d

DESIGN, DEVELOPMENT& IMPLEMENTATION OF THE MPK, A MOTION PLANNING KERNEL

By

Ian J. Gipson

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

BACHELOR OF APPLIED SCIENCE

in the school of

Engineering Science

© Ian J. Gipson, 1999

SIMON FRASER UNIVERSITY

August, 1999

All rights reserved. This work may not be reproduced

in whole or in part, by photocopy or

other means, without permission of the author.

Approval

NAME: Ian Gipson

DEGREE: Bachelor of Applied Science (Engineering Science)

TITLE OF THESIS: Design, Development & Implementation of the MPK, a Motion Planning Kernel

____
Dr. John Jones
Director
School of Engineering Science, SFU

EXAMINING COMMITTEE:

Chair and Academic Supervisor

____
Dr. Kamal Gupta
School of Engineering Science

Committee Member

____
Dr. John Dill
School of Engineering Science

Date Approved ____________________________

Abstract

The Motion Planning Toolkit (MPK) is a joint research project conducted with input from researchers at Simon Fraser University, the National Research Council, and International Submarine Engineering. The MPK system is a programming project that I undertook as partial fulfillment of my bachelor's thesis requirements, representing my effort to create a general software toolkit that programmers could use in the development of motion planning algorithms. A motion planning algorithm attempts to plan a path for a robot operating in a complex environment such that the robot will not collide with its surroundings.

The MPK system is made up of two distinct sub-projects; a code toolkit and a web based application. The code toolkit refers to a software library that third party programmers will use to speed up development of motion planning algorithms. It consists of code that allows programmers access to general robotic data structures and algorithms. The web based application serves as a platform for demonstrating the functionality of the MPK as well as for allowing us to test and debug the underlying code.

The web based application is a client server application running over the Internet. Implementing it this way allows researchers who may be interested in the MPK to evaluate its performance through the web, before they go through the trouble of downloading and compiling what could easily be a very large body of code. It also allows them to evaluate the suitability of various planners to specific problems, and perform some degree of benchmarking on different algorithms.

My focus on this project was to come up with an overall system design, then to focus on the implementation details of two key components, geometry and kinematics. In addition, I also designed other areas of the system, including the server side of the web based application. Currently, the core system architecture is fairly complete. I anticipate that it will require only minor changes to support future development work. The implementation, however, is not complete.

Acknowledgements

I would like to acknowledge the contributions of many individuals who helped to make this project a success. Specifically, the assistance and guidance of Dr. Kamal Gupta. Additionally, I would like to thank Dr. John Dill, along with Michael Greenspan of the National Research Council and Eric Jackson of International Submarine Engineering for their contribution and suggestions during the design phase of the project. I would like to acknowledge Gillian Lo, Javier Francisco Blanco and Juan Manuel Ahuactzin for their contributions with regard to helping to implement various portions of the project as well as Yong Yu for his advice and sample code. Lastly I would like to thank Morten Lind Petersen for his suggestions and interesting discussion of the MPK project.

Table of Contents

Approval ii
Abstract iii
Acknowledgements iv
Table of Contents v
List of Figures & Tables vii
List of Acronyms and Technical Terms viii
Chapter 1 Introduction 1
1.1 Robot Motion Planning: What is it? 1
1.2 Current State of the Field 1
1.3 MPK Project 2
1.4 My Contribution to the Project 5
1.5 Thesis Layout 6
Chapter 2 MPK Design 7
2.1 Maintainability 7
2.2 UML Overview 7
2.2.1 Objects 8
2.2.2 Object Containment 1
2.2.3 Class Inheritance 2
2.3 User Interface Functional Specifications 4
2.3.1 User Creates a Robot in the System 5
2.3.2 User Designs an Environment 6
2.3.3 User Formulates a Planning Task 6
2.3.4 Choosing a Planner 8
2.3.5 Evaluating the Results 9
Chapter 3 System Code Design Overview 11
3.1 Basic Types of Mathematical Entities 15
3.1.1 3D Vectors 15
3.1.2 N Dimensional Vectors 16
3.1.3 4x4 Matricies 17
3.2 Universe 19
3.3 Kinematics Module 20
3.3.1 Frames 20
3.3.2 Links 21
3.3.3 Links Defined Using Denavit Hartenberg Parameters 23
3.3.4 Frame Manager 26
3.4 Geometry Module 28
3.4.1 ObjectBase 28
3.4.2 Polyhedral Meshes 32
3.4.3 Spheres 36
3.4.4 Line Segment 38
3.4.5 Groups 40
3.4.6 Future Improvements 42

3.5 Collision Detection 42
3.5.1 Simple Collision Detection 43
3.5.2 Vcollide 43
3.5.3 Solid 45
3.6 Planners 45
3.6.1 General Structure of a Planner 46
3.6.2 Linear 48
3.6.3 Random 49
3.6.4 ACA 50
3.6.5 Sequential 51
3.7 File I/O 52
3.7.1 Serializeable class 52
3.7.2 Polymorphic Deserialization 53
3.7.3 File Structure 54
3.8 Internet Connectivity 55
3.8.1 Client Server Architecture 56
3.8.2 Java Front End 56
3.8.3 C++ Server 58
3.8.4 UI Prototype 58
Chapter 4 Future Improvements 60
4.1 Moving Objects 60
4.2 Multiple Robots and Cooperative Planners 62
4.3 Movable Objects 63
4.4 Unknown Static Environments 64
4.5 Non-Motion Planning Tasks 65
Chapter 5 Conclusions 66
Chapter 6 References 68
Appendix 1: Denavit Hartenberg Links 70

List of Figures & Tables

Figure 1: Client Server System Architecture 3
Figure 2: UML Class 8
Figure 3: C++ Equivalent of an Object 8
Figure 4: Object with an Attribute 8
Figure 5: UML Visibility Modifiers 1
Figure 7: C++ Code Fragment for a Class with an Attribute 1
Figure 8: UML Object with an Operation 1
Figure 9: Example of an Aggregate Class 2
Figure 10: Code Generation of Aggregate Classes 2
Figure 11: Class Inheritance in UML 3
Figure 12: Code Fragment for Derived Class 3
Figure 13: Illustration of the IS-A vs. HAS-A Relationship 3
Figure 14: Illustration of Explanatory Arrows 4
Figure 15: User Workflow for the Web Based Application 5
Figure 16: Movement of Joints, and Selection of Start and Goal Configurations 8
Figure 17: Pull Down Planner Menu Allows Choice of Different Planners 9
Figure 18: Illustration of Shadow Images 10
Figure 19: Use Case Scenario for a Typical Planning Task 12
Figure 20: Relationship Between Planners, Interfaces and Collision Detectors 14
Figure 21: Vector4 Class 15
Figure 22: N Dimensional Vector Class 17
Figure 23: 4x4 Matrix Object 18
Figure 24: Both Robots and Obstacles Derive From Class Entity 19
Figure 25: A Universe Contains Many Entities 19
Figure 26: Frame Objects Derive From 4x4 Matrices 20
Figure 27: Frame Class Illustrating *baseframe* Attribute 21
Figure 28: Link Object Contains Geometry Objects 22
Figure 29: DH Link Object Inherits From Link Base Class 24
Figure 30: Frame Manager Stores all the Frames 26
Figure 31: All Geometrical Objects Derive from ObjectBase 29
Figure 32: Each Geometric Object Contains an Offset Frame 29
Figure 33: Mesh Object Inherits From Object Base 33
Figure 34: Mesh Objects Contain Vertex Points, and Facet Lists 33
Figure 35: Facet Object 34
Figure 36: Sphere Object Represents a Sphere with Position and Radius 37
Figure 37: Line Segment Object 38
Figure 38: A Group Both Contains and Derives From *ObjectBase* 41
Figure 39: Difference Between a Global and a Local Planner 46
Figure 40: Class Hierarchy of Planner Objects 47
Figure 41: Code of the Linear Planner 49

Table 1: The Modules Used in the MPK 11

List of Acronyms and Technical Terms

Term	Definition
ACT	All purpose software library for robotics. ACT is compiled for a UNIX environment using C, and provides some of the same functionality as the MPK system. Unfortunately, ACT is not an open source system.
Collision Detection Query	A request from a planner object to the collision detection module. The planner needs additional information about the 3D Structure of the universe. For example: is the robot in a collision state given the current set of joint variables?
configuration	A set of variables completely specifying the position of a robot.
CSG	Constructive Solid Geometry. Defining solid models in terms of regularized boolean operations performed on a small set of primitive objects. For example, a bowling ball would be represented by the difference of a sphere, and three little cylinders for the finger holes.
Denavit Hartenberg Parameters	A set of numbers that completely describe how one frame in a robotic kinematic chain relates to the previous frame. A, Alpha, D, and theta are the 4 parameters that make up the set.
DH Parameters	see Denavit Hartenberg Parameters
Euler Operator	A geometric operation used in the incremental construction of solid polyhedral models. Operators allow for modifications of the local geometry. Collapsing a face to become a vertex, or creating a hole in an existing face, are examples of Euler operators.
IGRIP	IGRIP is a modern Computer Aided Robotics system. It can be used for simulation of robot systems and as a tool in developing new manipulators.
ISE	International Submarine Engineering
MFC	Microsoft Foundation Classes. A set of wrapper classes that make Microsoft Windows programming easier to do when using C++
motion planning	The computational task of planning the motion of a robot from one configuration to another.
MPK	Motion Planning Kernel, this is the software project described in this document
NRC	National Research Council of Canada
ORD	Object Relationship Diagram. A diagram that describes the manner in which objects in an object oriented program relate to one another.

RTTI	Run Time Type Identification. A C++ mechanism for determining the type of object that a pointer is pointing to at run time. This allows for programming while using data structures that represent collections of objects of mixed type.
UML	Universal Modeling Language. A graphical descriptive language that standardizes the creation of object relationship diagrams (ORDs)

Chapter 1 Introduction

1.1 Robot Motion Planning: What is it?

"Motion planning, in its broadest sense, refers to the ability of a robot to plan its own motions."[1] More specifically, the basic motion planning problem[2] is defined as follows: given an initial and goal robot configuration, find a path between them, avoiding collisions with all obstacles along the way. Robotic motion planning incorporates many, somewhat diverse, areas of study including artificial intelligence and geometric reasoning. It also has applications to areas outside of pure robotics, including uses in animation and in the video game industry. In fact, many of the concepts used in motion planning fall under the broad header of geometric reasoning – solving problems based on geometric descriptions. These problems may include motion planning, grasping problems, part positioning, etc.

1.2 Current State of the Field

The field of robot motion planning comprises a large body of academic research and literature while remaining an active area of investigation. Although many researchers believe that aspects of the technology could be well utilized in industrial systems, there has been little direct industrial application of motion planning methods. One contributing factor for this lack of application is the scarcity of proper tools. Commercially available robot simulators (ACT, IGRIP, ROBCAD) tend to have some limited motion planning capabilities, typically oriented towards a particular technique. The majority of research efforts are focused on developing new representations and search methods, and the code artifacts tend to be of a home-grown variety, unsuitable for extensive reuse.

Since the late 1970s, a big tool in solving motion planning problems has been the notion of configuration space (C-space)[2] of the robot. This C-space represents an N

dimensional space in which each dimension corresponds to one degree of freedom of the manipulator[3]. A point in C-space, therefore, corresponds to a configuration of the robot where each joint is at a position represented by its value along the corresponding C-axis. The configuration represented by a point in C-space can either be a valid, collision free, one for a robot, or the robot can be in collision if placing it in that configuration would cause it to interfere with an obstacle. The regions that correspond to the robot being in collision with obstacles are termed C-obstacles. Since the mid 1980s, it has been apparent that obtaining an analytic description of C-obstacles is very difficult. The equations that govern its correspondence to the robot's position in Cartesian space are highly non-linear. After this time, use of a discretely sampled version of C-space has come into vogue. C-space is sampled, and a search mechanism operates on the discrete samples to complete the motion planning task. This search mechanism is what we describe as a *planner* for motion planning purposes. The context of planning system that I will use is that of a search mechanism (planner) in conjunction with a collision detector, a mechanism that can check if a discrete robot configuration is in collision or not.

Comparing different methods of motion planning is complicated by the lack of standardized tools; there is no way to place these planners on equal footing for the comparison. The relative advantages and drawbacks of any given method are often subtle and situation dependent. An enumeration of an algorithm's computational complexity is provided in most papers, but is rarely sufficient for a meaningful comparison of algorithms. Something more is clearly required.

1.3 MPK Project

The motion planning kernel (MPK) is aimed at addressing these issues and is a general system for testing various motion planning algorithms on different robot manipulator configurations. The system was designed with generality in mind, so as few limitations as possible were placed on manipulator structure and environment configuration. This design allows a uniform evaluation of the speed and effectiveness of various motion planning algorithms when applied to particular situations.

The system is an open source project that will be easily extensible by future researchers. This ease of extensibility will allow for the addition of new motion planning algorithms as they are developed and the underlying system will be general enough so as to make these additions as straightforward as possible. Currently, the system has been completed to the point where the basic motion planning problem discussed in 1.1 can be addressed. Users are currently able to define robots, and perform collision detection on the robots they have created. This functionality is all that is required for the creation of planners to solve the basic problem. In addition, several planners have already been implemented within the MPK framework for benchmarking purposes. Future additions to the project will include unknown environments, sensors, simulation, multiple robots, dynamic analysis, specification of forces, and speeds, etc. Provisions to allow for this type of functionality have been built in from the initial design phase through to the current state of the project.

The MPK has two distinct facets. First, it represents a code library that programmers can use when developing systems requiring robotic information or geometric reasoning tasks. Second, it represents an interactive system that researchers can use over the web to try out various motion planning algorithms on specific problems, evaluating planner performance and benchmarking speed. This web based application operates on a client server model, using a Java front end running through a web browser. The front end communicates with a server application running locally on an SFU machine.

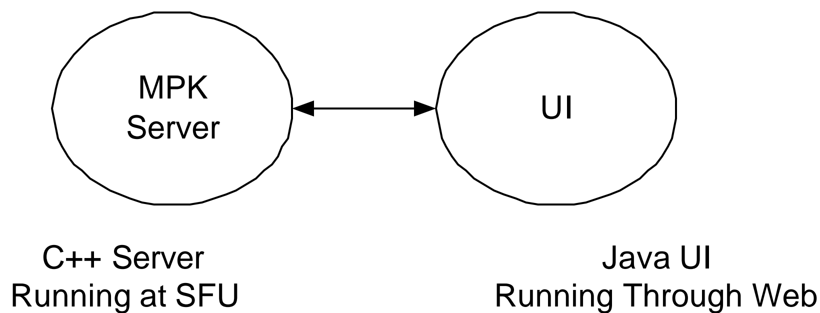

Figure 1: Client Server System Architecture

We chose a Java Front end because it is a cross platform, cross browser language, so it gives us a large base of people that can test out the MPK. C++ was chosen for the server side implementation due to the large number of third party libraries that exist in C/C++ that could be added to the server. PC platforms were chosen because one of the partners in the project, ISE, uses PC's almost exclusively.

What the MPK contributes to the field of robotics and motion planning is a general platform upon which to build programs that require a foundation in robotic geometry. The MPK provides users a code framework that allows them to set up robot kinematic structures easily and subsequently import geometry that will flesh out the description of the robot.

Once a robot has been described, the user can use it in a very high level fashion. All the mundane "bookkeeping" is taken care of by the MPK. Hiding the bookkeeping allows users to focus on more high level problems, such as the design of new algorithms, without getting needlessly bogged down in the details of designing a system that could easily become quite large.

Additionally, writing code in an MPK framework allows users immediate access to the wide range of sample data that we have created. Designing simple robots is not a difficult task, but as the complexity of kinematics and geometry increases, the time required to design the robot also increases. Using the MPK allows users to use pre-created libraries of robots, and working environments, making testing of algorithms a simpler task and allowing for algorithm benchmarking to be performed on an equal footing.

First and foremost, the MPK (Motion Planning Kernel) system is envisioned as a testbed in which researchers can evaluate different motion planning algorithms in situations that closely resemble the real world applications they may be considering. The user of the system should be able to specify a manipulator structure, the kinematics and geometry that define a robotic manipulator. They will also be able to specify the workspace of this

robot along with any obstacles that may exist within the workspace. The user will then define a task for the robot to perform. In the simplest motion planning example, this task would be simply a start and a goal configuration for the manipulator.

Once the user has specified all the criteria above, the evaluation can be performed. Multiple algorithms are available in the system that are capable of tackling the problem that the user has specified. The user will be able to run each of them in turn, observing the results, and measuring the time the algorithm took to perform its task. In this way, users can decide for themselves which algorithms best suit the particular problem they need to solve.

1.4 My Contribution to the Project

This thesis focuses on the design and implementation of several core areas of the MPK. The overall design of the MPK is discussed; along with a detailed design of two core areas to which I contributed. The two areas I worked extensively on are the Geometry module and the Kinematics module of the system. I left other aspects of the project such as the user interface, and Internet connectivity sections to others, although some work on these areas was necessary to get the system up and running.

The MPK system that has been developed to this point contains multiple motion planning algorithms, along with several different collision detection schemes for these algorithms to use. Currently, a prototype user interface to the MPK using the Microsoft Foundation Classes (MFC), has been designed. MFC was chosen because it speeds up UI prototyping on the PC. This UI is a standalone program that runs on a Windows platform. A web based application, complete with a JAVA front end has been developed that conforms to this prototype. See [8] for more information.

Since the MPK will serve as a code toolbox for future robotics based application development, object oriented structures (classes) for robotic manipulators, environments, collision detection, and motion planning have been created and made available to the end user. This modularity is done in such a way that making additions or modifications to

any of the above components will be easy to accomplish, but should be generally unnecessary. The robotic classes should be adequate as they stand for most tasks.

The extensibility of the system is due in part to its open source nature, in much the same way as other institutions provide academic software systems. For example, the University of Utrecht in the Netherlands provides an open source library for computational geometry – CGAL, and The University of North Carolina provides several libraries for collision detection of polyhedral meshes. Some of these libraries are used within the MPK system, in much the same way that I hope my system will find its way into other more complicated projects.

1.5 Thesis Layout

The overall MPK system design is described in Chapter 2. It includes an overview of the various modules of the system, along with a functional specification for the web based user interface.

Chapter 3 goes into far more depth of the system design than does Chapter 2. It describes the inner workings of the modules, as well as functional descriptions of all the important objects in the system.

Because the MPK is far from complete, Chapter 5 outlines various directions in which the project can be taken. Many of the suggestions outlined in this chapter already have the groundwork laid for their implementation.

Chapter 2 MPK Design

In order to design a software system of this magnitude, a very detailed design approach was followed. System maintainability is of the utmost importance, since other researchers will use the code written for the MPK in their software systems.

2.1 *Maintainability*

A big problem associated with systems as large as the MPK is that when future researchers examine the code, they often won't know where to begin. What will be beneficial to them is some sort of roadmap illustrating which sections of code interact with each other, and in what manner. In the context of an object oriented system, which the MPK is, this roadmap translates to the need for an Object Relationship Diagram (ORD). Many tools exist for creating such a diagram, some of which also incorporate automatic code generation and commenting. I chose to use an evaluation version of the Rose tool from Rational Software to assist me in this area.

Rational Rose is a software tool that allows me to draw universal modeling language (UML) diagrams, annotate them, and generate an automatic code framework from the UML structure. Once the UML diagram has been created, it is extremely easy to see what parts of the software interact with one another, and in what manner.

2.2 *UML Overview*

Given that several UML diagrams will be appearing throughout this document, a brief tutorial on the symbols and structure of UML is in order. I will tailor this discussion to UML as applied to an object oriented C++ program, since that is the area in which I used it in this project. Of course, UML is applicable to a wide variety of problems beyond C++ code, but that is beyond the scope of this document.

2.2.1 Objects

The basic entity in Object Oriented Programming (OOP) is the Object. In C++, this entity is equivalent to the *class*. UML represents classes graphically as shown in Figure 2.

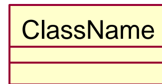

Figure 2: UML Class

A class defined as above would generate C++ code equivalent to that shown in Figure 3

```
class ClassName
{
};
```

Figure 3: C++ Equivalent of an Object

The name of the class is shown in the topmost field of the box in the UML diagram, followed by two additional fields below. These additional fields can contain class *attributes* and *operations*. Attributes are stored in the first field and are best thought of as data members of the C++ class. Figure 4 shows an example of a class with an attribute. The attribute is displayed with its name, followed by its data type, in this case: unsigned int, one of C++'s atomic types.

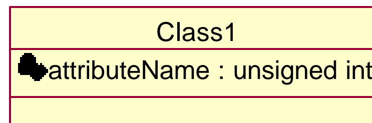

Figure 4: Object with an Attribute

Immediately to the left of the attribute's name, is a modifier specifying the visibility of the data member. These represent the C++ reserved words public, protected, and private. The set of UML visibility modifiers is shown in Figure 5.

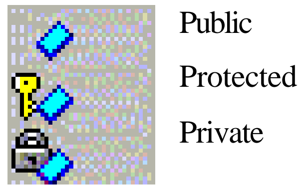

Figure 5: UML Visibility Modifiers

The plain square indicates an attribute with public visibility, the key, protected visibility, and the lock, private visibility.

The UML object shown in Figure 4 would generate C++ code similar to

```
Class class1
{
private:
     unsigned int attributeName ;
}
```

Figure 6: C++ Code Fragment for a Class with an Attribute

In addition to attributes, UML objects can also contain operations, shown graphically in the bottommost field of the class diagram. These are equivalent to methods or member functions in traditional Object Oriented terminology. A UML object with an operation is shown in Figure 7.

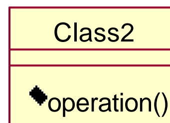

Figure 7: UML Object with an Operation

2.2.2 Object Containment

In addition to class attributes, UML also supports an *aggregation* modifier that helps create aggregate classes. The diamond arrow in Figure 8 indicates that ClassA contains ClassB. In fact, there are two separate aggregate class modifiers. The white diamond indicates that ClassA contains ClassB by reference – it contains a pointer to ClassB only,

while the solid black diamond indicates ClassA contains ClassB by value, it is completely contained by ClassA.

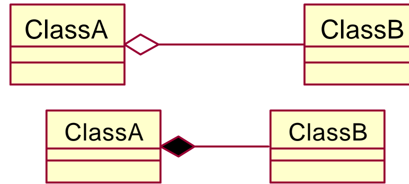

Figure 8: Example of an Aggregate Class

In Figure 8, the white diamond aggregation would generate code similar to the following:

```
class ClassB
{
};
class ClassA
{
private:
     ClassB* myClassB ;
}
```

Figure 9: Code Generation of Aggregate Classes

2.2.3 Class Inheritance

Object Oriented programming maintains that code reuse can be enhanced through the mechanism of class inheritance. This paradigm holds that a parent class will pass on all of its traits to classes that are derived from it. In order to make class derivation graphically clear in UML, an additional modifier is presented, that is the *generalization* modifier.

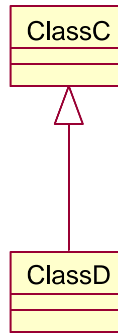

Figure 10: Class Inheritance in UML

In Figure 10, ClassD inherits from ClassC, and would generate code similar to the following:

```
class ClassD
{
};
class ClassC: public ClassD
{
}
```

Figure 11: Code Fragment for Derived Class

The difference between class inheritance and class aggregation can be explained through the IS-A vs. HAS-A relationship. Inheritance represents the IS-A relationship, while aggregation represents a HAS-A relationship. Figure 12 shows the difference between the two. In this example, a sailboat IS-A boat, thus it inherits from the boat class. The sailboat also HAS-A sail. The parent class (Boat) in turn HAS-A captain, thus the child class, sailboat, also HAS-A captain through inheritance.

Figure 12: Illustration of the IS-A vs. HAS-A Relationship

2.2.4 Explanatory Arrows

Not all the symbols in UML have a direct code equivalent. Explanatory arrows for example are used merely to convey a concept, and not to refer to any code per se.

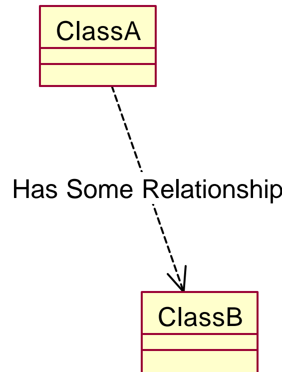

Figure 13: Illustration of Explanatory Arrows

The explanatory arrows shown in Figure 13 are a perfect example of this. The two classes have some relationship to one another, but it is not as well defined as the IS-A or HAS-A relationship. Mostly, this is used as a descriptive tool to explain some additional relationship between two classes.

2.3 User Interface Functional Specifications

As mentioned in section 1.3, one aspect of the MPK is an interactive system in which users design robots and environments, and test various motion planning tasks. This application makes use of much of the code in the code toolkit aspect of the project, and therefore guarantees that the code toolkit is adequately exercised from a software testing perspective.

A complete design of the interactive system includes a web based front end, communicating with a server system resident on a PC based Windows NT system. Running the server on a Windows NT PC platform is not a design requirement, per se, however my existing familiarity with this platform made design much easier. From the user's perspective there should be no difference perceived with regard to what particular platform the server is running on.

The typical scenario in which a user would operate this application is shown in Figure 14

1. Design a robot
2. Design an environment composed of obstacles
3. Formulate a planning task
4. Choose algorithms, and run a test
5. Evaluate the results

Figure 14: User Workflow for the Web Based Application

Once the user completes this cycle, they will likely wish to iterate on some part of the design. Thus, the ability to modify any part of the user's design is an integral consideration in all functional areas of the program. In particular, the user may want to alter the motion planning algorithm and the planning task (start and goal configuration) they requested. Alterations to the robot and the environment obviously will take place, but are likely to be less frequent.

2.3.1 User Creates a Robot in the System

Designing a robot is a non-trivial task for the user. It involves specifying the kinematic structure of the along with the physical geometry representing each link. Currently the MPK supports the most common way of describing kinematic structure; that of Denavit-Hartenberg (DH) notation, making specification of a robot rather straightforward. All the user need supply to describe a link is the DH parameters associated with it, and the previous link in the chain. Since many physical robots already have DH parameters defined for them, often from the factory, specifying them is not likely to be difficult.

Previous link information for each link in the chain is required so that the MPK can support branched robot structures such as a robotic hand. DH parameters by themselves can only specify a single, unbranched, open chain robot. Adding the additional information allows us to maintain the familiar DH notation, but support a wider array of robots.

The geometric description of a robot is somewhat more difficult for the user to define interactively. An online description of complex polyhedral meshes is not a feasible solution to the geometry problem because it is far too arduous for a user to specify.

Instead, the MPK web based interface supports reading predefined geometry from the user's file system through VRML 1.0 and 2.0 file formats. Also supported will be the online definition of simple geometric entities – rectangular boxes, spheres, etc using the mouse and the keyboard. Although a user using only the web based application may be somewhat limited with regard to the complexity of robot and environment that can be created, the tradeoff in terms of usability and rapid prototyping is a good one.

For the time being, the prototype, and the web based application force the user to select a robot from a menu of pre-created robots. This removes the need for a complex UI that would allow them to design a robot online.

2.3.2 User Designs an Environment

The environment, or workcell of the robot is also something the user needs to define the geometry for. Again, both loading from file or online description are supported. In addition to specifying the environment one time, it is extremely likely that the user will want to modify the configuration of the environment during use of the system, meaning that the interface had to support moving and deleting obstacles already in existence. Supporting these features proved to be one of the more difficult user interface aspects of the system.

Designing an environment for a robot is also a task that is in its infancy in the prototype and in the web based application. Currently, several environments can be selected off a menu, while additional obstacles can be placed using the mouse.

2.3.3 User Formulates a Planning Task

Formulating a planning task is one of the simpler pieces of interaction a user has to perform. All the planners currently in the system require the same information about their task – a start and a goal configuration. Once a user has specified these two configurations, the planner can go to work. The way in which a user would specify the start and goal configuration is very intuitive – sliders exist that alter the values of various joint variables, and a drop down list box allows you to select between different joints. The user receives immediate feedback as to what the changes in joint values are doing to

the robot by observing the robot moving in its environment onscreen. For additional feedback, a checkbox indicates whether or not the current robot is in a collision configuration with its environment.

As Figure 15 illustrates, a selector exists that allows users to pick which joint of the robot to move. This selector enables a slider that allows individual joint values to be altered. The rendering window immediately reflects the changes the user makes. Push buttons are present for selecting start and goal configurations.

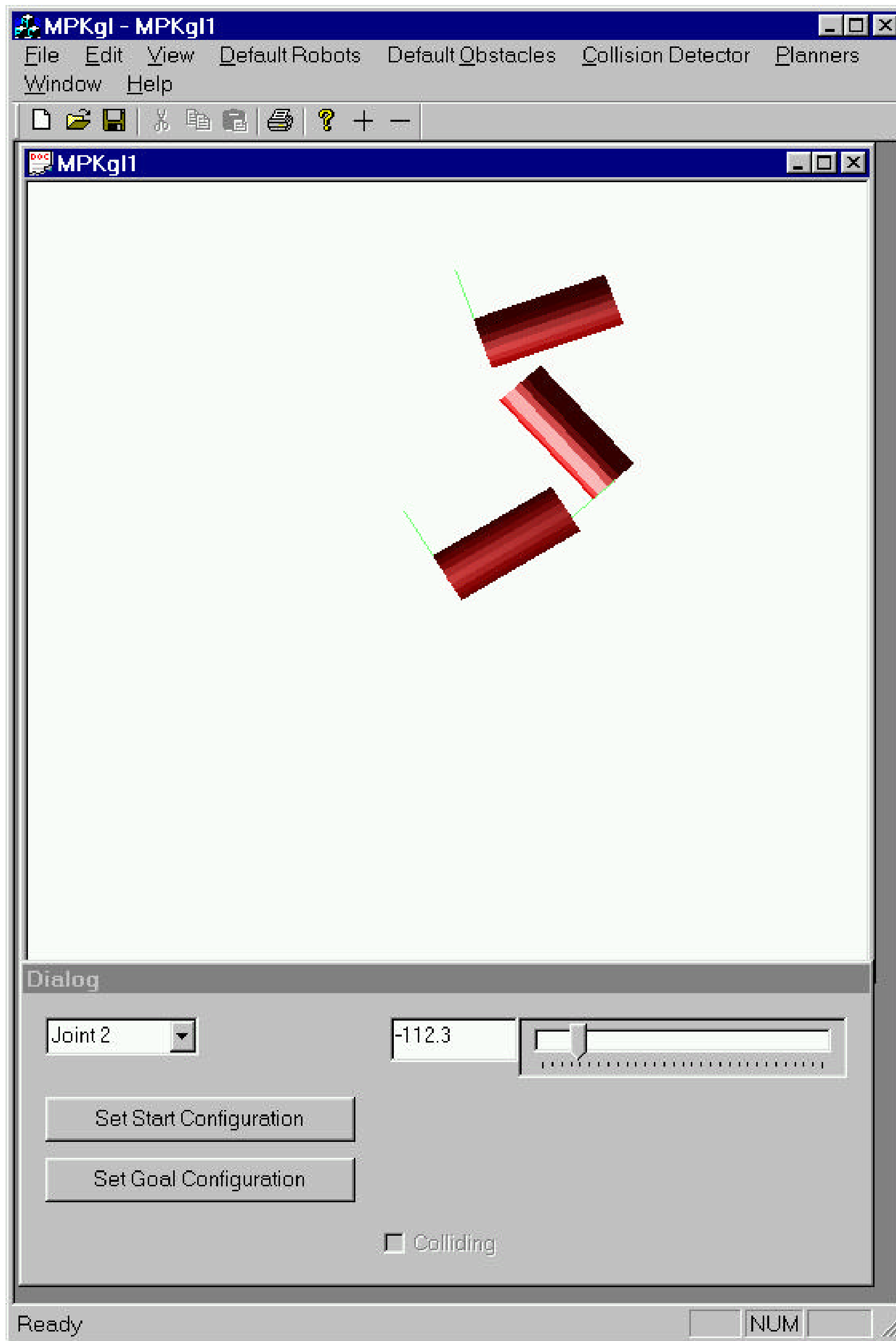

Figure 15: Movement of Joints, and Selection of Start and Goal Configurations

2.3.4 Choosing a Planner

Choosing a planner and allowing it to perform a planning task are also fairly trivial for the user. The pull down planner menu contains a list of planners from which the user can select, and the collision detector menu allows a choice of collision detection algorithms.

Some collision detection algorithms require additional parameters to tune their efficiency – when chosen, these will bring up an additional dialog box in which all necessary parameters can be specified.

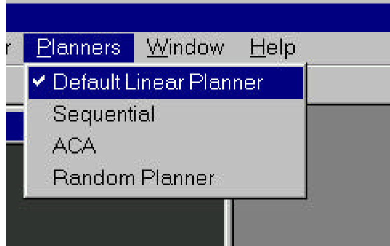

Figure 16: Pull Down Planner Menu Allows Choice of Different Planners

Once a planner and a collision detector have been chosen, the user merely needs to press the plan button, and observe the results. One drawback to the current system, however, is that once a planner has been engaged and has started its planning task, there is no way to abort without terminating the entire program. This is because the entire planning operation is running in a single thread, and it does not check the system clock or any other signal to determine when to quit. The lack of abort capability has the unfortunate consequence of causing you to have to shut the program down entirely if the combination of planner parameters and workcell configuration results in extremely long computation times. This drawback needs to be further addressed when a more in depth user interface is developed. There should be a method of terminating a planning task that is taking too long. One common test case for this situation in a planner is the situation in which there is no valid path from start to goal. In this case, the planner will often take maximal time while searching for a path because it doesn't know when to quit. The user may have specified this environment by mistake, not realizing there was no path, and should be given the option of aborting.

2.3.5 Evaluating the Results

In order for the user to determine whether a given planner performed adequately on the robot and environment provided, there needs to be a mechanism for evaluating the resultant path. A path can be visualized in two ways, the obvious way is to show an animation of the robot moving through the path, while the second way is to show some of

the robot's intermediate positions as "shadow images" onscreen. Figure 17 shows how shadow images appear in the prototype UI. The number of shadows is user selectable.

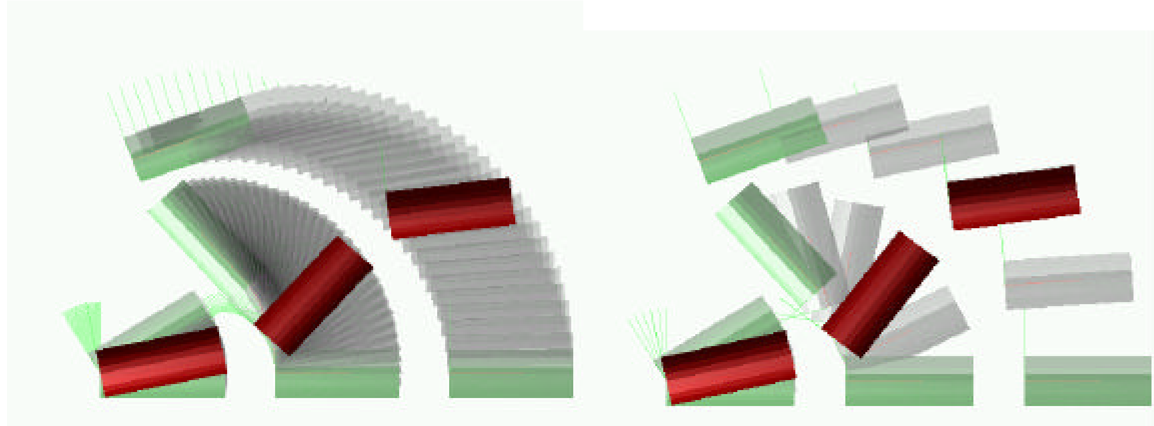

Figure 17: Illustration of Shadow Images

Another aspect of evaluating the results of a planner is that the planner may not have been developed correctly, and may produce an invalid path; a path that in fact causes the robot to collide with some obstacles. In this event, the system will show the planned path and animation, but during the animation if the robot is interfering with any obstacles, a message box will be shown to the user stating that the computed path does in fact graze an obstacle.

Chapter 3 System Code Design Overview

Now that the functional description of the interactive system is complete, the design structure of the underlying code toolkit can be described with less confusion about what the individual components are meant to do. Keep in mind that the interactive system described in section 2.3 is meant to function as both a demonstration of the MPK, and as a debugging tool. The entire set of MPK functionality that has been implemented should be exercised by the interactive system.

Conceptually, the MPK code is divided into several modules that interact heavily with one another.

Module	Function
Universe	Contains information about the robot, and environment.
Collision Detector	Acts as an intermediary between the Planner and the Universe. Responds to collision queries posed by the planner.
Planner	Contains the algorithm for path planning that will be implemented using the collision detector.

Table 1: The Modules Used in the MPK

This division of the system is not arbitrary, it corresponds with the notion that motion planning can be broken down into some sort of sampling of C-space, and performing a search on that space. The planner mechanism effectively places the samples, collision detector acts as an evaluator, testing each sample for collision. This breakdown is consistent with the formulation presented in[1].

A typical use case scenario would involve creating a Universe object, then populating it with robots and obstacles. Typically, only one robot would exist at a time, but nothing

from the universe aspect of the system prevents the existence of multiple robots[1]. A collision detection object would then be instantiated that used this universe, and performed whatever internal optimizations it needed to make collision checking faster. Once a collision detector exists, a planner can subsequently be instantiated that uses that collision detector, takes a start and goal configuration, and produces a path. Figure 18 illustrates the use case scenario outlined above.

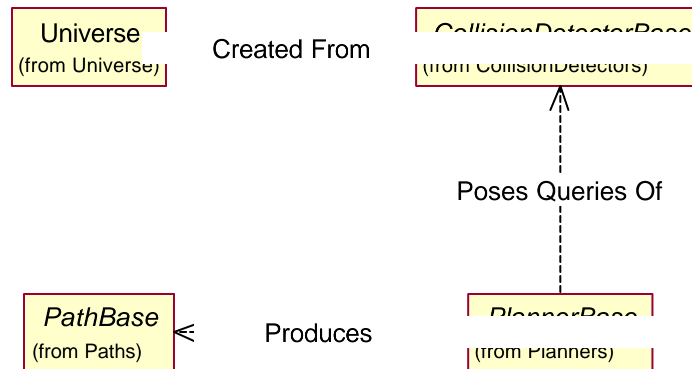

Figure 18: Use Case Scenario for a Typical Planning Task

The planner interface to the environment is provided via a collision detection module. This interface allows the planner to access information about the number of degrees of freedom that the robot permits, as well as information regarding how the robot and the environment interact, generally termed a collision query. Collision queries can be widely different depending on the particular needs of a planner, however they generally are not overly complex. Several queries that I have already included are

- How many degrees of freedom are there?
- Point probe collision query
- Line probe collision query (two types)

With the point probe query, the planner needs to know whether, in a given configuration, the robot is in interference with itself or the environment. Is it in a *collision* state? This

---

[1] Of course if no planners exist for multiple robots, then having them exist is almost useless. Developing these planners requires a lot of additional thought.

type of query is termed a point probe query because it corresponds to testing a single point in C-space. A second type of query is that of the line probe query. Given two configurations of the robot, is the path between them free of collisions? This query type tests a line segment in C-space, hence the terminology describing it as a linear query.

Collision detection is a huge field in itself, and the body of pre-existing work is very large. There is no need for us to re-invent the wheel. The most desirable manner of supporting collision detection is to permit as many "off the shelf" packages to be plugged in as possible, and allow the programmer or the user of the interactive system to choose the one they feel is appropriate. In fact, the MPK can be used to evaluate the utility of different collision detection schemes in much the same way as it evaluates the performance of motion planning algorithms.

The problem with allowing multiple off-the-shelf collision detection code libraries to be used with the MPK is that they usually have different interfaces. Distilling the information from different collision detection modules is a problem in itself, so to simplify the task of the planner developer, a standard method of interface to collision detection libraries was settled upon. Essentially, for each distinct interface to a collision detection library that is possible, an interface class is written. An interface is an abstract class that cannot be instantiated, but exposes several functions that must be implemented in any class that inherits from it. This mechanism serves to isolate the planner from the implementation of the collision detection library. The planner only uses the interface class, thus any class that inherits from that interface could also be used by that planner.

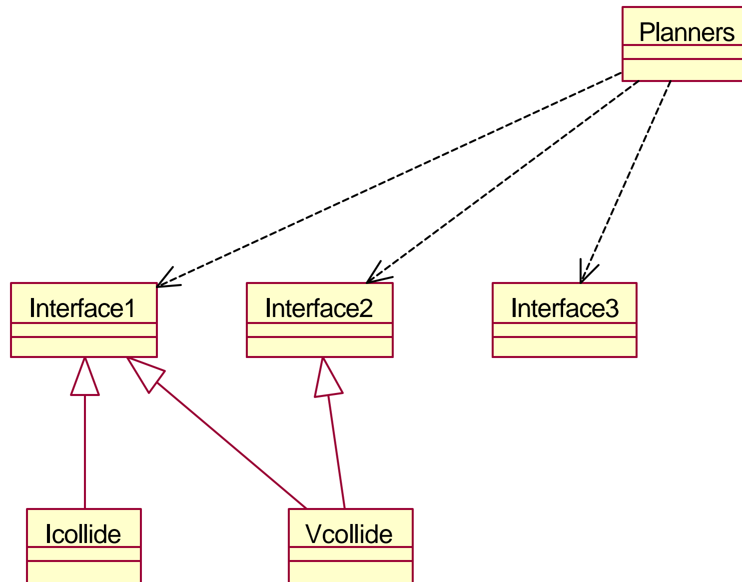

Figure 19: Relationship Between Planners, Interfaces and Collision Detectors

In the diagram shown above, a planner that requires interface1 can use either the Icollide or the Vcollide libraries, while a planner that needs interface2 could only use Vcollide. A planner requiring interface3 would be unable to use either of the collision detection libraries shown. The low-level collision detection objects shown in the diagram represent wrapper classes I have created for each of the third party libraries.

When off the shelf collision detection libraries are identified and added to the MPK, they should be made into classes. Whatever interfaces the library can support should be made parents of the newly created object; i.e. the new class should inherit from them. The functions that the interfaces require must then be written within the wrapper class created for the library, so that it conforms to the interface. Now the new class is ready for use. Authors of planners are isolated from this mechanism because they can write the planner so that it uses one or more of the abstract interfaces. When the planner object is later instantiated, it must be provided a collision detector object, but if the object does not inherit from the required interfaces, the planner object will reject it. Because of multiple inheritance, a collision detection object can potentially support many different interfaces, but the author of the planner only needs to know about the existence of those that are required by that particular planner.

3.1 Basic Types of Mathematical Entities

Before I discuss the operation of the universe, collision detectors and other specific code modules, I begin with an explanation of the basic types and libraries that had to be created as a foundation for the MPK.

3.1.1 3D Vectors

In a system that will make large scale use of computational geometry and manipulations in a 3D scene, one of the most fundamental primitive types is that of the Vector. This class was written from scratch rather than being culled from some other library so that I could maintain complete control and understanding of the workings of this object.

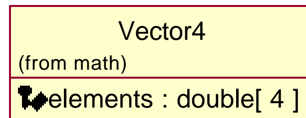

Figure 20: Vector4 Class

The *Vector4* class represents a 3D vector using homogenous coordinates[4]. This representation permits me to represent vectors with infinite length easily, and it fits well with homogenous matrices, which are used for rigid body transforms and will be discussed in section 3.1.3. Because a vector is such a basic type that most, if not all programmers of the system are likely to use, many of the common operations performed on a vector are provided in the form of overloaded (overridden) operators or member functions.

Member function name	Functionality
Operator[]	Used to index the contents of the vector. Only indexes from 0 to 2 are valid, because the non-homogenous values are returned.
Operator+	Vector addition
Operator-	Vector subtraction
Operator*	Scalar multiplication. This operator permits function calls of the form *vector * 5.0* ;
Operator==	Equality operator. Compares two vectors to determine if they are equal.
Magnitude	Returns the length of the vector
MagSquared	Returns the square of the length of the vector. The relatively expensive square root calculation can be avoided in many algorithms by comparing the MagSquared of vectors rather than their magnitudes.
Dot	Dot product of two vectors
Cross	Cross Product of two vectors
Projection	Projection of one vector onto another
Normalize	Returns a vector in the same direction, but with unity length.

I regret the name chosen for this object within the MPK code. Vector4 does not accurately describe the function of this class. In the spirit of writing "self documenting code", in which class names should be very descriptive of the function of that class, this object should be called Vector3H. Where the H indicates the vector is stored using homogenous coordinates.

3.1.2 N Dimensional Vectors

Certain constructs in the MPK system, most importantly points in configuration space, are best represented in terms of an N dimensional vector. C-space may be of arbitrary dimensionality; so it is necessary to permit the structure that contains a point in C-space to contain any number of elements.

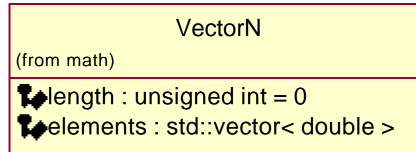

Figure 21: N Dimensional Vector Class

As Figure 21 indicates, the *VectorN* class contains two data members, its length, and a Standard Template Library (STL)[5] structure containing an array of doubles. In STL notation, a vector represents a dynamically resizable array of elements; in this case, doubles. The STL structure does not provide operations on the vector that resemble mathematical functions. These were written separately. Also of note is the fact that unlike *Vector4*, *VectorN* is not specified in homogenous coordinates during any of the operators below. This fact would be obvious had I properly named the Vector4 class Vector3H. The absence of the H in the VectorN class name would serve to indicate that homogenous coordinates were not used.

Member function name	Functionality
Length	Used to determine the current size of the vector
SetLength	Alters the length of the current vector
Operator[]	Used to index the contents of the vector. Only indexes from 0 to Length() - 1 are valid
Operator+	Vector addition
Operator-	Vector subtraction
Operator*	Scalar Multiplication
Operator/	Scalar Division
Operator==	Equality operator – used to determine if two VectorNs are equivalent
Operator!=	Non-Equality operator returns the boolean complement of operator==

3.1.3 4x4 Matrices

In addition to Vectors in 3d, many geometrical operations require the presence of 4x4 homogenous matrices[4]. As such, a 4x4 matrix object is included as part of the MPK.

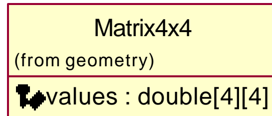

Figure 22: 4x4 Matrix Object

The Matrix4x4 class represents an arbitrary matrix containing 16 elements in 4 rows and 4 columns. It is not restricted to being a rigid body transformation, so care must be taken when assigning values to the matrix if the programmer intends it to maintain a structure that can be interpreted as a rigid body transform. Several access functions are provided to make maintaining rigid body transforms easier.

Member function name	Functionality
Operator*	Matrix x Matrix multiplication
Operator*	Matrix x Vector multiplication
Operator()	Used to index the contents of the vector. Used with two parameters – matrix(0, 2) would index the second element of the zeroth row of the matrix. Keep in mind that bot rows and columns are enumerated starting with 0.
Inverse	Inverts the 4x4 matrix. Inverse can be an expensive operation if called often.
Scale(factor)	Modifies the current matrix by a postmultiplying it by a 4x4 scaling matrix
Translate(vector)	Modifies the current matrix by postmultiplying it by a 4x4 translation matrix that translates by the vector passed as a parameter. This operation maintains a rigid body transform
Rotate(theta, vector)	Modifies the current matrix by postmultiplying it by a 4x4 translation matrix composed by rotating theta degrees around the vector passed as a parameter. This operation maintains a rigid body transform.

3.2 Universe

The universe object is the object that the system must first populate before any other tasks can be accomplished. It will typically contain one or more robots and the obstacles that represent the environment.

In order for the universe to store both robots and obstacles, these two types of object are derived from the same abstract base class, entity.

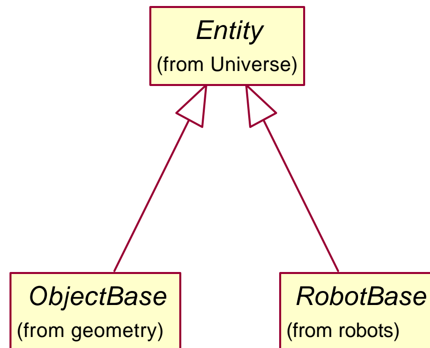

Figure 23: Both Robots and Obstacles Derive From Class Entity

This being said, there is no real distinction, as far as the universe is concerned, between a robot and an obstacle. All items that can be contained in the universe are referred to as *entities*, and are stored in the same list structure.

Figure 24: A Universe Contains Many Entities

The entity class exposes the following functions

Member function name	Functionality
Clone	Correctly duplicates the entity that is pointed to by a pointer in a polymorphic manner.
IsInterfering	Determines if two entities are interfering with one another
SetBaseFrame	Sets the frame that this entity is defined with respect to
SetFrameManager	Sets the frame manager that this entity should be using to

	access frames.
GetTransform	Gets that transform of the frame that this entity is defined in relative to the base frame.
CanCheckInterference	Determines whether or not an interference check is allowed between two entities.
BaseFrame	Determines the base frame that this entity is defined with respect to

Because the universe contains a list of pointers to entities, any object that inherits from this class can exist within the universe without adverse consequences. Later, it will be demonstrated that constructs such as open chain robots as well as polyhedral meshes and other objects derive from entities.

3.3 Universe - Kinematics Module

The universe must also contain a kinematic description of the entities contained within. This description is necessary so that users can specify the state of the universe using joint variable notation.

3.3.1 Frames

The kinematic description revolves around the concept of frames and links. A frame is defined as would be expected in a robotic simulation program; it represents a 4x4 rigid body transformation matrix.

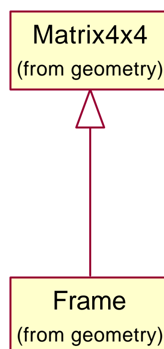

Figure 25: Frame Objects Derive From 4x4 Matrices

Because it derives from a generic 4x4 matrix, the frame object has many member functions overridden. Operations such as matrix multiplication, and inversion already exist for this class.

In addition to the concept that a frame represents a transformation matrix, is the notion that a frame is always defined relative to some other frame, its base frame. In order to implement this concept, each frame contains an additional field representing its base frame.

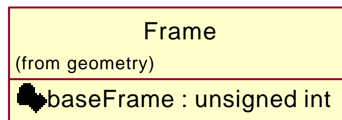

Figure 26: Frame Class Illustrating *baseframe* Attribute

The *baseframe* reference is defined to be an unsigned integer rather than an explicit pointer to another frame structure. Essentially the way I wrote this amounts to rewriting of pointer mechanism of C++. I implemented it that way for several reasons. First, it can be difficult to maintain a tree structure using pointer memory management. Second, an additional structure the *Frame Manager;* to be discussed subsequently, exists that contains all the frames in the system and I wanted this structure to handle all the memory management for the frames. If pointers were maintained to these structures, copying frames while maintaining memory integrity becomes difficult. It also becomes complicated to index specific frames in the tree without actually traversing the tree every time a frame is required. The *baseFrame* number is an index into this frame manager. Thirdly, using an unsigned integer to represent the number of the frame makes more intuitive sense to a programmer using the system. Debugging is easier if the base frame of frame 2 is listed as '1' rather than '0x00004FEC'.

3.3.2 Links

Robotic kinematics necessitates the concept of links. In the MPK software system, these are objects that encapsulate a mathematical function describing how a frame is altered by changing the value of a joint variable. Usually links are described using Denavit – Hartenberg notation, although in the future they may not always be.

A link, like a frame, must be defined in relation to some pre-existing frame. It also must control the motion of one or more frames itself. Since it encapsulates a function that maps a joint variable q to a frame, it also has a range over which the input variable is valid, typically referred to as the joint limits of a particular joint. A special case exists for some classes of joint, particularly revolute joints, wherein no joint limits exist. These joints can rotate infinitely in one or more directions. To accommodate this case, an additional flag must be present to indicate that the joint variables "wrap around".

In the MPK system, a link is used to completely describe the structure of one link of a robot. It contains a kinematic description as well as the geometry of that link.

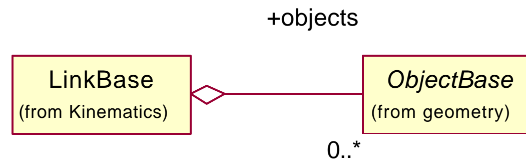

Figure 27: Link Object Contains Geometry Objects[2]

As described, a link object completely encapsulates the motion of one or more frames, corresponding to one or more joint variables. Each link object contains member functions for setting joint variables. Once the joint variable has been set, the frames that depend on that joint variable can be updated to reflect this new value by calling the *UpdateFrames()* method.

Member function name	Functionality
SetBaseFrame(unsigned int)	Specify the frame that is the parent for this link. All links must be defined within a pre-existing frame.
UpdateFrames()	All the frames that this link controls are updated to reflect new joint values
BaseFrameNum()	Find out the frame that is the parent of this link
Clone()	Duplicates the link based on a pointer to that link

[2] The 0..* indicates that the *LinkBase* can contain from 0 to infinite numbers of objects

SetFrameManager()	Specifies the frame manager that handles memory management for the frames this link controls.
SetJointVariable(double)	Alters the joint variable that drives this link
JointMax(double) JointMin(double)	Set the max and min values for the joint variable.
JointWraps(bool)	Specifies whether or not the joint can wrap around its joint limits. For example, a revolute joint that can spin an infinite number of times.
DoesLinkControlFrame(int)	Determines whether or not a link controls the motion of a specific frame
Serialize()	Allows the data in the structure to be written to file or to a stream.
DeSerialize()	Reads in the data for the structure from a file
DeSerializeAbstract()	Reads in the data for the link structure correctly even if the data is stored for one of the derived classes of LinkBase

3.3.3 Links Defined Using Denavit Hartenberg Parameters

The class structure for links is meant only as an abstract base class that other classes will inherit from. An abstract link is neither intended to be instantiated, nor is instantiation possible. In order to use a link, one must choose a particular variety of link, and instantiate that instead. Since Denavit Hartenberg notation is very common, one derivative of the link base class is that of a Denavit Hartenberg (DH) link.

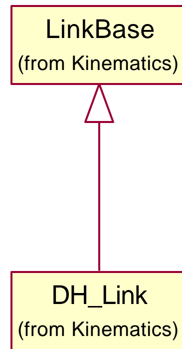

Figure 28: DH Link Object Inherits From Link Base Class

In C++, and most other object oriented languages, if you want to store a collection of objects of mixed type, they must all derive from one common base type. In the MPK, a robot is made up of a collection of links, which can potentially be of several different types. The reason that a DH_Link must inherit from *LinkBase* is so that all links in a robot can be stored in one simple structure. *LinkBase* is the common base type for all link classes.

The DH Link structure has additional data members that are specific to DH notation. Floating point values representing the A, D, θ and α parameters are required. In addition to these numeric parameters for DH links, one of the parameters is controlled by the joint variable. In order to make this control as transparent as possible, yet at the same time efficient, this is accomplished by including a *Q* data member that is a pointer to one of the floating point DH parameters. Thus when the user changes the joint variable, they change the value pointed to by the *Q* pointer. The concept of the Q pointer allows one DH link structure to represent prismatic and revolute joints without loss of generality.

In actual fact, a programmer using the system should be completely unaware of the existence of the *Q* pointer. It will be hidden by the access functions, particularly *SetControllingParameter()* and *SetJointVariable()*. *SetControllingParameter()* is used to specify which of the DH parameters is controlled by the joint variable. The *SetJointVariable()* function performs the task of updating the joint variable that controls

this link. Specifying a DH link that uses the theta parameter to control its motion would look something like:

```
DH_Link link ;
Link.SetControllingParameter( DH_THETA ) ;
Link.SetJointVariable( 30.0 ) ;
```

Adjusting the joint variables is not immediately reflected in an adjustment of the frame that this link controls. Modification of the frame values is triggered only when the *UpdateFrames()* method is called. This member function allows the programmer to specify precisely when the update operation will be performed. In this case it seems like a bit of extra work for the programmer, the *UpdateFrames()* function is fast, and might as well be called every time a joint variable changes. However, for different link types, the function may be significantly more computationally expensive, and in such cases, it is better to leave the timing of when the function will be called in the hands of the programmer.

A Denavit Hartenberg link controls the position of one frame in the frame manager. The formula used to determine the position of the frame is included as part of the Appendix.

The *DH_Link* class contains many member functions, most of which are directly inherited from the base class *LinkBase*. Many of these inherited functions are overriden in the derived class, and must be implemented in a manner specific to the DH notation. The functionality, however, is the same as that indicated in the base class. Additional member functions are outlined below.

Member function name	Functionality
SetAlpha(double)	Specify the Denavit Hartenberg parameters for the joint
SetA(double)	
SetD(double)	
SetTheta(double)	
SetControllingParameter()	Specify which of the parameters is connected to the joint value.
SetJointVariable	Alter the parameter that is connected to the joint variable
FrameNum()	Since a DH link can only control one frame, it is easier to call this function than the *DoesLinkControlFrame()* function of the parent class.
AddObject()	Adds geometry to the frame that this link controls.

3.3.4 Frame Manager

Both the Frame class and the Link class above depend on the presence of a frame manager object. This object exists as a part of a universe to keep track of a hierarchy of frames. Each universe and each collision detection module will have one frame manager apiece. The frame manager also handles memory allocation for any frames that are referenced in one particular universe. It does this allocation by storing all the frames in a dynamic array.

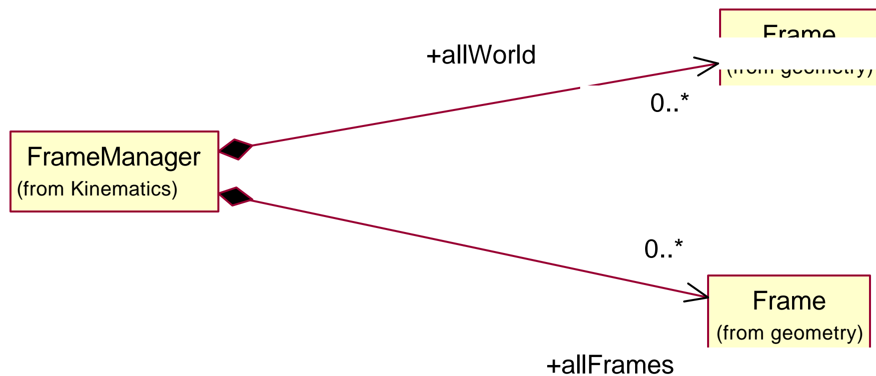

Figure 29: Frame Manager Stores all the Frames

The *allFrames* member shown in Figure 29 represents the storage space for each individual transformation frame. Each of the frames stored in *allFrames* depends on a link in order to be updated, and represents a transformation relative to the base frame of that particular frame.

In order to speed up certain computations, an additional array of frames is maintained by the *frameManager*. The *allWorld* array represents transformations relative to the world frame, frame 0. The primary reason this array is maintained is that matrix multiplication is an expensive operation if it needs to be done many times. One common operation is computing the transformation matrix relating two frames. During this process, the transformations between each of the frames in question and a common base frame are computed. Because this function is called for in a variety of situations, often without a change in joint variables, it would be beneficial to retain some of the intermediate variables that were computed, which is exactly what maintaining the allWorld array does.

Methods that the frame manager supports were designed so that the programmer using the MPK can access all the necessary information about a given frame easily. The methods include those to add additional frames, determine the contents of a frame, determine the relative transformation matrix relating two frames, etc.

Member function name	Functionality
AddFrame()	Creates a new frame, allocates memory for it, and returns an integer indicating its frame number. Similar like malloc() or new() in C/C++
GetNumberOfFrames()	Returns the total number of frames controlled by this frame manager.
Operator[int]	Returns a specific frame.
ValidateFrame(int)	Checks the parent frame of the specified frame to make sure there are no circular references. Frames that are not defined relative to the base frame, frame 0
GetFrame(int)	Returns a copy of the frame indicated

GetFrameReference(int)	Returns a pointer to the frame indicated
GetTransformRelative(int, int)	Returns the transformation matrix relating two frames.
BaseFrame(int)	Returns the base frame of the frame in question.
SetBaseFrame(int, int)	Alters the base frame of the frame in question.
MarkFrameChanged(int)	Mark a frame as having changed, so that all the cached frames that depend on it will be recomputed.

3.4 Geometry Module

In order to maintain a description of the robot and the environment, geometrical data structures and functions are included in the MPK. Some of the objects in this module derive from *entity* and as such can be added to the universe as obstacles or added to a link as link geometry. Other objects, like *VRMLreader* are access objects used to access files and load geometry from this source.

3.4.1 ObjectBase

All the geometrical objects that exist within the confines of the MPK derive from a common base class, that of *ObjectBase*. This class in turn inherits from *Entity*.

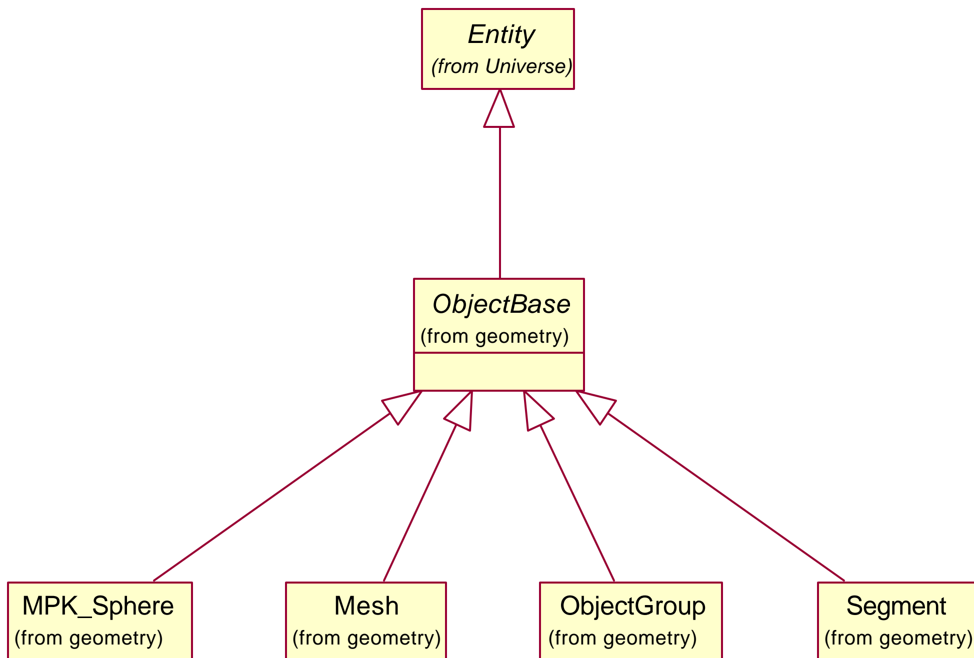

Figure 30: All Geometrical Objects Derive from ObjectBase

The *ObjectBase* class enforces several operations on all the geometric objects that derive from it. In addition, it ensures that all objects will contain a *Frame* data member. This frame represents the offset of the geometric object in the frame in which it resides. The purpose of this additional frame is that most geometry will not be defined so that its origin matches the origin of a frame that a link controls. This will be especially true of geometry loaded from a preexisting file. For example, Denavit Hartenberg notation specifies that the axis of joint motion coincides with the Z axis of the frame. However, what if the geometry file has interchanged the Z and the X axis? This "offset frame" can be used to correct this problem.

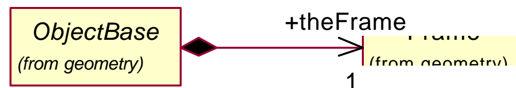

Figure 31: Each Geometric Object Contains an Offset Frame

During all computations involving a geometry object, the offset frame must be specifically taken into account. The offset frame is taken into account by default in all

computations that are provided in the MPK, but future extensions must keep this property in mind.

The fact that the *ObjectBase* class inherits from the *Entity* class means that it also inherits properties of the *Entity* class and abstract functions that must be written. Some of these inherited properties and functions bear further explanation. The *Clone* member function is extremely vital to the operation of the MPK as a system. It makes dynamic object polymorphism possible; enabling us to store lists of different objects, all derived from the same base class without memory allocation problems. The *Clone* function is a virtual function. What this means is that when you have an object inheritance hierarchy, and you have a pointer to an object that resides at some level of the hierarchy, there will be no ambiguity when calling member functions of this object. Virtual functions in C++ provide a built in mechanism for Run Time Type Information (RTTI). What occurs in this instance is that programmers often have collections of pointers to objects. This collection may be a linked list, or array of pointers to objects in an inheritance tree. The programmer does not know exactly what the type of the pointer may be. For discussion, let's say that the pointer could point to an object of the grandparent class, the parent class or the child class. If the grandparent class contains a certain member function, then so do the classes that derive from it, so this function can be called on any of the objects in the inheritance tree. However, if that function has been overridden in either of the child classes, when the function is called, you will get unexpected results, because the function that is executed will be the one appropriate for the grandparent, not the parent or the child.

Virtual functions allow function ambiguity to be resolved at runtime. An additional data member indicates what the actual type of the pointer is, and the correct member functions are called, without the programmer having to do any additional work. The drawback to writing code in this way is that every member function call has the additional overhead of one pointer dereferencing operation. Usually this extra overhead is a small price to pay for the software engineering power you receive in return.

The *Clone* function provides a polymorphic way of duplicating a pointer to an object. It will be used in occasions where you have a pointer to an object that derives from *Entity*, but you are unsure of the precise type of the object. There are ways of determining the type, but it is not necessary to do so in this instance. If you had a pointer to an object of known type and wanted to duplicate it, it would be a simple matter of writing

```
Entity* newEntity = new Entity( oldEntity ) ;
```

However, we don't know the type of the pointer in here, so we cannot use the above syntax. Instead, we write:

```
Entity* newEntity = oldEntity->Clone() ;
```

This function performs the task of memory allocation, invoking the C++ operator *new* on its own data type. Since the function is virtual, we can be assured that the correct amount of memory is allocated, and the object's data is copied correctly. Since memory has been allocated dynamically, care must be taken to ensure it is deleted properly. Freeing the memory is the responsibility of the code that invokes the *Clone* function. *Clone* must be used in much the same way as C++/C functions *new* or *malloc* would be used in C++/C. The memory must be deleted when no longer required.

Additional member functions defined abstractly at the *ObjectBase* level but implemented by each of the derived classes are the *CanCheckInterference* and *IsInterfering* member function. *CanCheckInterference* is used in error trapping to determine if collision detection between two geometrical objects can be performed. Object vs. Object collision detection is implemented via member function of the individual objects. If a programmer has a polyhedral mesh object and a sphere object and wishes to determine if they intersect, a separate collision detection object does not first have to be created. The member function:

```
objectA->IsInterfering( objectB ) ;
```

can be called. However, this function must be polymorphic, determining the type of objectB and calling the correct code at runtime. Unfortunately, as the MPK system grows, not all combinations of objects will have interference code that is capable of

determining whether or not the pair intersects. To determine whether or not an intersection test exists for a pair of objects, the additional member function

```
objectA->CanCheckInterference( objectB ) ;
```

should be called.

Member function name	Functionality
IsInterfering	Used to perform a one on one interference check with another object.
GetFrame	Returns the offset frame associated with this object
SetFrame	Sets the offset frame associated with this object

The current version of the MPK supports the following object types

1. Polyhedral meshes
2. Spheres
3. Line segments
4. Groups of objects

3.4.2 Polyhedral Meshes

Polyhedral meshes are the most important geometrical data type in the MPK system. Most robots that are simulated in other systems exist in a polyhedral mesh format. Since one of the underlying tenets of the MPK system is that researchers be able to port their existing problems to our system easily, polyhedral meshes were included as a primary data type. Other reasons for including polyhedral meshes are their flexibility, and general ease of use. Most preexisting collision detection libraries presume the polyhedral meshes to be the *only* data type that is being used.

A mesh is a surface representation of a polyhedral object. It is a set of connected polygonal faces that form a surface. A mesh is a surface model only, however. It does not contain inside/outside information about the object that it defines, only information about the surface. It also does not have to be a closed object. A single triangle is a valid,

albeit very simple, mesh. This mesh has no interior or exterior, simply a description of a surface in three space.

Like all geometric objects in the system, polyhedral meshes derive from the *ObjectBase* class.

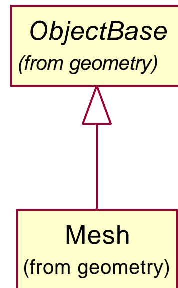

Figure 32: Mesh Object Inherits From Object Base

In addition to the basic data members provided through inheritance, the mesh also contains several other data members.

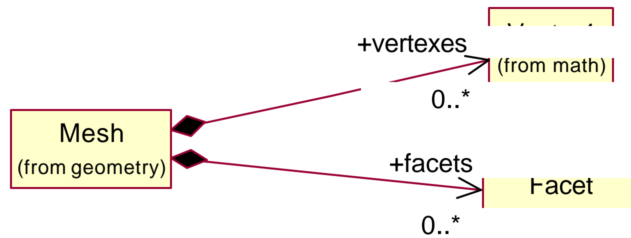

Figure 33: Mesh Objects Contain Vertex Points, and Facet Lists

Each *Mesh* object maintains an array of vertex points. These represent the positions in 3D space of the vertexes of the mesh. By itself, verticies only provide enough information to describe the convex hull of the mesh. We also need data representing the manner in which the verticies are connected into facets. This data is stored in a separate array of *Facet* objects.

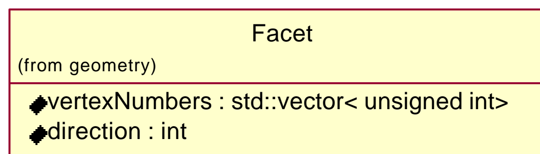

Figure 34: Facet Object

The facet object contains an array of integer values representing vertex numbers, and a direction flag indicating whether the vertexes are intended to be enumerated in clockwise or counterclockwise order. The vertex numbers are meant to represent indexes into the vertex array.

The data representation I have chosen for the Mesh object was not selected arbitrarily. It is a format widely used to store information about mesh type geometry. It shares a high degree of similarity with the format used to store mesh geometry in the Virtual Reality Markup Language 1.0 (VRML 1.0) file format. It should be stressed, however, that a data structure and a file format are very different concepts not to be confused with one another. Just because this representation is similar in description to VRML, it is not “using VRML” or “tied in to VRML”.

The structure of the mesh object should make it clear that although polyhedral solids can be stored in a mesh format it will be as a surface model only. There are no checks in place for the validity of the polyhedral object. It can self intersect, it can be closed or unclosed, etc. Polyhedral objects are difficult to deal with at the best of times. They are usually not generated by hand and when generated by third party tools, they are usually error free.

Polyhedral meshes can be used to represent a closed region of space. If a mesh is to be used in this instance, there may exist a need to solve the point containment problem, the problem of deciding whether or not a given point is inside or outside a mesh. In this case, there are algorithms that can be applied that attempt to solve this problem, provided that the mesh obeys certain constraints. The mesh must be completely closed, and the orientation of vertexes on each of the faces must adhere to a consistent organizational scheme. If a need is discovered to solve the point containment problem, the next version of the MPK will include meshes that represent solid objects in this manner.

Member functions of the *Mesh* class exist to support constructing a mesh from the ground up. *AddVertex* and *AddFacet* can be used to incrementally construct the mesh. As stated above, there are no checks in place to verify the validity of a mesh, thus vertexes and facets can be added in arbitrary order. Ease of construction of mesh objects is a distinct advantage as compared to using proper solid models in which incremental construction is significantly more difficult, often requiring the use of difficult Euler operators. Euler operators allow solid models to be constructed; however their use is extremely non-intuitive to most users. The order of adding vertexes and faces is extremely vital when using Euler operators, in stark contrast to the Mesh representation we are using.

Another *Mesh* object member function is the *TransformVerticies* method. This function alters the mesh by multiplying the position of each point by a 4x4 transformation matrix, enabling the user to rotate, scale and translate the *Mesh* without altering the frame in which it resides. *TransformVerticies* is an expensive operation because it must be applied to each of the vertexes in the *Mesh*, but it can save significant time if it allows us to avoid matrix multiplications later, during collision detection.

As mentioned above, each geometric object in the MPK system is responsible for being able to detect interferences between itself and other geometrical objects, meaning that for every pair of objects for which an intersection test exists, one of the pair must contain the member function for that interference test. However, only one of the objects needs to have the routine, not both. Also, if an interference test does not exist, the program should fail gracefully. In order to accomplish this error trapping, the member function *entitya.CanCheckInterference(entityb)* exists. This function takes as a parameter a pointer to a second entity. If a collision routine exists for this pair of objects, then the function returns true, otherwise it returns false. This routine can be used as a method of error trapping while debugging programs.

For polyhedral meshes, at present, the only interference tests that exist are between one mesh, and another mesh. Interference tests with spheres, line segments, etc. are not permitted at this time. These will be added in subsequent upgrades of the toolkit. These

tests are missing because at present, to perform the low level interference checking for meshes, the Vcollide library is employed and it does not support anything but mesh-mesh tests.

One of the last member functions present in the polyhedral mesh class is the *Splice()* function. This function merges two meshes together. It is a very rudimentary algorithm that simply appends the vertex lists, and adjusts the facet enumeration numbers of the meshes accordingly, but it will speed up collision detection routines if used. If an object is represented by several meshes, its collision detection will be faster if those meshes are first spliced together (a onetime cost) then sent to the collision detection routines.

Member function name	Functionality
AddVertex	Adds a vertex to a mesh
AddFacet	Adds a facet to a mesh
ReadFromIcollideFile	Deserializes a mesh from an Icollide file
SetCoordinates	Alters the vertex list to match a new list passed as a parameter.
TransformVerticies	Transforms all the vertices by the matrix passed as a parameter. Can be used to rotate, scale, or translate the geometry.
Splice	Joins two meshes together to form one single structure. Joining is accomplished simply by appending vertices and facet information.

3.4.3 Spheres

The second primitive resident in the MPK geometry module is that of the sphere. Denoted *MPK_Sphere* due to naming conflicts with the *Vcollide* collision detection library, the sphere is the most efficient geometry representation in the MPK system. Sphere – Sphere collision detection tests, for example, are the fastest interference tests; faster even than tests of axis aligned bounding boxes like those used by Vcollide.

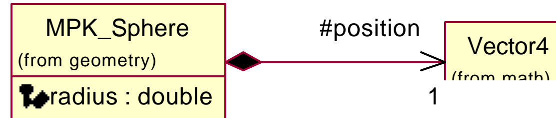

Figure 35: Sphere Object Represents a Sphere with Position and Radius

The sphere object is represented using a Vector for the position of the sphere, and a floating point value for the radius. The inclusion of the position vector as part of the sphere is not entirely necessary, given that each entity in the MPK system exists inside a moving frame controlled by a robot's joint, and all entities within one of these frames have an "offset" frame. The position vector could simply be included in this offset frame. Spheres however do not require all the information about the offset frame to describe their position. Only the translation portion is valid because spheres are rotationally invariant. To speed up interference testing, alterations of the sphere's offset frame are directly translated into alterations in the position vector, and the offset frame always remains the identity matrix, allowing the matrix/vector multiplication that moves the sphere to be performed once, rather than each time an interference test is performed.

Little more need be said of the sphere. Its member functions are exceedingly simple. Accessor methods for getting and setting radius values and positions, along with the requisite interference checking routines as described in section 3.4.2. The only collision detection routine contained within the sphere object is the sphere vs. sphere intersection test. All other intersection tests involving spheres are owned by different objects.

Member function name	Functionality
Radius	Determine what the radius of the sphere is
Position	Determine the position of the center of the sphere
SetRadius	Set the radius
SetPosition	Set the position of the sphere

3.4.4 Line Segment

The line segment primitive was added to permit the construction of extremely simple robotic links. That being a link composed only of a line segment joining its endpoints. The line segment class is very simple, as Figure 36 illustrates

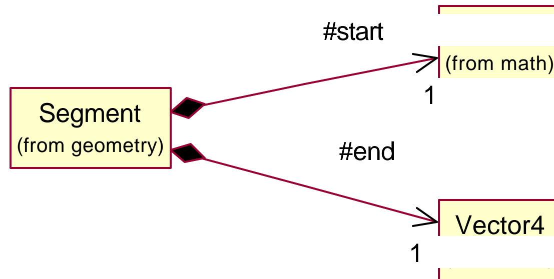

Figure 36: Line Segment Object

The segment object derives from the entity class, and contains two vectors. One represents the start of the line segment, the other the end. This representation is slightly misleading because a line segment is not a directed entity. Start and end have no meaning other than to distinguish the two points on the segment.

Like the sphere object, the line segment's member functions are very simple. Only accessor functions and interference checking functions have been written for the segment object. This time, however, things get a little more complex. In addition to the segment vs. segment interference test, this class contains a segment vs. sphere interference test.

It is important to note that there are numerical issues surrounding the segment class. An ideal line segment has zero width. In our system, a line segment is a segment connecting two points in 3D space. Since the endpoints of the segment are used to define it, any modifications of the end points also modify the line. Problems arise during interference testing. Two lines intersect one another only if all four endpoints lie in the same plane. This concept is simple, but when the coordinates of the endpoints are represented with a finite precision on computer, 4 point coplanarity almost never occurs, meaning that the interference test between two segments almost never returns a true value indicating interference. This problem can be solved to some extent by utilizing a numeric constant ε

for mathematical calculations where any number with magnitude less than ε is treated as zero. The ε solution would permit segments that very nearly intersect to be treated as intersecting. This capability is not present in the MPK, but can be added in subsequent versions.

Another note when using segments is that many motion planning algorithms discretely sample C-space. Adjacent samples are tested for interferences, and if none are found, it is assumed that the path between the two is also collision free. This assumption falls apart when segments are used. For example, two skew segments are separated by distance d. Let's make d equivalent to the joint variable being modified by the planner. The planner wants to test if the path from d = -1 to d = 1 is collision free. Suppose, we know that this path is *not* free, and collision occurs when d=0. However if the planner discretizes the path into 4 points, d=[-1.0, -0.33, 0.33, 1.0], and tests each of them, no interferences are will be found. In fact, unless the planner happens to perform a discretization in which d=0 is one of the points tested, it will never encounter an interference. To some extent, the ε solution presented above will also solve this problem.

For the two reasons mentioned above, segment vs. segment interference tests in 3D are unreliable, and should not be expected to produce good interference results. Line segments should not be used to construct non planar robots because links constructed of segments would not produce interferences with other links constructed out of segments that were not in the same plane.

Interference tests in which the two segments are in the same plane are a different story. In this case, the interference test will produce valid interferences, and the sampling problem is eliminated in all cases except when the segments are precisely parallel meaning that if the robot being constructed is planar, segments are still a good choice to represent the links.

Although segment vs. segment interference tests in 3D are unreliable, that does not mean that line segments should not be used in a 3d robot description. Interference tests

between segments and other primitives do not share the problems described above. For example, the segment vs. sphere calculation that resides as a member of the segment object is numerically robust. It produces correct results in all cases with no instabilities or problems of any sort.

Member function name	Functionality
SetStart SetEnd	Specify the start and the end points of the line segment

3.4.5 Groups

One convenient feature of many 3D scene descriptions is the concept of grouped objects. This function is often found in scene description languages such as VRML, and it allows different primitives to be grouped together and treated as a single entity. This operation corresponds to the constructive solid geometry (CSG) union operation.

In keeping with this tradition, an *ObjectGroup* object is included in the MPK system. This class, like all the primitives, derives directly from the *ObjectBase* class; however, a group object has another relationship with *ObjectBase*. Groups also *contain* an array of *ObjectBase* pointers, providing the group object with the means to amalgamate several primitives together. In fact, because the group contains an array of *ObjectBase* pointers, it is a polymorphic group. Making a group polymorphic means that the group can contain any of the primitives that derive from *ObjectBase*, even other groups!

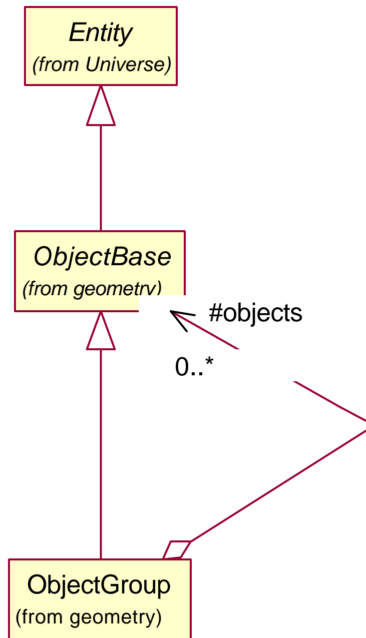

Figure 37: A Group Both Contains and Derives From *ObjectBase*

In order to access the group object properly, methods must exist that allow the insertion and deletion of objects from the group. These methods are in addition to those that operate on the group in a manner similar to normal objects. In fact, simple operations on groups were remarkably easy to program. Interference tests between a group and another object are simply a logical OR of interference tests between the objects in the group and the object.

Because the interference checks for groups and other objects are simply remappings of the interference tests to the objects contained in a group, interference tests between groups can be performed without worry on any type of object. The test will always exist.

Member function name	Functionality
AddObject	Adds an object to the group
Operator[]	Gets an object stored in the group
Size	The number of objects in the group

3.4.6 Future Improvements

The set of objects that is currently included makes the MPK an extremely flexible system. Since polyhedral meshes are included as canonical objects, just about any robot can be approximated. Most 3D cad packages support outputting geometry in mesh format, so it can be used with the MPK package.

Some additional geometry objects would be nice to include for the sake of speeding up collision detection. Most notably, polyhedral solids would be a good addition to the MPK. These objects are similar to meshes, however a distinct inside and outside exists, enabling collision detection to bound both the interior and exterior of the solid, speeding up many calculations.

Also, to combat the problems associated with constructing 3D robots from line segments, it would be nice to include cylinders and rectangular solids in the MPK. Cylinders are a computationally cheap method of describing a robot's link. Intersecting a cylinder with other geometries is much cheaper than a polyhedral mesh describing the same surface. Along the same vein, rectangular solids are even cheaper to perform collision tests with. These solids can be represented with meshes, but their symmetry makes computation cheaper if treated as separate entities. Both these entities make sense in terms of the interactive MPK system. If users are designing robots online, they need access to simple geometric entities out of which robots are easily built. Rectangular solids and cylinders certainly fit this bill.

3.5 Collision Detection

Collision detection is an area that has been touched upon briefly in preceding sections. To reiterate, a collision detection object is an object that takes the universe and its current state, including robot descriptions, obstacles, etc. and structures it so that collision detection queries are fast and optimized. A collision detection query represents the motion planning algorithm's interface to the robot and the environment. For the MPK system, the types of motion planning algorithms that are being developed are very general ones. Ideally these planners should know very little about the structure of the

robot and the environment. That being said, the link between the planner and the robot/environment should be a very narrow one – the collision detection module forms this link.

Collision detectors that are currently integrated into the system include

1. A homegrown, simple collision detector
2. Vcollide
3. Solid

Future iterations will include many others including one developed by Michael Greenspan of NRC in [11].

3.5.1 Simple Collision Detection

Simple collision detection is what I use to describe the brute force collision detection scheme that I wrote into the MPK myself. It takes every object in the universe and tests it against every other object in the universe using the built in interference testing routines that each object contains as data members. Because it uses the built in collision detection routines and performs no optimizations, this collision detecting object is very robust. It can handle any object vs. object interferences that I've written code to handle. This amount of robustness is not present in some of the more sophisticated collision detection schemes.

Simple collision detection is the only scheme that, at present, can handle interference tests for robots that contain line segment or sphere entities; some of the fastest entities in the system. Its performance on polyhedral meshes leave something to be desired however. The core of the mesh vs. mesh collision detection routines is the Vcollide library.

3.5.2 Vcollide

The University of North Carolina provides the Vcollide library in the public domain through their computer graphics department [9]. It is built on top of the RAPID library for collision detection, and makes improvements on it using a sweep and prune algorithm.

Vcollide handles only polyhedral meshes. It does not require nor use solid objects or any geometry besides meshes. Practical limitations on robustness aside, Vcollide appears to be an extremely efficient collision detection scheme. As a rough benchmark for the speed of the algorithm, the three link robot case to be discussed later in which all the links are polyhedra, it took on average 0.2 ms for each collision detection query, corresponding to about 5,000 collision detection queries per second. This number is a rough estimate only. The actual rate of collision detection queries depends on the complexity and shape of the underlying polyhedrons.

Unfortunately, Vcollide provides only information about whether or not a collision has occurred. Getting information regarding the how far apart two entities are and other complex functionality is not supported. The linear interface is also not directly supported by the Vcollide library; however, I have added default linear support to the interface to make this possible. Two different versions of the linear collision detection interface are provided. Both of these inherit from the point collision detection interface. Each of the linear interfaces supports testing a path between two configurations, one discretizes this path into preset number of sub points, and tests each of these, assuming that the entire path is collision free if none of these points contain a collision. Forcing a set number of sub points is a very computationally expensive method of testing a path if the configurations being considered are close together.

A second scheme discretizes the path into sub points just like the last one, but to avoid putting in too many sub points, a limiting criteria is enforced. Each dimension in C space is given a minimum discretization limit. This limit indicates that a path in C-space will be sampled with points no further apart than a certain distance along that dimension. When generating the sub points to sample, these limits are kept in mind. The distance between the start and the goal configuration in each dimension is measured, and divided by the maximum sampling distance in that dimension. Whichever dimension mandates the most sampling points is the one used to “drive” the path. The number of sampling

points required by this dimension is used in discretizing the path ensuring that all dimensions have satisfied the maximum sampling width.

3.5.3 Solid

The solid collision detection library is another third party collision detection library that has been incorporated into the MPK. It was developed by Gino van den Bergen, and placed into the public domain via the GNU free software license [10]. Solid supports geometry composed of any of several basic primitives, polygon meshes, cones, boxes, cylinders and spheres. It uses the Gilbert-Johnson-Keerthi algorithm to accelerate collision queries.

Solid supports collision queries that provide information on whether or not collisions occurred, which primitives caused the collisions, closest point information and approximate angle of collision information. The entire set of solid functionality has not yet been included in the MPK, only enough to place it on par with Vcollide has been implemented.

3.6 Planners

The MPK system will eventually be used to evaluate the performance of a wide array of motion planning algorithms. Programmers will develop their own algorithms within the MPK framework and add them to the system. However, the interactive MPK must have several "baseline" algorithms for users to test out on the web. As such I have implemented certain algorithms, often with the help of other developers. Assistance is noted where significant.

Also of note is the concept of global vs. local planning as well as complete vs. incomplete planning. The difference between complete and incomplete planning is straightforward; a complete planner is a planner that will find a path to the goal if one exists, whereas an incomplete planner is not guaranteed to do so in all cases. On the other hand, global vs. local planning is more of a spectrum. A global planner is designed to plan the path of a robot completely from start configuration to goal configuration. It is very powerful, and ideally should find a path to the goal if such a path exists, albeit sometimes taking a long

time to complete its task. On the other hand, a local planner is a planner that is not nearly as powerful. It is meant to be extremely fast, rather than complete. In many situations, a global planner will use a local planner as a tool to plan intermediate paths while it searches for the goal.

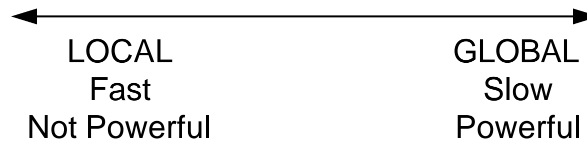

Figure 38: Difference Between a Global and a Local Planner

The difference between a global and a local planner is not a yes or no difference, it is a spectrum onto which a planner will fall. Some planners are very fast and tend to have more of a local flavor, while others will be slower, but more powerful.

3.6.1 General Structure of a Planner

The code structure of a motion planning algorithm is consistent with the object oriented nature of the MPK. A planner is an object that derives from the *PlannerBase* abstract base class. This parent class forces each planner object to implement several member functions.

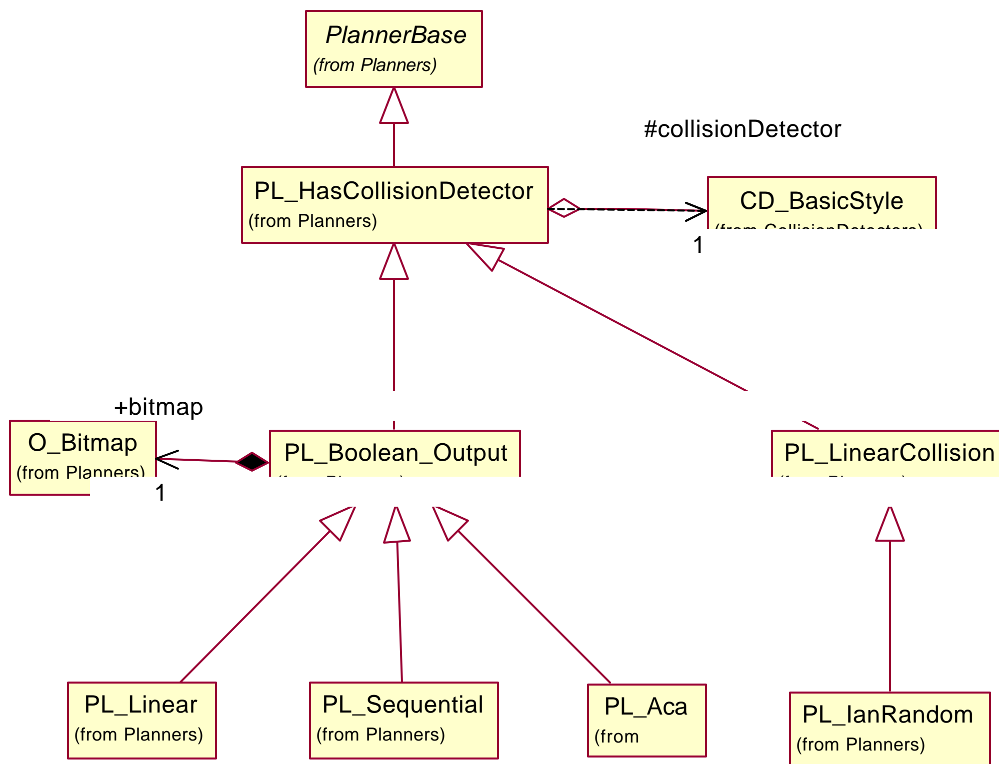

Figure 39: Class Hierarchy of Planner Objects

As Figure 39 above attempts to illustrate, everything in the planner hierarchy inherits from the abstract base class *PlannerBase*. This inheritance permits anything in the hierarchy to be stored in polymorphic storage structures such as arrays of pointers, etc. *PlannerBase* forces all the planners beneath it in the hierarchy to expose certain member functions, which are implemented as pure abstract functions in the *PlannerBase* class. Included in these functions are some that specify the start and goal configurations for the path planner to use. Also, a *Plan()* function to tell the planner to begin calculation, as well as a mechanism for retrieving the path, *GetPath()* are included. Getting the path presents some problems in itself because there may be more than one type of path in the MPK at some point. What this function returns is a pointer to a *PathBase* object that may actually point to a class that is a child of *PathBase*. It is the responsibility of the calling code to sort out what the actual type is and treat the path appropriately.

In truth, all the planners inherit from a class slightly further down the tree from *PlannerBase*, *PL_HasCollisionDetector*. This class is the same as the *PlannerBase* class, except that it maintains a pointer to a collision detector object.

Beneath this class, the structure of the hierarchy diverges slightly. The classes *PL_LinearCollision* and *PL_BooleanOutput* indicate that the planners deriving from them will use different interfaces to the collision detection objects. *PL_LinearCollision* uses the *CD_Linear* interface, while *PL_BooleanOutput* uses the point probing collision detection routines found in *CD_Bool*. The word output appended to *PL_BooleanOutput* indicates that for the time being these classes also contain an output window for debugging purposes.

3.6.2 Linear

The first planner to find its way into the MPK is what I call a straight line planner. It is a local planner that operates in much the same way as the *CD_Linear* collision detector, in that it attempts to plan a path by checking several points along a linear path in C-space between the start and the goal. The Linear planner uses only the *CD_Bool* interface to collision detection, not the *CD_Linear* interface.

The path that is returned by the planner is a sequence of points intermediate between the start and the goal. If there is no complete path, a partial path, ending in an interference is returned.

This planner is not sophisticated. It was the first "proof of concept" planner that was included in the MPK and its presence persists only because it is good for testing purposes due to its simplicity. It is guaranteed never to crash or contain memory leaks. Unfortunately, it is not a very powerful or complete motion planning algorithm. It fails to find a valid path if there are any obstacles between the start and goal configuration. Because this planner was written to make full use of the MPK interfaces, it took only about 15 minutes to code and test completely, making clear to me the benefits of using

the MPK system. To illustrate the ease of implementation, the code of the linear algorithm is presented in Figure 40.

```
path.AppendPoint( startConfig ) ;

const long int steps = 1000 ;
path.Clear() ;

//test all points from start to goal
Configuration current = startConfig ;
Configuration offset = goalConfig - startConfig ;
for( long int i = 0; i <= steps; i++ )
{
      if( collisionDetector->IsInterfering( current ) )
      {
            return false ;
      }
      path.AppendPoint( current ) ;
      current += ( offset * (1.0 / steps ) ) ;
}
return true ;
```

Figure 40: Code of the Linear Planner

Note how little setup and teardown the MPK requires the planner to perform. The only setup is to use the start and goal configurations stored in the *startConfig* and *goalConfig* variables. These can even be used as is if the programmer desires. And as for teardown, putting the path points into the *PA_Points* structure is all that is needed.

3.6.3 Random

To address the problems with limited capability that the linear planner contained, but still retain the simplicity of implementation it had, I wrote a slightly more complex planner. The Random planner attempts to plan a path from start to goal that avoids simple obstacles. The linear planner simply connected a line from start to goal in C-space, meaning that if that straight line was blocked, no path was reported. The random planner

attempts to avoid the obstacle by placing one intermediate point at random in C-space, and using it as a "via" point.

If the path using the via point is still not collision free, a new random via point is chosen. This algorithm repeats until a valid path is found, or a specified number of iterations has been exceeded. This planner is still quite limited, since many paths will generally require more than one via point in order to move from start to goal.

3.6.4 ACA

The realm of global planners included in the MPK framework starts with the ACA planner. This planner is global, and is guaranteed to find a path from start to goal within a given resolution if one exists, given sufficient time. A brief description of the algorithm as it was defined in [12] is as follows.

Working outwards from the start configuration, the planner places "landmarks" in c-space. A landmark is defined to be a collision free point in c-space. The first landmark is defined to be the start configuration. Associated with each landmark are several "embryos", these are additional points in c-space near the parent landmark that are also collision free. For each embryo connected to a landmark there exists a path between its location and the landmark's location that can be found using a local planner.

The algorithm begins by using the local planner to determine if there is a path between the first landmark and the goal. If there is not, then one of the embryos in the system will become a new landmark. The embryo that undergoes this metamorphosis is defined to be the one that maximizes the distance between it and any other landmark already in existence. This embryo, once chosen, becomes a landmark, and must generate random embryos of its own. The new landmark is then tested to see if the local planner can reach the goal from there. If not, then the algorithm repeats itself. A graph is kept of the tree of landmarks, with connections between landmarks and their parents. Once a landmark can reach the goal, this graph is traversed to build the ultimate path.

This algorithm is very good at completing its task and finding a path, however, when the robot must maneuver through tight spaces, often an extremely large number of landmarks must be placed, and the algorithm may take a long time to complete its planning task.

Juan Manual Ahuactzin, the original architect of ACA (also a regular summer visitor to the Robotics Lab), and I ported this algorithm from its original form to a form compatible with the MPK. Juan modularized his algorithm so that calls to the collision detection routine were encapsulated in wrapper functions, allowing me to easily point them to the proper MPK calls. He also structured his code so that I could easily convert it from its native C to object oriented C++. In addition, I made a few minor improvements. Originally, there were parameters in the algorithm hardcoded using #define statements, presumably to make the C code more easily reconfigurable. I removed all #define statements. Some were converted to data members of the object and others were eliminated, meaning that limitations such as a maximum number of joints are no longer present, and parameters such as the number of embryos to create can be set at runtime, rather than at compile time.

3.6.5 Sequential

The sequential planner was ported to the MPK system by fellow researcher Gillian Lo. Her efforts are documented in [13]. This planner attempts to use the serial nature of a robot to its advantage when planning a path. It breaks the robot up, and plans a path one link at a time, starting with link 1, the link closest to the base in the kinematic chain.

The path for each individual link is computed by discretizing joint space, and computing a 2D "bitmap" representing collision locations for all possible values of this joint, and all positions of the robot in the paths of previous links.

This algorithm is very robust. If a path exists from start to goal, it will *almost* always find it. There are a few special cases in which a path exists, but will not be found. There are a few parameters that can be tweaked with regard to the operation of this planner. As mentioned, the resolution of the bitmap can be altered, as can other internal parameters,

level of backtracking, maximum number of links, etc. In future I would like to revisit this planner, and make each of these parameters user configurable.

3.7 File I/O

During the later stages of this project, file IO became an important issue. While porting the sequential planner, it became apparent that there were bugs in the algorithm that was provided. In the process of debugging, test cases of robot/obstacle configurations that caused the planner to break were discovered. Functionality to save these test cases for later tests was desired. That way, once the bug had supposedly been fixed, the exact test case that was used to discover it could be used to test the fix.

For the time being, file I/O is a function of the user interface. Although in later iterations of the MPK as an interactive web system, file I/O may become a task that the server performs. In the PC prototype of the MPK user interface, the system contains a universe, replete with robot and obstacles. It contains a start and a goal for the robot, as well as information regarding which collision detection scheme, and planner to use to compute the path. If a planning task has been completed, it will also contain a path. In order to satisfy the need to save test cases, all this information must be output to disk.

As mentioned, the user interface is responsible for file I/O. When the user selects save file in the PC prototype menu, a file is created on disk. Henceforth, that file is treated as a *stream*. In c++ terminology, a stream is an object used for I/O. Whether it is to disk, screen, or some other device, streams are always treated the same way. Treating I/O as stream I/O also allows me to standardize the interface to I/O routines for various classes. It also allows me to write polymorphic I/O routines. Henceforth, I/O is described in terms of *serialization*. Serialization is the process of turning an object into a text string, while deserialization is the reverse process, creating an object from text. Serialization and deserialization are terms commonly used in file I/O.

3.7.1 Serializeable class

All elements in the MPK system that can be output to disk inherit from the *Serializable* class, which is an abstract base class that forces its children to expose two member

functions: *Serialize()* and *Deserialize()*. Both of which require a stream as a parameter. Serialize requires an output stream, while deserialize requires an input stream. In addition to the Serialize and Deserialize member functions, I have chosen to overload the stream output operators that are native to the standard library of C++; the operators '>>' and '<<'. These operators allow you to output objects to a stream in a straightforward manner. One can simply write

```
outputStream << myObject ;
```

and serialization has been accomplished, making this an extremely convenient operator in cases where many items must be serialized one after another. Writing

```
outputStream << object1 << object2 << object3 ;
```

is easier than writing

```
object1.Serialize( outputStream ) ;
object2.Serialize( outputStream ) ;
object2.Serialize( outputStream ) ;
```

These stream output operators do not require much additional coding effort however, because internally they simply utilize the Serialize/Deserialize operators

3.7.2 Polymorphic Deserialization

One problem inherent in Deserialization (reading objects from file) is encountered when collections of objects have been stored. Polymorphism permits lists and arrays of pointers to a base class to be stored. When these are subsequently serialized, the member function

```
pointerToUnknownObjectType->Serialize( stream ) ;
```

must be called, properly serializing the object to which the pointer points. However, when reading this object back from the file, you need to allocate memory for an object, and know which serialization routine to call. Memory allocation should not really be the responsibility of the client's code. As such, I implemented a static member function in the parent class of the object hierarchy that is capable of performing Deserialization of all its children. Thus all the programmer needs to do is call

```
ObjectBaseClass* ptr = ObjectBaseClass::Deserialize( stream ) ;
```

and the object will be correctly read from the file. In order to permit this functionality, some constraints are placed on the serialization of these objects. In particular, flags indicating the type of an object are essential in order to be able to properly read them.

3.7.3 File Structure

The file structure in its current state is extremely primitive. It permits loading and saving of all robots and environments that can currently be created using the PC prototype of the interactive system. This file format is not meant to be edited by hand, therefore little consideration is placed on readability or tolerance for errors in the file. Only primitive support for comments is provided. These are drawbacks; however, manual editing of the file is not something that is likely to be performed by many users. Editing kinematic descriptions is much easier performed online, in an interactive manner. Similarly, geometry is likely to be read from file rather than typed in manually by a MPK user.

Chapter 4 Internet Connectivity

It is a desired aspect of this project that people be able to make use of the MPK system from remote locations, specifically via the web, which will allow other researchers to see what the capabilities of the system are, and how the MPK can be adapted to service their needs. Although it was not something that I had planned to spend much time on, the Internet connectivity has begun design and is mentioned here for completeness. At present, we are capable of demonstrating use of the MPK via the web. A user can choose a robot from a short list of predefined robots, they can select some predefined obstacles, a collision detection library, and a planning algorithm, then tell the server to plan a path. The user can then see this path animated in a browser window. Much of the front end work was done by Javier Fransisco Blanco, a visiting student from the University of Salamanca, Spain, and will be documented in a forthcoming work[8].

I spent time working on development of the server side of the application. What I managed to accomplish was to lay the groundwork for future development of this application. At present the server application is capable of listening on a port for any client applications to connect. When this happens, the server spawns a child thread to process commands from this particular client. Only a few requests are currently supported. The client application is allowed to choose from several default robot configurations, much the same as the menu options in the PC prototype. The client can then select a collision detector type, a planner type, a start and goal configuration, and tell the server to begin planning. Once planning is complete, the client can request the server return the path it has computed. All the above functionality is working, but only in a very rudimentary form. I have not performed much “crash proofing” of the server application. As it stands, the program uses core MPK code, and is fairly robust simply because the

objects it uses are robust, however, the communications aspect of the server is new code, and not yet thoroughly tested.

4.1 Client Server Architecture

The primary architecture of the interactive MPK system consists of a web based front end, communicating with a server daemon running locally on one of SFU's computers. The front end will allow the user to design a robot and environment, and set start and goal configurations. Essentially it will allow the remote user to specify a problem to be solved by a planner through the MPK system. Once the problem has been fully specified, it will be sent to the server application for processing. When the server has completed this task, it will return the results it found – the resultant path.

Communication between the client and the server is done through raw TCP/IP socket connections. Our other options included CORBA and DCOM, but we rejected these because of the long learning curve involved with each of them, along with issues of compatibility and cost. COM/DCOM is only compatible with Microsoft platforms, while CORBA would cost the users of the system money to have installed. TCP/IP is free, very general, and not difficult to program.

4.2 Java Front End

The front end is required to provide a user friendly interactive environment for the user. The UI is the only part of the system they will be seeing at first, as opposed to the code toolkit, so it is essential that it aspire to be easy to use, and comfortable to design path planning problems with. Several problems should be pre-defined, so that users can begin to see what the MPK is all about right off the bat.

Some criteria for the front end are as follows:

1. The front end must work across multiple platforms; Sun, PC, Mac, etc.
2. It should be browser independent.
3. It should allow as wide an array of researchers to experiment with the MPK as possible.

4. It should be easy to use, and have a simple UI that researchers using multiple languages could understand it.

In order to satisfy these constraints, we chose the Java development language in which to write the front end. Java functions on all browsers, and on multiple platforms. It was designed to be a cross platform language. One drawback to the Java approach is that because it is not compiled code; it tends to be a little bit slower than a standalone application might be. Speed is not really a factor though, because the front end does not have to be very fast. Animating robot paths smoothly would be nice, but the important aspect is that researchers be able to see that the task has been correctly planned. In fact, since planning is the most time consuming aspect of the use case, and it's already running on our server, which should be very fast, the speed aspects of Java are not a big issue.

One question that arose during preliminary development of the Java UI centered around 3D display technology. On the PC prototype, OpenGL is the default display technology used. Choosing this was basically a moot point. All Windows NT machines come with OpenGL installed. However, for the web based front end, the choice of display methods is not so simple, several options exist, each with its own pros and cons. JavaGL[6] is a loose wrapper shell over OpenGL. It requires that OpenGL be installed on the client machine, and mimics the functionality that OpenGL provides to the PC prototype. Java3d[7] is a more powerful architecture that makes more complex graphics routines possible. Unfortunately, it requires additional components to be installed on the host machine, in addition to OpenGL. The third option is one that requires no additional components to be installed whatsoever, this is simply to use native Java drawing routines to render the robot to the screen.

The rendering option that we have preliminarily chosen is Java3D. Since both Java3D, and JavaGL require additional installation effort on the client machine, and the Java3D solution makes development easier, it seemed like the logical choice. At this time, development of the front end has completed to a point where it emulates the PC prototype

with regard to specifying the start and goal configurations of the robot, and visualizing the paths that are returned

4.3 C++ Server

The front end will communicate with a dedicated server operating on one of SFU's machines. The server is responsible for coordinating communications with multiple remote users at the same time. It will receive commands from the remote machines, and process these commands. Once planning tasks have been completed, it will send the results back to the remote UI for display to the user.

The server application must be very robust. Because additions and modifications are going to be made constantly to the interactive system, the server application will be periodically changing. Some of these changes will undoubtedly contain bugs, some of them crash bugs. With stability in mind, the design of the server should be such that if a remote user executes a command that causes the server to crash, it can recover automatically from that crash. Equally importantly, if multiple users are accessing the server at the same time, one user should not be able to crash the system for everybody. In order to accomplish this level of fault tolerance, the server architecture will be that of a multithreaded system. Users first connect to the primary server, which assesses the needs of that particular user. It will then spawn a secondary MPK server that process commands issued by the user. There is a one to one relationship between remote users and secondary servers, so that if a user crashes his server, other users are unaffected. Presently, this mechanism is in place, although robustness issues have not yet been tested.

4.4 UI Prototype

Because the Java UI was not completely developed when this paper was begun, a PC prototype of the user interface was developed. The PC prototype allows me to demonstrate aspects of the MPK project without having to focus on UI design, which is not my primary area of interest. It also allowed me to debug the MPK code without having to operate within the client server framework. The UI Prototype was developed in C++ using the MFC library that is distributed with Microsoft Visual C++. This library

automates certain Windows programming functions, and in general accelerates the pace of development.

Eventually, all the functionality that the UI prototype allows will be permitted in the remote interface. The level of convergence between the two interfaces is not complete at this point. To properly simulate the Java UI, the C++ prototype should spawn a server object, and communicate only with it. At present, the C++ prototype does not spawn a server, all the MPK functionality is accessed by directly calling MPK functions. This method of invoking commands is the route a programmer would take if they were going to use the MPK as part of some other system. What it does, however, is mask some of the complexity of developing the Java UI that will be encountered later. Because the C++ prototype has access to the full set of the MPK library, operations like rendering a robot are simplified. The Java UI will have to implement these operations itself.

Chapter 5 Future Improvements

Due to time limitations of a BASc thesis, some aspects of the system that had been slated for development in this cycle could not be attended to. These aspects of the MPK will be addressed in subsequent versions. These all fall under the broad heading of *extended functionality.*

5.1 Moving Objects

One area of the MPK's kinematics and collision detection routines is the concept of moving objects. Moving, or time varying objects refer to a planner's ability to plan a path for a robot that avoids an obstacle whose position changes with time. Time varying obstacles, in their simplest form are objects in the universe that exist in frames that are not controlled by joint variables that the planner can modify at will. These obstacles exist in frames that are controlled instead by time. A simple example of a time varying object would be a ball that is rolling across a table at a constant velocity. The equations of motion of this object are extremely simple, and completely specified well in advance of any planning task. Maneuvering a robot from start to goal while avoiding such an object would constitute the planning task. In this case, the start and goal configuration might be the same, causing the robot's path planner to plan a path that simply avoids the rolling ball, but doesn't rally go anywhere.

The concept of time is in itself a sticky topic to introduce to the MPK. There are several different possible ways to treat time. It can be a concept known only to planners, and they deal with time varying objects as a special case. Time can be treated by planners and collision detectors as a separate parameter, and special care must be taken when specifying values for it – the values can never decrease for example, because "going back in time" is not permitted.

I propose that time be treated like a joint variable just like all the others, with as little special consideration as possible. Since the entire time history of the obstacle is known, we can specify its position given the value of time, making it possible to leave queries about collisions with the obstacle at a given time in the hands of the collision detection object. A different, time enabled, collision detection object will have to be derived. Point collision queries will remain unchanged, specify a vector in C-space and determine if there is a collision or not. Only now, one dimension of C-space is controlled by time T. Linear queries will have to be modified, however to preserve directionality. In standard, time independent queries, an implicit assumption exists that if you can traverse a path from C_1 to C_2, two points in C-space, the path from C_2 to C_1 can also be traversed. However, if time is inserted as one of the elements in the C vector, this is no longer true. In fact it is completely false. If you can travel from C_1 to C_2, that means that time(C_2) > time(C_1). Travelling the other way is tantamount to going back in time, and must be ruled impossible. This criterion affects any areas of the MPK that can be considered *local planners*.

Planning algorithms may also have to be modified to recognize the lack of symmetry this represents. Graph search algorithms must now operate on directed graphs, rather than undirected ones, for example. It may be necessary to add constraints to the collision detection object to specify maximum joint movement rates, allowing the collision detector to recognize that moving from point A to point B in C-space in a given amount of time may be impossible due to physical limitations on the movement of joints.

Notwithstanding the modifications that must be made to the collision detection routines, changes to the kinematics structure will also have to be made. Some way of representing the movement of an object with respect to time must be created. This representation could be either a list of frames and corresponding times that are interpolated, or some more complex mathematical function description. Either way, representing the motion is a fairly complex routine. Interpolation of frames is non-trivial. One cannot simply interpolate the matrix elements that underlie the frame. A quaternion representation of

the rotational component can be stored, along with a translation vector, which could then be linearly interpolated, or interpolated using some more complex spline method.

Time varying objects must also support the notion of multiple robots, bringing to the fore one of the sticky situations I was discussing previously. It is conceivable that one of the more common time varying objects a user might want to simulate using the MPK would be that of an additional robot operating on some fixed program. This situation would be equivalent to a robot whose path had already been planned to move through the environment. The second robot would now have to plan its path *around* the pre-existing robot. In itself, no design problems more challenging than those discussed earlier for the general moving objects problem are posed in this case. A problem does arise if the concept of having more than one robot in the system at the same time is carried to the next logical step – cooperative planning.

5.2 Multiple Robots and Cooperative Planners

Having more than one robot in the MPK system is not a problem. The kinematics module supports robotic structures of a wide variety of types as long as they are all open chain robots. Two two-link robots would be represented as 4 links, defined in a specific manner. For planning purposes, the entire universe, all 4 degrees of freedom are treated as one planning task.

Treating the entire universe as one planning task, to be performed by one planner eliminates an avenue of research from being pursued by the MPK, that of two planners working in tandem to solve the problem. Because the universe described above represents two distinct robots, it is conceivable that there could be two different algorithms planning their motion. Unfortunately, allowing more than one planner causes system design problems. At present, there is no built in method for these two planners to communicate with one another to synchronize their tasks. When collision detection queries are made, the entire state of the universe must be specified. If two planners are operating in tandem, how does one keep track of the positions of joints for the robot that is being controlled by the other?

One proposed method for the MPK to allow synchronizing the planners is to do nothing special. The programmer who writes a planner object that simulates cooperative planning will have to handle this problem. That programmer can use multiple threads to operate each individual planner, and can place semaphores to restrict access to the collision detection object to one planner at a time. Leaving things in the programmer's hands does not seem to be an ideal solution by any means. Too much work is left to the third party programmer. If cooperative planning is deemed to be an important area in which the MPK will be used, additional attention should be allocated to this problem.

5.3 Movable Objects

A problem that is likely to arise are cases in which the robot in the system picks up an object that was originally considered an obstacle, moves it, puts it down, and continues on its way. This situation could occur because moving the object was the planning task that the planner for the robot was supposed to solve a path for. It could also occur if the path for the robot from start to goal is completely blocked by an obstacle. In this situation, an extremely sophisticated planner could grasp the obstacle, move it out of the way, then proceed to complete its task.

Kinematically speaking, the concept of a robot picking up an object is not difficult to deal with. The robot has a tool frame defined, which is the frame that represents the position of the robot's gripper. In order to pick up an obstacle, all the kinematics module needs to do is transfer the object from its current frame into the tool frame of the robot. Some transformation matrix will have to be applied to the object so that it maintains its position in world space, and the base frame of the object will switch. It's as simple as that.

Questions of validity do arise however. Should the MPK prevent a robot from picking up an object if the gripper and the object are too far apart? Probably not, it's best to leave this behavior firmly in the hands of third party programmers. There should be a method of checking the distance between the gripper and any obstacle, but preventing a pickup should not be forced. Also, if an object can be picked up, it can also be put down again,

bringing up another minor problem. If the robot should happen to release an object, the frame it is defined with respect to switches to frame 0, the world frame. However, if the robot released the object in mid air, it will stay there, floating in space. This is all fine and dandy if we're simulating the Canadarm on the space station, but for more earthly applications, gravity should pull the object down to the first stationary object. Unfortunately, neither the presence of gravity, nor any dynamic simulation capabilities are included in the MPK at this point. These are potential, future areas of programming interest, but dynamics could not be added for quite some time. For the immediate future, it should be sufficient to allow the robot to release an object anywhere, forcing third party programmers to ensure that they only drop objects in intelligent locations.

5.4 Unknown Static Environments

Yet another improvement to the MPK is the inclusion of unknown static environments, a type of environment about which the planner has only a limited set of knowledge. Commonly found in sensor based planning is the case where the planner initially knows virtually nothing about obstacles in the environment. It must progressively "scan" the environment at different positions to learn enough to complete its task. Scanning can be done via a laser depth scanner, proximity sensors, etc.

Simulating these sensing devices would be the most difficult aspect of incorporating unknown static environments into the MPK. The notion of a Sensor class would have to be created, along with a collision detection object that could store collision, unknown, and free space information. Likely, the collision detection object in this case would maintain two representations of the universe. The true representation, and the representation that can be "known" from any scans that have taken place.

Adding this functionality to the MPK would be extremely desirable, considering much of the current research being conducted at SFU[14] and elsewhere is moving in the direction of sensor based planning.

5.5 General Geometric Planning Tasks

Although the MPK system's original goal was to provide a framework for motion planning algorithms, there is no reason that it should not be used to develop other geometric reasoning tasks as well. Some possible uses for the MPK include automatic grasping, inverse kinematics, positioning and part mating, non-holonomic path planning. These tasks are better explained in [2]

Chapter 6 Conclusions

I have implemented a core subset that demonstrates the proof of concept of the MPK. What has been accomplished is an overall framework for the entire system, along with several highly fleshed out modules. The geometry and kinematics modules were meant to be my primary areas of focus, and as such, these are the modules that are most complete.

Although the system is not completed, what can be done given the current state of the MPK is still quite remarkable. From a code toolkit standpoint, programmers can use the MPK to construct any robot that can be described via DH parameters, including branched structures, and parallel manipulators. Geometry can be loaded from disk, and used to represent the links of the robot. Only a few file formats are supported at present, a subset of the VRML 1.0 standard, along with the Icollide/Vcollide format.

Once a programmer has designed the robot and the obstacles, performing collision detection is a simple matter of instantiating one object. All the internal collision detection optimizations are handled in a transparent manner, removing this, sometimes complicated, mechanism from the domain of the user's work. Assuming that most people using the MPK from a programming standpoint will be using it to develop motion planning algorithms, these programmers are all ready to go. Once a collision detection object has been instantiated, their planner can make queries of it, until it has solved whatever problem the programmer had asked it to solve.

As far as the interactive MPK system is concerned, development is also quite far along. Originally, I had planned not to perform any development work on the web-based application, opting instead to demonstrate functionality of the MPK using a PC prototype.

At present, the PC prototype is a fairly well developed application. It capably demonstrates that the MPK code performs the tasks it was designed to perform. The PC prototype has proved to be an invaluable debugging tool for me, and the other developers working on the project. Using this prototype, developing new planners is extremely simple to do. The prototype contains several built in robot configurations, and the capacity to add obstacles simply by clicking the mouse. Adding a planner to the system is simply a matter of extending one of the existing planner objects in the system, and overriding one function, the *plan()* member function. The web based application is surprisingly almost as far developed. A proof of concept system that can demonstrate planning problems on predefined robots via the web can be demonstrated at present.

As mentioned in previous sections, improvements can be made to all areas of the MPK, *including* the sections I focused on, such as geometry and kinematics. The MPK is by no means "done". Further development is necessary to ensure that it achieves the goal of being a widely used system that aids the field of motion planning by allowing researchers to avoid programming overhead when developing new algorithms. Taking the MPK from its current infant stages through to completion would make an excellent master's thesis for one or more capable individuals.

References

[1] Gupta, K.K. "Overview and State of the Art" *in Practical Motion Planning in Robotics: Current Approaches and Future Directions,* 1995.

[2] Latombe, Jean-Claude. *Robot Motion Planning*. Klewer Academic Publishers, 1991

[3] Lozano-Perez T. *Automatic planning of manipulator transfer movements.* MIT Artificial Intelligence Lab, 1980

[4] Craig, John J. *Introduction to Robotics Mechanics and Control*. Addison Wesley Publishing, 1989, p30.

[5] http://www.sgi.com/Technology/STL/stl_introduction.html

[6] http://cml19.csie.ntu.edu.tw/~robin/JavaGL/

[7] http://java.sun.com/products/java-media/3D/

[8] INTERNAL DOCUMENT: Blanco, Javier Fransisco *Description of the Web Based Front End to the MPK,* 1999

[9] http://www.cs.unc.edu/~geom/V_COLLIDE/

[10] http://www.win.tue.nl/cs/tt/gino/solid/

[11] refer to greenspan's work

[12] Emmanuel Mazer, Juan Manuel Ahuactzin, El-Ghazali Talbi, Pierr e. Bessiere. *The Ariadne's Clew Algorithm: From Animals to Animats*: Th e Second International Conference on Simulation of Adaptive Behavior (SAB92). Honolulu, Hawaii, USA December 7-11, 1992.

[13] INTERNAL DOCUMENT: Lo, Gillian, *ENSC 492 project report:Adapting the Sequential Planner algorithm to an MPK framework,* 1999

[14] refer to Yong's work

Appendix : Denavit Hartenberg Links