Realtime motion planning for cameras in dynamically changing virtual environments

Bachelor's Thesis

Frank Schreiber

Supervisor: **Prof. Dr. Elmar Schömer**
Institute for Computer Science
Johannes-Gutenberg University Mainz

March 14th, 2008

I, Frank Schreiber , hereby affirm that I have independently composed this bachelor's thesis and that I have used no sources or aids other than those mentioned.

Hiermit versichere ich, Frank Schreiber, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine Anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Frank Schreiber

Mainz, March 14th, 2008

Contents

Bibliography

[CRM1988] J. Canny: *The Complexity of Robot Motion Planning*, MIT Press, Cambridge, Mass. 1988

[SES1991] E. Welzl: *Smallest enclosing disks, balls and ellipsoids* , Tech. Report No. B 91-09, Fachbereich Mathematik, Freie Universitat Berlin, Berlin, Germany, 1991.

[PRM1996] L. Karvraki, P. Svestka, K.-C. Latombe, M. Overmars: *Probabilistic roadmaps for Path Planning in High-Dimensional Configuration Spaces*, in *IEEE Transactions on Robotics and Automation*, 12(4), 1996.

[CGC1998] J. O'Rourke: *Computational Geometry in C*, Second Edition, Cambridge University Press, New York, USA, 1998.

[POT1999] T.-Y. Li, T.-H. Yu: *Planning object tracking motions*, Proc. IEEE Int. Conf. on Robotics and Automation, 1999.

[CG22000] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf: *Computational Geometry - Algorithms and Applications*, Second, Revised Edition, Springer-Verlag Berlin Heidelberg New York, USA, 2000.

[IBV2000] E. Marchand, N. Courty (2000): *Image-based virtual camera motion strategies*, Graphics Interface Conference, GI2000, pages 69-76, 2000.

[3MP2001] R. Fitch, Z. Butler, D. Rus: *3D Rectilinear Motion Planning with Minimum Bend Paths*, in: *Intelligent Robots and Systems*, IEEE/RSJ International Conference Volume 3, pages 1491-1498, 2001

[RTR2002] T. Akenine-Möller, E. Haines: *Real-Time Rendering*, Second Edition, A. K. Peters, Ltd., Wellesley, USA, 2002.

[MPC2003] D. Nieuwenhuisen, M. Overmars: *Motion Planning for Camera Movements in Virtual Environments*, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, January 30, 2003.

[MHT2004] R. Murrieta-Cid, A. Sarmiento, S. Kloder, S. Hutchinson, F. Lamiraux, J. P. Laumond: *Maintaining Visibility of a Moving holonomic Target at a Fixed Distance with a Non-Holonomic Robot*, LAAS/CNRS, France, 2004.

[AGC2004] O. Goemans, M. Overmars: *Automatic Generation of Camera Motion to Track a Moving Guide*, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2004.

[BIH2006] C. Wächter, A. Keller: *Instant Ray Tracing: The Bounding Interval Hierarchy*, in: Eurographics Symposium on Rendering, University of Ulm, Germany, 2006.

[PAL2006] S. M. LaValle: *Planning Algorithms*, University of Illinois, Cambridge University Press, 2006. http://planning.cs.uiuc.edu/

Introduction

In the mid 90's, a famous game development company produced and released a game called *The incredible machine*. It consists of using and placing many different objects like balls, ropes, buckets, rockets, gear systems and all sort of other things, to achieve single tasks like turning on a light bulb or collecting all balls into a basket. The amount and type of objects that were available to the player were limited, and the player had to position them correctly, so that when the simulation was started, the resulting chain reaction of events led to the required goal. Back then, the environment was two-dimensional and no complex physical calculations were made when simulating the *incredible machine*.
Nowadays, more or less ten years later, the computation power of common computers is sufficient enough to handle complex graphical and physical calculations in realtime[1].
Several students of the University of Mainz grouped together with the idea to make a 3D version of the good old *incredible machine* game. It should have all the features of the original game and include new features that can be achieved by today's graphics and physics hardware. Taking the environment one dimension further introduced many new problems. In order to position objects easily in $\mathbb{R}^3$, we needed to plan a new input GUI, so that it was intuitive enough for a player, who only has a common computer monitor and simple input devices like mice and keyboards. This implied that the camera viewing the scene could not be static, it had to be freely movable.
Another problem that arose was that, since the scene could be big enough or the physical simulations could make objects fly out of the view frustum, the camera could be looking at some point where nothing really happens, whereas the "real action" was happening elsewhere. We had two options to consider: to let the gamer move the camera by himself while the simulation ran, or to find a way to animate the camera, so that it would follow interesting objects or always see the most interesting place in the scene automatically.
The original game had two modes: building mode and simulation mode. The building mode was the interactive mode, where the gamer could place objects and configure them. The simulation mode was completely non-interactive; a player would start the simulation and watch it, until he or she decided to go back to the building mode. We wanted to respect that aspect of the original game, and amongst other benefits, we opted for the option to find a way to automatically animate the camera when the game was in simulation mode.
We had a camera motion planning problem which needed a realtime solution. In the past 30 years, there have been many publications about motion planning and possible solution approaches. Most of them focused on the robot motion planning problem, which requires the calculation of a movement and rotation plan for an object through a room without colliding with any obstacles. But it was only since the past five years when papers related to motion

[1] Realtime means that the calculations should not take longer than the desired display frame rate

planning for cameras started emerging. In the year 2003, a project in the Information Society Technologies (IST) programme from the European Community was founded to deal with *motion planning in virtual environments* (MOVIE)[2]. That project ran for three years and its members released 80 publications, some of them even dealt with camera motions. The literature from *MOVIE* gives a very rich impression on how complex and versatile the area of motion planning is, and shows that it gets very much attention from researchers worldwide.
Having studied the relevant literature regarding our problem, where we had to manipulate the position of the camera and the point where the camera should be looking at, considering the fact that a scene of an *incredible machine* can change very often and abruptly, it soon became clear that a new approach was necessary. A realtime solution for a motion planning problem in dynamically changing virtual environments has not yet been presented.
The camera in our problem had to react to the changes in the scene immediately. The movement of the camera should also be as smooth as possible, because it should be pleasant for the human eye; after all, it would be designed for when the audience puts all their concentration on what is happening on the screen, instead of interacting with the game. The camera should also move smoothly towards the desired place, and also turn around smoothly, so that the desired place is almost always seen. This bachelor's thesis talks about this motion planning problem and presents a realtime heuristic that satisfies all the before mentioned needs in many different application areas.

[2]http://www.give.nl/movie/, March 14th, 2008

Chapter 1

Motion Planning

1.1 The general motion planning problem

As mentioned briefly in the introduction, the general motion planning problem can be defined as the search for a way to move an object with a defined geometrical shape trough a set of objects, avoiding any collision with them, until it reaches a desired position and orientation. Following definitions can be made [CGC1998, page 294]:

Obstacles O**:** Objects in an environment defined by polygons or polyeders, with a specified position and orientation.

Actor Q**:** The object that will be moved

Source and destination positions $\vec{s}$ **and** $\vec{t}$**:** The path should begin and end at those points

Collision-free: $Q \cap o = \emptyset$, $\forall\, o \in O$

Path P**:** Set of linked line segments

Free path: A path whose line segments do not collide with any $o \in O$.

1.1.1 Robot motion planning

In robot motion planning, the actor Q is a robot R, which can be defined as a point, a circle, a line segment, a polygon or any geometrical shape. Robots can have different motion types, like translations and rotations. In $\mathbb{R}^2$, a robot that can only translate has two degrees of freedom, since it can be defined by its relative position (x,y) to some point of origin. The relative position is called the *configuration* of the robot, which can be defined as $R(x,y)$. If the robot can rotate as well, then it has three degrees of freedom and the configuration is defined as $R(x,y,\theta)$. Analogous, a translational robot in $\mathbb{R}^3$ has three degrees of freedom, and a robot that is free to translate and rotate in $\mathbb{R}^3$ has six degrees of freedom[CG22000, page 269].
The *configuration space* $C(R)$ is denoted as the space that can contain all combinations of the parameters of a robot. In $\mathbb{R}^2$, a robot that is free to translate and rotate has a three-dimensional configuration space, which is defined as $\mathbb{R}^2 \times [0 : 360)$, and can be represented as a cylinder.

Therefore, any configuration of that robot corresponds to a point on the surface of that cylinder. Furthermore, it is common to additionally define the following two terms:

- *forbidden configuration space*: This is the set $C_{\text{forb}}(R,O) \subset C(R)$ whose points corresponds to placements of the robot that intersect with one or more obstacles in O.
- *free configuration space*: This is the set $C_{\text{free}}(R,O) \subset C(R)$ whose points corresponds to placements of the robot that do not intersect with any obstacles in O.
- By definition, $C_{\text{free}}(R,O) \cap C_{\text{forb}}(R,O) = \emptyset$ and $C_{\text{free}}(R,O) \cup C_{\text{forb}}(R,O) = C(R)$

A collision-free path from one position and orientation of R to another would constitute a curve in the free configuration space.
How the free configuration space is defined depends on the geometry of the obstacles and of the robot itself. The robot motion planning problem consists on finding a path in $C_{\text{free}}(R,O)$.

1.1.2 Camera motion planning

In the case of camera motion planning, the actor Q is a camera, usually residing in an environment in $\mathbb{R}^3$. A camera can be considered as a pointlike robot with six degrees of freedom, since it can translate and rotate freely. So said, the camera motion planning problem is a specialization of the robot motion planning problem.
The configuration space $C(Q)$ has six dimensions, one for each degree of freedom of the camera. In the easiest of cases, the obstacles will only affect the three translatorial parameters of the camera. That means, no matter where the camera looks at or how it is rotated, the collision tests will all return the same result, since it is a pointlike actor. For scenes that don't change, the translational motion planning problem in $\mathbb{R}^3$ can be solved in $O(n^2 \log^3 n)$ time[CG22000, page 287].
But it all gets a lot more complex if the objects are allowed to change their positions during the camera motion, or if the viewing line of the camera affects the free configuration space. The complexity of the motion planning problem is exponential in the number of degrees of freedom of the actor, so finding a result can take a lot of time.

1.2 Previous work

1.2.1 Robot motion planning

There is a vast amount of work that has been done for robot motion planning. Not all of the previous work will be presented in this thesis, but at least it is important to know the basic approaches for solving the easiest cases of this problem, since the camera motion planning problem is based on them.

1.2.1.1 Road maps in free configuration space for pointlike actors

The most simple way to find a path for a pointlike actor in a static scene in $\mathbb{R}^2$, is to first construct a trapezoidal map [CG22000, page 122] of the free configuration space, and then

build a road map that goes through the center of each trapezoid and the middle of each vertical extension. The first step for the actor is to find the nearest path to the road map, and then from the roadmap to the endpoint. The resulting path is collision-free, since it consists of segments inside trapezoids and all trapezoids are in the free configuration space.

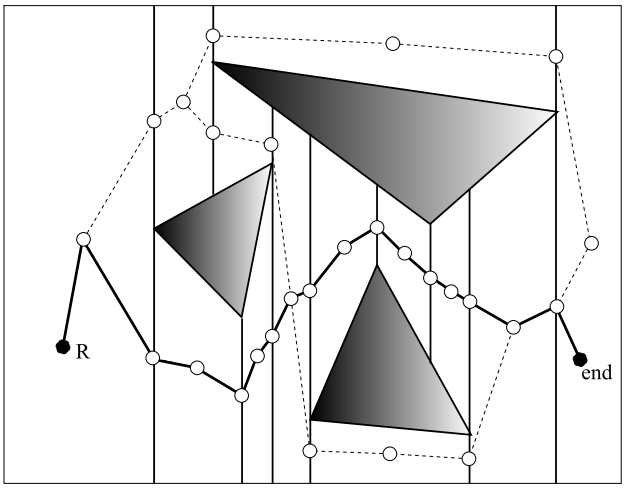

Figure 1.1: Trapezoidal map and road map in $C_{\text{free}}(R)$ for a pointlike actor R

1.2.1.2 Minkowski Sums

For an actor whose geometry consists of a convex polygon, and a set of convex obstacles, the same method like before can be used. For that, the free configuration space must be adapted. The motion planning problem for convex robots in scenes with convex obstacles can be reduced to the motion planning problem for pointlike robots, if the geometry of the obstacles is replaced by the minkowski sum of their geometry and the one of the robot[CG22000, page 275].

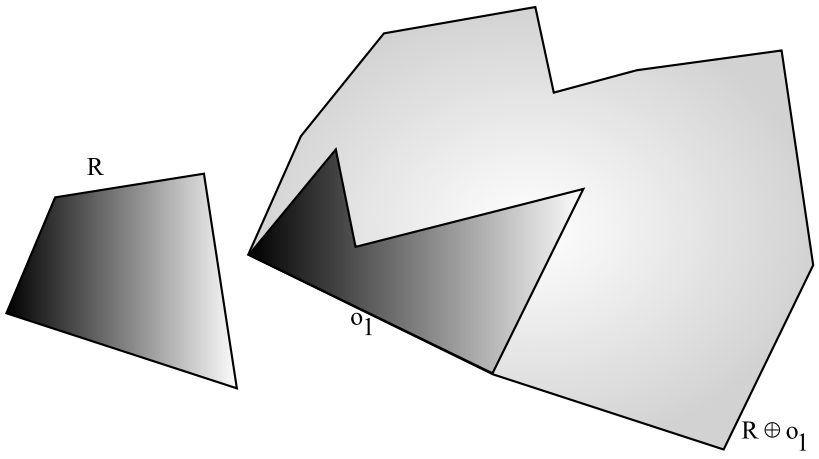

Figure 1.2: Minkowski sum $R \oplus o_1$ of a convex robot R and a polygonal object $o_1 \in O$

1.2.1.3 Probabilistic road map

As introduced in [PRM1996], there are also probabilistic methods to find paths. The *probabilistic road map* method creates and links random line segments in the free configuration space, and in this way builds up a roadmap. An actor wishing to find a path could use a shortest path search on that roadmap (which is a graph) to reach its goal.

1.2.2 Camera motion planning

The research on motion planning specifically for cameras began much later than for robots. In [MPC2003] one can see, that since the last decade, some research was done in finding a way to help the user in controlling the camera motion, by controlling the speed of the motion automatically or by giving the user clues on how to manipulate the motion to reach a specific goal position. These methods did not affect the camera motion.
Unrelated to motion planning, some research was done in determining ideal positions for fixed cameras, so that very interesting places could be seen from the corresponding positions. The user could then switch between the virtual cameras.
Later, some papers regarding tracking moving guides started appearing. Some of the solutions required by definition that the motion of the guide has to be known [POT1999, AGC2004], and some of them did not require that information [MHT2004].
In the area of computer games there has also been a lot of investigation on appropriate camera motions. For example in pursuit-evasion games, an actor has to find and follow an evasive target. Both deterministic and probabilistic algorithms have been proposed to solve this problem [MHT2004].
Another area of research was the image-based reactive behavior of a camera. In [IBV2000] a general method was proposed for controlling the camera motion in virtual environments. It deals with collision avoidance and guide tracking based only on information gathered from the image perceived by the camera. Although this problem resembles our problem described in the introduction, it works with image recognition instead of using information available from the virtual environment. In our problem, we can use the information provided by the virtual scene, more precisely the position and size of all objects, even if they are not visible.

1.3 Realtime camera motion planning in dynamic environments

As we can see, there has been a lot of work in many different areas of motion planning, and many solutions for specific motion planning problems have been presented. Unfortunately, almost all solutions solve specialized motion planning problems, and therefore cannot be easily adapted for our needs. Most of them assume that the environment is static and no obstacles will get in the way, or they deal with polygonal actors, and in many cases the running time is not fast enough for realtime applications. We had a different problem, and wanted a solution for a much more general problem, in order to be flexible.

1.3.1 Problem definition

Given a freely movable camera C, a set of freely movable objects with unknown motion O in a dynamic environment in $\mathbb{R}^3$, an interesting object $A \in O$, calculate a smooth collision-free (no intersections with any $O \setminus A$ allowed) camera motion from C to a point near A.

1.3.2 Problem solution

The given problem definition is very general and complex. Finding a solution for it cannot be computed deterministically. Therefore, a heuristical approach will be presented in section 2. Afterwards, the advantages, disadvantages and problems, the complexity of the problem, and possible modifications will be detailed in section 3. An example implementation of the presented heuristic can be found on the accompanying CD-ROM, more details can be found in appendix A.

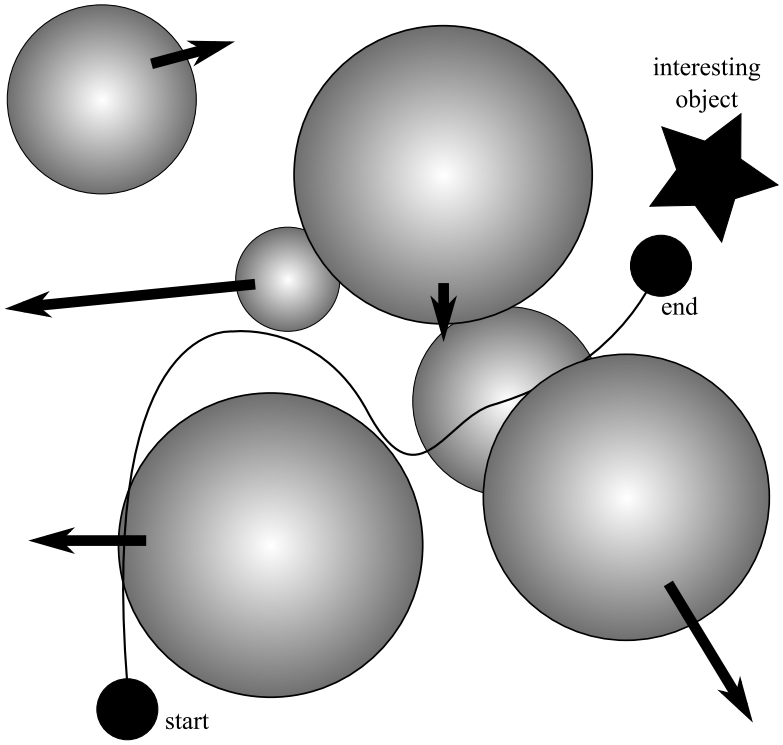

Figure 1.3: Example: A performed camera path from the original start position towards a location near the interesting object. In the scene there are six obstacles (symbolized as spheres), each with motions unknown to the camera (symbolized as black arrows).

Chapter 2

An approach to building a realtime heuristic

2.1 The overall idea

The orientation of a camera in $\mathbb{R}^3$ is defined by three vectors. One position vector $\vec{e}_t$ is the *eye* vector, which represents the position of the camera at the time t. Another position vector $\vec{c}_t$ is called the *center* vector, which symbolizes the point where the camera is looking at. Finally there's the *up* vector $\vec{u}_t$ which defines the orientation of the camera relative to its view direction $\vec{d}_t := \vec{c}_t - \vec{e}_t$. By definition, both vectors $\vec{d}_t$ and $\vec{u}_t$ should have a relative angle of 90 degrees to each other.

As we can see, we have three parameters which need to be taken into account when moving

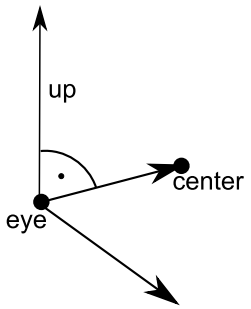

Figure 2.1: Orientation of a camera

the camera. The most interesting and difficult one is the position of the camera, which will be dealt with first. Then we'll discuss how to change the *center* and *up* vectors, so that the overall camera movement looks smooth and natural.

2.1.1 Defining the environment

The camera exists in a virtual environment, which can consist of several objects. The objects could be a representation of geometric solid objects, a complex mesh of adjacent triangles or any other form that defines geometry in $\mathbb{R}^3$. Each of those objects has an unique meaning in the environment, and therefore has a different value of interest. Objects with a low interest value are considered obstacles, because they represent the set of objects, which the camera does not want to see. The object with the highest interest value is the one the camera should be looking at continuously. Objects can have a varying interest value throughout the pass of time, depending on what each object represents, how it influences the environment or how it interacts with other objects around it. For example, a bouncing ball is most interesting when it is about to collide with the floor, and is less interesting when it is flying high. The camera must therefore deal with sudden changes of interest values, and head towards the currently most interesting object at all times. Let's define the position vector $\vec{p}_A$ as the position of the most interesting object $A \in O$, where O is the set of all objects in the virtual environment. All the other objects in the set $O \setminus A$ are then considered obstacles.

2.1.2 Preparations for the path finding algorithm

Let us plan and build a realtime path finding algorithm, which we will refer to from now on as a *discrete determined path finding* algorithm for the position of the camera. The most important criteria for the algorithm is that the path should never intersect any obstacle. We also want it to find a direct route towards $\vec{p}_A$ without taking too many unnecessary turns and without being too long. The algorithm must deal with collision detection, and since we want a realtime solution, determining if a polygonal path occludes a set of complex objects with different geometries would be very complicated and would take a lot of time to compute. Therefore, we will simplify our environment in a way so that calculating intersections can be done in realtime with a low performance cost. If we calculate the smallest enclosing sphere for each object using

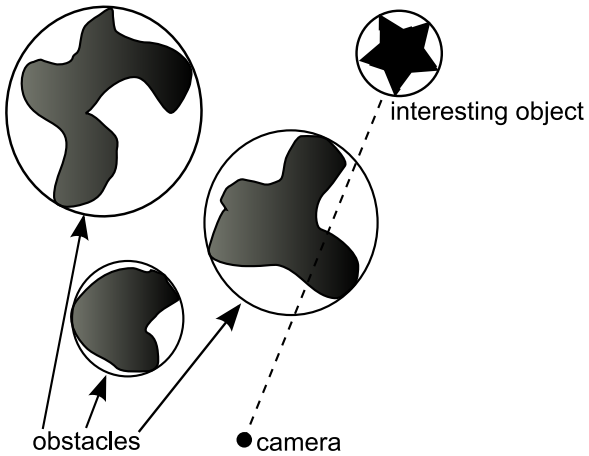

Figure 2.2: Virtual environment with objects and their smallest enclosing spheres

the algorithm of *Emo Welzl*[SES1991], which runs in $O(n)$ for n vertices, we can limit our collision detections to ray/sphere intersection tests. It is sufficient if we find a path that does not

collide with any of the enclosing spheres of the obstacles, since we don't require the smallest possible collision-free path.

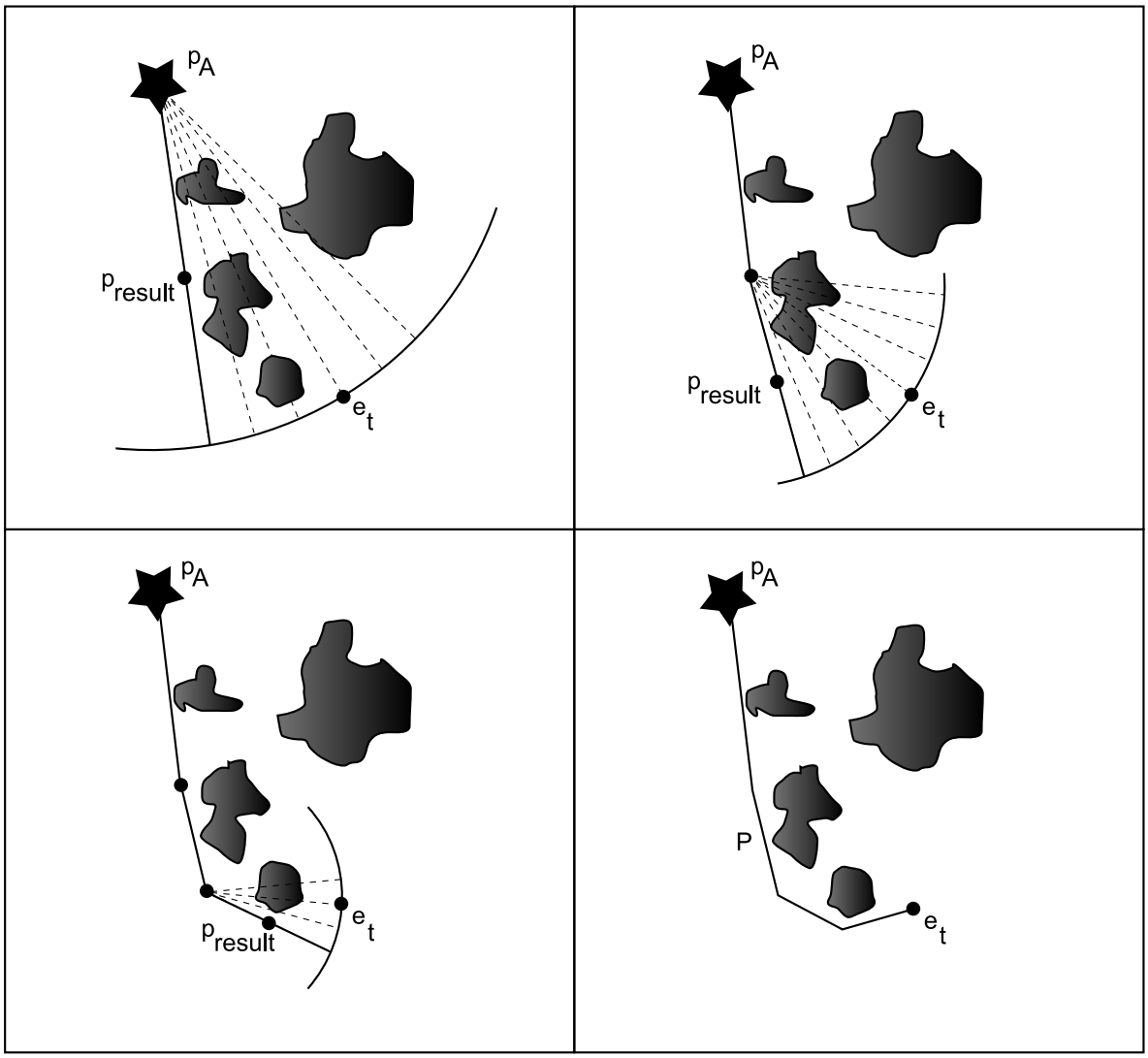

Figure 2.3: Example: path finding. The enclosing spheres are not shown for clarity.

2.2 Finding a path in $\mathbb{R}^2$

The following algorithm returns a set of points P that define a polygonal path between $\vec{e}_t$ and $\vec{p}_A$. The first step is simple: Get the line segment $\overline{\vec{p}_A\vec{e}_t}$ and check if it collides with any object in $O \setminus A$, by performing simple ray/sphere intersections, where the ray direction and length are defined by the line segment and the originating position is $\vec{p}_A$. If the line segment is collision-free, then it is a valid path segment. If not, we will need to test a new line segment.
For that, we will modify the line segment by discretely incrementing an angle $\alpha = 0$ and rotating the line segment around $\vec{p}_A$ by $\pm\alpha$. Every time they are tested for collisions, until a collision-free line segment is found. From that collision-free line segment, we take the point $\vec{p}_{\text{result}}$ that

is in the middle of it and add it to our result set P (which represents the path for the camera). Finally we execute the path finding algorithm again, this time from $\vec{p}_{\text{result}}$ to $\vec{e}_t$. Figure 2.3 shows an example. As we can see, the rays span a circle with $\vec{p}_A$ in the middle and $\vec{e}_t$ on its surface.

2.3 Finding a path in $\mathbb{R}^3$

The advantage of the before mentioned algorithm is that the cast rays have a determined tendency: they first point directly to the camera and then slowly stride away from the camera. In order to achieve the same behavior in $\mathbb{R}^3$, we can no longer rotate the ray with one angle to discretely approximate a sphere. But we can create a set of evenly distributed points K over the surface of a sphere with center $\vec{p}_A$ and radius $|\vec{e}_t - \vec{p}_A|$, which we can sort by their distance to $\vec{e}_t$, so that the rays cast from $\vec{p}_A$ all span that sphere and also slowly stride away from $\vec{e}_t$. We

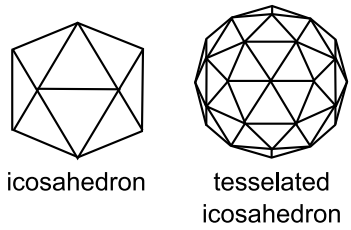

Figure 2.4: Icosahedron after one iteration of tessellation

can create K easily by taking the geometry of any platonic body and subdividing its triangles several times. The resulting subdivision will be a more precise approximation of a sphere. The icosahedron is a good choice for creating our set K, since very few subdivision iterations produce many more points than a tetrahedron or an octahedron would. Since K remains constant, it can be precomputed once before the motion planning starts. Later in the algorithm, we will work with a copy of K, whose points will be sorted by their distance to $\vec{e}_t$.

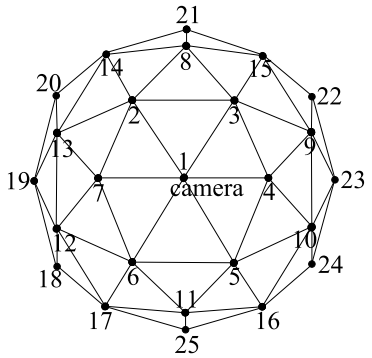

Figure 2.5: Example: order of points from K, sorted by their distance to the camera, which are then used as ray end points for the path finding algorithm

2.3.1 The algorithm so far

Algorithm 1 GetPath($\vec{p}_{\text{src}}$, $\vec{p}_{\text{dst}}$, P)

```
1:  Ray r⃗ ←Ray(p⃗_dst, p⃗_src)
2:  K ←SortedEnclosingSphere(p⃗_src, p⃗_dst)
3:  if Ray r⃗ occludes any obstacle then
4:     for all p⃗_k in K do
5:        Ray s⃗ ←Ray(p⃗_dst, p⃗_k)
6:        if Ray s⃗ does not occlude any obstacle then
7:           P' ← P
8:           p⃗_result ← p⃗_dst + s⃗/2
9:           P' ← P' + p⃗_result
10:          if GetPath(p⃗_src, p⃗_result, P') then
11:             P ← P'
12:             return true{We found a path!}
13:          end if
14:       end if
15:    end for
16:    return false{No path found}
17: else
18:    P ← P + p⃗_src
19:    return true
20: end if
```

In lines 7 to 14 we see, that for each ray $\vec{s}$ we cast from $\vec{p}_{\text{dst}}$ to $\vec{p}_k$ we let the algorithm execute recursively to see if a path is found. If that is not the case, we discard this ray and choose the next point from K. Otherwise, the returned set of points P' from the recursive call is a valid path which can be safely returned by copying it to the output set P.

2.4 Dealing with bad cases

The before mentioned algorithm is simple and can be executed in realtime, but it is not flawless. There are many cases, where no path can be found. The most simple case is when the interesting object is surrounded by obstacles, and the camera is outside the obstacle barrier. We can extend the algorithm in a way, so that it can find a path through such an obstacle ring. Let's introduce two new variables for our algorithm: the impatience i and the tolerance n. Whenever the algorithm does not find a path, it gets more impatient ($i \leftarrow i + 1$) and tries to find a path again. If the impatience reaches a certain tolerance level ($i \geq n$), the algorithm gives up and returns without a solution.
Let's modify our algorithm, so that when it would return no path, it instead reduces the radius of the sphere K by $\frac{i}{n}$ and tries to find a collision-free path to any of its points instead. If a path was found, then this workaround was successful, and we can continue finding our path towards the camera from that point onwards. If no path was found, then i is increased and the process is repeated.

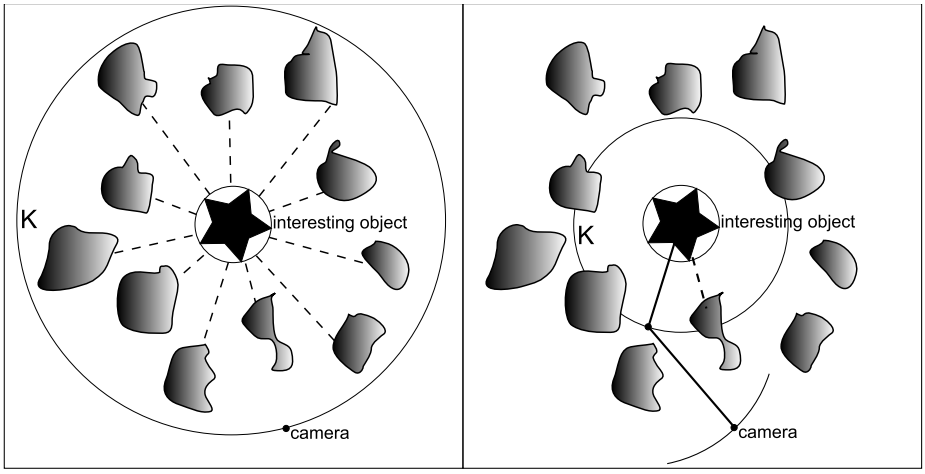

Figure 2.6: Left: The interesting object is surrounded. Right: A path was found with $i = 1$ and $n = 2$. Notice that the sphere was reduced to $\frac{1}{2}$ of its original radius, a collision-free path was found to a point on its surface, and from there a collision-free path was found to the camera.

There is no good or bad value for the tolerance n, because scenes can vary. If the icosahedron was tessellated very often (the amount of points in K is large enough), then the chance to find a path through such an obstacle ring is higher, and therefore no high fallback tolerance is needed. But having too many points in K makes the algorithm slower. Test results indicate that a tolerance level of $n = 3$ and tessellating the icosahedron three times ($|K| = 642$) is a good compromise.

2.5 Optimizations

2.5.1 Data structures

The runtime of this algorithm depends heavily on the runtime of the collision detection routines. Calculating the intersection between rays and spheres can be done very efficiently, as can be seen in [RTR2002, page 568ff]. But if there are many objects in a scene and each object is tested for collision with each ray, then the algorithm will perform slowly sometimes. One method to accelerate the collision detection can be to use data structures like a *Bounding Interval Hierarchy*[BIH2006] or a *kd-Tree*. With those data structures the collision detection tests for each ray limits itself to the number logarithmic to the amount of obstacles in the scene, instead of always testing all obstacles.
Of course, the costs of updating the data structures when the scene chances has to be taken into account. The scene should only change slightly and a local update in the data structure should be possible in order to accelerate the algorithm.

2.5.2 Reducing unimportant collision tests

Another optimization that can be done is to remove unimportant points from the tessellated icosahedron K. Since this algorithm tends to find smooth paths that point directly to the cam-

era, we can remove points from K that are farthest away from the camera. For that we can divide K in three different density regions. The region where the camera is in will get the full density of points, and the regions that are farther away will sequentially have their density halved. See figure 2.7 for an example in 2D. This will be an advantage if there are many objects between the camera and the interesting object. If there is a path from A to the camera that first leads away from the camera, it is not at all important in which way this path starts, since later we will only use the last segments of the path, those who reach the camera (see section 2.8).
Another possibility to remove uninteresting points from K, which is also implemented in the

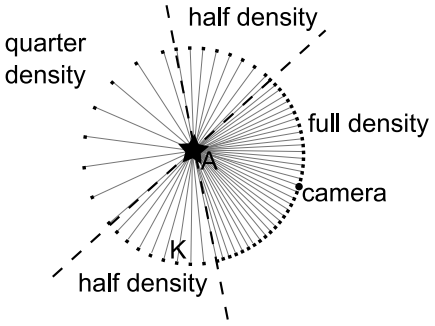

Figure 2.7: Removing unimportant points from K to improve overall performance

example program (see appendix A) can be to discard every two points from the last two thirds of K, and then discard the remaining last third of K. This alternative should only be used if backwards movements from $\vec{p}_A$ are not desired.
Removing points from K can improve the overall performance of the algorithm, but can drastically reduce the quality of paths and increase the chance of not finding any path at all. So if the scene is not too complex and the running time is acceptable, then this optimization step should be avoided.

2.5.3 More termination conditions

Since this algorithm calls itself to find a path, it should be guaranteed that no endless-cycles occur and that it does not try to find a path that would be too long. Paths with over 20 to 30 points will only be a result of a maze-like crawl through the objects and are not ideal for our purpose, so the *GetPath* method should return with no path if $|P| \geq p_{\max}$, where $p_{\max}$ is a predefined number, preferably not larger than 50.
To avoid endless cycles (and therefore endless recursions), we should always check with an epsilon comparison if $\vec{p}_{\text{result}}$ is already present in P. If that's the case, then the current path should no longer be calculated, since a cycle was built. In this case, the next point from K should be fetched and the loop should be continued, as if no path had been found.

2.6 The final path finding algorithm

Algorithm 2 GetPath($\vec{p}_{\text{src}}$, $\vec{p}_{\text{dst}}$, P, i)

if $i \geq n$ **then**
  return **false**{Sorry, this is more difficult than I thought! I give up.}
end if
if $|P| \geq p_{\max}$ **then**
  return **false**{No, this path is too long!}
end if
Ray $\vec{r} \leftarrow Ray(\vec{p}_{\text{dst}}, \vec{p}_{\text{src}})$
$\vec{p}_{\text{src}_{\text{new}}} \leftarrow \vec{p}_{\text{src}} + \vec{r} \cdot \frac{i}{n+1}$ {Reduce the radius of the sphere K}
$K \leftarrow$SortedEnclosingSphere($\vec{p}_{\text{dst}}$, $\vec{p}_{\text{src}_{\text{new}}}$)
if Ray $\vec{r}$ occludes any obstacle **then**
  for all $\vec{p}_k$ in K **do**
    Ray $\vec{s} \leftarrow$Ray($\vec{p}_{\text{dst}}$, $\vec{p}_k$)
    if Ray $\vec{s}$ does not occlude any obstacle **then**
      $P' \leftarrow P$
      $\vec{p}_{\text{result}} \leftarrow \vec{p}_{\text{dst}} + \frac{\vec{s}}{2}$
      if $\vec{p}_{\text{result}}$ is a duplicate of an existing point in P **then**
        continue
      end if
      $P' \leftarrow P' + \vec{p}_{\text{result}}$
      if GetPath($\vec{p}_{\text{src}}$, $\vec{p}_{\text{result}}$, P', i) **then**
        $P \leftarrow P'$
        return **true**{We found a path!}
      end if
    end if
  end for
  return GetPath($\vec{p}_{\text{src}}$, $\vec{p}_{\text{dst}}$, P, $i+1$) {No path found, try again with growing impatience}
else
  $P \leftarrow P + \vec{p}_{\text{src}_{\text{new}}}$
  return **true**
end if

2.7 The motion planner

Now that we have a path finding algorithm, we need to build a motion planner that uses it. Any software that needs to control the camera with this algorithm should perform some more operations before planning a path for the camera, and set the next position of the camera after the path has been planned.
First, we must check if the camera collides with any obstacle. This case can occur, if our algorithm failed for some reason (no path was found or too strict termination conditions were established), or if an obstacle moved with a high speed towards the camera. This case is very difficult to be prevented, and if it happens, we can't leave the camera inside the obstacle. That's

why we have to be able to reset the camera to a new position and new orientation. To give a more natural impression to the user, let us make him think that there are many different cameras in the scene, so when changing the position and orientation of our camera, the user should think that the view was switched from the current camera to another camera that is already in the scene.
Resetting the camera should contain the following operations:

- Get a new *eye* vector $\vec{e}_t$, which should be $k \cdot l_{\text{distance}}$ away from $\vec{p}_A$, $k \in [1, k_{\max}]$.
- Get a new *center* vector $\vec{c}_t$
- Calculate the corresponding *up* vector
- Set the camera speed $\vec{v}_t$ to zero

Another thing to consider is the possibility, that the interesting object can collide with an obstacle. If that's the case, there is a very high chance, that no path will ever be found, so we can spare ourselves the path planning method. In that case, we can either make the camera brake or move towards the interesting object along a linear path. Since we can never know where the camera will be when this happens nor how long it will stay there watching some uninteresting area, let us move the camera towards the interesting object. If it collides with an obstacle on its way, then it will be resetted as mentioned before. The following algorithm shows an example implementation for a motion planner:

Algorithm 3 PlanPath(Δt)

Point set $P \leftarrow \{\vec{p}_A\}$;
while $\vec{e}_t$ is inside any obstacle **do**
  $\vec{v} \leftarrow$ Random vector with length l_{distance}
  $\vec{e}_t \leftarrow \vec{p}_A + \vec{v}$
  $\vec{c}_t \leftarrow Randomvector$
  $\vec{d} \leftarrow \vec{c}_t - \vec{e}_t$
  $\vec{u}_t \leftarrow \left(\vec{d} \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \right) \times \vec{d}$
end while
if $\vec{p}_A$ is not inside any obstacle **then**
  if GetPath($\vec{e}_t$, $\vec{p}_A$, P, 0) returns a valid path **then**
    MoveEye(Δt, $P_{|P|-1}$, $P_{|P|-2}$)
    return
  end if
end if
{No path was found, move towards the interesting object}
$d \leftarrow \frac{\vec{p}_A - \vec{e}_t}{|\vec{p}_A - \vec{e}_t|}$
MoveEye(Δt, $\vec{p}_A$, $\vec{p}_A + d$)

2.8 Defining the motion of the camera

2.8.1 The position of the camera

Having calculated a movement path, we now have enough information to change the position of our camera. To calculate the next position after the time interval Δt, we only require the position of the camera at the moment t, the last two points of the calculated path $\vec{p}_1$ and $\vec{p}_2$, and the position of our interesting object $\vec{p}_A$.
The main question that arises at this moment is: why have we calculated an entire path, if we are only going to use the first two points from it? The main reason for calculating an entire path is to be sure, that the camera will definitely reach the desired object if it follows that path. There's also a high chance, that a very similar path or perhaps an even better path is calculated in the near future. Having the assurance that those first two points belong to a correct path at all times is enough to decide the next motion direction, and we can safely adjust the camera motion towards the beginning of the calculated path.

Our goal is that the camera performs a smooth movement in space without making abrupt travel direction changes. For that, we need to introduce a few extra properties to our camera:

- The measuring unit u that corresponds to one OpenGL spatial unit.
- The current speed s_t of the camera at the moment t, in $\frac{u}{s}$.
- A constant value $s_{\max}$ that represents the maximum speed that our camera can travel at, also in $\frac{u}{s}$.
- A constant value a that represents the acceleration of our camera, in $\frac{u}{s^2}$.
- A user defined value l_{distance} that signals the camera to keep a certain distance from our interesting object, measured in u.

The camera should know when to accelerate and to slow down. For the sake of simplicity, let's assume that the camera wants to accelerate up to its maximum speed if the distance between the camera and the interesting object is large enough. The camera should also start slowing down for a period of time t_{break} to achieve a full stop exactly when it reaches a distance l_{distance} from the interesting object. The period of time t_{break} and the needed break distance l_{break} can be calculated as follows:

$$t_{\text{break}} = \frac{s_t}{a} \tag{2.1}$$

$$l_{\text{break}} = \frac{a}{2} \cdot t_{\text{break}}^2 + s_t \cdot t_{\text{break}} \tag{2.2}$$

$$s_{t+\Delta t} = \max\left(0, \min\left(s_t \pm a \cdot \Delta t, s_{\max}\right)\right) \tag{2.3}$$

With this information, we should now be able to accelerate or slow down the camera. To determine whether to accelerate or slow down, we calculate the difference between $\vec{p}_A - \vec{e}_t$ and the sum of l_{break} and l_{distance}. If the difference is positive, then we accelerate by adding the timed acceleration to the camera speed as in equation 2.3. Otherwise we slow down by subtracting the timed acceleration. The speed should not exceed $s_{\max}$ nor become negative, hence the minimum and maximum functions in equation 2.3.

2.8.1.1 Interpolation with cubic Hermite splines

We need to determine the direction of movement for the camera to the moment $t + \Delta t$. We know the position of the camera e_t and the first point in the polygonal path p_1. A naive option would be to move the camera directly towards p_1 along a straight line. But since we want to achieve a smooth camera motion, and because the path P might not be the same from frame to frame, the resulting animation would have many ugly "jumps" and would not be smooth. We need to take into account the current motion direction $\vec{v}_t$ of the camera when calculating the next direction $\vec{v}_{t+\Delta t}$. Therefore we must find a way to interpolate between the current direction $\vec{v}_t$ and the direction $\vec{p}_2 - \vec{p}_1$ using the camera speed and Δt as interpolation parameters. For this, we need to use a spline function.

2.8.1.2 A quick overview of splines

The term *spline* stands for an equation or a set of equations describing a curve. There are many types of curves, the most simple one being a Bézier curve. Given a set of control points that define a polygonal path and using the *de Casteljau algorithm*, we could determine a point anywhere on a Bézier curve. The main disadvantage of Bézier curves is that their use is not always intuitive, because to determine points on a cubic Bézier curve, we would need to calculate four control points. The segment consisting of the first two and the last two control points would be the curve tangents at the beginning and the end of the curve.

2.8.1.3 Cubic Hermite interpolation

Unlike Bézier curves, the cubic Hermite curve is defined by a starting point h_s, an end point h_e and two tangents m_s and m_e. The main advantage of the Hermite spline is, that we must not calculate the control points if we already know the curve tangents and the start and end points. Since we can use $\vec{e}_t$ as h_s, $\vec{p}_1$ as h_e, $\vec{v}_t$ as m_s and $\vec{p}_2 - \vec{p}_1$ as m_e (see figure 2.8), the Hermite spline suits our needs much better than the Bézier curve. The interpolation function $h(\tau)$ with a scalar $\tau \in [0,1]$ is defined as follows:

$$h(\tau) = \left(2\tau^3 - 3\tau^2 + 1\right) \cdot h_s + \left(\tau^3 + 2\tau^2 + \tau\right) \cdot m_s + \left(\tau^3 + \tau^2\right) \cdot m_e + \left(-2\tau^3 + 3\tau^2\right) \cdot h_e \quad (2.4)$$

The Hermite curve segment defined by $h(\tau)$ is a cubic interpolant, since τ^3 is the highest exponent in equation 2.4, and therefore it has a continuity of C^2. A curve with higher continuity has a reputation of looking more natural when used in motion animations.

2.8.1.4 Using cubic Hermite interpolation to determine the next position

The only remaining value that needs to be calculated is the interpolation value τ for $h(\tau)$. We said earlier that we want to use $s_{t+\Delta t}$ and Δt as interpolation parameters, and we know that $\tau \in [0,1]$. Since we can never be sure how big Δt will be, we can never rule out the fact that τ can occasionally be greater than 1 when the frame rate is very low and the camera speed is very high, therefore we will scale down the interpolation parameter by half the length of the control polygon. In fact, we should actually scale down τ by the length of the Hermite curve

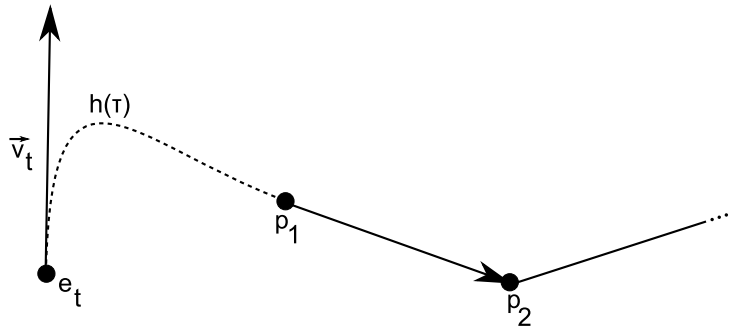

Figure 2.8: Interpolating with a cubic Hermite spline

segment, but since there is no closed equation for the length of a cubic Hermite spline, using half the length of the polygon $(\vec{e}_t, \vec{e}_t + \vec{v}_t, \vec{p}_1)$ will suffice.

$$\tau := \frac{s_{t+\Delta t} \cdot \Delta t}{\left(\frac{|\vec{v}_t| + |\vec{p}_1 - (\vec{e}_t + \vec{v}_t)|}{2}\right)} \tag{2.5}$$

Calculating the point on the Hermite curve with the parameter τ using equation 2.5 will give us a nearly optimal point where the camera should be headed to at the moment $t + \Delta t$, giving us the camera direction $\vec{v}_{t+\Delta t} = h(\tau) - \vec{e}_t$.

2.8.1.5 The algorithm

Putting it all together, we can now successfully perform the camera motion in time intervals Δt, as seen in the following algorithm:

Algorithm 4 MoveEye(Δt, $\vec{p}_1$, $\vec{p}_2$)

$s_t \leftarrow |\vec{v}_t|$
$\vec{d}_{\text{next}} \leftarrow \vec{p}_2 - \vec{p}_1$
$\vec{d}_{\text{final}} \leftarrow \vec{p}_n - \vec{e}_t$
$t_{\text{break}} \leftarrow \frac{s}{a}$
$l_{\text{break}} \leftarrow \frac{a}{2} \cdot t_{\text{break}}^2 + s \cdot t_{\text{break}}$
if $\left|\vec{d}_{\text{final}}\right| > l_{\text{break}} + l_{\text{distance}}$ **then**
  $s_{t+\Delta t} \leftarrow s_t + a \cdot \Delta t$
  if $s_{t+\Delta t} > s_{\max}$ **then**
    $s_{t+\Delta t} \leftarrow s_{\max}$
  end if
else
  $s_{t+\Delta t} \leftarrow s_t - a \cdot \Delta t$
  if $s_{t+\Delta t} < 0$ **then**
    $s_{t+\Delta t} \leftarrow 0$
  end if
end if
$\vec{v}_t \leftarrow \frac{\vec{v}_t}{|\vec{v}_t|} \cdot s_{t+\Delta t}$
$\vec{p}_{\text{next}} \leftarrow \text{CubicHermiteSpline}\left(\vec{e}_t, \vec{v}_t, \vec{p}_1, \vec{d}_{\text{next}}, \frac{s_{t+\Delta t} \cdot \Delta t}{\left(\frac{|\vec{v}_t| + |\vec{p}_1 - (\vec{e}_t + \vec{v}_t)|}{2}\right)}\right)$
$\vec{v}_{t+\Delta t} \leftarrow \frac{\vec{p}_{\text{next}} - \vec{e}_t}{|\vec{p}_{\text{next}} - \vec{e}_t|} \cdot s_{t+\Delta t} \cdot \Delta t$
$\vec{e}_{t+\Delta t} \leftarrow \vec{e}_t + \vec{v}_{t+\Delta t}$

2.8.2 The point where the camera looks at

The animation of the *center* vector $\vec{c}_t$ of the camera is not as complicated as the animation of the *eye* vector $\vec{e}_t$. We only need to know the point where the camera was looking at in the last frame, and the point where the camera should be looking at in the near future. We will also need the speed s_c, maximum speed $s_{\max}$ and acceleration a_c for the animation of the center point. The calculation of the speed should be analogous to the calculation in the previous section, so it will be skipped here.
Let's observe the current center $\vec{c}_t$, the desired center $\vec{c}_{\text{desired}}$ and the position of the camera $\vec{e}_t$, and create two corresponding direction vectors $\vec{d}_1$ and $\vec{d}_2$ originating from the point $\vec{e}_t$. In order for our calculation to work smoothly, both direction vectors should have an equal length, so we'll adjust the length of $\vec{d}_1$ vector to match the length of $\vec{d}_2$.
We define the point $\vec{c}_{\text{new}} \leftarrow \vec{e}_t + \vec{d}_1$, and the motion direction $\vec{d} \leftarrow \vec{d}_2 - \vec{d}_1$, as seen in figure 2.9. To calculate the new center point, we only need to add to $\vec{c}_{\text{new}}$ the direction $\vec{d}$ weightened by the speed and Δt: $\vec{c}_{t+\Delta t} \leftarrow \vec{c}_{\text{new}} + \vec{d} \cdot s_c \cdot \Delta t$.
This way, the camera will always move smoothly towards the desired center $\vec{c}_{\text{desired}}$. The trick behind the smooth motion lies within the acceleration and speed of the animation. The following algorithm shows an example implementation.

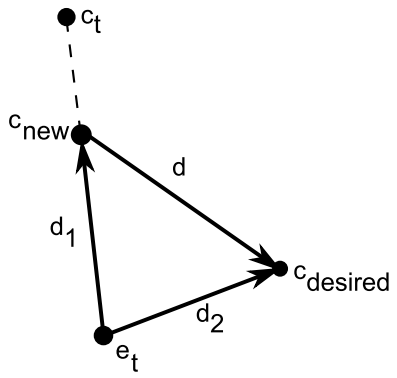

Figure 2.9: Calculating the center point

Algorithm 5 MoveCenter(Δt)

if $\vec{c}_t = \vec{c}_{\text{desired}}$ **then**
  return
end if
$\vec{d}_1 \leftarrow \vec{c}_t - \vec{e}_t$
$\vec{d}_2 \leftarrow \vec{c}_{\text{desired}} - \vec{e}_t$
$\vec{d}_1 \leftarrow \vec{d}_1 \cdot \frac{|\vec{d}_2|}{|\vec{d}_1|}$
$\vec{d} \leftarrow \vec{d}_2 - \vec{d}_1$
$t_{\text{break}} \leftarrow \frac{s}{a}$
$l_{\text{break}} \leftarrow \frac{a}{2} \cdot t_{\text{break}}^2 + s \cdot t_{\text{break}}$
if $|\vec{d}| > l_{\text{break}}$ **then**
  $s \leftarrow s + (\Delta t \cdot a)$
else
  $s \leftarrow s - (\Delta t \cdot a)$
end if
if $s > s_{\max}$ **then**
  $s \leftarrow s_{\max}$
end if
if $s < 0$ **then**
  $s \leftarrow 0$
end if
$\vec{c}_{\text{new}} \leftarrow \vec{e}_t + \vec{d}_1$
$\vec{c} \leftarrow \vec{c}_{\text{new}} + \vec{d} \cdot s \cdot \Delta t$

2.8.3 The up vector of the camera

The last thing remaining to do is to update the *up* vector of the camera, since the *center* and *eye* vectors of the camera are always changing. A default value for the up vector should be parallel to the Y-Axis, since that's the normal *up* direction for a human being. In order to maintain that default value and to prevent skews in the projection of the scene, these two constraints must be preserved:

- The angle between $\vec{d} \leftarrow \vec{c}_t - \vec{e}_t$ and $\vec{u}_t$ must be 90°, see figure 2.1.
- $\vec{u}_t$ should be in the same plane spanned by $\begin{pmatrix}0\\1\\0\end{pmatrix}$ and $\vec{d}$.

The first constraint is easy to solve. First, a *right* vector is calculated with the cross product between $\vec{d}$ and $\vec{u}_t$, and then the new $\vec{u}_{t+\Delta t}$ is calculated with the cross product between $\vec{r}$ and $\vec{d}$. The latter constraint requires an adjustment for $\vec{u}_{t+\Delta t}$. Let $\vec{l}$ be the direction from $\vec{u}_{t+\Delta t}$ to $\begin{pmatrix}0\\1\\0\end{pmatrix}$. Let us then add $\vec{l}$ to $\vec{u}_{t+\Delta t}$, weightened by Δt and a specified constant speed factor s_u. Afterwards, $\vec{u}_{t+\Delta t}$ must be normalized, see figure 2.10.
This way, an automatic "leveling" of the *up* vector will occur, ensuring that the horizon stays horizontal, and therefore undoing camera rolls[1] produced by the animation of the *center* and *eye* vectors of the camera.

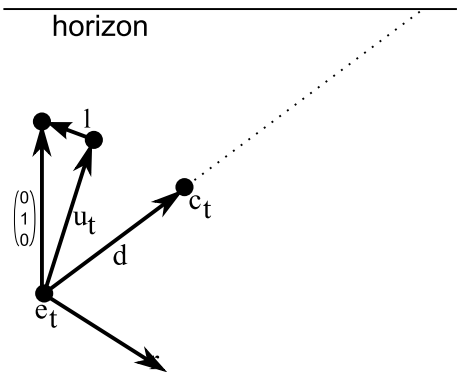

Figure 2.10: Automatic leveling of the *up* vector

Algorithm 6 MoveUp(Δt)

1: $\vec{d} \leftarrow \vec{c}_t - \vec{e}_t$
2: $\vec{r} \leftarrow \vec{d} \times \vec{u}_t$
3: $\vec{u}_{t+\Delta t} \leftarrow \frac{\vec{r}\times\vec{d}}{|\vec{r}\times\vec{d}|}$
4: $\vec{l} \leftarrow s_u \cdot \Delta t \cdot \left(\begin{pmatrix}0\\1\\0\end{pmatrix} - \vec{u}_{t+\Delta t}\right)$
5: $\vec{u}_{t+\Delta t} \leftarrow \frac{\vec{u}_{t+\Delta t}+\vec{l}}{|\vec{u}_{t+\Delta t}+\vec{l}|}$

[1] A camera roll is a rotation along the viewing line $\vec{d}$

Chapter 3

Notes and Comments

3.1 Evaluation

In this thesis a new realtime heuristic for solving a motion planning problem for cameras in changing environments was introduced. This algorithm should be considered as a first generalized approach to solving many other specialized motion planning problems.

3.1.1 Advantages

The main advantage of the *discrete determined path finding* algorithm is that it offers a solution for a general motion planning problem, and is not bound to restrictions assumed by other solutions. This algorithm:

- Does not need a bounding volume of the scene. Some motion planning algorithms depend on knowing the outer boundaries of the scene and calculating the intersection points of line segments with those outer boundaries. This algorithm only needs the position and the radius of the smallest enclosing sphere of each object in the scene, as well as the position and orientation of the camera.
- Does not need any preprocessing. The only thing that has to be preprocessed is the tessellation of an icosahedron, which can as well be calculated and stored into a file, so that the data is then read instead of being calculated every time the program is run. Other motion planning solutions, like the *probabilistic road map*[PRM1996] algorithm, depend on calculating paths once on a static scene and then using that information.
- Can cope with dynamically changing scenes. A vast majority of literature regarding motion planning [CG22000, PAL2006, MPC2003, AGC2004, MHT2004] always take for granted that a scene is static and that nothing changes while the actor (a robot or a camera) moves.
- Does not need any knowledge of the future. It only considers the position of the obstacles at the moment t, and only with this information it decides for the camera its next position $c_{t+\Delta t}$.

- Does not need any knowledge of the current motion or speed of the objects in the virtual environment. Some algorithms, like the one presented in [MPC2003], assume that the motion of the interesting object is known. This algorithm only reacts to changes instead of knowing or guessing future moves. This is called a *reactive behavior*.
- Defines a smooth motion for a camera by using cubic Hermite splines and natural speed variations by constant acceleration and de-acceleration values, and therefore produces a C^2 continuous movement.
- Can be used in realtime applications, which allows for scenes to change dynamically.
- Offers a solution for a problem that is at least NP-hard, see subsection 3.1.1.1.

3.1.1.1 Complexity

Motion planning problems in $\mathbb{R}^3$ are very difficult. Their complexity has been studied extensively over the past 30 years, some example results of that research are listed bellow:

- The general motion planning problem (also known as the *piano mover's problem*) is PSPACE-complete [PAL2006, page 300].
- The 3D shortest path problem among static obstacles is NP-hard [3MP2001, page 1].
- The Warehouseman's problem, which is a motion planning problem for a set of translating rectangles in $\mathbb{R}^2$ is PSPACE-hard [PAL2006, page 301].
- Even the *2D* asteroid avoidance problem (2D-AAP) is NP-hard [CRM1988, page 9].

2D-AAP consists of finding a collision-free path for a pointlike robot (with bounded velocity magnitude) through a set of convex polygons with linear translations of constant speed. A 3D version of this problem could be defined as follows, if a convex polyhedron in 2D-AAP would represent a convex solid body in 3D-AAP:

Problem: Given in $\mathbb{R}^3$ a freely moveable pointlike robot R with bounded velocity magnitude, a set of convex solid bodies O that can translate linearly with constant speed and two points p_{start} and p_{end}, calculate a collision-free path for R from p_{start} to p_{end}.

Obviously, 3D-AAP is a specialization of our problem, because our problem allows for arbitrary objects moving with unknown direction and speed, instead of requiring convex objects with linear translations and constant motion speed.
It can be proved that the 3D asteroid avoidance problem cannot be easier than the 2D version, so its complexity is definitely at least NP-hard. And since our problem defined in subsection 1.3.1 is a generalization of 3D-AAP, its complexity is also at least NP-hard. To solve 3D-AAP, it is only natural to consider an approximative algorithm, especially in realtime applications. The fact, that our algorithm is able to find a good solution for 3D-AAP (there are efficient ray/-convex polyhedron intersection tests like the one presented by Haines[1]) and for our problem defined in subsection 1.3.1, fully justifies its nature as a heuristic.

[1] E. Haines: *Fast Ray-Convex Polyhedron Intersection*, Graphics Gems II, pages 247-250

3.1.2 Disadvantages

This algorithm is *only* a heuristic, therefore it cannot always find an ideal solution. Nevertheless, test results show that the camera moves well most of the time, and the fallback methods when collisions occur are also agreeable, see Appendix A for a detailed description of the test cases and the algorithm performance.
A disadvantage of this algorithm is that it always returns the first path that it finds, and not the best or ideal path. A human being would have found a more direct path through obstacles, when this algorithm would have suggested a complete detour around the obstacles. Furthermore, on a scene with many obstacles, the paths calculated in every frame may be very different from each other. If the motion speed and motion direction of the camera would not be considered into the calculations, the camera movement would be very shaky.
Since the time difference between each frame, Δt, also plays a big role in the algorithm, the resulting paths can not be reproduceable. No program can guarantee that a sequence of time intervals will be the same each time the program runs, unless they are strictly defined (Δt is constant).
When the scene consists of only solid convex obstacles, this algorithm performs nicely. But if the scene consists of walls, floors and many box-like or complex polygonal geometric objects, then all those obstacles have to be approximated by spheres. One way to do this approximation is to build a bounding volume hierarchy, or more specifically a sphere tree.
Despite the before mentioned disadvantages, there are always ways to reduce their occurrences or to bypass them, so that the algorithm can be useful.

3.1.3 Problems

As with many heuristics, this algorithm has its worst cases and other problems. Despite sometimes not finding a path, its running time is unpredictable and depends on the complexity of the scene. It will return only if a path was found or if its termination conditions are reached and no path was found. In the last case (which is the worst one) the calculations may have been in vane and taken very long (around 200-300ms according to test results), and then return no path. And even when a path was found, it might have been a very complex path which could have been found shortly before the termination conditions were reached. But considering that those cases are very rare, the estimated running time is well enough inside the required boundaries of a realtime graphics application.
Another problem is that the camera can collide with obstacles. When the camera or an object has a motion with a high speed, then a collision is highly probable if the acceleration term of the camera is not high enough to avoid the obstacle. On the other hand, if an obstacle flies towards the camera from behind while the camera sees the interesting object and has no obstacles in sight, then the camera has no way of noticing that object and therefore can't plan an escape route to prevent a collision. In order to avoid such collisions, some other methods of collision avoidance have to be implemented in addition to this algorithm, see section 3.2.

3.2 Future work

The heuristic presented in this bachelor's thesis can be used as it was presented and implemented on the accompanying CD-ROM, but there is still much room for improvements. Some examples of the aspects that could be elaborated are:

- *Make obstacles repulsive*. As mentioned in subsection 3.1.3, the camera can easily collide with obstacles. If we give the obstacles a repulsive field that is only effective to the camera within a certain distance, the camera should no longer be able to collide with the obstacles. This method is an extract from the *potential field method*, which defines a potential field on the configuration space by making the interesting object attract the camera and making the obstacles repel it. The camera then moves in the direction dictated by the potential field[CG22000, page 288]. The disadvantage of the *potential field method* is that the camera can get stuck in local minima, where no path exists, that's why the *discrete determined path finding* algorithm could constitute a good technique to avoid such minima.
- *Allow negative speed values*. Now the camera can only move forwards in order to come nearer to the interesting object. But if the object also has a motion of its own, it can happen that it will be closer to the camera than desired. Therefore negative speed values should be allowed in order to move the camera backwards if necessary.
  Alternatively, the direction where the camera should move to can be adapted instead of allowing negative speed values. This can be again achieved by providing the interesting object with a repulsive field, so that when the camera gets too close, it gets a thrust away from it.

Many more extension possibilities are also thinkable, specially if this algorithm is used for more specialized problems (see section 3.4).

3.3 Possible modifications

- *Path finding from the camera towards the interesting object*: Instead of finding a path from the interesting object towards the camera, the algorithm can be easily changed, so that it finds a path from the camera towards the interesting object. This method would probably return similar paths, and would not necessarily decrease the rate of failure (not finding a path).
- *Dual path finding*: A combination of the current algorithm and the modified algorithm as described previously is also possible. Start two instances of the path finding algorithm, one from the interesting object and one from the camera, and let them try to find each other. Yet the details on how both paths should find each other must still be studied and elaborated. This modification could drastically improve the quality of returned paths, and may also decrease the rate of failure. But as two instances of the algorithm are run, it also increases the running time.
- *Relative position constraints*: The current algorithm makes sure that the camera tries to maintain a certain distance from the interesting object, but no constraints on the desired relative camera position towards the interesting object are given. It is thinkable that some

applications would need that the camera always looks towards the interesting object in one specified direction. This could be easily achieved by calculating the desired final camera position and then executing the path finding routine from there towards the camera. It is also thinkable, that the camera should keep the same height (usually the Y coordinate) as the interesting object. For that, the corresponding speed vector $\vec{v}_t$ can be altered each frame by a small addend, so that the camera always strives to maintain the desired height.

As we can see, there is much room for customizations and modifications to match many application areas.

3.4 Other application areas

There are many different scenarios where motion planning problems need to be solved. But some scenarios don't necessarily match the problem described in this thesis. Nevertheless, this algorithm can be taken as a base and then be adapted and transformed for different application areas:

3.4.1 Non-realtime motion planning

Let's assume that there is an application that needs to calculate a camera motion for dynamically changing scenes, but does not need a realtime solution. For example, the positions and orientations of the camera can be calculated with enough time during each frame, and then saved for future use. If that is the case, the algorithm can be tweaked to increase the quality of returned paths.

- First of all, don't let the algorithm return the first path it finds. Instead of returning from the function when a path is found, save the path in a set of possible paths and evaluate all paths after having filled that set, then choose the best path from it (for example the shortest path in length or speed, or the path with least narrow angles) and return it.

- The termination conditions can be softened. If the running time is not a concern, then a higher tolerance value n, a higher tessellation of the icosahedron K and a higher maximum path length $p_{\max}$ will definitely lead to better results.

- Since this process is ideal as a preprocessing step, the collected camera positions can be used to build a polygonal path and then it can be softened through interpolating cubic splines.

Such a modification has the advantage, that the calculated path is potentially an optimal smallest collision-free path, and it is reproduceable when it is saved and reused.

3.4.2 Realtime robot motion planning

Another thinkable scenario could be a robot that needs to plan its movement in realtime, while the surrounding environment is in constant change. With the help of latest image analysis and pattern matching methods, a robot equipped with one or more cameras or sensors can have the possibility to scan the nearby environment and calculate an approximate three dimensional model. If the obstacles in the environment are marked adequately, the robot can recognize them as obstacles and know their approximate position and size. In that case, with the information where the robot should be headed to, the presented algorithm can be used to quickly determine the next direction where the robot should be headed to. This could be considered a replacement for the method proposed in [IBV2000].

3.5 Conclusion

Having listed these other application possibilities, the presented *discrete determined path finding* algorithm is surely a good beginning for a wide range of motion planning problems, despite the smaller problems that can arise occasionally. An implementation of the algorithm along with several test cases are included on the enclosed CD-ROM. More details can be found in Appendix A.

Appendix A

Contents of the CD-ROM

A.1 Implementation

The implementation of the algorithm discussed in this thesis is contained on the CD-ROM accompanying this thesis. The software was developed using Microsoft Visual C++ 2005 and uses the following libraries:

- *freeglut*: "freeglut is a completely OpenSourced alternative to the OpenGL Utility Toolkit (GLUT) library"[1]. It is used to create and manage the window with the OpenGL rendering context.
- *glew*: The OpenGL Extension Wrangler is a library that allows for easy access to the OpenGL extensions, like for example the Shader functions used in this software.
- *libjpeg*: A free platform-independent library for loading images from JPEG files.

The corresponding licenses for those libraries can be found in Appendix B. Some algorithms were taken from [RTR2002, CG22000], detailed information can be found in the commented source code. Both the source code and compiled binaries for Windows of this software are included.

A.2 Directories

A.2.1 bin

The compiled binary files can be found in this directory.

- *MotionPlanningSSE3.exe* A binary executable file optimized with the SSE3 instruction set for increased performance. Runs only on newer CPUs.
- *MotionPlanning.exe* The generic binary without using any SIMD extensions. Should be compatible with most x86 CPUs.

[1] http://freeglut.sourceforge.net/, March 14th, 2008

A.2.2 src

The whole source code of the software and its required libraries can be found in this directory.

- *MotionPlanning.sln* The Microsoft Visual C++ 2005 solution file.

A.2.2.1 src/freeglut

The source code of the library *freeglut*.

A.2.2.2 src/glew

The source code of the library *glew*.

A.2.2.3 src/libjpeg

The source code of the library *libjpeg*.

A.2.2.4 src/MotionPlanning

The source code of an implementation of the presented *discrete determined path finding* algorithm.

A.3 Details of the implementation

A.3.1 User interface

The library *freeglut* is used to create a window and handle mouse messages. When the program is run, a console window will open with a greeting message and then another blank window will open. On that window, the user can use the right mouse button to open a context menu and select one scene or exit the program.

A.3.2 Scene Graph

The program uses a scene graph to store and visualize the objects in the virtual environment. Obstacles have a red or brown color and are represented as edgy convex objects. The interesting object is green and blue and is represented as a sphere. A huge static sphere mapped with a landscape texture encloses the scene and serves as an orientation aid for the user watching through the camera; the camera movements can be perceived much better if there's a visible horizon in the scene.

A.3.3 Animations

The objects in the virtual environment have the ability to move. A central animation manager in the program allows for rotations, linear and sine-like translations for all objects.

A.4 Test cases

In this implementation, four test cases have been prepared and can be seen by choosing them with the context menu. In all scenes, the objects with a high interest factor (shown as green spheres) have a radius of $1u$.

A.4.1 Scene 1

Figure A.1: A screenshot from Scene 1

- *Camera eye acceleration*: $75\frac{u}{s^2}$
- *Camera center acceleration*: $5\frac{u}{s^2}$
- *Camera maximum speed*: $300\frac{u}{s}$
- *Desired distance from A*: $15u$
- *Amount of obstacles*: 100
- *Obstacle radius*: $1u$

This scene demonstrates the ability of the algorithm to follow a moving interesting object A, while it flies through obstacles. It can be seen how the camera reacts to the obstacles getting in the way between A and the camera.

A.4.2 Scene 2

Figure A.2: A screenshot from Scene 2

- *Camera eye acceleration*: $15\frac{u}{s^2}$
- *Camera center acceleration*: $5\frac{u}{s^2}$
- *Camera maximum speed*: $25\frac{u}{s}$
- *Desired distance from A*: $7u$
- *Amount of obstacles*: 400
- *Obstacle radius*: between $0.5u$ and $0.8u$

In this scene there are many aligned equidistant columns, approximated by small obstacles. The interesting object A flies between them with a sine-like animation. This is a good test case to see how the algorithm performs when trying to avoid colliding with obstacles that get in the viewing line, while at the same time pursuing A.

A.4.3 Scene 3

- *Camera eye acceleration*: $5\frac{u}{s^2}$
- *Camera center acceleration*: $5\frac{u}{s^2}$
- *Camera maximum speed*: $30\frac{u}{s}$
- *Desired distance from A*: $10u$
- *Amount of obstacles*: 1
- *Obstacle radius*: $1u$

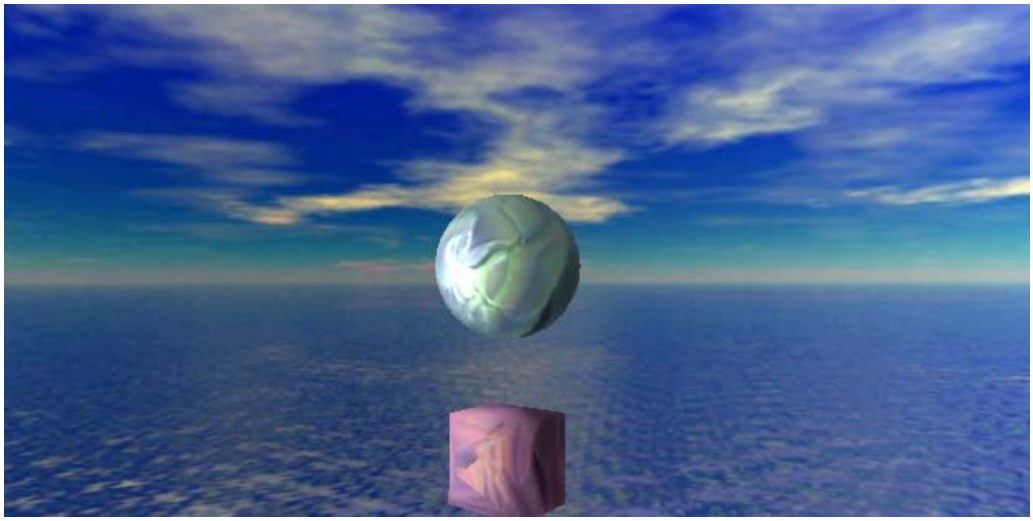

Figure A.3: A screenshot from Scene 3

This is a scene with two interesting objects A and B. A bounces over an obstacle (its height varies between 0u and 100u), while B remains static and has a rotation animation. The interest factor of B is 70, and the interest factor of A is inversely proportional to its height. When A is about to collide with the obstacle (its height is $\leq 30u$) then its interest factor is greater than the one of B. This test shows how well the camera performs when switching its view from one object to the other. One can see, that the camera will always strive to see the currently most interesting object in the scene.

A.4.4 Scene 4

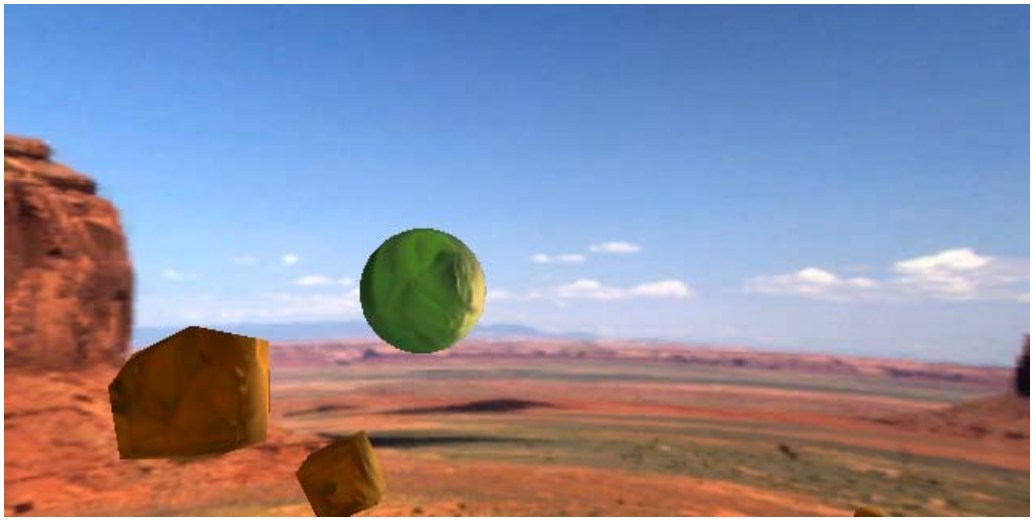

Figure A.4: A screenshot from Scene 4

- *Camera eye acceleration*: $75\frac{u}{s^2}$

- *Camera center acceleration*: $5\frac{u}{s^2}$
- *Camera maximum speed*: $125\frac{u}{s}$
- *Desired distance from A*: $10u$
- *Amount of obstacles*: 100
- *Obstacle radius*: between $1u$ and $3u$

In this scene all obstacles and the interesting object A have three sine-like animations, one for each degree of freedom. The speed and amplitudes of the animations are chosen at random. This way, all objects will keep a fixed maximum distance from each other, and their movements will seem to be completely unpredictable. The purpose of this scene is to test how well the algorithm performs on a chaotic environment. It can be seen that collisions between the camera and obstacles are not uncommon, and it's also a good way to test the fallback behavior of the camera (it gets a new position and orientation and its speed is set to zero).

Appendix B

Software Licenses

B.1 Freeglut

Freeglut code without an explicit copyright is covered by the following copyright:

B.2 OpenGL Extension Wrangler

GLEW is originally derived from the EXTGL project by Lev Povalahev. The source code is licensed under the modified BSD license[1], the SGI Free Software License B[2], and the GLX Public License[3]. The automatic code generation scripts are released under the GPL[4].

B.3 libjpeg

The authors make NO WARRANTY or representation, either express or implied, with respect to this software, its quality, accuracy, merchantability, or fitness for a particular purpose. This software is provided "AS IS", and you, its user, assume the entire risk as to its quality and accuracy.

This software is copyright (C) 1991-1998, Thomas G. Lane.
All Rights Reserved except as specified below.

Permission is hereby granted to use, copy, modify, and distribute this software (or portions thereof) for any purpose, without fee, subject to these conditions:
(1) If any part of the source code for this software is distributed, then this README file must be included, with this copyright and no-warranty notice unaltered; and any additions, deletions, or changes to the original files must be clearly indicated in accompanying documentation.
(2) If only executable code is distributed, then the accompanying documentation must state that "this software is based in part on the work of the Independent JPEG Group".
(3) Permission for use of this software is granted only if the user accepts full responsibility for any undesirable consequences; the authors accept NO LIABILITY for damages of any kind.

These conditions apply to any software derived from or based on the IJG code, not just to the unmodified library. If you use our work, you ought to acknowledge us.

Permission is NOT granted for the use of any IJG author's name or company name in advertising or publicity relating to this software or products derived from it. This software may be referred to only as "the Independent JPEG Group's software".

We specifically permit and encourage the use of this software as the basis of commercial products, provided that all warranty or liability claims are assumed by the product vendor.

[1] http://www.opensource.org/licenses/bsd-license.php, February 27th, 2008
[2] CD-ROM\src\glew\doc\sgi.txt
[3] CD-ROM\src\glew\doc\glx.txt
[4] CD-ROM\src\glew\doc\gpl.txt