Drone Path Planning

Marc Habermann
Saarland University
Saarbrücken, Germany

Submitted for the degree of Bachelor of Science
Saarland University
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung Informatik

Betreuender Hochschullehrer/Supervisor:
Dr.-Ing. Tobias Ritschel, Max-Planck Institut Informatik
Saarbrücken, Germany

Gutachter/Reviewers:
Dr.-Ing. Tobias Ritschel, Max-Planck Institut Informatik
Saarbrücken, Germany
Prof. Dr. Hans-Peter Seidel, Max-Planck Institut Informatik
Saarbrücken, Germany

Dekan/Dean:
Univ.-Prof. Dr. Markus Bläser, Universität des Saarlandes
Saarbrücken, Germany

Eingereicht am/Thesis submitted:
17.12.2015

Universität des Saarlandes
Fachrichtung 6.2 - Informatik
Im Stadtwald - Building E 1 3
66123 Saarbrücken

Declarations

Eidesstattliche Erklärung/Statement in Lieu of an Oath:
Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Signature:

Marc Habermann

Einverständniserklärung/Declaration of Consent:
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Signature:

Marc Habermann

Abstract

Unmanned Aerial Vehicles (UAV) become more and more popular today. The chairman of DJI which is one of the most important UAV producers told an online magazine that their turnover increased by a factor of four in the last years [1]. This development validates the increasing popularity of UAVs. But it is still challenging to control them and it is even more challenging to film important surface structures which are for example used for virtual model reconstructions. Therefore, the thesis presents an algorithm which is able to automatically plan the shortest drone path where all surfaces of an arbitrary mesh which the user can choose were seen. One of the main components to achieve this performance is a cost function which ensures that the drone saw most parts of a virtual model and that at the same time the travelling path is short. This function is then minimized with different optimization methods like the particle swarm and the Nelder-Mead method. The other part is a graphics processing unit-based (GPU-based) rendering technique to speed up the evaluation of the cost function. The result is a drone path which can be revisited, stored and used to reconstruct a 2D texture of a 3D surface. The findings were evaluated to demonstrate the efficiency of the path and the quality of the texture reconstruction.

Acknowledgements

A few years ago Prof. Dr. Hans-Peter Seidel and the Initiative Alte Schmelz originated a project for pupils from local schools. The goal was to teach them a professional modelling software called Blender [2] and to virtually reconstruct the historic buildings on the industrial area of the Alte Schmelz in Sankt Ingbert. Therefore, a team was built including Dr. Tobias Ritschel, Fabian Roschy and me. Dr. Ritschel was the head of the project and Fabian Roschy and me supported him as so called HIWIs (student assistants).

During the project we made use of an UAV to get good textures from the roofs of the buildings. We quickly recognized that the UAV control is really challenging and that the flight time is strongly limited. At the same time I wanted to write my bachelor thesis and we came up with the idea that it could try to solve these problems through automatically planning intelligent drone paths.

I want to thank my supervisor Dr. Tobias Ritschel for helping me to develop the ideas and thoughts presented in this work. I also want to thank the second reviewer Prof. Dr. Hans-Peter Seidel for giving me the opportunity to write my thesis at the Max-Planck Institut für Informatik in Saarbrücken. At last I want to thank Fabian Roschy for helping me to create the models which are used in the following and the other members of the Alte Schmelz project who supported me in any other way.

Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays drones are present in many daily life scenarios. This is due to the steep decrease in cost and also the control became more user-friendly. In general, this topic is no longer an expert field accessible only to a few people. One can get deep insights in UAVs with books like the “Handbook of Unmanned Aerial Vehicle” [3].

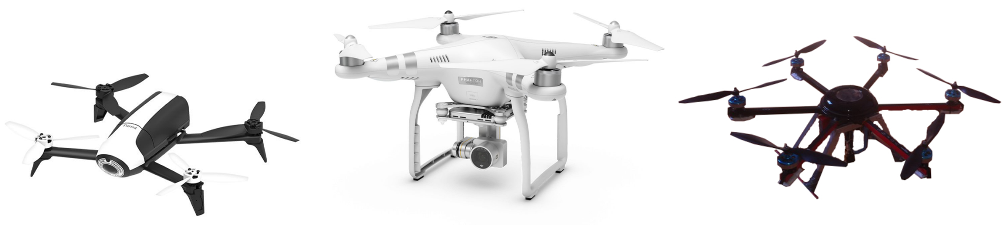

Figure 1.1: Different UAVs. Left: Parrot Bebop 2.0. Retrieved from the Parrot website [4]. Middle: DJI Phantom 3 Advanced. Retrieved from the DJI website [5]. Right: UAV which was used during the Alte Schmelz Projekt.

Additionally, there are many applications for custom users where a drone is useful. Two main fields can be mentioned. First the private sector where UAVs can be used for filming or just for fun. On the left hand side in Figure 1.1 one can see the Parrot Bebop 2.0 [4] which is a typical example for the private market since it is cheap, small and can be controlled via smartphones or tablets. But the possible range of rotation for the camera is limited since it cannot be rotated independently from the UAV. Second there are also many commercial applications where drones play an important role. When one thinks of professional flight videos they can be really helpful and are able to replace the expensive helicopter flights. Therefore DJI invented the Phantom 3 Advanced [5] (see Figure 1.1 middle) which is able to make pleasing flight videos since the camera is exchangeable and can be rotated in many directions. The last drone presented in the Figure is the one which was used during the Alte Schmelz Projekt. It is a mix of both previously mentioned UAV types.

The increasing number of drones was also recognized by many scientists. Therefore, the number of published papers regarding drones and drone applications increased in the last few years. While a search with the tag “UAV” within the years 2000 and 2001 in Google Scholar returns 2.590 results, the same tag delivers 24.800 entries within 2014 and 2015. Some of these will be presented in the following chapter.

Nevertheless, UAVs have two main drawbacks. On the one hand, although the control became easier it is still challenging to record appealing videos with it. On the other hand, the drones flight range is limited because it costs a lot of energy to fly. In consequence, the flight time is in general short. Therefore, many users want to film as much as they can until they run out of energy.

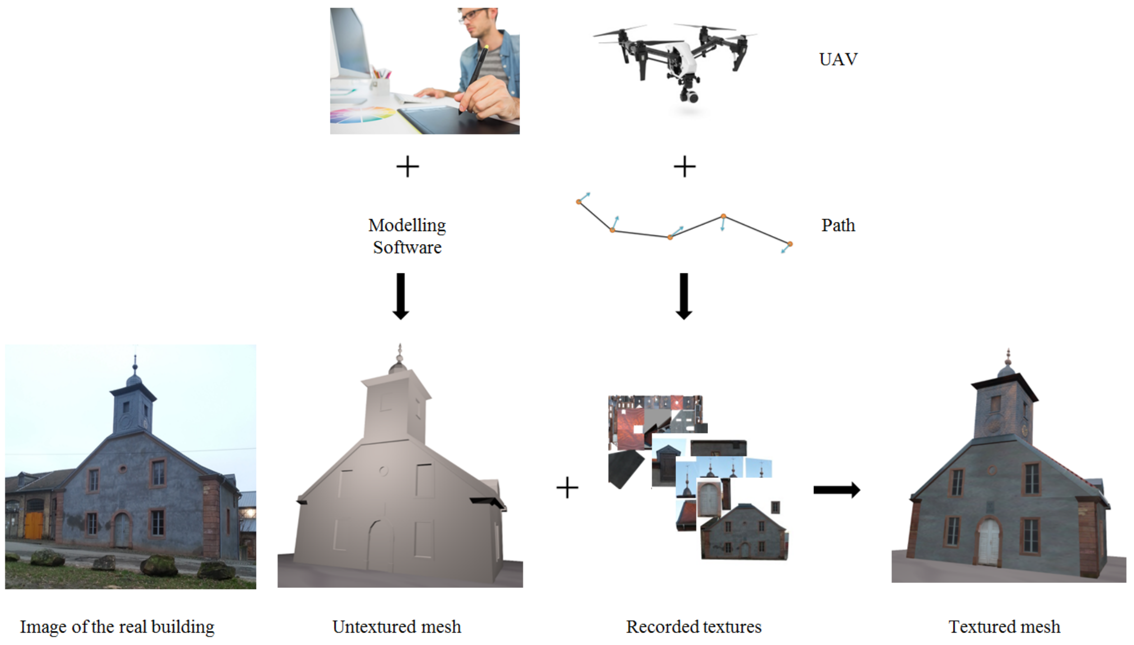

Figure 1.2: Pipeline of a modelling process. Parts of the graphic were retrieved from different websites [6, 7].

Imagine a graphic designer or a 3D model expert (see Figure 1.2) who has to construct a 3D model of a real building. To create a realistic reconstruction textures are essential. One can see that the untextured mesh looks unpleasant and synthetic. The simplest solution is to make pictures from the ground. But in many cases it is not enough to make the photographies from there. For example if one wants to model high buildings for Google Earth it is impossible to take good pictures from the upper part without leaving the ground. In this case, drones are a possible solution. With the recorded textures it is possible to create a textured mesh which looks much better and more realistic than the untextured one. But as already mentioned the range is limited. In consequence, it is desirable to record as much textures as possible in a finite time. Now the question arises: Which path will be the best one? Finding an answer is not easy. Another problem is that users often want to check if the drone path is good regarding other parameters than only the filmed textures. Therefore, a preview of the found path is also essential to give the user the opportunity of virtually looking at the drone behaviour.

Path planning itself is an interesting research theme. Alexander Schrijver wrote that "one can imagine that even in very primitive (even animal) societies, finding short paths (for instance, to food) is essential" [8]. Already in the earliest steps of computer science it was a prominent topic which has many applications. The most popular one which everybody knows is the navigation system. Even today many computer science groups spend their time on solving these kinds of problems in general.

The thesis picked up both themes and tried to combine them. On the one hand, the more practical part and on the other hand, the theoretical part. Therefore, an offline algorithm was constructed where virtual drone flights are simulated and evaluated towards their efficiency. The work can be segmented in two main parts:

1. At first the path planning itself which is a non-linear optimization problem is described in a mathematical way. Therefore, a cost function is constructed which ensures that the drone saw most of the surfaces of the mesh and at the same time it looks for a path which is short. These computations are done on the central processing unit (CPU).

2. The second part is a fast GPU-based implementation which evaluates the cost function. Therefore, the drone path is rendered in a 3D scene where the object of interest is placed. After each rendered path it is computed how good the drone filmed the mesh and how much energy the flight cost. These two values are then combined to get the value of the cost function.

The result is a virtual path where the drone saw most parts of the object e.g. a house model. Figure 1.3 left shows a single drone view. The observed textures are stored in a 2D texture map (Figure 1.3 right). At the same time, the path is really short. Since it is a virtual offline algorithm it can also be used to plan camera paths in the virtual environment.

So the thesis is able to solve the problem of planning efficient paths for arbitrary buildings. The only thing which is needed is a virtual model of the structure. It can also be used to animate a drone or to get a mapping between 3D views and a 2D texture map. At last it can visualizes which parts of the model are currently seen by the drone. All of these functionalities are important in practice.

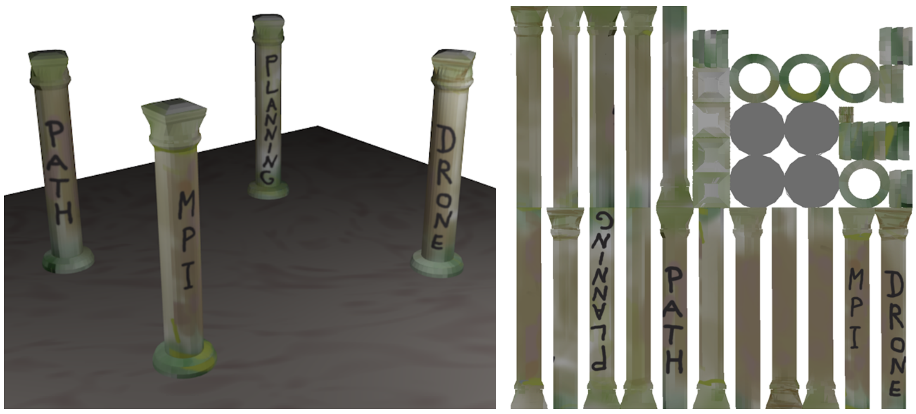

Figure 1.3: Left: Drone view of the virtual mesh. Right: Reconstructed texture map of the mesh.

1.2 Overview

The second chapter contains the related work. The first section summarizes the previous research regarding drone animation (Section 2.1). In the following section different path planning approaches are presented and it is explained why these are not able to deal with the previously explained problem and their limitations are shown (Section 2.2). The next section (Section 2.3) explains the general idea behind different optimization algorithms which are used in this thesis. The last section presents the framework in which the thesis was implemented (Section 2.4).

The third chapter shows the different parts of the program. First, the general idea behind the program pipeline is described (Section 3.1). Afterwards, how the animation model for the drone works is explained (Section 3.2). The cost function which consist of two different parts will be discussed in Section 3.3. Two different kinds of algorithms are presented in Section 3.4 and 3.5. Finally, the user input is described (Section 3.6) and the final output of the program is shown (Section 3.7).

The fourth chapter contains the evaluation of the thesis. First, different technical comparisons are made (Section 4.1 and 4.2) regarding for example the evaluation time. Then, the different implementations and algorithms are compared to illustrate the different efficiencies (Section 4.3).

The final chapter contains thoughts about future work (Section 5.1) and a conclusion which summarizes the main ideas (Section 5.2).

Chapter 2

Related Work

2.1 Drone Animation

There are already papers which can plan camera paths in virtual. The DJI GROUND STATION [9] allows the user to place 2D points on a 2D map. The result is a route where the drone visits each of the points. Since the customer can only place the position in 2D the control is still unintuitive. The QGROUND CONTROL system [10] allows the people to place the waypoints in a 3D scene. Therefore, the program control became more user-friendly. But both of these papers have several drawbacks. First, they do not give a virtual preview of the drone flight to improve the path if there is something missing in the video. Second, they do not allow the user to do precise adjustments like to the speed of the drone. All in all, the tools are not practicable for professional movie makers who want to plan exciting and pleasing quadrotor flights and they are also not able to deal with the previous presented path planning problem.

Joubert et al. [11] presented a drone application where the user can also plan the path in a virtual 3D scene. Therefore, the customer has to insert keyframes where the position and the viewing direction of the drone has to be specified. In contrast to the previous papers, the user can now review the created path on the screen. During the preview, one can precisely adjust the path and e.g. the speed of the drone. Finally, the program also informs the user if physical limits are reached e.g. when the speed of the drone is too high for a realistic flight. The paper is well suited for cinematographers since they are able to precisely plan routes in virtual environments without being afraid of physical limits. Regarding the previously presented problem, this paper is able to solve it since the user can manually try routes. Afterwards, he can compare the seen and the energy which has to be spent until he found a more or less good solution. The problem is that the papers algorithm does not check against collisions with buildings. Additionally, the user gets no visual feedback of how much of the textures on the mesh was already seen. The worst point is that one has to try it manually. For a good result the user has to plan multiple routes and he has to evaluate each individually in order to find a good solution. These steps cost a lot of time and it is not guaranteed that the solution is the optimal one or even near an optimal solution.

All the previous papers have one thing in common. The user has to define the path manually. But there are also papers which are focused on automated path planning like the work of Srikanth et. al. [12]. They invented an application for drones to achieve a rim illumination for the object of interest ,e.g., a human. Therefore, a light source is placed at the drone. If the relative position between the human and the photograph changes, the drone also changes the position in a way that the rim illumination is still realized. Lee et. al. [13] presented an algorithm that enables an UAV to track a ground vehicle autonomously. But both papers plan paths in a reactive manner. It means that the drone path is determined by the moving of another object ,e.g., the vehicle on the ground. Although these papers demonstrate that it is in general possible to create automated drone flights it cannot solve the thesis' path planning problem since the mesh in the scene does not move and its position does not determine the position or the viewing direction of the UAV.

Hrabar et. al. [14] presented a paper which tries to solve the problem of autonomously flying through an urban canyon. Therefore, they invented a stereo vision and optic flow based algorithm which enables an UAV to automatically travel through an outdoor environment without collisions. The main difficulty regarding the thesis' problem is that the approach is an online algorithm. In consequence, they are not able to solve the task offline in a virtual environment which is desirable as already mentioned.

Nikolos et. al. [15] published a paper which deals with the problem of path planning in a known environment. In contrast to the previous paper they constructed an offline algorithm which is able to determine a drone path from a starting point towards an end point in a 3D scene. In this case the problem is that they not involve the factor of how much the drone saw during it travels. Since they have no measure of how meaningful the seen parts of the scene really were they cannot decide if the UAV saw enough from the object of interest or not.

All in all, the previous work shows how interesting drone applications are and how wide the field of application really is. It starts with the obvious filming approach followed by the automated illumination and there are still other applications not mentioned here. At the same time it demonstrates that the thesis' problem cannot be solved by the existing papers in a good and easy way.

2.2 Path Planning

This section presents previous work in path finding. It tries to illustrate why existing algorithms are not able to deal with the drone path planning problem. Therefore, the algorithms and the extension to the thesis' problem will be described followed by a detailed explanation where these approaches will fail.

2.2.1 Travelling Salesman Problem

As already described above, the problem is the following: A virtual mesh is given which consists of different surfaces. The goal is to find a path which has a small distance. At the same time the drone camera should see each surface of the mesh in a good viewing position like Wolfgang Stuerzlingers method [16]. Now the next question arises: What is a "good viewing position" for the drone? The first criterion for a good one is that the surface is completely visible. Second, the angle between the surface normal and the viewing direction of the drone should be as small as possible. Otherwise, the perspective distortion would be too large. At last, also the distance between the surface and the UAV should not be too huge since the resolution of the filmed texture will decrease. With these three constraints it is possible to determine a perfect position for the drone and the corresponding viewing direction. In the following, the drone position and direction will be called *drone configuration*. If one assumes that these optimal configurations can be computed, then this could be done for each surface of the mesh. After this computation step one would have different points in 6D space representing the best drone configurations for the surfaces. If the drone visits each of these configurations one can be sure that the drone saw most parts of the mesh. Figure 2.1 illustrates the above thoughts.

Now it can be abstracted from the 6D model to a graph model. For the sake of simplicity it is assumed that the change of the viewing direction plays a minor role for the energy which the drone has to spend. Therefore, the distance between two position points is only taken into account. In the following, P is defined as the set which contains the position for each of the configurations. Each of the previously computed elements in P will now represent a node in a graph. The edges in the graph are determined as follows. Each node is connected to each other node. The Euclidean distance between the positions determines the weight for each edge.

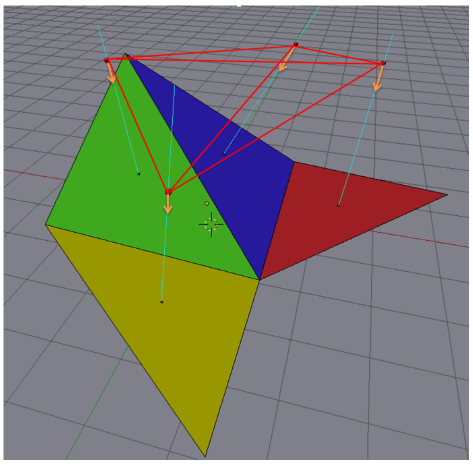

Figure 2.1: Visualization of the above thoughts about the TSP. The light blue lines represents the surface normals. The red dots are the camera position. The orange arrows illustrate the cameras viewing direction. The red lines shows the connections between the configurations.

With the previous thoughts a undirected graph $G(V, E)$ can be defined as follows:

- $V = \{v | v \in \mathbb{R}^3 \wedge v \in P\}$
- $E = \{(u, v, w) | u, v \in V \wedge w = weight(u, v)\}$
- $weight\left(\begin{pmatrix} a \\ b \\ c \end{pmatrix}, \begin{pmatrix} d \\ e \\ f \end{pmatrix}\right) = \sqrt{(a-d)^2 + (b-e)^2 + (c-f)^2}$

V contains the nodes of the graph whereas E is the set of the edges. A path p can now be defined as a list L of the edges where the drone travels along. The total distance for p is the sum of all edge weights contained in L. The goal is now to find a way through the graph so that at the end each node $v \in V$ was at least visited once. At the same time the total distance should be the least possible one. This goal is also known as the Travelling Salesman Problem (TSP) which is a combinatorial optimization problem. Imagine a graph only has 15 nodes means a mesh that only has 15 surfaces. The number of possible paths is then already more than a billion. This example shows how huge the computational effort would be already for really simple 3D meshes. Since the weights are computed with the Euclidean Distance one has the Euclidean TSP. From the theoretical aspect it is known that it is NP-complete [17]. So almost certainly no algorithm exists which has a polynomial worst-case runtime.

Usual scenes have much more than 1000 surfaces. In consequence, the TSP cannot be solved since the computational effort is too huge. The evaluation time would take several days or even more.

There already exist a lot of approximation algorithms. Christofides' 1,5 - approximation [18] solves the metric TSP in polynomial time. But there are still other difficulties which will be described in the following subsection.

2.2.2 Shortest Path in 3D Space

Previously it was assumed that the shortest path between two graph nodes is always the direct way between the two points. But when there is one or multiple geometries intersecting the direct route the assumption no longer holds. If there are no constraints made towards the geometry of the obstacles, it turns out that also this problem is in general NP-hard [19]. In consequence, the Euclidean distance cannot longer be used for the weight calculation. Therefore, one has the general TSP instead of the Euclidean TSP which is even worse for path calculations.

2.2.3 Art Gallery Problem

Above the problem was described in a combinatorial manner. But one can also think of it as a problem of computational geometry. The question of a shortest path can then be rephrased in the following way which is also called Art Gallery Problem.

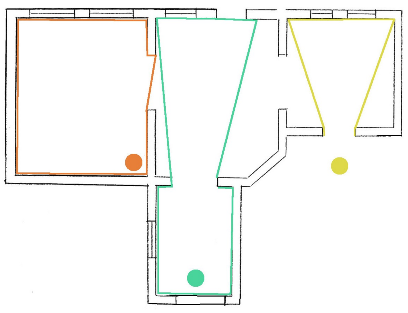

Figure 2.2: Art Gallery Problem in 2D.

The name comes from the following interrogation: One assumes that there is an art gallery which has a two-dimensional groundplan. The goal is now to observe each point inside this plan. Guards can be placed for surveillance. The question is now how many guards have to be placed at the minimum such that each point inside the groundplan is at least observed by one guard. Figure 2.2 visualizes the problem in 2D. The dots represent the guards. The colored boxes represent the room which the corresponding guard can observe. One can notice that some space is unobserved in this example.

For the drone path planning problem the guards represent the configurations of the drone in space. The goal is to find the minimal number of them in 3D space so that all surfaces of the mesh were seen by at least one guard. Although the distance between the found points would also play a role for the energy, which the UAV has to spent, it would be a good guess for a minimal path. The main point why this approach is not practicable is that Lee and Lin [20] showed that already for the original 2D case the problem of finding the minimum number of guards is NP-hard.

2.3 Optimization Algorithms

Previously it was shown that neither the graph-based approach nor the geometrical method are able to provide a solution for the path planning problem. The alternative for solving this problem are optimization algorithms which iteratively try to minimize a cost function. In the following section different optimization methods are introduced and explained since they are used in the minimization step in the thesis.

2.3.1 Downhill-Simplex

The Downhill-Simplex optimization or Nelder-Mead method was invented by John Nelder and Roger Mead in 1965 [21]. It is an approach which is able to solve non-linear optimization problems. The main pro of the Downhill-Simplex optimization is that it does not need to compute the derivatives compared to other algorithms like Newton's method.

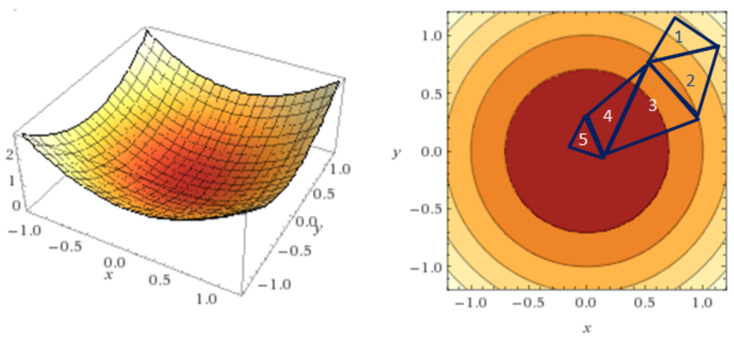

Figure 2.3: Illustration of a Downhill-Simplex optimization for a 2D function. Designed by Wolfram Alpha [22].

It takes a function $f : \mathbb{R}^n \to \mathbb{R}$. Now an initial simplex is constructed out of $n + 1$ points $x_i \in \mathbb{R}^n$ where $i = 1, ..., n + 1$. They have to be chosen in a way that they not lie in a hyperplane which has a smaller dimension. For example if one has a 2-dimensional function $f : \mathbb{R}^2 \to \mathbb{R}$ the simplex will be a triangle. If the points lie on the same line the triangle is degenerated.

The goal of the algorithm is now to move the simplex in a way that the function values become smaller which means the algorithm looks for local minima. Figure 2.3 shows a typical simplex optimization for the function $f(x, y) = x^2 + y^2$. The moving behaviour of the simplex is described in Figure 2.4 and can be influenced with the parameters α, β, γ and σ. After they are set they will remain constant during the optimization. There are different termination criterions. The algorithm can for example terminate if a constant number of iterations is reached or if the worst function value and the best function value are nearly the same. The best point x_{best} is then stored in x_0.

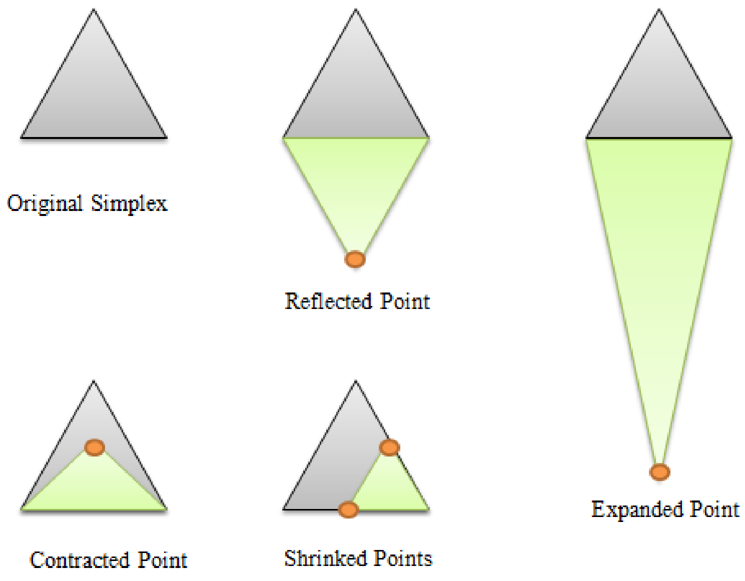

Figure 2.4: Different movements of the simplex in 2D. The grey triangles represent the original simplex. The green ones show the moved version. The orange dots visualize the updated points.

One drawback is that the algorithm often finds a local minima but it does not need to be the global optimum. Another problem is that the starting simplex influences the result. So different initializations can lead to different local minima. To avoid this behaviour a new simplex can be constructed around the old best point x_{best}. Then this one is again optimized. These steps can be repeated several times.

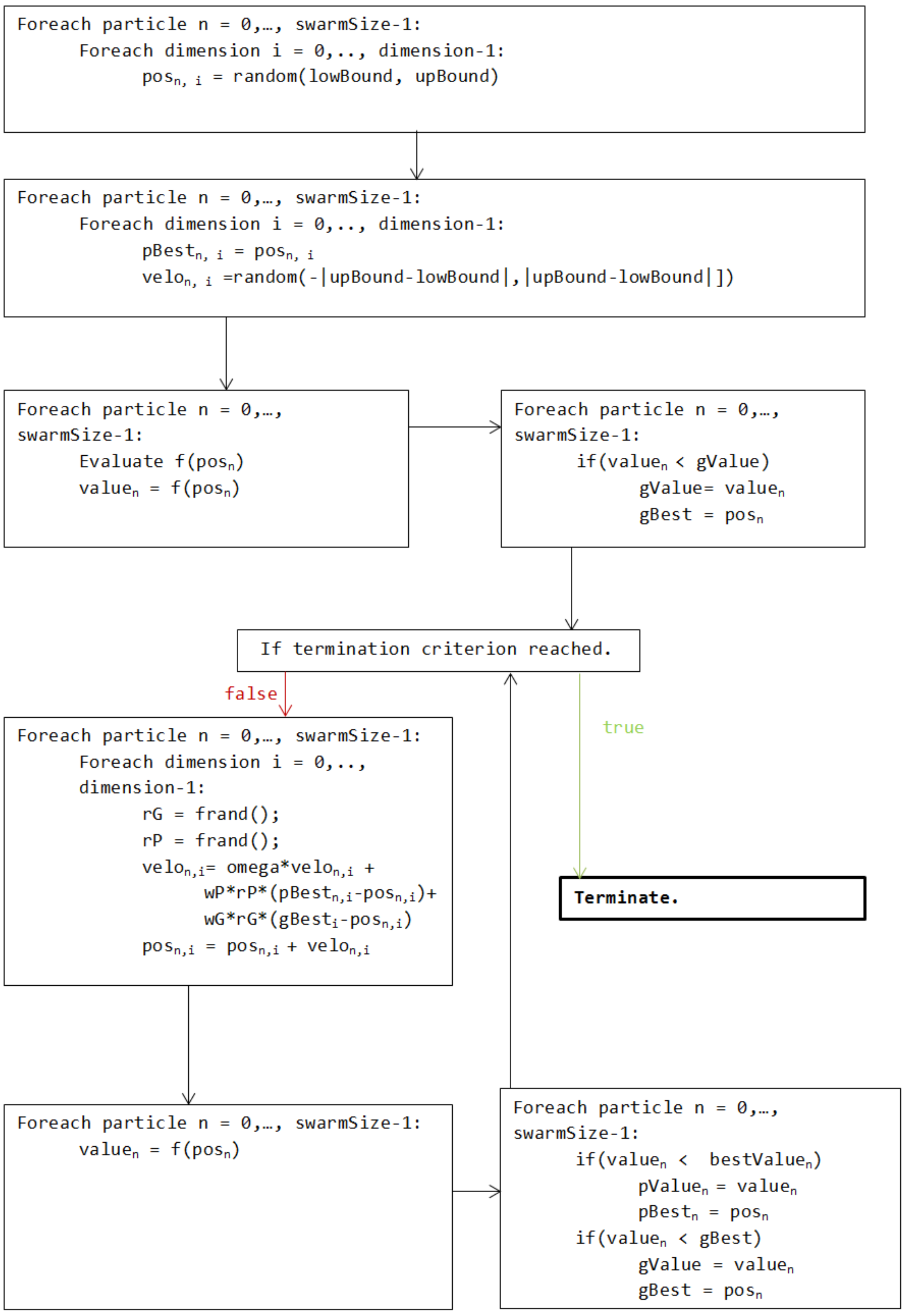
Foreach particle n = 0,…, swarmSize-1:
Foreach dimension i = 0,.., dimension-1:
pos$_{n,i}$ = random(lowBound, upBound)
Foreach particle n = 0,…, swarmSize-1:
Foreach dimension i = 0,.., dimension-1:
pBest$_{n,i}$ = pos$_{n,i}$
velo$_{n,i}$ =random(-|upBound-lowBound|,|upBound-lowBound|)
Foreach particle n = 0,…, swarmSize-1:
Evaluate f(pos$_n$)
value$_n$ = f(pos$_n$)
Foreach particle n = 0,…, swarmSize-1:
if(value$_n$ < gValue)
gValue= value$_n$
gBest = pos$_n$
If termination criterion reached.
false
true
Foreach particle n = 0,…, swarmSize-1:
Foreach dimension i = 0,.., dimension-1:
rG = frand();
rP = frand();
velo$_{n,i}$= omega*velo$_{n,i}$ +
wP*rP*(pBest$_{n,i}$-pos$_{n,i}$)+
wG*rG*(gBest$_i$-pos$_{n,i}$)
pos$_{n,i}$ = pos$_{n,i}$ + velo$_{n,i}$
Terminate.
Foreach particle n = 0,…, swarmSize-1:
value$_n$ = f(pos$_n$)
Foreach particle n = 0,…, swarmSize-1:
if(value$_n$ < bestValue$_n$)
pValue$_n$ = value$_n$
pBest$_n$ = pos$_n$
if(value$_n$ < gBest)
gValue = value$_n$
gBest = pos$_n$

Figure 2.5: Overview of the Particle Swarm algorithm.

2.3.2 Particle Swarm

The Particle Swarm Optimization was invented by James Kennedy and Russell C. Eberhart in 1995 [23]. They were inspired by the social behaviour of bird flocking and fish schooling and the resulting moving behaviour of them. The travelling of such a swarm has two main factors: On the one hand, there is the best experience of each individual and on the other hand, there is the best experience which the swarm made. Based on these factors the swarm will go in certain directions. The inventors tried to model this behaviour where a particle stands for a single animal. The whole fish or bird population is represented by the set of the particles. These thoughts lead to the particle swarm optimization which tries to optimize non-linear problems with simulating intelligent swarm moving. The algorithm is also a gradient-free optimization method.

Similar to the Downhill-Simplex method it takes a cost function $f : \mathbb{R}^d \to \mathbb{R}$ and minimizes the costs iteratively. The size of the swarm determines the number of the particles. Each particle n has a position or experience $p_n \in \mathbb{R}^d$, a best position $p_{\text{best},n} \in \mathbb{R}^d$ which contains the best position which the particle previously had and the corresponding best achieved value $p_{\text{value},n} \in \mathbb{R}$.

The particles also have a velocity $v_n \in \mathbb{R}^d$ which is responsible for how they move and how p_n will look like after it is updated. To determine v_n there are also weights needed called ω, w_g and $w_\text{p} \in \mathbb{R}$. The best position of the whole swarm and the corresponding cost function value are stored in $g_\text{best} \in \mathbb{R}^d$ and $g_\text{value} \in \mathbb{R}$. The algorithm assumes that the positions which are possible are bounded from below and from top. The bounds will be called l and $u \in \mathbb{R}$ in the following. The goal is now to move the particles' position in a way that the resulting function values become smaller. The whole algorithm is presented in Figure 2.5.

The particle swarm algorithm can also stuck in local optima but it is easier to optimize h-d functions compared to the Downhill-Simplex method.

2.4 Plexus

The dataflow network Plexus invented by Tobias Ritschel is a visual computing software which is used on the Max-Planck Institute for Informatics at Saarbrücken. It supports CPU and parallel GPU computations and is designed in a way that the core framework can easily be adapted to new requirements.

Since Plexus already has a lot of important components for graphical computations like voxelization it was the perfect framework to implement the thesis. A main application consists out of different smaller programs called *devices*. These devices usually have an input and an output which can be connected to a *graph*. The thesis' program consists out of already existing devices and newly developed devices which will be presented in the following sections. Figure 2.6 shows the user interface of Plexus where each box represents a device and the lines are the connections between them.

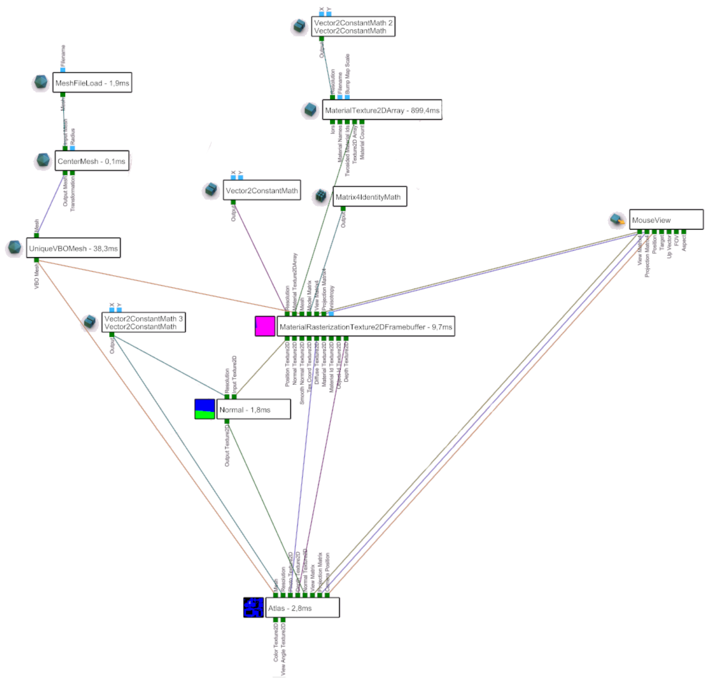

Figure 2.6: Plexus interface.

Chapter 3

Drone Path Planning

In the following chapter, the main work of the thesis is presented. It starts with a short overview of the program pipeline to illustrate the general program flow. Afterwards, the different main components and minimization algorithms are described. Finally, the user initialization, the program control and the output of the application are explained.

3.1 Program Overview

Figure 3.1 illustrates the program flow of the thesis. The algorithm takes as input a mesh for which the user wants to have the reconstructed texture map with the corresponding shortest drone path. The `Path Calculator` device is the central processing device of the application. It computes the next positions and directions for the UAV.

The current configuration, which is defined through the interpolated vector of the previous and the next configuration, will be sent to the `Path Render` device. In this step the actual drone view is rendered from the corresponding direction and position. If the end of the whole path is reached, a per pixel rating which tells how good a texture map pixel was seen by the drone and another map for the current visible parts is computed. The rating is encoded in a 2D texture which is sent to the `Path Rating` device. It sums up all the pixel values and computes a single float which is called *rating* in the following.

At the same time, the `Path Calculator` also sends the interpolated configuration to the `Collision Checker` device. Here it is checked if the drone hits the mesh. The result of the collision check is sent to the `Path Burden` device. It takes the old and new configuration and computes the amount of energy which has to be spent for this piece of the path. The sum of all parts will be called *burden*. Additionally, the device takes the result of the collision check for further computations described below.

Now the previous computed results are brought together in the `Evaluator` device. The rating and the burden are compared and it is decided if the current path is better or not. This feedback is sent to the `Path Calculator` since it is necessary for the next steps. On the other hand, the `Evaluator` informs the `Final Texture Map` device to update the old texture since a better path was found. If the path was not better, the `Final Texture Map` remains the same.

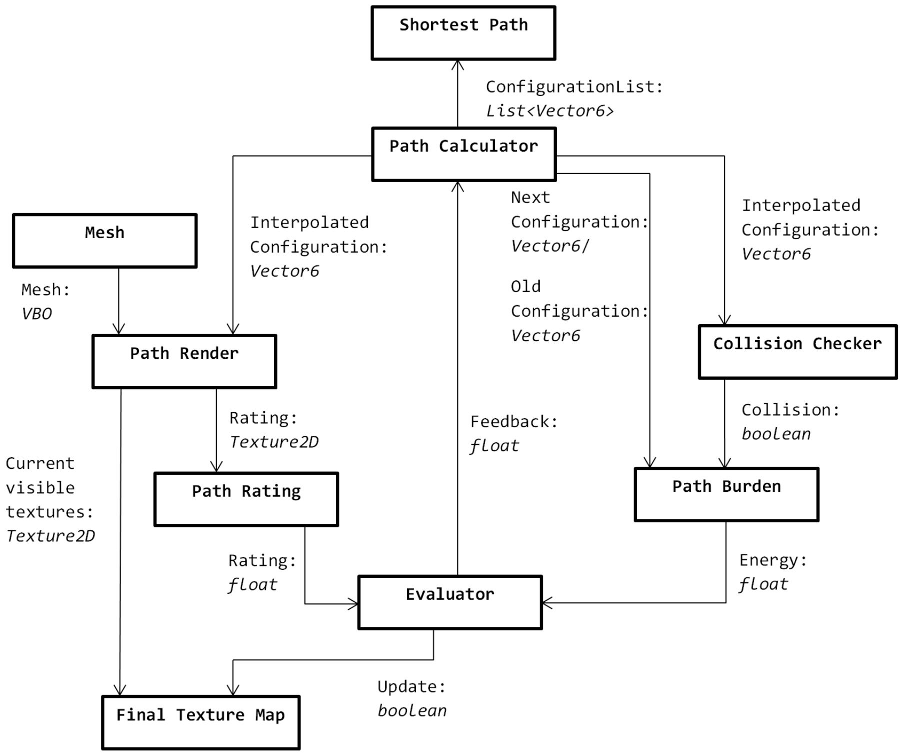

Figure 3.1: Overview of the program pipeline.

When one thinks of minimization the previous steps evaluated a path and the result of the evaluation is the cost function value. The `Path Calculator` now takes the value and starts the minimization which includes the computing of new configurations. These are again send to the `Path Render`, `Collision Checker` and `Path Burden`. Finally, the above steps are repeated iteratively until the `Path Calculator` reached its termination criterion. The best path is then stored as a list of configurations. The reconstructed texture is stored in the `Final Texture Map` and can be reviewed by the user.

In addition, a preview of the computed path is implemented which is not mentioned in the Figure above. Some functionalities which are used in the components were already given in Plexus. For instance the device which computes the viewing and projection matrix out of a known position and direction was implemented. Other ones which were available in Plexus are rasterization devices which visualizes the current scene, sink/source devices which enables temporary storage of data, mipmap devices which enables calculations on textures and mathematical devices which can solve scalar and vector computations.

3.2 Camera Animation

To create a virtual UAV a mathematical model of it has to be defined. Usually, a camera, which can be rotated, is placed on real drones. The thesis just considers this camera and abstracts from all other parts. The two basic parameters for its virtual model are the position which tells the program where in 3D space the drone takes place and the viewing direction which defines in what direction the UAV is looking at the moment. Both are encoded as a 3-dimensional vector. Additional parameters are the field of view and the aspect. Virtual cameras require the near and far clipping plane. Therefore, these two values can also be set.

To create an animation the drone has to travel along predefined lines in 3D space which can be linear or curved. The thesis animates the UAV along linked straight ones which are defined through a list of configurations. Each list element contains a 3D position (see orange dots in Figure 3.2) and a viewing direction (see blue arrows in Figure 3.2). The whole list will be called a *path*. The *length* of the path is $N - 1$ where N is the number of configurations in the list. The travel from one configuration to the next will be called a *route*. Figure 3.2 illustrates the terms.

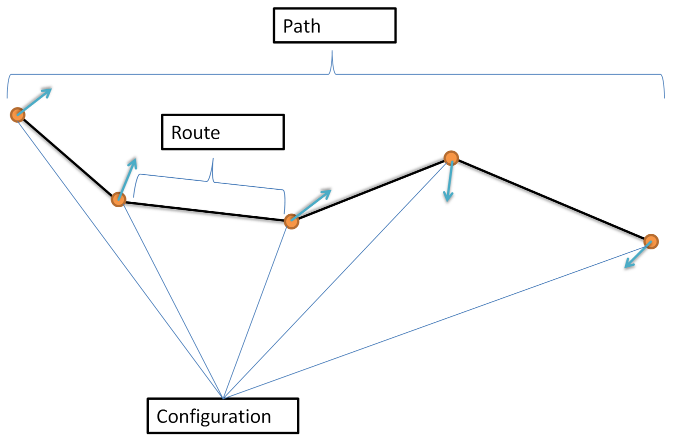

Figure 3.2: Visualization of a path in 2D.

The actual elements which define the current drone position $p_{\text{current}} \in \mathbb{R}^3$ and viewing direction $d_{\text{current}} \in \mathbb{R}^3$ are stored in the variables p_{old}, $d_{\text{old}} \in \mathbb{R}^3$ representing the old configuration and p_{next}, $d_{\text{next}} \in \mathbb{R}^3$ representing the new configuration. If the drone is between two configurations, the interpolated position and viewing direction is calculated to determine the exact current drone configuration. When the drone reaches the next configuration p_{old} and d_{old} are set to the previous p_{next} and d_{next} respectively. The new p_{next} and d_{next} are computed in different ways in the `Path Calculator` depending on the optimization algorithm which is used. Now the interpolation starts from the beginning. Figure 3.3 shows the implementation and how the exact computation is done for a route.

With the above animation model paths can smoothly be rendered. In consequence, the rating and burden can virtually be computed. How these two are exactly determined and how they are used in the cost function will be discussed in the next section.

```
//check against invalid drone speeds
if(speed < 0.f)
{
  LOG_ERROR << "Negative␣Speed␣not␣allowed";
}
else
{
  //direction in which the drone flights
  Vector3f travelVec = (pNext-pOld).normalized();
  //update the current position
  pCurrent = pCurrent+ travelVec*speed;
  //length between the current point on the road and the end
  // point
  float lengthToCurrentPoint = (pNext - pCurrent).length();
  //length of the current route
  float currentPathLength = (pNext - pOld).length();
  //ratio between the full path length and remaining way
  //which the drone has to pass
  float t = lengthToCurrentPoint/currentPathLength;
  //interpolated current direction
  Vector3f dCurrent = t*dOld + (1-t)* dNext;
  //interpolated target vector
  tCurrent= dCurrent.normalized()+pCurrent;
}
```

Figure 3.3: Computation of the virtual drone parameters for a route.

3.3 Cost Function

The cost function is the essential key for the later minimization. The constraints which are important for a good flight namely the quality of the reconstructed texture map and the short travelling path for the drone were already mentioned. Now it is time to build a mathematical equation which simulates this behaviour.

To ensure such a UAV movement the following function will be minimized:

$$\underset{t\in\mathbb{R}}{\operatorname{argmin}} \int_0^1 (1 - \mathit{ratingPoint}\,(p(t), v(t))) + \mathit{burdenPoint}(p(t), d(t))\,\mathrm{d}t \tag{3.1}$$

$$\begin{aligned} = \underset{t\in\mathbb{R}}{\operatorname{argmin}} & \int_0^1 (1 - (\mathit{ratingPoint}(p(t), v(t))\,\mathrm{d}t) \\ & + \int_0^1 (\mathit{burdenPoint}(p(t), v(t))\,\mathrm{d}t) \end{aligned} \tag{3.2}$$

t is the parameter to be minimized. $p : \mathbb{R} \to \mathbb{R}^3$ and $v : \mathbb{R} \to \mathbb{R}^3$ are functions which take a scalar value and return the position and the viewing direction of the drone on the path with respect to the scalar. In consequence, $(p(t), v(t))$ encodes the current drone configuration. The function $1 - \mathit{ratingPoint}(p(t), v(t))$ takes this configuration for the UAV and ensures that the quality of the reconstructed texture is good. The function $\mathit{burdenPoint}(p(t), v(t))$ avoids large paths. The integral in Equation 3.1 sums the values up to a total result for the path. Equation 3.2 implies that the rating and burden can be computed separately which is described in the following sections.

3.3.1 Collision Checker

In previous drone animation applications one saw that collision detection plays a minor role which is fine since the customer itself can design the path. But the thesis tries to find paths without any user interaction. Therefore, collision detection is necessary. It is done in the `Collision Checker` device which one already saw in the overview. As mentioned in Section 2.4 Plexus enables GPU computation. Therefore, this step is GPU-based since “good speeds and accuracy can be achieved using existing graphics hardware” [24].

To realize a fast collision checking it is assumed that the mesh is placed in a big bounding box (BBB) which has its center in the origin. The boxes side lengths are encoded as l in the following. Additionally, a 3D texture t_{input3D} is given which has a resolution r for each dimension. Each pixel in this texture represents a smaller bounding box (SBB) with edge length l/r in BBB. Now BBB is sliced in r pieces along the x-direction. An orthographic camera with viewing direction parallel to the x-axis is placed at the x-axis right in front of the current slice. Then the scene is rendered to the corresponding 3D texture layer in a way that in regions where a mesh is present the texture will be green. Otherwise it will be black. After each slice was rendered to the corresponding layer this step is repeated for the y and z-axis but with blue and red colors respectively. The repetition is necessary since it is possible to miss surfaces. Imagine it is only rendered along the x-axis and there is a surface where the normal is perpendicular to the viewing direction of the camera. The surface will not be rendered and as a consequence, the pixel values will not be set to green although there is an object. At the end of this step, each pixel in the in the 3D texture encodes if there is a mesh present then the color would be unequal to black. Otherwise, the color would be black. Figure 3.4 shows the summary of a 3D texture containing the comic house mesh (see also Figure 3.6 top left). One can see that for example the grassy ground which is perpendicular to the y-axis is rendered in blue at the bottom row and the red parts represent two of the four house walls. Since the normals of the ground are perpendicular to the x and z-axis it will only be rendered when the camera direction is parallel to the y-axis. So the projection along each axis is really necessary. This device was already implemented in Plexus and was the basis for the collision detection. The process is also called voxelization.

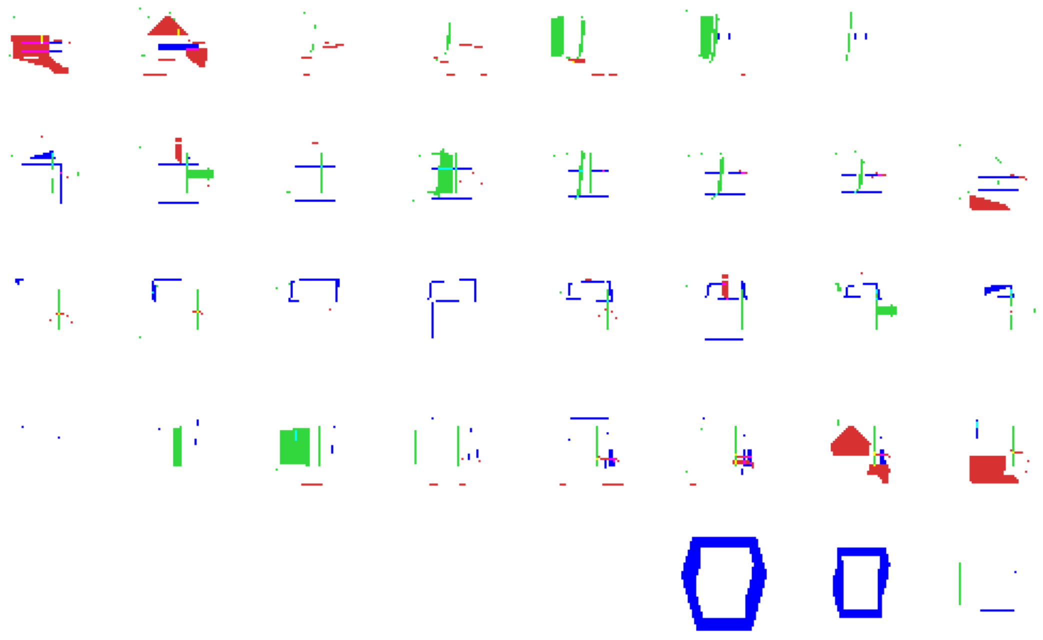

Figure 3.4: Summary of the slices in a 3D texture. The black parts are converted to white ones.

```
void main()
{
  //check if drone is in range of the mesh
  if(abs(p.x)>range || abs(p.y)>range || abs(p.z) > range)
  {
    outputTexture2D = 0.f;
  }
  //drone is in range of the mesh
  else
  {
    bool collision = false;
    highp int texRange = int(s+1.f);
    for(int i1 =-texRange; i1 <= texRange ; i1++)
    {
      for(int i2 =-texRange; i2 <= texRange ; i2++)
      {
        for(int i3 =-texRange; i3 <= texRange ; i3++)
        {
          //review the texture projected along the x-axis
          vec4 colorZ =texelFetch(input3D,ivec3(x+i1,y+i2,z+i3),0);
          //review the textures projected along the y-axis
          vec4 colorX =texelFetch(input3D,ivec3(y+i1,z+i2,x+i3),0);
          //review the textures projected along the z-axis
          vec4 colorY =texelFetch(input3D,ivec3(z+i1,x+i2,y+i3),0);

          if(colorX.y !=0.f || colorY.z != 0.f || colorZ.x !=0.f)
          {
            collision = true;
          }
        }
      }
    }
    //collision detected
    if(collision)
    {
      output2D = 1.f;
    }
    //no collision
    else
    {
      output2D = 0.f;
    }
  }
}
```

Figure 3.5: Shader code of the `Collision Checker` device.

The collision detector now takes the actual drone position $p \in \mathbb{R}^3$, the speed of the UAV $s \in \mathbb{R}$ and the previous computed 3D texture t_{input3D}. The virtual position corresponds to a coordinate in the 3D texture which are as x, y, $z \in \mathbb{R}$ given to the shader. First, it is checked if the drone is in BBB in general. If this not the case there will be no clashes since it is assumed that the mesh only takes place inside BBB. Otherwise, the collision device has to check the pixel value at (x, y, z) in the 3D texture. Since the drone is translated in a discrete way it can happen that the UAV skips over an obstacle. It is especially the case when the speed is high. To avoid this behaviour also the environment around the pixel coordinates (x, y, z) is checked. The size of the area depends on the speed of the UAV. If the mesh was placed at these locations the pixel value will be unequal to black as mentioned above. So it is known that the drone collides. The output is a 1x1 texture t_{ouput2D} which is white if a clash was detected else it is black. Figure 3.5 shows the collision shader implementation. The result will be used in the following section which describes how the burden is computed.

3.3.2 Path Burden

Previously the burden was described in the second integral in Equation 3.2 in a mathematical way. But for the thesis' purpose it has to be translated in the discrete domain. The burden encodes the energy which the drone has to spend for a path as already mentioned. Now in the discrete variant and in the path model one can say that it is defined as:

$$\begin{aligned} \sum_{i=0}^{N-1} & burdenRoute_{\text{trans}}(cList(i), cList(i+1)) \\ & + burdenRoute_{\text{rot}}(cList(i), cList(i+1)) \end{aligned} \tag{3.3}$$

cList is the list which contains the configurations. $burdenRoute_{\text{trans}}(a, b) : \mathbb{R}^6 \times \mathbb{R}^6 \rightarrow \mathbb{R}$ computes the burden for travelling from start position a to the end position b. $burdenRoute_{\text{rot}}(a, b) : \mathbb{R}^6 \times \mathbb{R}^6 \rightarrow \mathbb{R}$ computes how much the UAV has to spend for changing the viewing direction. The `Path Burden` device implements the burden in different ways depending on how exact the user wants to compute it. For a configuration c in the following $c.pos()$ will return the position and $c.dir()$ will return the direction. The three different modes are listed below:

- **Geometrical computation without rotation.** The simplest computation for the burden of a single route is the geometrical one which means that physical components like the gravitation and the mass of the drone are ignored.

Also the amount of energy which has to be spent for the rotation is set to zero here. In consequence, the route burden is defined as the Euclidean distance between the two positions:

$$burdenRoute_{\text{trans}}(a,b) = \|(a.pos() - b.pos())\| \tag{3.4}$$

$$burdenRoute_{\text{rot}}(a,b) = 0 \tag{3.5}$$

- **Geometrical computation with rotation.** The second variant also includes the difference between the starting direction and the end direction since rotation also costs a reasonable amount of energy.

$$burdenRoute_{\text{trans}}(a,b) = \|(a.pos() - b.pos())\| \tag{3.6}$$

$$burdenRoute_{\text{rot}}(a,b) = \left(1 - \frac{\langle a.dir(), b.dir()\rangle}{\|a.dir()\| * \|b.dir()\|}\right) \tag{3.7}$$

- **Physical computation.** The physical burden function tries to simulate a more physical correct behaviour. Therefore, the $burdenRoute_{\text{trans}}(a,b)$ for a route is separated in two components. It is differentiated between how high the drone flights and how far at all. For the height difference $\triangle h(a,b) : \mathbb{R}^6 \times \mathbb{R}^6 \to \mathbb{R}$ the UAV has to spent enough energy to overcome the potential energy $E_{\text{pot}}(a,b) : \mathbb{R}^6 \times \mathbb{R}^6 \to \mathbb{R}$. Since the drone weight $m \in \mathbb{R}$ and the gravitation constant $g \in \mathbb{R}$ are known it can be computed:

$$\triangle h(a,b) = \begin{cases} b.pos().y - a.pos().y & \text{for } a.pos().y < b.pos().y \\ 0 & \text{for } a.pos().y \geq b.pos().y \end{cases} \tag{3.8}$$

$$E_{\text{pot}}(a,b) = m * g * \triangle h(a,b) \tag{3.9}$$

For the remaining part it is assumed that the UAV needs a constant amount of voltage $u \in \mathbb{R}$ and current $i \in \mathbb{R}$ per time where it travels along the route. Since the length $\triangle s(a,b) : \mathbb{R}^6 \times \mathbb{R}^6 \to \mathbb{R}$ and the drone speed $s \in \mathbb{R}$ are given the travel time $\triangle t(a,b) : \mathbb{R}^6 \times \mathbb{R}^6 \to \mathbb{R}$ and $E_{\text{trans}}(a,b) : \mathbb{R}^6 \times \mathbb{R}^6 \to \mathbb{R}$ can be determined:

$$\triangle s(a,b) = \|(a.pos() - b.pos())\| \tag{3.10}$$

$$\triangle t(a,b) = \frac{\triangle s(a,b)}{s} \tag{3.11}$$

$$E_{\text{trans}}(a, b) = u * i * \triangle t(a, b) \tag{3.12}$$

Finally, $burdenRoute_{\text{rot}}(a, b)$ is equal to Equation 3.7 and $burdenRoute_{\text{trans}}(a, b)$ is defined as:

$$burdenRoute_{\text{trans}}(a, b) = E_{\text{pot}}(a, b) + E_{\text{trans}}(a, b) \tag{3.13}$$

If a route is determined the burden can directly be computed with one of the three models and there is no rendering needed. But it was said that the computation also considers clashes. Therefore, the `Collision Checker` gives feedback which is used to update the burden. If a collision is detected during render $burdenRoute_{\text{rot}}(a, b)$ and $burdenRoute_{\text{trans}}(a, b)$ are set to infinite.

3.3.3 Path Render

The rating for a path which was defined in the first integral in Equation 3.2 will also be translated in a discrete setting. It describes how good the textures were seen by the drone. For the whole path the rating will be equal to:

$$\sum_{i=0}^{N-1} ratingRoute(cList(i), cList(i+1)) \tag{3.14}$$

$cList(i)$ is again the ith configuration in the list which determines the path. Now the function $ratingRoute(a, b) : \mathbb{R}^6 \times \mathbb{R}^6 \to \mathbb{R}$ has to be defined. It represents the seen part of the mesh when the drone flow from configuration a to b. Therefore, the camera is animated (see Section 3.2) along the route and the resulting frames show the current view of the drone. During this step a device called `Atlas` which is part of the `Path Render` processes each one as follows.

The `Atlas` gets the rendered UAV view t_{view2D} (see Figure 3.6 top left) which shows the diffuse color of the mesh. The device also receives the same view but as a normal texture t_{normal2D} (see Figure 3.6 top right)which contains the surface normals of the model. The last texture which is given to the device is a depth map t_{depth2D} of the drone view. The position of the UAV $p_{\text{cam}} \in \mathbb{R}^3$ and the currently processed fragment position $p_{\text{frag}} \in \mathbb{R}^4$ of the mesh are also given. Finally, the device has the view matrix $\mathsf{V} \in \mathbb{R}^{4\times4}$ and the projection matrix $\mathsf{P} \in \mathbb{R}^{4\times4}$. It creates two UV maps called t_{rating2D} (see Figure 3.6 bottom right) and $t_{\text{diffuse2D}}$ (see Figure 3.6 bottom left) where p_{frag} has a corresponding texture coordinate.

First, one needs a mapping from the pixels in t_{rating2D} and $t_{\text{diffuse2D}}$ and the ones in the input textures. In consequence, p_{frag} has to be processed such that each surface point or fragment of the mesh has a corresponding coordinate in t_{view2D}, t_{normal2D} and t_{depth2D}. Therefore, first p_{clip} is defined by the the product $(\mathsf{P} * (\mathsf{V} * p_{\text{frag}}))$. The final pixel mappings and the resulting screen coordinates $c \in \mathbb{R}^3$ are arrived by a perspective divide $p_{\text{clip}}.xyz/p_{\text{clip}}.w$ and a shifting.

It was already mentioned in Section 2.2.1 that different constraints are given to reconstruct a precise texture map:

- **Depth.** First of all the filmed surface should not be too far away from the camera since the resolution of the filmed texture will decrease. Therefore, the `Atlas` looks at t_{depth2D} at position $c.xy$. The found value should be smaller than a constant threshold.

- **Viewing angle.** Another constraint was that the angle between the surface normal and the viewing direction should not be to huge. Therefore, the device computes the vector $k \in \mathbb{R}^3$ which is defined by $p_{\text{cam}} - p_{\text{frag}}.xyz$. The angle θ between k and the normal which is contained in t_{normal2D} at position $c.xy$ encoded as a RGB-value is computed. If it is greater than 70 degree, the angle is too huge for a good texture reconstruction.

- **Occlusion.** The surface should also not be occluded by another one. If this is true, $c.z$ has to be smaller than the corresponding pixel value in t_{depth2D} since a depth map pixel encodes the actual depth between the camera and the fragment which is rendered at that place. If $c.z$ is higher than this value it is known that the surface is occluded.

- **Viewing field.** Finally, the last and most obvious constraint is that the surface should be in the viewing range of the drone. Therefore, $c.x$ and $c.y$ have to be between zero and one.

If one of the above constraints is false the red value in the corresponding pixel in t_{rating2D} will be set to zero and the pixel in $t_{\text{diffuse2D}}$ will be blue, which signalizes that in the current view configuration no good texture reconstruction can be found for these surface parts. In Figure 3.6 one can see that for instance a part of the grass on the ground in the texture map is rendered in blue since the house occludes the ground partially.

If the above constraints are fulfilled for the pixel in $t_{\text{diffuse2D}}$ it will be set to the corresponding one in t_{view2D}. The red value of t_{rating2D} will be set to $cos(\theta)$. It encodes a rating for each pixel of the texture map. The range is between zero and one where zero represents the worst viewing angle and one encodes the best possible angle. In consequence, good ratings can be achieved when the angle is near zero which means that the drones viewing direction is almost parallel to the surface normal. Additionally, the blue color of each pixel which is part of t_{rating2D} is set to 0.7 to separate between non texture map pixels and texture map pixels. The lower right picture in Figure 3.6 illustrates the rating procedure. It can be seen that the smaller roof facing towards the camera is rendered in a bright pink which indicates that the red values were near one. This is caused by the fact that the camera's viewing direction is almost parallel to the surface normal. The main function of the `Atlas` shader is listed in Figure 3.7.

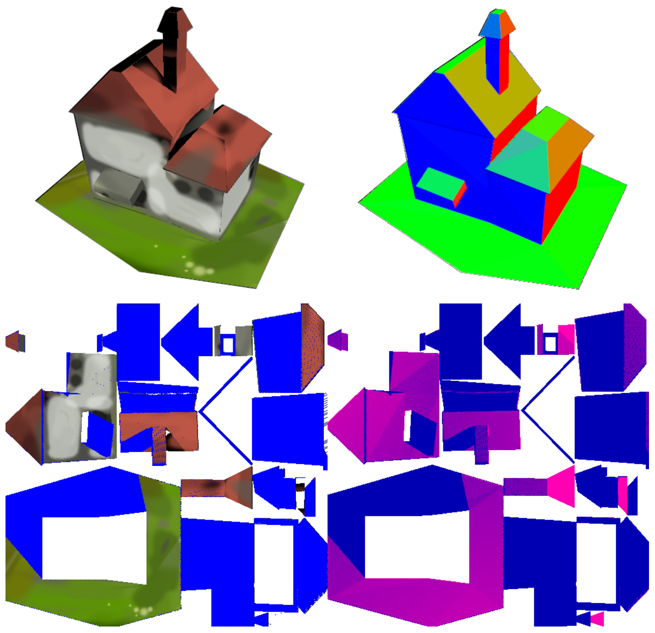

Figure 3.6: Input and output textures of the `Atlas`. Top left: Input diffuse texture. Top right: Input normal texture. Bottom left: Output diffuse texture of the `Atlas`. Bottom right: Output rating texture of the `Atlas`.

The current diffuse map and the corresponding rating map can be evaluated but it is also important to remember which parts were already seen by the drone. Therefore, a second device called `Global Atlas` is constructed which is also part of the `Path Render`. It takes t_{rating2D} and $t_{\text{diffuse2D}}$. At the same time, it holds his own diffuse texture map $t_{\text{globalDiffuse2D}}$ and rating texture map $t_{\text{globalRating2D}}$. For each coordinate in $t_{\text{globalRating2D}}$ it is compared if the corresponding pixel in t_{rating2D} has a better value. If this is true the RGB-value in $t_{\text{globalRating2D}}$ and in $t_{\text{globalDiffuse2D}}$ are updated to the values in t_{rating2D} and $t_{\text{diffuse2D}}$ respectively. If the drone reaches the next configuration and each frame was rendered, the global textures contain the seen parts of the mesh and the achieved rating for a route.

```
void main()
{
  vec4 viewSpacePosition = viewM * vec4(fragmentPosition, 1);
  vec4 clipSpacePosition = projectM * viewSpacePosition;
  vec3 screenCoord = clipSpacePosition.xyz / clipSpacePosition.w;
  screenCoord.xyz = screenCoord.xyz * 0.5 + vec3(0.5);

  vec3 toCamera = cameraPosition-fragmentPosition;
  vec3 normal = texture(normalTexture2D, screenCoord.xy).xyz;

  //CONSTRAINTS COMPUTATION
  //depth
  bool notTooFarAway = texture(depthTexture2D, screenCoord.xy).x
      <0.9987777f;
  //viewing angle
  float cosTheta = dot(normal, toCamera) / (length(normal) * length
      (toCamera));
  bool goodViewAngle = cosTheta <= 1.f && cosTheta> 0.342020143f);
  //Occlusion
  bool notOccluded =screenCoord.z <=  texture(depthTexture2D,
      screenCoord.xy).x +0.00001f;
  //Viewing field
  bool inViewingField = screenCoord.x >=0 && screenCoord.x <1 &&
      screenCoord.y >=0 && screenCoord.y < 1;

  //CONSTRAINTS FULFILLED
  if(notTooFarAway &&goodViewAngle && notOccluded & inViewingField)
  {
    ratingColor.x =cosTheta;
    diffuseColor = texture(viewTexture2D, screenCoord.xy);
  }
  //CONSTRAINTS NOT FULFILLED
  else
  {
    ratingColor.x = 0.f;
    diffuseColor = vec4(0.f,0.f,1.f,0.f);
  }

  //set the blue color to encode if this pixel is part of the
      texture map
  ratingColor.z = 0.7f;
}
```

Figure 3.7: Shader code for the `Atlas` device.

The `Global Atlas` can also be reset in two different ways. The scenario is the following: The drone travelled along n routes. Now the actual $n+1$th route is rendered and $t_{\text{globalRating2D}}$ represents the seen of the path with the length $n+1$. But the application tries different possibilities for the last part of the path. Therefore, $t_{\text{globalRating2D}}$ and $t_{\text{globalDiffuse2D}}$ have to be reset to the seen of the first n routes after each try. The `Global Atlas` is implemented in a way that the `Path Calculator` device can signalize that the textures have to be reset to the old state. Additionally, when a whole path was tested it is sometimes needed to reset the global textures to the initial state which means a black texture. The `Path Calculator` can also force the `Global Atlas` to do this.

3.3.4 Path Rating

$t_{\text{globalRating2D}}$ encodes the rating but in form of a texture instead of a single float value. Therefore, a map is created where pixels are set to white when the corresponding blue channel in $t_{\text{globalRating2D}}$ was set to 0.7. The `Path Rating` device creates a mipmap of this texture and takes the value contained in the 1×1 level which holds the average pixel value of the original map. The number will be called c_{texmap}. The red values in $t_{\text{globalRating2D}}$ are then used to create a second map where these values determine the pixel color ranging from zero to one. Again a mipmap is created and the value in the 1×1 level is taken which is called c_{rating} in the following. Now $c_{\text{rating}}/c_{\text{texmap}}$ is finally the rating as a float which has a range between zero and one. As a result the function value of $ratingRoute(a, b)$ can be computed and used for the next steps.

3.3.5 Evaluator

The `Evaluator` device gets the previously computed burden and rating for a route. These values are then used to compute the cost:

$$\begin{aligned} costRoute(a,b) = \alpha * (1 - ratingRoute(a,b)) + \beta * burdenRoute_{\text{trans}}(a,b) \\ + \gamma * burdenRoute_{\text{rot}}(a,b) \end{aligned} \tag{3.15}$$

where a is the starting configuration of the current route and b is the end configuration. The weights α, β and γ are used to balance the different factors of the cost function since the range of their possible values is different. The burden for example can theoretically lie between zero and infinite but the rating is always between one and zero. At the same time, the weights can be used to ensure a certain behaviour of the drone. For example, by increasing γ the UAV tries to avoid large rotations.

With these thoughts it is possible to get a mapping between routes and the corresponding cost values. In consequence, the cost function $costPath(clist) : \mathbb{R}^{6\times N} \to \mathbb{R}$ for a path where *clist* is the configuration list is known:

$$\begin{aligned} costPath(clist) = \alpha * &\left(1 - \left(\sum_{i=0}^{N-1} ratingRoute(cList(i), cList(i+1)\right)\right) \\ +\beta * &\left(\sum_{i=0}^{N-1} burdenRoute_{\text{trans}}(cList(i), cList(i+1))\right) \\ +\gamma * &\left(\sum_{i=0}^{N-1} burdenRoute_{\text{rot}}(cList(i), cList(i+1))\right) \end{aligned} \tag{3.16}$$

The device also holds the best cost c_{best} which was computed for the previous processed routes or paths. If the currently computed cost c_{current} is better than c_{best}, then c_{current} is stored and the `Evaluator` signalizes the `Final Texure Map` device that a better route or path was found. The device updates the texture maps $t_{\text{bestRating2D}}$ and $t_{\text{bestDiffuse2D}}$ to the current global rating and diffuse map. If c_{current} is not better, then the textures in the `Final Texture Map` device and c_{best} remain the same. At last the `Evaluator` sends the current cost to the `Path Calculator` device for the minimization which is described in the following sections.

3.4 Greedy Path Calculator

This section shows two different greedy algorithms which try to find the shortest path by optimizing over each route separately. It means a starting configuration is chosen and then the next optimal point is computed. This optimal solution is the new start configuration. These steps are repeated until a convergence criterion is reached. The cost function for a route will then be $cost_a(b) : \mathbb{R}^6 \to \mathbb{R}$ since the starting configuration a is known and the end configuration b is optimized. Each of the two presented greedy optimization methods are inheritance of the `Path Calculator` device.

3.4.1 Random Walk Approach

The random walk optimization approach gets an initial starting configuration which can be chosen by the user. Afterwards, a constant number of random target configurations are rendered and evaluated. This is done in different steps. First, a random target point is computed by the `Path Calculator`. The resulting route is rendered and evaluated. The `Path Calculator` gets the cost which is assigned to this route by the `Evaluator`. It is checked if this is the best value compared to the previous best one. If this is true the actual end configuration is stored and the corresponding global rating and diffuse texture are stored in $t_{\text{bestRating2D}}$ and $t_{\text{bestDiffuse2D}}$. Afterwards, $t_{\text{globalDiffuse2D}}$ and $t_{\text{globalRating2D}}$ are reset to the state before the actual route was rendered and the next random configuration is rendered and evaluated. If all random paths are tested the new starting configuration is set to the best found configuration. $t_{\text{globalRating2D}}$ and $t_{\text{globalDiffuse2D}}$ are set to $t_{\text{bestRating2D}}$ and $t_{\text{bestDiffuse2D}}$ respectively. Then again a constant number of different random routes are rendered and evaluated. These steps are repeated until the rating is better than a constant threshold.

A pro of this method is that it is really easy to implement. But it does not take into account which previously rendered routes where good or not. For instance, it could be that the random configurations are all in the same region. Although after the first run it is clear that this region has a low rating and high burdens. Moreover the performance is also not really fast which will be showed in the evaluation section. Therefore, the next greedy algorithm was implemented to overcome this drawback.

3.4.2 Realization of the Downhill-Simplex Optimization

As already mentioned, the random walk approach does not consider the already computed results in any way. Therefore, the Nelder-Mead method was chosen to improve the result and the speed of the computation. The general algorithm was already explained in Section 2.3.1. Now the adaptations to the concrete problem are described.

The Downhill-Simplex algorithm takes a function $f : \mathbb{R}^n \to \mathbb{R}$. For this problem n will be six as already mentioned since the next optimal configuration has to be computed which has a 3-dimensional position and a 3-dimensional viewing direction. In consequence, one has a simplex which has seven points. These points or configurations x_i $i = 0, ..., 6$ are randomly initialized. The route for x_i is rendered from the starting configuration to x_i and the achieved cost value is stored. Then the global texture maps are reset to their previous value and the next x_i is processed in the same way until each point has a cost value. Now the reflected point is computed, rendered, evaluated and the texture is again reset. These steps are also done for the expanded point, contracted point or when the simplex needs to be shrinked. Depending on the achieved cost values the simplex is moved in a certain direction and the new points are again evaluated.

If the algorithm found a best configuration the above procedure is repeated six times and seven optimal points with potential different coordinates are given. This is caused by the fact that the found minima depend on the random initialization of the starting simplex. These points form a new simplex which is again optimized to increase the probability of finding the global minimum instead of a local one. After this optimization terminates, the resulting final best configuration is stored and replaces the old starting configuration. Also $t_{\text{globalDiffuse2D}}$ and $t_{\text{globalRating2D}}$ are updated to the seen of the best configuration. The next optimal configuration is computed like before. Finally, if the rating is higher than a constant threshold, the path finding algorithm terminates.

The main pro of this method is that it also includes previously computed knowledge to find a local minimum. In consequence, it detects local minima faster and in the end, better configurations than the random approach which will be demonstrated in the evaluation section. But there are still problems remaining. For instance, the algorithm is not able to always find the global minimum. On top of that the optimization works in a greedy fashion which means that only over the next configuration a optimization is done. To overcome the later mentioned issue the following section presents an algorithm which optimizes over the whole path instead of single routes.

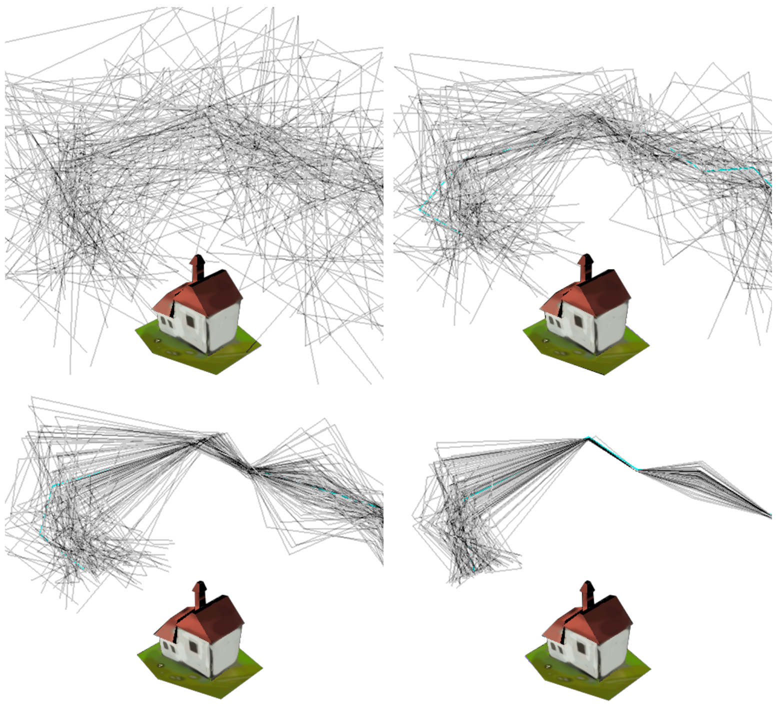

Figure 3.8: Visualization of the evolution of a swarm. Connected lines represent a particle. Top left: Initial swarm. Top right: Swarm after 4 iterations. Bottom left: Swarm after 8 iterations. Bottom right: Swarm after 16 iterations.

3.5 Non-Greedy Path Calculator

Non-greedy now means that the following algorithm tries to optimize over all configurations at the same time. It is also an inheritance of the `Path Calculator` device.

3.5.1 Realization of the Particle Swarm Optimization

The problem dimension increases to $N \times 6$ since N configuration have to be found and each configuration has 6 dimensions. The cost function which is optimized has then the form $f : \mathbb{R}^{N\times 6} \to \mathbb{R}$ and is equal to $costPath : \mathbb{R}^{N\times 6} \to \mathbb{R}$. The greedy algorithms took only the results of a single route since they optimize over each one separately. The cost function of the swarm optimization takes instead the whole path and returns a value which represents the result of it as already mentioned. Since the dimension of the problem increases rapidly the initial guess is crucial for good outputs. Therefore, it is not enough to just take random paths at the beginning. To overcome this drawback two methods were constructed:

- First, a log file which contains the configurations of a Downhill-Simplex optimization can be uploaded. Then, these configurations are used to initialize the first particle in the swarm. The method is called *DS initialization* in the following.
- The other method makes use of the fact that the object of interest is placed in the middle of the coordinate system and that its bounding box is known. The positions of the configurations are then all initialized in a way that they lie on a circle in the x-z-plane with a radius greater than the bounding box of the object. The center is on the y-axis. The radius and the height in y-direction can be adjusted. The corresponding viewing directions are just the normalized negative position vectors. Again, the first particle of the swarm is initialized with the previous computed configurations. This method will be called *circle initialization* in the following.

Since the first particle is a good initial guess the other particles can be sampled around this one.

Now each particle represents a path (see Figure 3.8). To get the corresponding function value the path is rendered and evaluated. Afterwards, all devices are reset and the next particle is rendered and evaluated. If these steps are repeated for the whole swarm, the initialization of the swarm is done and the iterative optimization is started in a way which was already described in Figure 2.5. During this optimization the particles align around the optimum which you can see in Figure 3.8. The termination criterion is reached when a fixed number of iterations is achieved or when the particles are strongly aligned. The best path is then stored in the swarms best particle.

The main pro of the method is, as already said, that it is able to optimize over a whole path instead of a single route. But at the same time the method is not as robust as the greedy algorithms which can be recognized during the initialization.

3.6 User Input

First of all, the user has to choose the object of interest for which he wants the drone path and the reconstructed texture map. He can create his own mesh e.g., in Blender [2] or other modelling softwares [25]. Then he has to export the triangulated model in the .obj format. Texture information such as the texture map and the texture itself should be in the .mtl file. In the Plexus interface he can choose the system path to the .obj file. The imported .obj is used to initialize a vertex buffer object (VBO) for GPU rendering. The mesh will be displayed in the virtual scene.

Afterwards, the user has to load the texture map image in Plexus. It works similar to the loading of the mesh. He just has to choose the right system path. Additionally, he can determine the resolution of the texture in Plexus.

Before the algorithm starts the user can determine a starting position and viewing direction for the greedy algorithms. Since it is often useful to constrain the initial point. Therefore, the functionality was realized.

The user has the possibility to choose between different focuses in the path calculation as already mentioned. If he wants for example a path where the drone rotates not too much around the own axis the user has to set the weighting for the rotation higher. In consequence, higher rotation values lead in total to higher function values and therefore to a higher influence in the minimization. At the same time, he can choose the number of random routes which are tested and he can define the initialization method for the particle swarm algorithm.

Also the simulation and the computation speed can be adapted by the user. Therefore, he can determine how fast the virtual drone should fly. Apart from the choice of the mesh and the corresponding texture all of the previous described initializations can be skipped and are just optional parameters for advanced user settings.

The user also has the possibility to control the application via keyboard. With 'S' he can start the path calculation. During the computation he can pause the program run with ’P’. In the paused mode he can then look at the already computed path parts in a preview by tipping ’A’. If the program terminated successfully a summary can be created by pressing ’W’. The summary contains all important facts of the path which were computed e.g., the total distance for the drone. The results are then stored in a Log.txt file.

Finally, it can also be chosen between the different optimization methods, namely random walk, Downhill-Simplex and particle swarm. But also the computation of the burden can be chosen. The possibilities were already mentioned in Section 3.3.2.

3.7 Program Output

After the user initialized the program and the termination criterion was met several outputs are given.

First of all, the user gets a texture map with the final reconstructed texture which enables him to get an idea of how the real reconstruction will look like and where distortions will be possible. On the other hand, he also gets the rating map which visualizes which parts of the mesh were seen by the drone and how good the viewing angles were.

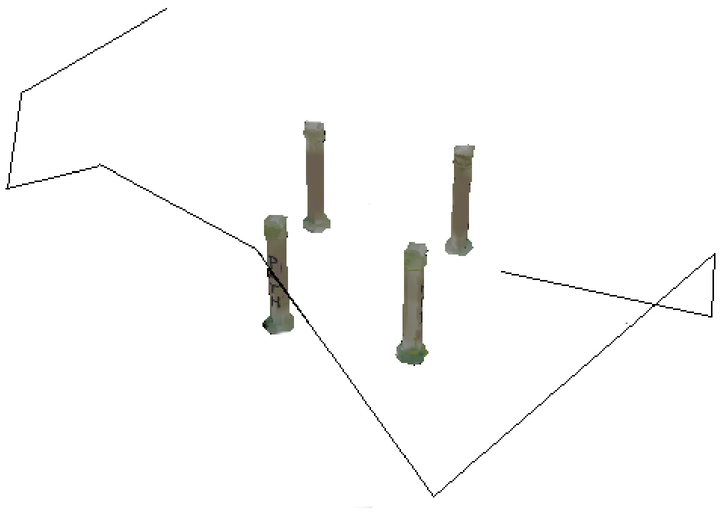

Figure 3.9: Overview of the reconstructed drone path.

Second, the user gets a .txt file called *log*. This file contains all important facts about the calculated path like the overall burden for it and the final rating which is achieved. Figure 3.10 shows a typical log. The first line is only used if the random approach is chosen as optimization algorithm. Then, the number indicates how many random routes are tested for each path element. Afterwards, the total burdens for translation and rotation are presented followed by the overall achieved rating. Then, the path length is mentioned and the weights which the user has chosen for the cost function are shown. Finally, the log file also contains each configuration computed by the algorithm.

Third, there is a device already mentioned in the last section which enables the user to review the final drone path. Therefore, he can start the animation which shows the path in the drone perspective. Below[1] one can find a video link which presents a rendered travelling.

At last, there is also a 3D scene available where the 3D path is drawn into together with the model to give a better overview of the whole travelling of the drone (see Figure 3.9).

```
==========Log of the Drone Path Planning Project =============
Minimum iterations per path: 200
Total Way Cost: 123.678
Total Rotation Cost: 0.473539
Total Rating: 0.781841
Total Time: 1.32704
Number of partial paths: 3
Rating Weight parameter: 200
Cost Way Weight parameter: 1
Cost Angle Weight parameter: 100
-------------------------------------------------------
Computed Positions & Viewing Directions:
-------------------------------------------------------
Pos: 40 | 0 | 0
Dir: -1 | 0 | 0
-------------------------------------------------------
Pos: 40.0668 | -17.4749 | -9.10086
Dir: -0.997719 | 0.0187758 | 0.0648392
-------------------------------------------------------
Pos: 34.2315 | -28.4319 | -17.6536
Dir: -0.792031 | 0.393885 | 0.466413
-------------------------------------------------------
Pos: 20.9492 | -30.1529 | -30.0341
Dir: -0.669116 | 0.553386 | 0.496033
```

Figure 3.10: The Log.txt file.

[1]`https://www.youtube.com/watch?v=whVabrvCgQw`

Chapter 4

Evaluation

After the program itself was explained in the last chapter, the evaluation now tries to visualize and validate the results of the thesis. Therefore, different test settings will be described and compared. For all of them the geometrical burden function which includes rotation was used.

4.1 Quality of the Reconstruction

First of all, a good quality of the reconstructed texture map is the main goal of the application. Since the idea was that modelling experts can use this map for the texturing of virtual models it is essential to achieve a high precision.

4.1.1 Different Models

The quality of a reconstruction should not depend on the geometry of the input mesh. Therefore, different models which were modelled and textured in Blender were tested (see Figure 4.1). In consequence, the ground truth diffuse texture maps are known and can be used for comparison. The mesh quality varies from simple structures to more complicated ones which have more surfaces. For each of them the reconstructed texture map will be compared to a ground truth texture map (see Figure 4.2). Additionally, the final rating texture is presented which visualizes which parts where seen by the drone. The colors of the original rating textures are remapped to grey values for better visualization and comparison. Bright grey values imply good viewing angles and vice versa. The maps have a resolution of 512×512. For all reconstructions the Downhill-Simplex implementation was used as the optimization algorithm. Each run was stopped after a fixed time. In consequence, the achieved ratings vary.

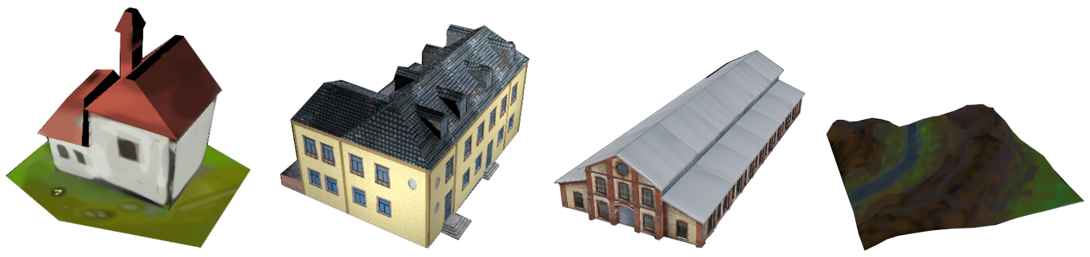

Figure 4.1: Different models. Left: Comic house (64 vertices, 120 tris). Middle left: Technisches Büro (1460 vertices, 2831 tris). Middle right: Industriekathedrale (14170 vertices, 11926 tris). Right: Terrain (4225 vertices, 8192 tris).

The simplest model is the "Comic house" mesh (see Figure 4.1 left). It has a size of 5.34 kilo-byte which means it has only a small number of surfaces. The reconstructed texture is pretty good and has only minor issues as you can see in Figure 4.2. The rating texture shows that also most of the mesh parts where visible during the flight. The rating for the path was about 75%.

The second model is a little bit more complicated and is called "Technisches Büro" (see Figure 4.1 middle left). The mesh has a size of 0.194 mega-byte. The reconstructed texture has more holes where no reconstruction can be made. But if one takes a look at the rating texture it is visible that in regions where a good rating was achieved also the reconstruction was good. This implies that if the program would run longer and found additional routes also the quality of the reconstructed texture will increase and it has less unfilled holes.

The last building model is the "Industriekathedrale" (see Figure 4.1 middle right) which has a huge number of surfaces and textures. The VBO has a size of 2.41 mega-byte. The reconstructed texture contains smaller parts where the texture was wrongly mapped. This is caused by the fact that the allowed angles between viewing direction and surface normal were not too restrictive. By increasing this constraint also these small issues will disappear. But on the other hand, the computation time would increase since more views are discarded in the atlas device.

Finally, also a model which is not a building was tested. It has the name "Terrain" (see Figure 4.1 right) and a size of 951 kilo-byte. The reconstruction was really good although it was not a building. This implies that the algorithm works probably for arbitrary meshes.

The above results illustrate that the method can handle meshes of different difficulties. In the previous chapter Figure 3.9 showed that also incoherent meshes are not a problem for the algorithm. Moreover it shows that by decreasing the performance or increasing the computation time the results will be better. Finally, often a rating of about 75% already suffices to produce visually pleasing results.

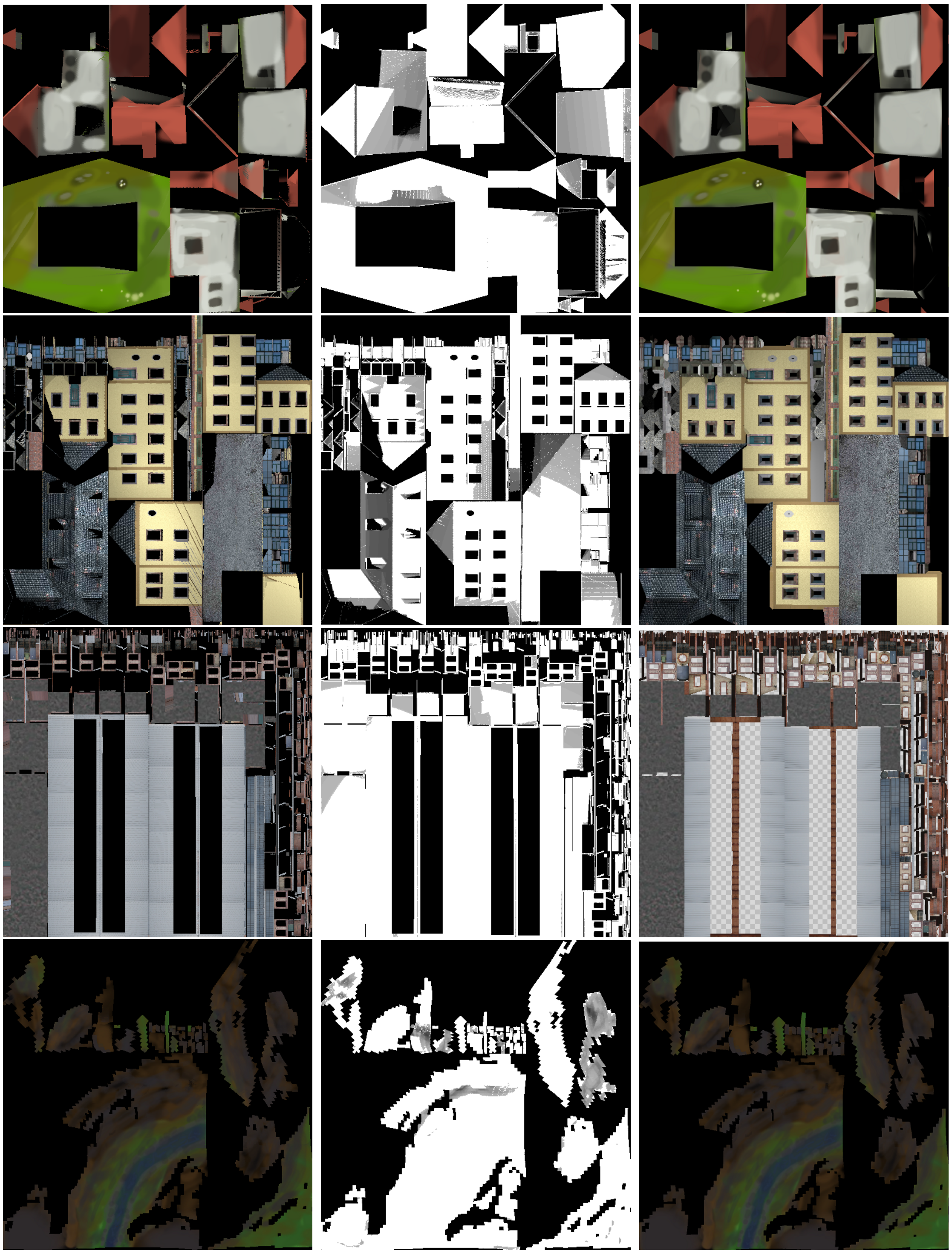

Figure 4.2: Quality of the reconstructed texture maps. Left column: Reconstructed texture map. Middle column: Rating texture map. Right column: Ground truth texture map.

4.1.2 Length of the Path

This section evaluates how much a single route contributes to the whole path. The Downhill-Simplex optimization and the "Technisches Büro" mesh were used to produce the results. Different program runs were done where the algorithm terminates after a fixed number of routes n. For each n the experiment was repeated 10 times. The resulting ratings for n were averaged. $\triangle n$ is defined as the difference of the average values for the ratings for n routes and $n-1$ routes. Table 4.1 shows the results for path lengths between one and five. The middle column contains the average values and the right column shows the corresponding $\triangle n$.

The first rating which is already pretty huge is caused by the fact that in the beginning nothing was visible. Therefore, each new information increases the value enormous.

The development of the values in the last column indicates that each route has at the end a positive effect towards the rating which is plausible since the drone saw more after an additional route. It means that the thesis' algorithm works in a stable way and ensures that each route has at the end a positive effect towards the final rating.

At the same time the last column also indicates that for increasing n the additional advantage becomes smaller. This behaviour is caused by the fact that it is more difficult to see new texture parts on a single route when most surfaces were already visible. In consequence, after a certain number of routes it is no longer profitable. The observation can be used to formulate an additional termination criterion. If $\triangle n$ is smaller than a constant threshold the program can stop. For the tested models presented in the previous section and the Downhill-Simplex approach it turns out that a path length of 10 suffices to reconstruct the texture almost perfectly.

Table 4.1: Number of routes for a path and the corresponding ratings.

Number of routes n	Average of the ratings (in percent)	$\triangle n$ (in percent)
1	43.4	43.4
2	57.2	13.8
3	67.6	10.4
4	76.3	8.7
5	84.3	8

4.2 Performance

After the quality of the results is validated it is also important that the program runs in appropriate time intervals. Therefore, the speed of the optimization algorithms and how the performance correlates to the resolution of the input and output images was tested. The comic house was the rendered model. The performance tests were made on a XMG P505 laptop which has a Intel Core i7-5700HQ and a NVIDIA GeForce GTX970M.

4.2.1 Random Walk Performance

As a first step, the random walk algorithm was examined. The number of rendered random targets for each route were set to 1000 to achieve similar results compared towards the Downhill-Simplex method. The termination was set to the point when a rating of 78% or higher was achieved. The program runs several times and the average time until the termination criterion was met was 10 minutes. But when the threshold was set higher than 78% the time increases really fast since it is difficult to get good new routes with the simple random walk algorithm when most of the textures were already seen. The performance test therefore shows that this approach is well suited for scenarios where the time is not that much important and were the texture ratings do not have to be higher than 78%.

4.2.2 Downhill-Simplex Performance

Also the Downhill-Simplex optimization method was tested. The termination criterion was the same as before for the random walk performance test. The achieved average time was 7.31 minutes. The main con of the previous performance test was that the random walk approach was not able to find good paths when most of the textures were already seen which is the case for ratings higher than 78%. In these rating regions the Downhill-Simplex approach performs well. So for higher rating results the Nelder-Mead algorithm is more suited.

4.2.3 Particle Swarm Performance

The performance of the particle swarm highly depends on how the particles are initialized. If the circle method is chosen the computation of the starting particles does not take too much time but the later optimization will take more time. If the swarm is initialized with a previous computed Downhill-Simplex path the additional effort for this computation has to be taken into account. At the end the final optimization will be shorter in time. But also the particle optimization itself has highly varying evaluation speeds depending on swarm size and path length. Therefore, an average value would not be sufficient to describe the performance. In general, the evaluation takes between 20 minutes and 2 hours.

4.2.4 Increasing Image Resolution

This section analyses the performance of the program when the image resolution for the texture maps increases. Therefore, 1000 random routes with a drone speed of 1.5 were rendered and the time was measured. Table 4.2 shows the results. One can see that the additional time which is needed when increasing the resolution is not too high. So the program also performs well when texture maps of higher resolutions are used.

Table 4.2: Evaluation times for different texture map resolutions.

Image resolution (as number of pixels)	Passed time (in seconds)
256×256	166.75
512×512	175
1024×1024	199.33

4.3 Efficiency of the Found Paths

Previously, the general quality of the texture map reconstruction was tested and the performance of the different optimization algorithms was compared. Now the approaches are evaluated towards their path efficiency. This means it is tested how short the found paths are and how good the achieved rating was.

4.3.1 Comparison: Random Walk vs. Downhill-Simplex

First, the random walk approach is compared with the Downhill-Simplex method. Therefore, both algorithms run several times. The termination criterion was reached after a fixed time. The resulting energies which the drone has to spent for translation and rotation and the achieved ratings were averaged. Table 4.3 shows the results.

Table 4.3: Comparison of the random walk approach and the Downhill-Simplex algorithm.

	Random walk	Downhill-Simplex
Translation burden:	127.9	85.6
Rotation burden:	60.9	29.4
Rating	72.5	71.5

The burden of the random walk approach is higher with a factor of 4/3 and the energy which has to be spent for the rotation is even twice as much. The resulting ratings are almost equal. In consequence, the found paths which were computed with the Downhill-Simplex are much more efficient compared to the results of the random walk approach.

4.3.2 Comparison: Downhill-Simplex vs. Particle Swarm

Finally, also the Downhill-Simplex and the particle swarm are compared towards their efficiency. Therefore, the Nelder-Mead method was started and stopped after a path length of seven. The first column in Table 4.4 shows the achieved results. Afterwards, the particle swarm was tested in two different ways: At first, the previous computed Downhill-Simplex route was taken as an initial guess. In the last run the circle method was chosen for the initialization. The results can be revisited in the the second and third column in Table 4.4. One can see that in general the particle swarm delivers better results. When it is initialized with a Downhill-Simplex result the ratings are only slightly better than the pure Downhill-Simplex outputs. Caused by the fact that the swarm can also stuck in local minima. But in this scenario the particle swarm is also a good opportunity to validate the efficiency of the input path. The circle method achieved the best results.

Table 4.4: Downhill-Simplex algorithm vs. the particle swarm optimization.

	Downhill-Simplex	Particle swarm (DS Initialization)	Particle swarm (Circle Initialization)
Trans. burden:	163.27	161.90	123.79
Rot. burden:	27.01	26.81	21.66
Rating	71.32	74.90	75.20

Chapter 5

Conclusion and Future Work

5.1 Future Work

Previously, the evaluation showed that the algorithm performs well in terms of reconstruction quality, performance and efficiency of the path. But there are still things which can be improved in further studies.

For instance the optimization can be adapted that it finds local or global minima in a more robust way. Especially when the function to be optimized is high dimensional as in the non-greedy optimization algorithm. There are already papers like the work of Jamian et. al. [26] and Pham et. al. [27] which are focused on optimizing high dimensional functions. Therefore, they could help to increase the performance. The burden which has to be spent for a route or path is an approximation as mentioned before. To compute precise energies one could made real performance charts of different drone types and use this information in the burden part of the cost function.

For the Downhill-Simplex optimization and for the particle swarm the parameter choices can be varied and tested since these values highly influence the result of the optimization. Moreover the performance and quality testing should be further improved by taking additional models.

One of the main cons at the moment is that the thesis is implemented so that the drone path can only be rendered in virtual but there is no real drone which can travel along the computed path. Section 2.1 already presented papers which are able to go from virtual to real. Therefore, it would be a nice adaptation which might also be considered for this thesis.

Finally, the drone animation itself is at the moment unrealistic since it is assumed that the drone can fly around corners without reducing the speed and the drone path consists of straight lines. One paper presented in Section 2.1 had a nice method to define drone paths in a smooth and consistent way. This approach can be used to make the UAV paths more pleasant. The current camera can be rotated in all viewing directions. But real drone cameras are often constraint to certain angles which are not mentioned in this thesis. A possible solution is the model of Joubert et. al. [11] since they already consider these limitations.

5.2 Conclusion

The thesis showed that it is in general possible to plan drone paths automatically where the main target was to optimize a path for a drone such that the drone saw as much as possible and at the same time the travel path should be efficient.

Therefore, an algorithm was implemented which consists of two main parts. First the cost function which ensures that the above constraints are fulfilled was constructed and minimized. Second a GPU-based evaluation of this function was implemented to speed up the process. The cost is structured so that the user can control the behaviour of the drone by changing the weights. The optimization is realized in different ways and the desired algorithms can be selected..

The output consists of two reconstructed texture maps, a log file which contains the informations about the path, a preview where the user can see how the drone records the model and a 3D overview of the found solution.

The evaluation showed that the paths are quite small and the resulting reconstructions have a good quality. The algorithm also performs well for high resolution texture maps and for meshes with a large number of surfaces. The performance test showed that each of the optimization has his specific pros and cons. The main future work will be focused on adapting the program in a way that the computed paths can be used for real drones.

Bibliography

[1] Nina Trentmann. Drohnen sollen endlich schlauer werden. http://www.welt.de/wirtschaft/article139665552/Drohnen-sollen-endlich-schlauer-werden.html. Accessed: 2015-12-09. Cited on iii

[2] Blender Foundation. Blender. http://www.blender.org/. Accessed: 2015-12-09. Cited on iv, 34

[3] Kimon P. Valavanis and George J. Vachtsevanos. Handbook of Unmanned Aerial Vehicles. 2015. Cited on 1

[4] Parrot. Parrot Bebop 2.0. http://store.parrot.com/de/bebop/515-product.html/farbe-white. Accessed: 2015-12-06. Cited on 1

[5] DJI. DJI Phantom 3 Advanced. http://www.dji.com/product/phantom-3-adv. Accessed: 2015-12-07. Cited on 1

[6] Inc. Controlled Color. Conveying Information with the Expertise of a Roselle Graphic Designer. http://controlledcolor.com/wp-content/uploads/2015/02/Impact-of-Graphic-Design-on-Your-Business.jpg. Accessed: 2015-12-07. Cited on 2

[7] DJI. Inspire 1 V2.0. http://store.dji.com/de/product/inspire-1-v2-free-intelligent-battery-tb47. Accessed: 2015-12-04. Cited on 2

[8] Alexander Schrijver. On the History of the Shortest Path Problem. *Documenta Mathematica*, 2010. Cited on 3

[9] DJI. DJI Ground Station. http://www.dji.com/product/pc-ground-station. Accessed: 2015-12-04. Cited on 6

[10] Lorenz Meier, Petri Tanskanen, Lionel Heng, Gim Hee Lee, Friedrich Fraundorfer, and Marc Pollefeys. PIXHAWK: A Micro Aerial Vehicle Design for

Autonomous Flight using Onboard Computer Vision. *Autonomous Robots 33, 1-2*, 2012. Cited on 6

[11] Niels Joubert, Mike Roberts, Anh Truong, Floraine Berthouzoz, and Pat Hanrahan. An Interactive Tool for Designing Quadrotor Camera Shots. *SIGGRAPH Asia*, 2015. Cited on 6, 45

[12] Manohar Srikanth, Kavita Bala, and Frédo Durand. Computational rim illumination with aerial robots. *Computational Aesthetics*, 2014. Cited on 7

[13] Jusuk Lee, Rosemary Huang, Andrew Vaughn, Xiao Xiao, and J. Karl Hedrick. Strategies of Path-Planning for a UAV to Track a Ground Vehicle . *AINS Conference*, 2003. Cited on 7

[14] S. Hrabar, G. Sukhatme, P. Corke, and K. Usher. Combined optic-flow and stereo-based navigation of urban canyons for a UAV. *International Conference on Intelligent Robots and Systems*, 2005. Cited on 7

[15] I.K. Nikolos, K.P. Valavanis, N.C. Tsourveloudis, and A.N. Kostaras. Evolutionary algorithm based offline/online path planner for UAV navigation. *IEEE Transactions on Systems, Man and Cybernetics*, 2003. Cited on 7

[16] Wolfgang Sturzlinger. Imaging all visible surfaces. *Graphics Interface*, 1999. Cited on 8

[17] Christos H. Papadimitriou. The Euclidean travelling salesman problem is NP-complete. *Theoretical Computer Science*, 1977. Cited on 9

[18] Nicos Christofides. Worst-Case analysis of a new heuristic for the travelling salesman problem. *Office of Naval Research*, 1976. Cited on 10

[19] John Canny and John Reif. Lower Bounds for Shortest Path and Related Probelms. *28th Annual IEEE Symposium on Foundation Compututer Science*, 1987. Cited on 10

[20] D.T. Lee nad A. Lin. Computational complexity of art gallery problems. *IEEE Transactions on Information Theory*, 1986. Cited on 11

[21] John A. Nelder and Roger Mead. A simplex method for function minimization. *The Computer Journal*, 1965. Cited on 11

[22] Wolfram Alpha. Wolfram Alpha computational knowledge engine. `http://www.wolframalpha.com/`. Accessed: 2015-10-02. Cited on 11

[23] James Kennedy and Russell C. Eberbart. Particle Swarm Optimization. *IEEE International Conference on Neural Works*, 1995. Cited on 14

[24] Michael Boyles and Shiaofen Fang. Slicing-Based Volumetric Collision Detection. *ACM Journal of Graphics Tools*, 1999. Cited on 20

[25] Autodesk Foundation. Maya. `http://www.autodesk.de/products/maya/overview`. Accessed: 2015-12-01. Cited on 34

[26] J. J. Jamian, M. N. Abdullah, H. Mokhlis, M. W. Mustafa, and A. H. A. Bakar. Global Particle Swarm Optimization for High Dimension Numerical Functions Analysis. *Journal of Applied Mathematics*, 2014. Cited on 44

[27] Nam Pham and Bogdan M. Wilamowski. Improved Nelder Mead's Simplex Method and Applications. *Journal of Computing*, 2011. Cited on 44