Abstract

An important aspect of Virtual Environments is navigation. Most of the time, the user controls the camera directly, allowing full access to the environment. But in some situations, it might be more desirable to have the software automatically generate locations for the camera and the paths needed to travel between those locations. This should enable the user to concentrate on other, more important tasks.

The goal of this thesis is to determine if it is possible and feasible to provide users with the option to select an object[1], and then have the camera automatically travel to a destination from which the object can be viewed in its entirety, with as little obstruction in front of the camera as possible. In order to achieve this, various methods have been researched in previous literature for the phases needed to create this mode of navigation.

First, the various phases of path planning are discussed. Static and dynamic environments are discussed separately for these phases, since they require different approaches.

The pre-processing phase involves creating a roadmap, which can be done either manually or automatically. Since manual roadmap generation is not an option, we focus on the methods that enable software to automatically generate good roadmaps. Most of those methods are variations of the Probabilistic Roadmap Method (PRM). One of these variations randomly generates sample points, of which some can be used as either guard nodes or connectors between guard nodes [2]. Another variation that uses Voronoi diagrams to generate nodes seems to be a better fit for our goal [3]. It also uses circle-smoothing to create a C^1-continuous path, which is a requirement for natural camera navigation. It is also possible to use deterministic Halton points [18] instead of random points to obtain a natural result with uniform coverage of the environment and consistent running times. A tricky part of roadmap generation is to make sure that narrow passages have decent node coverage. This can be done by using the bridge test [17] in addition to a more uniform sampling method.

The second phase is the actual path planning algorithm. A few variations on the Dijkstra and A* search [24] algorithms are discussed. A* search is a well known and easily implementable search algorithm, and we will use it for both types of environment. However, dynamic environments require an altered version of the A* search algorithm that supports collision detection. One such algorithm is the Lazy PRM method [20].

Then, there is the post-processing phase of path planning, where camera orientation and speed during the course of the path will be calculated. It is essential that the viewer gets cues about where he is going, that the path is smooth and C^1 continuous, and that the speed of the camera depends on the curvature of the path, to prevent motion sickness [6]. Also, it is recommended that a speed diagram is used, which gradually accelerates and decelerates speed when needed instead of abruptly changing it. Using a speed diagram makes the path C^2 continuous and thus even more smooth.

Finally, literature on camera positioning is briefly discussed, but we discover that brute force algorithms are still the driving force behind most of the methods. We will have to find some optimizations for brute force camera positioning that have not been mentioned in these papers.

[1] However, the method for selecting the object is a choice that should be left to the application designer and it will not be discussed in this thesis.

After the extensive literature section, a few possible optimizations to speed up camera positioning are suggested. Most of these are based on bounding volumes and reducing the amount of collision detection required.

The next chapter describes our implementation, both the result and the difficulties and other interesting things we encountered during development. A manual of the application is provided in an appendix. Only the path planning methods for static environments have been implemented.

Finally, we reach a conclusion and decide if selection-based navigation is possible with the techniques described in this thesis.

Contents

1. Introduction

1.1 Background and main goal

There are many different modes of navigation that can be used within virtual environments. The mode that most people are familiar with is direct navigation, in which you control the camera either directly, or by controlling another object which in turn controls the camera (like a character or vehicle in a computer game).

Sometimes, however, this is not the most useful mode of navigation. Imagine a user who is exploring a virtual city and wants to visit the landmarks. It would be possible to have the user read a map to get to his goals, but it would be easier if the user could just select the landmark of choice and have the software do the rest of the navigation.

In order to achieve such an effect, a navigational mode could be created which is able to find paths through the virtual city. In addition to this, the user will appreciate a clear view of the selected landmark, with no trees or other buildings obstructing his vision. This means that the goal position should have as little occlusion of the selected landmark as possible.

Also, when object manipulation is the main goal of a program (e.g. a 3D modelling and animation suite), a user can get confused by directly controlling his viewports. This is mainly because most of the time, similar controls are used for object manipulation and navigation. If the user is in the wrong operating mode, his actions might have an undesirable effect. Sometimes it is not even noticeable at first, because rotating a viewport and rotating an object might have the same apparent end result in certain applications.

It could be possible to give the user an option to select the object he wishes to work on (from a list, diagram or other viewport), and have the camera position change to a position that is optimal for editing the selected object. Teleporting directly to such a position could leave the user confused about his position and orientation, which is why this situation also benefits from the camera following a smooth path.

1.2 Goal

In this thesis, some of the known methods for accomplishing these goals will be discussed. Additionally, a sample application has been made, which implements some of the principles mentioned here. The process of creating this application will be described, as well as the problems that have been or could have been encountered. The written part of this thesis will deal with two situations: static and fully dynamic environments. An example of a static environment is a virtual city (assuming traffic is not accounted for in the simulation), while a good example of a dynamic environment is a 3D animation program. The separate treatment of these two types of environments has a reason, which will also be discussed.

There has been a lot of previous research on path finding. It actually consists of two parts, one of which is the actual path finding algorithm. The path finding algorithm requires a graph to search through. Such a graph is called a roadmap. A roadmap can be specified either manually or dynamically. For this thesis, road mapping should be a dynamic process for both static and dynamic environments. Path planning is used in a lot of fields, of which computer games and robotics offer the most relevant solutions to this problem.

The difference between dynamic and static environments is that dynamic environments do not have a fixed roadmap. Most of the roadmap calculation algorithms are preferably executed as a pre-processing step, to make sure the actual path finding can be done in real time. However, this approach will not work in a very dynamic environment because it does not have a fixed roadmap. In turn, a road mapping/path finding approach that works very well in dynamic environments will also work for static environments, but will be less efficient than a separate optimized approach for static environments. Most of the previous research deals with generally static environments, sometimes combined with low-level collision detection to prevent hitting certain dynamic entities, or algorithms to deal with known movement patterns.

We will have to find a good approach for static environments and another approach for dynamic environments.

Determining a desirable target position for the camera is something that has not been previously researched as extensively as path planning. Very few specific algorithms could be found, but there were some papers with background information that could prove useful. Most of the information found on this subject came from papers on (interactive) cinematography, which is not very surprising, considering the importance of camera positions in that field.

Again, there is a difference between static and dynamic environments here. In dynamic environments, it will not be possible to store a calculated camera position, because the object of focus might be moved, or another object might have been moved into a position where it obstructs the calculated view. Neither will it be possible to have a pre-processing algorithm for calculating target positions.

The movement of the camera should be natural. Guidelines and algorithms to generate natural camera movement have been well documented. Again though, for dynamic environments, we cannot rely on any sort of pre-processing algorithms.

By researching and combining existing algorithms, creating new ones and comparing some of them with the help of a sample application, a general navigation mode could be created that lets the user navigate by simply selecting a target object. The main goal of this thesis is to create a basic version of such a navigation mode and to show that it works.

2. Literature

This chapter contains the results of a literature study consisting of path finding (including road mapping as a separate subject), camera movement and camera placement. There will be two sections for each subject: one for static environments and one for dynamic environments. Also, unless stated otherwise, environments will be assumed to be in 3D space.

While the target camera position should be determined before searching for a path (because you need a goal to find a path to), path finding will be discussed first. Path finding has been researched more extensively than camera placement, and contains principles that are needed for the other subjects. Camera movement is closely related to path planning and thus will be discussed immediately afterwards. Finally, we will discuss the camera placement algorithms that could be found in existing literature.

Another matter to consider is that path finding basically consists of three phases, one of which is the actual path planning. From here on, we will refer only to the separate phases, and the term “path planning” or “path finding” will refer to just the actual path planning algorithm, without the pre- and post-processing phases. The following are the three phases that will be discussed:
- Pre-processing (chapter 2.1 and 2.2)
- Path finding (chapter 2.3 and 2.4)
- Post-processing (chapter 2.5)

These phases will each be discussed separately for static and dynamic environments. Because they are all dependent on the output of the previous phase, we will make a final decision on the best techniques to use for each phase in a certain type of environment before continuing to the next one.

There are several types of dynamic environments, and some benefit more from a static environment approach.

A strategy game, for instance, benefits more from a static approach than from a dynamic one. The environment itself rarely changes, since the only dynamic obstructions are units. They can be easily avoided with collision response algorithms that eliminate the need for calculating a new roadmap. In some cases, unit collisions are even completely ignored in games.

Then, there are environments with semi-dynamic objects. We will use the term semi-dynamic object for an object that follows a known and fixed path or other movement pattern. This type of object moves, but the movement can be precisely anticipated, which makes it possible to deal with this kind of movement with an adaptation of a static approach. An example of this type of environment would be a puzzle game, like a maze with moving obstructions.

Finally, there are the truly dynamic environments, in which obstructions are controlled by the user or an unpredictable (complex) algorithm. This is the type of dynamic environment we wish to support in our navigational mode. Good examples are 3D modelling and animation software packages.

2.1. Pre-processing for static environments

The pre-processing step for static environments mainly consists of creating a roadmap. Like stated before, a roadmap is a graph. More specifically a graph that encompasses a virtual environment and only has edges where the existence of one would not cause an intersection with an object in the environment. Also note that most roadmaps are undirected graphs, unless explicitly stated otherwise.

Not all path finding implementations require a pre-generated roadmap though. In some fields where agents are used (like robotics) in an environment unknown to the agent, a roadmap can be constructed on the fly [1]. However, since the static environment in our case is fully known by the software, we will use pre-generated roadmaps.

So, the first task at hand is generating a roadmap from the geometrical data we have. There are several ways of generating roadmaps, and we will discuss several of them.

Another important feature that can sometimes be calculated in the pre-processing step is path smoothing [3, 6]. The most natural looking movements follow a path that is C^1-continuous. This means that the derivative of the path should be (C^0-) continuous, ensuring a path without sudden quirks that might distract the user. Some of the road mapping algorithms presented here will be able to generate a pre-smoothed roadmap, while others will need to have the path smoothed once it is found. An algorithm for smoothing the calculated paths in roadmaps that are not pre-smoothed will be discussed in the post-processing section later on.

Manual roadmap generation

Manual roadmap generation would be the easiest to implement. Basically, this is where a designer or programmer creates a suitable roadmap, so the software does not have to generate one. Instead of having to process geometry, the application is already provided with a set of nodes and edges, which are created by the designer or programmer.

There are some interesting advantages to this method. Firstly it speeds up the programming process because no road mapping algorithm implementation is required. It also speeds up the pre-processing phase because generating a roadmap is not necessary anymore. Finally, a good designer will be able to create a very efficient and optimal roadmap. Efficient, because he will be able to see all the spots that need to be reached and is able to create nodes and edges to achieve this for every scenario, and optimal because he will be able to use nodes only where necessary. A computer-generated roadmap will not necessarily be able to reach every reachable part, and, without optimizations, it will contain redundant nodes and thus be more complex than needed.

The disadvantage is that a designer has to create roadmaps manually for each environment. Depending on the number of environments needed, a computer-generated road mapping solution might be cheaper for a company. In certain types of computer games, manual roadmap generation would be the way to go, but there is another drawback to using manually generated roadmaps. If the environment needs to be changed, the designed roadmap will quickly become useless. A designer cannot redesign a roadmap every time a client redesigns his version of the environment, so a different method should be used in those cases. There are exceptions to this though. For instance, some first-person shooter games use a manually generated roadmap for the AI (Artificial Intelligence) controlling certain entities in the game. A normal user does not have to be able to change his environments, but a computer savvy user might want to create his own levels for the game. In that case, it is possible to have the advanced user also create a roadmap to match his creation.

Advantages:
- Nice roadmaps.
- Less programming needed.
- No pre-processing needed.

Disadvantages:
- More designer work needed.
- New environments take more time to create, because a roadmap has to be designed as well.

Generating roadmaps using synthetic vision

Another way of generating roadmaps is by using synthetic vision [1]. This paper describes a method for path planning by using autonomous agents without access to the internal representation of the world, with a goal of developing a synthetic vision system that resembles real-world vision and can still operate in real-time.

The method is based on the agent rotating and taking snapshots with depth information. The gathered information is then used on multiple levels; on a low level to avoid collision, and on a higher level to plan intermediate destinations. The highest level is creating a map of the environment and an accessibility graph, that can be used for path finding.
Combining all gathered depth information to correctly form the vision buffer requires certain corrections and transformations, which are explained in the paper. Also described are two different levels of collision avoidance methods (short range and medium range) and map construction. Since this method is based on an agent without access to the internal representation of a map, and the camera in this thesis does have access to that information, the method used to construct a map has some differences from the other discussed methods.
While this method results in natural looking roadmaps which are constructed in a natural way (the same way a human would do it mentally), it is restricted to 2-dimensional movement, which makes it less usable. However, with some adaptations, it is probably possible to have this method work with stairs and slopes as well, and it might be very suitable for architectural walkthroughs or AI paths in computer games.

Advantages:
- Most ‘natural’ method, because it is based on how humans make a mental roadmap.
- Very useful for environments with AI entities.

Disadvantages:
- Restricted to 2D movement.
- More difficult to implement than most other methods.

Using pre-fabricated roadmaps

It is also possible to have a computer-generated roadmap, and store it for later use with the same environment, just like what happens with manually generated roadmaps. However, the only advantage to this method is that it saves calculation time. It does not share most of the advantages of manually generated roadmaps, and it does share some of its disadvantages.

Though if the software is able to automatically recalculate a roadmap whenever it detects that the environment has been altered, this can be a useful optimization to a good road mapping algorithm. Keep in mind that this is only useful for static environments that need to be changed infrequently. Recalculating a roadmap whenever a change is detected will still not suffice for dynamic environments, because calculating a new roadmap takes a lot of time.

Advantages:
- No pre-processing needed in familiar environments.
- Good for optimizing another, fully functional road mapping method.

Disadvantages:
- Only useful for static environments that are often used without any changes.
- It is not a method in itself, and needs a fully functional road mapping method to work.

Grid-based roadmap generation

With grid based road mapping, the environment is divided into a grid of evenly spaced cells. When a cell is completely empty, it means there are no obstacles in that part of the grid. Such a cell will have a node in the center of it. However, when there is an object in a cell, the entire cell is considered to be an obstacle and no node will be present. After the nodes are generated, every node can be connected to each of its neighbouring nodes (which results in a maximum of 8 edges per node in 2D or 26 in 3D) without requiring any additional collision detection. This is an important advantage of grid-based road mapping.

In this case, we use an evenly spaced 3D grid as a roadmap. This type of grid-based roadmap generation would be relatively easy to implement, but the results will not always be useful. If the grid is too sparse, minor passages might be overlooked. On the other hand, a grid that is too dense will result in a very complex roadmap with a lot of redundancy, which in turn increases the time needed by the path finding algorithm.

This does not mean grid-based road mapping cannot be a good solution. There are ways to optimize this method. By using an octree [14, 15] instead of a grid, a more economic result can be obtained because of the adaptive spacing. An octree is a recursively defined data structure. Each level of refinement contains either 0 children or is subdivided into 8 children (one for each corner of a cube). By using an octree, the resulting grid will only be sparse in the areas where it is really needed and thus be more optimal.

Octrees can be generated by using the following pseudo-code algorithm:

```
refine (cell)
        compute input body (the parts of the environment that are clipped by the cell)
        if an input body is there
                        and the curvature is above some threshold
                        and level < maximum level of refinement
                allocate space for eight children for this cell
                refine (each child)
        else
                cell is a leaf, no further refinement needed
        end if
```

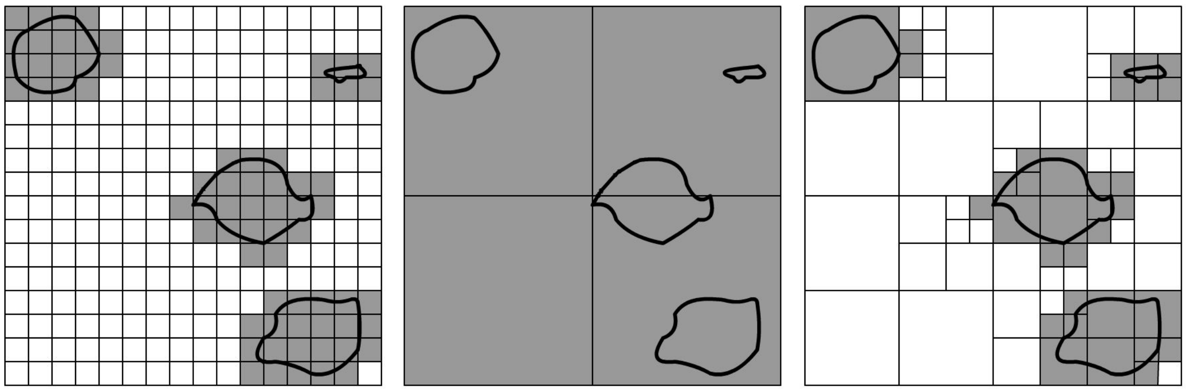


Fig. 2.1.1: Examples of possible grid-based roadmaps, in 2D for visualisation purposes. Shaded cells are not traversable.
a. A dense grid, with a lot of redundant nodes and edges.
b. A sparse grid, where some of the openings between objects cannot be traversed. In fact, this grid is so sparse that a miniscule part of one object makes a huge cell untraversable.
c. An octree-based grid (actually shown is a quadtree, the 2D version of an octree). As you can see, this results in the most efficient grid-based roadmap.

The result may look promising, but as we will see, there are more flexible algorithms to obtain a similar roadmap.
Another advantage to a purely grid-based roadmap is that all angles are exactly 45 degrees or multitudes of 45 degrees. This can speed up certain other calculations, like path smoothing, because every fixed value saves some calculation time.

Advantages:
- Easy to implement, octree-based is slightly harder but still relatively easy.
- No collision detection needed for edges.
- Cubic design enables heavily optimized post-processing algorithms.
- Deterministic nature results in the same roadmap every time, which is less confusing to the user.

Disadvantages:
- Grid can easily get too dense or too sparse.
- A lot of redundancy when no optimizations are added.
- Hard to tell exactly how dense the grid should initially be to obtain maximal coverage.

Probabilistic Road mapping Method (PRM)

The Probabilistic Road mapping Method, or PRM [2, 3, 4], is a method that uses randomly or semi-randomly selected nodes. There are many variations on this algorithm. It seems to be able to generate the best roadmaps (when optimized), and it has been widely used in previous research. It shares some of the disadvantages of grid-based road mapping, because with randomly positioned nodes, it is even less certain if you are able to cover a certain area.
A few variations on PRM and some possible extensions of them will be discussed here.

Advantages:
- In general, PRM approaches will result in more natural looking roadmaps.

Disadvantages:
- A different roadmap is generated every time, which might confuse a frequent user.
- It can be hard to tell when to stop sampling.

PRM using a radius for the camera

Nieuwenhuisen and Overmars [6] describe a PRM variation specifically aimed at camera movement. In fact, the goal of their entire method is exactly the same as the main goal of this thesis, minus determining goal camera position and orientation for a selected object. Their method has been successfully implemented in a system for architectural walkthroughs and urban planning, which makes this a very practical method.

The road mapping part of their algorithm is based on the fact that a camera needs a certain amount of clearance to prevent the user from thinking a collision occurs when an object is really close (but not hitting) the camera. This means that, unlike most other PRM variations, collision detection will not be point and line-based, but sphere and cylinder-based. First, a random set of nodes is generated. A node may only appear in the roadmap if it does not intersect with an object within a certain distance of the node (basically each node has a bounding sphere that may not collide with another object in the scene). Edges are not represented by straight lines, but by cylinders with the same radius as the node's bounding sphere. For each node, edges to all nearby nodes will be created, and only added to the roadmap if there are no object intersections with the edge's bounding cylinder.

Sometimes it is necessary to limit the roadmap in a certain dimension (e.g. only have the camera be able to move at a certain height). This can easily be accomplished by using a fixed value for that dimension and only generating random positions in the other two dimensions.

The set of neighbouring nodes must be chosen wisely. If a large set is chosen, the algorithm will take a longer time to process because of all the collision detection that is needed, but the roadmap will become dense, which leads to good motions. A small set will be faster to calculate, but it might result in longer camera paths. The authors opted for a fixed number of neighbouring nodes for their application.

A very important part of any PRM algorithm is knowing how many nodes need to be generated for a good result without too much redundancy. In this PRM method, the authors check whether the number of connected components changes or not. If it does not change anymore when nodes are added, it can be assumed there are enough nodes present and the roadmap is ready for use.

Advantages:
- Good method supplied for detecting when to stop adding sample points.
- Camera keeps a certain distance away from objects.
- Method aimed at camera motion, which is the goal of this thesis.

Disadvantage:
- Seems to be a little processor-intensive because of the collision detection.

Constrained PRM using local planning

The method proposed by Salomon, Garber, Lin and Manocha [2] again uses random sample points as nodes, with the addition of the configuration space formulation. In this formulation, the avatar is represented by a point in a higher dimensional space C, so any planning task can be interpreted in terms of a point robot. This is done so the constraints on the avatar's movement can be separated from the actual algorithm for finding a path that satisfies those constraints. A point can either be in C_{free} or in $C_{blocked}$. C_{free} (also called free space or workspace) is the set of all nodes for which the avatar does not collide with an object, while $C_{blocked}$ is the set of all nodes where the avatar collides with at least one object. Also, they specify a set of goals for their sampling strategy. The samples should be distributed uniformly in the environment, and there should be an upper bound to the distance between two nodes in a worst-case scenario. So far, this is nothing really different from most PRM variations in principle.

However, they have a special approach for connecting the nodes. Instead of performing a simple collision test for a straight line between two nodes, they use a local planner to simulate an avatar walking that path. This allows for more constraints to be used and enforced, which can be useful in certain situations. Also it takes into account that being confined to walkable surfaces means edges will not be straight lines when you have a stairway or a hill in between two nodes.

The nodes in C_{free} and the edges connecting them form the roadmap. Next, the nodes will be divided into two different types of nodes. Some of the sampled points will be rejected, because they can easily be reached from other sampled points. They offer no extra connectivity, so they are redundant. If the reachable area for a node does not contain other nodes, that node is considered a guard node. However, if the node turns out to be connected to at least two other guard nodes when a radius higher than the one of the reachable area is used, it is considered a connector. Connector pruning can be used after the sampling stage to eliminate redundant connectors before they enter the roadmap. If pruning is used, a connector will not be added to the roadmap when it connects the same two guard nodes as one of the existing connectors.

An advantage of using guard nodes is that the reachable areas of the guard nodes can later be used to estimate the coverage of the environment, which is useful for determining when to stop adding sample points.

This method might not be entirely suited for the form of camera navigation we desire. As Nieuwenhuisen and Overmars [3, 6] suggest, the camera's path should be C^1 continuous. For this method, this is not the case. Interpolation could be added in real-time, after a path has been found, but it might cause problems. Because when interpolation is used, new collisions might occur in the path, which should be accounted for. Also, the constrained nature of this method might result in good roadmaps for avatars, but not for a free-flying camera. A camera that is walking stairs or climbing hills on the way to its goal, although realistic, is not a desirable effect for us.

Advantages:
- Flexible, can be used with many constraints because of the local planner.
- Contains a good pruning algorithm and a good method to estimate coverage.

Disadvantage:
- Most constraints are not needed, so this method has some unnecessary complex parts.

PRM using Voronoi diagrams

Nieuwenhuisen, Overmars, Kamphuis and Mooijekind [3] propose a road mapping method using Voronoi diagrams as a guide. In the generated roadmap, there is always a certain amount of clearance from objects in the environment. Also, their method results in C^1-continuous roadmaps, so less post-processing is needed for smoothing a calculated path.

First, we will explain what a Voronoi diagram is. A Voronoi diagram [7] consists of Voronoi Regions (or Voronoi Cells), separated by Voronoi Lines which are in turn connecting Voronoi Points. The main property of a Voronoi Point is that its position is equidistant to three of the points (or objects, in the case of road mapping) that the Voronoi diagram was built around. This also means that it is possible to create a circle with its center on the Voronoi Point, which intersects with all three of the original points. Each Voronoi Point is connected to three other Voronoi Points by Voronoi Edges. The original points in the scene will now be encased in Voronoi Cells, as shown below.

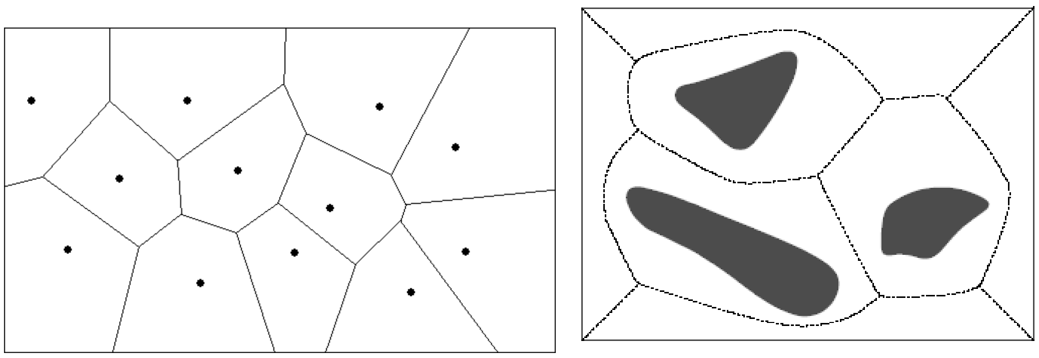
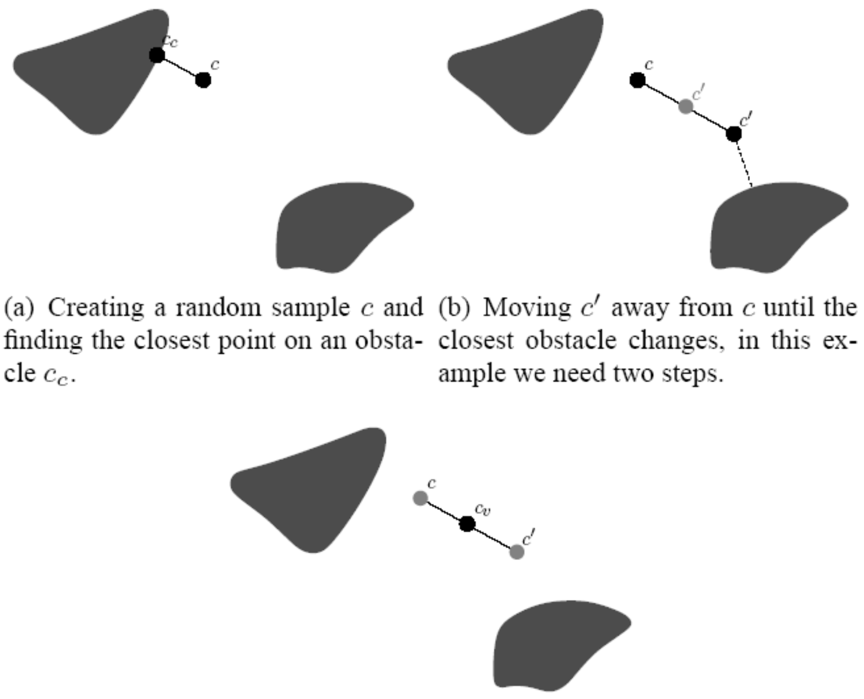


Fig 2.1.2: Example of a regular Voronoi diagram [8] and a Voronoi diagram based on objects instead of points [3]

This translates into a road mapping algorithm as follows. When the location of a point is sampled and determined to be collision free within a certain radius, it is retracted to the Voronoi diagram. This means that it is not necessary to actually calculate a complete Voronoi diagram, because that could not be called a Probabilistic Road mapping Method since is it based on pure calculation. The samples are still randomly chosen here, but unlike other algorithms, this one relocates the sampled points. First, a point of one of the nearby obstacles is used that is closest to the sampled point c, and that point will be called c_c. Next, a new point will be generated called c', which is positioned at a distance equal to the distance between c and c_c, but in the opposite direction from c, so that c is exactly in the middle of c_c and c'. We keep moving c' away from c until a point on an object is found that is closer to c' than the distance to the closest object was in the previous step. Then, a point c_v in between c and c' should be found that is equidistant to the two different objects. This can be done by using a binary search with precision ε. When c_v is found, it will be at most at a distance of ε from the actual Voronoi diagram. c_v can now be added to the roadmap.



(a) Creating a random sample c and finding the closest point on an obstacle c_c.

(b) Moving c' away from c until the closest obstacle changes, in this example we need two steps.

(c) Finding the sample c_v that is on the Voronoi diagram using binary interpolation.

Fig 2.1.3: Converting a sampled point to a point on the Voronoi diagram of the environment [3]

Next, the edges must be retracted to the Voronoi diagram as well. In this case, retraction is the process of altering edges to approximate a certain path. We use retraction because straight edges between the nodes might get very close to objects and not keep enough clearance from them. This can be solved by retracting edges to the Voronoi diagram until every part of the edge is at least some pre-specified distance away from the object. A way to obtain this result is as follows: if (part of) an edge is to close to an object, the edge is split in two parts of equal length, and the middle point is retracted to the Voronoi diagram as described earlier. This procedure is recursively applied to the two new edges, until the required result is obtained. There might be a problem where the recursive splits will continue infinitely for edges leading through narrow passages. This can be stopped by using a minimum edge length as a precondition for splitting. At a certain length, splitting again will not help anymore, so the algorithm should stop there.

Another problem which might occur is that edges could overlap each other. Detecting such an overlap can be done by checking, for each pair of edges e_i and e_j, how far their endpoints are away from the other edge. If this distance is smaller than a certain predefined distance, we try to project the points of e_i on e_j and vice versa. If at least one of these projections is successful, we call the two edges overlapping and can proceed to join them. There are four different types of overlap, which are all shown in fig. 2.1.4.

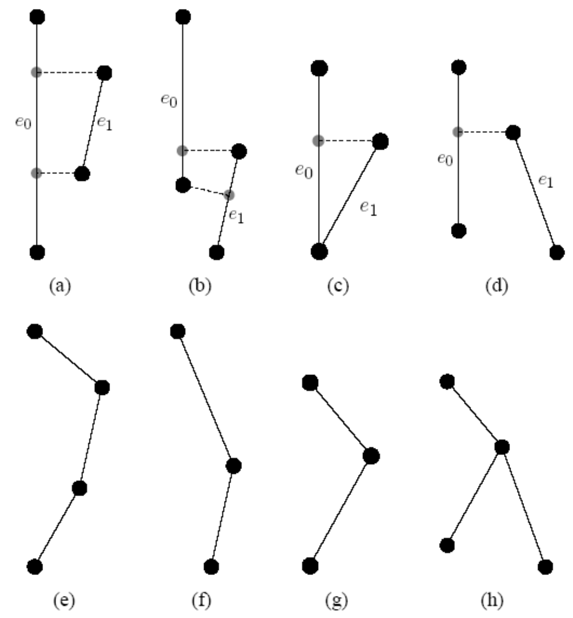


Fig 2.1.4: The four different types of overlap and how to join the overlapping edges [3]

There are some additional improvements that can be made to the generated roadmap. The first one is to make sure that a path exists through every passage between two objects in the environment. This can be achieved by applying techniques for finding paths in narrow corridors [17] or allowing cycles in the roadmap [4]. The latter will be discussed in the chapter about pre-processing for dynamic environments, but we will briefly explain the first technique here.

A way to find paths in narrow corridors is by using the bridge test [17], which increases sampling density in narrow passages. A bridge is a line segment that runs between two points that collide with an object in the environment (e.g. are located in collision space as opposed to collision-free space), thus crossing a gap. Those points are the originally sampled points. In the middle of the bridge another point is added, which, if collision-free, can be used as a node for the roadmap. If a line segment is of short length, it means there is a small passage. Using only the bridge test to generate a roadmap will result in a very sparse roadmap outside of the narrow areas, because nodes will only be placed in the middle of two objects, so it is best combined with another (more uniform) sampling technique. This brings us back to the subject, namely the Voronoi diagram based approach. Adding the bridge test to the sampling process will result in more useful nodes to retract to the Voronoi diagram.

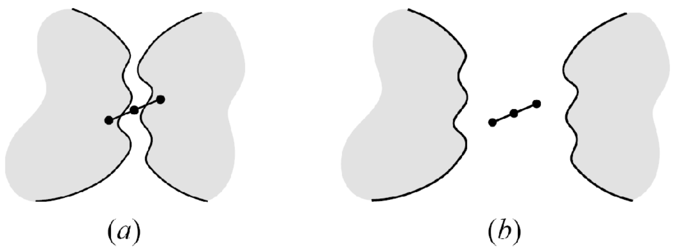


Fig 2.1.5: Building a short bridge between two nodes that are in collision is easier with a narrow passage (a) than it is with a wide space (b). Note that the 'bridge' pictured in (b) is not a correct bridge, since the end nodes are not in collision. By favouring short bridges, node density increases in smaller passages like (a). [17]

The other possible improvement to the roadmap is purging redundant nodes. If a node has two neighbouring nodes that can be directly connected by an edge that has enough clearance from the surrounding objects and is collision free, the middle node and its edges can be removed from the roadmap and be replaced by a newly created edge between its two neighbours. This leads to longer edges, which in turn results in less unnecessary rotations.

Subsequently, circular blends are added to create C^1 continuity, because our roadmap still consists of only straight line segments with sharp angles. This smoothing is important for our goal, because it will ensure the camera travels on a path that feels natural to the user. Smoothing using circular blends results in every part of an edge consisting of either an arc or a straight line, which decreases the need for complicated mathematical formulas. In order to determine how to smooth parts of the roadmap, the degree of a vertex needs to be known. The degree of a vertex is simply the number of edges connected to it. For a vertex with degree 1, no smoothing is done, because it is either an ending point or a starting point. If a vertex has degree 2, the blending is done by finding the centers of the two edges, and connecting them by a circle arc that touches both edges. For a degree higher than 2, we must find the centers of all incoming edges, and add a blend for every possible pair of these edges. So for 3 edges, there will be 3 blends. For 4 edges, there will be 6 blends, and so on.

With these blends added, a problem may occur. It is possible that an arc does not have enough clearance from surrounding obstacles because, unlike the original straight edges, it has not been retracted to the Voronoi diagram. If this is the case, we simply replace the blend by another blend that has a smaller radius until our minimum clearance is maintained.

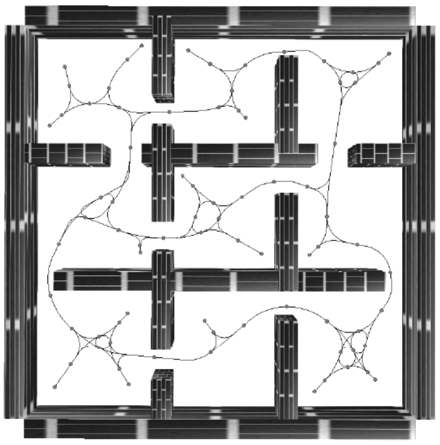


Fig 2.1.6: Results of using a roadmap generated by this PRM.
Note the pre-calculated circular blends. [3]

A limitation on this algorithm is that it is based on a two-dimensional representation of the environment. It is possible to calculate spatial Voronoi diagrams [9], but this is more complicated and will also complicate the circular blending process. However, for this application, a fixed height leading to a two-dimensional roadmap computation will also suffice in most cases.

Advantages:
- Path smoothing is done in the pre-processing phase, which speeds up the path finding phase.
- Combines advantages of PRM with those of a deterministic road mapping method.
- Most elaborate approach so far.

Disadvantage:
- Based on a 2D representation of the environment, so it is not fully 3D.

Choosing an alternative sampling method

In a comparative study of PRM planners, Geraerts and Overmars [16] compare a lot of ways to solve separate road mapping issues. One of those issues is selecting the best sampling method. For a range of test cases, several sampling methods were compared. The standard random sampling method was compared to a grid sampling method, Halton sampling [18] and cell-based sampling. Grid based sampling is not the same as a completely grid-based roadmap, but it samples points that are positioned in a grid and thus results in less redundancy than in a completely grid-based roadmap. On the other hand, it also results in less coverage in some narrow passages. Halton sampling uses points from the Halton point set, which is used in discrepancy theory to obtain a good coverage of an area. This method is better than using a grid. However, note that unlike the other sampling methods discussed here, Halton sampling is deterministic! In a cell-based approach, random configurations are taken within cells of decreasing size in the environment. The first sample is generated somewhere in the entire space. Next, the workspace is split into 2^3 equally sized cells. In a random order, a configuration is generated in each space, after which the spaces are subdivided again, etc.

After comparing these four methods, it was not clear which one was better in general. The random methods can have very different running times for each run of the same environment, while the Halton method is more consistent. The differences in running times given are not large enough to make a clear selection, although it appears from their results that a basic random method is in general the most efficient way to go. The problem with random is that results may and will vary. This makes it interesting to study Halton points, because they will generate more consistent results.

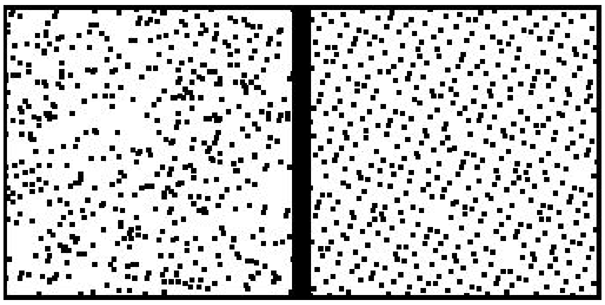


Fig. 2.1.6: Random sampling versus sampling on a Halton point set. The latter results in a more uniform sampling with a random feel to it [18]

Both Hammersley and Halton methods produce a uniformly distributed set of points, but the main advantage of Halton points over Hammersley points is that Halton points appear to be more random. With Hammersley points, it is often possible to directly spot a pattern in the set of points. This is something we want to avoid, because patterns look artificial instead of natural and might distract the user. Halton points are the hierarchical version of Hammersley points, which makes it possible to use them incrementally [18]. This means that in order to generate Halton points, we must first be able to generate Hammersley points.

Each non-negative integer k can be expanded using a prime base p as follows: $k = a_0 + a_1 p + a_2 p^2 + ... + a_r p^r$, where each a_i is an integer in $[0, p-1]$. Now we define a function Φ_p of k by

$$\Phi_p(k) = \frac{a_0}{p} + \frac{a_1}{p^2} + \frac{a_2}{p^3} + ... + \frac{a_r}{p^{r+1}}$$

If $p = 2$, the sequence of $\Phi_2(k)$ for $k = 0, 1, 2, \ldots$, is called the Van der Corput sequence. When d is the dimension of the environment that is to be sampled, any sequence $p_1, p_2, \ldots, p_{d-1}$ of prime numbers defines a sequence $\Phi_{p_1}, \Phi_{p_2}, \ldots, \Phi_{p_{d-1}}$ of functions. Their corresponding k-th d-dimensional Hammersley point is

$$\left(\frac{k}{n}, \Phi_{p_1}(k), \Phi_{p_2}(k), \ldots, \Phi_{p_{d-1}}(k)\right) \text{ for } k = 0, 1, 2, \ldots, n-1,$$

where $p_1 < p_2 < \ldots < p_{d-1}$ and n is the total number of Hammersley points.

The Hammersley algorithm is not hierarchical due to the first coordinate k/n, which results in different sets of points for different values of n. In a Halton algorithm, this is solved by replacing it with another Van der Corput sequence based on prime number arithmetic, resulting in three (the number of dimensions needed) of those sequences with different prime numbers p_1, p_2 and p_3, to insure independence of n by construction. The resulting formula for generating Halton points is

$$\left(\Phi_{p_1}(k), \Phi_{p_2}(k), \Phi_{p_3}(k)\right) \text{ for } k = 0, 1, 2, \ldots, n-1.$$

As can be seen in figure 2.1.6, this sampling method looks like a random sampling method, but it is deterministic and more evenly distributed. This combines the best of both worlds, namely natural looking patterns, consistent running times and an evenly spaced distribution. Still, there is no proof that using Halton points has any real speed advantages over using standard random samples. However, we will use Halton points to obtain optimal and consistent results. Another advantage of the deterministic nature of Halton points is that it enables you to test the same roadmap with different environment setups. This makes it easier to debug certain things, mainly path planning algorithms.

Advantages:
- Using Halton points results in more consistent running times.
- The distribution of sample points is more evenly spaced than it is with a random method.
- Can be faster than random sampling and more efficient than grid sampling.

Disadvantage:
- No clear speed advantage.

Conclusion

There are a lot of road mapping methods that result in a functional roadmap for a static environment. Most of them rely on some form of random sampling to create a natural looking distribution of nodes without the use of deterministic algorithms. Halton sampling [18] is a method of generating samples that look similar to random samples, but they are distributed more evenly and calculated by a deterministic algorithm. This results in more consistent running times and better coverage. For most probabilistic road mapping methods, Halton sampling can be a good alternative to random sampling.

The probabilistic road mapping method that results in the most efficient and good looking roadmaps is the approach based on Voronoi diagrams [16]. The bridge test [17] can be used with the Voronoi method to make sure the roadmap is able to handle narrow passages. Unfortunately, the Voronoi approach is based on a 2D representation of the environment and we need to generate a roadmap for a 3D environment.

We will use a PRM algorithm based on Halton sampling to generate a static roadmap, and smooth the path during the post-processing phase. Bridge testing could be a useful addition to the road mapping phase to ensure node coverage of small passages.

2.2. Pre-processing for dynamic environments

The main difference between static and dynamic environments is that a generated roadmap can quickly become useless. A roadmap, in its most simple form, only has edges when the two nodes that make up the vertices of the edge can be connected. When an object is moved, new edges may be needed and old edges might become obstructed. This is not automatically reflected in the roadmap, so it becomes useless. With more specialised approaches, like the one using the Voronoi diagram of an environment as a guide, this effect is even worse because the sampling points are not evenly distributed. A big object might cause a large area to be without sampling points. When such an object is moved, the area where it originated from can become useless, because it does not contain any sampling points.

There are some ways in which the algorithms for static environments can be adapted for use in dynamic environments. The most obvious one would be to let the user manually recalculate the roadmap whenever the scene is changed. A better way would be to have a thread running in the background doing the same thing whenever a change in the environment is detected, and replacing the old roadmap by the new one when it is completely generated.

The problem is that the pre-processing phase is, in most cases, the most time-consuming part of the entire path finding process. For the sake of clarity, we will define two types of dynamic environments. Fully dynamic environments, in which any object can move at any time in any direction, and semi-dynamic environments, in which objects travel in known paths and patterns. The main difference between the two is that in semi-dynamic environments, it is possible to predict the position of the object at a certain time in the future. This information can be used in addition to a special type of roadmap, to find paths without having to calculate a new roadmap every time the environment changes. A method to do this will be discussed in this chapter.

No new road mapping algorithms will be presented here, only additions and alterations to the previously discussed road mapping methods for static environments.

Roadmap-based motion planning in semi-dynamic environments

Van den Berg and Overmars [5] propose a method for motion planning in semi-dynamic environments that is based on motion prediction. The scene is divided into a static part and a dynamic part, which is only possible for semi-dynamic environments. We need a method for motion planning in a fully dynamic environment, but since a method for a semi-dynamic environment might offer some insightful tips or algorithms, we will discuss it here.

An interesting thing to note is that this paper was written with robotics in mind, and most of the examples use a 2D roadmap. However, this method works for 3D roadmaps as well.

The term “configuration space” is often used by road mapping methods for static environments. When motion is introduced, another dimension is added to the equation: time. Instead of a configuration space *C*, we will now use a state-time space *C*, to be consistent with other literature. State-time space consists of pairs (x, t), where x is an element of *C* describing the robot’s state (or camera state, for our application), and t is a scalar denoting the time. The robot is represented in state-time space, and all objects (static and dynamic) in the environment are converted into static objects in state-time space, called space-time obstacles. Finding a path through state-time space alone is not sufficient, because the robot also has to be confined to other constraints and cannot go back in time. To accentuate the difference, a path obeying the dynamic constraints is called a trajectory.

A roadmap is generated for the static objects in the scene in order to reduce the need for collision checks during run-time. In this paper, they say a roadmap with smooth paths is preferred here as well. Also very important is the fact that the roadmap needs to contain cycles to insure there are alternative paths that can be found. How roadmaps with cycles can be generated will be discussed later in this chapter.

A robot *R* will have a maximum velocity v_{max} in the environment, and there is a roadmap available for the static part of the environment. Let *s* and *g* be the start and goal nodes in the roadmap, and t_0 be the starting time. The goal of the path finding algorithm will be to find a trajectory starting at *s* on t_0, and reaching g as quickly as possible without any collisions.

It is hard to tell which parts of this method fall under pre-processing and which parts fall under path planning, but we will continue discussing this method in the path planning chapter for dynamic environments because the paper concentrates on that area.

Advantages:
- Separates static objects from dynamic objects, speeding up the path finding process.
- Almost any road mapping method can be used for the static part of the environment, as long as it is smoothed and contains cycles.
- Finds the fastest path instead of the shortest one. More about this in the path finding chapter.

Disadvantages:
- Only usable for semi-dynamic environments which have been completely defined and planned, including all motions that will occur. This means no user motions are accepted for dynamic objects in the environment.

Generating roadmaps with additional cycles

One of the prerequisites for the previously discussed method was a roadmap with cycles. While most road mapping methods, like the ones previously discussed, automatically generate cycles due to the nature of a roadmap, it is sometimes useful to add even more cycles to a roadmap. Cycles are actually very useful in roadmaps for dynamic environments. Not only do they increase the number of path possibilities, which is useful for dynamic environments, but a roadmap with additional cycles also results in shorter paths compared to some of the PRM approaches with less variation between different runs of the algorithm.

Nieuwenhuisen and Overmars [4] describe how to add cycles into a roadmap. The reason roadmaps do not have cycles most of the time, is that it involves expensive collision checks, without resulting in more coverage of the free area. The writers propose a connection strategy resulting in a graph with cycles, while keeping the number of collision checks low.

Just adding edges to the generated roadmap *G* will result in a high processing time, so a selection of useful edges must be made. We use a value *K* to determine the degree of usefulness. An edge $E(c, c')$ is *K*-useful, when $K * d(c, c') < G(c, c')$, where $d(c, c')$ is the distance between *c* and *c*', and $G(c, c')$ is the distance needed to go from *c* to *c*' in the graph. Only cycles that improve the graph distance between *c* and *c*' by a substantial factor *K* will be added to the graph. A smaller value of *K* will add more cycles to the graph, while a larger value will add less. If $K < 1$, all edges are allowed, while no edges are allowed when *K* is infinite.

However, computing $G(c, c')$ is a big problem in itself, since adding an edge to the roadmap can influence all graph distances. This means that a shortest path algorithm (like the Dijkstra algorithm) has to be executed for each edge that is added, which can dominate the running time when there are a lot of vertices in the environment. The way to speed this up, is to remember that we do not need the exact distance to all other vertices, like in the Dijkstra algorithm, but we only want to know if $G(c, c')$ is larger than $K * d(c, c')$. In order to narrow down the needed calculations, we use in-between vertices and absolute distances to estimate path lengths until either c' is reached, in which case we do not add the edge (because it would be useless), or the estimated distance for a vertex is larger than $K * d(c, c')$, in which case we do add the new edge to the roadmap. Because, since the estimate is always the correct minimum distance, we can use that information to determine whether the actual graph distance can be reduced by adding an edge. This pruned version of the Dijkstra algorithm is called the usefulness test.[2]

Advantages:
- Makes roadmaps more useful for dynamic environments because of the alternative paths that are possible when cycles are added.
- Results in shorter paths.
- The basic idea of a roadmap with cycles is useful for fully dynamic environments.

Disadvantage:
- Roadmap calculation takes longer for static and semi-dynamic environments.

Using a fully connected graph as a roadmap

Based on the previous topic, a fully connected graph is a graph in which each node is connected to all other nodes. This makes it unnecessary to add cycles, because all possible edges are already present in the roadmap. However, because it is a fully connected graph, no collision checks are done during the pre-processing phase. That would simply take too much time, especially in a dynamic environment, where they would have to be recalculated very often. Instead, the collision checks should be performed during the path finding phase, because that is when the last known configuration of the environment is active. This is a very primitive method that is also rather inefficient for use in static environments. n nodes will result in $n(n-1)/2$ edges, which means the complexity is quadratic.

Another issue is the technique that is used for generating the nodes for a graph like this. While random and grid-based point sets immediately come to mind, the Halton set [18] discussed earlier is also a candidate. Because the environment should be covered as much as possible, a completely random set of nodes is not a good idea. It does not guarantee coverage of an area (some points might be concentrated in a specific small part of the environment, while other parts remain node less). Halton points are again a good way to go, but the road mapping stage is not the problem with this method.

The roadmap will be too complex to use for larger environments, since the path finding algorithm has a lot of options to explore, and has to perform collision detection as well. This is why most other methods only connect nodes that are within a certain distance from each other. We will not use this method, which is an example of sacrificing accuracy for speed. This method might result in the shortest and most natural paths in the end, but the process it too time-consuming.

Advantages:
- Easy to implement.
- Fast pre-processing phase because there is no collision detection.
- Good connectivity and a lot of alternate paths can be found.
- Can be used with a lot of sampling methods.

[2] Although this is a path finding algorithm, it is needed to generate the roadmap. Therefore it belongs in this section of chapter 2 and not in the path finding section.

- Also useful for fully dynamic environments.

Disadvantages:
- Needs an adapted path finding algorithm with collision detection (which will be discussed in the upcoming Lazy PRM section), which makes it slower.
- Path finding algorithms will have too many options to explore, and no idea which options have potential and which ones do not. This results in a very high complexity, which is something that should be avoided in the path finding stage.
- Very inefficient for use in any real-time application.

Customizing roadmaps at query time

While this paper by Song, Miller and Amato [19] is not about dynamic environments, it does offer a concept that looks interesting and might even be useful for fully dynamic environments.

They start off with a very coarse roadmap, and refine and validate it in the path finding stage. This is a concept that is interesting for semi-dynamic environments, because it means you could calculate a course roadmap for the static objects in the scene and include the dynamic objects in the path finding phase. It is, however, also useful for fully dynamic environments. Since validation also occurs in the path finding phase, a very course standard graph could be used for the environment, which could be more specifically filled in during the path finding phase. The most basic version of this is called "Lazy PRM" [20], which we will discuss later on. A slightly more advanced version of this is Fuzzy PRM [21], but this technique will not be discussed because it involves node validation in the pre-processing phase.

The technique used for creating the customized roadmaps suggested in this paper is called C-PRM. While neither nodes nor edges are validated during the pre-processing phase when using Lazy PRM (and only nodes are validated when using Fuzzy PRM), C-PRM uses approximation methods to 'validate' nodes and edges in the pre-processing phase. The reason for this is that it makes the algorithm work better for environments with narrow passages and other tricky features.

During the pre-processing phase, the clearance can be calculated for each node and each edge. This speeds up the path finding process, which can now search for the nodes with the highest clearance. They also suggest a grid-based distribution for the nodes can be used in some scenarios. Edge evaluation can be done by using binary resolution approximation or overlapping spheres approximation. Binary resolution approximation is when the midpoint of an edge is evaluated, which means that the algorithm has to determine whether the midpoint is in c_{free} or not. This is followed by an evaluation of the midpoints of the resulting sub-segments if no collision has occurred, and so on. This results in many collisions being detected early on. Overlapping spheres approximation uses clearance information stored in the nodes. A sphere is created for each node, the radius of which is the maximum radius that results in a collision-free sphere. When two spheres overlap, it means there is a (direct) collision-free path between the two corresponding nodes and an edge can be created. However, when there is no overlap, that does not mean a path between the two nodes does not exist. The algorithm does not continue searching after the collision sphere check, so these potential edges will not be found.

Since we wish to find a solution that works with fully dynamic environments, validation in the pre-processing stage seems like a useless thing to do. Still, some of the techniques discussed in this paper may prove useful, so we will get back to it in the next chapter.

Advantages:
- Very flexible system with a faster pre-processing phase but a slower path finding phase.
- Introduces some interesting approximation techniques.

Disadvantage:
- Not suited for fully dynamic environments in its current form.

Lazy PRM

The Lazy PRM algorithm by Bohlin and Kavraki [20] was designed to be used by industrial robots, some of which have over sixty degrees of freedom. The main reason for creating the algorithm is that industrial environments require fast single path finding queries without an extensive pre-processing phase. This is because configuration space changes frequently, due to the robot grabbing a tool or material, or a new object entering the environment. It is interesting to note that the authors make the following statement in their literature discussion: "Most implemented PRMs show that it is computationally more efficient to distribute nodes densely and use a relatively weak, but fast, local planner.". This basically means that it is better to have a lot of unexplored options and a fast path finding algorithm, than have a well-connected roadmap with few nodes (and thus also fewer options to explore when the configuration space is altered). This is similar to the grid density issue we discussed earlier in a way. Figure 2.2.1 shows three different node densities used in Lazy PRM roadmaps and their results. The last two densities can both be used successfully, the choice mainly depends on the requirements of the software. If high accuracy is more important than speed, the third option is the best way to go. However, for our application, speed is the most important factor.

The road mapping phase in the Lazy PRM method is very basic and does not involve collision detection. We can use most of the methods described above to generate nodes and edges for a roadmap. The Lazy PRM method focuses more on the path finding stage, which will be discussed in the next chapter.

Advantages:
- No intensive road mapping phase needed.
- Can be easily adapted for use in fully dynamic environments.

Disadvantages:
- Path finding takes longer. Path finding is a more crucial phase than road mapping, because it is real-time. However, a longer path finding stage is pretty much unavoidable for fully dynamic environments.

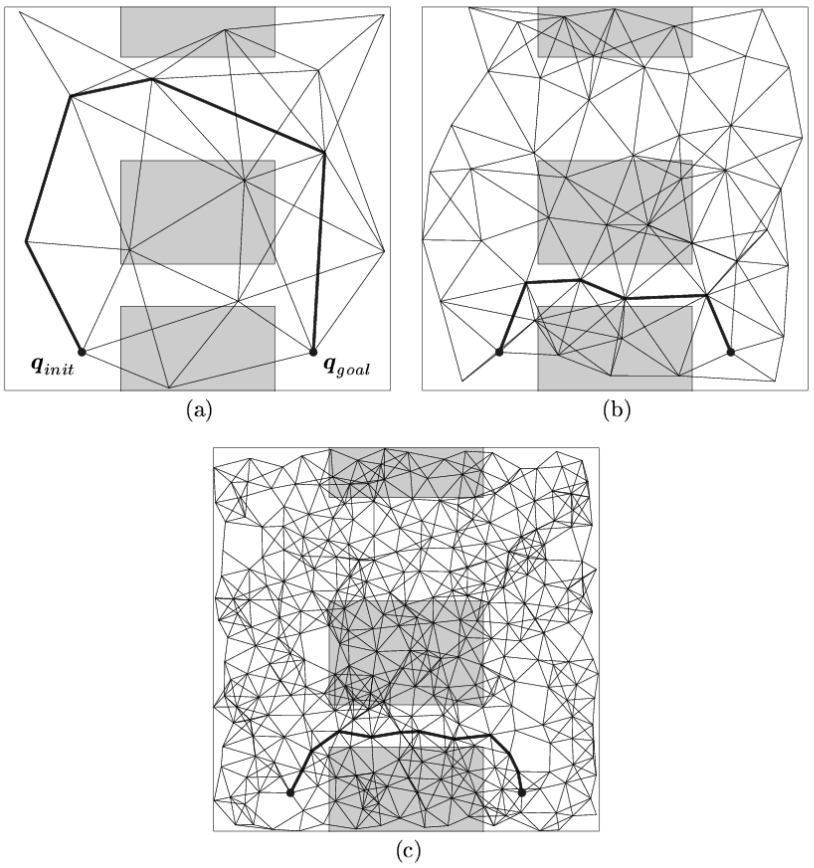


Fig. 2.2.1: Lazy PRM for a two-dimensional environment with three obstacles. The roadmap in (a) is too sparse to find the shortest feasible path, while the roadmap in (c) has a higher density than needed, which results in more calculations. [22]

Dealing with narrow passages

Dealing with narrow passages is an important factor in generating useful roadmaps. We have already discussed the bridge test [17] in the previous chapter, but in the resulting roadmap there is no longer a uniform distribution of nodes in the parts without narrow passages, which could lead to unnatural paths. Also, it is used in the road mapping stage. The basic theory behind this algorithm could be useful in the path finding phase for a fully dynamic environment, but it is good to take a look at alternatives because this is such an important process. Lack of a good algorithm to handle narrow passages will severely limit the usability of the software. For a sandbox virtual environment, it is not really necessary most of the time. But in more conventional environments (e.g. levels for a computer game, architectural design), corridors, windows and doors are often used and could be the only way from point A to point B in certain cases. Or the narrow passage could be part of the shortest path, which cannot be found without a decent algorithm for dealing with narrow passages. A longer path, which could be ten times as long, will be the end result when there are no edges directly connecting the two sides of the passage in the shortest path..

So it might be a good idea to look at another approach to see if it could be better suited for use in a fully dynamic environment. Song, Miller and Amato [19] refer to a paper [23] by D. Hsu, L. E. Kavraki et al. when they mention PRM methods that are aimed at solving problems with narrow passages. One of the writers, D. Hsu, also contributed to the paper on the bridge test a few years later, which leads to believe that the bridge test is a more evolved version of the method discussed in this paper. Since we might still be able to use some of the basic principles we can find in this paper, it will be discussed now.

This paper discusses two road mapping algorithms, of which the first one is a basic PRM algorithm and the second one is more interesting. In the second algorithm, a roadmap is generated in two stages. First, a dilated version (*F'*) of the free space (*F*) is generated. This basically means that the borders of the free space will expand. This broadens the narrow passages, which is the whole reason behind the dilation. Next, a roadmap *R'* is generated for *F'*. Some nodes in *R'* will be located outside of the original *F*, so they will be repositioned by resampling, resulting in the final roadmap *R*. A couple of those nodes will now be located inside a narrow passage, thus enhancing connectivity for the roadmap.

Advantage:
- The basic idea of dilation could be useful for fully dynamic environments, when used locally.

Disadvantages:
- A lot more difficult to implement than the bridge test.
- Seems to be more processor-intensive as well.

Conclusion

A roadmap for a fully dynamic environment has to be flexible. For this reason, the pre-processing phase will not have to do much more than generating a roadmap with uniformly distributed nodes. The real tests will take place in the path finding stage, which will make use of temporary, dynamic (sub-)roadmaps, like a roadmap generated by the Lazy PRM algorithm [20]. Also, the roadmap will have to keep a lot of path options open, so it can sometimes be beneficial to add additional cycles to the roadmap.

2.3. Path planning

When the user-selected target position is known to the application (in this case, that means the position is calculated by the camera position determination algorithm), the path planning and post-processing phases come into action.

Contrary to road mapping, path finding is pretty straightforward and well-defined. The start (current) position and goal position are added as nodes which will be connected to the generated roadmap. If these nodes end up in the same connected part of a roadmap, a path exists. This path can be calculated in different ways, which will be discussed here.

Some of the papers describing the road mapping methods we discussed also state a preference regarding path finding algorithms. For instance, the A* algorithm (or a similar method) is used in [3, 4, 5], while [6] suggests an edge-based version of the Dijkstra algorithm and [2] uses the IDA* algorithm.

The Dijkstra algorithm

While the Dijkstra algorithm [24] itself is rarely used, most other path finding algorithms are based on it, so it is important to have some knowledge about this algorithm.

The environment is represented by a weighted graph, called a roadmap. Nodes represent locations, while edges represent direct connections between locations. The weight of an edge is equal to a certain cost that is needed to travel between the two nodes that are connected by that edge. That cost can be time, distance, fuel usage or even scenery (a more beautiful environment has a lower cost), but we will use distance as the cost of an edge. The goal is to find the shortest path between two nodes, in this case the start and goal position nodes that have just been added.

The Dijkstra algorithm finds the shortest distance from a certain node to all of the other nodes in the graph (which means it can be heavily optimized, as we will see). A precondition is that all the weights are non-negative and not zero, but that is never the case with roadmaps, so no additional checks will have to be performed. The complexity of the Dijkstra algorithm is $O(N^2)$, which means it grows quadratically with the amount of nodes. This can be a problem in a relatively large roadmap which needs to be searched in real-time.

Two values have to be found for each node v. We need the length $g[v]$ of the shortest path from v to the starting node, and we need the node $ed[v]$ which was the last node on the shortest path used to reach v. With these values calculated, we can determine which path is the shortest path to a certain node, the cost of the shortest path, and the order in which the nodes in the path are traversed.

The pseudo-code for Dijkstra(G,w,s) is as follows [24]:

G	= graph with node-set V
w	= weights of edges between nodes in V
s	= starting node, an element of V
Q	= priority queue with nodes of which the minimal distance to s has not been calculated yet, sorted by their weight g
R	= set of nodes, of which the shortest distance to s has already been calculated

Q.ExtractMin() returns the vertex *v* in *Q* with the smallest *g*[*v*] value.
Q.DecreaseKey(node, length) replaces the length value of a node in *Q* with a smaller one.

```
$R = \{\}, g[s] = 0, Q = \{s\}$
**for all** $v \in V \setminus \{s\}$ **do**
    $g[v] = \infty$
    $Q$.insert($v$)
**while** ($Q \neq \{\}$) **do**
    $u = Q.ExtractMin()$
    $R$.insert($u$)
    **for all** $v \in$ Neighbours($u$) **do**
        **if** ($g[v] > g[u] + w(u,v)$) **then**
            $g[v] = g[u] + w(u,v)$
            $Q.DecreaseKey(v, g[u] + w(u,v))$
            $ed[v] = u$
```

When a result is found, the shortest path can be reconstructed using this algorithm:

s	= starting node
d	= destination node

```
$P = \{d\}, c = d$
**while** $c \neq s$
    $P$ = ed[$c$] + $P$
    $c$ = ed[$c$]
```

Like stated before, it is not such a good idea to use the Dijkstra algorithm in our application. The first problem is that it finds the shortest path to every other node. This can easily be changed by halting the algorithm when the path to the destination node is found, but that is not sufficient yet. Another problem is that all nodes in the graph are added to *Q*, which makes searching and adding to *Q* expensive operations. This can be solved by only adding the neighbouring nodes of the node that is being researched. But it is still not optimized enough.

Advantages:
- Finds the shortest paths from the starting node to all other nodes.
- Unambiguous: No result for a certain query means there is no path between those nodes in the roadmap.

Disadvantage:
- Computationally expensive because of redundant searches.

The A* algorithm

This is an altered version of the Dijkstra algorithm which incorporates the solutions to both problems mentioned in the previous paragraph, and adds a heuristic. What this means, is that an additional value is used to determine which node is closest to the destination point, so the search can start with that node. The heuristic value is generally the straight-line (Euclidian) distance between a node and the destination node. Generally, this increases efficiency, but in a worst case scenario the complexity is equal to that of the Dijkstra algorithm. For example, if you have a maze in which the correct solution would require you to start by moving to the left. However, since going to the right seems faster, the wrong path will be chosen and it will have to be retraced back to the start.

Still, the A* search is a better choice than the Dijkstra algorithm. So, what does it do? Like stated before, the algorithm introduces a heuristic $h(u, v)$, which in this case is the Euclidian distance between a node and the destination node. For each step, the node with the minimal sum $f[v] = g[v] + h[v]$ will be selected.

The actual pseudo-code algorithm for AStar(G,w,s,d) will look like this:

(all characters have already been defined in this and the previous paragraph)

```
$R = \{\}, f[s] = g[s] = 0, Q = \{s\}$
while ($Q \neq \{\}$ and $d \notin R$) do
    $u = Q.ExtractMin()$
    $R$.insert($u$)
    for all $v \in$ Neighbours($u$) do
        if ($g[v] > g[u] + w(u,v)$) then
            $ed[v] = u$
            $g[v] = g[u] + w(u,v), f[v] = g[v] + h(v)$
            if $v \in Q$
                $Q.DecreaseKey(v, g[u] + w(u,v) + h(v))$
            else
                $Q$.insert($v$)
```

Generating the actual path is done in the same way as it is done in the Dijkstra algorithm.

Like stated before, the worst-case complexity of the A* algorithm is equal to that of the Dijkstra algorithm. In general, however, the complexity of the A* algorithm is much lower. If h(u, v) is an underestimation of the actual distance, A* will calculate the minimal distance. This is the case when the Euclidian distance is used as a heuristic. On a side note: if $h(u, v) = 0$ for each (u, v), the A* algorithm is equal to the Dijkstra algorithm.

Advantages:
- Always finds path with minimal distance when used with Euclidian distance.
- In general, is pretty fast and of less complexity than the Dijkstra algorithm.
- Can be easily altered for use in different situations [24, 25] (e.g. environments with unexplored areas, variations in maximum terrain speeds)

Disadvantage:
- Still has the same complexity as the Dijkstra algorithm in very complex environments.

The Iterative Deepening A* (IDA*) algorithm

Since this algorithm is the preferred choice of B. Salomon et al., in their paper about path planning in complex environments [2], we will take a look at what makes it different from a regular A* algorithm.

In the IDA* algorithm [24], nodes with a value of *f*[*v*] above a certain treshold will be removed from the queue. If no result is found, the search starts again with a higher treshold. This process is repeated until two consecutive resulting paths are of the same length.

Advantage:
- Less memory is needed for this algorithm compared to the original A* algorithm.

Disadvantages:
- Takes longer to calculate, thus less useful for our application, because the path finding phase should be real-time and the user should not have to wait for a noticeable amount of time before the camera starts moving.

Bi-directional search

Some people, like me, solve a maze by going from finish to start instead of the other way around. Whether this makes sense or not is not the issue here, but it does translate to yet another basic idea that might be useful. Intuitively, in a complex environment it might be a good idea to start searching in both directions together until the paths of both searches cross. Of course, we were not the first ones to think of this, which means there is literature about it that can be discussed.

To find out if this could be useful, we created a test environment in an A* search example program. After letting the program find a path, we switched the start and destination nodes and noticed a pretty big difference in the amount of nodes being searched.

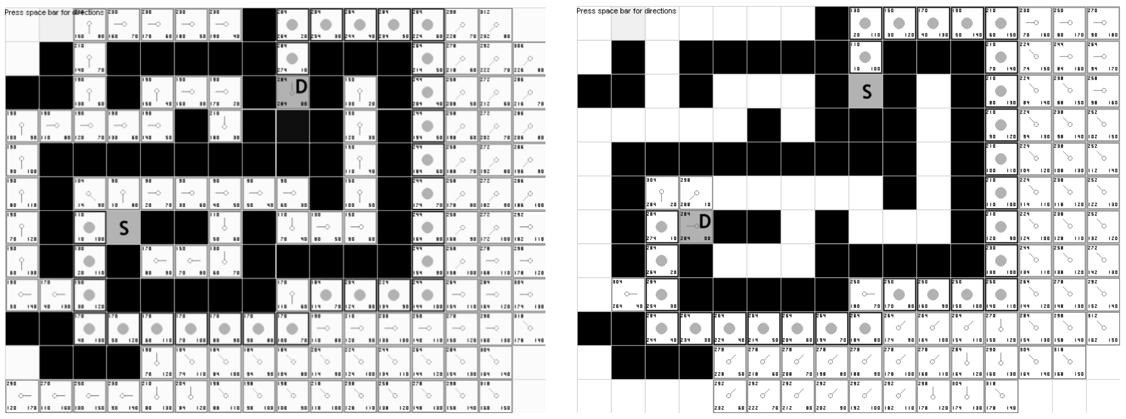

Press space bar for directions
D
S
Press space bar for directions
S
D

Fig 2.3.1. In this example, swapping the start and destination positions results in the same path, but less nodes need to be visited before the final path is obtained [25].

There are two main ways of performing a bi-directional A* search [26]. With front-to-front bi-directional search, the two searches (forward and backward) are done simultaneously and linked together. Instead of choosing the best forward search or backward search node, the algorithm chooses a pair of nodes (x, y) with the best $g(s, x) + h(x, y) + g(y, d)$. This is the type of search I had in mind.

However, there is also another type of search, called the retargeting approach. In the retargeting approach, a forward search is performed for a short time. When the time is up, the node that is closest to the goal will be selected. Now the backward search starts, but it does not search for the starting point. Instead, it now searches for the node that was selected after the forward search. The backward search is also halted after a specific time, and another candidate node is selected. This time, that is the node closest to the forward search candidate node. Next, the forward search will search for a path between the forward and the backward search candidate, and this process will continue until the two nodes are equal.

Advantage:
- Speeds up the search in complex environments.

Disadvantages:
- Will slow down the search process in less complex environments (e.g. sandbox or modelling environments). However, this is irrelevant since the total time required by the search process will still make it usable in real-time.
- 3D environments generally have more path options than 2D environments, which makes bi-directional search less useful.

Conclusion

These algorithms can be used for all types of environments, but A* search [24, 25] seems like the best choice for most (static) environments. Only in a really complex environment will bi-directional search be noticeably faster [26]. In the next chapter, we will take a look at fully dynamic environments and see if they can use these algorithms as well.

2.4. Path planning in dynamic environments

While the path planning algorithms from the previous chapter can also be used for dynamic environments, they need to be altered or expanded, depending on the road mapping algorithm used. This often includes doing collision checks on edges instead of relying on the road mapping algorithm to generate a roadmap without invalid edges. Roadmaps generally do not last long in a dynamic environment, which is why the path finding phase also needs to be handled differently. First, we will discuss path finding algorithms that go with road mapping algorithms that we talked about in chapter 2.2. After that, we will discuss some generic dynamic path finding principles.

Path planning in semi-dynamic environments

We now return to the method for road mapping and path finding in static environments that was described by Van den Berg and Overmars [5], so we can take a look at the path finding phase.

Like stated in chapter 2.2, the term "trajectory" is used instead of "path", because time is also a factor. Also, due to the dynamic part of the environment, it is not always the best choice to arrive in a node as soon as possible. Therefore, performing a Dijkstra-type search in the roadmap is not a good idea. Sometimes, the robot (or camera, in this case) will have to wait before moving to a certain node. The robot must arrive during the free interval of a node, which is defined as the maximum continuous segment in time in which a robot configured at that node is collision-free. Also note that is has no benefits to arrive later inside a free interval, so it is optimal to arrive at the beginning of the free interval.

The problem can be expressed by modelling each free interval for a node as a vertex in an implicit (directed) graph, called the interval graph. An edge exists in the interval graph when there is an arc connecting the corresponding nodes in the roadmap, and a suitable trajectory exists between the intervals connected by the edge. In this graph, a suitable trajectory from start to destination can be found.

The interval graph is searched using a modified A* search algorithm. The graph is searched by sending probes through the roadmap searching for a trajectory to the destination node. This is done on two levels. A local trajectory of a single arc in the roadmap, and a global trajectory, which is the full trajectory that can be obtained by analyzing the results from the local trajectory probes.

Also, there are two types of local trajectories possible: advancing and returning trajectories. Advancing trajectories are the most common ones, they move in the direction of the destination node. Returning trajectories return to the same node in a later free interval, which sometimes happens when a robot has to make room for a moving obstacle and has no other place to go.

Advantages:
- Finds the fastest path instead of the shortest path.
- Is the best way to go for semi-dynamic environments.

Disadvantage:
- Will not work for fully dynamic environments.

Dynamic A* (D*)

Once again, we will venture into the domain of robotics. In a fully dynamic environment with just a basic roadmap, the camera is basically an explorer, trying to find a path in an unknown environment, but using a roadmap as a guideline. When searching for a path finding algorithm to deal with fully dynamic environments, an algorithm called “Dynamic A*” seems to have the right name for the task at hand.

The premise of D* [27] (as Dynamic A* is most often called) is that a robot has a map that is either complete, empty or partially filled with information about the environment. This is very similar to the situation our camera is in when the path finding phase starts for a fully dynamic environment. The problem space is formulated as a set of states, which denote robot locations. States are connected by directional arcs (in our case, all arcs are bi-directional) which also have an associated cost. Also, every state except for the goal state has a pointer to a next state.

Like A* search, D* search also keeps an *OPEN* list of states. Every state has a tag that stores whether or not it is on that list (or a possible candidate for the list). Not only does every node have a heuristic, but every node also has a key function. The key function is the initial heuristic value of the node. The first heuristic value is stored in the key function, while the actual heuristic value may change over time. This enables the key function to classify a state on the *OPEN* list as either a *LOWER* state (when the key value is smaller than the current heuristic) or a *RAISE* state (when the key value is equal to the current heuristic). *RAISE* states are used to get information about path cost increases, while *LOWER* states are used to gather information on path cost reductions. This is done by removing states from the *OPEN* list. When a state is removed, it is expanded to pass cost changes to its neighbours, which are in turn put on the *OPEN* list to continue the process. States are ordered by their key function, and the minimum key function from the *OPEN* list is used as an important threshold. If a path costs less than or just as much as the minimum key amount, the path is considered optimal. If not, the path may not be optimal. When the heuristic is the Euclidian distance, the resulting path is called the optimistic optimal path.

There is, however, a huge disadvantage to this algorithm for use in our application. The generated path will sometimes contain redundancy in the form of movement that looks like backtracking and (semi-)loops, which is not good for use with camera movement. It is not natural when the camera appears to explore the environment when the user considers it known. And since the user is able to see and manipulate the entire environment, he assumes the computer knows the environment. So no matter what the path finding algorithm is, the resulting path must appear to the user as being optimal. When a camera moves to the left and then back to the right when a shorter path would have been found if it just went to the right in the beginning, it performs a redundant move in the eyes of the user. Either the path will have to be post-processed to remove redundancy, or an other algorithm should be considered.

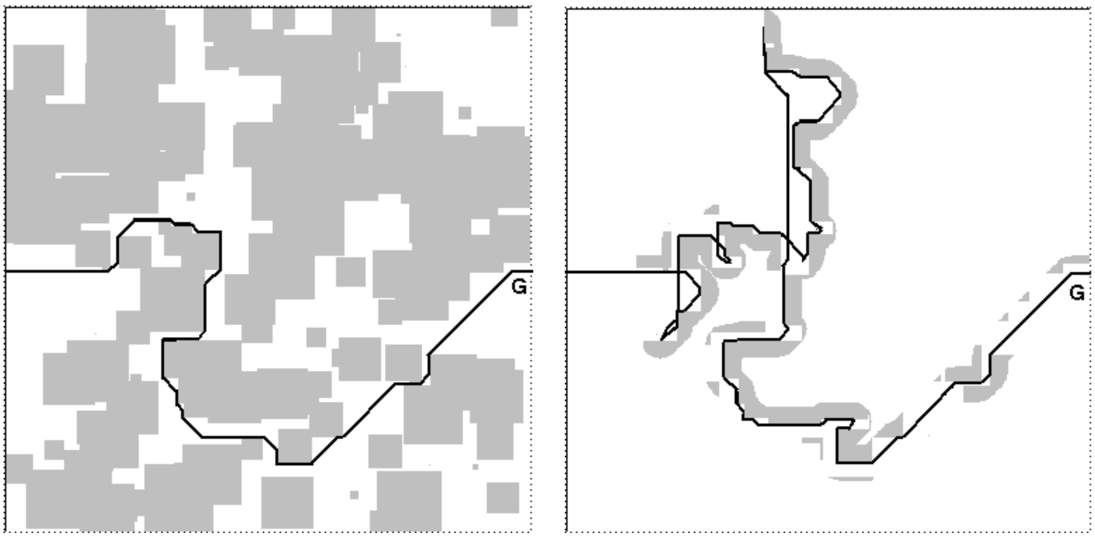


Fig 2.4.1: Result of the D algorithm for a completely known environment and an unknown environment, resulting in an optimal optimistic path with redundancy (especially in the top part) [28]*

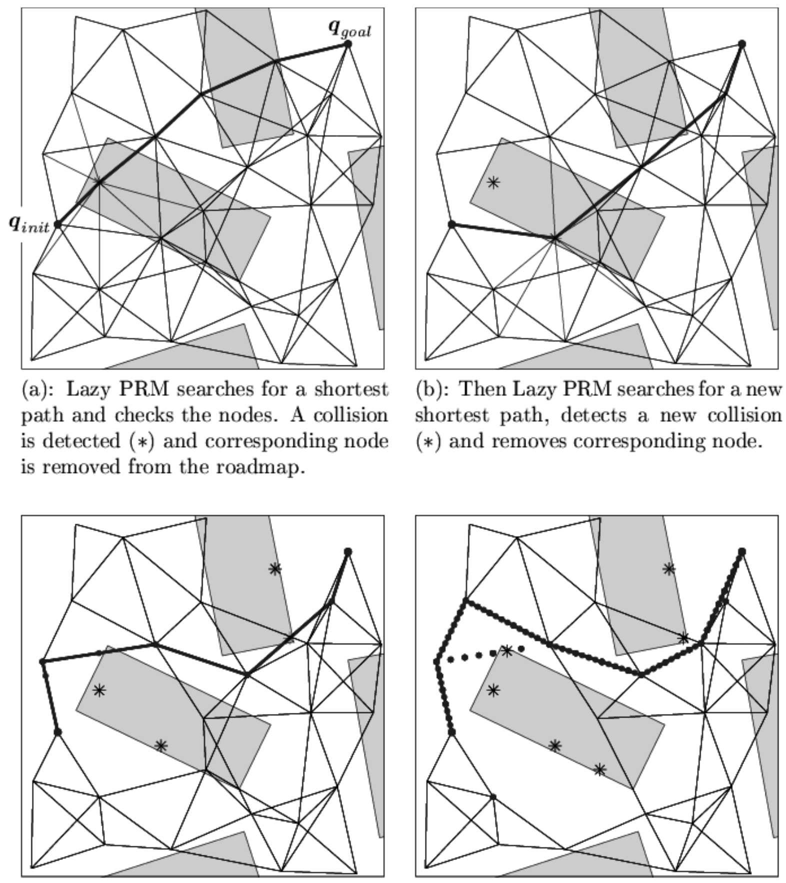
Advantage:
- Works well in fully dynamic and unknown environments.

Disadvantage:
- Might end up generating redundancy in the path, resulting in unnatural camera movement.

Lazy PRM, continued

We previously selected Lazy PRM [20] as the road mapping method of choice for use with fully dynamic environments. The paper on Lazy PRM also includes a path finding phase, the most important part, which uses the A* search algorithm. We will now discuss this path finding phase.

The main principle behind Lazy PRM is to minimize the number of collision checks. It works as follows: The roadmap consists of a number of uniformly distributed nodes, connected by edges where two nodes are sufficiently close together. Neither the edges nor the nodes have been tested for collisions. There is a starting position, (q_{init}) and a goal position (q_{goal}). By estimating the length of a path, the shortest feasible path is selected. This is done one edge at a time. When a collision occurs, the corresponding node and its edges are removed from the roadmap, and the search resumes. There are two ways in which this procedure can end. If a feasible path exists between q_{init} and q_{goal}, the shortest one will be found. When no feasible path is found, q_{init} and q_{goal} are located in two disjoint components of the roadmap. In this case, we can either report failure, or use a procedure called node enhancement to add relevant nodes to the roadmap.



(a): Lazy PRM searches for a shortest path and checks the nodes. A collision is detected (∗) and corresponding node is removed from the roadmap.

(b): Then Lazy PRM searches for a new shortest path, detects a new collision (∗) and removes corresponding node.

(c): After a few iterations, a sequence of feasible nodes is found. When checking the edges with a coarse resolution a collision is found (∗). The edge is removed from the roadmap, and the planner searches for a new shortest path.

(d): Eventually, the planner finds a path whose nodes are collision-free, and whose edges are collision-free to a certain resolution.

Fig 2.4.2: The Lazy PRM algorithm in action. [22]

Search, check and remove is the best way to describe this path finding algorithm. Its goal is to identify and remove colliding nodes and edges from the roadmap until the shortest path from start to destination can be found. Only a collision checker for points is needed, because edges are discretized and checked with a certain resolution. This might not be the best way to go in environments with really small or thin objects though.

First the nodes are checked, because removing a node will result in all its edges being removed as well. The node checking algorithm uses bi-directional search, but does not name the actual term. The reasoning behind this choice is that the probability of finding the shortest feasible path is higher when a node is closer to the start or destination node. So we start at both ends and work to the center. When a collision is found, the node is removed and we will have to search for a new shortest path.

When all nodes along the path are in collision-free space, the edges will be checked in a similar way, and again working from the outside in. Collision checks are first performed with a coarse resolution, which will be refined when no collisions are found. When a collision is found, the edge will be removed and a new shortest path will be have to be found. If no collision is found along the path, the algorithm will terminate and return the collision free path.

If the search procedure does not yield a result, it is possible to use a node enhancement algorithm. Half of the new nodes will be uniformly distributed, and the other half will be randomly distributed within a certain radius of a seed. This ensures probabilistic completeness, which means the probability of finding a path will approach 1 as time goes to infinity. Seeds are cleverly selected by using the mid-points of deleted edges. To avoid cluttering after repeated refinement stages, we will only select random edges whose end-nodes were generated in the uniform part of the node enhancement process. New nodes are distributed around the seeds using a multivariate normal distribution.

Advantage:
- Seems to be the best method to use for fully dynamic environments.

Disadvantage:
- Edge collision detection by discretizing the edge might miss very thin objects, and a better result might be obtained by using line-mesh intersection checks.

Conclusion

Once again, the Lazy PRM approach [20] seems to be the best choice for path finding in fully dynamic environments. However, it might be a good idea to make a few changes. Using a line-mesh intersection test instead of using multiple point checks seems like a good idea. Also, it might be possible to have another type of node refinement by using the bridge test instead of the node enhancement algorithm described here, or a combination of the two. And since we are planning on using this algorithm in a dynamic environment, it is not a good idea to actually remove nodes and edges from the roadmap. A better way of dealing with collisions is to either work on a copy of the roadmap, or to put the colliding nodes and edges on a blacklist during execution of the A* algorithm and prevent them from being used.

2.5. Post-processing

In order to facilitate smooth camera motion, there has to be a post-processing phase in which the path is adapted for use by a camera [6].

First, the path has to be C^1-continuous, which basically means it has to be smooth and not have abrupt changes in direction. In the Voronoi diagram based algorithm described by Nieuwenhuisen, Overmars et al. [3], this is implemented in the road mapping stage. However, since we will use a roadmap based on Halton sampling [18] for static environments and a derivative of the Lazy PRM approach [20] for fully dynamic environments, we will need to perform smoothing in the post-processing phase.

Then, there is the matter of orientation. The nodes can only be used to obtain a position, the orientation during travel should be obtained from the path, while the orientation at the destination node should be determined by the camera positioning algorithm.

However, if the camera is oriented according to the curvature of the path, this will result in unnatural motion. The viewer should get cues about where the camera is going. A good way to solve this is to have the orientation vector point to the position that is about one second ahead of the current position [6]. Also, the horizon of the camera should be as straight as possible.

Finally, we need to take movement speed into consideration. Speed cannot be equal on all parts of the path, since speed has a greater impact when the camera is navigating through heavily curved parts. Therefore, the camera speed must be dependent on the curvature of the path. In order to solve this, a speed diagram has to be generated.

Out of these principles, path smoothing and the speed diagram need some more explaining, so we will discuss them more in depth now.

Path smoothing

There are a few different ways to make a path smoother. You can change the path into a Bezier curve or you can use interpolation instead of strict path movement, but neither of those methods is suitable for our application. Using a Bezier curve could result in a lot of new collisions which have to be detected, and it is slightly more processor intensive to calculate than some other solutions. Real-time position interpolation, based on a current position and one or two target positions, also needs additional collision detection. In fact, if the camera would collide when interpolation is used, the necessary collision response will be all but smooth. And we need the camera to move in a smooth path.

Fortunately, Nieuwenhuisen and Overmars [6] have a solution: circular blends. First, we must consider the fact that the shortest path is not always the fastest path for camera motion. Sometimes it is better to have a slightly longer path with less sharp corners, so the speed in the turns can be higher when we get to the speed diagram. This will result in a higher average speed, which makes it possible for a slightly longer path to still be faster than the original path.

The process of calculating a circle arc blend is shown in figure 2.5.1. We take a node *v* and two of the edges (*e* and *e*') attached to it. We have to find the bisecting line of *e* and *e*', and the midpoints of *e* and *e*' (which will be called p_m and p'_m). Next, we will generate a circle. The midpoint closest to *v* should touch the circle, and the center point of the circle should lie on the bisecting line. The resulting arc touches both edges and thus removes the first-order discontinuity at node *v*.

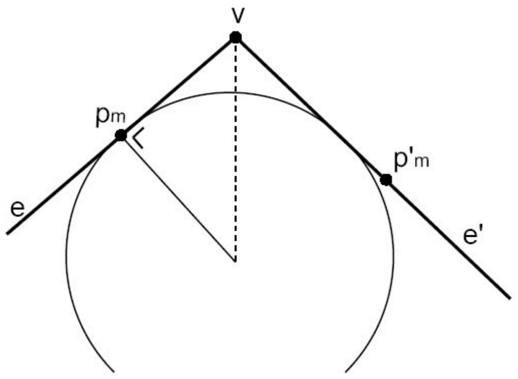


Fig 2.5.1: Creating a circle arc [6]

To find the center of the circle, we take a line perpendicular to the edge of which the midpoint was selected. The center of the circle is the intersection of that line with the bisecting line of *e* and *e'*. As you can see in the figure, the radius can be calculated by taking the distance between the center of the circle and the midpoint that is closest to *v*.
Like with all other smoothing methods, this might introduce new collisions. This can be solved by performing a binary search on the radius to find the largest possible radius (and thus, the fastest path), which is collision-free. The bottleneck in this process is performing collision checks on the arcs. The arcs will actually be approximated by using a number of short line segments that will be checked. This will finally result in a smooth roadmap.

Speed diagram

This stage of the post-processing phase is also described in the Motion Planning for Camera Movements in Virtual Environments paper [6].
The premise to the speed diagram is that the camera should move as fast as possible on a path. Different parts of a path have different top speeds, which depend on the curvature of the path. The highest speeds can be found at straight line segments, while speeds in arcs depend on the radius of the arc. A bigger radius means a higher speed is possible, because the angle of the turn is less sharp than it is for an arc with a smaller radius.
When all the speeds of the different segments are put into a diagram, we see a step function. This is not preferable, because abrupt changes in speed nullify the smoothness of the path. In order to smoothen speed changes, we use quadratic acceleration and deceleration to ensure smooth transitions in speed.

Conclusion

The post-processing phase is pretty clear, thanks to the work of Nieuwenhuisen and Overmars [6]. However, use of arcs, a speed diagram and the need to be able to 'look ahead' in time will require a very flexible structure for storing paths. This is why we will develop a separate structure for the final path, instead of making one general structure for both roadmap and final path. This separation of structures will save some memory and computation time, and it will also be less ambiguous (and thus easier to understand).

2.6. Camera position determination

First, let us explain why the explanation of camera position determination algorithms had to wait until after the path finding chapter. The whole path planning process roughly consists of two parts: pre-processing (which includes mainly road mapping) and real-time processing. The camera position will be determined in between those two phases, because it might be able to use information generated by the road mapping algorithm. This will be explained later on in this chapter.

While the task at hand might seem straightforward, not that much research has been done in this area. Camera placement is normally considered to be heavily involved with aesthetics and/or usability, which are both heavily reliant on user preference. This means that camera positioning is usually left as a task for the user to perform, which is not what we want here. There is one field in particular in which the user does not position the camera himself, and that is the field of interactive cinematography [10, 11, 12].

Most of the research for interactive cinematography deals with setting up a lot of camera constraints. Out of these constraints, the most important ones to deal with for this application are visibility and occlusion. Constraints can be used to simplify the search for an optimal camera position. Say, first, we exclude all positions that cannot have a full view (regardless of occlusion) of the object. Then, we exclude the positions that are too far away from the object and would result in the object taking up too little screen space to be useful. Now the visibility constraint is solved, and the resulting area will be all that needs to be tested by the occlusion constraint.

Still, the described research is based on brute force methods for finding the optimal camera position. This can be improved in several ways. These ways will be discussed in the next chapter.

Potential Visibility Regions

A way to optimize the process is to use Potential Visibility Regions (PVR) [13]. These are regions in which the polygons are shaded according to preference by a designer (manually). More preferred areas are shaded lighter than less preferred areas. These regions are used together with a visibility map for the potential occluding objects, to render the potential visibility geometry to a buffer in order of most desirable regions to least desirable regions. The brightest colours in the resulting buffer denote the most desirable position for the camera to move to. The problem with this method, however, is that these regions still have to be defined manually and there are no existing algorithms to accomplish this.

Advantage:
- Speeds up the process of finding a suitable camera position.

Disadvantage:
- Still needs manual shading of polygons, but our goal is to make everything as dynamic as possible.

Other optimizations

We have been unable to find any further optimizations to the brute-force algorithm in existing literature. Everyone still seems to use brute-force methods. The only optimization they use, is checking one constraint at a time, decreasing the feasible camera area for the next constraint that will be checked. It would be ideal if there were some way to easily calculate some of the constraints and thus removing the need for a lot of brute-force checking.

Conclusion

This seems to be a relatively unexplored area in the domain of virtual reality, while it is definitely important. Using brute-force search is going to put a heavy strain on the real-time part of our program, which could cause a noticeable lag between the time when the user selects an object, and the time when the camera starts moving towards it. We should try to prevent any noticeable lag, so it is necessary to try to come up with a major optimization. That is what we are going to do in the next chapter.

2.7. Overview

While the post-processing stage is pretty well determined in previous literature and camera placement still requires optimizations, road mapping and path finding algorithms come in many different variations. For this reason, we will now show a brief comparison between the different discussed road mapping and path finding methods and various techniques to possibly improve them.

Road mapping techniques

Technique	*Main advantage*	*Main disadvantage*
Manual roadmaps	Good roadmaps	Needs designers
Roadmaps based on synthetic vision	Most natural method which results in good roadmaps	Currently restricted to 2D movement
Pre-fabricated roadmaps	Eliminates pre-processing stage in familiar and unaltered environment	Useless in dynamic environments
Grid-based roadmaps	Easy to implement	Hard to determine grid density
Probabilistic roadmaps	More natural results	Not always uniformly distributed
PRM using a radius for the camera	Camera keeps a certain distance from objects	More processor-intensive due to advanced collision detection
Constrained PRM	Flexible, thanks to local planner	A lot of redundant constraints
PRM using Voronoi diagrams	Elaborate approach focused on pre-processing and resulting in natural looking roadmaps	Not directly suitable for use in 3D environments
Halton points	Combines consistency and uniformity of deterministic sampling methods with the natural look of random sampling	No clear speed advantage
Motion planning in semi-dynamic environments	Good solution when all motion in the environment is known beforehand	Relies on pre-planned motions and cannot deal with user motions
Adding cycles to existing roadmaps	Makes roadmaps more suitable for dynamic environments because it creates more path options	Roadmap calculation takes longer
Using a fully connected graph as a roadmap	Maximal connectivity	Maximal complexity
Customizing roadmaps at query time	Very flexible system with a fast pre-processing stage	Not directly usable for fully dynamic environments
Lazy PRM	Most useful method for fully dynamic environments	Path finding phase will take longer
Dealing with narrow passages	Using dilation is a good idea	More complex than the bridge test algorithm that was discussed in the section about PRM using Voronoi diagrams

Path finding techniques

Technique	*Main advantage*	*Main disadvantage*
Dijkstra	Always finds a path	Too complex
A*	Faster than Dijkstra and always finds the fastest path when used with Euclidian distance	In a worst-case scenario, it is still as complex as Dijkstra
IDA*	Less memory needed	More calculation needed and thus less useable in real-time
Bi-directional search	Speeds up search in complex environments	Less useful in 3D environments
Dynamic A*	Works well in fully dynamic and unknown environments	Might generate paths with a lot of unnatural redundancy
Lazy PRM	Most useful method for fully dynamic environments	The edge collision detection method used here is not always sufficient

Using these techniques

For most of the techniques mentioned here, it is possible to use one or more of the road mapping methods and one of the matching path finding algorithms to enable us to find a path between two points in the environment. This path can then be used by the post-processing algorithms described in [6] to create natural camera movement. In chapter 4, we will show which algorithms made it into our application, but that is by no means the only way to do it.

Camera placement needs more optimization and will be discussed in the next chapter.

3. Optimizing camera placement speed

This chapter describes some possible optimizations and considerations for camera placement, which will be tested in the application that comes with this thesis. While most of these optimizations were not mentioned in any of the literature on camera placement, it is very likely that they are used.

A small reiteration of what needs to be done:

- The visibility constraint must be solved.
    - The object should be fully visible on screen.
    - The object should not be too small to be useful.
- The occlusion constraint must be solved.
    - There should be no other objects in front of the desired object.
    - If this is not possible, a viewing position resulting in minimal occlusion should be used.

Determining the minimal distance from camera to object and camera direction (visibility constraint)

This can easily be done by using the bounding sphere of the selected object. A bounding sphere contains three parameters: x, y and z values for the position of the center, and a value for the radius. The center position can be calculated by finding the minimum and maximum x, y and z, and then calculating the values that are in the exact center of the minimum and maximum for the desired dimension.

One way to do this is mentioned in the OpenGL Technical FAQ [29]. Basically, they project the sphere onto the xy-plane and enclose it in a bounding box. This is a good way to start, since it results in a projection plane that can be used as the far plane for the camera frustum. However, they use the resulting bounding box for a glOrtho operation, which reshapes the object to fit the screen. This is not desirable for us. Also they position the camera on the z-axis on a distance equal to sum of the radius and the distance to the near plane of the camera frustum. This means they do not actually calculate the camera distance. So we still need to find another way of determining the viewing distance.

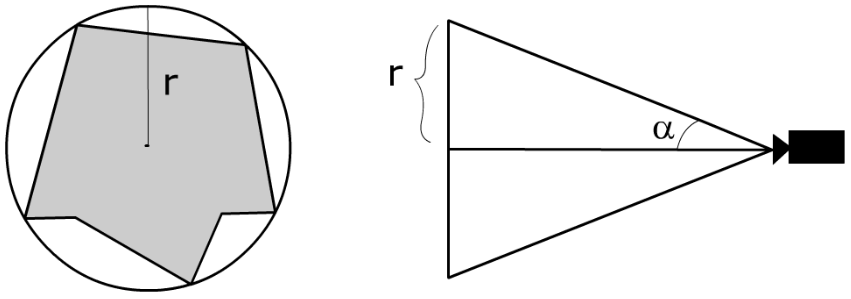


Fig. 3.1: Using the bounding sphere radius (r) and half of the camera field-of-view (α) with a tangent calculation will result in a good starting distance to use.

We can use the radius of the bounding sphere as the height in the tangent for an angle that is half the size of the camera's field-of-view. The distance between the camera and the center of the sphere can now be calculated by the following simple formula:
distance = *radius* / *tan*(*FOV* / *2*).

However, if we choose this exact distance, the camera frustum (a trapezoid shaped area which contains everything that is visible for the camera) intersects the bounding sphere. This means that there is a small chance that the frustum also intersects with the object itself, which in turn results in an incomplete view of the object. In order to prevent this, we increase the distance by a little bit. Adding half of the radius to the already calculated distance is a good way to do that. Remember, this viewing distance is an approximation. It is possible that a smaller viewing distance also results a full visibility, or that the minimal viewing distance is different for each camera angle. We will only use the bounding sphere approximation here.

Calculating a maximum camera distance is also something that could be considered. However, moving the camera away from the object actually decreases the visibility percentage. So not only would the object appear smaller to the viewer, but less of the object will be visible, as illustrated in figure 3.2. On the other hand, moving the camera even closer to the object will often result in the object not being completely visible, even though it might be a good way to avoid some obstructions. We can therefore conclude that the minimal camera distance is actually the only distance we should use, and that finding the right camera position and camera angle are more important tasks. Since the camera's orientation should be aimed at the center of the object in order to have the object appear centered on the screen, we only need the correct camera position, which can be found by using the methods described above. Now, the visibility constraint is solved.

Using bounding volumes to detect obstructions (occlusion constraint)

Since we have already calculated the bounding spheres, we can see if they prove useful for the occlusion constraint as well. We can test if a frustum intersects with a sphere, or even if a sphere is completely contained within the frustum. In the second case, we can be almost certain that the object is occluding our view, because the center of the object is in front of the far plane of our frustum. However, in the first case, further investigation is needed.

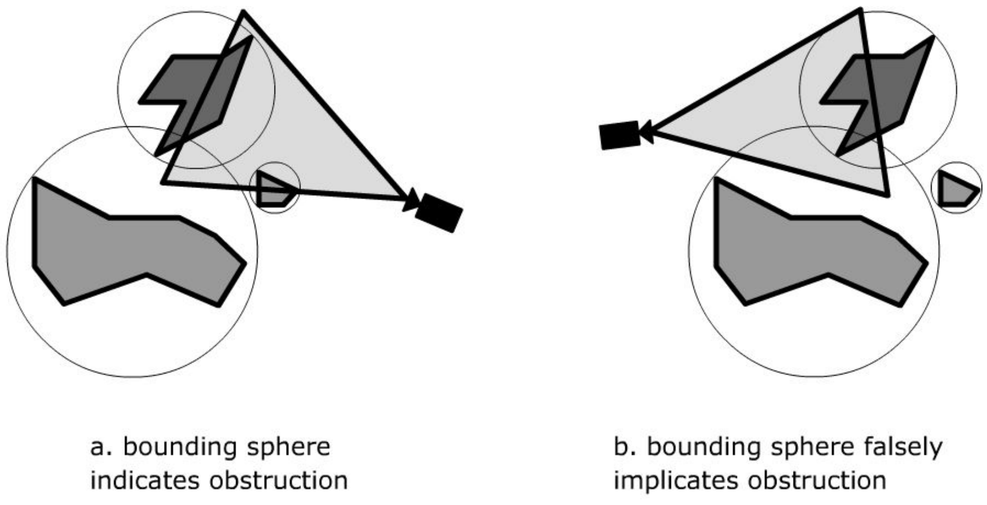

a. bounding sphere indicates obstruction
b. bounding sphere falsely implicates obstruction

Fig 3.3: Using bounding spheres alone is not always sufficient.

As figure 3.3b shows, the frustum intersects with a bounding sphere, but not the actual object. When an object's bounding sphere is inside the frustum, we can refine the search by using another bounding volume called the object-aligned bounding box.

An object-aligned bounding box rotates, moves and scales with the object. The main advantage of an object-aligned bounding box is when it is used on an object that is disproportionate, e.g. a long wall that is a lot longer than it is high or thick. These objects are very common in virtual environments (again, things like long walls or vehicles). Using object-aligned bounding boxes for preliminary collision detection increases accuracy for such objects. We can now check if such an object-aligned bounding box is inside of the frustum and use the result to determine whether to continue searching or not.

Determining the visibility percentage of an object

When it turns out that no camera points completely satisfy the occlusion constraint, it will become necessary to know the percentage of the object that is visible without occlusion. This percentage can later be used to determine the best camera position out of the ones that have been tested.

The most exact way to determine the percentage of visible vertices is to perform a collision check between a line from a vertex of the object to the camera position, and do this for every vertex in the object. When you divide the amount of vertices that collide with an object by the total amount of vertices, you get the visibility percentage. Since we already know which objects potentially collide, thanks to the bounding volume tests, we do not have to check for collisions with all of the objects in the scene. This might still be a costly process and it could be faster to use an estimated value. Also, it does not take long lines into account. In some cases, this results in an inaccurate percentage.

For instance, the occlusion of a few vertices make the algorithm decide the visibility is 90%. However, those few vertices are connected with the rest of the vertices by long lines and the actual visibility of geometry is more like 50%. The algorithm might decide to place the camera here, while the camera would be better off on a location with 80% vertex visibility and 60% geometry visibility. Still, we will not use more expensive geometry visibility tests because they would slow down the process.

Estimated values could speed up the process even more, at the cost of reduced accuracy. Some useful variables for calculating an estimate are the sizes of the bounding volumes and the percentages of the bounding volumes that are visible in the screen. An object-aligned bounding box only has eight vertices, so it will be pretty fast to perform the aforementioned camera position-to-vertex collision checks on the bounding volume instead of the actual object.

The search process

In most cases, the first tested camera position will not satisfy both the visibility and the occlusion constraints and thus not be a valid camera position. Now we will have to find other positions to start searching. Since the visibility constraint offers us a very fast way to determine new positions, we will start searching for camera positions that satisfy the visibility constraint. We do this by positioning the camera on another point of the virtual camera sphere you get when you would create a sphere with the minimum camera distance as its radius and the object center as its center point. In order to simplify and speed up this process, we only use the points on the xz-plane, and thus rotate the camera around the y-axis located at the focus object's center. Also note that we use the world y-axis instead of the object's y-axis. This also eliminates the need for having the camera look up or down, which might be annoying to some users.

So what we do, is rotate the camera around the y-axis centered in the object. The question is: how many steps do we use to complete the camera circle? In figure 3.3, we compare some possible step amounts. As you will be able to see, decreasing the step size in this scenario does not necessarily increase the relative chance of finding a good camera position. However, in scenarios with a lot of obstructions, the user will benefit from a smaller step size, since the absolute number of options to explore is simply higher. We will use a step size of 10 degrees, which results in 36 possible camera positions. That should be enough in most cases, judging from the frustum overlap as seen in figure 3.2.

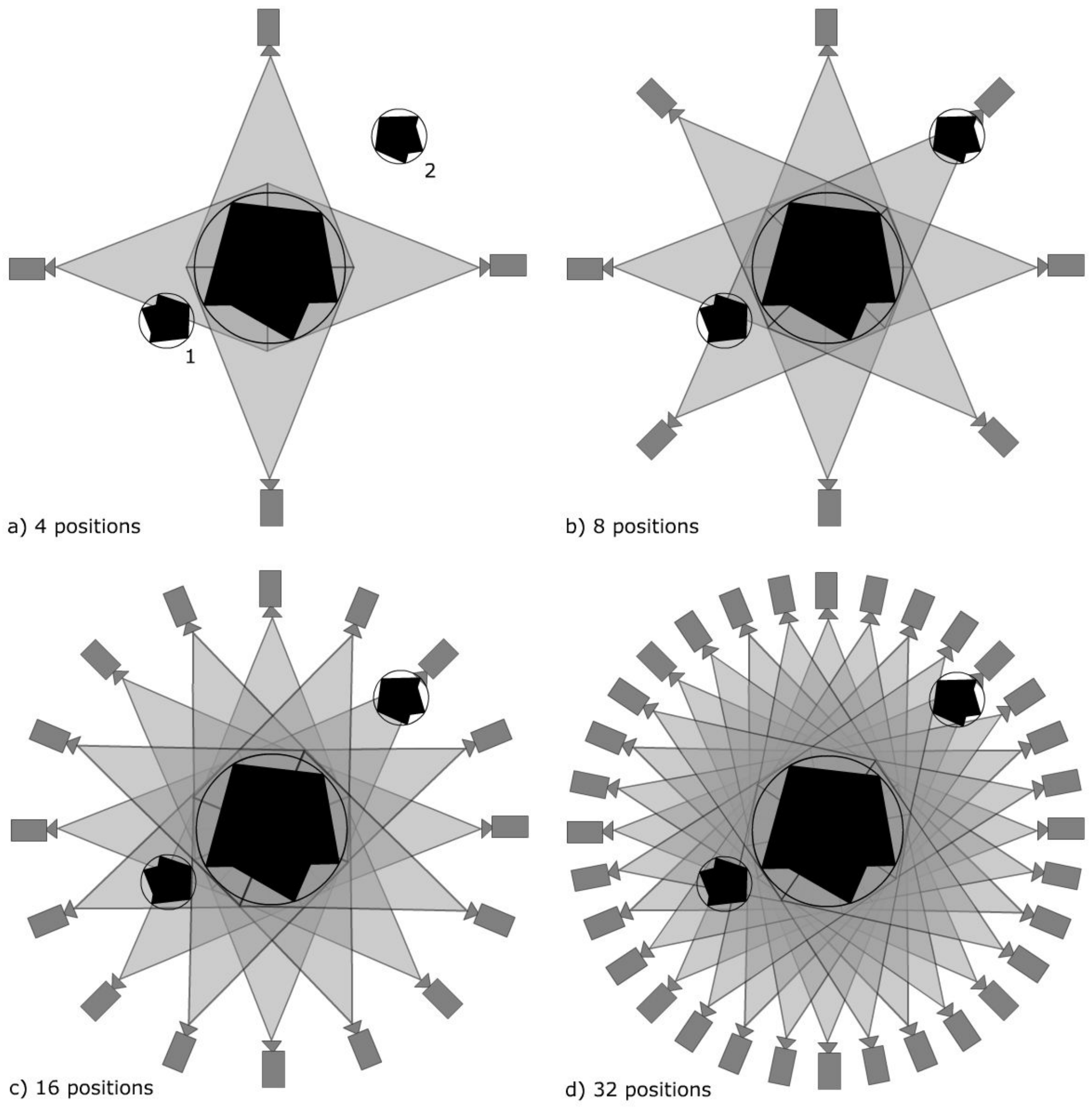


Fig. 3.4: Four different step sizes, resulting in different amounts of camera positions. Note how especially the last image seems to have a lot of redundant positions. Table 3.1 shows how many camera positions detect a certain obstruction. Since none of the positions detect both obstructions, it is also possible to determine the percentage of useful positions.

# of pos.	4	8	16	32
Obstr. 1	1 / 25%	2 / 25%	3 / 19 %	8 / 25 %
Obstr. 2	0 / 0%	1 / 13%	1 / 6 %	3 / 9 %
Usable pos.	3 / 75%	5 / 62%	12 / 75 %	21 / 66 %

Table 3.1: As you can see, the percentage of usable camera positions does not increase when the step size reduces. In fact, in some cases it even reduces that percentage. This table clearly shows that using a smaller step size does not always result in a higher percentage of success when all positions are searched. However, since we stop searching once we find a position that satisfies the occlusion constraint, it can still be useful to have a smaller step size. It also gives the algorithm more options to choose from when it has to select a position based on the visibility percentage of the object, which happens when the occlusion constraint cannot be completely solved.

As soon as a camera position completely satisfies the visibility and occlusion constraints, it is selected. However, when all positions have been searched and none of them fully satisfies the constraints, we will select the position with the highest visibility percentage.

Expanding the search

If this basic search process does not yield a satisfactory result, it is possible to expand the search instead of settling for a camera position with a lower visibility percentage. However, the following methods will not be implemented in our application because they are not required in order to show the effectiveness of selection-based navigation.

Until now, we did not fully utilize the visibility constraint. We used a circle with a radius based on the minimal distance from the camera to the object center, but we are allowed to use any position that lies inside of the d_{min} sphere. The d_{min} sphere is the sphere that uses the object center as its center, and the minimal camera distance as its radius.

An easy way to do this would be to divide the sphere into several circles on planes parallel to the xy-plane, and generating new circles moving from the inside out. We move from the inside out since a smaller angle is preferred when it comes to looking up or down, and the closer the y position is to the center, the smaller that angle will be.

Pre-calculating camera positions

Until now, we have persisted in letting the camera positions be calculated in real-time, right before the real-time part of the path finding algorithms. Since camera positioning is done somewhere in between the pre-processing and real-time phases, it could be added to either side. We assumed it would be added to the real-time side, but this also means the camera positioning could be added to the pre-processing phase. However, this is only the case for static environments. Dynamic environments change and thus pre-calculating camera positions for all possible objects is not useful in such a scenario.

In a static environment, let us see how we would put camera placement in the pre-processing phase. For each object, an optimal camera position could be calculated. This has advantages and disadvantages. The disadvantages are that, on average, more memory would be needed. Also, processing time for the pre-processing phase increases, while some (or most) of the generated positions will never be used. This holds especially true for very complex environments. The main advantage is the decreased amount of computation time needed for the real-time phase. If the camera positioning algorithm proves to consume a lot of time, it would be a good idea to add it to the pre-processing phase. We will only implement it in the real-time phase however, since it works for both static and dynamic environments that way.

Other considerations

In relatively wide environments, it is very likely that the first position that is tested is a valid position. For this reason, we have to determine the best and most aesthetically pleasing camera position to start the search with, because we could now actually have a choice in the matter and can state our preference. In the real world, when you walk towards an object and there are no obstructions in between you and the object, the final viewing direction vector is equal to the direction vector from the starting point to the object center. So this is also the best way to go for virtual camera placement. Note that this might not result in the most pleasing camera angles when there are obstructions in the environment, but when that is the case, we settle for any valid camera position.

Another subject to think about is user movement. Will selection-based navigation be the only mode the camera supports, or will the user also be able to move the camera manually? This heavily depends on the application. In a virtual tour (e.g. of a city or a museum) it is better to guide the user in the right direction and not confuse him. Looking around or rotating objects (e.g. statues) should be possible though, or the interactivity level would be too low. In a modelling environment, the user will want to be able to see the object from all sides. While selection-based navigation is useful for the initial camera movement towards the object, it is better to allow the user to move freely around the object than it is to have him rely on object rotation. When there is an object hierarchy or a lighting set-up, object rotation would not have the same result as a camera rotation around the object.

When it takes the application a long time to calculate the camera position and/or the final path, the user will have to wait a short while before the camera starts moving after an object is selected. This is called a lag. One way to deal with this is to have the camera start moving immediately towards the general direction of the goal, but this has its difficulties. It requires additional collision detection, and renders the path that is being calculated invalid, since the starting position has changed. However, for more complex applications, it is a good idea to look deeper into ways to reduce lag. Another way of doing this would be to use a heuristic, which uses the visibility percentage from a few of the calculated positions to determine which direction (clockwise or counter-clockwise) is more likely to generate a higher or even full visibility percentage, and calculate the visibility percentage for a new position in the according direction. Also, like with road mapping and path finding, using low-mesh proxy objects will result in an increased speed, since a full mesh is required to determine the visibility percentage.

Conclusion

We have found some useful optimizations and guidelines which will make camera placement faster than before. This will decrease the delay between when the user selects an object, and when the camera starts moving. Also we have determined the most natural camera position for wide open environments, which will result in a better experience for the user.

4. Implementation

In this chapter, we describe the resulting application, the techniques we used to get to this result, and some of the issues we encountered during development.

The SelNav library

Our implementation is in the form of a library of classes, which is used by a shell program with a graphical user interface. This method enables the library to be used for a whole range of existing applications. It is even possible to write a console interface for this library, but that would not really be useful in this case.

Because the selection-based navigation module (we will call it "SelNav" from now on) is independent of the main application, it is necessary to update all SelNav object positions, rotations and scales whenever they are changed for the corresponding object in the main application. Our implementation shows how this can be done.

This structure also enables the developer to use low-detail meshes for the SelNav module and high-detail meshes for the actual software, which would reduce calculations needed for mainly the road mapping and path finding stages. This is especially useful for smooth objects, because smooth objects often require a lot of faces to appear smooth to the user, while a low-detail internal mesh substitution would be almost as precise as the actual high-detail mesh.

For static environments, we use a road mapping method based on Halton point sampling [18] and a regular A* search path finding algorithm. However, we do not perform collision detection for nodes in the road mapping phase, only for edges. While this may result in isolated nodes, it also speeds up the process. No separate algorithms have been implemented for dynamic environments, but it is possible to recalculate the roadmap at any time. Using this library in a fully dynamic environment is still possible, but far from optimal.

We use the same camera placement methods and camera path smoothing methods as described in chapters 2.5, 2.6 and 3, with some slight simplifications. A speed diagram is not used, because the path is already C^1 continuous and that is the basic requirement. The speed still depends on the curvature of the path, but it is a step function instead of a smooth function. We reduce the range of possible camera positions to a 2D circle around the object center instead of the more advanced 3D sphere difference. We do not pre-calculate camera positions for all objects, nor do we perform any kind of compensation for lags. We do support user camera movement, but only for a second camera and not for the SelNav camera, and we also use the most natural camera starting position, based on the direction vector from starting point to object center.

Interface

By default, debug mode is enabled, which visualizes the entire process and shows possible error/status messages coming from the SelNav module. How to use the interface is described in appendix A, but the colours that are used for the messages and edges will be described here. When debug mode is disabled, the grid is still visible. The grid can be separately toggled on or off by using another console command. Using the grid is good for orientation purposes since there are no textures present and in some cases, no other objects are visible to give the viewer a landmark to focus on. However, using the grid might be confusing when the road mapping and path finding phases start and debugging mode is active, because they generate a lot of lines (some which will also be coloured in green).

Message colours:

- Grey:	Generic message from the main application.
- Green:	Positive message from the main application; a process was successful.
- Red:	Error message from the main application.
- Yellow:	SelNav related message, can either be positive, neutral or negative.

Most of the messages usually show the part of the application they belong to in brackets at the beginning of the message.

Line/node colours:

- White:	Node in the roadmap.
- Grey (normal):	Valid edge in the roadmap.
- Grey (thin):	Wireframe of a SelNav object, barely visible when combined with the actual 3D model in the main application.
- Red (thin):	Invalid edge. This is not present in the actual roadmap, but still shown for visualization (and originally debug) purposes.
- Green:	Edges that connect the start or destination node of the path to the original roadmap.
- Blue (thick):	The latest path that is being or has been traversed.

Using the SelNav library: structure of the SelNav module

For a more detailed UML class diagram of the SelNav module, see appendix B. In this paragraph, we will explain the basic structure. Figure 4.1 contains a simple diagram with the most important sections of the SelNav module and the required sections of a supporting main application. This shows the main structure and functionality of the SelNav module. We will now further explain how the structure fits inside of an application.

The SelNav module has its own internal representation of the objects in the virtual environment generated by the main application. This representation can be one-on-one, as in our implementation, but it is also possible to use a different representation. A good example would be to use a lower resolution mesh for the object representation in the SelNav module, which can be useful to reduce intersection calculations and speed up processing time. It is also possible to only selectively use objects in the SelNav module, and avoid storing an object in the SelNav module when the object is trivial (e.g. an eye or a coffee cup). The most important thing to keep in mind is to use the correct ID numbering. SelNav objects start at an ID number of 1 instead of the usual 0. This was needed to allow object ID 0 to be used for empty or temporary objects. Adding a new object to the SelNav module generates a new ID number, which could be stored in a hash table which links the object ID from the main application with the ID from the SelNav module, so the correct ID can be easily looked up when needed.
Once the objects are stored in the SelNav module, their position, rotation and scale should also be changed whenever a change occurs for the object in the main application. A more detailed explanation will follow in the next paragraph. Objects in the SelNav module are closely linked to objects in the main application, but they also enable all the extra functionality we need.
Generating a roadmap is very easy, once the correct settings have been sent to the SelNav module. One call to the road mapping method is enough to have the SelNav module do the

rest. It is slightly more complicated for path planning, since the path is also used for the camera. Calculating a new path is easy, again only one call is needed, but this time it needs a parameter: the object ID of the target object as it is stored in the SelNav module. This can be obtained from a hash table maintained by the main application.

Traversing the path is as easy as having the camera continuously request its position and orientation from the SelNav module. The SelNav module will use the information from the latest stored path to determine the correct position. A path contains a function that returns the position it has on the time of request, while internally keeping a timer that started when the path was generated. When the timer ends, no new positions will be generated and this will result in the camera receiving the same position and orientation until a new path is generated.

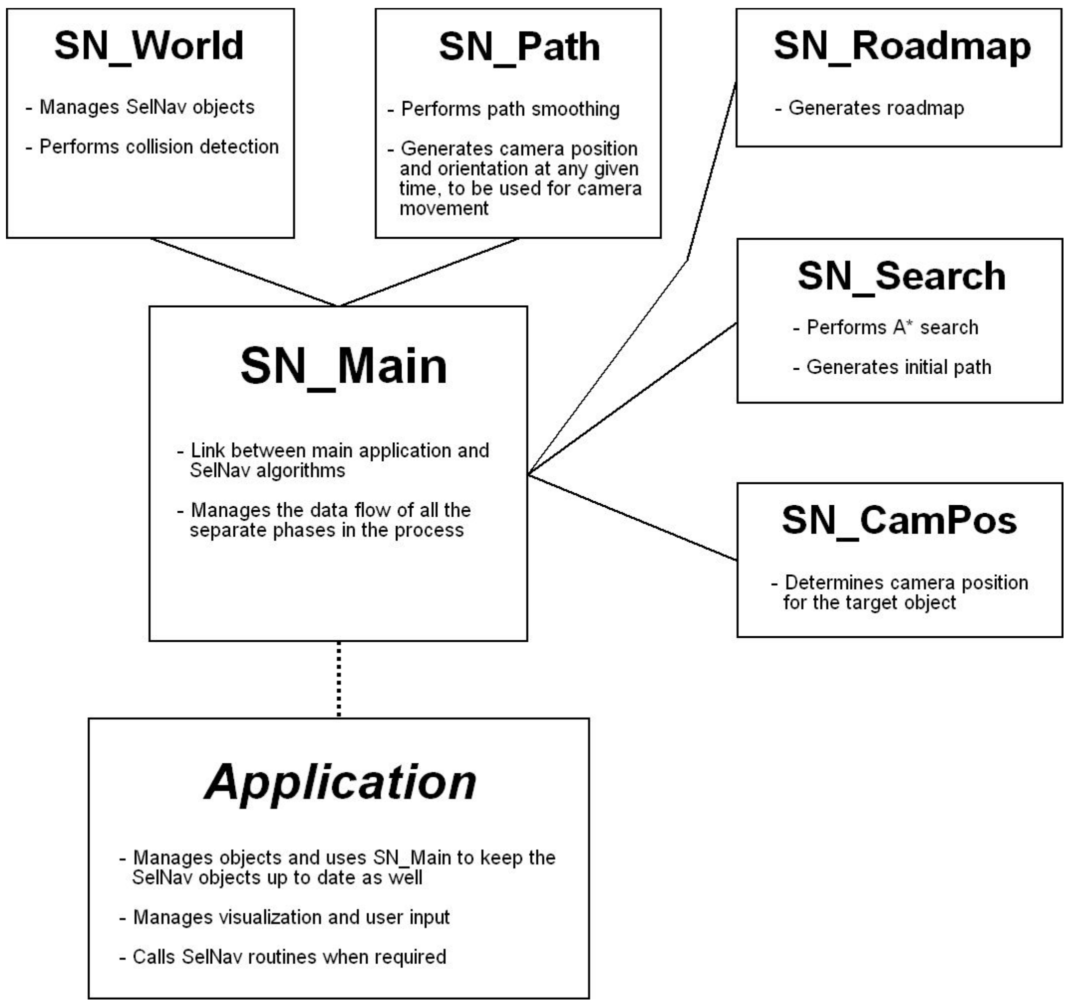

SN_World
- Manages SelNav objects
- Performs collision detection
SN_Path
- Performs path smoothing
- Generates camera position and orientation at any given time, to be used for camera movement
SN_Roadmap
- Generates roadmap
SN_Search
- Performs A* search
- Generates initial path
SN_Main
- Link between main application and SelNav algorithms
- Manages the data flow of all the separate phases in the process
SN_CamPos
- Determines camera position for the target object
Application
- Manages objects and uses SN_Main to keep the SelNav objects up to date as well
- Manages visualization and user input
- Calls SelNav routines when required

Fig. 4.1: Overview of the structure of the SelNav module and how it communicates with the main application. Only the most important (sometimes abstract) classes have been included, which means that storage classes for nodes, graphs, objects, etc. are not present in this diagram.

Using the SelNav library: input and output

There are some moments or situations when a SelNav procedure needs to be called. We will discuss all of them here, to explain the exchange of information between the main application and the SelNav library.

- Initialization:	To start a session, create an `SN_Main` object by calling it with an empty constructor.
- Adding objects:	This is probably the trickiest part. First, keep in mind that the object ID for a SelNav object is always 1 or higher. This might require a conversion if you start numbering your objects with 0 in your main application. Next, create a new object using the `createObject()` method, and store the `int` value it returns for further usage. When this is done, the mesh will have to be created. First, we need an array of `double` containing the vertices. Since every vertex has an x, an y and a z-coordinate, the size of this array will be three times the number of vertices. We use `getWorld->setMesh()` with the object ID we have received earlier, the array we just generated and the number of vertices. In order to connect the vertices, we have to generate an array of `int` containing the face triangle configurations. The size of this array is three times the size of the number of triangles, and every triangle consists of three `int` values containing the IDs of the three vertices (starting at 0) that connect to form that triangle. The vertex IDs must be in counter-clockwise order looking from the outside of the object in, or the collision detection algorithms will not function correctly! Next, send these values to the SelNav module by using `getWorld->setFaces()` with the object ID, the array of triangles and the total amount of triangles. When this is done, we need to have the object calculate somevalues, like the bounding sphere, and we call the `getWorld-> update()` function with the object ID in order to do this. Finally, we need to make sure the object is in the correct position, with the correct rotation and scale. This is done by sending the modelview matrix for the original object to the SelNav object. Information on how to do this is in the next section. Note that you do not have to store the exact same mesh in the SelNav module as is used by the internal representation. Sending a lower resolution proxy mesh, which still uses the same modelview matrix, will result in faster calculations.
- Updating object transformations:	Whenever an object in the main application is translated, rotated or scaled, the change (the new modelview matrix) has to be sent to the SelNav representation of the object as well. Again, note that an object ID conversion might have to be performed to make sure the SelNav object ID is correct. First, you will need the new modelview matrix. When using OpenGL, an easy way to get the modelview matrix for an object is to perform all transformations needed, and then save the modelview matrix into an array of `double` by using the OpenGL function `glGetDoublev(GL_MODELVIEW_MATRIX, modelview)` when you have already created an empty array called `modelview`. However, this does not result in the correct position. In order to

solve this, we replace the value at index 3 by the x position, the value at index 7 by the y position, and the value at index 11 by the z position. Next, when you have the modelview matrix, send it to the corresponding object in the SelNav module by using `getWorld->getObject(`*object ID*`)->setModelView()` with the new modelview matrix, stored in an array of `double` using OpenGL style ordering.

- Requesting the camera coordinates: Whether the camera is standing still or actively traversing a path, its position and orientation can be requested from the SelNav module by using the `getPath->getPos()` and `getPath->getOri()` methods. As you can see, those positions are actually stored in an `SN_Path` object contained within `SN_Main`. The first one returns the camera position, and the second one returns the target position (or lookat-vector). The up-vector should always be (0, 1, 0). When these methods are used in every rendered frame, they can be used to drive the camera (or an object representing the camera), thus handing full camera control over to the SelNav module.
- Generating a new roadmap: When all settings are correct and all objects are stored in the SelNav module with the correct modelview matrix, simply call `generateRoadmap()` to have the SelNav module generate a roadmap internally, which can later be used to find a path.
- Finding a path to an object: Calling `findPath()` with the object ID the target object has in the SelNav module is all you need to do. The SelNav module automatically finds a suitable camera position and orientation and generates a smoothened path from the current camera position and orientation to the new position and orientation. When the current camera position and orientation are continuously requested as described before, the camera will automatically start moving as soon as the path has been calculated.
- Changing settings: To change a setting, simply call the `changeSetting()` routine with an `int` containing the setting type, as specified in `SN_Main.h`, and a `double` representing the new value.
- Drawing the path, roadmap and mesh: By calling `drawGL()`, all roadmap nodes and edges, SelNav object wireframes and the final path will be visualized, if present.
- Requesting SelNav status messages: First, check if there are any status messages generated by SelNav using the `hasError()` method. If an error exists, enter a loop which repeatedly requests the first message in SelNav's message queue by calling `getFirstError()`. Continue this until the `hasError()` method returns false. You have now collected all status messages generated by SelNav up until the point when the search ended.

Issues and problems

Each potential edge is first checked for intersections with the bounding spheres for all objects in the environment. This is done by first using a line-sphere intersection test [32] for culling objects as much as possible. When a line segment does not collide with a sphere, there is still the possibility that the line is completely contained within the sphere, so this must be tested as well. If an intersection is detected or the line is completely inside the sphere, we use a ray-triangle collision detection algorithm [30] for the edge and each of the triangles of the potentially colliding object to determine if there is actually a collision. However, since ray is a line of infinite length, this might cause incorrect results, as shown in figure 4.2. In order to adapt this algorithm for use with line segments, which edges really are, we perform a second check when a collision is detected in one of the triangles of the mesh. For both end points of the line segment, we calculate the side of the triangle they are on. When they are both on the same side of the triangle, the line segment does not intersect and the search will continue. This can also be done by using segment-plane intersection checks [31], but that only changes the problem, because you now have an infinite plane instead of a finite triangle.

The line-sphere intersection method we use for culling severely speeds up the road mapping process, even in simple environments.

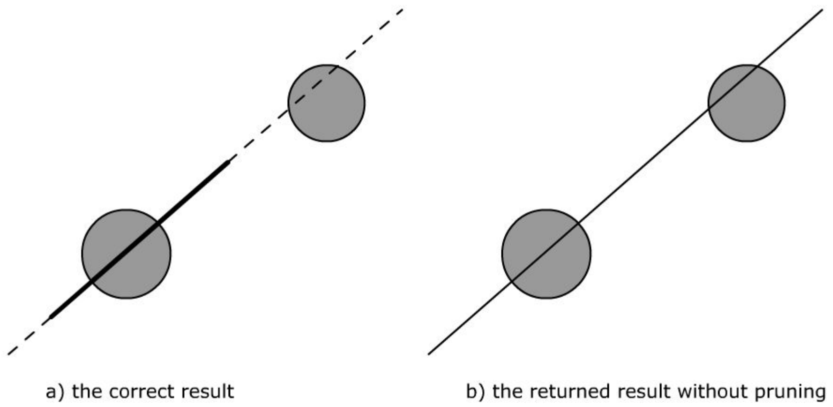


Fig 4.2: When checking for a collision between an edge and an object mesh, make sure to check only the line segment and not the entire line. While this may seem obvious, it is very important to do this correctly. Checking for line intersections will result in a roadmap missing some edges that are actually valid edges.

For node collision detection, we use a basic algorithm that checks if a point is on the inside of every triangle in a mesh [32]. However, this does not provide correct results when used with a concave object. Since we do not actually use node collision detection in our road mapping and path finding stages but only for the camera frustum, which is always convex, this is not a problem. Still, for completeness sake, it is a good idea to mention how node collision detection can be performed for concave objects as well, because this is needed for some other road mapping methods. A simple way to do this is to perform an intersection check for a line segment with the mesh. The line segment is the segment between the actual node, and a point that is outside of the object's bounding sphere. Now, we check how many triangles intersect with this line segment. If this is an even number, we will have entered and left the object, which means the node is on the same side as the outside point: outside of the object. If this is an odd number, it means the node is located inside of the object. All this requires is a collision detection method that does not immediately stop when an intersection is encountered, but instead counts all of the intersections.

Our implementation of the path finding algorithm had a problem when the path consisted of exactly 1 segment; it reversed start and destination positions. Once this problem was discovered, it was easily solved by testing whether the destination node's predecessor is the start node or another node, and having the path generated accordingly. Also, there was a problem with rotation when a new path was generated while the previous path was still at the earliest stage of being traversed, which is a stage that only involves rotation. This was solved by having the program detect if the new camera position is equal to the current camera position, and avoiding the path planning algorithm if that is the case. Instead, both the current and new camera position and orientation are added to a new path. Also, the path traversing algorithm had to be changed to deal with paths containing only rotation and no movement.

Another path finding problem will be the increase in nodes after a lot of paths have been generated. Since the starting and ending position are not necessarily represented by a node in the roadmap, we have chosen to add those nodes to the roadmap and using them for the search algorithm. This means that every path finding call will add two nodes to the roadmap on the start and destination positions, regardless of whether there is already a node present at that position. The cluttering that will occur because of this can be solved by working on a copy of the original roadmap during the path finding phase. The nodes will be added to the copy, but not the original, which will result in more consistent running times, instead of a system that is slowly but steadily getting more cluttered with (mostly redundant) nodes and thus slower. Another possibility is removing those two newly added nodes immediately after use. This is what is currently implemented. It is also possible to not store these nodes in the roadmap at all, but instead find the closest existing nodes and use them for the calculation. Afterwards, the start and destination node can be added to the path to complete it.

For the camera placement algorithm, one very important thing to remember was that C++ works with radials instead of degrees by default, so angles have to be converted all the time. Also there was a strange issue where the calculation of the initial (and preferred final) camera angle was off by exactly 180 degrees, when the z-difference was negative. This was easily solved, once the cause of the problem was discovered, by adding 180 degrees to the angle when the z-difference was negative.

How to correctly generate the camera movement from an external library was another issue. We have chosen to let the main application request a new position and orientation for the camera from the object that contains the final path. This object stores the starting time and path segments, and uses them in conjunction with the current time to calculate the new camera position and orientation. A path segment is either a straight line or a circle arc, and contains its own method for calculating the position on the segment at a certain time. Special care has been taken to ensure that the orientation will be correctly interpolated before movement starts and after movement ends, since the final camera orientation of the path does not necessarily equal the final camera orientation calculated by the camera placement algorithm. This was the hardest part, because a lot of weird errors occurred where the camera would look at a point at an infinite distance or the camera would not rotate smoothly, but just seem to switch orientations. This happened when the angle was around 180 degrees, since the rotating operation used linear interpolation (represented by a path segment) on the orientation positions instead of the actual camera angles. The solution was to use an arc segment to represent the difference in orientation positions instead of a line segment which I originally used.

Circle arcs were probably the trickiest feature of all to implement, together with the collision detection algorithms. The best way to store and draw circle arcs seems to be to represent them by an arbitrary axis in 3D, a starting point in 3D and a maximum rotation angle which can be either positive or negative, to determine the direction of the rotation. Storing the center point is also required to position the axis and for calculations involving the radius. Getting the length of a circle arc is easy once you know the maximum angle and the circle radius. To get the percentage of the circle that is used to make the arc, divide the absolute of the maximum angle by 360. The circumference of a circle is 2π * *radius*. Multiply this by the percentage of the circle that is actually used, and the length can be calculated by using formula 4.1.

$$\frac{|mAngle|}{360} * (2\pi * r)$$

Formula 4.1: Calculating the length of a circle arc

However, since this method uses the maximum angle, we must first calculate that angle. In order to do this, we need three points: the center point, the starting point and the ending point. We already have either the starting point or the ending point, since that was used to create the circle arc, as seen in chapter 2.6. The center can be calculated by finding the intersection of the bisecting line with the point we already have, and the second point can be either found by finding the position where its edge intersects with the sphere used to create the circle arc, or by finding the point on the second line that has the same distance from the node as the midpoint of the first line. Now we have to make sure that the starting point is the point on the first edge that will be traversed, and the ending point is the point on the second edge. This means it might be necessary to swap the two points, which is not a big deal in itself, but failure to do so will result in skipping and backward movement.

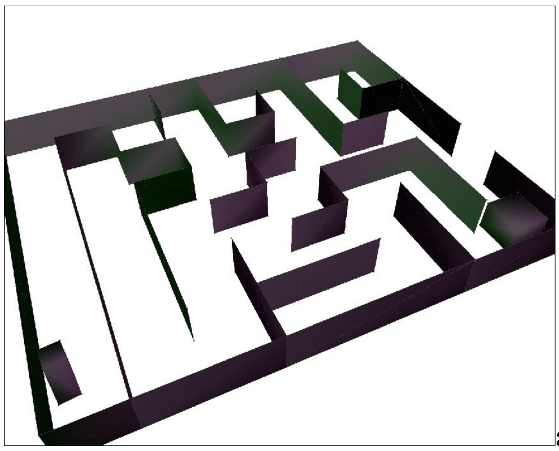
We use the three points we have to create two unit vectors. First, we create a vector from the center to the starting point, then another vector from the center to the ending point. Next, we normalize both vectors. We now calculate the dot product of these two unit vectors, which results in the cosine of the angle, since $a_u \cdot b_u = \cos\theta$ [33]. Taking the arc cosine of this result, will give us the angle we were looking for. However, this angle is always positive, because it does not take the traversing order of the points we used into account. We need to determine if the angle needs to be positive or negative. To do this, we need to rotate the starting point around an axis by the angle we just generated. That axis can be found by taking the normal from the plane created by the two vectors we previously calculated. If the rotated starting point does not result in the ending point, the angle should be negative instead of positive.

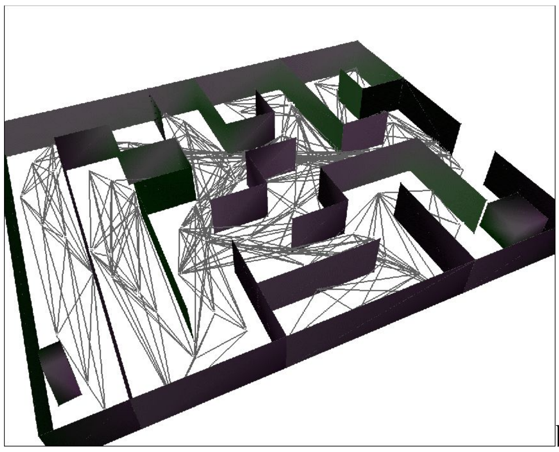
In order to optimize drawing speed for circle arcs, we will calculate a trajectory using discrete sampling and store it in an array. This array will then be used for rendering, which is more efficient than recalculating those samples for every frame.

Perhaps one of the most important problems that you can get confronted with is memory leakage. A small memory leak in something as seemingly insignificant as a get-method will result in a major slowdown or a lot of crashes when the program is being used for a while. Since there can be a lot of objects with a lot of vertices that have to be recalculated from time to time, it is important to make sure any previous representations are always deleted before a new one is generated. It is a good idea to keep this in mind from the beginning, because it can be a lot more difficult to solve a memory leak when development is in a later stage.

Results

The screenshots in figure 4.3 show the application in action. Since our testing application used the numbers on the keyboard to select objects, the first 10 objects were shaped like their corresponding object ID number to prevent any possible confusion.


a.


b.

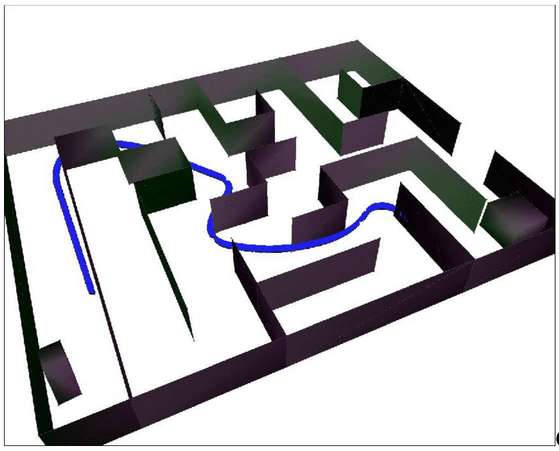

c.

Fig. 4.3: (a) shows an environment, (b) shows the same environment after generating a simple roadmap, and finally (c) shows the same environment with a generated path. The grid has been hidden in all screenshots, the colour scheme was changed from black to white to make the screenshots more suitable for print, and the roadmap has been hidden in the final screen, so the focus is on the path. The small dents in the resulting path are an OpenGL visualization issue, camera travel along the path does not have any discontinuities like that.

As you can see, a correct and smooth path is generated by using a simple roadmap based on Halton points.

Another very important aspect of this application is time. We want to have a path finding phase that is in real-time, which means we want to have the path finding phase take less than 1000 ms. Two seconds already seems like a long time when working in virtual environments, so we will say one second (1000 ms) is the absolute maximum for the path planning phase.

The road mapping phase may take longer, but of course we still prefer a faster road mapping algorithm over a slower one. For these tests, we use a testing environment consisting of 8 cubes and a few more complex objects, some of which badly accessible due to a cube blocking access to it. The objects in the environment consist of a total of 1422 triangles. A list of running times for the path planning and road mapping phases are given in table 4.2 and 4.3.

Settings	Avg. PRM time	Avg. Halton time
500 nodes, node distance 3.5	809 ms	753 ms
300 nodes, node distance 6.0	512 ms	485 ms
100 nodes, node distance 10.0	129 ms	119 ms

Table 4.2: Average running times for the two different road mapping algorithms implemented in the SelNav module. Tested environment has 1422 triangles. Testing system used: AMD Athlon XP 1900+ with 1.2GB RAM

Settings	Path finding phase running time
500 nodes, node distance 3.5	719 ms
300 nodes, node distance 6.0	313 ms
100 nodes, node distance 10.0	117 ms

Table 4.3: Average running times for the path planning phase (camera placement + path finding + path smoothing) in the same testing environment, using the roadmap based on Halton points. The path is always tested with the same starting position and the same ending object, which is behind a few obstructions. We can see that a lower node count will lead to a faster path planning phase. This is mainly because the path smoothing process is needed less often when there are fewer nodes in the resulting path. The camera placement algorithm, which determines the destination position and orientation, is independent of the amount of nodes, and thus relatively consistent in length.

We can see that Halton point sampling seems to be slightly faster than using a completely random sampling method, which is a pleasant surprise. The difference gets more noticeable when more nodes are used instead of wider connection ranges (= maximum distance between two nodes when an edge is allowed). This is good, seeing as how complex environments usually rely on more nodes and keep the same connection range.

Also, the path finding is in real time. There is a slight noticeable delay between pressing the key to start the process and seeing the camera move, but it is short enough to not be considered annoying. The reason path finding times are so close to road mapping running times, is that this is a relatively simple environment due to the use of the bounding sphere culling algorithms to reduce time. The path finding phase also includes path smoothing and camera positioning. These algorithms also take a certain amount of time, which will become relatively smaller as the environment grows more complex. The reason for showing these results is to make clear that designing an environment with culling methods in mind will greatly speed up the calculation process.

A second set of test results has been generated in an environment that consists of one large object (a house) with 1504 faces, and several smaller objects. In this case however, all of the other objects are located inside the first object's bounding sphere, effectively rendering the culling algorithms useless. Table 4.4 shows the performance of SelNav in that environment.

As you can see, the difference between the road mapping and path finding phases has increased significantly, and the path finding phase is still fast enough to work in real-time.

Settings	Road mapping phase	Path finding phase
500 nodes, node distance 3.5	2703 ms	672 ms
300 nodes, node distance 6.0	3641 ms	465 ms
100 nodes, node distance 10.0	953 ms	359 ms

Table 4.4: Average running times for the road mapping (Halton point based only) and path finding stages in a more complex environment. Surprisingly, the configuration with the most nodes is not the slowest configuration in the road mapping phase this time.

Conclusion

The final product contains some of the techniques described in this paper and uses them to fully support selection-based navigation. The road mapping, camera placement, path finding, path smoothing and path traversing stages have all been implemented and are functioning correctly. However, it still sometimes suffers from a slight noticeable delay between selection of an object and the start of the resulting camera movement. Still, the delay is not long enough to be considered annoying by most users.

5. Conclusion

We have discussed a lot of techniques for road mapping and path finding that could be used for selection-based navigation. To prove this, we selected a few of those techniques and built a working application around them that supports selection-based navigation using roadmaps. Our final application is based on Halton point sampling [18] and path finding with A* search [24]. We also have smooth, C^1 continuous camera movement [6] to avoid making the user feel uncomfortable. Our camera placement algorithms result in a good view of the selected object, although further optimizations are still possible.

For most of the phases needed in the process of creating this mode of navigation, there are multiple possible methods that can be used. Most of the methods have not been implemented by us, but some of them might yield better results in the end. It is always useful to have some alternatives when needed, so taking a closer look at one of the less discussed methods can be a good idea.

We feel that the resulting application, while not perfect, still demonstrates the possibilities of selection-based navigation. It is possible for the user to select an object and watch as the camera automatically moves to a good viewing position.

However, we still have not been able to test the Lazy PRM [20] method in a fully dynamic environment (or any other method , for that matter). Since we already have a slight noticeable delay between selecting an object and the camera movement starting, having a more complex path finding stage will result in a longer delay. This means the algorithms will be required to compensate for this delay, because an annoyingly long delay will have the user want to take control himself, whether the application allows for it or not. In either case, the user will not prefer selection-based navigation and avoid using it as much as possible, which is something we would not like to see. One of these optimizations is already possible with the SelNav module: using a low-resolution proxy mesh instead of actual objects. While it is not implemented in the main application, we recommend this optimization to be used.

We feel that in most applications, using selection-based navigation alone is not enough. It removes an important piece of interactivity from the user. Selection-based navigation certainly has its merits though, so it is up to developers to find a good balance with other, more conventional modes of navigation.

Bibliography

1. **Collision Avoidance and Map Construction Using Synthetic Vision** (2000)
P. Monsieurs, K. Coninx, E. Flerackers
Expertise Center for Digital Media, Limburg University Center, Diepenbeek, Belgium

2. **Interactive Navigation in Complex Environments Using Path Planning** (2004)
B. Salomon, M. Garber, M. C. Lin and D. Manocha
Department of Computer Science, University of North Carolina, Chapel Hill, NC, United States

3. **Automatic Construction of Roadmaps for Path Planning in Games** (2004)
A. Kamphuis, M. Mooijekind, D. Nieuwenhuisen and M.H. Overmars
Institute of Information and Computing Sciences, Utrecht University, The Netherlands

4. **Useful Cycles in Probabilistic Roadmap Graphs** (2004)
D. Nieuwenhuisen and M. H. Overmars
Institute of Information and Computing Sciences, Utrecht University, The Netherlands

5. **Roadmap-based Motion Planning in Dynamic Environments** (2004)
J. P. van den Berg and M. H. Overmars
Institute of Information and Computing Sciences, Utrecht University, The Netherlands

6. **Motion Planning for Camera Movements in Virtual Environments** (2003)
D. Nieuwenhuisen and M. H. Overmars
Institute of Information and Computing Sciences, Utrecht University, The Netherlands

7. **Voronoi Diagrams**
http://www.cip.informatik.uni-muenchen.de/~viermetz/cg/Voronoi_Diagram.html

8. **Voronoi/Delaunay applet**
http://www.cs.cornell.edu/Info/People/chew/Delaunay.html

9. **Voronoi tessellations generated by 3D point processes**
http://fyzika.ft.utb.cz/voronoi/

10. **A Model for Constraint-Based Camera Planning** (2000)
W. H. Bares, S. Thainimit and S. McDermott
Center for Advanced Computer Studies, University of Louisiana, Lafayette, LA, United States

11. **Intelligent Multi-Shot Visualization Interfaces for Dynamic 3D Worlds** (1999)
W. H. Bares and J.C. Lester
Department of Computer Science, North Carolina State University, Raleigh, NC, United States

12. **Realtime Constraint-Based Cinematography for Complex Interactive 3D Worlds** (1998)
W. H. Bares, J. P. Grégoire, and J. C. Lester
Department of Computer Science, North Carolina State University, Raleigh, NC, United States

13. **A Camera Engine for Computer Games: Managing the Trade-Off Between Constraint Satisfaction and Frame Coherence** (2001)
N. Halper, R. Helbing and T. Strothotte
Department for Simulation and Graphics, University of Otto-von-Guericke, Magdeburg, Germany

14. **An Octree Solution to Conservation-laws over Arbitrary Regions (OSCAR) with applications to Aircraft Aerodynamics** (1997)
E. F. Charlton
Aerospace Engineering and Scientific Computing
http://hpcc.engin.umich.edu/CFD/users/charlton/Thesis/html/node29.html

15. **Octree Partitioning Techniques** (1997)
M. Kelleghan
http://www.gamasutra.com/features/19970801/octree.htm

16. **A Comparative Study of Probabilistic Roadmap Planners** (2004)
R. Geraerts and M. H. Overmars
Institute of Information and Computing Sciences, Utrecht University, The Netherlands

17. **The Bridge Test for Sampling Narrow Passages with Probabilistic Roadmap Planners** (2003)
D. Hsu, T. Jiang, J. Reif and Z. Sun
Department of Computer Science, National University of Singapore

18. **Sampling with Hammersley and Halton Points** (1997)
T-T. Wong, W-L. Luk and P-A. Heng
Journal of Graphics Tools, vol. 2, no. 2, 1997
http://www.cse.cuhk.edu.hk/~ttwong/papers/udpoint/udpoints.html

19. **Customizing PRM Roadmaps at Query Time** (2000)
G. Song, S. Miller and N. M. Amato
Department of Computer Science, Texas A&M University, TX, United States

20. **Path planning using lazy PRM** (2000)
R. Bohlin and L.E. Kavraki
In proceedings of ICRA, 2000

21. **A two level fuzzy PRM for manipulation planning** (2000)
C. L. Nielsen and L. E. Kavraki
Department of Computer Science, Rice University, Houston, TX, United States

22. **Motion Planning for Industrial Robots** (1999)
R. Bohlin
Department of Mathematics, Chalmers University of Technology and Göteborg University, Göteborg, Sweden

23. **On Finding Narrow Passages with Probabilistic Roadmap Planners** (1998)
D. Hsu, L. E. Kavraki, JC. Latombe, R. Motwani and S. Sorkin
In proceedings of WAFR, 1998

24. **Game AI Programming** (2004)
P. Monsieurs
Part of the "Architecture and Algorithms for Computer Games" coarse, Hasselt University, Diepenbeek, Belgium

25. **A* Pathfinding for Beginners** (2004)
P. Lester
http://www.policyalmanac.org/games/aStarTutorial.htm

26. **Variations of A*** (2003)
A. Patel
http://theory.stanford.edu/~amitp/GameProgramming/Variations.html

27. **Optimal and Efficient Path Planning for Partially-Known Environments** (1994)
A. Stentz
In proceedings of ICRA, 1994

28. **Real-Time Replanning in Dynamic and Unknown Environments**
A. Stentz
http://www.frc.ri.cmu.edu/~axs/dynamic_plan.html

29. **OpenGL Technical FAQ – Viewing and using Camera Transformations**
http://www.opengl.org/resources/faq/technical/viewing.htm

30. **Intersections of Rays, Segments, Planes and Triangles in 3D**
D. Sunday
http://softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm

30. **Intersections of Lines, Segments and Planes (2D and 3D)**
D. Sunday
http://geometryalgorithms.com/Archive/algorithm_0104/algorithm_0104B.htm

31. **Object Collision Detection**
http://wiki.beyondunreal.com/wiki/OCD

32. **Intersection of a Line and a Sphere (or Circle)** (1992)
P. Bourke
http://astronomy.swin.edu.au/~pbourke/geometry/sphereline/

33. **Formula for the angle between two vectors**
http://chortle.ccsu.ctstateu.edu/VectorLessons/vch10/vch10_5.html

Appendix A: Application manual

Basic commands

Key:	*Function:*
W / A / S / D	Move camera forward, backward, left or right when in free camera mode. (Note: default camera mode = selection-based)
Q / Z	Move camera up or down when in free camera mode.
Arrow keys	Turn camera left, right, up or down when in free camera mode.
Spacebar	Hold spacebar to move and turn faster.
R	Generate roadmap.
Number keys 1-0	Select object 1-10. This also performs the actual selection-based navigation when in selection-based camera mode. (Note: a roadmap has to be generated at least once before this works.)
C	Open console (Console commands are listed below.)
F1	Shows help screen which lists a few functions to change the object manipulation commands. However, it is recommended that they are not changed by the user.
F2	Toggle fullscreen.
Esc	Quit application.

Object manipulation commands

Key/mouse action:	*Function:*
Left click on an object	Select object (but do not perform selection-based navigation).
Left button drag	Change x and y position of selected object.
Left button hold + scroll wheel rotate	Change z position of selected object.
Right button drag	Change rotation of selected object around x and y axes.
Right button hold + scroll wheel rotate	Change rotation of selected object around z axis.
Y / H	Change y scale of selected object.
G / J	Change x scale of selected object.
V / M	Change z scale of selected object.

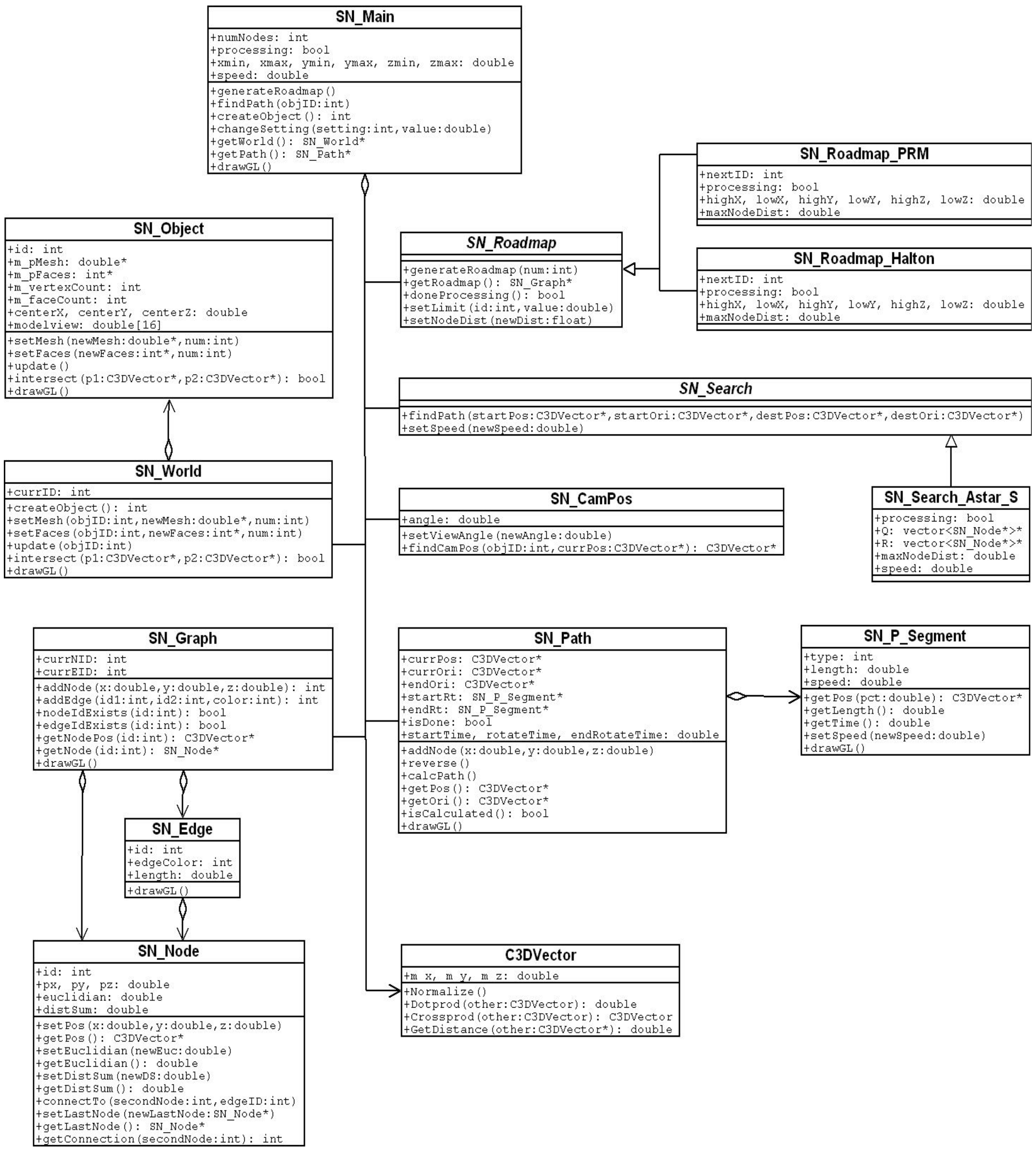
Console commands

Command:	*Function:*
Up / Down (arrow keys)	Scroll through previously entered commands.
set debug #	Toggles debug mode. 0 = off, 1 = on. (default = 0)
set grid #	Toggles grid. 0 = off, 1 = on. (default = 1). Note that using the grid can result in better viewer orientation during movement.
load *filename*	Loads a file containing a pre-specified environment.
save *filename*	Saves the current environment into a file. Only the objects, their

	parameters and the environment boundaries are stored, the roadmap, path and camera position will have to be regenerated. Spaces are allowed in the filename.
lm *filename*	Loads an .ASE model into the scene. (Note: objects cannot be deleted once added)
set cammode #	Select camera mode. 1 = free movement, 2 = free movement but with look-at enabled (awkward), 3 = selection-based navigation. (default = 3)
set nodedist #	Sets the maximum node distance for which edges are created between the nodes. (default = 10.0)
set numnodes #	Set the number of nodes to be generated. (default = 100)
set xmax #	Set the maximum x value of the environment*. (default = 4)
set xmin #	Set the minimum x value of the environment*. (default = -4)
set ymax #	Set the maximum y value of the environment*. (default = 4)
set ymin #	Set the minimum y value of the environment*. (default = -4)
set zmax #	Set the maximum z value of the environment*. (default = 4)
set zmin #	Set the minimum z value of the environment*. (default = -4)
set roadmap $	Select the road mapping method. prm = random sampling, halton = Halton point sampling. (default = halton)

* = setting the min and max value of a certain dimension to the same value will result in a 2D roadmap. However, the current and final camera positions do not take these constraints into account when a path is generated, and the resulting movement will not be fully 2D.

Appendix B: UML class diagram


SN_Main
+numNodes: int
+processing: bool
+xmin, xmax, ymin, ymax, zmin, zmax: double
+speed: double
+generateRoadmap()
+findPath(objID:int)
+createObject(): int
+changeSetting(setting:int,value:double)
+getWorld(): SN_World*
+getPath(): SN_Path*
+drawGL()
SN_Roadmap_PRM
+nextID: int
+processing: bool
+highX, lowX, highY, lowY, highZ, lowZ: double
+maxNodeDist: double
SN_Object
+id: int
+m_pMesh: double*
+m_pFaces: int*
+m_vertexCount: int
+m_faceCount: int
+centerX, centerY, centerZ: double
+modelview: double[16]
+setMesh(newMesh:double*,num:int)
+setFaces(newFaces:int*,num:int)
+update()
+intersect(p1:C3DVector*,p2:C3DVector*): bool
+drawGL()
SN_Roadmap
+generateRoadmap(num:int)
+getRoadmap(): SN_Graph*
+doneProcessing(): bool
+setLimit(id:int,value:double)
+setNodeDist(newDist:float)
SN_Roadmap_Halton
+nextID: int
+processing: bool
+highX, lowX, highY, lowY, highZ, lowZ: double
+maxNodeDist: double
SN_Search
+findPath(startPos:C3DVector*,startOri:C3DVector*,destPos:C3DVector*,destOri:C3DVector*)
+setSpeed(newSpeed:double)
SN_World
+currID: int
+createObject(): int
+setMesh(objID:int,newMesh:double*,num:int)
+setFaces(objID:int,newFaces:int*,num:int)
+update(objID:int)
+intersect(p1:C3DVector*,p2:C3DVector*): bool
+drawGL()
SN_CamPos
+angle: double
+setViewAngle(newAngle:double)
+findCamPos(objID:int,currPos:C3DVector*): C3DVector*
SN_Search_Astar_S
+processing: bool
+Q: vector<SN_Node*>*
+R: vector<SN_Node*>*
+maxNodeDist: double
+speed: double
SN_Graph
+currNID: int
+currEID: int
+addNode(x:double,y:double,z:double): int
+addEdge(id1:int,id2:int,color:int): int
+nodeIdExists(id:int): bool
+edgeIdExists(id:int): bool
+getNodePos(id:int): C3DVector*
+getNode(id:int): SN_Node*
+drawGL()
SN_Path
+currPos: C3DVector*
+currOri: C3DVector*
+endOri: C3DVector*
+startRt: SN_P_Segment*
+endRt: SN_P_Segment*
+isDone: bool
+startTime, rotateTime, endRotateTime: double
+addNode(x:double,y:double,z:double)
+reverse()
+calcPath()
+getPos(): C3DVector*
+getOri(): C3DVector*
+isCalculated(): bool
+drawGL()
SN_P_Segment
+type: int
+length: double
+speed: double
+getPos(pct:double): C3DVector*
+getLength(): double
+getTime(): double
+setSpeed(newSpeed:double)
+drawGL()
SN_Edge
+id: int
+edgeColor: int
+length: double
+drawGL()
SN_Node
+id: int
+px, py, pz: double
+euclidian: double
+distSum: double
+setPos(x:double,y:double,z:double)
+getPos(): C3DVector*
+setEuclidian(newEuc:double)
+getEuclidian(): double
+setDistSum(newDS:double)
+getDistSum(): double
+connectTo(secondNode:int,edgeID:int)
+setLastNode(newLastNode:SN_Node*)
+getLastNode(): SN_Node*
+getConnection(secondNode:int): int
C3DVector
+m_x, m_y, m_z: double
+Normalize()
+Dotprod(other:C3DVector): double
+Crossprod(other:C3DVector): C3DVector
+GetDistance(other:C3DVector*): double