VISUALIZATION TOOLS FOR MOVING OBJECTS

A Thesis

by

AIMÉE VARGAS ESTRADA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2005

Major Subject: Computer Science

VISUALIZATION TOOLS FOR MOVING OBJECTS

A Thesis

by

AIMÉE VARGAS ESTRADA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2005

Major Subject: Computer Science

ABSTRACT

Visualization Tools for Moving Objects. (December 2005)

Aimée Vargas Estrada, B.S., Universidad Nacional Autónoma de México (UNAM)

Chair of Advisory Committee: Dr. Nancy M. Amato

In this work we describe the design and implementation of a general framework for visualizing and editing motion planning environments, problem instances, and their solutions.

The motion planning problem consists of finding a valid path between a start and a goal configuration for a movable object. The workspace is, in traditional robotics and animation applications, composed of one or more objects (called obstacles) that cannot overlap with the robot.

As even the simplest motion planning problems have been shown to be intractable, most practical approaches to motion planning use randomization and/or compute approximate solutions. While the tool we present allows the manipulation and evaluation of planner solutions and the animation of any path found by any planner, it is specialized for a class of randomized planners called probabilistic roadmap methods (PRMs).

PRMs are roadmap-based methods that generate a graph or roadmap where the nodes represent collision-free configurations and the edges represent feasible paths between those configurations. PRMs typically consist of two phases: roadmap construction, where a roadmap is built, and query, where the start and goal configurations are connected to the roadmap and then a path is extracted using graph search techniques.

To Marco for all his love and support. Thank you for being always there for me.

ACKNOWLEDGMENTS

I want to thank the Mexican government, especially to Consejo Nacional de Ciencia y Tecnología (CONACYT), for the scholarship that allowed me to pursue one of my dreams.

My respect and gratitude go to an intelligent woman, my advisor Dr. Nancy Amato, for her guidance and support since the very first day I met her.

I also want to thank Dr. Donald H. House and Dr. John Keyser, two great professors I had the privilege to meet, for the comments and feedback that enriched this thesis.

I want to acknowledge the work of Jyh-Ming Lien who was my mentor and was always willing to help me.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION 1

A. Contribution 3
B. Outline of Thesis 4

II PRELIMINARIES AND RELATED WORK 5

A. Basic Motion Planning Concepts 5
B. Related Work 7
1. Motion Strategy Library 7
2. A PRM Planner for Rigid Objects in 3D 9
3. Vizmo 9
4. Modeling Articulated Objects 10
a. Link Connection and Denavit-Hartenberg Parameters 11
b. Transformations 13
5. Collision Detection 15

III REQUIREMENTS 17

A. Overview of Design Process 18
B. Definition and Analysis of Requirements and Functionality 19
C. Modular Organization of Functionality 27

IV DESIGN 29

A. Design of the User Interface 29
B. Class Design 30

V IMPLEMENTATION 33

A. I/O Files 33
1. File Formats 33
2. Creation of New Files 38
B. Geometric Model and Rendering 38
1. Object Selection 39
C. Class Implementation 40
1. Implementing Articulated Objects 42

CHAPTER Page

D. Interfacing Vizmo++ with our Motion Planning Library . 44
E. Status 45

VI APPLICATIONS: THE CAMPUS NAVIGATOR 51

VII CONCLUSION 54

REFERENCES 56

VITA 62

LIST OF TABLES

TABLE Page

I Visualization Tools for Motion Planning Available 11

LIST OF FIGURES

FIGURE Page

1 MSL 8

2 Vizmo 10

3 Denavit-Hartenberg parameters [1] 12

4 Joint reference frame [1] 13

5 Link frames: frame i is attached to link i [1] 14

6 Structure of a tree-like linked object [2] 15

7 Vizmo++ GUI 18

8 Function hierarchy diagrams 20

9 Query shown for the *narrow* environment 21

10 Path and movie recording interface for the *narrow* environment . . . 23

11 A roadmap node is selected and the *translation* tool is shown. 25

12 *narrow* environment: saving options 26

13 Class hierarchy (solid lines represent inheritance and dashed lines denote a "use" relationship) 31

14 Example of a BYU file describing a cube 34

15 Environment file description and visualization 35

16 Map file example and visualization 36

17 Path file description and visualization 37

18 Multibody description 40

FIGURE Page

19 Path image generated by Vizmo++ 53

CHAPTER I

INTRODUCTION

Visualization tools are meant to transform quantitative sets of data into meaningful images. Researchers in many areas can use tools to visualize and manipulate 2D and 3D data models to better understand their problems, to test hypotheses in virtual models or to explain their results. For example, researchers in fields like robotics and computational biology use visualization tools to visualize routes taken by their robots or their models of interacting molecules, respectively.

Motion planning is the problem of finding feasible paths for movable objects among obstacles [3]. A movable object could be a robot, a digital character or actor, molecules such as proteins or RNA, etc. Here we will refer to them as either robots or movable objects. The concept of feasible path is determined by the application: in robotics path planning, feasible paths are typically collision-free sequences of robot poses, whereas in computational biology we may be interested in the energetic feasibility of the path, e.g., paths through low energy regions of the conformation space.

The most successful approach to the motion planning problem is randomized planning. There are several randomized techniques available. Most of these techniques can be classified as either tree-based [4, 5, 6, 7, 8] or map-based [9, 10, 11, 12, 13, 14, 15, 16]. In addition to robotics, these techniques have been successfully applied in fields like virtual prototyping [17], graphic animation [18], medical surgery [19], and even computational biology [2].

This thesis follows the style and format of *IEEE Transactions on Robotics.*

Although motion planning deals with geometric models that represent obstacles and robots in the workspace, many motion planning methods find it convenient to work in the robot's Configuration Space (C-Space) [20]. In this case, planners work with sets of configurations that correspond to workspace positions and orientations of the movable object. C-Space is a useful abstraction that enables all motion planning problems to be treated in a standard manner and facilitates the development of general planning methods.

Motion planning methods use and generate information that represents a solution to a given problem. We are interested in visualizing the robot, the obstacles, and planner outputs (e.g., paths and roadmap). The visualization should interpret and present views of the information through a tool that allows researchers to observe and gain better understanding of the instance of the motion planning problem and of the computations of the motion planning methods.

The visualization tool presented in this thesis can help users to evaluate planner outputs that otherwise would be hard to analyze. Planners generate roadmaps or graphs whose nodes represent robot configurations and whose edges represent paths (sequences of configurations) connecting configurations. A path, which the robot will follow to reach its goal, is a sequence of collision-free configurations. Since the configuration of even a rigid body in 3D is represented by six values, without a visualization tool it would be difficult to understand the quality of the solution found by a given planner just by looking at the generated raw data. Indeed, configurations map to objects placed in a 3D workspace and one natural way to understand motion planning methods is through 3D visualization. That is the approach taken in our work.

A. Contribution

Research in motion planning is very active and there are many planners available. Since many randomized planners share a common structure (e.g., they generate a graph of robot configurations and solution paths are sequences of configurations), a tool supporting the common needs would potentially assist many researchers. There is a need to support a variety of movable objects and basic visualization and manipulation of the elements of the environment. Some research groups have developed their own visualization tools [21, 22, 23]. However none of them support both visualization and editing of the solutions and environments. Also, none of them integrate collision detection into the visualization, animation, and editing tools.

The main contribution of this thesis is a new 3D tool for visualizing and editing motion planning environments, problem instances, and their solutions. Our tool offers a self-explanatory graphical user interface (GUI) that allows users to become familiar with it in a short period of time, and above all, provides most of the functionality they need. To our knowledge, there is no available tool that supports both visualization and editing of workspace environments.

Researchers in the motion planning community take advantage of visualization tools to see and compare solutions generated by different planners, and to analyze their methods by looking at the distribution of the nodes, the connectivity of the roadmap, and the quality of the path. The visualization tool we present in this thesis supports all these uses for a variety of planning methods and hence can be generally useful for the motion planning community. Ultimately, we hope to release our tool to other research groups to facilitate comparison of different planners.

Our tool, Vizmo++, was developed following Software Engineering [24] and Computer-Human Interaction [25] guidelines to implement a well-designed object oriented application that is easy to maintain and extend.

- Vizmo++ will help users in their understanding and evaluation of different planner strategies and solutions through straightforward visualizations and through visualization and animation of path configurations. For example, since different planners generate different distributions of nodes, roadmap visualization can help to better understand how each method works by looking at the distribution of the nodes and their connectivity.
- Vizmo++ will enable users to interact with and edit the environment. For example, it will let users manipulate obstacles and robot configurations, set queries, save new environments to be able to work on them later, or select and move nodes and thus editing existing or creating new paths and roadmaps.
- Vizmo++ will interface with the planners available in our motion planning library to enable users to run new queries. Our tool provides a convenient interface to select planners and set their parameters. Vizmo++'s structure eases the integration of new planner options and hence encourages and facilitates experimentation with new methods and parameters.

B. Outline of Thesis

Chapter II describes basic motion planning concepts and previous work. Chapter III presents the definition of requirements and functionality of our tool, and describe how Vizmo++ will model an articulated object. Chapter IV presents the design of our application. In Chapter V we present implementation details. Chapter VI presents a different application of Vizmo++: the Campus Navigator, which is a web based-application to help users to find their way across the Texas A&M campus. Chapter VII describes our conclusions and future work.

CHAPTER II

PRELIMINARIES AND RELATED WORK

A. Basic Motion Planning Concepts

Motion planning researchers are interested in visualizing the robot, the obstacles, and the solution path. In general, they are interested in the visualization and manipulation of the solutions generated for a given motion planning problem.

The motion planning problem consists of finding a valid path between a start and a goal configuration for a movable object [3]. A configuration is defined as an n-tuple of values that represents the position and orientation of the object. The workspace is the place in which the robot moves. In traditional robotics and animation applications, the workspace is composed of one or more objects (called obstacles) that cannot overlap with the robot. The configuration space (C-space) [20] is an n-dimensional space (n being the number of degrees of freedom (DOF) of the robot) consisting of all (i.e., feasible and infeasible) robot configurations. The robot is represented as a point in C-space.

The motion planning problem is known to be PSPACE-hard [26] – at least as hard as an NP-complete problem. All complete algorithms developed so far take exponential time in the number of DOF of the robot. A practical and broadly used approach to solve the motion planning problem is through randomization. Randomized algorithms have proved useful in finding approximate solutions to intractable problems such as motion planning. In these methods, the movable object's C-space is sampled at random and the samples are connected to form collections of feasible paths in C-space. We can classify most randomized planners as either roadmap-based

[9, 13, 14] or tree-based [4, 6, 5]. Notable examples of roadmap-based and tree-based planners are the Probabilistic Roadmap Methods (PRMs) [9, 11, 12, 10, 13, 14, 15, 16], and the Rapidly-exploring Randomized Trees (RRTs) [5], respectively.

Probabilistic Roadmap Methods (PRMs) [9] are roadmap-based methods that generate a graph or roadmap where the nodes represent collision-free configurations and the edges represent feasible paths between those configurations. A roadmap may contain one or more connected components. The path will be a sequence of configurations that describe the set of movements that the movable object needs to perform to reach its goal. PRMs typically consist of two phases: roadmap construction, where a roadmap is built, and query, where the start and goal configurations are connected to the roadmap and then a path is extracted using graph search techniques.

Several PRM variants have been developed. The original PRM [9] samples nodes uniformly. Lazy PRM [11] avoids collision detection during roadmap construction to speed up the roadmap construction at the cost of slightly reducing the speed of query processing. Fuzzy PRM [12] introduces the concept of a "fuzzy" roadmap where roadmap nodes are validated but edges are not until queries are processed. C-PRM [10] builds a coarse roadmap only performing an approximate validation of nodes and edges, then, in the query phase, the roadmap is refined and validated only in necessary regions thus decreasing total processing costs and typically also improving performance. The same roadmap can be used (customized) to support different query requirements (e.g., variations in the clearance threshold). Other variants deal with the problem of narrow passages which are difficult to sample. Obstacle-Based PRM (OBPRM) [13] generates nodes on or near C-obstacle surfaces. The Medial Axis PRM (MAPRM) samples the configuration space uniformly and then samples are retracted onto the medial axis of the free space [27, 14, 28] or an approximation of it [29]. The Bridge Test method [15] has a hybrid sampling strategy to increase the density of the samples inside narrow passages. The Gaussian PRM [16] biases the samples close to the C-obstacles with a Gaussian distribution.

Tree-based planners grow a tree by picking a feasible configuration and nodes are added to the tree according to some expansion strategy. The first such method, Randomized Path Planner (RPP) [4], builds a graph that connects the local minima of a potential function that is defined over the robot's C-space. Rapidly-exploring Randomized Trees (RRTs) [5] grow a tree of samples starting from the robot's initial configuration and have been shown to be effective in exploring high-dimensional spaces. The Ariadne's Clew Algorithm [6] grows a search tree using genetic algorithms. An algorithm for expansive configuration spaces [7] tries to sample the space that is relevant to the query.

Since there are many algorithms available and they produce different types of graphs, there exists a need to evaluate their differences and their performance. One way in which researchers compare methods is by comparing the sample distribution, the graph structure and graph connectivity.

B. Related Work

There exist 3D visualization tools to support PRMs developed by different research groups. Some of them are available upon request and others can be downloaded from their web sites. Among those tools are the Motion Strategy Library [21], the PRM Planner for Rigid Objects [22] and Vizmo [23].

1. Motion Strategy Library

The Motion Strategy Library (MSL) [21] was developed by the research group of Professor Steven M. LaValle at the University of Illinois at Urbana Champaign. MSL

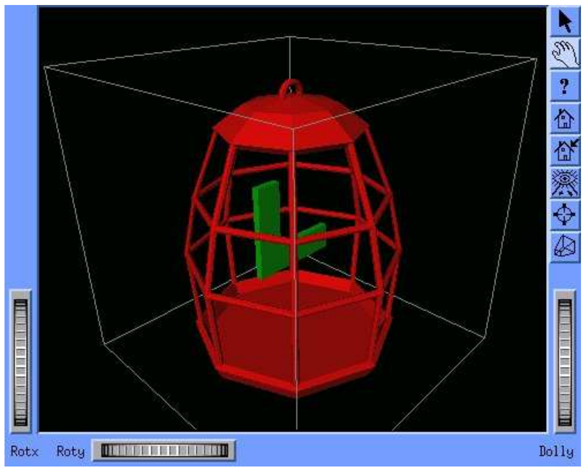


Fig. 1. MSL

was designed for easy development and testing of motion planning algorithms. MSL is open source and free.

MSL is an object-oriented software package developed in Linux that uses the following software packages:

- FOX GUI Toolkit [30] for the user interface.
- OpenGL [31] for a GL-based renderer, Open Inventor [32] which offers more accurate shading, and OpenGL Performer [33] for high performance graphics. Figure 1 shows the MSL's GUI using Open Inventor rendering.
- Proximity Query Package (PQP) [34] for collision detection.

MSL lets the user attempt to solve problems with any of its embedded planners based on Rapidly-exploring Random Trees (RRTs) [8], Probabilistic Roadmaps (PRMs), and Forward Dynamic Programming (FDP) [35]. Planners and some parameters can be configured through the interface.

MSL does not show the roadmaps or RRTs directly on the 3D-environment. Instead it allows users to generate a two-dimensional plot of the projection of the connectivity graph on the plane formed by any two variables selected by the user. This plot does not show any information about the obstacles, providing only information regarding the feasible configurations and connections.

MSL shows the 3D environment and gives the user the ability to control the view. However, it does not allow any other type of interaction. The interface is not user-friendly and some functions are available only through hot-keys.

2. A PRM Planner for Rigid Objects in 3D

The Physical and Biological Computing Group led by Professor Lydia Kavraki at Rice University, has developed the PRM Planner for Rigid Objects in 3D [22].

This application is UNIX-based and uses the following software packages:

- The XForms [36] GUI toolkit.
- Geomview [37] for model visualization.
- The RAPID [38] bounding box-based collision checker.
- libSVM for simple vector and matrix support (distributed with version 1 of RAPID).

Problems can be solved by using any of the embedded planners.

3. Vizmo

Vizmo (see Figure 2) was the first 3D visualization tool for motion planners developed in our research group [23]. This tool was implemented in C++, however its design was not fully object-oriented. Vizmo uses the following software packages:

- Tool Command Language Tcl/Tk [39] for the GUI.
- The Visualization Toolkit (VTK) [40] for 3D rendering.

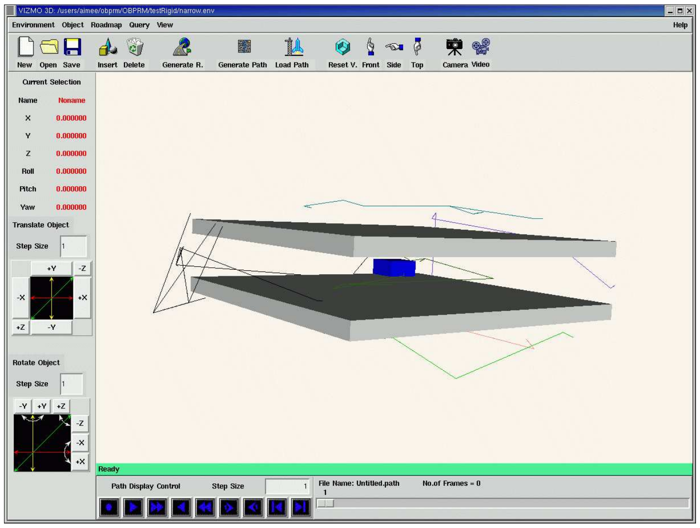


Fig. 2. Vizmo

Vizmo offers basic functionality, allowing users to interact with and manipulate some of the elements in the environment, such as changing the color of the objects and animating the path. However, some useful characteristics are missing.

Vizmo supports only rigid bodies and does not support our work with articulated objects, closed chain systems [41], and biomolecules. Vizmo does not support roadmap representation. For example, the configurations on the roadmap cannot be seen, and it animates the path but the path configurations cannot be visualized. Also, the GUI needs to be improved so it can be more intuitive. Finally Vizmo was difficult to maintain and extend so we decided to develop a new version that could support previous and new needs.

Table I summarizes the characteristics of the visualization tools presented.

4. Modeling Articulated Objects

The moving objects Vizmo++ can display are of two types: rigid and articulated. The geometry of a rigid body is given as a polyhedron described as a set of triangles. If the moving object is composed of more than one body, then the description of each

TABLE I

Visualization Tools for Motion Planning Available

Tool	Robot		CD* Visualization	Editing
	Rigid	Articulated		
MSL [21]	yes	yes	no	no
PRM Planner [22]	yes	info. not available	no	no
Vizmo [23]	yes	no	no	no
Vizmo++	yes	yes	yes	yes

* Collision Detection

body is given in an independent input file. Connection information is also given so that each rigid body can be linked to another rigid body(ies) to form an articulated object. As part of our work, we had to define the way we would describe articulated objects.

An articulated object can be defined as a set of bodies connected by joints forming a chain. We will call those bodies links. The chain can be connected by different types of joints. The most used types of joints are *revolute*, which represents a single rotational DOF about a given axis, and *prismatic*, which represents a translational DOF along a given axis.

Each joint has a joint axis which is a vector about which a link i rotates relative to link *i-1*. We can describe the relationship between the links of a chain through their joint axes. We use Denavit-Hartenberg (DH) notation [42, 2] to specify the connection between links.

a. Link Connection and Denavit-Hartenberg Parameters

The DH parameters are four numbers that describe the relationship between each pair of links that compose an articulated object: two distances and two angles (a_i, d_i,

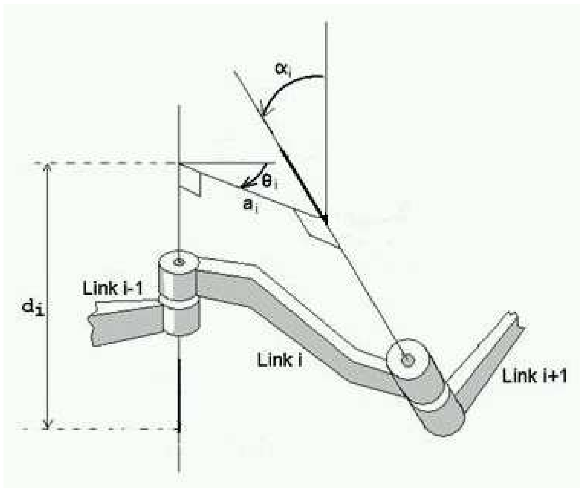


Fig. 3. Denavit-Hartenberg parameters [1]

θ_i, α_i) (see Figure 3). Following Craig [42], a_i (the link length) is a distance that can be measured between any two axes in 3D space, this distance is measured along the common perpendicular between the two axes. That perpendicular is unique except when the two axes are parallel. α_i (also called the link twist) is the angle formed between axis *i* and axis *i-1*. Links that are neighbors have a common joint axis. d_i (the link offset) is the distance along this common axis from one link to the next. θ_i (also called the joint angle) is the parameter that defines the amount of rotation about the common joint axis between link *i-1* and link *i*.

To describe the location of a link in a serial chain, reference frames are attached to each link. There is a convention for naming and assigning those frames (see Figure 4):

- Frame *i* is attached to link *i*.

- The origin of frame *i* will be located where a_i intersects the joint *i* axis.

- The *z*-axis of frame *i* will always be in the joint axis.

- The *x*-axis of frame *i* will point along the common perpendicular a_i from joint *i* to joint *i+1*.

- The *y*-axis is formed by the right-hand rule.

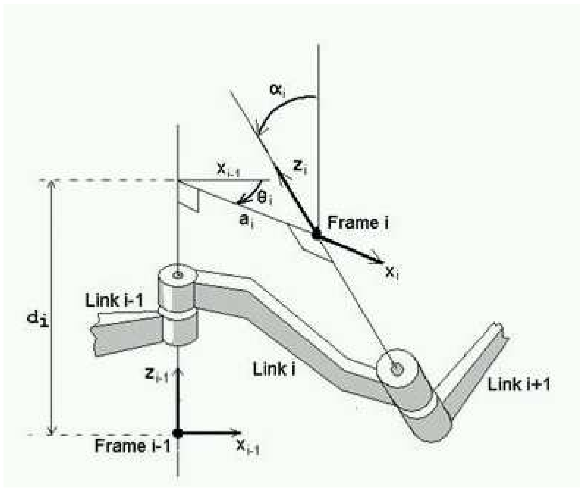


Fig. 4. Joint reference frame [1]

b. Transformations

Once each link has a reference frame attached to it, we can determine the transform that will define the location of frame *i* relative to frame *i-1*. This transformation will be a function of the four parameters we described before (a_i, d_i, θ_i, α_i). We will define the transformation as four transformations (see Figure 5):

1. A translation along z_{i-1} with displacement d_i
2. A rotation around z_{i-1} with angle θ_i
3. A translation along x_{i-1} with displacement a_i
4. A rotation around x_i with angle α_i

Multiplying the four previous transformations, the following general transformation is obtained (see [42] equation 3.6):

$$\mathbf{T_i^{i-1}} = \begin{pmatrix} cos\theta_i & -sin\theta_i & 0 & a_{i-1} \\ sin\theta_i cos\alpha_{i-1} & cos\theta_i cos\alpha_{i-1} & -sin\alpha_{i-1} & -sin\alpha_{i-1} d_i \\ sin\theta_i sin\alpha_{i-1} & cos\theta_i sin\alpha_{i-1} & cos\alpha_{i-1} & cos\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (2.1)$$

This transformation contains position and orientation information. The position is

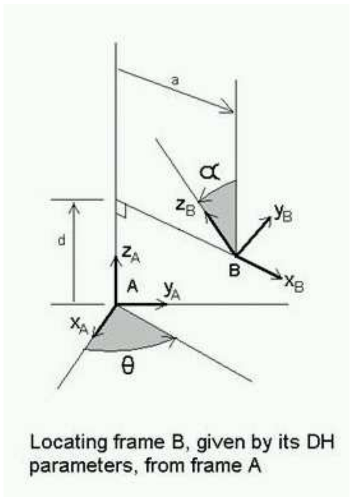


Fig. 5. Link frames: frame i is attached to link i [1]

a 3×1 vector and the orientation is a 3×3 rotation matrix composed of three unit vectors of the coordinate system i relative to *i-1*, set as columns in the matrix.

Now, to know the transformation that relates frame n to frame 0, we multiply the link transformations to define a single transformation (see [42] equation 3.8):

$$T_n^0 = T_1^0 T_2^1 T_3^2 ... T_n^{n-1} \tag{2.2}$$

The above describes how a serially linked chain is typically modeled. To describe a tree-like model we follow the approach presented in [2] where an extra reference frame is added to each joint as described below and as shown in Figure 6. This approach decouples the link's body frame and its joint specification, allowing each joint to have an independent representation.

- A body frame is attached to each body i, we call it F_i, which is independent of any joint connection. The center of mass is usually used as the origin of this body frame.
- Reference frames are attached to each joint that connects $body_i$ and $body_j$. To define the joint transformation, "DH-frames", DH_i and DH_j, are assigned to

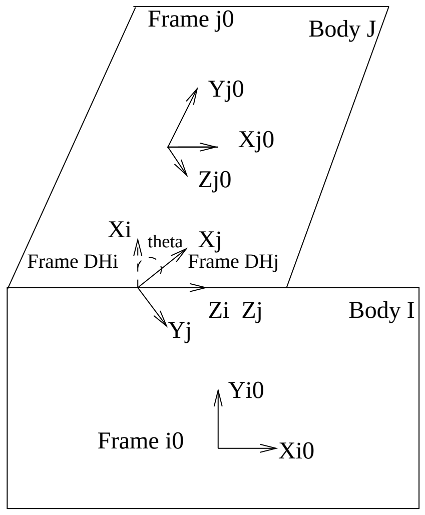


Fig. 6. Structure of a tree-like linked object [2]

$body_i$ and $body_j$ respectively. DH parameters define the connection between those DH frames.

- To obtain the transformation from F_i to F_j (T_j^i), we multiply the transformations (as in 2.2) we are given in the environment input file: from F_i to DH_i ($T_{DH_i}^{F_i}$), then to DH_j and to F_j:

$$T_j^i = T_{DH_i}^{F_i} T_{DH_j}^{DH_i} T_{F_j}^{DH_j} \tag{2.3}$$

5. Collision Detection

The validity test is the most expensive and repetitive operation in solving the motion planning problem. For example, during the construction of PRM roadmaps, configurations are repeatedly tested to keep only valid nodes and edges. A robot configuration is not valid if it violates any given constraint. One common constraint is that the robot should not overlap with the obstacles in the environment. This particular constraint is tested with libraries specialized in collision detection. Efficiency and robustness are two important factors to consider when choosing a collision detection library because collision detection is repetitive and expensive.

In Vizmo++ the validity test is also a repetitive operation since, if the robot or a node is being moved after the collision detection option was turned on, the validity test is computed until the option is turned off. In the workspace we model each robot and obstacle as a *multibody*, which is a collection of one or more bodies. Multibody descriptions (position and orientation) and connections are given with respect to a global coordinate frame. Each body has its own properties such as position, orientation, and connection information (for the case of articulated objects). We perform collision detection by testing each body of a multibody against each obstacle in the environment. We use an external collision detection library called RAPID (Robust and Accurate Polygon Interface Detection) [38].

RAPID is a small library which is easy to install and use. It computes a hierarchical representation of models using oriented bounding box trees (OBBTrees). An OBB is a rectangular bounding box with arbitrary orientation placed around a collection of polygons. At run time, the algorithm traverses two trees (one for each element that is being tested for collision) and tests for overlaps between their oriented bounding boxes [43].

We chose to use RAPID because of its performance. In [44], M. Reggiani *et al.* experimentally evaluated collision detection libraries, within the context of motion planning in 3D workspaces. Their results show that RAPID performs better (in general) for non-convex models and its performance is not bad when compared to V-Clip, which performs better for models of moderate size and with few non-convex characteristics.

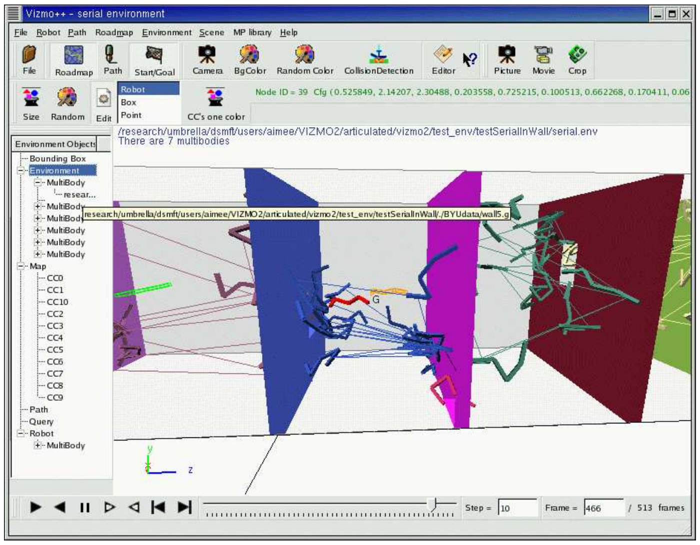
CHAPTER III

REQUIREMENTS

Our objective was to develop a tool to support the visualization and manipulation of the elements of the environment (e.g., rigid and articulated robots, obstacles, roadmap, path, etc.) that can be generally useful for motion planning researchers.

Vizmo++, a new version of Vizmo [23], is a 3D visualization/authoring tool conceived to display and manipulate the elements in the moving object's workspace and the information generated for any motion planner (roadmap and path), but specialized for PRMs. Vizmo supported only rigid bodies. The GUI was composed of a menu that allowed users to visualize the roadmap (whose nodes were rendered as cubes) and the path as a series of robot configurations. It allowed the visualization of the models (robot and obstacles) in either solid or wire mode. The interaction with the environment was limited to zooming in and out and rotation of the scene. In addition, the architecture of the application did not easily support the addition of new functionality. Hence, it was necessary to change the design to support the new set of required characteristics and functionality we describe below (Section B).

This new version of Vizmo provides an improved interface that eases interaction with the objects in the environment. It gives users the possibility to modify, add or delete static objects, and to change the position and orientation of the moving object. Vizmo++ also supports visualization of collision between objects and offers an interface to our motion planning library (Figure 7 shows Vizmo++'s GUI). The new architecture of Vizmo++ is based on the object oriented programming paradigm to allow better extendibility of the code.



```
Node ID = 39  Cfg (0.525849, 2.14207, 2.30488, 0.203558, 0.725215, 0.100513, 0.662268, 0.170411, 0.06
/research/umbrella/dsmft/users/aimee/VIZMO2/articulated/vizmo2/test_env/testSerialInWall/serial.env
There are 7 multibodies
research/umbrella/dsmft/users/aimee/VIZMO2/articulated/vizmo2/test_env/testSerialInWall/./BYUdata/wall5.g
```

Fig. 7. Vizmo++ GUI

A. Overview of Design Process

To achieve our goal and develop a tool that could be easily maintained, we applied Software Engineering principles, which are used as a guide to help in the process of software development. The Spiral model [24] has been widely used to develop interactive systems. This model is an iterative process that starts with defining objectives and an analysis of requirements so the system can be divided into different components. The next stage is the design of those components, followed by the implementation and evaluation of each component, and finally they are put together and the system as a whole is tested. Following this design and development process, once one of the components has been finished, this same approach is used on the next component. This process continues until all components have been developed. Moreover, it is well known that software development is not a linear process, so at each step of this process, we can always stop and go back to the previous stage.

After defining the requirements for Vizmo++, we decided to separate the development of the GUI and the development of the functions available to the users. Our first step was the selection of the GUI library we would use. Then, we worked on the design of the user interface (UI): the distribution of the elements such as menus, buttons, and the main window. From this point we started the design of the main classes from which we would build up the next version of Vizmo++ with a new and more friendly user interface. That process implied the redesign of the original classes. Once the new UI was implemented, we worked on its integration into the redesigned classes so we could have the "old" functionality working with our new GUI. Next, we started to work in more detail on the analysis and design of each of the new features Vizmo++ would support. The order of implementation of these features depended on the required technology and importance of those features for the current users.

In a second stage of development, we extended Vizmo++ to support articulated objects, which provided additional functionality to our tool. To achieve this extension, we needed to go back to the definition and analysis of requirements stage of the Spiral model, then redesign our classes and finally implement the new functionality. At this point the collision detection module was included in the system.

B. Definition and Analysis of Requirements and Functionality

In motion planning research, in addition to manipulating the workspace and the robot, we also need to visualize and manipulate the information generated by the planners, e.g., the path produced when answering a query or a roadmap produced during preprocessing. A requirement for Vizmo++ is that it must provide users with a nice way to interact with, visualize, and edit such information. It must also support a variety of movable objects such as rigid and articulated objects, and closed chain systems. Recall, as shown in Table I, no other available tool supports both visualization and editing of workspace environments.

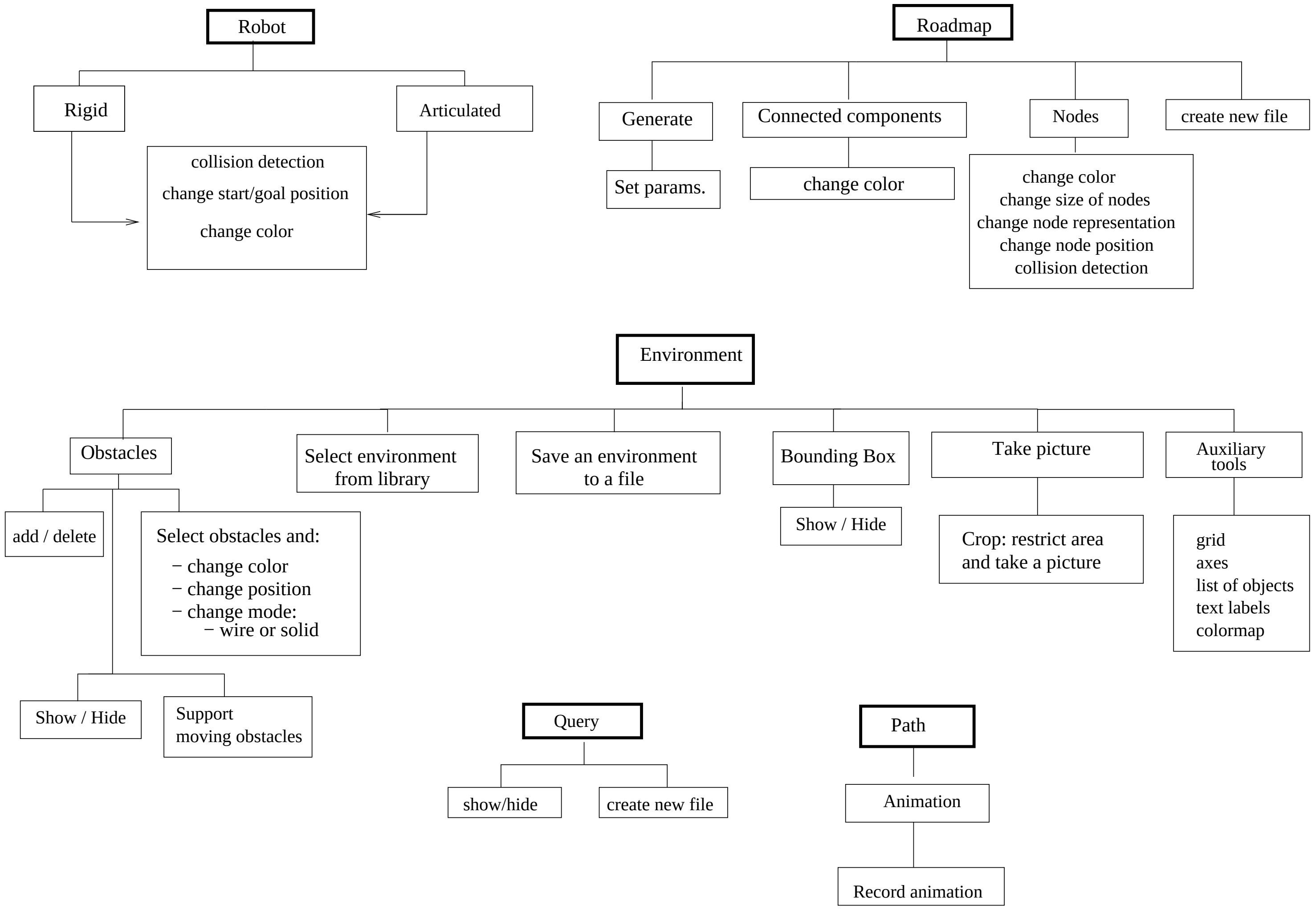


Fig. 8. Function hierarchy diagrams

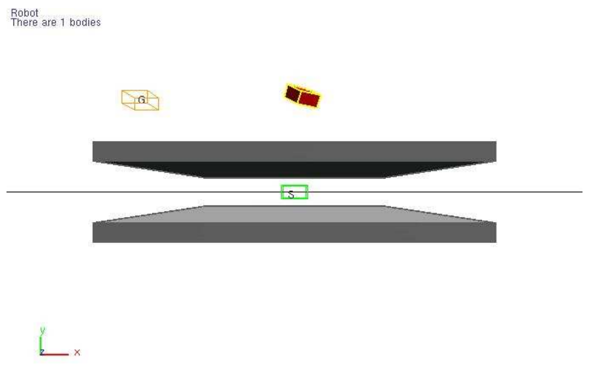


Fig. 9. Query shown for the *narrow* environment

We initiated the process of defining Vizmo++'s functionality by first identifying the objects we wanted to visualize and the operations we wanted to perform for them. This continued the definition and analysis of requirements, where we identified all the operations to be supported for each object. We did this analysis and summarized it into a *function hierarchy diagram* [45] which is shown in Figure 8. These diagrams are a hierarchical representation in which the objects to be visualized (e.g., robot, path, environment, roadmap, and query) are shown as parents and the operations the user could perform with those objects are shown as children. We describe the purpose of each functional requirement next.

Robot functions: There are a number of functions we need to support in addition to robot visualization.

- Vizmo++ must enable users to select the robot and move it, rotate it, or change its color. Enabling users to move the robot will allow them to set new start and goal configurations.
- Collision detection checking is necessary to alert the users if they have selected or created an invalid configuration.
- If the problem has more than one robot, then Vizmo++ must also support the visualization of multiple robots.

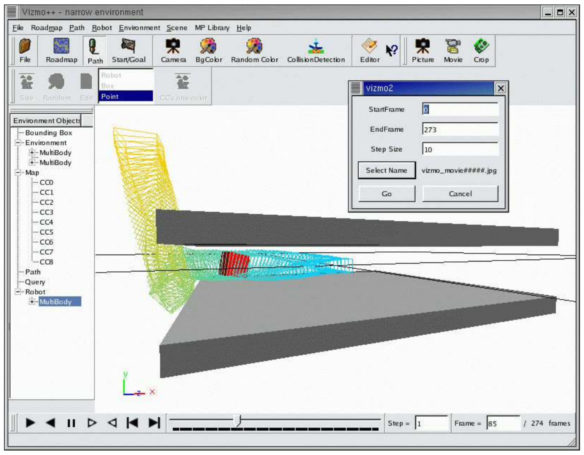
- Robot construction. Vizmo++ must enable users to build their own articulated objects.
- Articulated robot control. Users can modify any robot configuration by selecting any of its links and changing the value of the corresponding DOF.

Query functions: The initial and final configurations of the robot can be shown to give users visual information about where the robot will start moving and where it should stop. Figure 9 shows an example for the *narrow* environment which consists of two parallel plates. The robot's initial position is between these plates and the goal is to move the robot to a position away from the plates.

- It can happen that more than one query is given. Support for handling multiple queries should be provided.
- A query can be described as a sequence of configurations to be visited in order.
- Different colors and text labels are used to ease the identification of the start and goal configurations.
- New start and goal positions can be set by moving and rotating the robot. They can be stored in a new query file.
- A query file can be edited using the text editor embedded in our interface.
- Users can execute queries using the GUI by clicking a button.

Path functions: The visualization of path configurations helps users to see and analyze the path. Normally, it is rendered as a sequence of robot configurations in wire mode.

- The path can be animated.
- The path can be saved as a movie. Figure 10 shows the interface for movie production.


Vizmo++ - narrow environment
File Roadmap Path Robot Environment Scene MP Library Help
File Roadmap Path Start/Goal Camera BgColor Random Color CollisionDetection Editor Picture Movie Crop
Size Random Edit Robot Box Point CC's one color
Environment Objects
Bounding Box
Environment
MultiBody
MultiBody
Map
CC0
CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
Path
Query
Robot
MultiBody
vizmo2
Select Name
Go
Cancel

StartFrame 0
EndFrame 273
Step Size 10
vizmo_movie#####.jpg
Step = 1
Frame = 85 / 274 frames

Fig. 10. Path and movie recording interface for the *narrow* environment

- While the robot is following the path, users may edit the environment, e.g., moving an obstacle.
- Collision detection could be activated to alert the user changing the color of the robot if it collides with an obstacle while following the path.
- The path can be altered, e.g., selecting a configuration and changing it.

Roadmap functions: The visualization of the roadmap must help users to get as much information as needed to analyze it. In the following, we present the requirements for a roadmap.

- Connected components. Since a roadmap may contain one or more connected components, they will be presented in different colors to help users differentiate them. A connected component's color can be set (or changed) randomly or all connected components can have the same color. Users can also set the color of a particular connected component.
- Roadmap nodes. Users need to be able to decide how they want to view roadmap nodes. They can also change their position, add new nodes to the roadmap or delete nodes from the roadmap.

  - Nodes can be rendered as points, cubes, or actual robot configurations (poses) and the size of the nodes can be modified.
  - A node can be moved by selecting it and moving it with the mouse or by directly editing the values for any of its degrees of freedom. Figure 11 (*narrow* environment) shows the roadmap and how a node is selected to be moved. If desired, collision detection can be turned on to guarantee that, when a node is moved, the new configuration is valid.
  - A node may be added to the roadmap (valid or non valid). Its position can be manipulated as when moving a node. If we enable users to add nodes to the roadmap, they might be able to add nodes in difficult regions of the C-space, (e.g., narrow passages) that might help the planner to find a solution. This function will need to test if the new node's configuration is valid, e.g., if it is inside the bounding box or it is collision free.

- Edges. In addition to enabling users to add nodes, we want to enable them to add edges between nodes. To help users to add a valid edge or to validate existing edges, collision detection will need to be available, e.g., by calling a local planner.
- Creation of new roadmap files. If the roadmap has been modified, users need to have the option of saving that roadmap to a file where the new graph information will be stored. Users may want to select and save one or more connected components of a particular roadmap into a different map file.
- Roadmap generation. Users can do this in two ways: manually or automatically.
  - Manually, by editing an existing roadmap and saving it to a file, or by adding nodes to an empty roadmap.

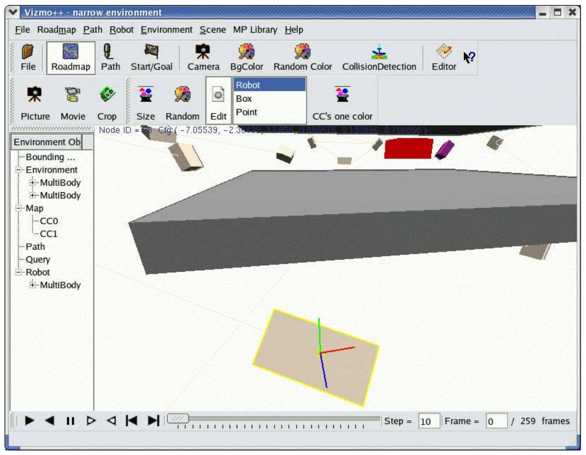


Fig. 11. A roadmap node is selected and the *translation* tool is shown.

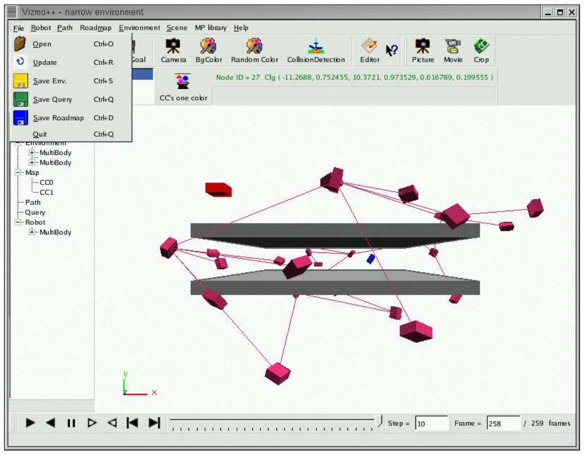
  – Automatically, by using the default planner (which can be set by the user) that is invoked when the user requests by clicking a button.

- Paths: Visualization of nodes and edges that are part of the path. Even though users can visualize the path and the roadmap at the same time, it is hard for them to distinguish which nodes and edges were used in the path. Vizmo++ can offer the user the option of visualizing only particular regions or paths of the roadmap. It can also show, e.g., in different color, if any portion of the path is non valid.
- Edges can be selected and the robot is animated following the configurations along the edge.

Environment functions:

- Users can select the environment they want to work with by selecting the input files through a graphical interface that allows them to browse their directories.
- Auxiliary tools must be provided so users can have more information about the environment:

  – Visible axes can be shown to locate objects in 3D space.



Node ID = 27 Cfg (-11.2688, 0.752455, 10.3721, 0.973529, 0.616789, 0.199555)

Fig. 12. *narrow* environment: saving options

- Text labels can be used to annotate useful information such as the number of nodes in the roadmap and their configurations, the location of the input files, and how many links (components) the movable object has.
- A color map to select the colors of objects.
- A tree-like list of the objects the user is visualizing.
- An *undo* function so that the user can undo changes.

• The bounding box. If a bounding box was given as part of the input, this bounding box can be shown. If the bounding box was not given, Vizmo++ can compute one. Users can modify or hide the bounding box.

• Users can save images of the entire environment or of a predetermined region.

• Functions for the obstacles are needed that are analogous to those provided for the robot.

- An obstacle can be selected and its color, position or orientation can be modified. Obstacles can also be scaled.
- An obstacle can be visualized in wire and solid mode.
- Users can add or delete obstacles from the scene. An obstacle will be added by selecting a geometry file from a library or directory, and it will

  be deleted after the user selects it and selects the appropriate option from a context menu.
  - An obstacle can be static or not. Vizmo++ can support static and moving obstacles.

- Users can rotate and translate the camera along the X, Y, and Z axes, or zoom in or out.
- Multiple camera views can be provided to the user so they can switch the view of the environment, e.g, from a camera mounted on the robot.
- After a user has edited an environment, they can save it into an environment file. An environment file stores information about the robot and obstacles such as initial configurations, geometry files, and relationships among objects. See Figure 12.

C. Modular Organization of Functionality

Once the requirements and functionality were determined, we subdivided our application into the following modules:

- *reader and writer* modules to provide the functions for reading and writing data from and to the environment, query, roadmap, and path files.
- An *editor* module to provide functions for allowing users to modify the environment.
- A *collision detection* module to provide visual feedback when an object is in collision with other objects.
- An *external planner* module to interface Vizmo++ with our motion planning library.

The *reader* module reads in the BYU (Brigham Young University) format [46] for the geometric description of the objects. We selected the BYU format because of its simplicity and because every other format of which we are aware can be translated into the BYU format. Vizmo++'s structure eases the implementation of any other format. The format of the query, roadmap, environment, and path files is presented in Chapter V, Section 1. The *writer* module provides the functions for writing processed data or images into files. From the function analysis we know we want to write new environment, roadmap and query files.

The *editor* module allows users to build their own environments and modify them. The *collision detection* module gives visual feedback to the user whenever an object is in collision with any other object, e.g., by changing the moving object's color when it is in collision.

The *external planner* module interfaces our application with the planners available in our motion planning library. Users can run new queries through a convenient graphic interface where planners and parameters can be selected and then saved into a script that can be executed or edited using the basic text editor embedded in Vizmo++.

CHAPTER IV

DESIGN

Interactive systems need to be evaluated at each stage of development. We followed the Iterative Design approach [47] which states that once a component has been partially or completely implemented, we can receive feedback from users and based on that we can detect problems in the design of the interface or problems in the functionality of the system, and thus we can re-evaluate alternatives. Our design has to take into account the compatibility and interaction with the motion planning library developed in our research group.

We split our design into the design of the interface and the design of the classes. For the design of the interface, we applied User Centered Design [48]. Working closely with the final users allowed us to know their specific needs. The object oriented programming paradigm was used to design and implement Vizmo++. This will ease maintainability. Also, object-oriented programming offers a natural way of defining functions and data in terms of a class hierarchy.

A. Design of the User Interface

The definition of requirements leads to the specifications of how the different parts of an application will be used. Users were a central part in the design of Vizmo++'s GUI because we wanted to offer them a self-explanatory interface that allows them to become familiar with it in a short period of time. Getting feedback on the usability of the application helped us in the definition of interface actions.

Another factor we considered in the design was the distribution of the menus, toolbars, and buttons. We had several presentations and releases of Vizmo++ so users could use it and in this way we got feedback from them related to the way the options were distributed, the ease of use, how intuitive the interface was, and what other options users would like Vizmo++ to provide. The users helped us to identify the tasks most commonly performed: hiding and showing elements (e.g., the roadmap, the path, and the start and goal configuration), randomly coloring the objects, opening environment files, and editing the roadmap. All these tasks were made available through buttons and toolbars, so the users did not have to open menus each time they needed to show or hide elements or change properties of the roadmap.

We made other functions available through buttons such as collision detection checking, background color, and resetting the position of the camera. We also provided a toolbar to make movies and to take snapshots. In general, we tried to reduce the number of clicks the user had to make in order to get a response to some action.

Each GUI library has its own philosophy and this needs to be considered for the design of the GUI. What we looked for was an easy to use and learn library, portable, with good documentation and with a wide set of widgets that can be used in our interface. We decided to use Qt [49], which is a C++ multi-platform toolkit GUI that facilitates the creation of applications. This library includes a wide set of controls (widgets) that provides standard GUI functionality. The communication among objects is achieved through what is called Signals and Slots (used for communication between objects), that replace the call back technique. Qt also implements the Events model to handle mouse inputs and key presses.

B. Class Design

Object oriented design is a well known method for modeling real or abstract objects by expressing relationships between those objects. Each object is described as set of

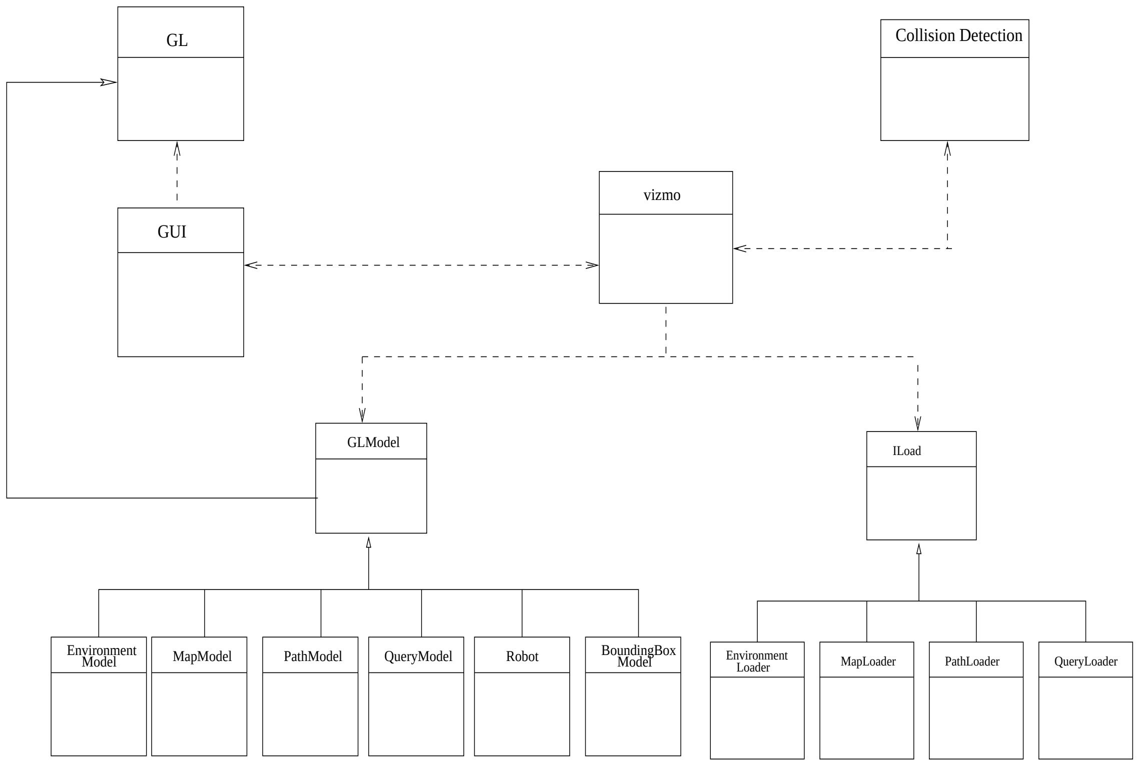


Fig. 13. Class hierarchy (solid lines represent inheritance and dashed lines denote a “use” relationship)

attributes and operations (methods).

Developing the function hierarchy diagrams discussed in Chapter III helped us to identify and define the main classes and methods we would need to implement in Vizmo++. We use Unified Modeling Language (UML) notation [50] which is a standard for modeling software used to produce class diagrams. Class diagrams help in the understanding of the software architecture by showing relationships among classes.

Figure 13 shows a high level class diagram. In this diagram we represent inheritance with solid lines and hollow arrows. The dashed lines denote a "use" relationship among classes. For example, all the GUI classes use methods from the *GL* class which implements OpenGL functions. We omitted the attributes and methods.

The class *vizmo* is the main class of Vizmo++. We can classify our design in the following main modules:

- *GUI* implements the graphical user interface. This allows us to isolate all the functionality at the interface level. This module communicates with the methods that implement the functions related to the geometric model of the objects.

- *ILoad* is an interface that contains methods to read and load the input files: environment, the path, the roadmap, and query.

- *GL* implements OpenGL related functions such as selection of objects, translations, rotations, camera position.

- *GLModel* contains the functions to build and render the model of the robot, the obstacles, the roadmap, the path and the bounding box.

- *Collision Detection* implements the functions to perform collision detection. It uses the RAPID library [38].

CHAPTER V

IMPLEMENTATION

A. I/O Files

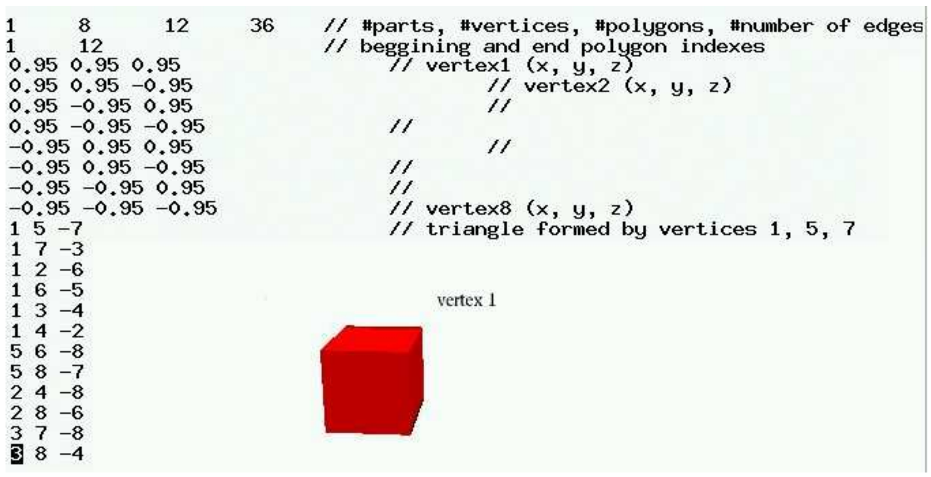
In Chapter III section B, we identified the objects we wanted to visualize. The information we need for visualization effects is stored in the following input files: environment, roadmap, path, and query. It is in these files where we obtain the location of the geometry files, the initial position and orientation of each multibody, the number of joints of the robot and their description, the number of nodes in the roadmap and node configurations, the configurations of paths found by planners, and the start and goal configurations. Vizmo++ reads and interprets all this information. The details of each file format are described in the next section.

1. File Formats

Model Files: The geometric descriptions of the objects to be visualized are given as input to Vizmo++ in the BYU format [46]. BYU describes a model in terms of its surfaces, which are composed of two-dimensional objects. BYU is an easy and widely used way of defining a model.

Our model is a polyhedron composed of a set of polygons, which are described as a set of triangles by convention. In the BYU format the header of the file has four values which are: number of parts, number of vertices, number of polygons, and number of edges. Then each value is decomposed into a list:

- The list of parts is defined by two index values, the beginning and the end polygon numbers that form the model.



```
1     8      12     36    // #parts, #vertices, #polygons, #number of edges
1     12                  // beggining and end polygon indexes
0.95 0.95 0.95                 // vertex1 (x, y, z)
0.95 0.95 -0.95                        // vertex2 (x, y, z)
0.95 -0.95 0.95                        //
0.95 -0.95 -0.95               //
-0.95 0.95 0.95                        //
-0.95 0.95 -0.95               //
-0.95 -0.95 0.95               //
-0.95 -0.95 -0.95              // vertex8 (x, y, z)
1 5 -7                         // triangle formed by vertices 1, 5, 7
1 7 -3
1 2 -6
1 6 -5
1 3 -4
1 4 -2
5 6 -8
5 8 -7
2 4 -8
2 8 -6
3 7 -8
3 8 -4
```

Fig. 14. Example of a BYU file describing a cube

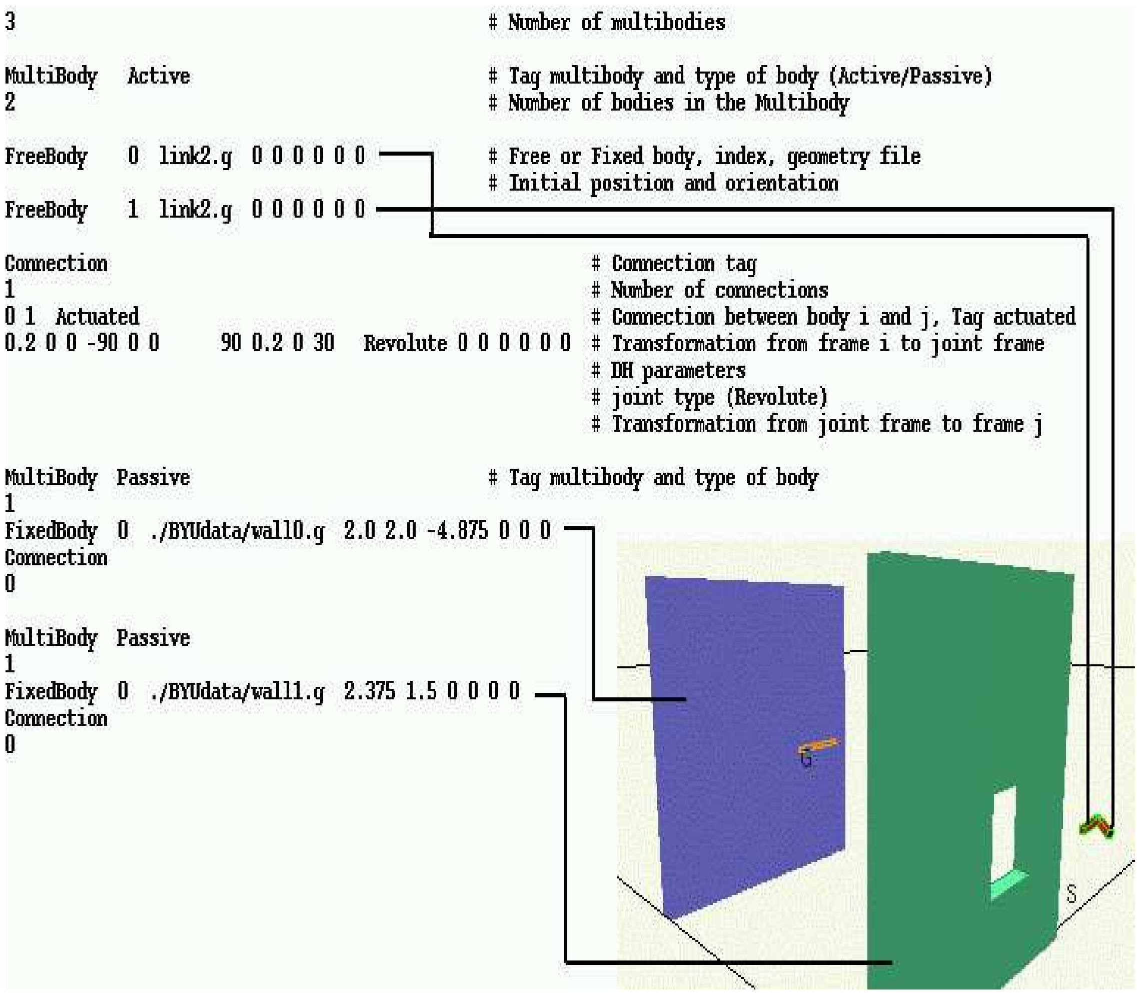
- The list of vertices provides the x, y, and z coordinates for each vertex.

- The list of edges describes the connectivity among the vertices. Each row contains vertex IDs. The edges are thus combined to form polygons.

All coordinates are with respect to a local coordinate frame. An example for a cube is shown in Figure 14.

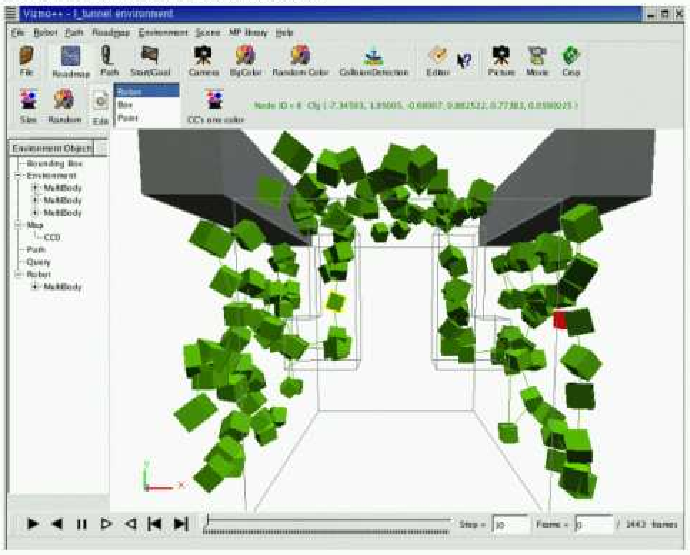
Environment File: The environment file defines the entire environment, that is, how many objects and of what kind they are (robots or obstacles). For each object, the location of its geometry file is given as well as its position and orientation. If the object is articulated, connection information is also given as Denavit-Hartenberg parameters (see II-4) and the constant transformation from one frame to the other. See Figure 15 for an example file and its visualization.

Map File: The map file is generated by a planner for a specific environment. This file has a preamble that provides information about the command line executed to compute the roadmap, the name of the environment file, the parameters sent to the local planner, the collision detection libraries and the distance metrics used. The actual graph information follows the preamble. As part of the graph information, the number of nodes and edges are given in the graph definition. Each row has the node



```
3                                           # Number of multibodies

MultiBody   Active                          # Tag multibody and type of body (Active/Passive)
2                                           # Number of bodies in the Multibody

FreeBody    0  link2.g  0 0 0 0 0 0         # Free or Fixed body, index, geometry file
                                            # Initial position and orientation
FreeBody    1  link2.g  0 0 0 0 0 0

Connection                                           # Connection tag
1                                                    # Number of connections
0 1  Actuated                                        # Connection between body i and j, Tag actuated
0.2 0 0 -90 0 0     90 0.2 0 30   Revolute 0 0 0 0 0 0  # Transformation from frame i to joint frame
                                                     # DH parameters
                                                     # joint type (Revolute)
                                                     # Transformation from joint frame to frame j

MultiBody  Passive                          # Tag multibody and type of body
1
FixedBody  0  ./BYUdata/wall0.g  2.0 2.0 -4.875 0 0 0
Connection
0

MultiBody  Passive
1
FixedBody  0  ./BYUdata/wall1.g  2.375 1.5 0 0 0 0
Connection
0
```

Fig. 15. Environment file description and visualization



```
Roadmap Version Number 061300
#####PREAMBLESTART#####
../obprm -cd PQP -f l_tunnel -bbox [-19,19,-16,17,-14,14] -bbox_scale 1
-gNodes BasicMAPRM nodes 155 -cNodes closest components
#####PREAMBLESTOP#####
#####ENVFILESTART#####
l_tunnel.env
#####ENVFILESTOP#####
#####LPSTART#####
3
straightline
rotate_at_s 0.5
a_star_distance 6 9
#####LPSTOP#####
#####CDSTART#####
3
vclip
RAPID
PQP
#####CDSTOP#####
#####DMSTART#####
5
scaledEuclidean 0.9
euclidean
minkowski 3 3 0.3333
manhattan
com
#####DMSTOP#####
#####GRAPHSTART#####
146 145 146
0 -15.8859 0.00460175 0.749008 0.454433 0.233178 0.831292 -1 -1 -1  2 139 1 50  87 1 26
1 15.6058 2.74597 1.57064 0.0508319 0.767051 0.0189148 -1 -1 -1  2 129 1 52  117 1 20
2 15.7869 -9.82882 10.4726 0.570614 0.0767746 0.815274 -1 -1 -1  2 118 1 56  81 1 50
3 14.7802 -3.11739 0.707067 0.427835 0.542743 0.667979 -1 -1 -1  2 95 1 17  19 1 70
4 -11.1658 11.4872 8.5885 0.769221 0.0525795 0.96673 -1 -1 -1  1 90 1 45
5 -15.7733 6.58582 -10.3904 0.0040902 0.000502785 0.192952 -1 -1 -1  1 110 1 50
6 15.2885 7.21374 -3.72586 0.944288 0.884822 0.445171 -1 -1 -1  1 129 1 31
7 -15.939 -3.43206 -5.60616 0.0210121 0.238803 0.0371094 -1 -1 -1  1 125 1 47
8 -7.34593 1.95605 -0.68907 0.882522 0.77383 0.0590025 -1 -1 -1  2 70 1 42  53 1 45
9 15.9156 6.54351 5.22283 0.358774 0.610365 0.415004 -1 -1 -1  1 58 1 7
10 7.09463 13.4708 -10.3699 0.059656 0.0130721 0.391159 -1 -1 -1  1 121 1 13
                               ...
                               ...
                               ...
```

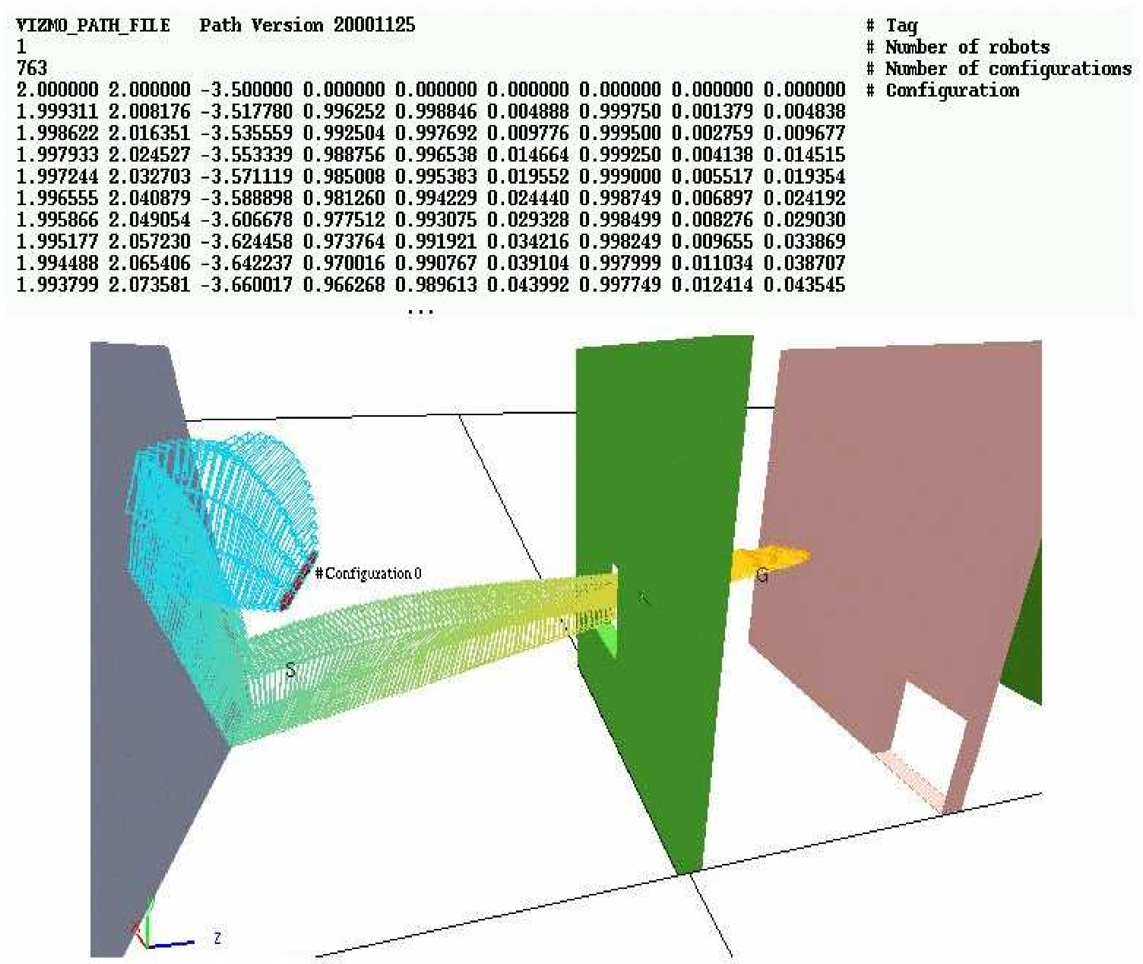
Fig. 16. Map file example and visualization

ID, its configuration, the number of edges adjacent to this node, and the list of edges, provided as a list of node IDs. Each node ID is followed by the planner ID identifying the planner used to generate that edge and an optimal edge weight. See Figure 16.

Path File: The path file stores the list of configurations that takes the movable object from the start configuration to its goal configuration. These configurations are computed by a planner. Like the roadmap file, the path file has a preamble providing set up information. The first line identifies this as a path file and gives its version. The second line provides the number of robots and the third line provides the number of configurations the file stores. Next, there is a line for each configuration in the path.

In each line, the first six numbers indicate position and orientation. If the object is articulated, the following numbers indicate the angle between bodies. Figure 17 shows an example of a path file for a 6-DOF robot.

Query File: The query file is an input file for the *query* program, it stores two lines that describe the start and goal configurations of the movable object. We use this information to place the robot at the initial configuration when an environment is loaded.



```
VIZMO_PATH_FILE   Path Version 20001125                                                  # Tag
1                                                                                        # Number of robots
763                                                                                      # Number of configurations
2.000000 2.000000 -3.500000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000       # Configuration
1.999311 2.008176 -3.517780 0.996252 0.998846 0.004888 0.999750 0.001379 0.004838
1.998622 2.016351 -3.535559 0.992504 0.997692 0.009776 0.999500 0.002759 0.009677
1.997933 2.024527 -3.553339 0.988756 0.996538 0.014664 0.999250 0.004138 0.014515
1.997244 2.032703 -3.571119 0.985008 0.995383 0.019552 0.999000 0.005517 0.019354
1.996555 2.040879 -3.588898 0.981260 0.994229 0.024440 0.998749 0.006897 0.024192
1.995866 2.049054 -3.606678 0.977512 0.993075 0.029328 0.998499 0.008276 0.029030
1.995177 2.057230 -3.624458 0.973764 0.991921 0.034216 0.998249 0.009655 0.033869
1.994488 2.065406 -3.642237 0.970016 0.990767 0.039104 0.997999 0.011034 0.038707
1.993799 2.073581 -3.660017 0.966268 0.989613 0.043992 0.997749 0.012414 0.043545
                                          ...
```

Fig. 17. Path file description and visualization

2. Creation of New Files

Environment File: From the environment file we retrieve the values of the variables the EnvironmentLoader class read at the beginning of the user's session. From there we can get the multibody information. Since the rotation of a body is described as Euler angles in the environment input file and we use quaternions to compute and represent rotation (this is described in more detail in section C.1), we need to convert quaternions back to Euler angles before writing the environment file.

Map file: In the map file, as stated in the previous section, each node stores its current position and orientation, besides the information about its connectivity (what edges and nodes this node is connected to) and whenever a configuration is moved, we update the configuration's values. When a user wants to save a new roadmap we call methods from the *Graph* class to write the new graph.

The *Graph* class implements methods to create and modify a graph, e.g., add nodes and edges, delete nodes and edges, extract information about nodes and edges such as edge weight, vertex information, etc.

Query File: To store new start and goal configurations in a file, we added two methods to the Robot class: the first method aids in the storing of the current robot configuration as a start or goal position and the second method returns the two new configurations to be saved into a file.

B. Geometric Model and Rendering

All the objects in the workspace (static or not) are described using their boundaries. With this description, we can draw them with polygons. There are several free software options to render 3D images: Persistence of Vision Ray-Tracer (POV-Ray) [51], Renderman [52], OpenGL [31], and Visualization ToolKit (VTK) [40], among others.

All these tools incorporate a broad set of rendering, texture mapping and other visualization functions. We decided to use OpenGL because of its portability and for its fast response time, necessary since we are developing an interactive tool. We don't want to offer a very high quality image at the cost of poorer response time. OpenGL offers a good image quality that suffices for our needs.

All OpenGL surfaces are generated as collections of triangular surface patches. Each polygon description is given as input through the geometry files described in the Model Files section in Chapter V.

1. Object Selection

Vizmo++ allows users to select objects from the scene and perform operations over them such as changing color, changing position, changing orientation, deleting, or changing rendering status (solid/wire). After editing the environment, users can save their changes to query, map and environment files.

To select an object from the scene, we use the OpenGL API that provides a mechanism to select objects. In general, we need to detect which objects are close or at the location of the mouse coordinates. The following steps need to be performed to execute this action:

1. Get the window coordinates of the mouse.
2. Enter selection mode.
3. Redefine view volume.
4. Render the scene.
5. Quit selection mode and get the selection buffer.

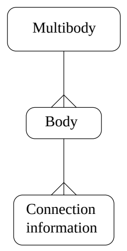


Fig. 18. Multibody description

C. Class Implementation

As described in Chapter III section 5, each object in the workspace is a *multibody.* In Vizmo++ we defined the class *Multibody* as one that will store information about how many bodies compose the multibody and how many connections it has. Then the class *Body* keeps information about body position, orientation, the name of the geometry file, its index, a list of connections and its current transformation. The class *Connection* stores the DH parameters, and the constant transformation (position and orientation) of the local frame of the body and the joint frame. Figure 18 depicts this relationship. At the finest level, each body is conceived of as a *Polyhedron.* It is in this class where the body is built and its RAPID bounding volume hierarchy is constructed. This class is also in charge of drawing the solid and wire models.

The *Robot* class is in charge of calling all the methods needed to build the moving object after all the information has been stored in the Multibody class. The *Robot* class is also in charge of computing the necessary transformations so each body can be rendered in the correct configuration with respect to the links that the current body is attached to. Articulated objects are modeled as tree-like structures, following the information given in the environment file.

The *Environment-related* classes are in charge of reading all the information stored in the environment file, feeding that information to the *Multibody* class, and then asking for the creation of the models of each of the elements of the environment.

Since a roadmap can be composed of one or more connected components, the *Map-related* classes will be in charge of reading the map input file and feeding information to the *Connected Component* class, which will dictate the way the nodes will be rendered (points, boxes, robot configurations). We also use the class *Graph*, which is a data structure used to extract and hold information related to the nodes and the edges of the roadmap. This is the same class used by the motion planning algorithms. The *Configuration* class inherits from the *Graph* class. This class stores information related to each node's configuration and the amount by which the node has been moved from its original position. It implements its own drawing methods to render a node.

As stated before, the path is a sequence of configurations listed in the path input file. All those configurations are read and stored and then used to create the wire model of each path configuration in the *PathModel* class. We obtain the model of the robot from the *Robot* class. The same logic is applied for rendering the query though the *QueryModel* class will be in charge of it.

In our analysis, we decided to split the implementation of the GUI and the implementation of the models, so we can make Vizmo++ modular. No change to the GUI will affect the way the models work but there is a close relationship between the operations we want to perform with the objects in the environment that should map to options offered in the GUI. The class *vizmo* works as the interface between the *GUI* and model classes. Some of the classes that have to do with the GUI need to be related to the class that manages all the interaction that the user can have with the environment such as rotation and zoom in/out. We implement this functionality in the *GL* class.

The *ILoad* class is an interface class which is inherited by all the "loader" classes shown in Figure 13. This enables our application to expand the number of files that could be read. If a new input file is necessary at some point, the developer just needs to create a new “load” class and in this way we also keep the “reading” standard of Vizmo++.

The classes that implement OpenGL related functions, e.g. selection of objects, translations, rotations, are gathered in the *GL* module for simplification purposes. The *GLModel* class inherits from *GL* and all the “model” classes inherit from *GLModel* to build and render the objects. As in *ILoad*, *GLModel* allows an easy extension of the kind of objects Vizmo++ can visualize.

1. Implementing Articulated Objects

As stated in Chapter III, the first stage of development included just rigid bodies. We needed to extend Vizmo++ to support articulated objects. First, we had to go over the process of requirements analysis to detect the classes that would need to be updated:

- *Environment class*: in the first stage of development, we used a constant to set the number of DOF to six, so there was no way of computing the actual DOF of an articulated object. We had to add code to compute the DOF dynamically and code to read all the information related to the connection information that is given for each joint in the environment file. This led us to add more variables to the MultiBody and Body classes to store all the data.

- *MultiBody and Body classes*: this is one of the classes where more changes were needed. As stated in the previous paragraph, new variables were needed to store information about the connections between links. We had to keep a tree structure to know how each link was related to the rest. We also had to keep references to the current transformation of each body.

- *Robot class*: the method in charge of configuring the robot was completely changed. We had to compute the necessary transformations to obtain the actual position and orientation of a link given the position and orientation of the previous one, as stated in Chapter IV section B. In order to perform those transformations, we had to implement the following classes:

- The *Transformation* class that is in charge of generating the transformation variables (a 3D vector for position and a matrix for orientation). It also implements the multiplication operator as in equation 2.2 and a method to transform DH parameters into a matrix using equation 2.1.

- An *Orientation* class that implements the methods to obtain the rotation matrix given the Euler angles according to equation 2.71 in [42]:

$$\mathbf{R_{XYZ}}(\gamma,\beta,\alpha) = \begin{pmatrix} c\alpha\ c\beta & c\alpha\ s\beta\ s\gamma - s\alpha\ c\gamma & c\alpha\ s\beta\ c\gamma + s\alpha\ s\gamma \\ s\alpha\ c\beta & s\alpha\ s\beta\ s\gamma + c\alpha\ c\gamma & s\alpha\ s\beta\ c\gamma - c\alpha\ s\gamma \\ -s\beta & c\beta\ s\gamma & c\beta\ c\gamma \end{pmatrix} \tag{5.1}$$

where c stands for the cosine function and s for the sine function.

The initial rotation of a body is given as Euler angles. We call these angles α, β, and γ.

Quaternions are a more compact representation of a rotation, so we decided to represent rotations as quaternions. A quaternion is a four dimensional complex number that represents a rotation w about a unit vector $[x, y, z]$, here referred to as $q = [w, x, y, z]$. Quaternions allow the implementation of smooth and continuous rotations. In our implementation, we compute the quaternion for each given Euler angle as in Equation 5.2. Then the final rotation is computed by multiplying those quaternions.

$$qx = (c_x, s_x, 0, 0) \qquad qy = (c_y, 0, s_y, 0) \qquad qz = (c_z, 0, 0, s_z) \tag{5.2}$$

where:

$$c_x = \cos\frac{\alpha}{2}, s_x = \sin\frac{\alpha}{2} \qquad c_y = \cos\frac{\beta}{2}, s_y = \sin\frac{\beta}{2} \qquad c_z = \cos\frac{\gamma}{2}, s_z = \sin\frac{\gamma}{2}$$

When needed, as in the case of computing collision detection, the resulting quaternion is converted to a rotation matrix following Equation 5.3 (equation 2.91 from [42]).

$$Rot = \begin{pmatrix} 1-2(y^2+z^2) & 2(xy-wz) & 2(xz+wy) & 0 \\ 2(xy+wz) & 1-2(x^2+z^2) & 2(yz-wx) & 0 \\ 2(xz-wy) & 2(yz+wx) & 1-2(x^2+y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.3}$$

Since we extended Vizmo++'s capabilities to save environment, map, and query files, we needed to convert quaternions back to Euler angles. Given that we already had the conversion from quaternion to matrix, we needed just to convert from matrix to Euler angles.

- *Configuration class*: we had to redesign the methods of the class to make them support more than six DOF because we were only considering six variables to store the configuration of a robot. We use a simple data structure to dynamically make it store the configuration of the robot given its number of DOF.

D. Interfacing Vizmo++ with our Motion Planning Library

We want Vizmo++ to be independent of the planners. The interface between Vizmo++ and our motion planning library was implemented through the generation of scripts. We offer the user the functionality of generating a new command line and saving it to a file to run a planner. In this way users may be able to easily call other motion planning libraries and display information generated by any planner as long as it provides the information in the formats supported by Vizmo++.

E. Status

Chapter III describes the requirements we determined were necessary for a tool such as Vizmo++ to support the visualization and manipulation of motion planning environments, problem instances, and their solutions. In this section we outline the status of the current implementation of Vizmo++.

Robot functions:

- Implemented.
  - Support for a single movable object.
  - Support for rigid and articulated movable objects.
  - Partially implements collision detection visualization. Collision checking is performed only when the robot or obstacle is being moved using the mouse.
  - Users can modify the movable object's color, position and orientation.
- Not implemented.
  - Vizmo++ does not support the visualization of multiple robots. This can be done by making the *Environment*-related classes able to read and store the information for multiple robots.
  - Vizmo++ does not implement the robot construction function. New classes would need to be implemented so users are able to draw a skeleton representation of the links of the robot and to set their DH parameters. Then,

a rigid body representation can be constructed for each link based on user preferences.

  - Users cannot change each DOF of the robot. Robot selection is already available and it would need to be updated to let users select individual links instead of selecting the entire body.

Query functions:

- Implemented.
  - Users can visualize the start and goal configurations, which are shown in different colors.
  - New queries can be created, stored in a new query file, and executed using a GUI. Users can also run new queries automatically by just clicking a button. These queries are a pair of configurations.
- Not implemented.
  - Vizmo++ supports one query at a time. The QueryModel class will need to be updated to handle more than one query.
  - Vizmo++ currently allows users to define a query consisting of two configurations, the start and the goal. It would need to be extended to support sequences consisting of more than two configurations, e.g, adding and selecting nodes that will be part of the query.
  - Vizmo++ should offer users the option of changing the color of the start and goal configurations. This can be done by adding a function to the QueryModel class to dynamically set the color of these configurations.

Path functions:

- Implemented.
  - Provides path visualization, animation, and movie recording.
- Not implemented.
  - Collision detection currently cannot be turned on to validate a path. The user should be able to activate collision detection to detect when the robot collides with obstacles while following the path. This can be done by calling the collision detection module every time the scene is drawn instead of calling it only when the robot is moved with the mouse.
  - If more than one robot is given, then the classes that are in charge of reading in the environment information and the path file and those that are in charge of building the model of the robot will need to be updated.
  - The path cannot be altered. Selection of robot configurations that are part of the path will need to be provided. The PathModel class will need to be updated to handle the selection and manipulation of a configuration.

Roadmap functions:

- Implemented.
  - Vizmo++ supports roadmap editing. Users can:
    * Select nodes.
    * Change the shape, size, and position of the nodes.
    * Add nodes and edges. Vizmo++ currently does not support collision detection for nodes and edges.
    * Change the color of the connected components randomly or by selecting a connected component and manually selecting a color for it.

  - Generate roadmaps manually and automatically as described in Chapter III Section B.
  - Save new roadmaps in files.

- Not implemented.
  - If a node is being moved the user should be able to turn on collision detection to determine if they are selecting an invalid configuration. This function could be implemented using the collision detection function already developed for the robot.
  - Add nodes. The user should be able to turn on collision detection or a bounding box test for a node to test whether the node's configuration is valid.
  - Add Edges. The user should be able to turn on collision detection to test if an edge is valid, e.g., by calling a local planner.
  - Visualization of nodes and edges that are part of the path. Even though users can visualize the path and the roadmap at the same time, it is hard for them to distinguish which nodes and edges were used to compute the path. Vizmo++ can offer the user the option of visualizing only particular paths or portions of the roadmap. This information can be given as an additional input file or the user might select particular options in a GUI.
  - If more than one robot is given the *Map*-related classes will need to be updated accordingly.
  - Even though users are able to save roadmaps, they cannot select one or more connected components to be saved in file. Currently, a connected component can be selected so Vizmo++ will need to enable users to select

more than one connected component at a time and offer them the option of saving those connected components in a different map file.

  - Edge animation. The edges can already be selected. The same approach used for animating the path can be followed to make the robot move along the edge.

Environment functions:

- Implemented.
  - Vizmo++ offers auxiliary tools to give more feedback to the user using text labels and visible axes, a tree-like list of the objects in the scene, and a color map is provided to change the color of the objects randomly or by hand.
  - If the bounding box is provided, users can show it or hide it.
  - Users can save images of the entire environment or of a predetermined region.
  - Users can add, delete, change color, position, and orientation of obstacles. They can be rendered as solid or wire models.
  - Users can zoom in or zoom out, and rotate the scene.
  - Users can save a modified environment into a new environment file.
- Not implemented.
  - An *undo* function so that can undo operations. Vizmo++ will have to keep track of the events and store the objects and the properties that were modified.

- The bounding box. If the bounding box was not given the users should be able to ask Vizmo++ to compute it, and the user should be able to modify it to focus on particular regions of the environment. The class that is in charge of rendering the bounding box should implement the functions to compute and modify the bounding box.
- Moving obstacles. The obstacles are defined as a multibody, so the same functions that were implemented for the robot can be applied to obstacles as well. Vizmo++ would need to be able to identify when it is dealing with a moving obstacle so it can enable motion functions. Then, obstacle motions could be stored as paths to load and show in Vizmo++.
- Support for multiple cameras. Currently, Vizmo++ handles only one camera. The classes in charge of the camera control have the structure necessary to add and handle more than one camera.

CHAPTER VI

APPLICATIONS: THE CAMPUS NAVIGATOR

The Campus Navigator is a web-based application that helps users navigate the large and complex Texas A&M University campus [53]. In this project, motion planning techniques are applied to find a path that users can follow to go from a given start position to their final destination. The main modules of this application are the user interface, the path generator, and the database.

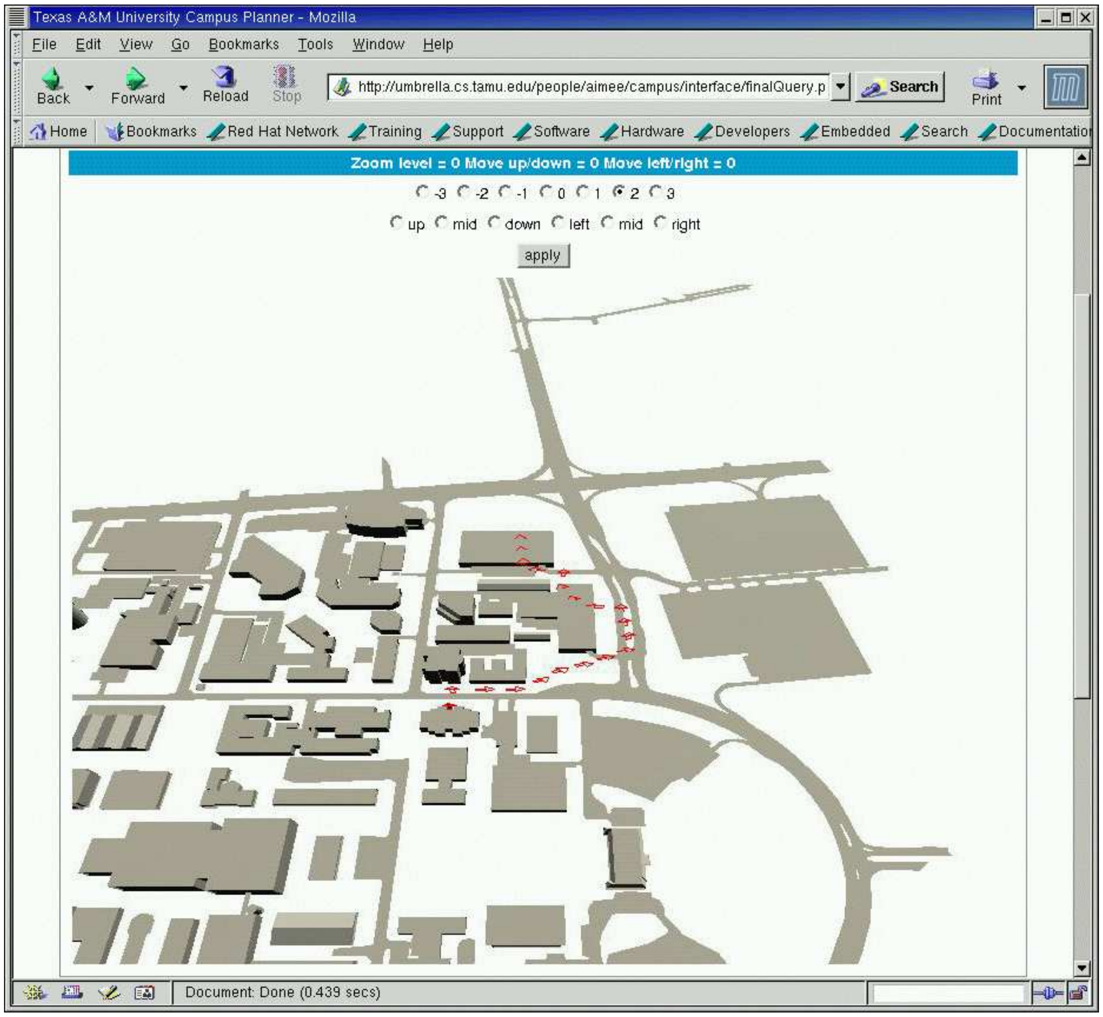
We use 2D models of the campus that are based on CAD (Computer Aided Design) models of buildings, streets, and side walks in the campus that are maintained by the TAMU Administrative GIS office (http://www-agis.tamu.edu/). Buildings, parking lots and bus stops can be used as start and goal positions.

Since the planner may consider all possible transportation methods (car, motorcycle, bicycle, walking, wheelchair, or bus), all the information available is used in finding a path. The roadmap was built by hand using a roadmap editor written in Java. The roadmap contains all the valid paths in the campus, represented by the places and the transportation methods. We need the roadmap to be easily maintained and efficiently accessed from other programs so we decided to store it in a database.

We use Vizmo++ to automatically generate and display the path found [54] and the 3D model of the campus (see Figure 19). The CAD models were converted to the BYU format so Vizmo++ could read and display them. Once the user sets the transportation mode, the initial position, and the final destination, a query is generated and sent to the path generator which will compute a path. Once the path has been generated, Vizmo++ will render it and take a snapshot of it, which is then sent back to the web interface along with a set of directions that are also automatically generated (see Figure 19).

There are things we still need to work on:

- The current version of the Campus Navigator supports one transportation means at a time. We want to expand it so combinations of transportation methods can be taken into account to generate more than one alternative for the users to get to their destinations.
- Vizmo++ could be used as a means to interactively define new start and goal positions by allowing the user to click on the map, so the user can dynamically set new queries.
- Vizmo++ could be used to automatically generate movies so users can see an animation of the defined path.
- The path needs to be optimized according to user preferences, e.g., showing the shortest path.
- More accurate directions need to be provided to the users.


Texas A&M University Campus Planner - Mozilla
File Edit View Go Bookmarks Tools Window Help
Back Forward Reload Stop
http://umbrella.cs.tamu.edu/people/aimee/campus/interface/finalQuery.p
Search
Print
Home Bookmarks Red Hat Network Training Support Software Hardware Developers Embedded Search Documentatio
up mid down left mid right
apply

Zoom level = 0 Move up/down = 0 Move left/right = 0

-3 -2 -1 0 1 [✓ 2] 3

Document: Done (0.439 secs)

Fig. 19. Path image generated by Vizmo++

CHAPTER VII

CONCLUSION

We presented Vizmo++, a tool for visualizing and editing motion planning environments, problem instances, and their solutions. Vizmo++ was developed following Software Engineering techniques and Computer-Human Interaction guidelines to implement a well-designed object-oriented application that is easy to maintain and extend. The use of Qt for the GUI, RAPID for the collision detection, OpenGL for the rendering, and C++ as our programming language, made it possible to port Vizmo++ to Mac OS and Windows with some minor changes. The original development platform was Linux.

Vizmo ++ offers a self-explanatory graphical user interface that provides functionality that no other known 3D visualization tool provides. In particular, it:

- allows users to interact with the environment, modify it and create new environments by:
  - adding and deleting obstacles, and
  - modifying object properties (color, position, orientation)
- supports collision detection visualization,
- allows new queries to be created using a GUI,
- provides a convenient interface to select planners and set parameters for new queries,
- supports two rendering modes, solid and wire,
- provides path visualization, animation, and movie recording,
- supports roadmap editing, and

- provides auxiliary tools to give feedback to the user.

We are convinced Vizmo++ can be generally useful for the motion planning community and we hope to release our tool to other research groups to facilitate comparison of different planners in the near future.

REFERENCES

[1] R. Manseur. Denavit-Hartenberg Parameters. Electrical and Computer Engineering Dept., University of West Florida. Accessed October, 2005. [Online]. Available: http://uwf.edu/ria/robotics/robotdraw/DH_parm.htm

[2] G. Song and N. M. Amato, "A motion planning approach to folding: From paper craft to protein structure prediction," Department of Computer Science, Texas A&M University, Tech. Rep. TR00-001, January 2000.

[3] J.-C. Latombe, *Robot Motion Planning.* Boston, MA: Kluwer Academic Publishers, 1991.

[4] J. Barraquand and J. C. Latombe, "Robot motion planning: A distributed representation approach," *Int. J. Robot. Res.*, vol. 10, no. 6, pp. 628–649, 1991.

[5] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 1999, pp. 473–479.

[6] P. Bessiere, J. M. Ahuactzin, E. G. Talbi, and E. Mazer, "The Ariadne's clew algorithm: Global planning with local methods," in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, vol. 2, 1993, pp. 1373–1380.

[7] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," *Int. J. Comput. Geom. & Appl.*, pp. 2719–2726, 1997.

[8] S. M. LaValle and J. J. Kuffner, "Rapidly-Exploring Random Trees: Progress and Prospects," in *Proc. Int. Workshop on Algorithmic Foundations of Robotics (WAFR)*, 2000, pp. SA45–SA59.

[9] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE*

Trans. Robot. Automat., vol. 12, no. 4, pp. 566–580, August 1996.

[10] G. Song, S. L. Miller, and N. M. Amato, "Customizing PRM roadmaps at query time," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2001, pp. 1500–1505.

[11] R. Bohlin and L. E. Kavraki, "Path planning using Lazy PRM," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2000, pp. 521–528.

[12] C. L. Nielsen and L. E. Kavraki, "A two level fuzzy PRM for manipulation planning," *IEEE/RSJ International Conference on Intelligent Robotics and Systems*, pp. 1716–1722, 2000.

[13] N. M. Amato, O. B. Bayazit, L. K. Dale, C. V. Jones, and D. Vallejo, "OBPRM: An obstacle-based PRM for 3D workspaces," in *Robotics: The Algorithmic Perspective. Proceedings of the Third Workshop on the Algorithmic Foundations of Robotics*, P. K. Agrawal, L. E. Kavraki, and M. Mason, Eds. Natick, MA: A.K. Peters, 1998, pp. 155–168.

[14] S. A. Wilmarth, N. M. Amato, and P. F. Stiller, "MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, vol. 2, 1999, pp. 1024–1031.

[15] D. Hsu, T. Jiang, J. Reif, and Z. Sun, "Bridge test for sampling narrow passages with proabilistic roadmap planners," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2003, pp. 4420–4426.

[16] V. Boor, M. H. Overmars, and A. F. van der Stappen, "The Gaussian sampling strategy for probabilistic roadmap planners," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, vol. 2, 1999, pp. 1018–1023.

[17] S. Sundaram, I. Remmler, and N. Amato, "Disassembly sequencing using a motion planning approach," in *Proc. IEEE Int. Conf. Robotics and Automation*

(ICRA), 2001, pp. 1475–1480.

[18] O. B. Bayazit, J.-M. Lien, and N. M. Amato, “Roadmap-based flocking for complex environments,” in *Proc. Pacific Graphics*, Oct 2002, pp. 104–113.

[19] A. Schweikard, R. Tombropoulos, L. E. Kavraki, J. Adler, and J.-C. Latombe, “Treatment planning for a radiosurgical system with general kinematics,” in *Proceedings of the IEEE/RJS International Conference on Robotics and Automation (ICRA)*. San Diego, CA: IEEE Press, 1994, pp. 1764–1771.

[20] T. Lozano-Pérez and M. A. Wesley, “An algorithm for planning collision-free paths among polyhedral obstacles,” *Communications of the ACM*, vol. 22, no. 10, pp. 560–570, October 1979.

[21] S. M. LaValle. Motion Strategy Library. Accessed October, 2005. [Online]. Available: http://msl.cs.uiuc.edu/msl/

[22] A PRM planner for rigid objects in 3D. Physical and Biological Computing Group. Accessed October, 2005. [Online]. Available: http://www.cs.rice.edu/CS/Robotics/links.html

[23] R. Isaac, “Vizmo 3D: A Visualization Tool for Motion Modelling,” Masters Project Report, Department of Computer Science, Texas A&M University, College Station, TX, 1998.

[24] I. Sommerville, *Software Engineering*, 5th ed. Reading, MA: Addison-Wesley, 1996.

[25] R. M. Baecker, J. Grudin, W. A. S. Buxton, and S. Greenberg, *Readings in Human-Computer Interaction: Toward the Year 2000*, 2nd ed. San Francisco, CA: Morgan Kaufmann Publishers Inc., 1995.

[26] J. H. Reif, “Complexity of the mover’s problem and generalizations,” in *Proc. IEEE Symp. Foundations of Computer Science (FOCS)*, San Juan, Puerto Rico,

October 1979, pp. 421–427.

[27] S. A. Wilmarth, “A Probabilistic Method for Rigid Body Motion Planning Using Sampling from the Medial Axis of the Free Space,” Ph.D. dissertation, Department of Mathematics, Texas A&M University, College Station, TX, 1999.

[28] S. A. Wilmarth, N. M. Amato, and P. F. Stiller, “Motion planning for a rigid body using random networks on the medial axis of the free space,” in *Proc. ACM Symp. on Computational Geometry (SoCG)*, 1999, pp. 173–180.

[29] J.-M. Lien, S. L. Thomas, and N. M. Amato, “A general framework for sampling on the medial axis of the free space,” in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, September 2003, pp. 4439–4444.

[30] J. van der Zijp. (1997-2005) FOX Graphical User Interface Toolkit. Accessed October, 2005. [Online]. Available: http://www.fox-toolkit.org/

[31] (2004) OpenGL. [Online]. Available: http://www.opengl.org/

[32] (1993-2003) Open Inventor. Silicon Graphics, Inc. [Online]. Available: http://oss.sgi.com/projects/inventor/

[33] (1993-2005) OpenGL Performer. Silicon Graphics, Inc. [Online]. Available: http://www.sgi.com/software/performer/

[34] (1999) Proximity Query Package (PQP). University of North Carolina. [Online]. Available: http://www.cs.unc.edu/~geom/SSV/

[35] S. M. LaValle. (1999-2004) Planning algorithms. [Online]. Available: http://msl.cs.uiuc.edu/planning/

[36] T. C. Zhao and M. Overmars. (1995) XForms. [Online]. Available: http://world.std.com/~xforms

[37] (1996) Geomview. Geometry Center, University of Minnesota. Accessed October, 2005. [Online]. Available: http://www.geomview.org/

[38] RAPID: Robust and Accurate Polygon Interference Detection. UNC Research Group on Modeling, Physically-Based Simulation and Applications. Accessed October, 2005. [Online]. Available: http://www.cs.unc.edu/~geom/OBB/OBBT.html

[39] Tool Command Languaje (Tcl/Tk). ActiveState. Accessed October, 2005. [Online]. Available: http://www.tcl.tk/software/tcltk/

[40] The Visualization Toolkit VTK. Kitware Inc. Accessed October, 2005. [Online]. Available: http://public.kitware.com/VTK/

[41] S. LaValle, J. Yakey, and L. Kavraki, "A probabilistic roadmap approach for systems with closed kinematic chains," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 1999, pp. 1671–1676.

[42] J. J. Craig, *Introduction to Robotics: Mechanics and Control*, 2nd ed. Reading, MA: Addison Wesley, 1989.

[43] D. M. S. Gottschalk, M.C. Lin, "OBBTree: A hierarchical structure for rapid interference detection," in *International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH).* New Orleans, LA: ACM Press, 1996, pp. 171 – 180.

[44] S. C. M. Reggiani, M. Mazzoli, "An experimental evaluation of collision detection packages for robot motion planning," in *IEEE International Conference on Intelligent Robots and Systems, IROS'02.* Lausanne, Switzerland: IEEE Press, 2002.

[45] P. Dorsey and P. Koletzke, *Oracle Designer/2000 Handbook.* Berkeley, CA: Osborne McGraw-Hill, 1997.

[46] (2002) Movie.BYU file format. Brigham Young University. [Online]. Available: http://lc.cray.com/doc/movie/

[47] J. D. Gould, "How to design usable systems," in *Readings in Human-Computer Interaction: Toward the Year 2000*, 2nd ed. San Francisco, CA: Morgan Kaufmann Publishers Inc.; Excerpt, 1995, pp. 757–789.

[48] D. Petrelli, A. D. Angeli, and G. Convertino, "A user-centered approach to user modeling," in *Proc. ACM, International Conference on User Modeling*, 1999, pp. 255–264.

[49] Qt. Trolltech. Accessed October, 2005. [Online]. Available: http://www.trolltech.com/products/qt/

[50] (1997-2005) Unified Modeling Language (UML). Object Management Group. [Online]. Available: http://www.uml.org/

[51] "Persistence of Vision Raytracer (pov-ray)," Persistence of Vision Raytracer Pty. Ltd, 2005. [Online]. Available: http://www.povray.org/

[52] (1996-2005) RenderMan. Pixar. [Online]. Available: https://renderman.pixar.com

[53] J. Kim, J.-M. Lien, A. Vargas, R. Pearce, and N. M. Amato. (2002) Texas A&M Campus Navigator. Parasol Lab, Texas A&M University. [Online]. Available: http://parasol.tamu.edu/groups/amatogroup/research/campus-nav/

[54] B. S. Sandhu, "A visualization tool to study the motion of complex 3d objects in space," Fellows Thesis, Department of Computer Science, Texas A&M University, College Station, TX, 2003.

VITA

Aimée Vargas Estrada received her B.S. in computer engineering at Universidad Nacional Autónoma de México (UNAM), Mexico City, in 1996. She was awarded a two-year scholarship from the Consejo Nacional de Ciencia y Tecnología (CONACYT), México, to pursue her master of computer science at the Department of Computer Science, Texas A&M University. Aimée Vargas may be reached at 301 Harvey R. Bright Building, College Station, TX, 77843-3112. Her email address is aimee@cs.tamu.edu.