A framework for real-time nonholonomic path planning in huge terrain datasets

Michael Guthe and Reinhard Klein[1]
Universität Bonn, Römerstraße 164, 53117 Bonn, Germany

Abstract. When users navigate through a terrain dataset, they typically control the camera directly. However, such control is difficult and often results in rather unsteady camera motions that can easily cause motion sickness, especially in virtual reality. In this paper we describe a new technique for automatic generation of a camera motion path using nonholonomic motion planning techniques. Using our approach, the user simply specifies an object of interest, e.g. on a map or from a list of landmarks, together with the desired viewing distance associated with that object. Then the system automatically computes a smooth camera motion from the current position and orientation to show the specified point. Our framework allows the simple implementation of different navigator metaphors that specify the camera behaviour. Due to the size of recent terrain datasets and the complexity of nonholonomic motion planning, using standard approaches is not possible. To solve this problem, we present a multi-scale approach using a simple and fast path planning to find coarse resolution paths and iteratively refine them. Although developed for terrain exploration, our framework is general enough to be used in any virtual environment. One of the key advantages of our method is, that the user can take over control at any time, e.g. to examine a close-by object, and switch back to automatic navigation later. This way our path planning framework can also be regarded as navigation assistant.

1. Introduction

In many virtual environment applications, like games and terrain rendering systems, the user must navigate through the environment to inspect it and eventually perform certain tasks, where navigation means steering a virtual camera through the environment. In most of today's systems this is done using the mouse and/or arrow keys on the keyboard. Such direct control has a number of disadvantages:

1. Motion from direct user input can cause motion sickness due to unsteady movement and unforeseeable changes in direction.
2. Navigation is difficult, especially for users that “just want to have a look” or do not know the environment.
3. Shorter and/or better paths than that chosen by the user may exist.
4. Navigation requires a lot of attention while the user should preferably concentrate on more high-level tasks at hand.

Therefore, automatic path and motion planning has received a lot of attention recently. The goal is, that the user simply specifies a target point (e.g. on a map), together with an optional direction or a distance to the target, and the system automatically computes a smooth motion from the current position and orientation to this target, while avoiding collisions with obstacles in the scene and possibly considering other constraints, like kinematic or dynamic properties of the vehicle.

In this paper we describe a new technique for navigation through large virtual environments by the example of terrain datasets. Our approach is based on motion planning techniques, in particular the nonholonomic trajectory planning that has been successfully applied to many

[1] {guthe,rk}@cs.uni-bonn.de

problems, ranging from navigation of autonomous vehicles to steering of robot arms. To this end, we consider the vehicle as rigid body that can be represented by a six-dimensional configuration, where the current movement direction and the viewing direction do not need to be identical, i.e. a vehicles could be able move sideways. Our method then generates a sequence of control inputs for a dynamic simulation of the vehicle. During this simulation, three spaces are required: the configuration space, defined as position and view direction, the derivative space, which is a representation of the current movement, and the control function space, that is composed of the movement derivatives. Note, that the first two are often combined to the so-called state space. The motion generated by our method results in a smooth and natural path, which is much less inclined to produce motion sickness than a direct camera control by the user.

Since the distance between start and goal can be arbitrarily long (e.g. up to 20,000 km on earth), we developed a multi-scale algorithm for path planning. The method is fast and partial results are available for time critical computing, which is important, since camera motions need to be computed in real-time if the user can specify targets interactively. The method has successfully been integrated in a system for exploration of huge terrain datasets [16] and experiments show that the resulting motions are indeed pleasant to watch. In addition to the fully automatic navigation it is also possible for the user to take control of the camera at any time. This can be important, if an interesting object or landmark appeared, that the user wants to inspect. Later the user can switch back to automatic navigation and continue to travel to the previous target. Therefore, the system can be considered as navigation assistant for situations, where the user does not know the way to the target, or does not want to move by hand.

2. Related Work

In first path planning systems, a robot was regarded as point that can move freely, either on the ground, or in space. For rigid objects, the abstraction of the configuration space (C-space) can be made. A point in C-space is then a unique representation of a robot configuration, which is typically expressed as vector of position and orientation parameters. The configuration space is then partitioned in free (C_{free}), contact and blocked sets. A free path in configuration space is then a continuous curve in C_{free} connecting a start and end configuration. Since path planning is a continuous problem, it needs to be discretized to solve it numerically. In early techniques either grid based approaches [14] or visibility graphs [15] were used. Later also a decomposition into collision free convex cells was proposed [4]. The grid cells are then considered as nodes and their adjacency relation as edges of a graph. This way the path planning problem leads to a shortest path problem which is well studied in graph theory, and where the complexity exponentially depends on the degrees of freedom (e.g. 2^d).

Imposing further dynamic constraints leads to the next level of complexity. These can either be constraints that can be represented as explicit functions of the configuration variables and time, or constraints on the differentials of the configuration variables and time. If only the first type of constraints is present, the problem is called holonomic and can simply be solved by removing all invalid configurations from C_{free} and thus from the graph. When at least one constraint of the second type exists, the problem is called nonholonomic. Since the constraints are defined over additional variables (e.g. derivatives), some problems can be made holonomic by extending the set of variables at the cost of increasing dimensionality. However, this is only possible, if the constraint function is integrable. One of the first nonholonomic path planning algorithms was introduced along with the problem of parking a car [8]. Nonholonomic path planning algorithms often calculate control functions [3], instead of an actual path, to reach a goal configuration, since the constraints are often defined on first or second order derivatives of the configuration variables (e.g. a maximum acceleration) rather than the configuration variables themselves. To generate smooth paths, sinusoids can

be used as control functions [11]. During movement in a virtual environment, the state is then updated, with the current control input, using the laws of dynamics. Since in our case further variables like direction and speed have to be taken into account, in addition to a 3-dimensional position, the dimensionality of the grid and therefore the complexity of the graph cannot be handled anymore.

A method to reduce the complexity of the graph is the probabilistic roadmap algorithm [7], were only a small number of samples is drawn from the state space and a transition graph between them is generated. At runtime, only two additional transitions have to be calculated, one from the start to the first node in the graph and one from the last node to the goal state. Although recently good heuristic to generate the sample states for the roadmap have been proposed [12], for larger environments (e.g. as in terrain rendering), the roadmap becomes too large and thus the graph too complex.

Since the paths generated by traditional motion planning systems, do not regard aesthetical aspects, like steady rotations and accelerations or visibility of the target, specialized path planning algorithms were developed for camera motions. Most previous work on planning camera motions in virtual environments is directed toward systems in which the camera must follow an object (like in third-person games). A distinction can be made here in systems where the motion of the object is known beforehand [10] and systems where this motion is not known [1,2,6]. Similar problems have been studied in robotics where a robot with a camera must track certain targets [5,9]. Recently, automatic generation of smooth camera paths has been studied [13] using a probabilistic roadmap approach.

3. Aesthetical Aspects

For the generation of a camera path, two different types of constraints can be observed. The first type are hard constraints, like maximum rotation speed and acceleration, or avoidance of collisions. These constraints are of boolean nature and can be enforced by forbidding transitions between states that exceed the specified limits. The second type of constraints are more difficult to account for, since they cannot be expressed as explicit criterion (e.g. a path is not interesting or boring, but may be less or more interesting than another one) and not all of them can be fulfilled at the same time. For example, a short path may be less interesting, or an interesting path may have too many direction changes and would thus promote motion sickness. Therefore, only a trade off between these fuzzy constraints can be made.

3.1. Boolean Constraints

First of all, avoiding collisions is achieved by defining states where a collision occurs as invalid by removing them from C_{free} and thus not include them in the search. In some cases this might not be enough, since we also have to exclude all that states from C_{free}, where a collision is inevitable within the next n seconds.

Since the paths are generated for camera motions, a high continuity of the path (at least C^2) is required, and therefore, we use sinusoidal functions. Other constraints, e.g. maximum rotation speed or restriction of the movement direction and maximum speed, depend on the actual navigator metaphor that is used to generate the camera motion. The framework proposed in Section 4 is flexible enough to support different types of navigators, by defining their movement characteristics and constraints. The only functions that need to be specified for an additional type of navigator, are basically only a function to calculate the movement from an input state and a given control input, as well as a function to calculate a lower bound for the remaining cost (see Section 4).

3.1.1. Free-Flight Navigator

The simplest type of navigator is one that can move independently in all the coordinates. The only additional constraints here are maximum velocities and maximum accelerations in each direction. In this case it might happen that the navigator stops and starts in a different direction. Therefore, a tight coupling between movement and viewing direction is not reasonable. A possibility for the view direction is the current target, which would reduce the number of constraints, since the camera does not need to move towards the target at the end of the path. Since this however leads to sideways movements, that can cause motion sickness, we use the following combination of target direction and current movement direction:

$$\vec{v}_{view} = \frac{\vec{v}_{target}}{\left\|\vec{v}_{target}\right\|} + c_{view}\,\frac{\vec{v}_{move}}{v_{max}},$$

where v_{max} is the maximum velocity and c_{view} a user specified constant to steer the influence of the movement direction. Since discontinuities still occur when the target changes, a maximum rotation speed rot_{max} is used to determine the final view direction.

3.1.2. Plane Navigator

The probably most natural navigation metaphor for terrain rendering is the plane or flight navigator. Here the view direction is coupled with the movement direction and is traditionally specified using two angles, one for the direction and one for the so-called pitch. When we disregard start and landing, the navigator cannot decrease its speed below a minimum due to aerodynamic constraints, such that nonholonomic path planning algorithms are required.

3.1.3. Helicopter Navigator

Another possible metaphor is a helicopter which is a combination of both previous types. In the xy-plane, it moves similar to the plane navigator, with the exception that it can stop in mid-air, but in the z-direction it can hover up and down practically independent from the planar movement. However, since a helicopter cannot tilt down without moving forward, there are much less possible target states than for the plane navigator.

3.2. Fuzzy Constraints

We call a path aesthetically pleasing, if the control is of low frequency and if interesting objects or landmarks are visible. The first goal can be achieved, when a penalty p_c (e.g. half a second) is added each time a control input changes and thus a slightly longer path is tolerated in exchange for a smoother movement. The second goal is much harder to achieve, as it is not always clear what is interesting. Therefore, we use the heuristic, that interesting objects are probably visible, if the height of the path above ground is not too large. On the other hand, if the path is too close to the surface, only very close objects are visible. As compromise between those two aspects, a penalty is added to the path cost, if the height becomes too low or too high. This penalty p is zero inside an interval $\left[h_{min}, h_{max}\right]$ that is assumed to be an acceptable height and increases linearly above and below that height:

$$p = \begin{cases} p_h\left(h_{min} - h\right) & : \quad h < h_{min} \\ p_h\left(h - h_{max}\right) & : \quad h > h_{max} \\ 0 & : \quad \text{else} \end{cases}$$

4. Path Finding Framework

For larger environments, calculating a complete path using nonholonomic motion planning algorithms becomes too slow, due to the high dimensionality of the state space (five to six dimensions are required for the navigators discussed in Section 3). The main idea to solve this problem is based on the following observation: on larger scales, path planning of a real-world vehicle can be assumed to be approximately a holonomic problem, since on a larger scale the turning radius of a vehicle can be considered as a point. To find the appropriate scale, we determine the turning diameter of the navigator at maximum speed. With a grid step size larger than the turning diameter, the problem can be considered as holonomic. To exploit this fact, we use a multi-scale approach. First, an A* algorithm [14] is used to generate a coarse path. In this algorithm a steps size of the turning diameter can however still be too small to calculate a shortest path of several thousand kilometres in real-time. For this reason, a hierarchical approach is used, starting with a step size of e.g. about 1000 km on earth, to be able to even support global-scale path planning. The following algorithm is performed to hierarchically generate sub-targets at a distance of n-times the current step size:

```
while(StepSize > Diameter) {
    if (Distance(Start,Target) > n*StepSize) {
        Path[] = PerformAStar(Start,Target,StepSize)
        Target = Path[n]
        StepSize = StepSize/2

    }

}
```

The A* graph search algorithm improves Dijskstra's algorithm by calculating a lower bound on the rest cost to the target (typically Euclidian distance). Instead of expanding the search at the node with the lowest accumulated cost from the starting point, the search is expanded at the node with the lowest total cost. To compensate for different speed of the navigator in different directions, we use a slightly modified metric:

$$\left\| \begin{pmatrix} x \\ y \\ z \end{pmatrix} \right\|_{A^*} = \sqrt{x^2 + y^2 + \left(w_z z\right)^2} ,$$

where w_z specifies the cost ratio of a movement in z direction compared to x or y direction, i.e. the maximum horizontal speed divided by the maximum vertical speed.

When the step size falls below the turning diameter, the problem cannot be considered holonomic any more. Now we have to solve a nonholonomic problem, where the target is defined by the new sub-target from the hierarchical A* algorithm. For an efficient calculation of the required control input to reach this target, we discretize the state space with a fixed time step of e.g. one second. Furthermore, we assume that the control input variables are binary, with sinusoidal changes of the control input between two time steps [11]. Note, that direct transition between +1 and –1 is not allowed. Now an A* algorithm can be used to calculate a path which adheres to all constraints. The only difference is, that the graph is generated locally on demand. When a node of the search tree is expanded, child nodes for all possible control inputs are generated. The new state of the each child node is calculated by simulating the behaviour of the navigator with the according control input for one time step.

In this algorithm we have to distinguish between sub-targets that are generated by the holonomic part of the algorithm and the global target. While there are no specific constraints for the sub-target, different constraints might be specified for the global target. In this case, the navigator has to stop at a specified distance while facing towards the not occluded target point. Therefore, in most cases the problem does not have a unique solution. E.g. in Figure 1, where the final target is located at the red point, we search for a position with a predefined distance to the target. In this case, the solution would that parts of the sphere from which the

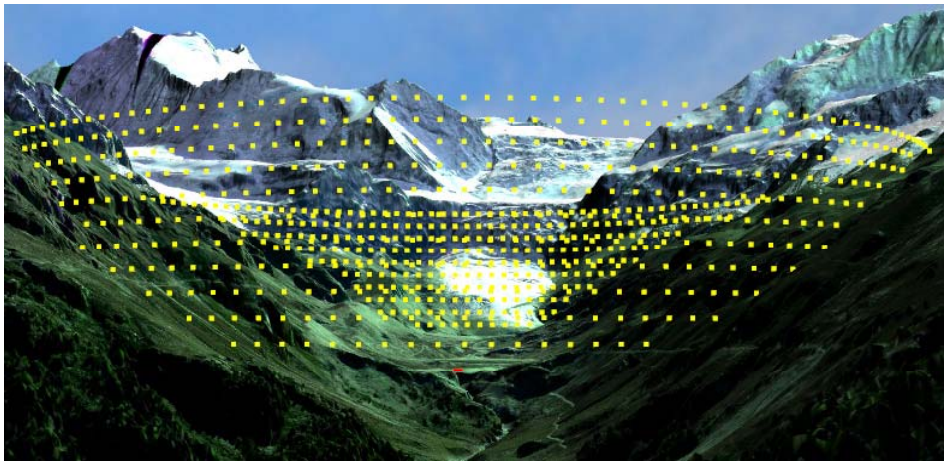

Figure 1: Example of valid beacon points (yellow) – shown at 2 degree resolution – for a specified target point (red).

target point is visible. Since this is again continuous, a set of discrete beacons is calculated and the currently fastest reachable is used. For terrain rendering, it is sufficient to calculate a minimum declination angle for every direction (we use 360 discrete directions). If the declination is higher than the navigator is able to face downward, the closest possible direction is used instead. Figure 1 shows all valid beacons for a specified target point. Additionally to reaching a beacon point, the navigator has then to face the target point to reach a valid goal state. Therefore, the target direction has to be integrated into the rest cost estimation.

4.1. Priority Queue

The key to an efficient implementation of all graph search algorithms, independent of whether Dijkstra's algorithm or A* is used, is a fast priority queue. The standard approach of using an AVL-tree leads to a complexity of $O(n \log n)$. Considering that several hundred thousand entries need to be sorted, it is clear, that such a queue will be the bottleneck of the algorithm. However, not all entries inserted into the queue are processed during the A* algorithm since all states with a higher estimated total cost than the shortest path are not considered any more. We exploit this fact by first performing a coarse sorting of the inserted elements using a heap. To scale an arbitrary positive floating point value into the interval of discrete bins, the following function is used:

$$key = \frac{priority \cdot (\#bins - 1)}{priority + (\#bins - 1)},$$

where the priority is proportional to the estimated total cost of the entry. Since the estimated total cost is always higher than that of the start state and this difference is increasing relatively slow if the rest cost estimation is accurate, the priority is calculated as follows:

$$priority = c_{pscale}\left(cost - cost_{start}\right),$$

with the scaling constant c_{pscale} which depends on the number of bins.

First, each bin contains an unsorted list of its entries. When the first entry of a bin is required, which occurs when all bins below are empty, the entries in the bin are sorted into an AVL-tree. If a new entry is added to an already sorted bin, it is inserted into the AVL-tree, since the probability is very high, that the next entry of this bin is required. When enough bins are used and the scaling constant is high enough, the average runtime becomes linear in the number of entries. The worst case runtime becomes $O(n + m \log k)$, where n is the number of inserted entries, m the number of entries read from the queue and k the average number of entries per bin. This increased the performance of our algorithm by a factor of about 20.

4.2. Rest-Cost Estimation

Since the derivatives of the state variables can have a high impact on the required time to reach a target configuration, a rest cost estimation based on Euclidian distance alone is not a good choice. The constraints discussed in Section 3 restrict the maximum acceleration and possibly the maximum speed, as well as rotation speed. The bound on acceleration and possibly maximum speed can be accounted for with the following estimation for the time t required to travel a given distance d:

$$t = \begin{cases} \frac{2\sqrt{d \cdot a_{max} + \frac{1}{2}\left(v_s^2 + v_e^2\right)} - v_s - v_e}{a_{max}} & : \text{ if } \sqrt{d \cdot a_{max} + \frac{1}{2}\left(v_s^2 + v_e^2\right)} \le v_{max} \\ \frac{2v_{max} - v_s - v_e}{a_{max}} + \frac{d - \frac{1}{a_{max}}\left(v_{max}^2 - \frac{1}{2}v_s^2 - \frac{1}{2}v_e^2\right)}{v_{max}} & : \text{ else} \end{cases},$$

where v_s and v_e are the signed start and end velocities, and a_{max} is the maximum allowed acceleration. This not only applies to straight paths, but also for travelling in independent dimensions, as for the free-flight navigator.

When a constraint on the rotation speed exists, this leads to an upper bound for the curvature. To estimate the minimum path length under these constraints, we made the following two observations:

1. The shortest bounded curvature path in 2d from a staring point and direction to a goal point and direction is always composed of two circular arcs (one at the start and one at the end of the path) and a linear tangential conjunction between them.
2. The radii of these circular arcs are always those with the maximum allowed curvature.

These observations lead to the four possible configurations shown in Figure 2. The shortest path length of these configuration is then a lower bound for the remaining path length.

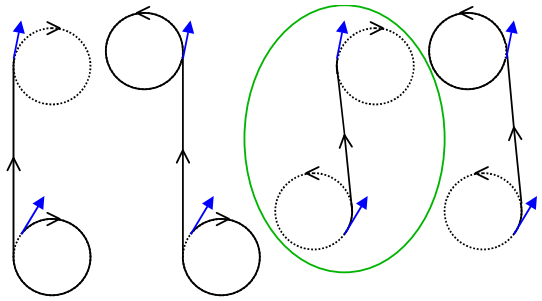

Figure 2: Example for shortest bounded curvature path when start and goal are far apart.

From the example in Figure 2 the idea may rise, that the shortest path is always the one that turns towards the target point at the start and towards the target direction at the end of the path. This however is not true when start and end point are very close, since a tangential linear path between the two circles cannot be constructed, if the circles intersect and have different orientation as shown in Figure 3.

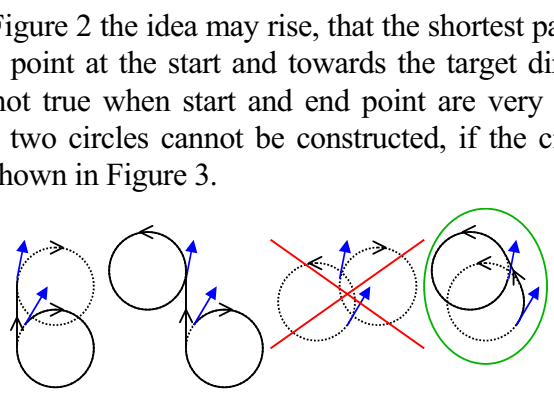

Figure 3: Example for shortest bounded curvature path when start and goal are close.

The added penalty system, reduces the efficiency of the A* approach, since the estimated rest cost is a too low underestimation. Therefore, we keep track of the average penalty per time step and assume that future time steps will have at least half of this average penalty. Note, that this is a trade-off between quality and speed, as a too high estimated penalty leads to sub-optimal paths.

4.4. Time Critical Computing

Although this multi-scale algorithm is very fast, the next control input may be required before the complete path has been calculated. Additionally, due to the high dimensionality of the state-space, the number of nodes in the search tree might become too high to keep all of them in memory. Both problems are solved by using a both, time- and memory-critical computing approach. The basic idea is to prune the search tree, whenever a new control input is required or a maximum number of states in the search tree is reached. One of the current root nodes children is selected and added to the generated path. Then the sub-trees of all other child nodes are removed and the selected child becomes the new root of the search tree. This approach however leaves the question, which child node should be selected as next step of the path. In other words: from all current leaf nodes of the search tree, the most promising one needs to be found. Since the leaves of the search tree are expanded in order of the estimated total cost, it is not reasonable to use this as criterion for the most promising leaf, as this will always be the one that would be expanded next. On the other hand, selecting by estimated rest cost will lead to the generation of high penalty paths. The most reasonable heuristic to determine a promising leaf node is therefore, to add the estimated rest cost and the accumulated penalty. Note, that this is equal to the estimated total cost minus the path time.

5. Results

The path finding framework was evaluated with the three presented navigator types using the Turtmann Valley dataset [16]. The maximum memory allocation for the search tree was set to 20 Mbytes, resulting in $n_{steps} = 476,625$, since each node of the search tree needs 44 bytes. As time step $\Delta t = 1 sec$ was used and the number of bins was set to $10,000$ with the scaling value $c_{pscale} = 10 \frac{1}{sec}$. For the aesthetical constraints, the following penalty settings were used: $p_c = 0.5 sec$, $h_{min} = 175m$, $h_{max} = 225m$, and $p_h = 0.001 \frac{sec}{m}$.

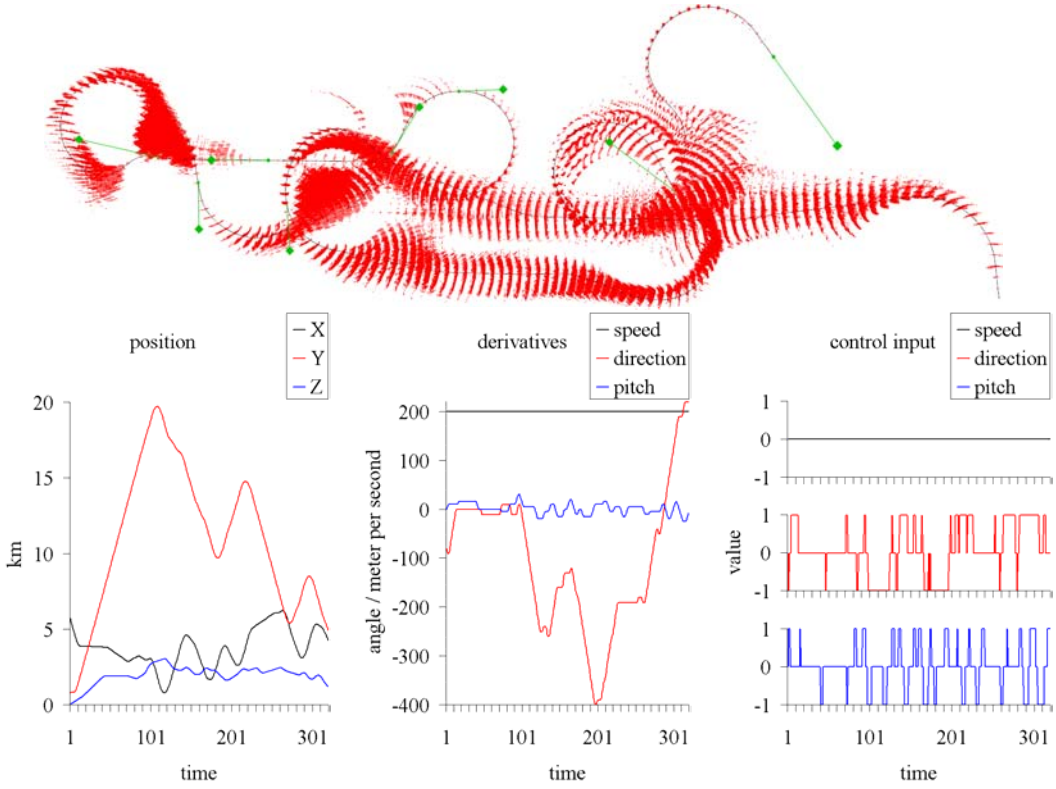

Figure 4: Generated path and samples in state-space (red) for the plane navigator (beacon and target point are shown in green) with position, derivatives and control input.

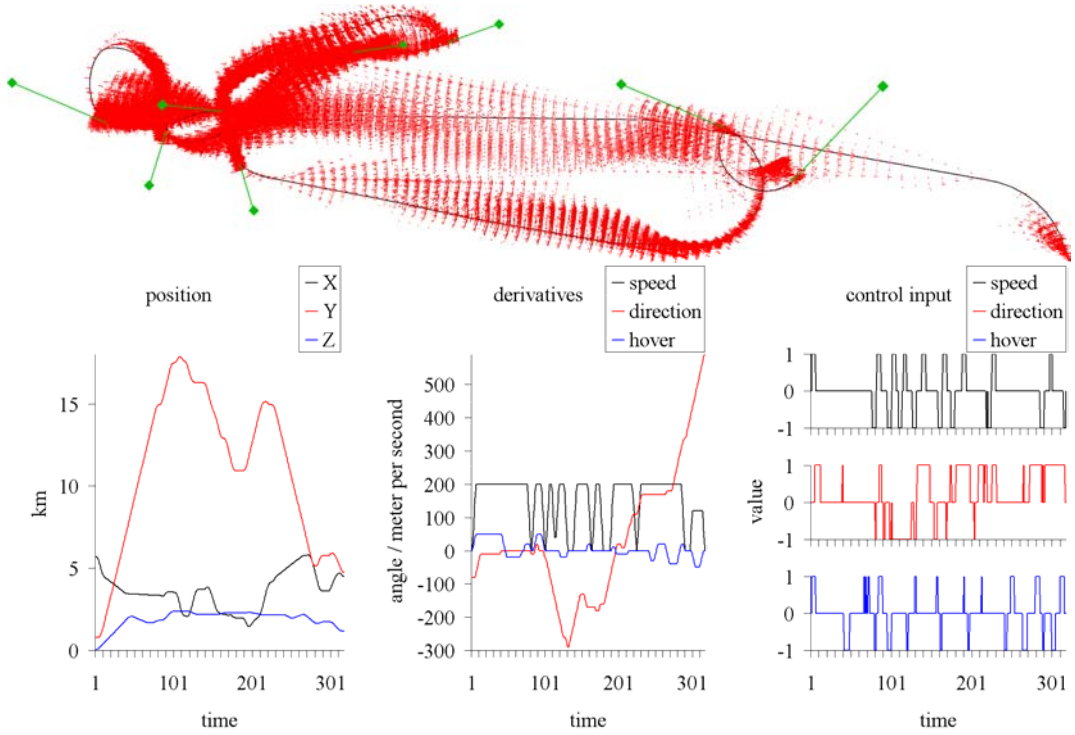

Figure 5: Generated path, position, derivatives and control input for the helicopter navigator.

The generated camera path for the plane navigator – with the boolean constraints $v_{min} = v_{max} = 200 \frac{m}{sec}$, $rot_{dir} = 10 \frac{deg}{sec}$, $rot_{pitch} = 5 \frac{deg}{sec}$, and $pitch_{max} = 50°$ – is shown in Figure 4. In the top image, the path is shown in black, while the samples in state space are shown in red. The target and beacon points are drawn in green, connected with a line to depict the current view direction. Below, the position, derivatives and generated control input are shown. Figure 5 shows the path, derivatives and control for the helicopter navigator with $v_{min} = 0 \frac{m}{sec}$, $v_{max} = 200 \frac{m}{sec}$, $hover_{max} = 50 \frac{m}{sec}$, $a_v = 40 \frac{m}{sec^2}$, $a_{hover} = 10 \frac{m}{sec^2}$, and $rot_{max} = 10 \frac{deg}{sec}$. Finally, Figure 6 shows the path, derivatives and control for the free-flight navigator with $v_{max} = [200 \quad 200 \quad 20] \frac{m}{sec}$, $a_{max} = [40 \quad 40 \quad 4] \frac{m}{sec^2}$, as well as $c_{view} = 2$ and $rot_{max} = 20 \frac{deg}{sec}$ for the view direction.

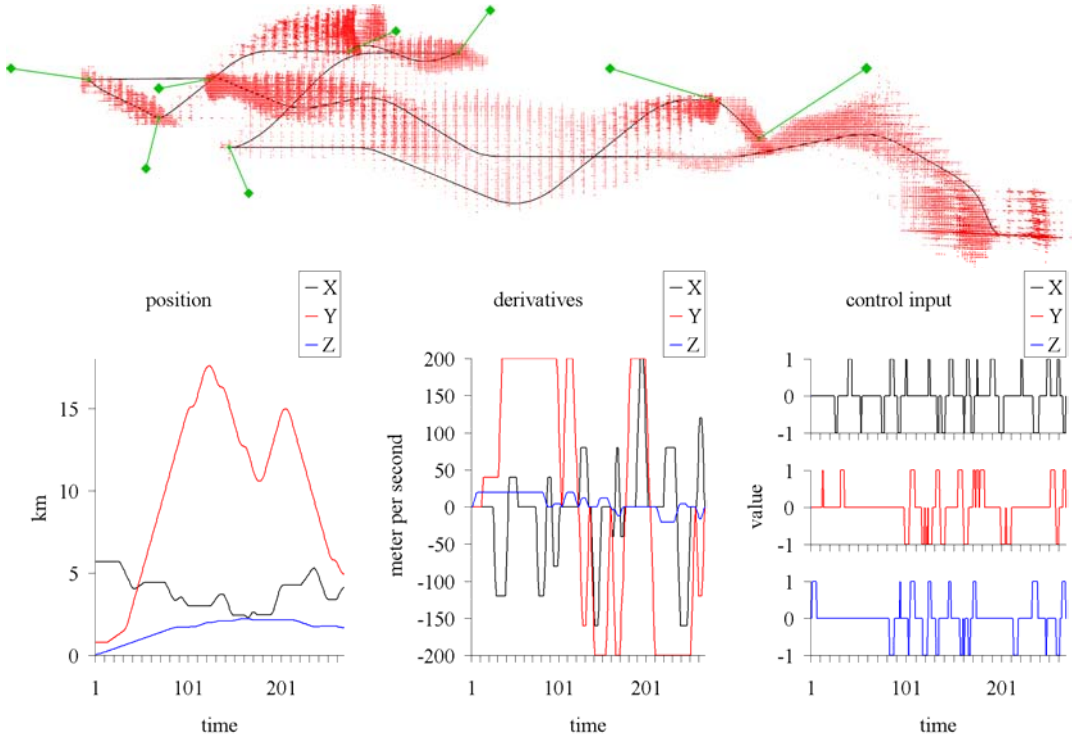

Figure 6: Generated path, position, derivatives and control input for the free-flight navigator.

The accompanying video shows a combination of the three navigator metaphors for the interest points on the Turtmann Valley used for evaluation. During the flight, the already determined part of the path is shown in green, while the currently most promising rest path is shown in yellow. It can be observed that on the used PC (Athlon 3000+), the search tree is pruned due to the maximum number of nodes most of the time, which demonstrates the real-time capabilities of the presented path finding framework.

6. Conclusion and Future Work

We have presented a general framework for real-time automatic navigation in large virtual environments. As basis for the navigator not only short travelling time while respecting nonholonomic constraints, but rather aesthetical aspects like steadiness and generation of interesting paths were in the foreground. The path finding is running well in real-time on a standard PC, even though it is only a background process while most of the CPU time is spend for terrain rendering. The resulting motions are smooth and much less inclined to produce motion sickness than manual navigation or flight along shortest paths. Since the user can take over control at any time at switch back to automatic navigation when he desires, the method can be regarded as navigation assistant.

References

[1] W. H. Bares, J. Grgoire, and J. Lester, Realtime constraint-based cinematography for complex interactive 3d worlds, *IAAI-98: Proc. of the10th Conference on Innovative Applications of Artificial Intelligence*, 1998, pp. 1101-1106.

[2] N. Courty and E. Marchand, Computer animation: a new application for image-based visual servoing, *Proc. IEEE Int. Conf. on Robotics and Automation*, 2001, pp. 223-228.

[3] T. Fraichard, Dynamic trajectory planning with dynamic constraints: a 'state-time space' approach, *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 1993, pp. 1393-1400.

[4] G. Giralt, R. Sobek, and R. Chatila, A multi-level planning and navigation system for a mobile robot: a first approach to Hilare, *6th Int. Joint Conf. on Artificial Intelligence*, 1979, pp. 335-337.

[5] H. H. Gonzales-Banos, C. Y. Lee, and J.-C. Latombe, Real-time combinatorial tracking of a target moving unpredictably among obstacles, *Proc. IEEE Int. Conf. on Robotics and Automation*, 2002.

[6] N. Halper, R. Helbing, and T. Strothotte, Computer games: A camera engine for computer games, *Computer Graphics Forum* **20**, 2001.

[7] L. Kavraki and J.-C. Latombe, Randomized preprocessing of configuration space for fast path planning, *Proc. IEEE Int. Conf. on Robotics and Automation*, 1994, pp. 2138-2145.

[8] J. P. Laumond, Feasible trajectories for mobile robots with kinematic and environment constraints, *Intelligent Autonomous Systems*, 1987, pp. 346-354.

[9] S. M. LaValle, H. H. Gonzales-Banos, C. Becker, and J.-C. Latombe, Motion strategies for maintaining visibility of a moving target, *Proc. IEEE Int. Conf. on Robotics and Automation*, 1997.

[10] T.-Y. Li, T.-H. Yu, Planning object tracking motions, *Proc. IEEE Int. Conf. on Robotics and Automation*, 1999.

[11] R. M. Murray and S. S. Sastry, Nonholonomic motion planning: steering using sinusoids, *IEEE Trans. on Automatic Control* **38**(5), 2003, pp. 700-716.

[12] D. Nieuwenhuisen, A. Kamphuis, M. Mooijekind, and M. H. Overmars, Automatic construction of high quality roadmaps for path planning, *Technical report UU-CS-2004-068 (Universiteit Utrecht)*, 2004.

[13] D. Nieuwenhuisen and M. H. Overmars, Motion planning for camera movements in virtual environments, *Technical report UU-CS-2003-004 (Universiteit Utrecht)*, 2003.

[14] N. J. Nielsson, A mobile automaton: an application of artificial intelligence techniques, *1st Int. Joint Conf. on Artificial Intelligence*, 1969, pp. 509-520.

[15] A. Thompson, The navigation system of the JPL robot, *5th Int. Joint Conf. on Artificial Intelligence*, 1977, pp. 749-757.

[16] R. Wahl, M. Massing, P. Degener, M. Guthe, and R. Klein, Scalable compression and rendering of textured terrain data, *Journal of WSCG* **12**(3), 2004, pp. 521-528.