Planning Camera Motion in a 3D Environment

Li Han

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

March 2006

Your file Votre référence
ISBN: 0-494-14320-7
Our file Notre référence
ISBN: 0-494-14320-7

Abstract

Planning Camera Motion in a 3D Environment

Li Han

Most systems for automatic navigation inside a 3D virtual environment often require pre-decomposing the free space into some kind of uniform units or pre-calculating the path or road map. In this thesis, we propose a new technique for navigation through in a 3D virtual environment without lots of pre-calculation so that the system allows users change their interest target during the camera motion, and even allow users dynamically adjust the 3D objects inside the 3D virtual environment without too much recalculation involved. The technique has successfully been integrated in a system for walkthroughs in a 3D virtual environment.

Acknowledgments

I am full of gratitude to everyone who helped me when I was working on my thesis.

First of all, I really appreciate Dr. Peter Grogono for introducing me to Computer Graphics and helping me to discover the beauty of it. I took great advantages from some OpenGL materials written by him when I came into contact with Graphics at the very beginning. After I started my Graphics research under his supervision, he always honestly shared his opinion with me, especially gave lots of suggestions and comments on this thesis.

Secondly, I would like to thank my parents and my husband, Steve Giroux, who always strong supported and understood me during my study. Without them, this work would not have been accomplished.

Last but not least, many thanks to all the members of the examining committee for their time, informative and well documented remarks.

Contents

List of Figures

List of Tables

1 Introduction

From the movie special effects that captivate us, to medical imaging to games and beyond, the impact that 3D graphics have made is nothing short of revolutionary. Computers are getting faster and faster, with enhanced graphic capabilities providing the resources on which graphic technologies can evolve. Specifically, 3D technologies have been getting more and more attention the last few years.

The field of 3D graphics is expansive and complex. Our goal is to present a series of fairly technical yet approachable articles on 3D graphics technology. Virtual environments become widely presented, and the crucial of all virtual environment systems is to effectively control the viewpoint or virtual camera. In many virtual environment applications, like games, architectural walkthroughs, urban planning systems, CAD model inspection systems, and training systems, the user must navigate through the environment to inspect it and perform certain tasks.

Therefore, a classical assignment is to simulate camera movement through the virtual environment freely and automatically. Many applications either use pre-defined or pre-calculated paths for the animated presentation of scenes, or they leave the camera control entirely up to the user. Such direct control has a number of disadvantages. It is difficult for inexperience users, it results in rather ugly motions that easily lead to motion sickness, and it requires a lot of attention while the user should preferably concentrate on more high-level tasks.

We are looking for methods and tools which support the co-existence of user-controlled movement and animation for the purpose of presenting objects and actions in a 3D virtual environment.

We propose a new technique for navigation through in a 3D virtual environment. The technique avoids pre-decomposing the free space into smaller units and also avoids pre-calculating the path or road map. But of course, the system has to know about the 3D environment first and then manage to generate a path automatically. Hence, the first goal of this thesis is to build up a 3D virtual environment, which is made of 3D models (e.g. tables or chairs) and separators (e.g. walls or ceilings). The objects inside the environment all have a bounding box, which is pre-defined by the system while the user is building the 3D virtual environment. Then, we allow the camera to simulate collision-free walking through the virtual environment either manually or automatically.

Certainly, the emphasis of our technique is automatic walkthrough in a 3D virtual environment. To start with, the user specifies target of the camera, the system computes a camera path, and then optimizes the initial path. Finally, the path is refined to present the user with a smooth and pleasant camera motion. Obviously, the central task is to find a path, in which the camera must follow a "human" behavior, that is, staying at a particular height above the ground. In a 3D space, the user moves through the space at fixed height and, therefore the camera movement generally is reduced to 2D. Because we indeed move in 3D space, the camera is considered as a directed line instead of a point to prevent the camera flying over an object from the top.

This thesis is divided into 6 chapters, in which contributions are introduced step by step as follows:

Chapter 1 introduces readers to the concept of 3D graphics, the problem of controlling camera movements in 3D virtual environment in general, and the goals of this work.

Chapter 2 represents previous solutions to relative area. We start with analyzing those previous works done in relative subjects. Then, we represent the more advance solution.

Chapter 3 describes the design of the program used to buildup a 3D virtual environment that is suitable for the algorithms of the camera walkthrough manually and automatically presented in this thesis.

Chapter 4 defines the implementation of the *Virtual Environment Builder* and the *Walkthrough Controller* that are described in Chapter 3.

Chapter 5 shows the results obtained and uses several screenshots of a sample scene created by our program.

At the end, Chapter 6 gives the conclusion of this thesis and suggests some of the possible future work.

2 Background and Related Work

Over the past years, extensive research has been done on supporting camera motion in virtual environments, including to assist the user in controlling the camera and to control the camera automatically.

2.1 Manually Camera Control

A number of authors have studied techniques to support the motion directed by the user. For example, A.J. Hanson and E.A. Wernert [9] proposed a unified mathematical framework for incorporating context-dependent constraints into the generalized viewpoint generation problem. Detailed examples have been worked out and presented for the particular case of a 3D through-the-screen display controlled by a 2D mouse. The basic strategy is to supply a set of view-determining data at each sample point of a "virtual sidewalk," along with possibly state-dependent procedures to create the actual view to be presented. Ultimately, it is up to the designer to limit the viewer's freedom of navigation enough to focus attention and prevent loss of context, but not so much as to disturb the feeling of exploration and discovery appropriate to the viewer's task.

J. D. Mackinlay, S. K. Card, and G. G. Robertson [10] described a technique called *point of interest logarithmic motion*, which offers an improvement in the techniques available for targeted 3D viewpoint movement, and which is useful when users wish to rapidly access many objects or objects with great detail. In this technique, the user starts by selecting a point of interest. This information is used to simplify the user's control task, resulting in movement that is both rapid and controlled. On each

animation cycle, the viewpoint is moved the same relative percentage of the distance toward the point of interest target. Thus the movement is rapid when the user is distant, but slow and controlled when very near the target.

Even though the main focus of the system described in this thesis is automatic camera control (in other words, path finding), our system also provides the basic manual camera control by keyboard or mouse to allow users navigate inside a 3D virtual environment. The manual walk-through functions include moving the camera linearly (forward/backward, right/left, and up/down), turning the camera (yaw/pitch/roll), and rotating the camera with respect to a particular point. However, the methods to assist the user in controlling the camera [9, 10] do not assist the user in actual navigation to a particular goal. Consequently, we next discuss other related research that has been done on automatic camera control.

2.2 Automatic Camera Control

The area of research on automatically controlling the camera can be roughly divided into two groups of techniques, including planning camera motion as a sequence of shots, generating camera motion to track a moving guide, and generating a collision-free camera motion from a start to a goal placement through a virtual environment.

2.2.1 Planning Camera Motion as a Sequence of Shots

There are some papers that study the computation of effective fixed camera positions to assist the user in performing certain (manipulation) tasks, the so-called "shot systems", and the shots are often calculated by solving a set of constraints [11, 12, 13]. For example, S. M. Drucker and D. ZSeltzer [12] proposed a method of encapsulating

camera tasks into well defined units call "camera modules". Through this encapsulation, camera modules can be programmed and sequenced, and thus can be used as the underlying framework for controlling the virtual camera in widely disparate types of graphical environments. These systems do not plan obstacle-avoiding motions.

2.2.2 Generating a Camera Motion to Track a Moving Guide

Most previous work on planning camera motions is directed toward systems in which the camera must follow an object: third-person games are a good example. This area can be roughly divided into two groups of techniques.

To start, one family of techniques assumes the path of the guide to be unknown. For instance, H.H. Gonzalez-Banos, C.Y. Lee, and J.C. Latombe [14] proposed an algorithm that computes a motion strategy based exclusively on current sensor information – no global map or historical sensor data is requested. The algorithm is based on the notion of *escape risk* and the computation of an *escape-path tree*. The escape-path tree is a data structure storing the most effective escape the observer's field of view. Also, N. Halper, R. Helbing and T. Strothotte [15] introduced a way to compute visibility constraints-based for an arbitrary number of points. Their camera system works for arbitrary dynamic scenes and spatial complexities of environments. The camera needs no specialized collision information. They were the first to provide a constraint-solver based on existing camera state and motion characteristics. The method produces intelligent "nearest-best-fit" frame-coherent camera animations in real-time by reacting to future conditions.

The second group of techniques assumes the motion of the object is known beforehand. O. Goemans and M. Overmans [5] presented a technique to generate a

camera motion such that the camera tracks a guide moving through a known environment along a known path. The motion planner applies a single-shot (*Probabilistic Roadmap Method*) approach to construct a graph in the free configuration space. A substantial performance gain is accomplished by a technique which determines whether a node or edge should be added to the roadmap based on its usefulness. The resulting camera path is smoothed to improve the path quality.

Our problem setting is rather different from the ones studied above. It requires motion planning rather than reactive behavior, and therefore the techniques above are not suitable to solve our problem, which is generating a collision free camera path from Start to Goal.

2.2.3 Generating a Collision Free Camera Motion from Start to Goal

In this section, we present work that has been done in the field of camera path planning.

2.2.3.1 Intelligent Path-Finding

B. Stout [7] summarized two major path-finding approaches, in which the vicinities are made of tiles – rectangular pixmaps of predetermined size. The first approach is *finding the path while moving*, and there are three obstacle-avoidance strategies:

Movement in a random direction. If the obstacles are all small and convex, the entity can probably get around them by moving away a little bit and trying again, until it reaches the goal. A problem arises with this method if the obstacles are large or if they

are concave; in this case, the entity can get completely stuck, or at least waste a lot of time before it stumbles onto a way around.

Tracing around the obstacle. Fortunately, there are other ways to get around. If the obstacle is large, one can do the equivalent of placing a hand against the wall and following the outline of the obstacle until it is skirted. The problem with this technique comes in deciding when to stop tracing.

Robust tracing. A more robust heuristic comes from work on mobile robots: "When blocked, calculate the equation of the line from your current position to the goal. Trace until that line is again crossed. Abort if you end up at the starting position again." This method is guaranteed to find a way around the obstacle if there is one. However, it will often take more time tracing the obstacle than is needed, making it look pretty simple-minded-though not as simple as endless circling.

The second approach is *planning the path before moving*. The entire path is calculated in advance before any move. There are several path algorithms, including Breadth-first search, Bidirectional breadth-first search, Dijkstra's algorithm, Depth-first search, Iterative-deepening depth-first search, Best-first search, and A* Search. We will not discuss this approach in detail. The reason is that the second approach is not related to what we are doing since we do not pre-calculate the entire path.

Compared to the approach that we are proposing in this thesis, the background and environment are rather different. Our system is based on a 3D environment, which is using 3D objects, instead of 2D images made of tiles, for the background/environment and the vicinities. However, we have a generally similar concept of the obstacle-avoidance strategies, for example, moving in certain direction, tracing around the

obstacle, and combined all the strategies so that we are able to include most of the situations. Due to different build-up environment, the implement algorithms are certainly different.

2.2.3.2 Robot Motion Planning

J.C. Latombe [16] presents three general concepts for robot motion planning if a start configuration and a goal is given and a path to the goal is calculated while avoiding obstacles. The presented methods are called *roadmap*, *cell decomposition* and *potential fields*.

Roadmap methods capture the connectivity of the free space in a network of one-dimensional curves R, called a roadmap. For path planning the initial and goal configurations are connected to R, and R is searched for a path between these points. The visibility graph and Voronoi diagrams are examples of techniques based on roadmaps.

Cell decomposition methods decompose the free space into cells in such a way that a path between any two configurations can be easily generated. A connectivity graph is generated and with this a continuous free path is computed.

Potential field methods discretize the free space into a fine rectangular grid. A particle moves through the grid under the influence of attractive forces (introduced by goals), and repulsive forces (introduced by obstacles) thus generating a path. Compared with roadmap and cell decomposition, potential fields are more efficient. However, they may not always find a solution and they may get stuck in local minima of the potential.

S. Beckhaus, F. Ritter, and T. Strothotte [6] introduced *CubicalPath*, a dynamic potential field-based camera control system that helps with the exploration of virtual environments. The potential field adjusts itself when objects of interest are viewed by the

camera. Therefore, objects lose their attraction and after a while the camera moves to the next interesting object. As the geometry is completely transformed into cubes, the CubialPath method operates only on the number of cubes that define the cube space. It uses an "abstract" and simplified version of the geometry data through its cubes.

S. Bandi and D. Thalmann [17] divide the space into a 3D grid of uniform cells. Then with the A* algorithm, which is a roadmap approach, the shortest path between given points x and y is computed.

2.2.3.3 Finding Path in 2D Games

C. A. Mandachescu [4] proposed a framework for game design in which all objects have a recalculated bounding box. Polygons with four or more edges (depending on the accuracy desired and the shape of the object) represent the bounding boxes. They are automatically generated at the creation of the object and are filtered to minimize the number of vertices while preserving the overall aspect of the object.

The vertices of all the bounding boxes from a region of interest (situated in the vicinity of the direct path from the Source to the Target) are dynamically triangulated using a triangulation algorithm. The result of such a triangulation is a 2D mesh situated in the empty space available for movement. None of the edges generated by triangulation will cross any hard object (source, target, obstacle).

A path from the Source to the Target is then derived by navigating on the edges generated by triangulation as well as on the contours of the hard objects. Further smoothing is done by removing redundant points from the discrete path while avoiding collisions.

2.2.3.4 Motion Planning for Camera Movements

D. Nieuwenhuisen and M. H. Overmars [8] described an approach to automatically planning camera motions in first-person views of virtual environments. The technique is based on a novel application of the Probabilistic Roadmap Planning approach originally developed in robotics. In this approach the user simply specifies a required goal position (and orientation) using, for example, a map, and the system automatically computes a smooth camera motion from the current position and orientation to the required position and orientation. As preprocessing the approach uses the probabilistic roadmap method to compute a roadmap through the environment. When a camera motion is required a path is obtained from the roadmap which is then improved by various smoothing techniques to satisfy camera constraints. The method described allows for free-flying camera motions, either on a constraint surface or in space.

All of the research described in this section depends on discretization of the entire virtual environment by predefined units or prior calculation of the path, and therefore is not applicable to our problem. However, we have gained different ideas and learned various methods by studying previous work. In this thesis, we propose a new technique to find a path for camera motion without decomposing the free space into discrete units and also without pre-calculating the path or road map.

3 Design

This thesis represents a solution for walking in a human-like way through a 3D virtual environment. In order to experiment with automatically controlled camera movement, we needed a three-dimensional virtual environment. Accordingly, the first task was to construct a suitable 3D environment. Since this is a complex task, we chose to start by implementing a software tool to help with the construction of such environments.

3.1 3D Virtual Environment Builder

The *3D Virtual Environment Builder* is a software tool used to build up a 3D virtual environment. It is a tool that includes its own types of objects and is able to load an object and a 3D scene from data files, adjust objects in the 3D scene, and save all the adjustment for further usage. In addition, it may generate a floor plan in order to provide an overall view of the entire 3D virtual environment as well as allow users indicate a target position for camera movements. The *3D Virtual Environment Builder* may also convert the position from the window coordinates (by mouse event) to the world coordinates, a feature that we will need later.

Figure 3-1 shows the generic architecture diagram of the *3D Virtual Environment Builder*.

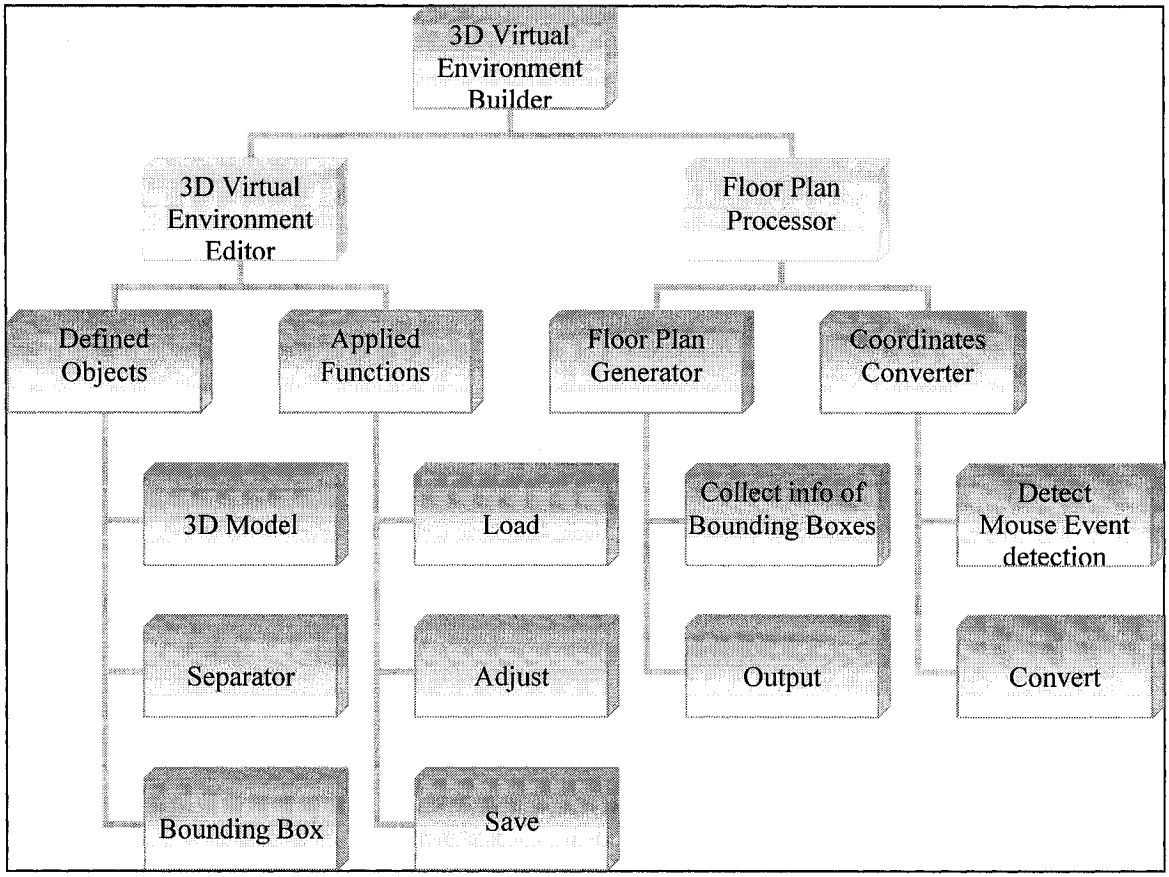

Figure 3-1: The generic architecture diagram of the *3D Virtual Environment Builder*

3.1.1 3D Virtual Environment Editor

The *3D Virtual Environment Editor* is used to create and modify a 3D virtual environment. A 3D environment, for example, a house or a building, contains multiple rooms separated by walls, and each room has various pieces of furniture and other accessories. Therefore, we consider only two major objects, furniture/accessory and wall/floor, in a 3D environment. Because the furniture or accessories usually have irregular 3D shapes, a bounding box is constructed to enclose each object; this makes it easier to locate the object's position and volume in a 3D space. Consequently, *3D Virtual Environment Editor* uses three types of objects: 3D model, separator, and bounding box.

3.1.1.1 Defined Objects in *3D Virtual Environment Editor*

First, a 3D model consisting of a triangular mesh is used to represent any furniture or accessory (such as desks and chairs) in the 3D virtual environment.

Secondly, a separator is made up of polygons. Separators are used to represent walls, floors and ceilings in the 3D virtual environment.

The last object is the bounding box. It is an area in the form of rectangle that encloses each object in the 3D virtual environment.

3.1.1.2 Applied Functions in *3D Virtual Environment Editor*

The *3D Virtual Environment Editor* applies basic functions to its own objects, including load, adjust, and save.

Above all, the *3D VE Editor* needs to be able to load 3D models, separators, and a 3D scene with many 3D models. To load any of them, the *3D Virtual Environment Editor* imports information from a data file in a specified format. It requires different kinds of

information for loading different objects. For instance, loading a 3D model needs the model's vertices, faces, texture information, and material information, and so on; loading a separator requires the separator's length in X, Y, Z axes, position, and orientation, and so on; loading a 3D scene with many 3D models needs the position and orientation for each model, and so on. After loading, users may make adjustments to the 3D scene. The adjustments provided by the software include modifying any object's scaling, translation, and rotation in the 3D world coordinates, varying a particular object's length, and altering a particular object's texture or color.

Users may save all the modification of the 3D virtual environment after making the adjustments. The *3D Virtual Environment Editor* will save all information into data files.

3.1.2 Floor Plan Processor

3.1.2.1 Floor Plan Generator

In order to have an overall view of the entire 3D virtual environment as well as allow users indicate a target position for camera moving automatically, a floor plan is generated automatically, and then displayed on the screen. Previously, we've mentioned that each object has its bounding box. Therefore, the *Floor Plan Generator* draws a floor plan by collecting information about all objects' bounding boxes, and then generating an overall view of the scene while representing all objects in a shape of box.

3.1.2.2 Coordinate Conversion

We allow users to specify a point-of-interest target while automatically generating a camera motion. Users may simply click on the floor plan to indicate the camera target

position; this provides a friendly interface. To achieve this, the program must transform a point from the window coordinates to the world coordinates. The role of the *Coordinates Converter* is to convert the window coordinates (2D) to the world coordinates (3D) whenever a mouse event occurs.

When users click on the floor plan with the mouse, the *Coordinates Converter* first converts the mouse coordinates from windows pixel (0 ~ windows size) to windows coordinates (-1 ~ 1). How do we transform coordinates from 2D to 3D? As we know, the floor plan is a top view, so we assume the camera always stay at the same height when moving automatically. Thus, the *Coordinates Converter* only needs to compute the values of X-axis and Y-axis in the 3D world coordinates and keep the value of Z-axis constant.

3.2 Walkthrough Controller

Once a 3D virtual environment is built by the *3D VEB Controller*, the *Walkthrough Controller* allows users to navigate freely inside either manually or automatically without hitting on any obstacles. Obviously, it mainly involves camera movements. The camera in our case is always a first-person camera giving a subjective viewpoint.

Figure 3-2 shows the generic architecture diagram of the *Walkthrough Controller*.

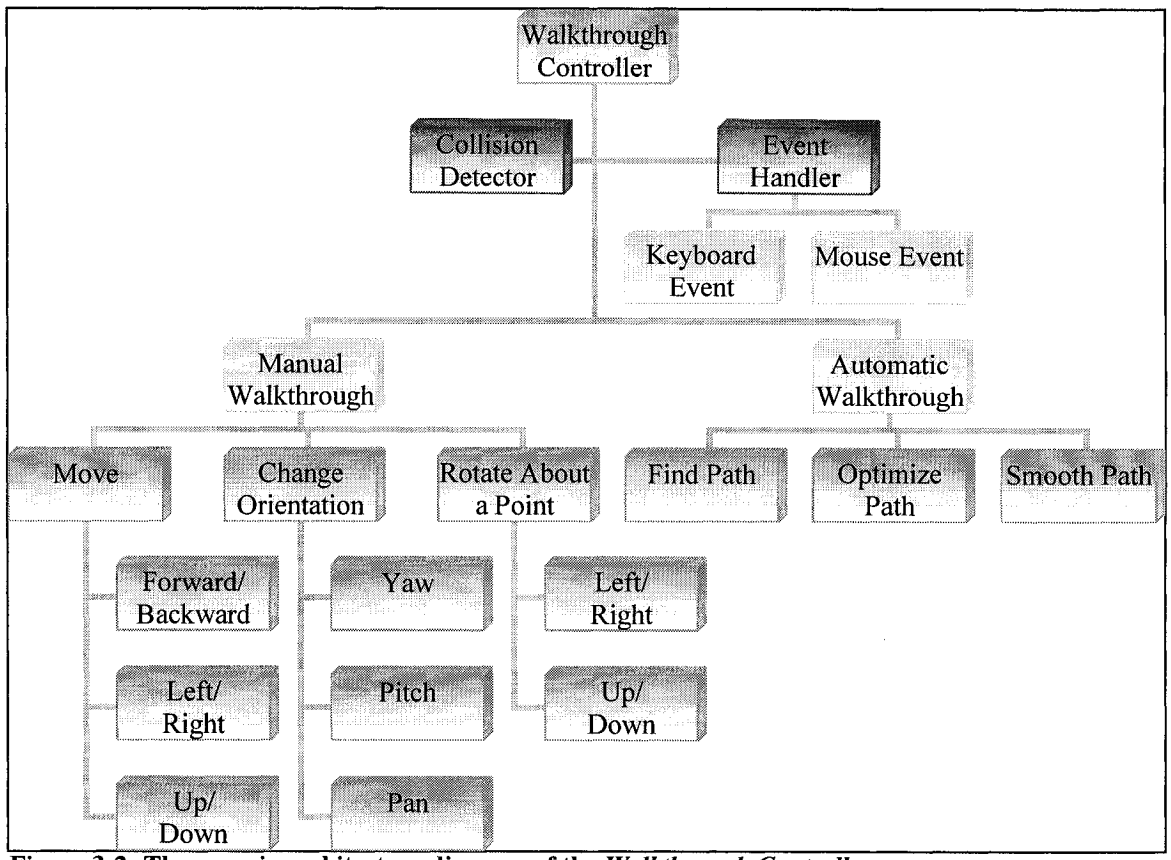

Figure 3-2: The generic architecture diagram of the *Walkthrough Controller*

3.2.1 Event Handler

An interactive mode is necessary because the *Walkthrough Controller* allows users to walkthrough inside the 3D virtual environment either manually or automatically. Consequently, the *Event Handler* accepts both keyboard and mouse events as shown in the list in Table 3-1.

Keyboard or Mouse	Functionality
f/F	Camera move forward
b/B	Camera move backward
Ctrl + [Arrow Keys]	Camera move left/right/up/down
[Arrow Keys]	Camera yaw or pitch

Page Up/Page Down	Camera roll
Shift + [Arrow Keys]	Camera rotate about a point
F4 – F8	Set camera to a specified position (e.g. top or front etc.)
Esc	Exit
Mouse click on the floor plan	Camera move automatically to the target position

Table 3-1: Keyboard and mouse events

3.2.2 Collision Detector

Without any additional restrictions, the camera passes straight through any object in a virtual world. This, however, doesn't make any sense in the real world. Therefore, it is essential to avoid obstacles so that the simulation will be much more realistic. This requires that the program be aware of all the objects' positions in a virtual scene; it can do this by collecting all bounding box information. Then, before a single movement either manual or automatic, the *Collision Detector* determines if the next target position of camera movement can be accessed along with an arbitrary path.

3.2.3 Manual Walkthrough

The *Manual Walkthrough* contains the basic controls for camera movements so that users are able to control navigation in the 3D virtual environment. The basic controls (see Figure 3-3) include move forward/backward/left/right/up/down, pitch/yaw/roll, and rotate left/right/up/down about a point, which is considered to be the point at which the camera is currently looking.

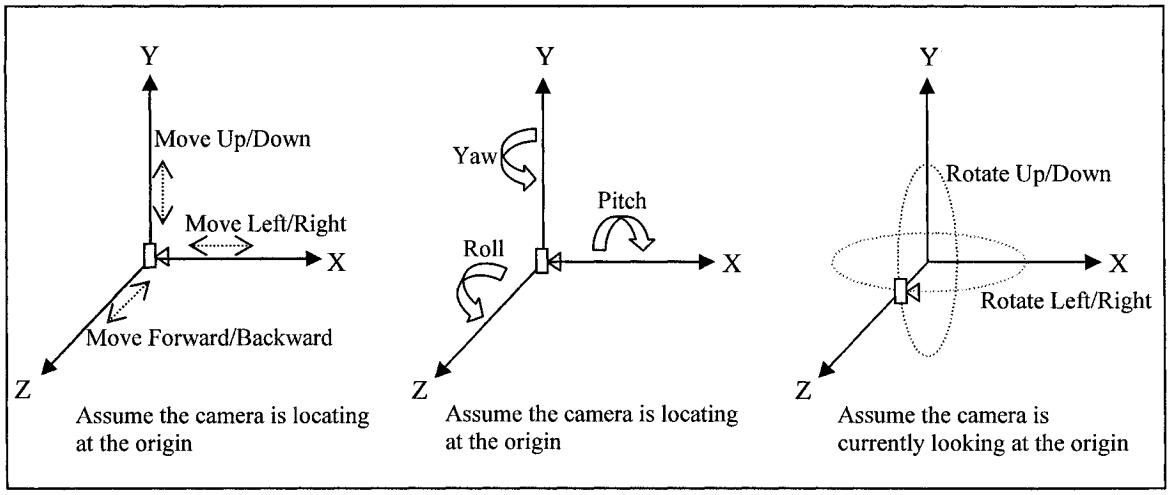

Figure 3-3: Manual Walkthrough

3.2.4 Automatic Walkthrough

Having to manually control the camera to a target position is tedious for users, especially when there are many obstacles in the way. Hence, when the user specifies a target of interest, the system automatically generates camera motions to reach that target. To start, the user indicates the target position. Given this input, the system first finds an initial camera path, and then optimizes this path by removing the redundant path nodes. Finally, the optimized path is smoothed to present users with a pleasing camera motion.

3.2.4.1 Find Path

The most important part of generating an automatic camera movement is to find a valid path from the current position of the camera to another point, called the target point, while avoiding obstacles. To do this, the program first detect if there is any collision between the camera current position and the target. If collision occurs, it finds the closest obstacle, and then tries to walk around the obstacle to approach the target. Meanwhile, the program builds up a tree diagram for the entire possible path nodes, and will look up the tree diagram until it finds a valid path. For example, as shown in Figure 3-4, we try to

go around the obstacle from its left side first. If it succeeds, we have a path: *Start* → *Left* → *Target*. Otherwise, we give up the *Left* node, and then try to reach *Target* from the *Right*. If it succeeds, we have a path: *Start* → *Right* → *Target*, otherwise the attempt to find a path fails.

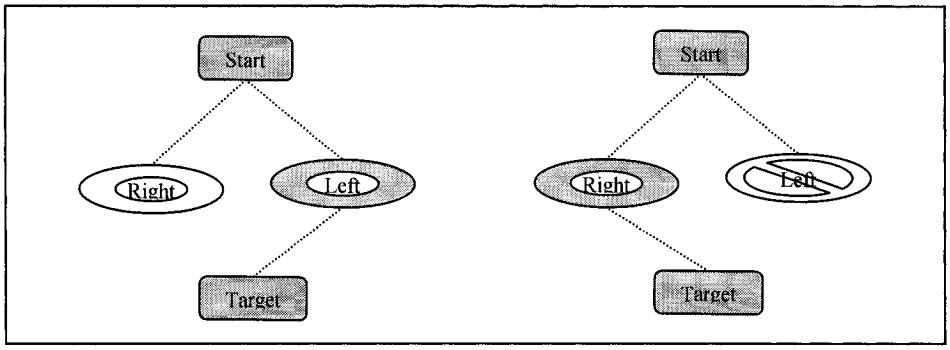

Figure 3-4: Finding a path.

3.2.4.2 Optimize Path

After finding a valid path from the start point to the destination point, optimizing the path is necessary in order both to reduce the total number of the path nodes and to make the path shorter. For example, assume we have 5 path nodes: *Pstart*, *PN1*, *PN2*, *PN3*, and *Ptarget*. If no collision occurs traveling from *Pstart* to *PN3*, *PN1* and *PN2* are redundant path nodes, which only make the path longer and less realitic. Therefore, we can remove all redundant nodes from the path, and rebuild a optimized path: *Pstart* → *PN3* → *Ptarget*.

3.2.4.3 Smooth Path

To present users with a pleasant camera movement, we finally must smooth the camera motion by keeping the camera always pointed in the direction in which it is moving, replacing sharp corners by smooth arcs, and moving the camera in a constant speed.

4 Implementation

The *3D Virtual Environment Builder* and the *Walkthrough Controller* were developed in Visual C++ with OpenGL libraries, including GL, GLU, GLUT, GLAUX, and GLUI etc.

4.1 3D Virtual Environment Builder

4.1.1 3D Virtual Environment Editor

As described previously, the *3D Virtual Environment Editor* mainly includes three types of objects: 3D models, separators, and bounding boxes.

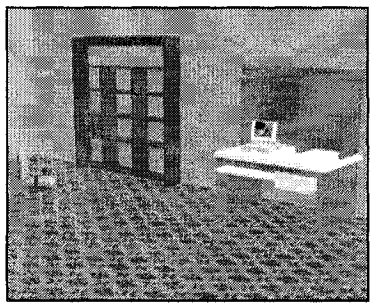

Figure 4-1: 3D models

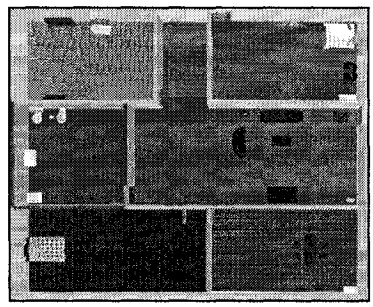

Figure 4-2: Separators

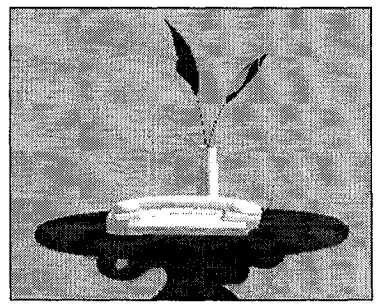

Figure 4-3: A bounding box

First of all, 3D models, which must be in 3D-Studio File Format (.3ds) [1], are used to represent any furniture in the 3D environment (e.g., desks and chairs), as shown in Figure 4-1. The *3D Virtual Environment Editor* can import 3DS files directly so that on the user does not have to spend time building models. Moreover, instead of hard coding, a file named *3d.txt* saves all of the information related to a 3D model, including the 3DS file name, scaling ratio, translation value in X, Y and Z axis, a rotation matrix, and some information related to its bounding box. Thus, we can easily adjust the size, position and orientation of every 3D model in the scene as well as its bounding box.

The data structure of *3d.txt* is defined as:

```
struct t3DInfo
{
        char   strFile[255];                          // 3DS file name
        float  scale;                                 // Scaling
        float  translateX, translateY, translateZ; // Translation
        float  rotateMatrix[16];                      // Rotation
        float  Clex, ClenY, ClenZ;        // Bounding box length
        float  CtranslateX, CtranslateY, CtranslateZ;  // BB translation
        float  CrotateMatrix[16];                     // BB rotation
}
```

The transformation matrix of a 3D model (*M3d*) is constructed as follows:

$$M3d = \begin{pmatrix} 1 & 0 & 0 & translateX \\ 0 & 1 & 0 & translateY \\ 0 & 0 & 1 & translateZ \\ 0 & 0 & 0 & 1 \end{pmatrix} \times rotationMatrix \times \begin{pmatrix} scale & 0 & 0 & 0 \\ 0 & scale & 0 & 0 \\ 0 & 0 & scale & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Secondly, separators, which are indeed made up of polygons, represent walls, floors and ceilings in the 3D virtual environment as shown in Figure 4-2. The applied adjustment for walls includes translation, rotation, length modification, and texture alteration, etc. By putting them in appropriate positions and setting different textures, walls can be used to separate and distinguish different areas (e.g., rooms and corridors). Likewise, all information about separators is saved in a file named *separator.txt* for customized subsequent adjustments.

The data structure of *separator.txt* is defined as:

```
struct dSpInfo
{
        int    textureIDTop;              // Texture ID of top
        float  repeatTop;                 // Texture repeat time of top
        int    textureIDBottom;           // Texture ID of bottom
        float  repeatBottom;              // Texture repeat time of bottom
        float  lenX, lenY, lenZ;          // Length
        float  translateX, translateY, translateZ;// Translation
        float  rotateM[16];               // Rotation
        float  color;                     // Color
        int    door;         // Door indicator of Floor Plan Generator
}
```

The transformation matrix of a wall (*Msp*) is constructed as below:

$$Msp = \begin{pmatrix} 1 & 0 & 0 & translateX \\ 0 & 1 & 0 & translateY \\ 0 & 0 & 1 & translateZ \\ 0 & 0 & 0 & 1 \end{pmatrix} \times rotationMatrix$$

The last object, bounding box is an area in the form of rectangle enclosing each object in the 3D virtual environment as shown in Figure 4-3. In other words, each object in this 3D virtual environment has its own bounding box. A 3D model's bounding box can be customized by resizing as well as repositioning, but a separator's bounding box is fixed by programmatically expanding certain distance according to the separator's own size. Details of the bounding box will be covered in a later section, *Camera Controlled Movement: Collision Detection.*

Furthermore, the *3D Virtual Environment Editor* provides a friendly user interface (see Figure 4-4) for the previously mentioned customized adjustments, which includes modifying any object's scaling, translation, and rotation in the 3D world coordinates, varying certain object's length, and altering certain object's texture or color, etc.

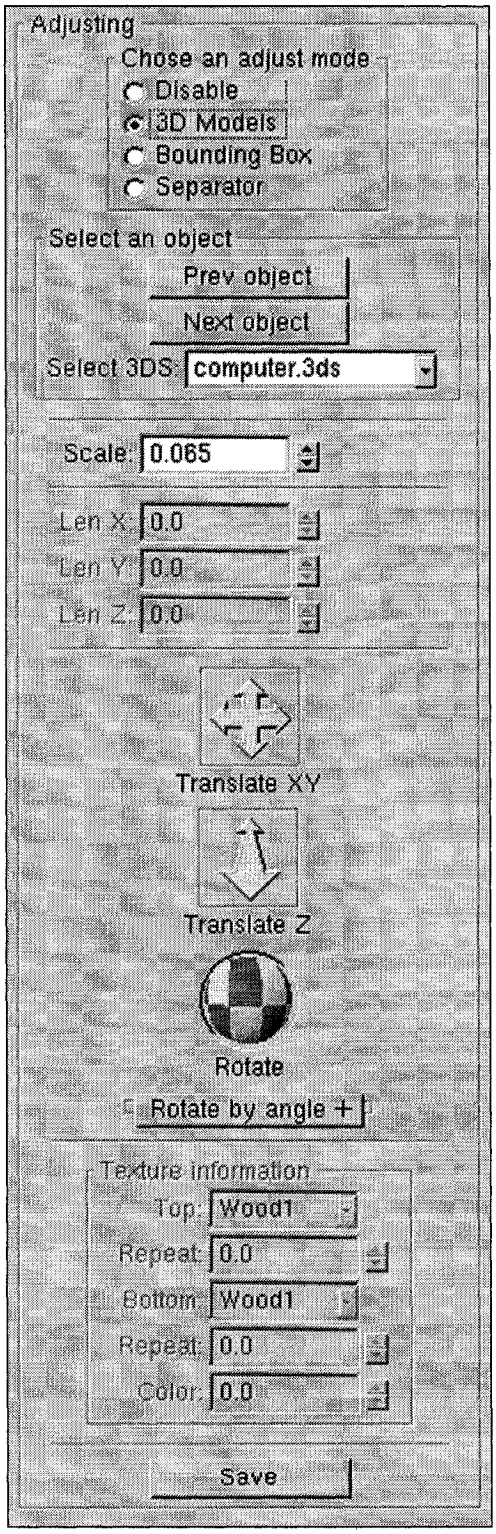

Figure 4-4: The *3D Virtual Environment Editor*

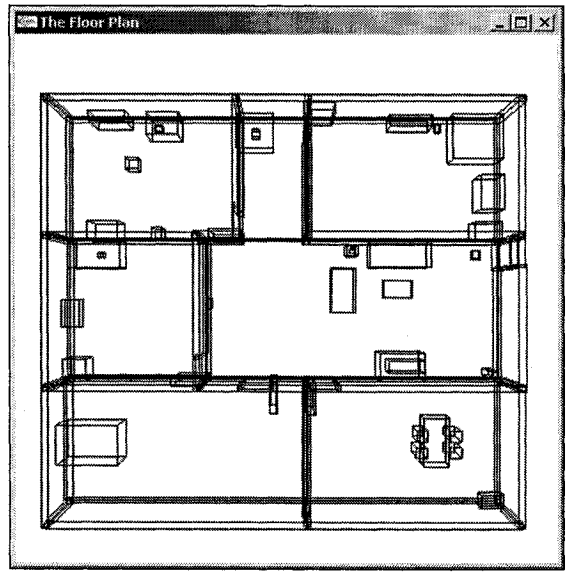

Figure 4-5: A generated floor plan

4.1.2 Floor Plan Processor

4.1.2.1 Floor Plan Generator

A floor plan is generated by collecting information about all objects' bounding boxes. For instance, the transformation matrices of a 3D model bounding box (*M3dB*) and a separator bounding box (*MspB*) are respectively constructed as:

$$M3dB = M3d \times \begin{pmatrix} 1 & 0 & 0 & CtranslateX \\ 0 & 1 & 0 & CtranslateY \\ 0 & 0 & 1 & CtranslateZ \\ 0 & 0 & 0 & 1 \end{pmatrix} \times CrotationMatrix$$

$$MspB = Msp$$

After constructing the transformation matrices above, the *Floor Plan Generator* is able to draw all the objects' bounding box with their appropriate length at the correct position. The length of a 3D model's bounding box (*len3dB*) is given in the file of 3D models (*3d.txt*), and the length of a separator (*lenSpB*) is generated programmatically by extending the separator's length, which is given in the file of the separator (*separator.txt*), with a certain distance. Therefore,

$$len3dB = (ClenX, ClenY, ClenZ)$$

$$lenSpB = (lenX + extend, lenY + extend, lenZ + extend)$$

Figure 4-5 is a generated floor plan of the 3D virtual environment shown in Figure 4-2. Note that the floor plan indicates the position of the door of each room. In the screen view, doors are shown in red.

4.1.2.2 Coordinates Converter

When users click on the floor plan, the *Walkthrough Controller* will move the camera move smoothly from the current position to the target position automatically. To be aware of the target position, the *Coordinates Converter* needs to convert the clicked position from the windows pixel coordinates to the 3D world coordinates. The procedure for coordinate conversion is shown in Figure 4-6.

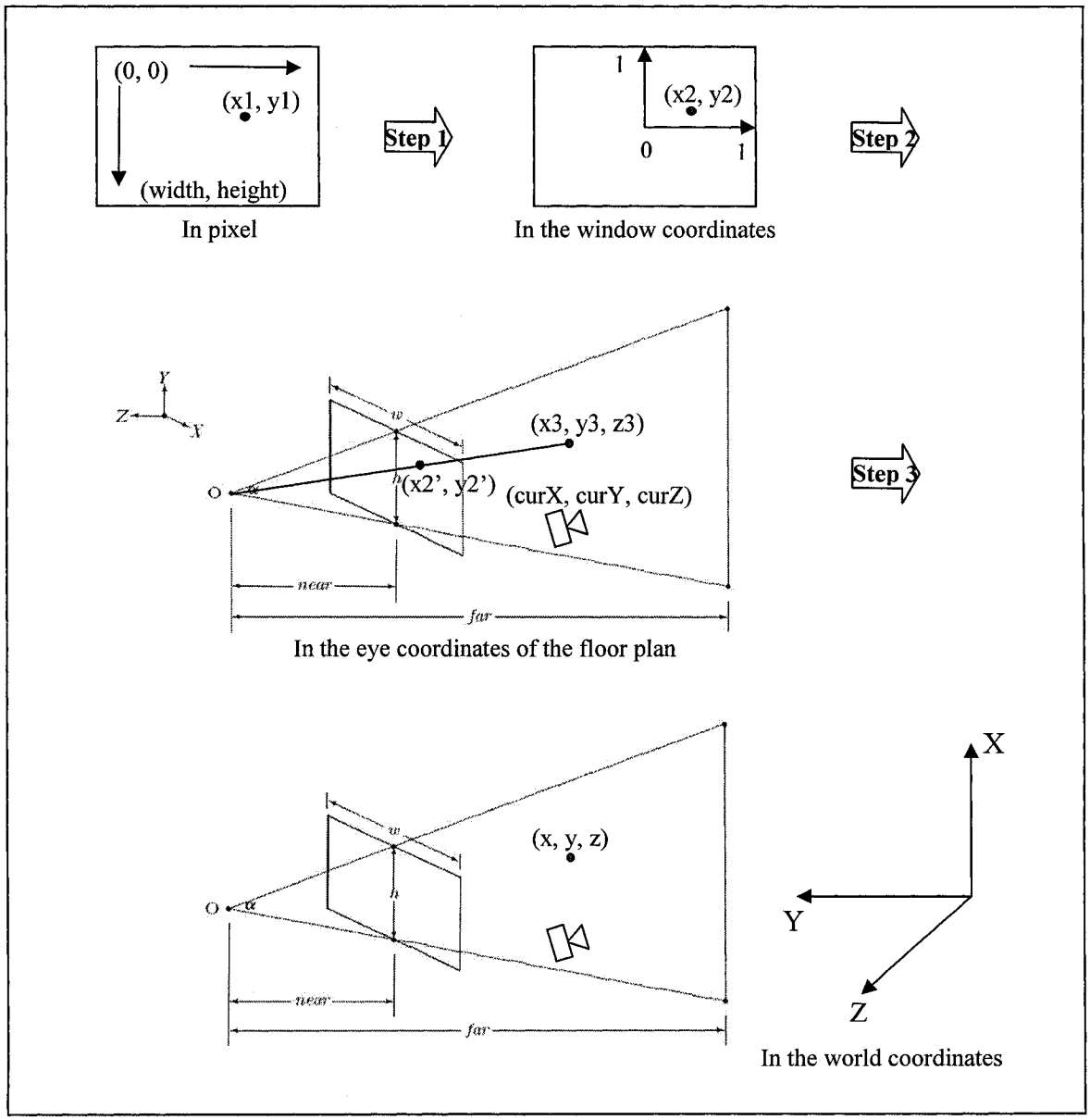

Figure 4-6: Coordinate conversion

Step 1: Once a mouse-click event occurs, the *Coordinates Converter* gets the clicked position in windows pixel coordinates. The first step is to convert the position from windows pixel coordinates *(x1, y1)* to windows coordinates *(x2, y2)*, given by:

$$x2 = \frac{x1 - width/2}{width/2} \qquad y2 = \frac{height/2 - y1}{height/2}$$

where *width* and *height* are the actual window size.

Step 2: The next step is to convert from window coordinates to eye coordinates. First, we need to know *(x2', y2')*, which is the value of *(x2, y2)* in the eye coordinates:

$$\frac{x2'}{w/2} = \frac{x2}{1} \Rightarrow x2' = \frac{x2 \times w}{2} \qquad \frac{y2'}{h/2} = \frac{y2}{1} \Rightarrow y2' = \frac{y2 \times h}{2}$$

Then, the *Coordinates Converter* gets the *z* component from the current camera position *(curX, curY, curZ)* in the eye coordinates, and then convert the *x* and *z* component of the clicked position from 3D window coordinates *(x2, y2)* in 2D to eye coordinates *(x3, y3, z3)* in 3D.

$$z3 = curZ$$

$$\frac{y3}{z3} = \frac{y2'}{near} \Rightarrow y3 = \frac{z3 \times y2 \times h}{2 \times near}$$

$$\frac{x3}{z3} = \frac{x2'}{near} \Rightarrow x3 = \frac{z3 \times x2 \times w}{2 \times near}$$

where *curZ* is the *z* component of the current camera position in the eye coordinates.

Step 3: The last step is to transform from the eye coordinates to the world coordinates. In this step, the *Coordinates Converter* simply gets the model-view matrix by calling a GL function as follows:

```
glGetFloatv(GL_MODELVIEW_MATRIX, transM.m);
```

Assume that *(x, y, z)* is the clicked position in the world coordinates, so:

$$transM.m \times \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x3 \\ y3 \\ z3 \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \frac{\begin{pmatrix} x3 \\ y3 \\ z3 \\ 1 \end{pmatrix}}{transM.m}$$

The position in the world coordinates can be consider as transforming *(x3, y3, z4)* by the inverted matrix [2] of *transM.m*. Consequently, the final form of the equation is:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = Inverse(transM.m) \times \begin{pmatrix} x3 \\ y3 \\ z3 \\ 1 \end{pmatrix}$$

4.2 Walkthrough Controller

4.2.1 Event Handler

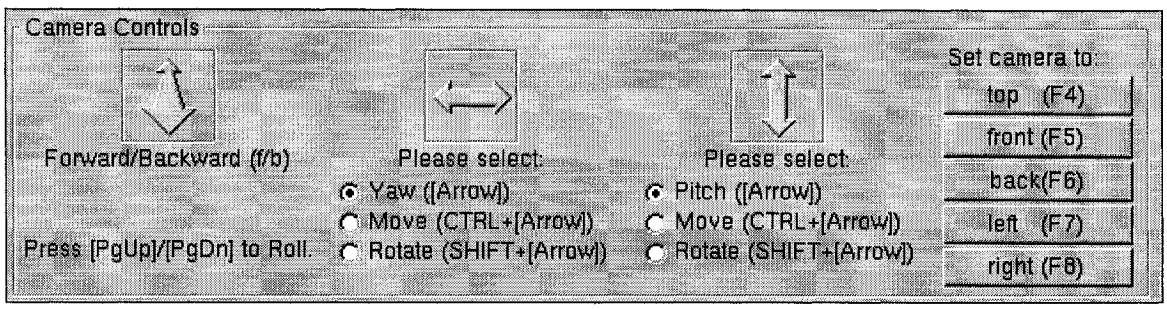

Figure 4-7: The *Walkthrough Controller* UI

Table 3.1 in section 3.2.1 lists all of the acceptable keyboard and mouse events. However, a good interactive system does not require users to remember the relationship of the keys and functions. Therefore, a friendly user interface of the *Walkthrough*

Controller is provided, as shown in Figure 4-7. It includes all the camera controls and provides hint for the corresponding hot keys.

4.2.2 Collision Detector

Collision detection is critically important in walkthrough simulation. As we described previously, there are various different shape objects in a 3D scene; therefore, each object is enclosed by a bounding box. In our case, we have chosen to use *axis-aligned bounding boxes* (AABBs) to bound objects and to use them in a method checking for collisions between such a box and the camera. The collision detection is accomplished by using a ray/polygon intersection check. Note that only bounding boxes can be embedded in a method but not an object because the object information comes from a specified data as we describe in *4.1.1 3D Virtual Environment Editor*.

The reason for choosing AABBs is that the geometric attributes used in the intersection calculation do not change. The AABBs is defined by its minimum and maximum points (6 floats), face normals (6), and face vertices (4 vertices of 6 faces) as follows:

```
struct Vertices
{
      float        x1;          // Minimum x
      float        x2;          // Maximum x
      float        y1;          // Minimum y
      float        y2;          // Maximum y
      float        z1;          // Minimum z
      float        z2;          // Maximum z
};

struct AABBs
{
      Vertices     vertex;
      Coordinate   faceNormal[6];      // The normal of 6 faces.
      Coordinate   faceVertex[6][4];   // Four vertices of 6 faces.
}
```

Figure 4-8 shows an AABBs bounding box and details of its definition. There are six faces in total: front (0), right (1), back (2), left (3), top (4) and bottom (5). In addition, the order to travel the vertices of each face is counter-clockwise (left-top, left-bottom, right-bottom, and right-top).

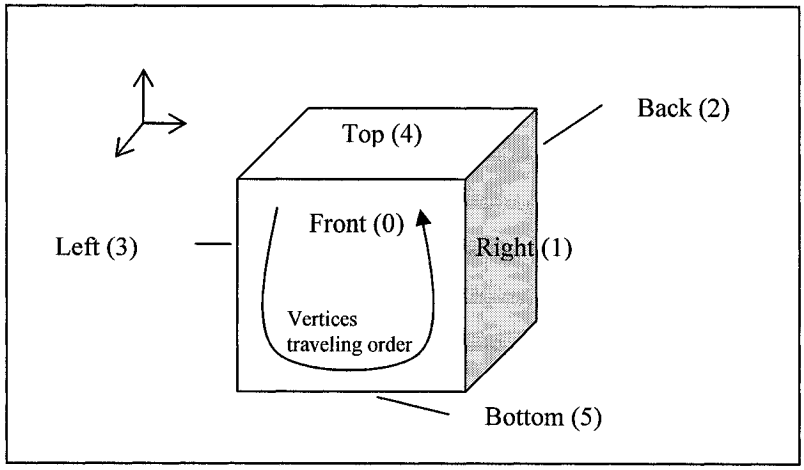

Figure 4-8: An AABBs definition.

In the first stages of the project, we assumed that the camera is a point as shown in Figure 4-9, like a bug flying through a building. To provide a more realistic simulation of walking through, the camera is currently modeled by a segment of a given certain height as shown in Figure 4-10 to prevent the camera from giving the appearance of flying over obstacles.

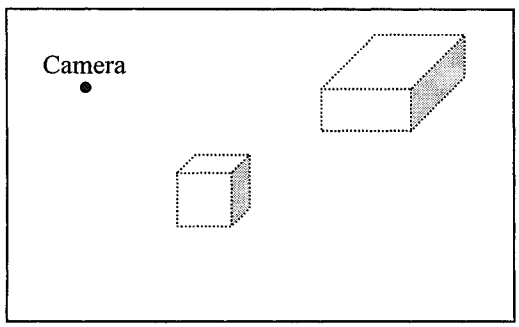

Figure 4-9: Assume camera as a point.

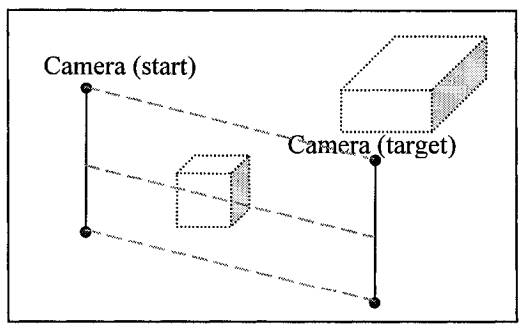

Figure 4-10: Assume camera as a segment.

In the current camera setting, when camera movement occurs, we check to see if the camera collides with any obstacle by using a ray/polygon intersection check.

However, it would not be accurate if we checked only the ray/polygon intersection with rays of the two vertices of the camera segment. Therefore, we check also the ray of one of the mid-points of the camera segment as shown in Figure 4-10. If more accuracy is required, more intermediate points will need to be involved in the computation, which will then require longer to execute.

To reduce the computation to a minimum during movement, we check only the obstacles that could actually generate a collision [3]. We start by creating an Active Area that will enclose the end points (*start* and *target*) of the camera segment as shown in Figure 4-11.

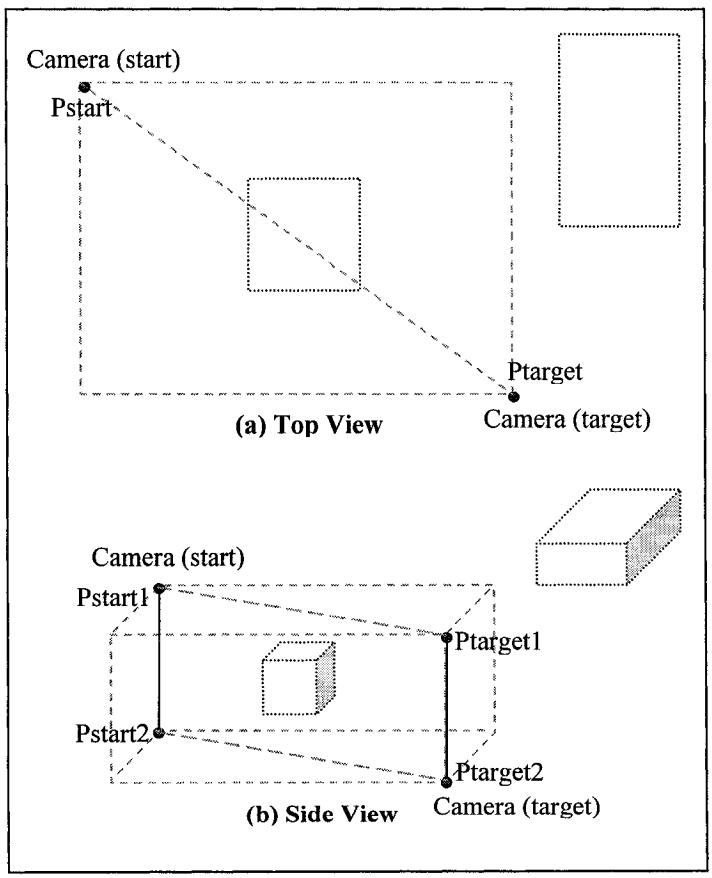

Figure 4-11: The box of Active Area.

$\overline{Pstart1Ptarget1}$ and $\overline{Pstart2Ptarget2}$ are the diagonals of the top face and bottom face of the Active Range box as shown in Figure 4-11 (b). The height of the camera

$\overline{Pstart1Pstart2}$ (or $\overline{Ptarget1Ptarget2}$) is the height of the Active Area box. According to each obstacles position, the *Collision Detector* only checks the obstacles that locate inside the Active Area for collision detection so that the program will be more efficient.

To detect if collision occurs, the program computes the intersection distance from the ray defined by camera *start* and *target* position against all face planes of the obstacles inside the temporary range. If distance is negative, it returns no intersection. Otherwise, it computes the intersection point using the distance check if the intersection point is inside the polygon. If so, it returns the intersection of the closest colliding obstacle.

4.2.3 Manual Walkthrough

When camera movements occur, the *Manual Walkthrough* method finds the appropriate values of the parameters for the OpenGL function *gluLookAt()*, which are *eye* position, *at* position, and *up* vector, to place the camera with an appropriate position and orientation.

Because transforming a point (e.g., *eye* or *at* position) to a certain place involves some matrix calculations, first of all, a Matrix class was built to implement most of the matrix manipulations, for instance, loading an identity, translation, scaling, or rotation matrix, multiply two matrices, and inverting a matrix, etc. Then, in the Camera class, we linearly move the *eye* and *at* position along X, Y, or Z axis when moving left/right, up/down, or forward/backward; rotate and translate the *at* position and the *up* vector correspondingly when doing pitch/yaw/roll; rotate and translate the *eye* position and the *up* vector when turning left/right or up/down. The basic functions of the camera class are listed below:

```
// Camera look-at.
void lookAt(void);

// Camera move along X (left/right)
void Camera::moveX(float direct);

// Camera move along Y (up/down)
void Camera::moveY(float direct);

// Camera move along Z (forward/backward)
void Camera::moveZ(float direct);

// Camera pitch about a unit vector (vx, vy, vz)
void Camera::pitch(float direct, float vx, float vy, float vz);

// Camera yaw about a unit vector (vx, vy, vz)
void Camera::yaw(float direct, float vx, float vy, float vz);

// Camera roll about a unit vector (vx, vy, vz)
void Camera::roll(float direct, float vx, float vy, float vz);

// Camera roate (left/right) along a unit vector (vx, vy, vz) about a point P
void Camera::turnX(float direct, float vx, float vy, float vz, Coordinate p);

// Camera roate (up/down) along a unit vector (vx, vy, vz) about a point P
void Camera::turnY(float direct, float vx, float vy, float vz, Coordinate p);
```

As an example, the following shows the method for camera rolling:

```
void Camera::roll(float direct, float vx, float vy, float vz)
{
        float angle = stepAngle * direct;

        // (vx, vy, vz) is a unit vector that the camera rolls about.
        transM.loadRotateMatrix(angle, vx, vy, vz);

        // camY is the current actual UP vector of the camera.
        up = transM.transformPoint(camY);
}
```

To obtain a realistic effect during a walkthrough simulation, the Camera class should always keep the *up* vector of the camera appearing to point upwards in the view. For instance, even if the camera currently is looking up or down, moving up/down or forward/backward should always be horizontal instead of along with the Y or Z axis of the eye coordinates. See the case of forward in the following figure (Assume X-axis of the world coordinate is the floor level).

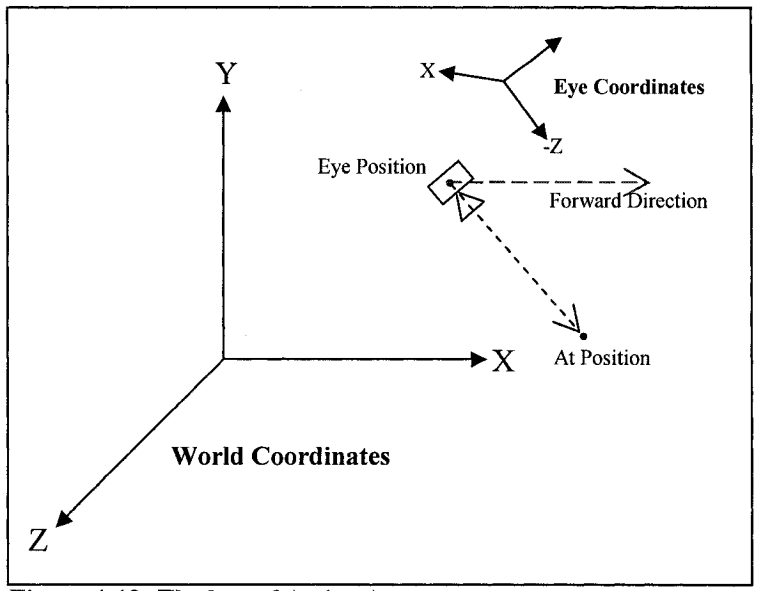

Figure 4-12: The box of Active Area.

Take yawing as another example. In walkthrough simulation cases, the ideal yawing motion is obtained by rotating the *at* position and the *up* vector about the Y-axis of the world coordinates rather than the eye coordinates. Otherwise, after some combined motions of pitch and yaw, the camera will be out of the horizontal, which is less realistic because people generally keep their eyes in a horizontal plane however they move their head.

Therefore, the solution is that in walkthrough simulation, some movements are still relative to axes of the eye coordinates, but some movements need to be with respect to the axes of the world coordinates. Consequently, one of the nice things about our Camera class is that it does this calculation ("twisting" the camera along the line of sight) automatically.

4.2.4 Automatic Walkthrough

4.2.4.1 Find Path

We apply four principal rules on finding a valid path in order to cover most of the cases in a virtual environment built by the *3D Virtual Environment Editor*. As described previously, all objects are enclosed with a bounding box; hence, we only have cuboidal obstacles in the 3D dimension and rectangular obstacles from the top view. Assuming that the camera keeps its height constantly during an automatic walkthrough, we consider only two dimensions (top view) while finding the path and will take the third dimension (height) into account for collision detection while the camera is moving. The following illustrates how the four principal rules for a path work, and we illustrate them with some specific cases.

Rule #1: Intersection

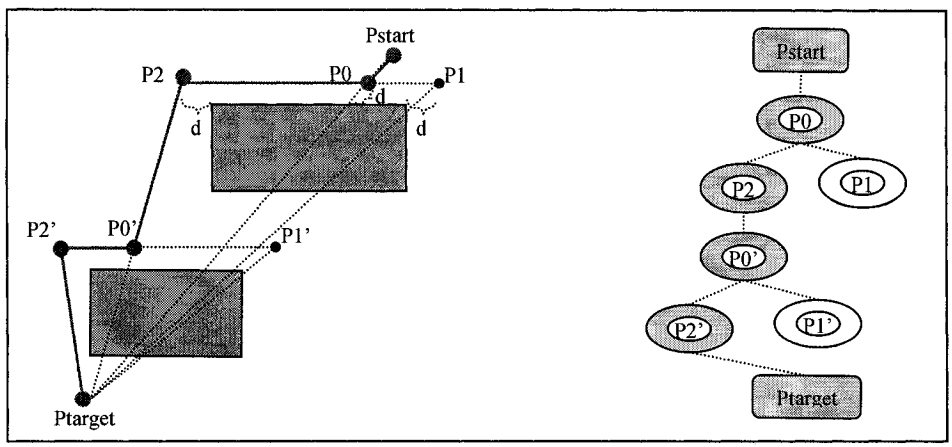

Figure 4-13: Rule #1: Intersection.

(1) Find Point *P0* by moving the intersection of segment $\overline{PstartPtarget}$ with a distance *d* towards *Pstart*.

(2) Find Points *P1* and *P2* by traveling parallel along border of the closest obstacle in both directions.

(3) Between P1 and P2, pick one of them to be a path node if and only if:

- it does not have any intersection with the same obstacle while connecting itself and *Ptarget*;
- it does not have any intersection with any obstacle while connecting itself and *P0*.

Note that *P1* will be first checked; *P2* won't be considered anymore if *P1* is valid to be a path node.

(4) Set the newest path node as *Pstart*, and then go back to step (1) until the path meet the destination *Ptarget*.

Rule #2: Around Obstacle

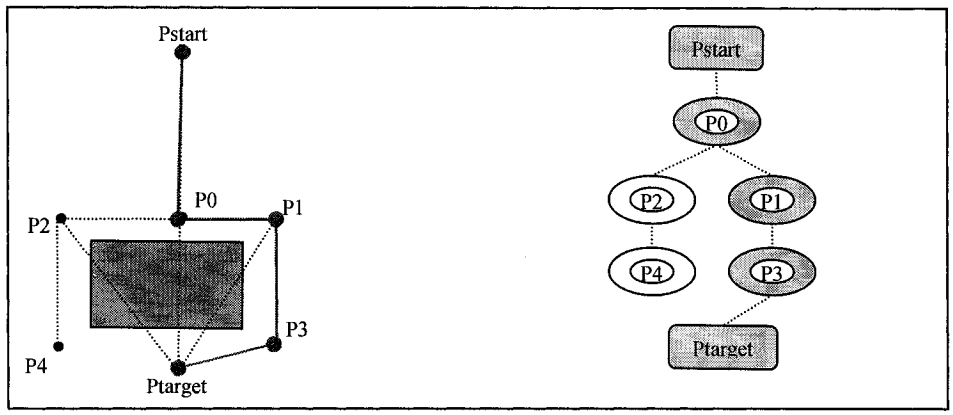

Figure 4-14: Rule #2: Around Obstacle.

(1) By applying Rule #1, we find *P0*, *P1*, and *P2*. If neither of *P1* and *P2* is valid to be a path node, find *P3* and *P4* by traveling parallel along the two neighboring borders of the current obstacle.

(2) Chose one of them to be a path node by checking for collision detection. Note that *P4* won't be checked if *P3* is valid.

Rule #3: Recursive Sub-path

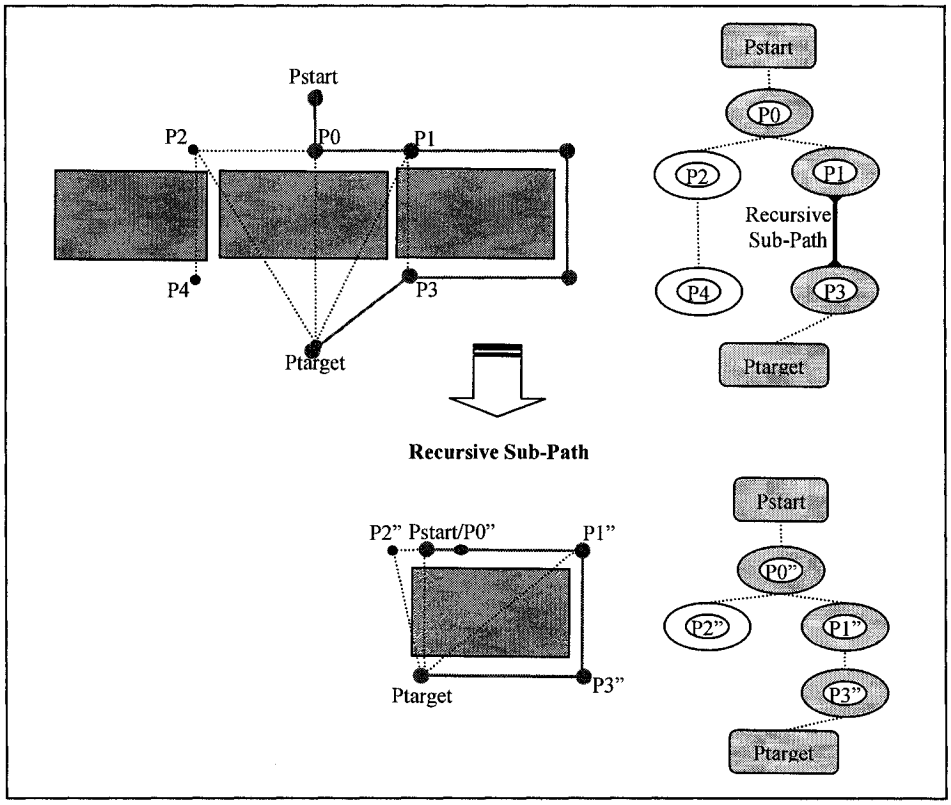

Figure 4-15: Rule #3: Recursive Sub-path.

(1) With Rule #1 and Rule #2, we find *P0*, *P1*, *P2*, *P3*, and *P4*.

(2) However, both path *P1* → *P3* and path *P2* → *P4* have collision with other obstacles. We will find a sub-path from one of those two paths by recursively calling the Find Path process. Note that *P2* → *P4* won't be checked if *P1* → *P3* is a valid path.

(3) By applying Rule #1 and Rule #2, we find a sub-path *P0"* → *P1"* → *P3"* that leads to the goal.

Rule #4: Eliminated Point

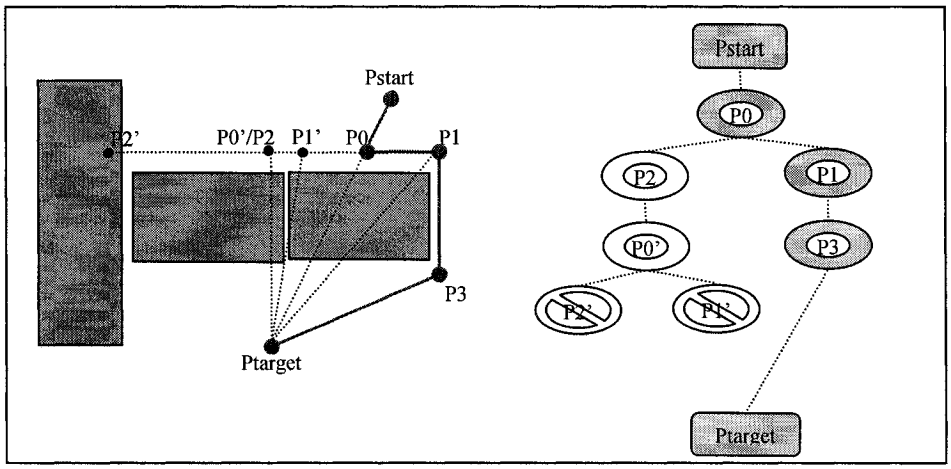

Figure 4-16: Rule #4: Eliminated Point.

(1) By applying Rule #1, chose *P2* to have a path *Pstart* → *P0* → *P2 / P0'*, and then, find *P1'* and *P2'* at *P0'*.

(2) If, after choosing *P1'*, we find out that *P1'* → *Ptarget* collides with the same obstacle again, then path finding will stuck in a loop with these two objects. The solution is to eliminate *P1'*.

(3) *P2'* may not be accessible because it is located inside an object. In this case, *P2'* is eliminated.

(4) From the path node tree on the right in Figure 4-15, we can see the path is not able to reach the destination via the branch of *P2*. Consequently, we give up the *P2* branch, and go back to *P1*.

(5) With Rule #2, we find *P3*, which can finally reach *Ptarget*.

Note that an eliminated point can no longer be used with any rules.

With all four principal rules, we can find a valid path in most of the situations. Note that we never apply these four rules randomly, instead, we use them in order. For example, Rule #1 is always the first rule to use for path finding, and Rule #2 is used only if there is no valid path found with Rule #1. Likewise, Rule #3 is used only if both Rule

#1 and Rule #2 don't work and so on. Figure 3-17 shows a sample for finding a valid path from *Pstart* to *Ptarget* with Rule #1, Rule #2, and Rule #3.

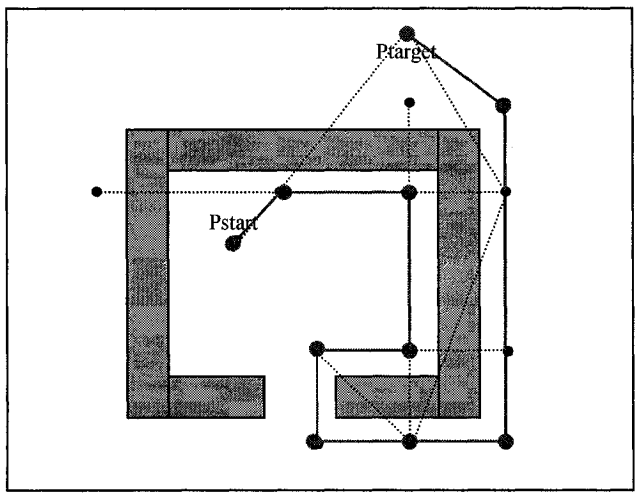

Figure 4-17: Find a valid path with Rule #1, #2, and #3.

4.2.4.2 Optimizing the Path

After finding a valid path from the start point to the destination point, optimizing the path is necessary in order to reduce the total amount of the path nodes and make the path nicer. The optimizing solution is to remove the redundant path nodes.

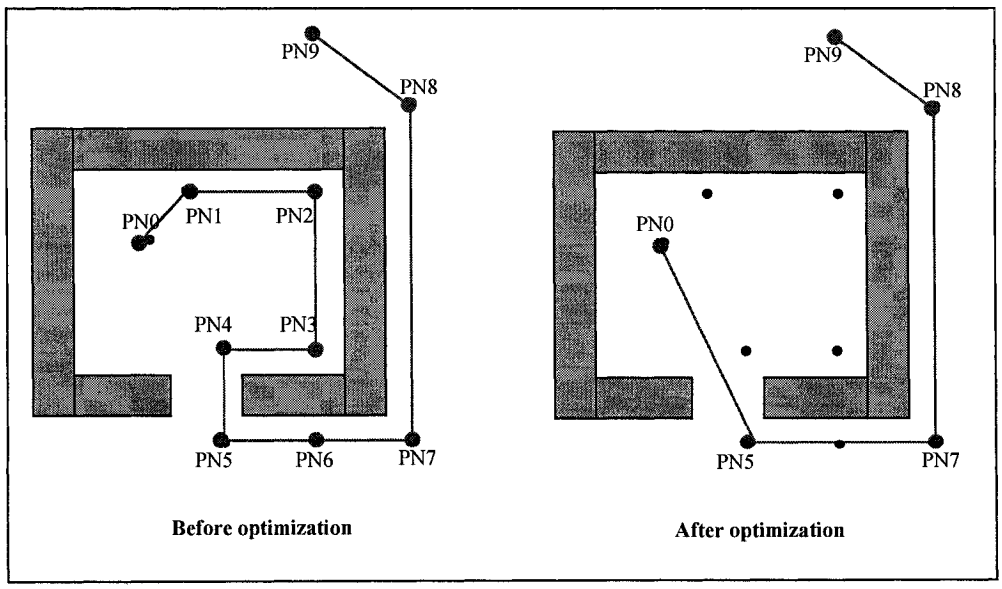

Figure 4-18: Optimizing a path.

In Figure 4-18 (same example as Figure 4-17), there are a set of path nodes *PN0*, *PN1*, … , *PN9*. The optimizing process is shown as following:

(1) Begin with *PN0*.

(2) Check the segment from this point to one of other points (in reverse order, *PN9*, *PN8*, *PN7*…) for collision detection.

(3) If only the segment $\overline{PN_nPN_{n+1}}$ doesn't have any collision occur, go back to step (2) with PN_{n+1} until the last path node.

(4) If one of other segments $\overline{PN_nPN_m}$ also doesn't cause any collision, remove the path node between *n+1* and *m-1*. For example, see Figure 3-9, no collision occurs with segment $\overline{PN0PN5}$, thus *PN1*, *PN2*, *PN3*, and *PN4* are removed from the path node set.

(5) Go back to step (2) with PN_m until the last path node.

4.2.4.3 Smooth Path

We have three processes for path smoothing: Camera Orientation, Corners with Circular Arcs, and Constant Speed.

Process #1: Camera Orientation

The goal of this process is to set the camera orientation to where it is going. Let's say the camera is currently looking at a point *Pat*, so *Pat* has to move to a desirable position, which is a point that aligns with the next path node. Note that either $\overrightarrow{PN0Pat}$ or $\overrightarrow{PN0Pat'}$ is a vector that shows the direction of the camera currently looking at. Thus, when the orientation is changing, the *Pat* is always turning about the

current camera position along the circle with a certain radius as shown in Figure 4-19, in which the arrows show the camera orientation.

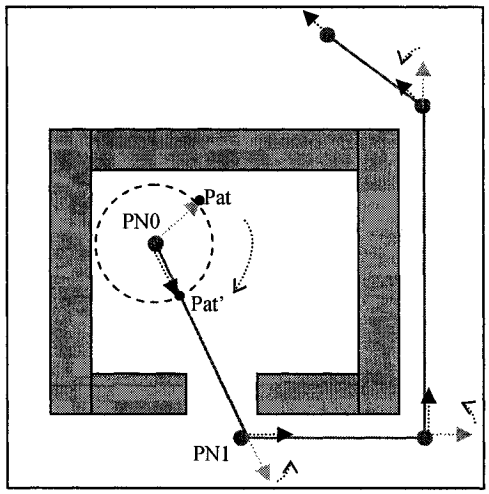

Figure 4-19: Change the camera orientation.

The first step is to find the equation of the straight line between the point of the camera current position and the point of the next path node position (which is the camera orientation target). For instance, in Figure 4-18, we will begin with finding the equation of $\overline{PN0PN1}$. The equation is represented as:

$$y = mx + b \Rightarrow mx - y + b = 0$$

where $m = \frac{y1 - y0}{x1 - x0}$, which is the slope of the straight line.

Then, according to the slope value and the relative position of where the camera currently looking at, we can decide the camera should change its orientation clockwise or counter-clockwise. The following figures list all the cases:

- When having positive slope ($m>0$), *Pat* is:

  (1) on the left side of the line if $mx-y+b < 0$;

  (2) on the right side of the line if $mx-y+b > 0$;

  (3) on the line if $mx-y+b = 0$.

Therefore, the camera will adjust its orientation correspondingly as shown as follows:

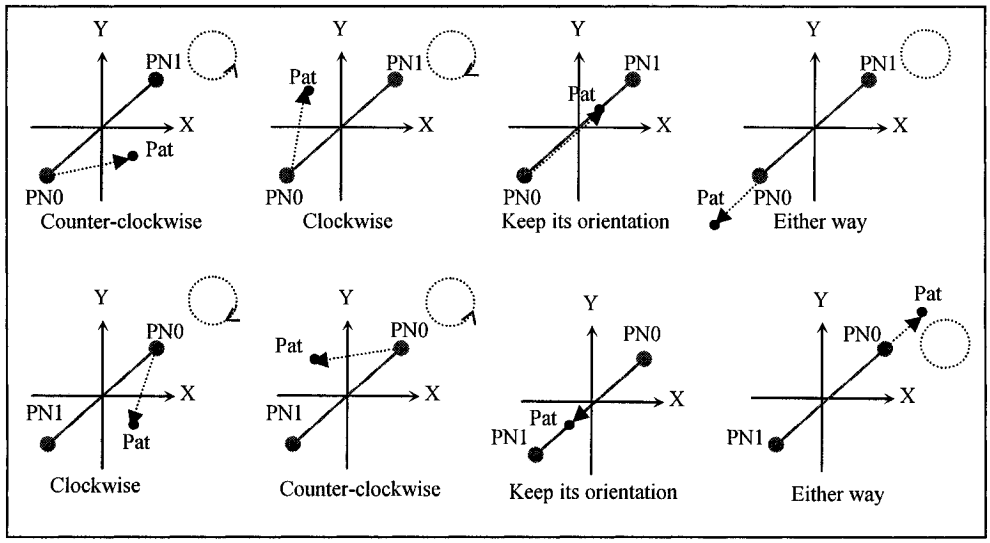

Figure 4-20: Change the camera orientation while having positive slope.

- When having negative slope ($m<0$), *Pat* is:

  (1) on the left side of the line if $mx-y+b < 0$;

  (2) on the right side of the line if $mx-y+b > 0$;

  (3) on the line if $mx-y+b = 0$.

  Therefore, the camera will adjust its orientation correspondingly as shown as follows:

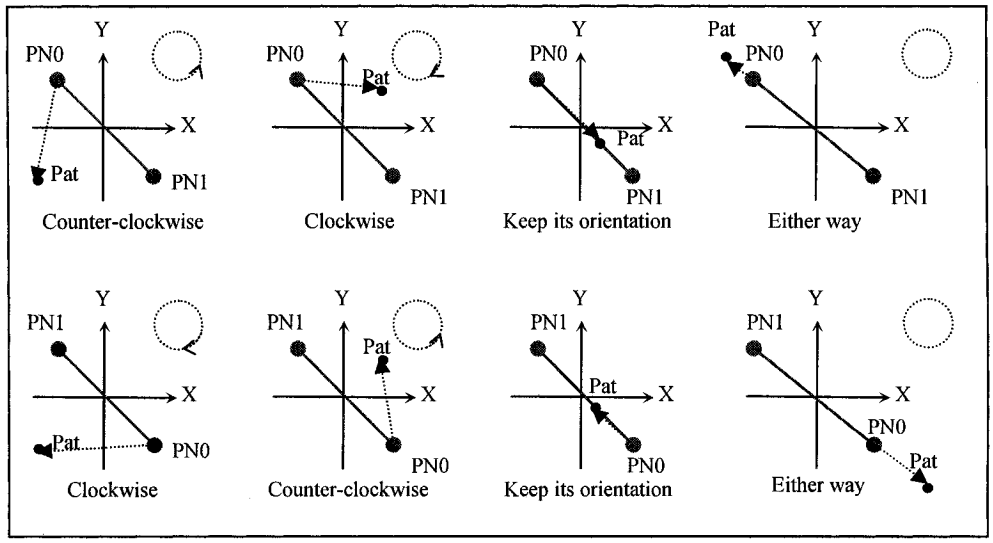

Figure 4-21: Change the camera orientation while having negative slope.

- When having zero slope (*m=0*), *Pat* is:

  (1) above the line if *mx-y+b > 0*;

  (2) below the line if *mx-y+b < 0*;

  (3) on the line if *mx-y+b = 0*.

  Therefore, the camera will adjust its orientation correspondingly as shown as follows:

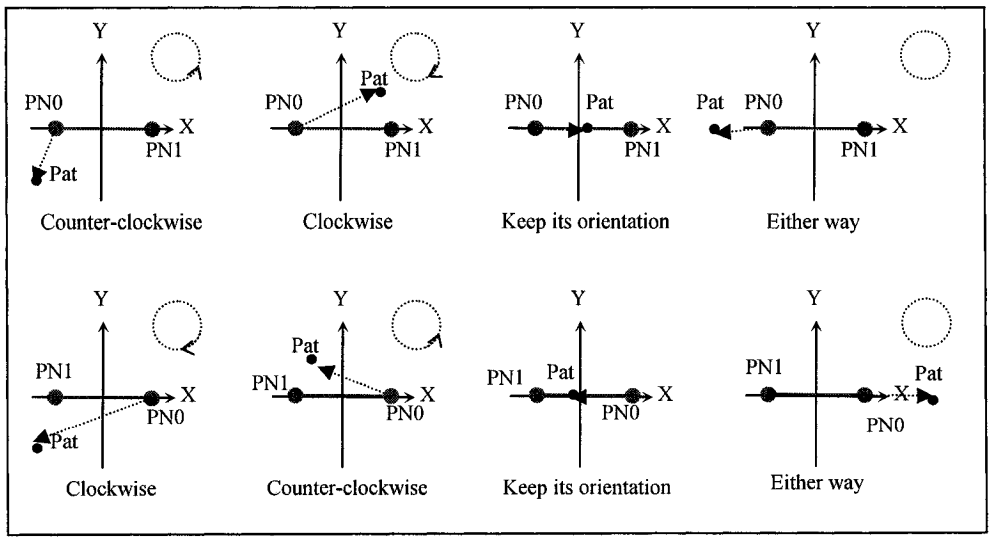

Figure 4-22: Change the camera orientation while having zero slope.

- When having no slope (i.e., infinite slope), *Pat* is:

(1) on the left side of the line if $mx-y+b > 0$;

(2) on the right side of the line if $mx-y+b < 0$;

(3) on the line if $mx-y+b = 0$.

Therefore, the camera will adjust its orientation correspondingly as shown as follows:

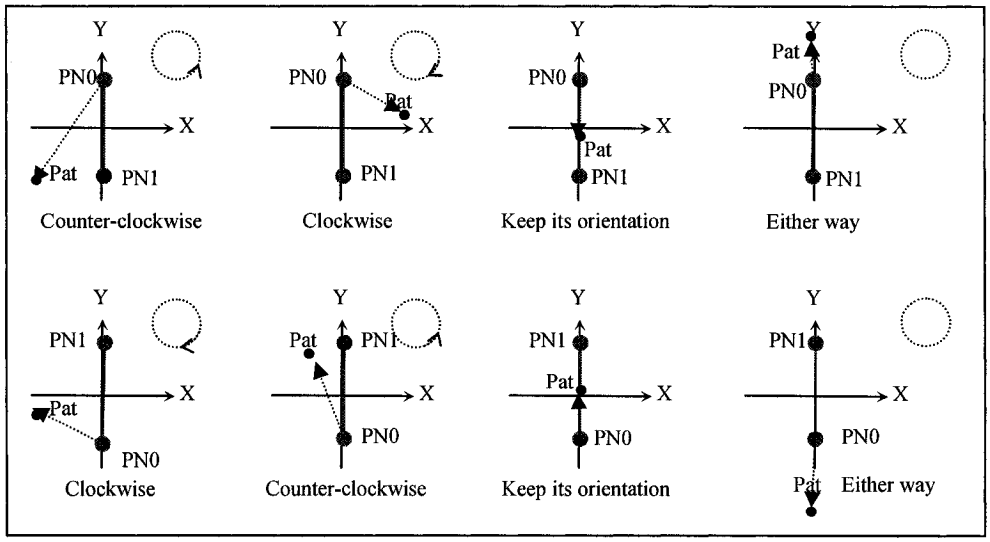

Figure 4-23: Change the camera orientation while having infinite slope.

Process #2: Corners with Circular Arcs

In this process, we are willing to give circular arcs at the corner so that users will have a pleasing navigation with the automatic smoothed camera movements. Notice that we are not giving a curve because collision might happen with curves as shown in the dashed line of Figure 4-24 (a). Hence, we only put a circular arc at the corner part of the path and the rest of the path keep in straight line as shown in Figure 4-24 (b).

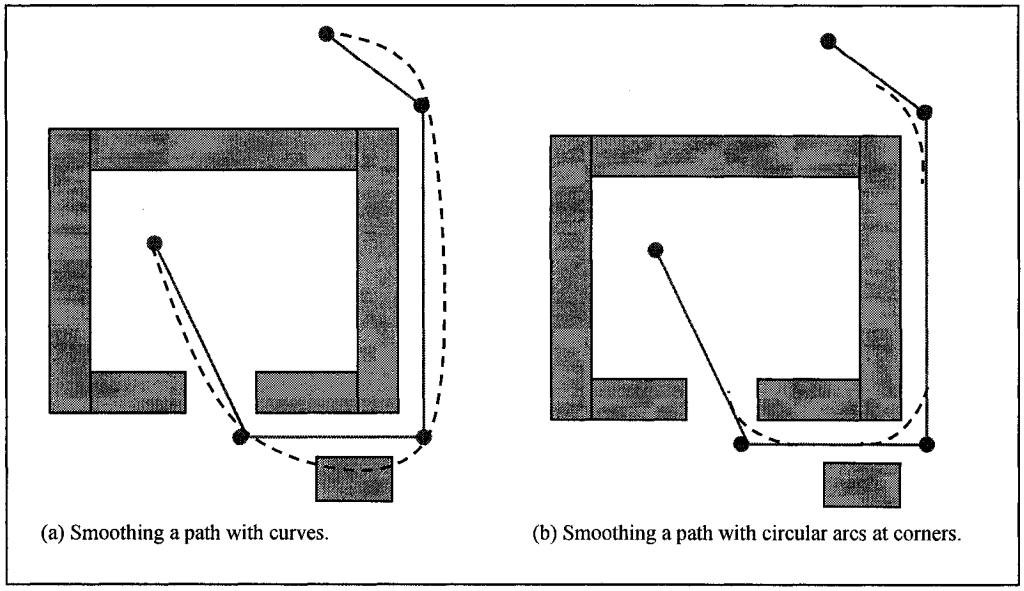

Figure 4-24: Smoothing a path with curves in (a) or circular arcs in (b).

The following figure, Figure 4-25, shows how to find the circular arc at a corner.

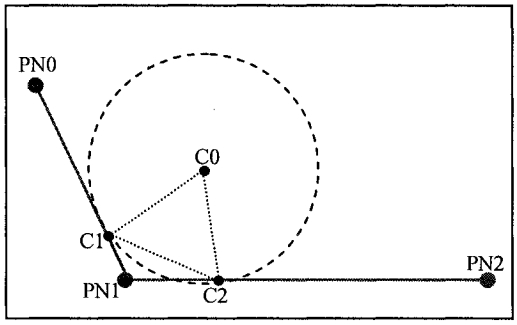

Figure 4-25: The circular arc $\widetilde{C1C2}$ at a corner.

Assume that we set $t = 0.2$. We have:

$$C1 = PN0 + (PN1 - PN0) \times (1 - t)$$

$$C2 = PN1 + (PN2 - PN1) \times t$$

Since we want an equilateral triangle $\Delta C0C1C2$, we find $C0$ by rotating the point $C2$ 60 degrees about $C1$. Then, we draw a circle about $C0$ with radius $\overline{C0C1}$ (or $\overline{C0C2}$). The circular arc $\widetilde{C1C2}$ is the circle corner we wanted.

<u>Process #3:</u> Constant Speed

With the previous two processes, the camera motion contains changing orientation (which is similar to a person staying in one place but turning his or her head to look somewhere else), moving along a straight line, and moving along a circular arc at the corner. To achieve the goal of moving constantly, we must keep a constant speed while the camera is changing its orientation and moving along either a circular arc or a straight line.

Combining Process #1 and Process #2, take Figure 4-26 as an example, assuming the start is *PN0* and the target is *PN2*. In this figure, we separate the actions from the Process #1 and Process #2 into (a) and (b) in order to illustrate more clearly even though the actions are actually continuous. The camera movement from *PN0* to *PN2* can be broken down into the following actions:

(1) At *PN0*, change the camera orientation from *Pat0* to *Pat1* along $\widetilde{P}at0\widetilde{P}at1$ as shown in Figure 4-26 (a).

(2) After looking right at the target position, move the camera linearly along $\overline{PN0C1}$ as shown in Figure 4-26 (b).

(3) At *C1*, change the camera orientation from *Pat1'* to *Pat2* along $\widetilde{P}at1'\widetilde{P}at2$ as shown in Figure 4-26 (a).

(4) Then, Move the camera from *C1* to *C2* circularly along $\widetilde{C}1\widetilde{C}2$ as shown in Figure 4-26 (b).

(5) After reaching *C2*, move the camera from *C2* to *PN2* linearly along $\overline{C2PN2}$ as shown in Figure 4-26 (b).

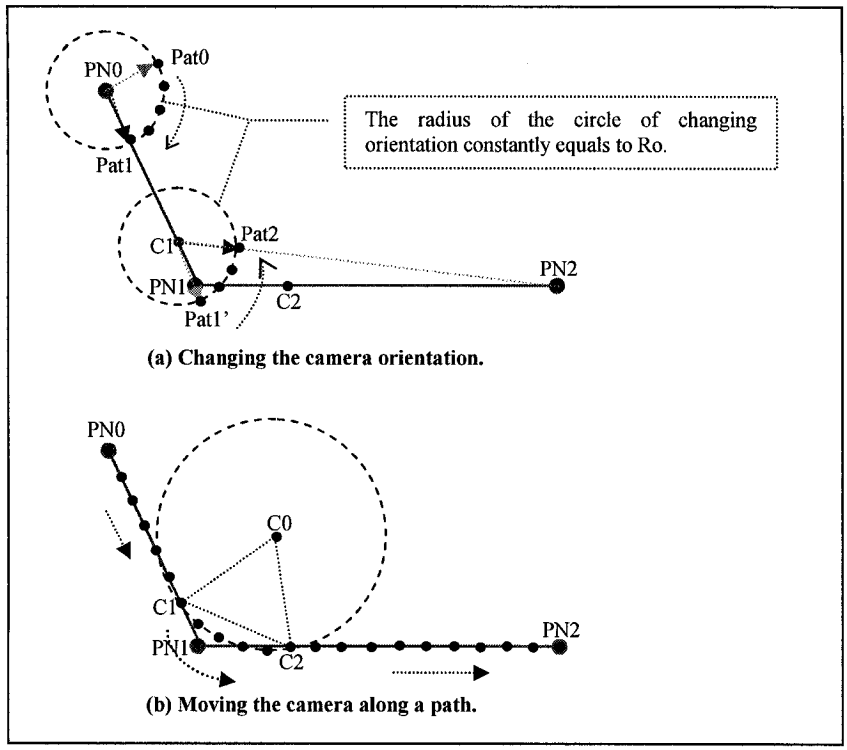

Figure 4-26: Keep moving in a constant speed.

In the Process #3, the program divides the path into equal steps (as shown in Figure 4-26) according to the length of the path so that the camera will move at constant speed. Assume that the user gives the program a constant speed named *cSpeed*. For the linear motion, the program simply moves the camera in distance *cSpeed* for each step:

$$distance_{\overline{PN0C1}} = distance_{\overline{C2PN2}} = cSpeed$$

For the circularly motion or turning, the program needs to computer how many degrees to move or turn on the circle path for each step.

$$2\pi Ro \times \frac{degree_{\tilde{P}at0\tilde{P}at1}}{360} = cSpeed \Rightarrow degree_{\tilde{P}at0\tilde{P}at1} = \frac{cSpeed \times 360}{2\pi Ro}$$

$$degree_{\tilde{P}at1'\tilde{P}at2} = \frac{cSpeed \times 360}{2\pi Ro}$$

$$degree_{\tilde{C}1\tilde{C}2} = \frac{cSpeed \times 360}{2\pi \overline{C0C1}}$$

After calculating the distance of each step for linear motion and the degree per step for circular motion, we can divide a path into steps and the camera is able to move with constant speed.

5 Result

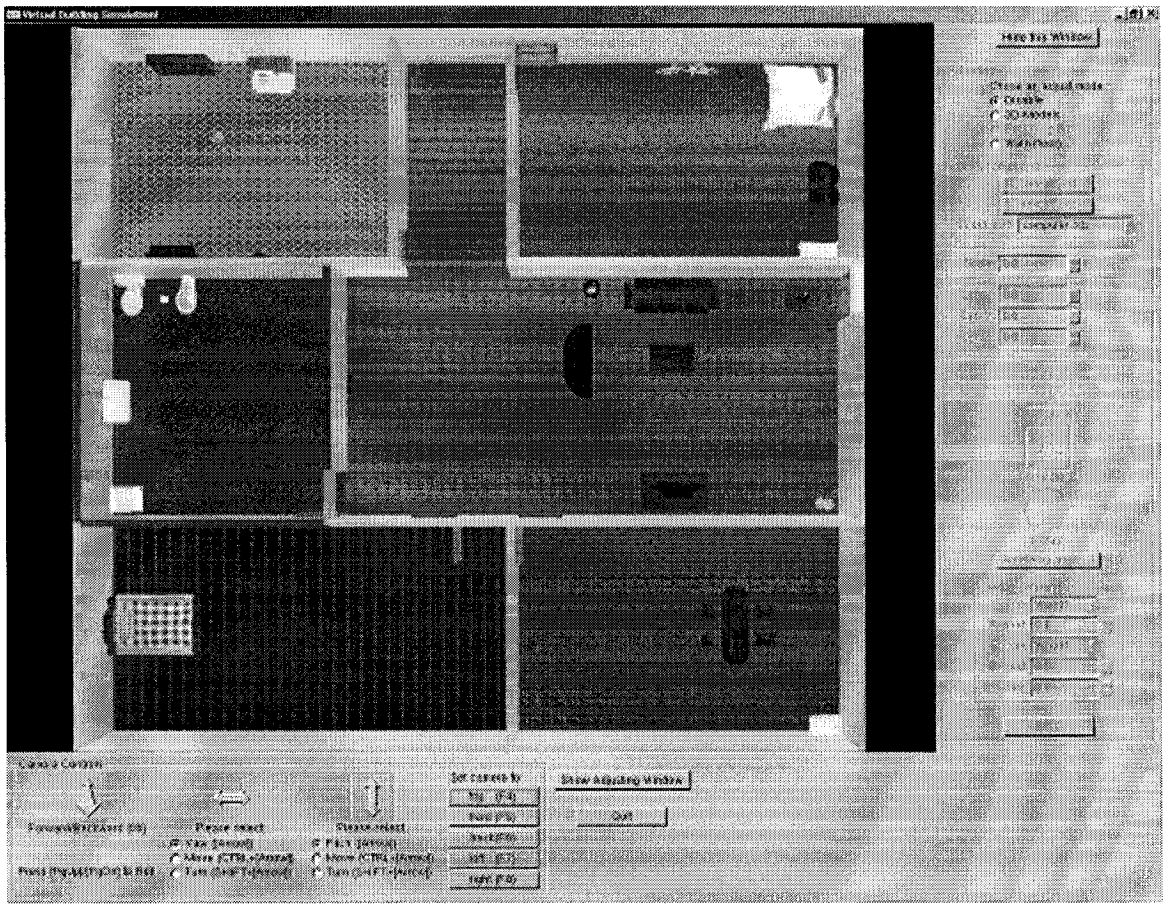

Figure 5-1 : *3D Virtual Environment Editor* User Interface

Our system provides such a friendly user interface (see Figure 5-1). The entire 3D virtual environment is displayed in the graphic main window. Figure 5-1 shows the top view, which allows users to adjust the 3D objects in place easily with the provided functions. Also, the view can be switched to the "human" view so that users can navigate through the 3D virtual environment. As an example, we are constructing a virtual house. While switching to "human" view, according to different position of the camera, the result is shown below from Figure 5-2 to Figure 5-8.

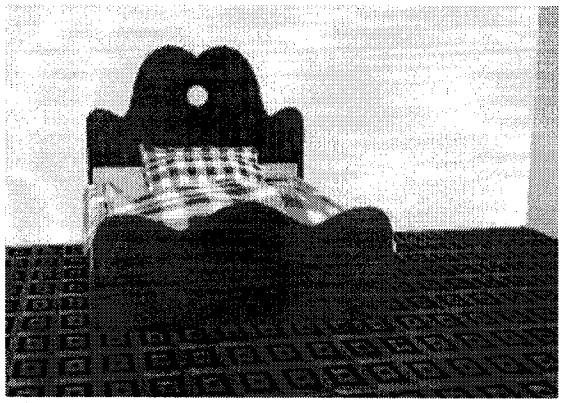

Figure 5-2 : Virtual house - Master bedroom

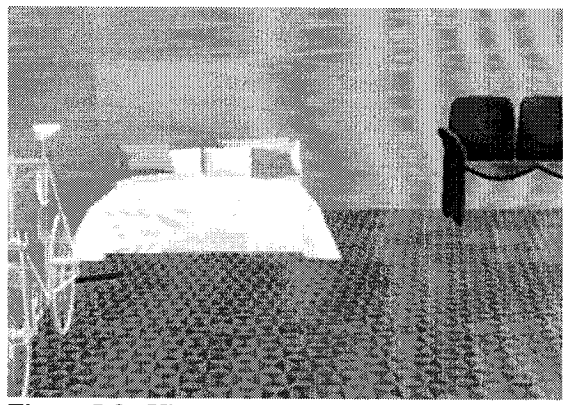

Figure 5-3 : Virtual house - Second bedroom

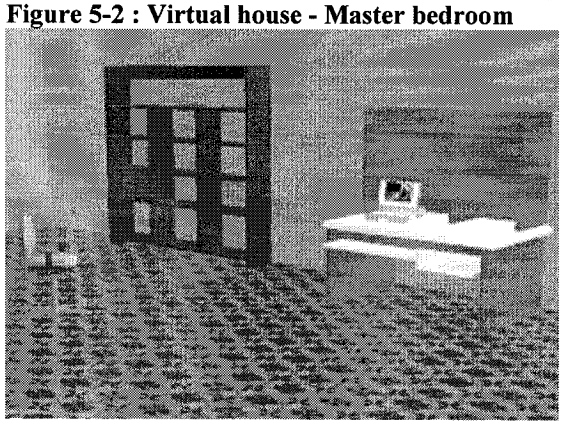

Figure 5-4 : Virtual house - Reading room

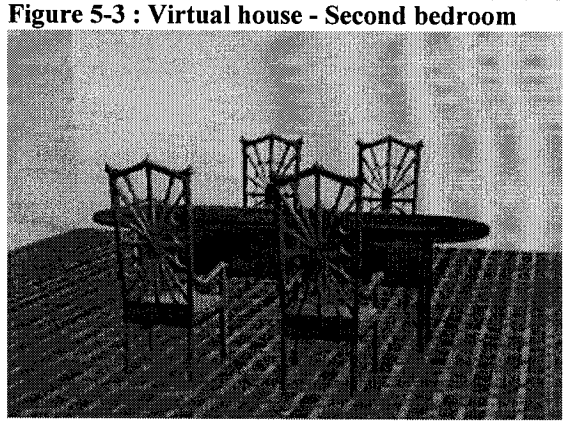

Figure 5-5 : Virtual house - Diner room

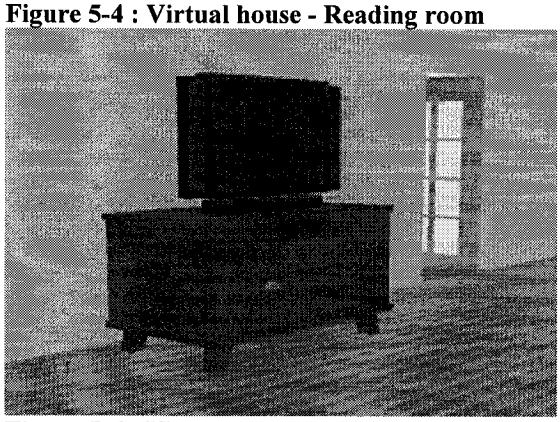

Figure 5-6 : Virtual house - Living room

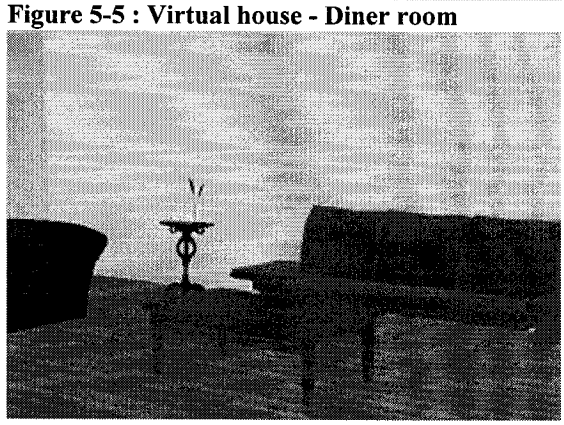

Figure 5-7 : Virtual house - Living room

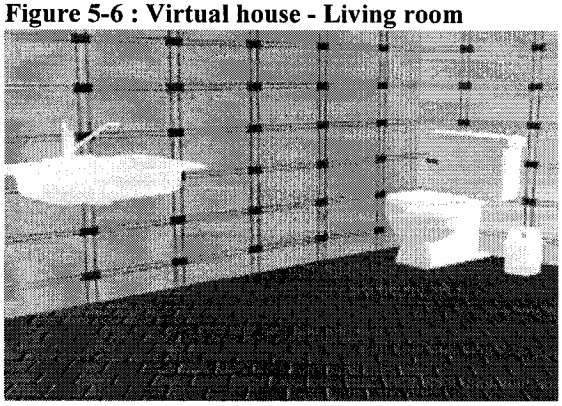

Figure 5-8 : Virtual house - Bathroom

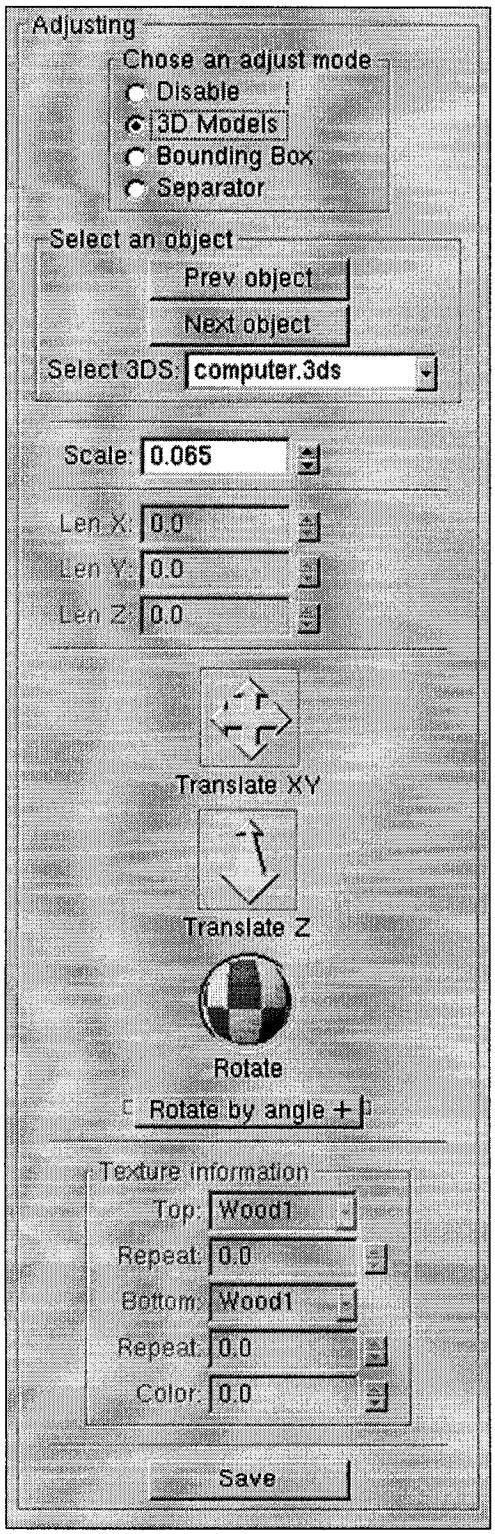

Figure 5-9 : *3D Virtual Environment Editor* User Interface – Left Command Window

Figure 5-9 shows the left command window of *3D Virtual Environment Editor* in Figure 5-1. The provided functions include modifying any object's scaling, translation,

and rotation in the 3D world coordinates, varying certain object's length, and altering certain object's texture or color etc.

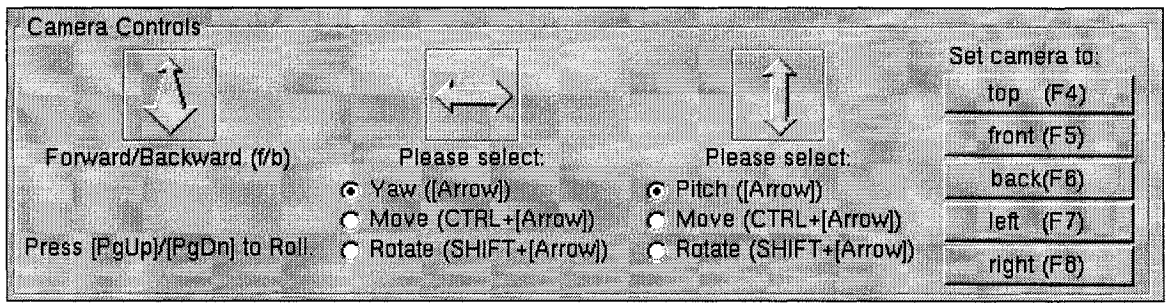

Figure 5-10 : *Walkthrough Controller* User Interface – Bottom Command Window

Figure 5-10 shows the bottom command window in Figure 5-1. It includes functions of moving forward/backward/left/right/up/down, pitching/yawing/rolling, and rotating left/right/up/down about where the camera currently looking at.

Figures 5-11 to 5-16 depict the floor plan of the virtual house. The floor plan indicates the position of the door of each room. In the screen view, doors are shown in red. For automatically walkthrough, the user simply mouse-clicks on a valid position inside the virtual house and the system will find a path automatically by applying the four path-finding rules respectively as we described previously. After a initial path is found, the system optimizes and smoothes the path so that the path is more reasonable and provides the user with a pleasant navigation. The figures below illustrate six situations in which the generated path describes camera movement from one room to another. Experiments show that our technique finds a valid path automatically in almost all cases inside a 3D environment.

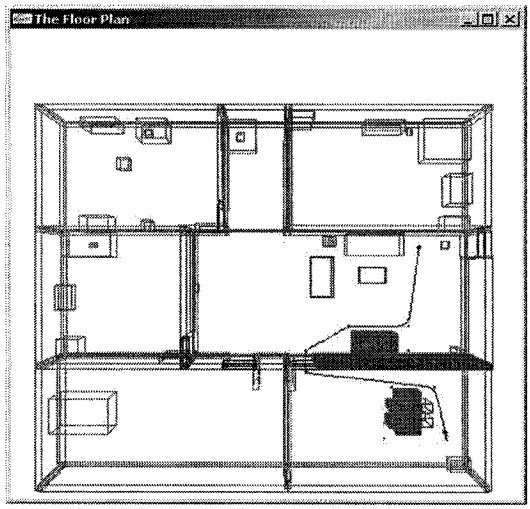

Figure 5-11 : Automatic walkthrough (1)

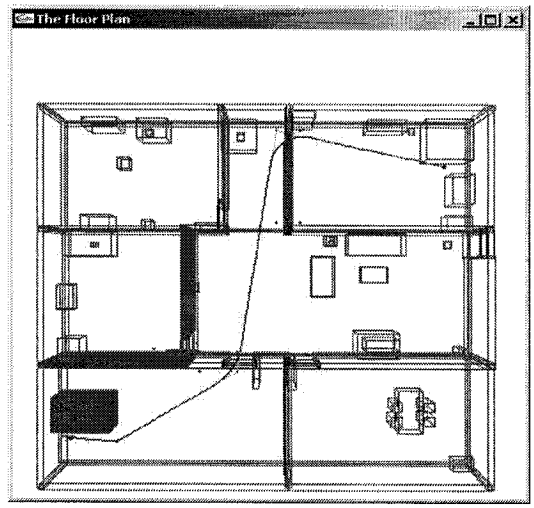

Figure 5-12 : Automatic walkthrough (2)

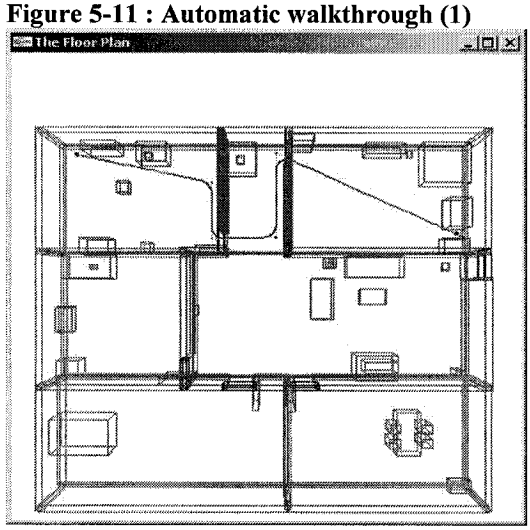

Figure 5-13 : Automatic walkthrough (3)

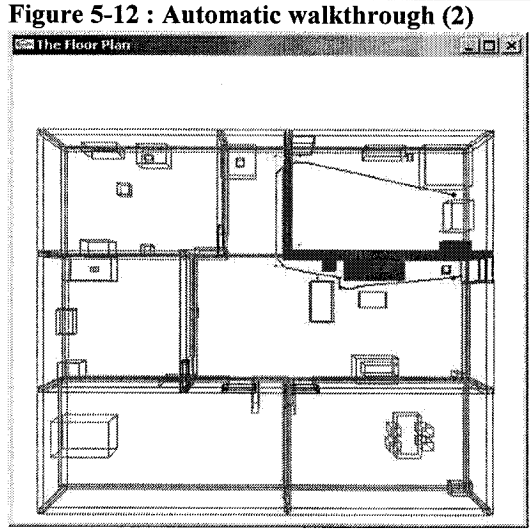

Figure 5-14 : Automatic walkthrough (4)

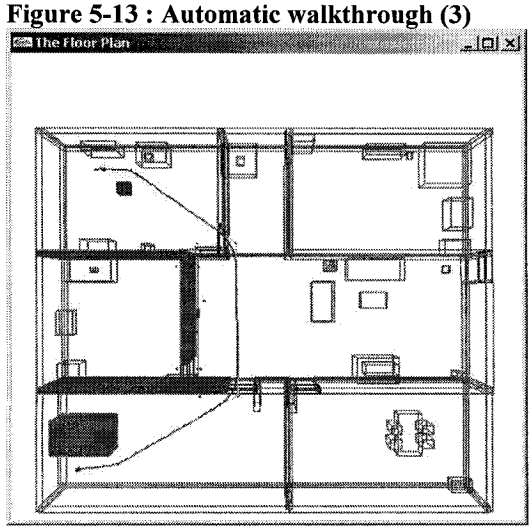

Figure 5-15 : Automatic walkthrough (5)

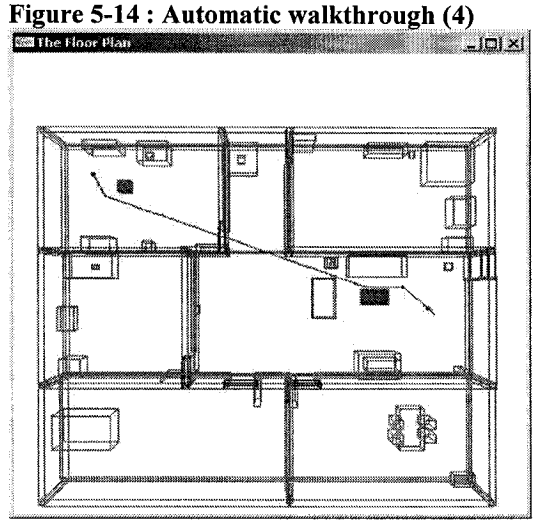

Figure 5-16 : Automatic walkthrough (6)

6 Conclusions and Future Work

In this thesis, we have described a new approach to automatically planning camera motions in a 3D virtual environment. The technique needs neither to pre-decompose the free space into any kind of uniform units nor to pre-calculate the path or road map. As a preliminary step, we developed a tool to dynamically modify the objects inside the 3D virtual environment. Because the space is not pre-divided into many uniform units, it is very easy for the system to handle the dynamic adjustments. There is only one thing the system needs to be pre-defined first, that is, a bounding box around each of the 3D objects. Cuboids have been chosen to represent the bounding boxes. Combining the approach of finding a valid path according to the four finding-path rules and the techniques of optimizing and smoothing the initial path, we obtained a system that can help with the exploration of virtual environments. Experiments with 3D virtual environment walkthroughs verified the quality of the resulting motions.

The system is quite reliable, yet there are few things can be done to make it better in the future:

- Improve the *3D Virtual Environment Editor* by adding functions such as read some other 3D objects beside 3DS files and import new 3D objects.
- Build a more complex 3D virtual environment to experiment and help to improve the finding path technique. A more complex 3D virtual environment implies to a building that has a few levels so that the camera motion will involve climbing up stairs and ladders.

- This thesis puts more emphasis on finding the path than on optimizing and smoothing it. Improving the techniques of optimizing and smoothing is left as future work.

References

[1] D. Ben. (2001). 3DS Loader Tutorial. Retrieved September, 2004 from http://www.gametutorials.com/gtstore/c-1-test-cat.aspx

[2] P. Grogono (2003). Concordia University Graphics Library. Retrieved October, 2004 from http://www.cs.concordia.ca/~grogono/CUGL/

[3] Alan W., F. Policarpo (date). The anatomy of an advanced games system II. In A. Editor (Eds.), *3D Games Animation and Advanced Real-time Rendering.* pp. 83-129, Addision-wesley.

[4] C.A. Mandachescu (2003). *Path Finding in 2D Games*. Dept. Computer Science, Concordia University, Montreal, Quebec, Canada, April 2003.

[5] O. Goemans, M. Overmans (2004). *Automatic Generation of Camera Motion to Track a Moving Guide*. Institute of Information and Computer Sciences, University of Utrecht, the Netherlands, July 2004.

[6] S. Beckhaus, F. Ritter, T. Strothotte (2000). Cubicalpath – Dynamic Potential Fields for Guided Exploration in Virtual Environments. Proceedings Pacific Graphics 2000 (Hong Kong, China, October 2000), pp. 387-395, Los Alamitos, 2000. IEEE Computer Society.

[7] B. Stout (1999). *Intelligent path-finding.* http://www.gamasutra.com/features/19990212/sm_01.htm, 1999.

[8] D. Nieuwenhuisen, M.H. Overmars (2003). *Motion Planning for Camera Movments in Virtual Environments*. Institute of Information and Computing Sciences, Utrecht University, the Netherlands, January 2003.

[9] A.J. Hanson, E.A. Wernert (1997). *Constrained 3D Navigation with 2D Controllers.* Computer Science Department, Indiana Unviersity, 1997.

[10] J.D. Mackinlay, S.K. Card, G.G. Robertson (1990). *Rapid Contolled Movement Through a Virtual 3D Workspace.* Computer Graphic (SIGGRAPH '90 Proceedings) 1990, pp. 171-176.

[11] W.H. Bares, S. Thainimit, S. McDermott (2000). *A Model for Constraint-based Camera Planning, Smart Graphics.* 2000 AAAI Sprint Symposium, AAAI Press, 2000, pp. 759-774.

[12] S.M. Drucker, D. ZSeltzer (1995). *CamDroid: A System for Implementing Intelligent Camera Control.* P. Hanrahan and J. Winget (Eds), SIGGRAPH Symposium on Interactive 3D Graphics, 1995, pp. 139-144.

[13] N. Halper, P. Olivier (2000). CAMPLAN: A Camera planning Agent, Smart Graphics. 2000 AAAI Spring Symposium, AAAI Press, 2000, pp. 92-100.

[14] H.H. Gonzalez-Banos, C.Y. Lee, J.C. Latombe (2002). *Real-Time Combinatorial Tracking of a Target Moving Unpredictably Among Obstacles.* Proc. IEEE Int. Conf. On Robotics and Automation, 1683-1690, Washington D.C., May 2002.

[15] N. Halper, R. Helbing, T. Strothotte (2001). *Computer Games: A Camera Engine for Computer Games.* Computer Graphics Forum Volume 20, Issue 3.

[16] J.C. Latombe (1991). *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[17] S. Bandi, D. Thalmann (1998). *Space Discretization for Efficient Human Navigation.* Proc. Of Euro graphics, volume 17, pp. 195-206, March 1998.