 discusses techniques for modeling the real world within a GraspIt! simulation through visual servoing.

2.1 Introduction to GraspIt!

GraspIt! has shown many promising results that combine real-time vision systems and simulation of both planning and executing grasps on physical robots. Additionally, it has many modular features, such as the ability to define multi-robot sets to create full robotic

platforms. A kinematic chain for one hand can be mounted onto various arms, each with a particular offset and transform [4]. In this same way, it is possible to mount the Cyton Alpha's gripper onto its arm, and mount the system on a virtual Pioneer robot, identical to the physical configuration used in the DI Lab. This configuration of mounted systems allows a user to separate the gripper and perform various calculations with solely its ability to grasp objects

There is an existing TCP connection and a simple test protocol for interacting with a GraspIt! simulation as well as a ROS node communication structure. Finally, there is an existing Matlab interface, which can compute joint torques to send back to GraspIt! [4].

Each time contact occurs between two objects, GraspIt! places a friction cone at the point of contact. The magnitude of these cones is representative of the frictional forces between the two surface materials and a static Coulomb friction model. Friction cones are larger when tangential forces are possible before slippage and thus large cones are indicative of a more stable grasp [5].

In addition to detailed collision detection, GraspIt! contains a direct means of computing the validity of a grasp and a system for establishing metrics, the Grasp Wrench Space (GWS)– which is further discussed in Section 2.2. A GWS is the 6-D space of forces and torques resulting from the grasp of a 3-D object and can even be projected into 3-D space for easy visualization by fixing three of the wrench coordinates – a technique more clearly defined in [6]. If a GWS includes the origin, then the grasp is considered stable because it can resist outside forces by means of scaling up the forces at the points of contact between the robot and the object. Purple indicators can show where a specific grasp is weakest to outside forces (such as gravity) as a part of the wrench space. This is defined by the distance from the origin to the closest facet on the boundary of the GWS [5].

Figure 2.1, borrowed from early GraspIt! paper [4], shows this concept very clearly. The upper left of this figure shows a projection of the GWS, while the lower left is a projection that shows the space of torques that can be applied without a net force acting on the object. The Barrett hand is grasping a blue mug in simulation in a manor very similar to a human. The purple indicators show the grasp is weakest when an outside force is applied upward

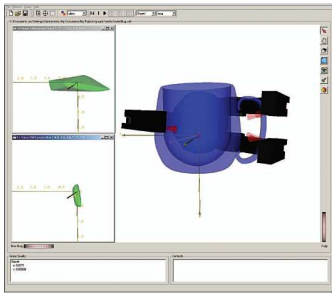

Figure 2.1: A completed force-closure grasp of the Barrett hand.

while the projection of the wrench space shows a variety of applicable horizontal forces but very little in the vertical direction. In short, the grasp shown in Figure 2.1 can easily move the object horizontally but would be ill-suited for vertical movement of the cup, because it must rely on friction between the plastic gripper and cup [4].

This process, however, only works for static grasps. Gravity, inertial forces, and collision response are not factors until the motion of each dynamic body is computed and added to the world. GraspIt! handles this process through a numerical integration scheme that computes changes in velocity of each body over small finite time steps. This is done primarily through two constraints – equality constraints for preventing bodies connected by a joint from separating, and inequality constraints to prevent other bodies from inter-penetrating. After each iteration of the dynamics is completed, GraspIt! can draw the contact forces between the contact points of two bodies and the dynamics can be freely paused and resumed in simulation. As discussed in the introduction above, GraspIt! handles many of these dynamic calculations by means of the xml files that make up one half of each object in the simulation world. Further details on this process are covered in Section 2.4 [4].

2.2 Grasp Wrench Space

In general, automatic grasp synthesis (dexterity, equilibrium, stability, and dynamic behavior) is the task of finding the combination of hand posture (intrinsic Degrees of Freedom) and position (extrinsic DOF). In order to determine the validity of any grasp synthesis approach, the Grasp Wrench Space and its quality measure act as a metric.

The Grasp Wrench Space (GWS) is a term signifying the search space of possible wrenches that can be applied by a grasp. It is found by taking the convex hull of all grasps that can be applied through each point of contact between a hand and an object [6]. The purpose of this is to establish a metric that takes into account not only existing contacts between a hand and an object but also potential contacts that can be realized by small changes in the current state. The GWS leads directly to a quality measure for the grasp in addition to allowances for dynamic behavior. If the origin is not contained in the GWS, the grasp does not have force-closure (it does not encase the object well enough) and the quality is zero. Otherwise the quality of a grasp is equal to the distance from the origin to the closest boundary of the GWS. Various permutations of the desired points of contacts and scaling applied to the wrench space can change the metrics leading to the overall quality measure [6].

2.3 Eigengrasps

Any hand posture is fully specified by its joint values and can be expressed as a point in a high-dimensional joint space. If n is the number of degrees of freedom of the hand, then a grasp posture p can be defined as:

$$p = [\theta_1, \theta_2, ...\theta_n] \in R^d \tag{2.1}$$

where θ_i is the i-th DOF [6]. Previous research suggests most human grasping postures derive from a much smaller set of discrete pregrasp shapes, implying that there should be clustering in the d-dimensional space of a grasp postures. In fact, research has shown that 80% of the variance within human grasps is a factor of the two principal components [6]. Further research at Willow Garage and work with the GraspIt! simulator refer to these principal grasp components as eigengrasps.

While the remaining DOFs are not useless and contribute more than simple noise, the eigengrasp concept allows for the design of flexible control algorithms that operate identically

across many presented hand models, allowing techniques developed as part of the GraspIt! simulator to be shared among various kinds of arm and hand models. Additionally, planning in the reduced space of two eigengrasps does not result in a posture where a robotic hand conforms perfectly to the surface of an object; however, the result is often close enough to such a posture that a stable grasp can be obtained with simple heuristics [6].

Even in cases such as the Cyton Alpha's gripper where there is very little to gain from a reduction in the posture space, eigengrasps can reduce the dimensionality of a wide array of problems, such as obstacle avoidance. If a table's surface (such as the one in our simulated world) were to prevent the execution of the best grasp, thus forcing a planning algorithm to find alternative solutions, the only additional cost incurred by the grasp planner is that of collision detection against the obstacle for each newly generated state, due to the reduced dimensionality of the grasp [6].

2.4 GraspIt! and Tracking in the Physical World

Closed-loop control of a robot with vision used in the feedback loop is commonly referred to as visual servoing. Visual servoing typically uses a model or feature based approach both, of which can be extracted from a CAD-like model of the object [5]. Thus, if objects were built in a 1:1 scale with the physical world as was the robot arm, an accurate tracking simulation could be derived. In [5], both styles of visual servoing are implemented and a position-based approach is used for driving the robot arm while image-based servo control is used to visually servo the target object to the desired pose.

A wire frame CAD model is first developed and the initialization of the tracking system is done manually, in which a number of corresponding points on the wire model and the image obtained from a camera are chosen by the user. After pose estimation is obtained, the model of the object can be projected over the image plane, and a search performed for the maximum discontinuity in the intensity gradient along the normal direction to the edge. The edge normal of the object is approximated in four directions: 0, 45, 90, and 135

degrees. This yields a displacement vector $d_i^{\perp}(x_i, y_i) = [\Delta x_i \Delta y_i]^{\Gamma}$ representing the normal displacement field of visible edges [5].

A 2D affine transformation is expressed:

$$\begin{bmatrix} x_i^{t+1} \\ y_i^{t+i} \end{bmatrix} = \begin{bmatrix} x & y & 0 & 0 & 1 & 0 \\ 0 & 0 & x & y & 0 & 1 \end{bmatrix} \Theta = A(x_i, y_i)\Theta \tag{2.2}$$

where $\Theta = (\alpha_1, \alpha_2, \alpha_3, \alpha_4, \Gamma_x, \Gamma_1)$ are the parameters of the affine model.

There is a linear relationship between two consecutive images with respect to Θ:

$$d_i(x_i, y_i) = A(x_i, y_i)\Theta' = A((x_i, y_i))\,[(1, 0, 0, 1, 0, 0)]^{\Gamma} \tag{2.3}$$

From this it follows that

$$d_i^{\perp} = n_i^{\Gamma} d(P_i) = n_i^{\Gamma} A(P_i)\Theta' \tag{2.4}$$

where n_i is a unit vector orthogonal to the edge at a point P_i. From Equation 2.4 we can estimate the parameters of the affine model, $\widehat{\Theta}'$ using a M-estimator ρ:

$$\widehat{\Theta}' = argmin\Theta' \sum_i \rho(d_i^{\perp} - n_i^{\Gamma} A(P_i)\Theta') \tag{2.5}$$

Affine parameter estimation along this normal flow can be used to compute their positions at time $t+1$, and the pose space can be searched for the best fit [5]. This pose estimation is sent to GraspIt! where the best grasp is computed and executed, then image based-servoing is used to place the object within its desired pose.

Chapter 3

Approach: cytonGrasp System

3.1 World Design

As a part of this project, several objects were created in Open Inventor. They are all 1:1 scale obstacles for the Cyton Alpha arm to interact with in simulation as part of the DI Lab world. Four die, each labeled by their number of faces (d6, d8, d10, and d20 respectively), were measured and weighed, and were created from them. Each has its own sourcecode, which when compiled and executed, creates an Open Inventor file of the scene graph.

Die were chosen because they are small and light enough that the Cyton Alpha can grip them, while the multiple face-sets of each time of dice make grasping them an interesting problem. Gaming die from a local hobby shop were selected to give any future experiments a degree of repeatability, as any dice matching the models created would be widely available. Figure 3.1 shows the icosahedron (d20) scene graph.

A small wooden table was also measured then modeled precisely in Open Inventor, raising the dice up to where they could be easily grasped by the mounted Cyton Alpha atop the Pioneer robot. Finally, a simplified Pioneer model was made; while inexact, it elevates the arm model to the proper height and offered rough coordinates for the Pioneer robot in simulation.

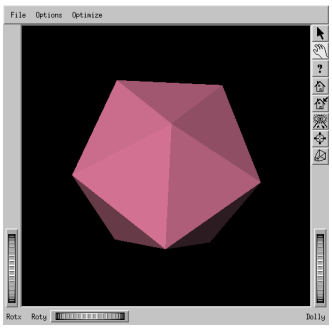

Figure 3.1: d20.iv scene graph in ivviewer

An approximation of the Pioneer robot is also implemented, which is a simple set of cubes representing the red robot's outer shell with a blue cube for the laser upon which the Cyton Alpha sits. The height of the arm, which is the only relevant information about from the Pioneer that the arm requires for grasping tasks, was carefully measured within a tenth of a centimeter. The simulated arm sits upon the box Pioneer, which in turn can be moved to any position in the XZ plane and rotated to different orientations.

3.2 cytonGrasp Architecture

A core idea of the cytonGrasp system is that the system can seamlessly switch between controlling the real and simulated arm. In order to do this, the same information needs to be available across both systems. This is already implemented to some degree, by having the simulated arm share its joints with the Cyton Controller's design. The program's API uses a hierarchical structure of joints and end effectors as sub-nodes for arms and objects. Boolean flags at the highest level control whether the real or simulated arm responds to commands or returns values. In this way the two arms can move independently, allowing for simulated preprocessing in addition to separate operation, or can be used to mimic each other to monitor for correctness or additional information as shown in [5].

Each object, obstacle, and end effector has a pose, representing its XYZ location in space as well as its orientation. This pose structure is shared by the Pioneer, gripper, dice and table, each of which can freely translate from Euler angles and Quaternion information. Each

joint has three versions of all location values: real, target, and simulated. When a movement command is issued to a joint, the target value is changed;k then, depending on which flags are active, the simulated and/or real arm approaches the target arm within a threshold.

The Cyton Control API interfaces with hardware by taking values from the end effector gripper information and is able to then calculate the best joint values for reaching that pose. The joint values are then in turn sent to the hardware interface moving the real joints. CytonGrasp mimics this behavior by making use of the effector and joint data structures. Instead of having the Cyton controller access the hardware controller directly, calculated joint positions are stored in the joint data structure where they can be freely sent to the hardware interface or back to GraspIt!, moving the simulated arm to match the calculated positions. This enables GrapIt! to not only provide information to the Cyton Alpha's controller but also benefit from its ability to process information about the Cyton Alpha's redundant joints.

This configuration for the flow of information between data structures allows GraspIt! to be fully utilized as a preprocessing tool, a key component of the simulator. GraspIt! can survey the GWS for gripping a particular object, calculating the best possible pose for the gripper to move the object in the desired way. This information is fed back to the end effector class in the form of a pose, which can calculate the Quanternion needed by the Cyton Alpha controller. Then, using Cyton Alpha's control structure, the optimal configuration of joints for reaching the targeted pose can be calculated and stored into separate (real and simulated) joints. This allows for a single go to target command to move each joint, real and simulated, into position for any configurations of the system.

By thinking of a real and simulated joint pair as a single entity (and doing the same with the real, simulated, and target end effectors), the flow of information becomes linear, as seen in Figure 3.2, regardless of the complexity of the task. A user could operate the arm, both in and out of simulation, using familiar commands and set end effector poses and joint angles, freed from both the complexity of the Cyton Controller's built-in data-types and the necessity of representing the positions of objects in Quaternions. Finally, by having

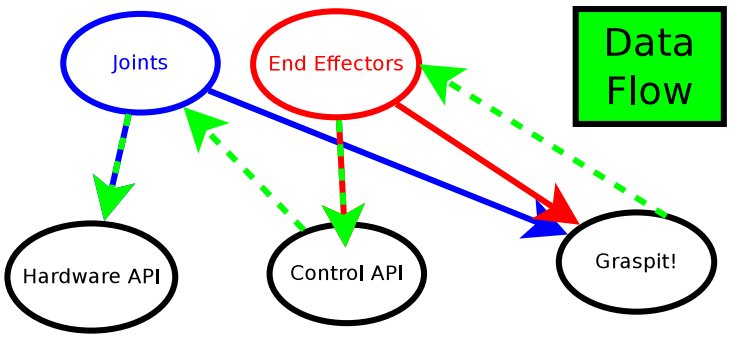

Figure 3.2: Flow of the planned grasp

the simulated arm moved in a method identical to the actual arm, the user needs to learn no different information to freely switch or combine the two.

As Figure 3.3 indicates, the joint and end effector organization enables the cytonGrasp system to have a very simple layout. All information relating to the Cyton Alpha (yellow) is funneled through the structures representing the eight joint pieces (blue) and the three end-effectors (red) and changed into a form usable by GraspIt!. The end effector, as well as the other location based elements of the GraspIt! world (the Pioneer and the dice) all have the same pose structure containing their information. This enables them to not only be accessed in the same way but have simple to use comparisons. The gripper will be at the object if the two poses are within a specified threshold.

This results in a system that is much more accessible, despite the various kinds of joints, models, and pieces (real and simulated) involved. Everything becomes either a joint or a pose, allowing the user to focus on application-specific details rather than worrying about the forms of information involved in using the Cyton Control API or GraspIt!.

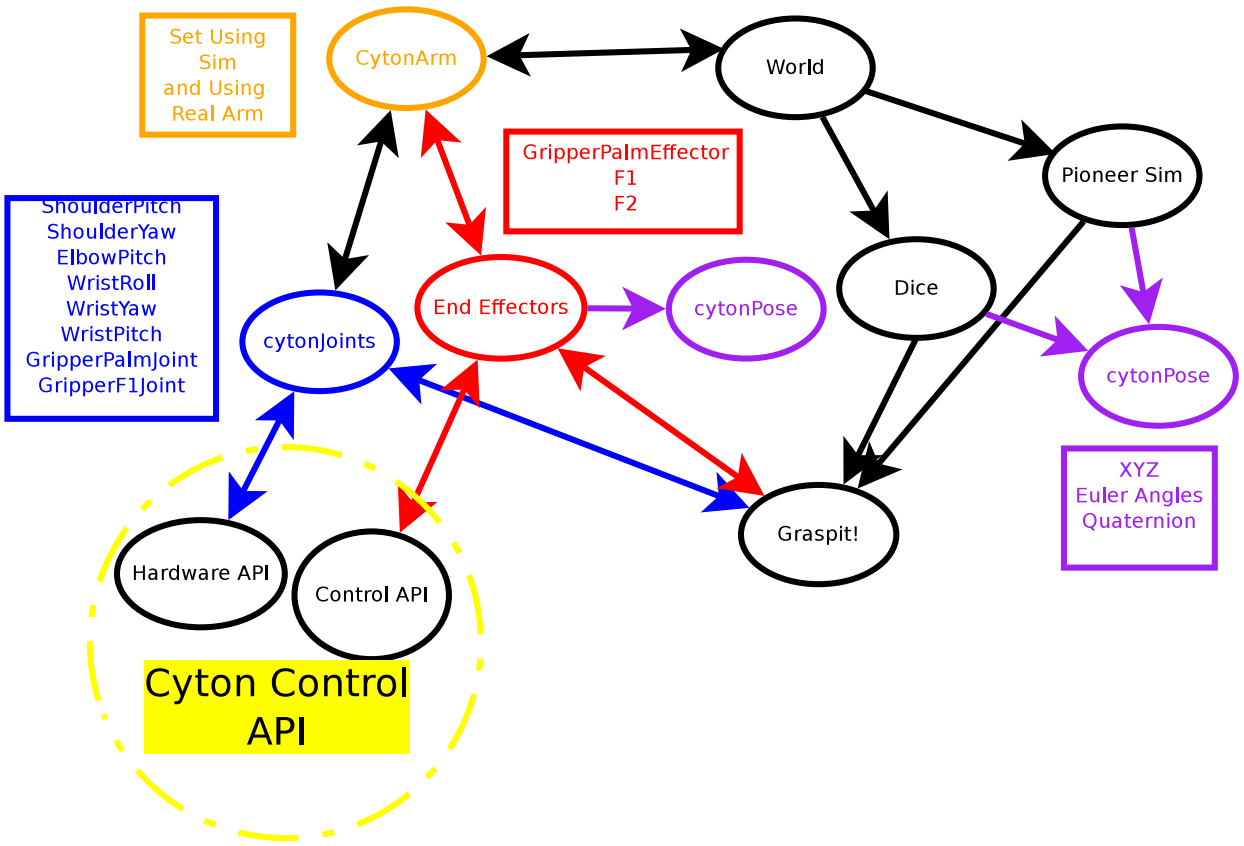

Figure 3.3: cytonGrasp Layout

3.3 Cyton Alpha Model

3.3.1 Cyton STL arm

The model used in the Actin Viewer packaged with the Cyton control API was created from a series of 97 binary STL images. The DI lab was provided with these images as part of an agreement with Mobile Robots from whom the Cyton Alpha was purchased.

First, these files were converted from binary to ASCII STL, then, using a Windows batch utility, they were converted into an older Open Inventor format. At this point, Open Inventor could read them and save them into the proper IV format so that they could be used with GraspIt!.

The IV files represented each individual screw, bolt, and piece of the Cyton Alpha, not simply the joints. Additionally, the file conversion caused certain bits of data such as

color and some position information to be lost. The full Cyton Alpha model needed to be reassembled in Open Inventor then converted to a more GraspIt! friendly form.

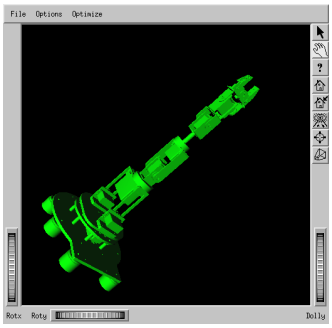

Figure 3.4: fullCyton.iv scene graph in ivviewer

From these models, eleven models for the robot were created (for the arm itself: the base, seven revolute joints, and two prismatic joints connected to a single fixed joint in the gripper). Fitting with the GraspIt! convention, each major degree of freedom on the Cyton Alpha was modeled as a single piece. These pieces were then moved into their correct configuration and two alternate color schemes were created. The first is a metallic gray color that closely resembles the original model, the other shows each of the newly defined pieces of the simulated arm clearly by giving each one a unique color. The orginal model pieces all defaulted to green due to their lack of color information (see Figure 3.4), in addition to other location and tuning adjustments, color was added to the final model (Figure 3.5).

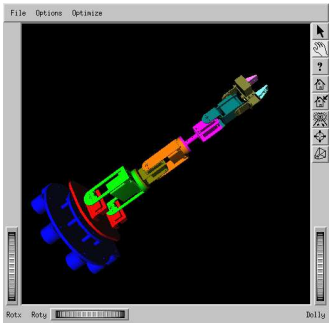

Figure 3.5: colorCyton.iv scene graph in ivviewer

3.3.2 Robot XML file

Any simulated robot in GraspIt! is represented as a series of DOFs linked together in kinematic chains. A DOF represents a single joint linked as a dynamic body to the base link, referred to as a "palm" for hand models.

The various IV files make up each joint in the kinematic chain as defined by an overall Robot configuration XML file, which uses Devanit-Hartenburg parameters to define the relationship between the joint and each previous joint out of the base. In general, a single robot is defined by three elements:

- the palm, the single pointer to the base joint:
- definitions of the DOF

  The DOF information supplements the object properties of each joint.

  - type: the kind of joint present. Joints can be either coupled or rigid (r) which is the default for a single DOF joint
  - defaultVelocity: the velocity used in an autograsp operation, the predefined speed of the joint for the simulated hand, a scaled value out of 100
  - maxEffort: this defines the force the DOF can apply at each joint it is connected to
  - Kp and Kd: coefficients for the PD force controller built into the joint's motor
- the kinematic chain

  Each kinematic chain is contained within its own tag, with the following properties which define how each joint relates to the others:

  - transform: the location of the origin of the palm in XYZ space. This is where the first joint in the chain is placed. It can contain a translation, a rotation, or both.
  - joint: this is a sub-tag of a kinematic chain containing the properties relating to that joint's location in the chain.

    Joints contain the following properties:

    - type: this is either Revolute or Prismatic relating to how the joint moves
    - The Devavit-Hartenberg (D-H) parameters of this joint, as subtags
    - minValue: the lower limit of motion for this joint
    - maxValue: the upper limit of motion for this joint
  - link: the link tag has a property "dynamicJointType", which defines the type of joint each link is within the kinematic chain

Each of these tags has values directly ported from either [1] or the CytonViewer's properties. Our simulated arm, like the real arm, has ten DOFs, seven (all revolute joints) in the arm and three in the gripper (1 fixed and the two prismatic "fingers") all of which connect to the base of the bottom turntable. The max effort, Kd and Kp, values for each of these joints are estimated within our simulation due to the fact that the PD controller force values were not available.

The min and max joint values for each kinematic are derived from the literature as described above. Each piece is linked, in order, up to the arm using the lowest turntable as the "palm." The default mass, center of gravity, and other object properties defined in the joints' individual XML files are computed in GraspIt! based on the individual joint shapes rather than defined within the XML.

3.3.3 Devavit-Hartenberg Parameters

Devait-Hartenberg parameters are a convention for specifying frames of reference between two joints in robotic applications. Each joint pair is a homogeneous transformation represented as a product of four basic transformations. This allows for a minimal representation via an exploitation of the common normal between two lines. It is currently the only way of specifying joint relations to GraspIt!.
The frame of reference is as follows:

- the z-axis is the direction of joint rotation

- the x-axis is the common normal between the two joints

  $z_n = x_n \times x_{n-1}$

- the y-axis follows from the right hand system.

The transformation from one joint's frame of reference to the next is then defined by the following four D-H parameters:

- θ: angle about previous z from x_{n-1} to x_n
- d: offset from previous z to the common normal
- α: angle about common normal from z_{n-1} to z_n
- a: (r) length of common normal x_n

The model of the Cyton Alpha arm is more complex than most example hands in GraspIt!. While the gripper, due to its limited mobility and small number of DOFs, is very simple and easy to model, the arm itself is kinematically redundant at multiple points resulting in a fair degree of complexity. Additionally, most of GraspIt!'s operations concern themselves with gripping and end effector positions and postures, thus there is very little support for complex arms at this time. The Cyton Alpha's arm, however, is in direct control of its gripper's position and thus must be a factor in any grasp planning or other simulated work.

The Cyton Alpha can be thought of as three sets of a repeating pair of joints between the base and the gripper. A turntable, with a limited rotation around the y-axis is always followed by a shoulder joint that bends in and out of the $x-z$ plane. All of the arm's regular joints are co-planar in the y axis. This results in several joints, many of which provide similar functionality, which share axes of rotation. Additionally, each joint's axis of rotation is 90 degrees from the prevoius joint, a convention not well suited for D-H parameters.

Due to numerous issues with defining the 3D model of the Cyton Alpha arm, specifically the complex nature of the D-H parameters and the definition of the joint relationships, a need for a simpler arm arose. The end result was the fauxCyton model, which represents

each of the seven joints in the arm sans, gripper with greatly simplified geometry using cubes for shoulder joints and cylinders for turntables and no interconnecting points at the links. With the aid of GraspIt!'s managing director, the D-H parameters for our simulator are being fine-tuned using this model to provide a more accurate simulation of the Cyton Alpha.

Chapter 4

Results

While the simulated world is still, partly, under construction, Figures 4.1 and 4.2 indicate the depth of the cytonGrasp's design. The simulation being built to exact scale will allow for countless approaches for planning or grasp synthesis within the box world. Due to the scale and complexity of the simulation, it can mirrior the real world and allow for a multitude of operations.

The GraspIt! user manual has a section which cautions readers that GraspIt! is not an off the-shelf product, but rather a large codebase and set of tools that can aid in modeling and developing algorithms and approaches [2].

Figure 4.1: Box world in DI Lab

Though the Cyton Arm is modeled in GraspIt! and many features are correct and ready to be used, it is increasingly likely that in order to address an actual problem or test an algorithm, or approach, the code will require some modification. The cytonGrasp system is set up to make use of a controller that is more than likely several years out of date, or

at minimum has no mechanisms allowing it to be edited or adapted to use new controller techniques. To this end, the Cyton Control API can be completely separated from the code at no real loss to the functionality of the rest of the program. The code for GraspIt! itself has yet to be changed in any meaningful way, but this could very easily be required in the work for which this system was developed.

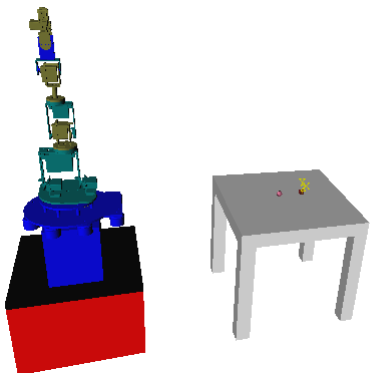

Figure 4.2: Box world in Simulation

A key issue with the arm in GraspIt! simulation, which a small work around corrects, is that the kinematic configuration for the Cyton arm cannot be modeled exactly. There is a bug in GraspIt! which does not allow for the particular sequence of perpendicular axes of rotation to build correctly. While a pair of fake joints (joint with no accompanying IV geometry files and thus no substance when modeled) allow for this to be corrected by setting the errant joint to a valid reference frame, there is no guarantee that GraspIt! will have this bug in the future, especially as the repair came from the current lead developer of the software.

This is but one small issue, which requires an in-depth technical explanation of the cytonGrasp system in order to maintain or change the software. Thus, a technical manual, while outside the scope of this thesis, is underway to supplement others in accessing and using GraspIt!, ROS, and the cytonGrasp system.

As stated in Section 1.4, since the purpose of this system is to provide tools for easing the use of both systems, it would not do for the cytonGrasp project itself to be inaccessible. The manual will include:

- An overview of the cytonGrasp controller similar to the models presented in Figure 3.3, but which includes directed links to the existing API
- A detailed explanation of how to detach the cytonGrasp API from either the Cyton Alpha Controller, or the GraspIt! Simulation support so that a user may make use of one end without invoking the other
- API documentation created by Doxygen for the system as a whole alongside implemented Cyton API and GraspIt! function lists for easy reference and explanation
- An overview of the specific set up for the CytonArm.xml and how it builds the GraspIt! Model
- An explanation and API documentation for the Open Inventor C++ code used to build the objects

Through this manual, and the efforts of my current and future colleagues the system will continue to evolve and will soon reach a more stable state.

Chapter 5

Conclusions and Future Work

5.1 Impact

This thesis primarily acts as an introduction to what is possible with GraspIt! guiding new users through current work in grasp planning. CytonGrasp is meant to act as a front end, making GraspIt! more user friendly for simple machine learning tasks. While there are many features of GraspIt! still under development and many new directions for learning in the domain of grasp planning, without the aid of a strong 3D simulation it would be difficult to develop or test new machine learning techniques or begin to approach these problems.

As stated in the approach, cytonGrasp is meant to be a box world, something simple enough that anyone with a Cyton Alpha arm can replicate it.

The models for the dice and table will be added to our own release of GraspIt! via ROS, and the models for our Cyton Alpha will be provided back to Mobile Robots so that using GraspIt! is an option for their customers as well.

The largest impact of this system is the various options for machine learning and planning tasks it provides. GraspIt! changes a robot operation issue into machine learning tasks and opens up avenues for future research involving grasp planning to the Distributed Intelligence Lab or anyone else who wishes to follow the work. The system could be extended to include the Barrett hand, the Nao robot, or any number of useful objects; it can even model human

hand data or any number of related machine learning tasks. GraspIt! and the core ideas of cytonGrasp transform arm manipulation from a mobile robot task into a machine learning task.

5.2 Conclusions

GraspIt! shows remarkable promise. During the course of this project, the number of schools participating in some form of research with GraspIt! has nearly tripled. The numbers of institutions using ROS are even greater. This is the direction new techniques and technologies in machine learning will be taking.

The learning curve for ROS and GraspIt! is steep, difficult enough that I wish our program had incorporated developing exposure to it in the same way that using Player and Stage was such a large part of my later education. The opportunities for learning and sharing information with our colleagues in new and exciting ways, however, make the obtuseness of the system seem like a trivial matter.

GraspIt! and many similar simulation toolkits are still in their early stages, and the system is still immature enough that it would not be surprising if it were a year or more until another stable release. ROS and programs like GraspIt! are going to be the future of robotics research.

5.3 Future work

There are many ways in which the cytonGrasp system or GraspIt! itself can be refined or extended. First, there are many components or subsystems that could be added which would aid in future research:

- Kp and Kd variables for KD force controllers need to be calculated. Currently they are only a very rough approximation.

- Stronger tutorials for Open Inventor and clear instructions for extending the GraspIt! simulated world; for this system to be useful it needs to be accessible
- A cleaner method of storing the joint and end effector positions could be added
- API simplification is needed in some areas
- Attaching alternate means of hardware desperation extending to arms beyond the Cyton Alpha
- A ROS message interface exists, but was not implemented as part of this work.

Secondly, there are several promising research directions based on this work such as the following:

- Using the Kinect or an overhead camera to track objects in simulation as illistrated in [5]
- Grasp planning via simulated annealing and Quaternion end effectors
- Eigengrasps and the GWS quality metrics introduced alongside the GraspIt! simulator provide ample research extensions in grasp planning, grasp synergy, and transfer learning applications.
- Several interesting machine learning techniques make use of the reduced dimensionality of eigengrasps. An SVM approach for grasping arbitrary objects has already been tested by Willow Garage, which could be extended to our arm, as well as various learning techniques for grasp planning.

Bibliography

Bibliography

[1] Robai, "Cyton alpha 7D 1G operations manual," 2009. 1, 22

[2] M. Ciocarlie and A. Miller, "Graspit! user manual version 2.1," 2011. 2, 3, 25

[3] J. Wernecke, *The Inventor Mentor:Programming Object-Oriented 3D Graphics with Open Inventor (TM), Release 2*. Addison-Wesley Publishing Company, 1993. 7

[4] A. Miller and P. Allen, "Graspit!: A versatile simulator for robotic grasping," *IEEE Robotics and Automation Magazine*, vol. 11, no. 4, pp. 110–122, 2004. 10, 11

[5] P. K. A. Danica Kragic, Andrew T. Miller, "Real-time tracking meets online grasp planning," *In Proceedings IEEE International Conference on Robotics and Automation, Seoul, Republic of Korea*, pp. 2460–2465, 2004. 10, 13, 14, 16, 30

[6] C. G. Matei Ciocarlie and P. Allen, "Dimensionality reduction for hand-independent dexterous robotic grasping," *IEEE / RSJ Conference on Intelligent Robots and Systems*, 2007. 10, 12, 13

Appendix

Appendix A

Libraries

A.1 Cyton Libraries

All libraries that end with .so.35 are symbolic links to their .2.4.0 counterparts, but are still required. The Qt Libraries which were removed from the final build are bolded for emphasis. 32 bit versions of each of these libs are located in ~/CytonArm/cytonarm/lib

- libboost_date_time.so
- libboost_filesystem.so
- libboost_iostreams.so
- libboost_program_options.so
- libboost_regex.so
- libboost_serialization.so
- libboost_signals.so
- libboost_system.so
- libboost_thread.so
- libcurl.so.3.0.0
- libcv.so.1.0.0
- libcxcore.so
- libecControl.so
- libecConvertSimulation.so
- libecConvert.so
- libecConvertSystem.so
- libecCytonControlInterface_render.so
- libecCytonControlInterface.so
- libecCytonHardwareInterface.so
- libecFilterStream.so
- libecFoundCore.so
- libecFunction.so
- libecGeometry.so
- libecGrasping.so
- libecImageSensor.so
- libecInputDevice.so
- libecInputDevices.so
- libecLoader.so

- libecManipulator.so
- libecMatrixUtilities.so
- libecMeasure.so
- libecRendCore.so
- libecRender.so
- libecSensCore.so
- libecSerial.so
- libecSimulationAnalysis.so
- libecSimulation.so
- libecSimulationStudy.so
- libecSocket.so
- libecStream.so
- libecTransport.so
- libecViewerCore.so
- libecVisualization.so
- libecVrml97.so
- libecWalking.so
- libecXml.so
- libhighgui.so.1.0.0
- libjasper-1.701.so.1
- libOpenThreads.so.10
- libosgDB.so.2.4.0
- libosgDB.so.35
- libosgFX.so.2.4.0
- libosgFX.so.35
- libosgGA.so
- libosgGA.so.2.4.0
- libosgGA.so.35
- libosgManipulator.so.2.4.0
- libosgManipulator.so.35
- libosgParticle.so.2.4.0
- libosgParticle.so.35
- libosgShadow.so.2.4.0
- libosgShadow.so.35
- libosgSim.so.2.4.0
- libosgSim.so.35
- libosg.so.2.4.0
- libosg.so.35
- libosgTerrain.so.2.4.0
- libosgTerrain.so.35
- libosgText.so
- libosgUtil.so.2.4.0
- libosgUtil.so.35
- libosgViewer.so.2.4.0
- libosgViewer.so.35
- **libQtCore.so**
- **libQtCore.so.4**
- **libQtCore.so.4.4.0**
- **libQtGui.so**
- **libQtGui.so.4**
- **libQtGui.so.4.4.0**
- **libQtOpenGL.so**
- **libQtOpenGL.so.4**
- **libQtOpenGL.so.4.4.0**
- libtiff.so.3.8.2
- libttf.so.2.2.0
- libz.so.1.2.3

Appendix B

ROS Packages

B.1 GraspIt! Packages

GraspIt! depends upon these packages; now that it has been officially released you need only install ros-diamondback-graspit-simulator 0.1.0-s1310033431 or a higher version from the package manager.

- arm_navigation
- common
- common_msgs
- control
- diagnostics
- diagnostics_monitors
- documentation
- driver_common
- executive_smach
- executive_smach_visualization
- geometry
- geometry_tutorials
- graspit_simulator
- image_common
- image_pipeline
- image_transport_plugins
- joystick_drivers
- kinematics
- laser_pipeline
- motion_planners
- motion_planning_common
- navigation
- object_manipulation
- perception_pcl
- physics_ode
- pr2_common
- pr2_controllers
- Toppr2_mechanism

- robot_model
- ros_comm
- ros_release
- rx
- simulator_gazebo
- simulator_stage
- slam_gmapping
- sql_database
- trajectory_filters
- vision_opencv
- visualization
- visualization_common
- visualization_tutorials

Vita

Nicholas Overfield was born in Evansville, Indiana on February 13, 1986. He came to the University of Tennessee in 2005 after receiving the Hope and Volunteer scholarships. He was an active member of the Computer Science department, taking on leadership rolls in the ACM and many other student organizations. After receiving his B.S. in Computer Science and English from UTK in 2009, he was awarded the Bodenheimer Fellowship and stayed at UTK to complete his Masters Degree in Computer Science as a member of the Distributed Intelligence Laboratory under Dr. Lynne Parker.