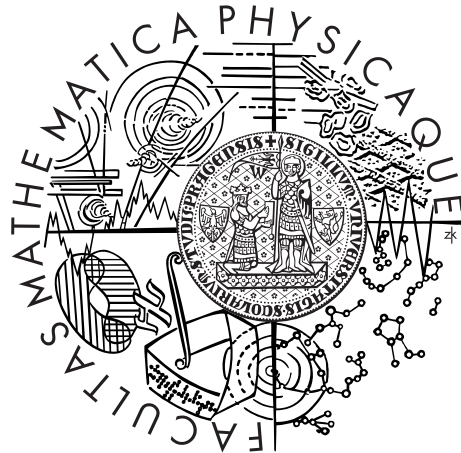


Charles University in Prague  
Faculty of Mathematics and Physics

## BACHELOR THESIS



Michal Ondrejáš

## Path planning in realistic 3D environments

Department of Theoretical Computer Science and Mathematical  
Logic

Supervisor of the bachelor thesis: prof. RNDr. Roman Barták, Ph.D.

Study programme: Computer science

Specialization: General Computer Science

Prague 2014

I would like to thank the thesis supervisor, prof. RNDr. Roman Barták, Ph.D., for his time and valuable counsel and leadership. I would also like to thank Mgr. Tomáš Plch, who was originally supposed to supervise this thesis, for his advice. Last but not least, I would like to thank my family for supporting me during my studies.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, July 29, 2014

Michal Ondrejáš

Název práce: Hledání cest v realistickém 3D prostředí

Autor: Michal Ondrejáš

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: prof. RNDr. Roman Barták, Ph.D.

Abstrakt:

Práce se zabývá implementací Drone3D — editoru 3D prostředí a hledání cest spolu s navigací kvadrikopty Parrot AR.Drone. Práce zkoumá principy aplikací běžících v reálném čase a vykreslování 3D grafiky s použitím rozhraní DirectX. Následně je pomocí DirectX implementována 3D grafika a uživatelské rozhraní editoru. Potom zkoumá různé možnosti implementace hledání cest — algoritmy a možnosti reprezentace prostředí. Je zvolena reprezentace prostředí jako síť krychlí a algoritmus *Lazy Theta\** na hledání cest a tento systém je implementován v editoru. Nakonec práce rozebírá experimenty s kvadrikoptérou Parrot AR.Drone a implementaci metody navigace kvadrikopty podle zadané nebo algoritmicky nalezené cesty. Řešení je ověřeno a zhodnoceno na základě vykonaných experimentů.

Klíčová slova: AR.drone, navigace, editor 3D prostředí, plánování trasy

Title: Path planning in realistic 3D environments

Author: Michal Ondrejáš

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D.

Abstract:

The thesis concerns with the implementation of a 3D environment editor with path-planning functionality and Parrot AR.Drone quadcopter control, named Drone3D. It explores the principles of creating real-time applications and drawing 3D graphics in DirectX, followed by the implementation of 3D graphics and user interface of the editor. Then multiple path-planning solutions are examined — algorithms and environment representation options. It is determined that the best approach is to represent the environment as a grid of cubes and use the *Lazy Theta\** path-planning algorithm. This system is then implemented as a part of the editor. Finally, experiments with the Parrot AR.Drone follow and a basic method of navigating the aircraft using a given or algorithmically found path is created. The method is implemented as a part of the editor and multiple tests are performed to verify and review the solution.

Keywords: AR.drone, navigation, 3D environment editor, path-planning

# Contents

<b>Introduction</b>	<b>3</b>
<b>1 3D environment editor</b>	<b>5</b>
1.1 Editor requirements . . . . .	5
1.2 Window layout . . . . .	5
1.3 Related applications . . . . .	6
1.4 Supported objects . . . . .	7
1.5 Program functions . . . . .	8
1.6 Use case . . . . .	8
<b>2 Chosen technologies and solutions</b>	<b>11</b>
2.1 API for 3D graphics . . . . .	11
2.1.1 DirectX and Windows game basics . . . . .	11
2.1.2 DirectX 3D model representation . . . . .	12
2.1.3 DirectX transformation pipeline . . . . .	12
2.1.4 Picking . . . . .	15
2.2 User interface concept . . . . .	16
2.2.1 Input processing . . . . .	16
2.2.2 Undo and redo . . . . .	17
2.3 Saving the environment into a map file . . . . .	17
<b>3 Editor implementation</b>	<b>18</b>
3.1 DirectX component . . . . .	19
3.1.1 Basic DirectX classes . . . . .	19
3.1.2 Camera class - scene view . . . . .	19
3.2 Models component . . . . .	21
3.2.1 Model class . . . . .	21
3.2.2 ModelGroup class . . . . .	23
3.2.3 Order of transformations . . . . .	23
3.2.4 ModelWrap class . . . . .	24
3.2.5 Creating a group - maintaining data consistency . . . . .	24
3.2.6 Destroying a group - maintaining data consistency . . . . .	24
3.3 Map component . . . . .	25
3.3.1 File structure . . . . .	25
3.3.2 Classes of the Map component . . . . .	26
3.4 User interface implementation . . . . .	27
3.4.1 Layout and interaction . . . . .	27
3.4.2 Basic classes of the GUI component . . . . .	28
3.4.3 Sprite class . . . . .	28
3.4.4 BigBrother class . . . . .	28
3.4.5 How it works . . . . .	29
3.4.6 Undo and redo - the Command pattern . . . . .	29
3.5 GameMain component . . . . .	30

<b>4</b>	<b>Path planning</b>	<b>32</b>
4.1	Environment representation . . . . .	32
4.1.1	Determining blocked cells . . . . .	35
4.1.2	Computing bounding ellipsoids . . . . .	36
4.1.3	Quadrocopter in the environment . . . . .	36
4.1.4	From a bounding ellipsoid to blocked cells . . . . .	37
4.2	A survey of path-finding algorithms . . . . .	38
4.2.1	Dijkstra algorithm . . . . .	38
4.2.2	A* . . . . .	41
4.2.3	Non-Admissible heuristics in A* . . . . .	42
4.2.4	Theta* . . . . .	42
4.2.5	Lazy Theta* . . . . .	43
4.3	Implementation . . . . .	44
4.3.1	Environment representation . . . . .	45
4.3.2	Blocking cells . . . . .	45
4.3.3	Lazy Theta* implementation . . . . .	46
4.3.4	Adapting the algorithm to quadrocopter navigation . . . . .	47
<b>5</b>	<b>Plan execution</b>	<b>48</b>
5.1	Parrot AR.Drone . . . . .	48
5.1.1	Sensors on the Parrot AR.Drone . . . . .	48
5.1.2	Flying with the Parrot AR.Drone . . . . .	49
5.2	Drone communication . . . . .	50
5.2.1	Structure of commands . . . . .	51
5.2.2	Incoming data streams . . . . .	51
5.2.3	Network interface . . . . .	52
5.3	Navigation routine . . . . .	53
5.3.1	Hover phase . . . . .	54
5.3.2	Flight phase . . . . .	54
5.3.3	Computing the change of drone parameters . . . . .	56
<b>6</b>	<b>Experiments</b>	<b>57</b>
6.1	Straight flight 3 metres . . . . .	58
6.2	30 degree turn . . . . .	59
6.3	90 degree turn . . . . .	59
6.4	Zig-zag . . . . .	61
6.5	Path from algorithm . . . . .	61
	<b>Conclusion</b>	<b>63</b>
	<b>Shortcomings and possible extensions</b>	<b>65</b>
	<b>Bibliography</b>	<b>66</b>
	<b>List of Tables</b>	<b>68</b>
	<b>List of Abbreviations</b>	<b>69</b>
	<b>Attachments</b>	<b>70</b>

# Introduction

In the recent years, quadcopters have appeared on the market as a new toy. While they certainly offer a lot of entertainment, we (and many others) see them differently. We see an opportunity for interesting experiments in the field of artificial intelligence and planning. Quadcopters along with other multi-rotor aircraft are very maneuverable and can be equipped with a wide variety of sensors. Thanks to that, a wide range of experiments can be performed with them. A lot has already been done in the field but there still is a lot of space left to fill.

The massive potential of multi-rotor aircraft for future applications motivated us further to start experimenting with them. They come in various sizes, and can be adapted or purpose-built to perform various tasks via the number of rotors and selection of equipment. They are suitable for various applications, ranging from entertainment through delivery services up to search & rescue operations or military applications. Therefore, we believe research in this field to be very important. We would like to create a founding base, upon which we (and others) could build in the future, or use it directly in real-world applications.

The most fundamental aspect of quadcopter operation is path planning. In order to be able to perform tasks, the aircraft (or navigator) has to possess the ability to navigate from point A to point B safely and efficiently. We will examine this in the thesis. We will study path-finding algorithms, pick the one we consider best, and evaluate its performance by experimenting with a real quadcopter. To make this easier, we will first create a 3D environment editor which will allow us to conduct the experiments.

As stated earlier, many experiments have already been performed. Most of them have used sensors outside of the aircraft. An example of that is using a system of cameras in a room to accurately determine the position or other parameters of the aircraft within. We aim to use only the sensors available on the quadcopter, and a computer as a controlling device. This is, however, just a small step in a big dream of one day having an autonomous aerial drone.

We will model various scenarios. The quadcopter will always be placed in a closed room with or without objects. We will model this room in the program and set start and goal positions in the real room as well as in the model. Objects will be obstacles that the quadcopter must avoid. Then the program will have to find a path from start to goal that the quadcopter can follow, and finally navigate the aircraft without crashing.

There are three main goals of this thesis. The first is to create a 3D editor, where it will be possible to model a real room with obstacles in a simple and intuitive way. The second goal is to implement a path-planning system. The third goal is to add the capability to navigate a quadcopter using the editor and the path-planning system. The application should be able to navigate a quadcopter from a given starting point to a given goal point within the room using the created model of the room to avoid obstacles and fly in a safe and steady manner.

The structure of the thesis is as follows. First, we will examine the features

of the environment editor. Then, we will introduce the chosen solutions and technology - DirectX. A concise discussion of the implementation of the editor will follow. Then we will move on to path-planning algorithms, take a look at a selection of the algorithms we considered and at the one we chose - *Lazy Theta\**. This section is, to a great extent, theoretical. Finally, we will introduce the Parrot AR.Drone 1.0 and discuss the performed experiments.



# 1. 3D environment editor

The editor is the core part of the application. All the other components are connected to it and extend its functionality. It allows the user to model the environment — a room with obstacles and the start and target locations. Then the path-finding algorithm can be run to find a path from start to target suitable for a quadcopter. Finally the application can connect to a Parrot AR.Drone quadcopter and navigate it along the path. Everything is done in three-dimensional space. In order to draw 3D objects and manipulate with them, a 3D graphics API (Application Programming Interface) is required.

To navigate a quadcopter, the application must repeatedly send commands to the aircraft with a very short delay. Thus, the application should be real-time. A new frame should be rendered at least 30 times per second.

## 1.1 Editor requirements

In this short paragraph, we discuss the necessary features our application must implement. Either because they are the core functions of our application or because a potential user simply expects to have such features at his disposal.

The most basic functionality comprises operations on objects in the environment. These operations cover the Addition of new objects and modification/removal of objects added before. The application must also provide a 3D view of the environment, so that the user can see what he has created. It should then have an intuitive user interface implemented in such a way, that the user can interact with the application seamlessly. It should also be possible to correct errors easily, thus undo and redo functionality is a must. It is also important to provide functions to save the environment into a file and load it later.

## 1.2 Window layout

Let us first take a look at what the application looks like. There are essentially two options how one can create a 3D environment editor. It is important to realize that this choice directly determines the way the user interacts with objects.

The first is to split the window into four smaller windows or sub-windows. These provide the user with a 3D view of the created world and three 2D projections, each aligned with one of the (orthonormal) axes. This is, however, rather complicated to implement, and more importantly, to work with. The user would have to use the 2D views to modify objects, and he could observe the changes in the 3D preview. This is perhaps easier to grasp but less efficient to use. For example, changing the dimensions of an object would require one operation per each of the 2D views, so 3 operations overall. Implementation-wise, to merge these operations into one, we would obtain the same result as when using only a single 3D window.

The other approach is to only create a 3D window, where the user will be able to both modify and view the environment. This is easier to implement and for the user to work with. Unlike in the example before, changing the dimensions

of an object would only require one operation overall. Furthermore, it is more natural, because the user is modelling the environment as he would in real life. Considering that, we chose this option. It is not without disadvantages, though. It is a lot more difficult to place objects at a precise position using the mouse, so it is more convenient to implement a way around this. One must remember that the 3D window is actually just a projection of the 3D world onto a 2D plane. So to interact with objects using the mouse, we must be able to transform 2D mouse pointer coordinates into a ray in our 3D space, and then intersect it with the object mesh. This is called picking and will be described more closely later (see chapter 2.1.4).

User interface layout is also a part of the window design and appearance. The user interface is split into three parts: a menu bar on top of the screen, an objects bar on the bottom of the screen and a context menu, which is only displayed near the mouse pointer when activated. Figure 1.1 shows the application window.

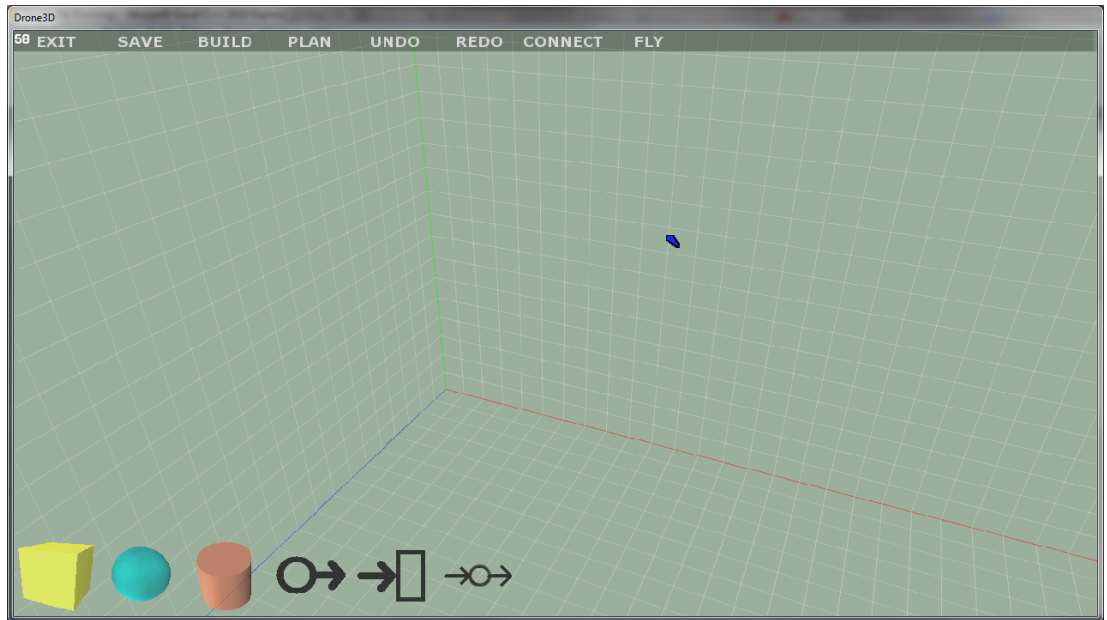


Figure 1.1: Window appearance

### 1.3 Related applications

There are many 3D editors available on the market, be it commercial or non-commercial solutions. At the lower end of the range are modelling applications designed for creation of single 3D models, scenes and animations. Namely, such applications include, for example, **3DS Max** [1] by **Autodesk**, **Google SketchUp** [2] and **Blender** [3]. On the other end of the spectrum, there are complete game engines with tools, capable of not only editing scenes but also scripting and implementing artificial intelligence solutions. These include the famous **CryEngine Software Development Kit** [4], **Unreal Engine 4** [5] and many other sets of tools.

We can look up to those solutions for inspiration, however, they are very advanced and we cannot hope to match their level of quality and feature-richness.

Our editor is differentiated from those applications in multiple ways. The editor does not support detailed modelling of single objects, rather is intended for modelling of rooms and other realistic 3D environments with simple objects. The user places pre-defined objects into the environment and can resize and rotate them. More complex objects can be created by grouping several simple objects together.

Furthermore, the entire editor is implemented with the support of the 3D graphics API. In comparison with the tools mentioned above, this mostly applies to the user interface of the application. The editor is also equipped with a path-planning system. We are not aware of modelling software with such features. Of course, game engines implement path-planning as part of their artificial intelligence solutions. Quadrocopter navigation also sets the application apart.

A related quadrocopter project is, for example, *ardrone\_autonomy* (see [6]). However, we are not aware of this project implementing an environment editor or a path-planning system. Multiple programs for manual quadrocopter user control are also available.

## 1.4 Supported objects

The editor supports three types of base objects: A cube, a cylinder and a sphere. These objects can be transformed into more general variants by scaling. The cube can be scaled into a rectangular cuboid, the sphere into an ellipsoid and the cylinder can be transformed into an elliptical cylinder (a cylinder with elliptical base).

More complex objects can be created by grouping multiple simple objects together. The following figure 1.2 shows a table and a chair modelled in the editor.

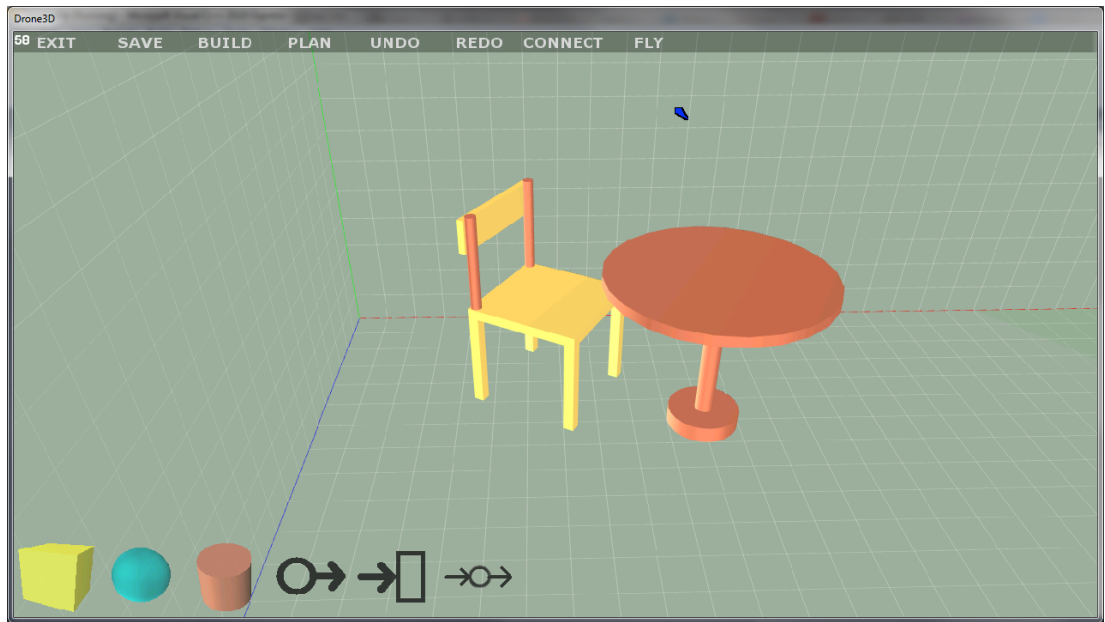


Figure 1.2: A table and a chair

## 1.5 Program functions

As we have already stated, the editor is intended for modelling of rooms or other 3D environments. The environment model is essentially a 3D map of objects. The environment model can then be used to perform path-planning and ultimately the path can be used to navigate a real quadrocopter within the modelled environment.

First, the environment must be created. That is done by setting its dimensions. This process creates three walls, which form the boundaries of the environment. Then, objects can be placed inside.

Objects can be positioned by moving them with the mouse, setting a precise position or by setting their offset. Offset is the distance of the object from the three boundary walls. Objects can also be scaled and rotated to properly represent their real-life counterparts. Furthermore, objects can form groups. A group can then be moved and rotated as a whole. Groups cannot be scaled as it is not clear how this operation should behave.

To help the user correct mistakes easily, undo and redo functions are implemented. They enhance the user experience greatly.

The modelled environment can be saved into a file to save progress. The environment model is essentially a map. Usually, environments of scenes in computer games are also called maps. The file can later be loaded for further editing and use.

Path-planning requires a starting location and a goal location to be set. This can be done by placing special objects into the environment. If a path is found, it is then displayed as connected line segments.

The path can then be used to navigate a quadrocopter. Alternatively, waypoints can be placed in the environment, which allows for the path to be set manually, skipping the path-planning process.

## 1.6 Use case

Let us present an example of using the program. We will model a room 4 m by 4 m, 250 cm tall. In the room is a table and a bed. The quadrocopter *start* location will be placed on the table and the *goal* location will be on the ground next to the bed. Buttons are displayed in brackets: [button].

After clicking the [NEW MAP] button in the main menu, the room dimensions are set (in centimetres). The scale parameter is left at its default value. After confirming the data by pressing the [OK] button, the editing process can start. It will be necessary to move the camera, which is done using the well-known WASD combo. The [R] and [F] move the camera up and down, respectively.

The model of the bed is a simple box. A cube is placed into the environment by clicking the [CUBE] button (first button from the left in the object bar). If the cube is not selected, it should be selected by clicking it with the [left mouse button]. The context menu is open by clicking the [right mouse button]. The [Resize] button is clicked and the cube is resized to 200 cm length, 80 cm width and 50 cm height to match the dimensions of the bed. The bed is on the ground and is 10 cm far from the rear wall of the room and 20 cm far from the left wall.

To position it in the program, the [Offset] button in the context menu is clicked, and the parameters are set. The pairing of walls in the application with the walls of the real room is up to the user. We consider the  $X$  offset to be the distance from the left wall and the  $Z$  offset the distance from the rear wall. The  $Y$  offset is always the distance from the floor to the object. The model of the bed is shown in figure 1.3.

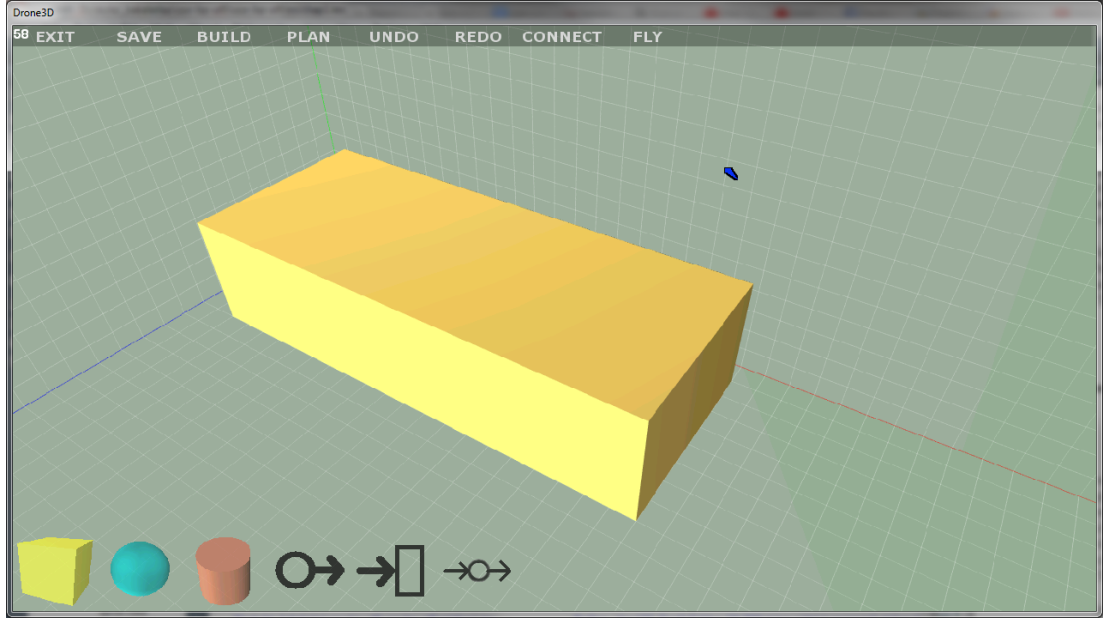


Figure 1.3: Model of the bed

The model of the table consists of three cylinders. The base is a cylinder 20 cm in diameter and 4 cm tall. The board of the table is supported by a single column 5 cm in diameter and 70 cm tall. The top of the table is a 3 cm thick elliptical board 50 cm wide and 70 cm long.

The table can be constructed at an arbitrary location. The three cylinders must be placed so that their centres align vertically. The [Position] button in the context menu provides the necessary functionality to place the object centre at a given location. Place the base of the table 40 cm from the left wall ( $X$  axis), 2 cm from the floor ( $Y$  axis) and 200 cm from the rear wall ( $Z$  axis). Then the base is resized using the context menu [Resize] button. The supporting column is scaled and positioned in a similar way, only at 39 cm height following its own dimensions. The board of the table is also scaled and positioned similarly at 74 cm height. The board and the column slightly overlap in the model, however, this does not cause any problems.

All the three cylinders are selected and a group is created by clicking the [Group] button in the context menu. Then the table can be manipulated as a single object. After clicking the [Move] button in the context menu, it is placed into position using the mouse. The drawn grid can be used to estimate the position of the table.

To be able to run the path-planning algorithm and quadcopter navigation, the *start* and *goal* locations must be set. There should always be at least 150 cm of free space above the *start* and *goal* locations to make sure the aircraft can safely take off and land. After clicking the [START] button (fourth button from the left

in the object bar) and then clicking the top of the table, the *start* location is set. The *goal* location is placed in the same way after clicking the [GOAL] button (to the right of the [START] button). The *start direction* is set by selecting the start location object and clicking the [Direction] button in the context menu.

By clicking the [BUILD] button in the menu bar the environment will be build for the path-planning algorithm. The blocked environment units are displayed as black cubes. This display can be turned off by pressing the [Z] button on the keyboard. The path-planning algorithm is run by clicking the [PLAN] button in the menu bar. A path is found and displayed as a purple line.

Figure 1.4 shows the finished model.

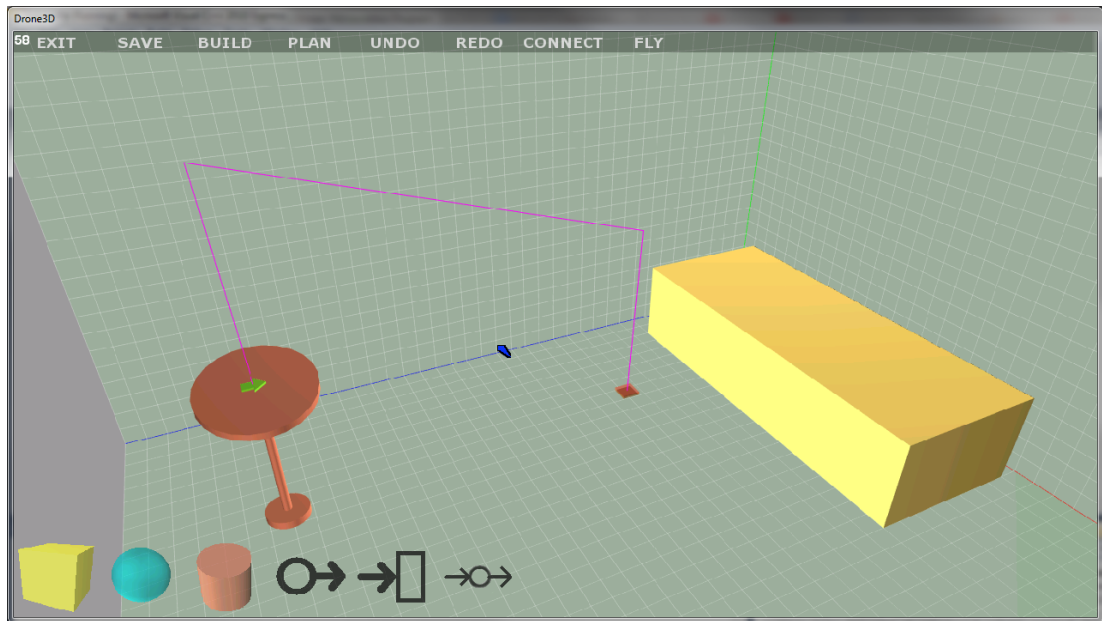


Figure 1.4: The finished model, also displaying the path

To connect to the quadrocopter, the [CONNECT] button is clicked. Before the application can connect to the aircraft, the computer must first be connected to it over Wi-Fi in the operating system. The quadrocopter should be placed and orientated to match the *start* location and *start direction*. Pressing the [FLY] button initiates the navigation and the quadrocopter starts flying.

Further information about the functions of the program and their use can be found in the attached user guide. There also is a stand-alone version of the user guide on the attached Disc.

## 2. Chosen technologies and solutions

In this chapter, we discuss the solutions and technologies we decided to use when implementing the editor.

### 2.1 API for 3D graphics

When choosing the API for 3D graphics, there are many options. One can choose either one of the dominant low-level DirectX [7] and OpenGL [8] APIs, or use a higher-level API like OGRE [9]. Actually, OGRE is a game engine, not just an API. Out of these, DirectX was chosen. First, it not only provides an API for 3D graphics, but also for input devices and other hardware. Second, it is the most widely used 3D graphics API, especially in game industry. That is why it was chosen over OpenGL. DirectX also defines its own standard for files defining 3D models, and has built-in methods to load models from such files. This made the process of making basic geometric objects for the application very easy. These were created in Google Sketchup with 3D Rad plug-in, both of which are available for free. DirectX and OpenGL are very similar and essentially have the same features, so we chose the more popular one of these two.

Using OGRE or any other engine would only add an unnecessary layer to the application framework, that would also add a lot of extra code. And since we were aiming for simple visuals in the editor, we saw DirectX as a better choice. OGRE is a great tool but not suitable for our project because it is simply too big and meant for projects with a much higher emphasis on graphics quality. OGRE would be more suitable for a game project with a lot of graphical content.

However, choosing DirectX has one seemingly major downside — limited portability. We did not consider this to be a big issue, though, because one can simply run a virtual machine or use a compatibility layer to run a Windows application on other systems.

#### 2.1.1 DirectX and Windows game basics

Using DirectX to develop games and other real-time applications is usually connected with the use of WIN32 API (Windows API). WIN32 API is a set of libraries for programming of Windows applications.

An important part of DirectX is the DirectX device. One can think of it as a layer between the application and a computer video card, and it provides video card access to the application. In the core, our application is very similar to a game. At first it is initialized, and then runs in an infinite loop. In this loop it either handles basic Windows messages or renders a new frame. The loop is called **message loop**.

Windows messages are part of the operating system and serve a purpose of transferring information between windows and from the system to a window (like user inputs). At the heart of this system is the message queue.

In a real-time application, an appropriate method of extracting messages from the message queue must be chosen. The WIN32 API provides two methods to extract messages from this queue. *GetMessage* and *PeekMessage*, and they differ greatly in their behaviour. *GetMessage* tries to retrieve a message from the queue and if there is none, it then waits until a message appears. This would cause the application window to refresh only when it receives a message, which is not real-time.

A real-time application must use the *PeekMessage* method, which does not wait for messages. So if the application receives a message, the message is processed. Otherwise a new frame is rendered by calling the *update* method of the application, as can be seen in the following figure 2.1.

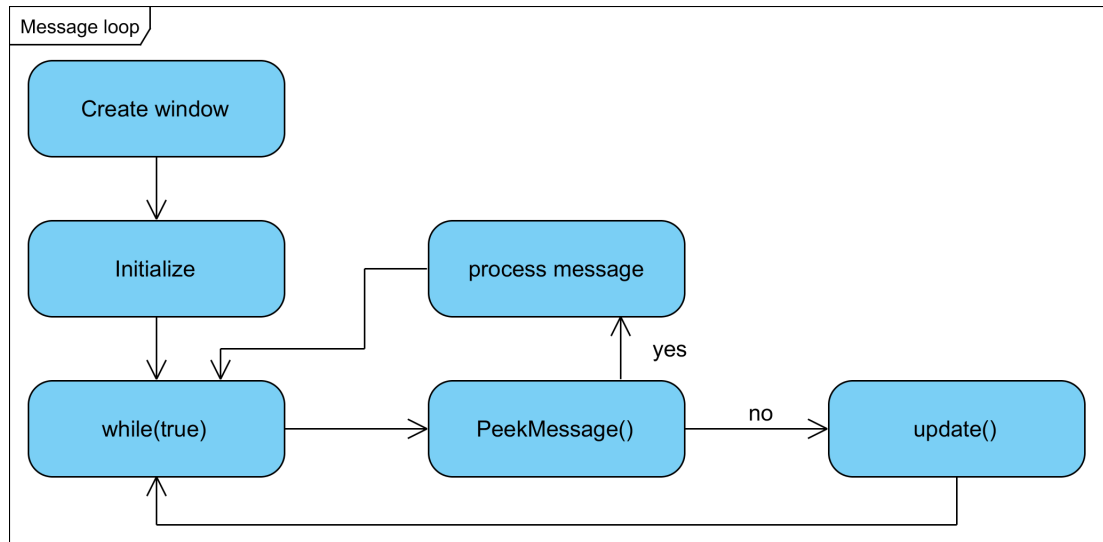


Figure 2.1: Standard game message loop

### 2.1.2 DirectX 3D model representation

3D models can be represented in multiple ways. One of the very common ways is the polygonal mesh representation (see [17]). In this representation, the 3D model is a mesh of polygons. DirectX also uses this representation.

Polygons (usually triangles) are composed of vertices. The polygons then define faces and faces together form the entire surface of the model. For the purposes of lighting, each vertex also has a normal vector. The normal vector determines how light reflects off the surface of the model.

DirectX defines a file format (**X file**, extension *.x*) to store 3D model data using the polygonal mesh representation. The file contains the vertex data: A list of vertex coordinates, vertex normal vectors and a list of polygons. Vertex colours are also stored in the file, and optionally texture coordinates and texture file names.

### 2.1.3 DirectX transformation pipeline

The modelled environment is viewed through a camera, which defines a point of view within the environment. The image that the camera can "see" is rendered



on the screen.

For a model to be rendered on the screen, it must be transformed multiple times. Before we explain the transformations, let us first introduce the spaces the model is gradually transformed into.

**Model space:** Is a 3D space with origin and three orthonormal vectors forming its basis. The model vertices and normal vectors are defined within this space.

**World space:** Is a 3D space like model space. However, unlike model space, it contains all the models. The camera is also positioned within the world space.

**Camera space:** Is again a 3D space like model and world space. In camera space, the origin coincides with the camera position and the  $Z$  axis is pointing in the direction the camera is looking.

**Projection space:** Is a 3D space of a cuboid shape.

**Viewport:** Is a 2D space, the rendering target to which the models are transformed. It is a part of the application window.

When a model is loaded from an **X file**, its vertex coordinates are in the model space. The data has to go all the way through the DirectX transformation pipeline in order to appear on screen. The transformation pipeline is illustrated in figure 2.2.

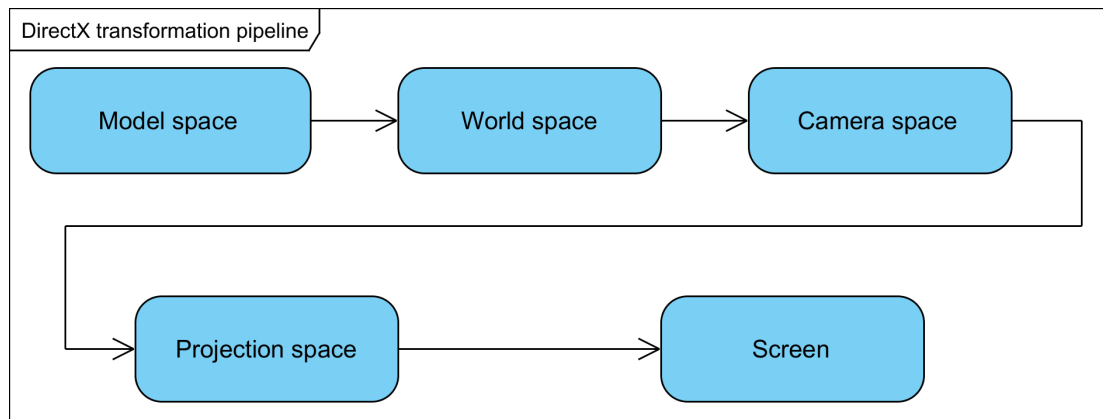


Figure 2.2: DirectX transformation pipeline

First, the model has to be transformed from the model space to the world space. This transformation is composed of three transformations — model scaling, rotation and translation. It is represented by a **world matrix**. To obtain this matrix, the model scale, orientation and position within the world must be set. A temporary matrix is computed for each of the 3 transformations, and then these matrices are multiplied to obtain the **world matrix**. This matrix is unique to each model.

Then the transformation from the world space to the camera space follows. The camera position and orientation within the world are used to compute a **view matrix**. This matrix is common to all models.

Finally, the transformation from the view space to the projection space is performed. It is represented by the **projection matrix**, which can be created using a built-in DirectX method and can use various coordinate systems. DirectX naturally uses a left-hand coordinate system, which is a Cartesian coordinate system. When you extend your left arm, bend your fingers upwards and extend your thumb. Your hand shows the direction of the  $X$  axis, your fingers show the direction of the  $Y$  axis and your thumb shows the direction of the  $Z$  axis.

In our editor, we use the method which creates a **projection matrix** for a left hand coordinate system. Perspective projection is used, which makes objects close to the camera appear bigger than objects far from the camera for a natural result. This matrix is again common for all models.

The **projection matrix** defines the viewing frustum. That is a volume in world space, in which objects (or their parts) are visible. In perspective projection, this volume is a cuboid. It can be visualized as a pyramid intersected by the parallel front and rear (back) clipping planes. The clipping planes are two planes at a set distance from the camera. The direction the camera is looking is a vector perpendicular to both these planes. If an object is closer to the camera than the front clipping plane, it is not rendered (clipped). Analogically, an object further from the camera than the rear clipping plane is also clipped. An illustration of the viewing frustum can be seen in figure 2.3.

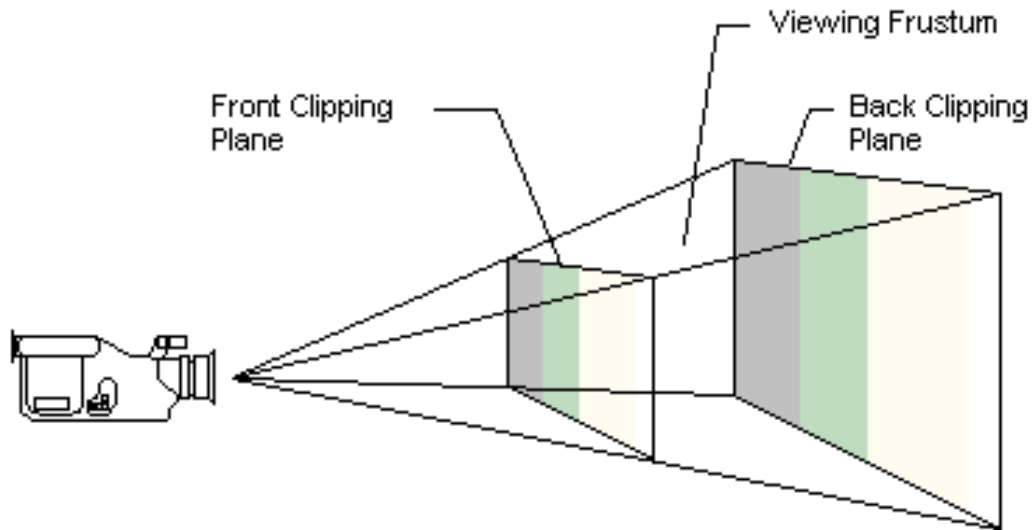


Figure 2.3: The viewing frustum  
Source: Microsoft Developer Network, [12]

The volume within the pyramid and between those planes is the viewing frustum. The objects in the viewing frustum are projected onto a 2D plane—the **viewport**. The **viewport** is the target area for rendering. It is a rectangular part of the application window, and it also defines depth range, into which the scene is rendered. This range is most often from  $MinZ = 0$  to  $MaxZ = 1.0$  and is not particularly interesting for us as we do not modify it or use it in any way.

The transformation from the projection space to the viewport is performed automatically. We do not need to compute nor set any matrices. DirectX automatically obtains the parameters of this transformation from the viewport data.

### 2.1.4 Picking

As we stated before, picking is a process of transforming coordinates from the 2D screen into a ray in the 3D space, and intersecting it with meshes of objects. In order to do that, we have to move along the DirectX transformation pipeline backwards. Picking is a key feature of the application and is illustrated in figure 2.4.

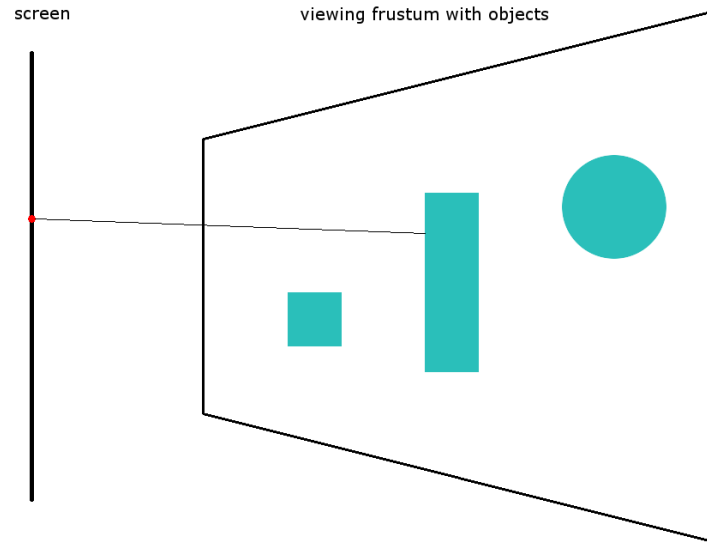


Figure 2.4: Picking illustration

Picking starts with coordinates (usually of the mouse pointer) within the window. Then a line within the viewport is created. The line is defined by two points, both with  $x$  and  $y$  components the corresponding mouse pointer coordinates, and  $z$  component  $MinZ$  and  $MaxZ$ , respectively.

Then both those points are transformed from screen space back to model space using inverse transformation matrices. it can either be done manually or using a built-in DirectX method.

In the editor, we use the build-in method *D3DXVec3Unproject*. The advantage of using the built-in method is that we save time and the method may be more optimized than our solution would be. This method takes the point, viewport, and the projection, view and world transformation matrices as parameters. It outputs the coordinates of the point in model space. On a side note, if coordinates in the world space are desired, the world matrix can be set to *identity matrix*.

Finally another another built-in DirectX method is used to compute the precise coordinates of the intersection of this now transformed ray and the model mesh. The method also returns a boolean value signifying whether or not the ray intersects the mesh.

In our application, object selection and movement are implemented with the use of picking. If the left mouse button is pressed, we perform picking with all objects. From the objects that intersect the ray, we choose the one closest to the camera, as that is the object in the foreground and the object the user intended to select or move.

In case the object is being moved by the mouse, the movement is smooth. The position of the object is updated every frame. The movement vector is computed as the difference between the intersection coordinates of two successive picking operations (one from the previous frame and one from the current frame). Between two successive frames, the user can only move the mouse by a little, thus the object also moves only slightly. Note that to move the object using the mouse, the user must hold the left mouse button. When the button is released, picking is not performed, thus the object does not move.

Picking is also used when new objects are being placed in the environment. It is performed with already existing objects and also with the planes representing environment boundaries. The intersection coordinates then determine the position of the new object.

## 2.2 User interface concept

The user interface of the editor consists of buttons. Usually, buttons in applications are implemented as standard Windows buttons. However, we wanted to try something different. We chose the approach game developers take — designing and implementing custom buttons using the 3D graphics API (DirectX). We took advantage of the fact that DirectX is more than only a 3D graphics API; it also provides an API which encapsulates input devices, DirectInput. The application also implements a custom cursor.

A button has an image which is rendered on the screen. Each button has a unique identifier and buttons also support highlighting.

The main advantage of this approach is that no further libraries are required. Furthermore, thanks to using DirectInput and the application being real-time, the user interface is very smooth and responsive.

However, this approach has one major disadvantage. It takes a lot more time and effort to develop custom buttons than to use a library or Windows buttons.

### 2.2.1 Input processing

To process inputs, an input handler is required. The input handler should communicate with the input devices and the GUI (Graphical User Interface) to process the inputs and control other components of the application accordingly.

The input devices are encapsulated by DirectInput. DirectInput obtains the state of keyboard keys (up or down) and mouse buttons (again up or down). It also obtains mouse movement speed, which is used to move the mouse cursor.

After the state of input devices is obtained, mouse parameters are sent to the GUI component to determine whether a button was clicked. An identifier of the button is returned to the input handler. Finally, the input handler processes the keyboard inputs and the clicked button.

However, certain inputs are more complex than pushing a button. For example, setting the dimensions of an object requires three values to be set by the user. Such inputs are obtained with the use of Windows dialogs.

### 2.2.2 Undo and redo

To provide undo and redo functions, the application must keep track of performed operations. Each of the operations must be somehow encapsulated within an action which also contains data about what the operation performs, so that it is possible to undo it.

To implement undo and redo, we chose the Command pattern (see [13]). An alternative we were considering was the Memento pattern (see [14]). We found the Command pattern to be superior to the Memento pattern for undo and redo implementation. The Memento pattern stores information about the state (essentially checkpoints), thus it can be very memory intensive.

The Command pattern provides much more flexibility by encapsulating operations into commands. Each command stores information about the change it performs — the affected objects and change of their parameters.

We store the commands in a stack-like structure. Thus, undone commands are lost when a new command is created. It would be possible to store the command in a tree-like structure to keep the entire history, however, we considered it unnecessary.

## 2.3 Saving the environment into a map file

Modelling complex environments may take a long time. Thus, it is necessary to provide a way of saving the work, so that it can be further edited later.

The file should contain all the data necessary to reconstruct the environment, when it is loaded. Essentially, three kinds of information should be saved in the file:

- Parameters of the environment (dimensions)
- Models with their parameters
- Groups and their parameters

We have two choices of storing the data. Either use an XML file or a simple text file with custom format.

We chose to store the data as a text file with custom format. The file is a list of all the parameters. First, the environment dimensions are listed. Then the model data and group data follow.

Using XML would require another library to parse the files. As the structure of the saved data is very simple, we considered XML to be unnecessary. XML would be more suited for storage of more structured data.

### 3. Editor implementation

Here we are going to take a look at how the editor is implemented. We use WIN 32 API quite extensively. Since the application is real-time, it runs in a loop. We can think of this loop as having three parts: input processing, transformations computing and image rendering. When the application is controlling a drone, it also receives data from the drone and sends instructions to it during this loop.

To process inputs, we take advantage of DirectInput and we also create custom buttons using DirectX. More complex inputs are obtained with the use of Windows dialogs. The concept of the user interface is described in chapter 2.2.

Windows dialogs run in a separate window and thread, so in order for our application to not process the inputs in these windows, the loop must be paused. Otherwise, DirectInput would register key and button states.

The editor consists of multiple components and each component comprises several classes. In the following figure 3.1 we present the Component diagram. The entire class diagram is on the attached Disc.

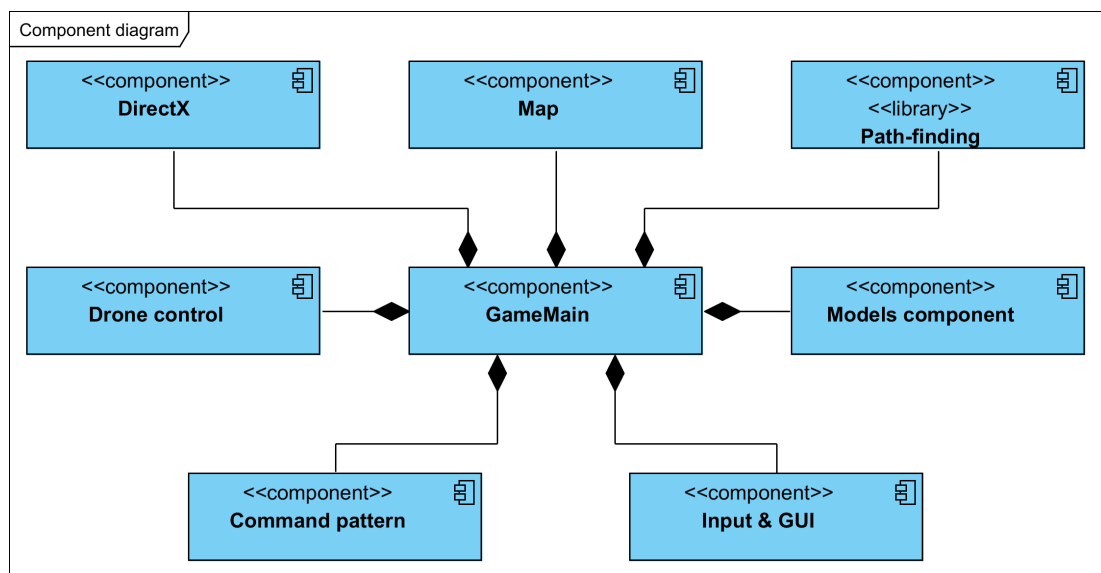


Figure 3.1: Component diagram

- **GameMain** component is in fact a single class, which exists only in a single instance. It serves the purpose of connecting all the other components together (as can be seen in the diagram).
- **DirectX** component comprises of classes that implement core DirectX functionality. It consists of a class to manage DirectX itself, a lighting manager, a text renderer, and the camera.
- **Map** component contains only two classes. One class encapsulates file operations—map loading and saving, and the other contains information about the map (room dimensions and scale) and is responsible for rendering the walls and a grid which serve as guidelines for the user.

- **Models** component encapsulates 3D models and their groups. It has three classes. One class representing a 3D model, then a class defining a group of models, and finally a wrapping class which encapsulates all the models and groups in containers and implements many access methods and operations.
- **Input & GUI** component contains multiple classes. There is the input manager, which encapsulates input devices. Then there is a class responsible for dialog windows and obtaining input data from them. And finally, the GUI classes: a class for 2D image rendering, a button class and a wrapper which encapsulates all GUI elements in containers.
- **Command pattern** component implements undo and redo functionality. It contains an abstract command class, from which the non-abstract specific command classes are inherited. These commands are held in a command stack within a wrapper class.
- **Path-finding** component implements the path-finding algorithm and related features like the representation of the environment required by the algorithm. The implementation of this component is described in chapter 4.3.
- **Drone control** component implements network communication between the application and movement methods for drone control. It also implements the algorithm to navigate the drone along the path provided by the path-planning component or defined by waypoints. This component is described in chapter 5.

## 3.1 DirectX component

The DirectX component implements classes to handle DirectX objects at the core of our application, and our camera. The main task of this component is to provide access to the computer video card.

### 3.1.1 Basic DirectX classes

In order to use DirectX for rendering, we first need to create a DirectX device. To do that, we use a DirectX manager class, **dxMgr**. This class also sets the parameters of the viewport we mentioned in section 2.1.3.

The **dxText** class is very simple and lets us render text.

The **LightManager** class is a container of lights, which provide lighting for the scene. Without them no objects would be visible. A light is specified by its position in the environment, its colour and range. Lights are not physical objects in our application and they do not correspond to any objects in the real world. They purely serve the purpose of illuminating the scene for rendering.

### 3.1.2 Camera class - scene view

The application implements a first person camera, which can be moved around and rotated. The camera is defined by the following parameters:

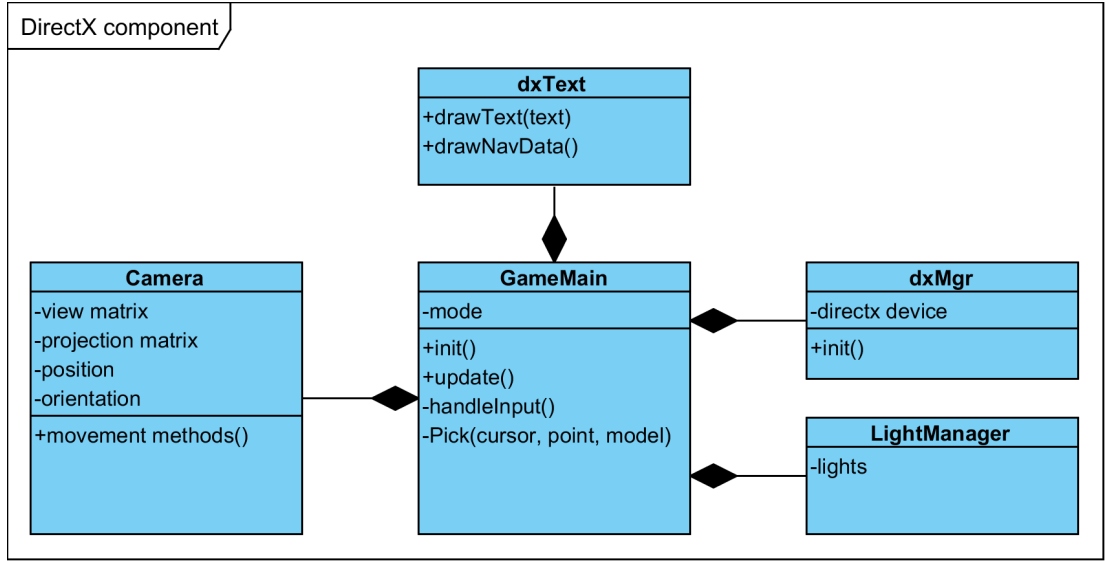


Figure 3.2: DirectX component class diagram

- **camera position:** a three-dimensional vector describing the position of the camera in the world
- **camera angles:** three floating point numbers, *roll*, *pitch* and *yaw*, representing the camera rotation around the *z*, *x* and *y* axes of the world coordinate system (world space), respectively. We only implement pitch and yaw rotations for our camera, as roll is unnecessary for the application.
- **camera vectors:** a set of three-dimensional vectors *look*, *right* and *up*. They point in the directions of where the camera is looking, where is the camera right and up. These vectors are calculated using the camera angles.

When the camera is first created, it is set to look at the coordinate system origin, and positioned to look along the Z axis of the coordinate system. The projection matrix is initialized using a built-in DirectX method with field of view, aspect ratio, near clipping and far clipping distances as parameters. Then the camera is placed in a more convenient location.

Each time some camera parameters change, the view matrix has to be recalculated. Camera movement is realized by adding the corresponding camera vector (multiplied by a small number) to its position, and then recalculating the view matrix. The camera can move in all three directions.

To calculate the view matrix, we implement our own method. We do not use DirectX built-in methods this time. We start with the world coordinate system unit vectors. We set the *look* vector to be (0, 0, 1), the *up* vector to be (0, 1, 0), and the *right* vector to be (1, 0, 0). Then we transform these vectors using the current camera angles, and we obtain the camera vectors. Finally, we fill the view matrix. It looks like this:

$$\begin{pmatrix} right.x & look.x & up.x & 0 \\ right.y & look.y & up.y & 0 \\ right.z & look.z & up.z & 0 \\ dot(right, position) & dot(look, position) & dot(up, position) & 1 \end{pmatrix}$$



The matrix is of size 4x4 and at first we set it to identity. This makes the last column (0,0,0,1), and we leave this column that way. The first column is then set to be the camera *right* vector and dot product (also called scalar product) of this vector and the camera *position* vector. The other columns are filled in a similar way, as can be seen in the matrix above.

## 3.2 Models component

This component encapsulates all the classes related to the 3D models in the program. There is the **Model** class itself, which represents the objects and their 3D models. Then there is the **ModelGroup** class, which represents a group of models. Operations can then be performed on those groups rather than on each object individually. Finally, the **ModelWrap** class implements a wrapper around the models and groups. It stores models and groups in containers and implements many data access methods and operations.

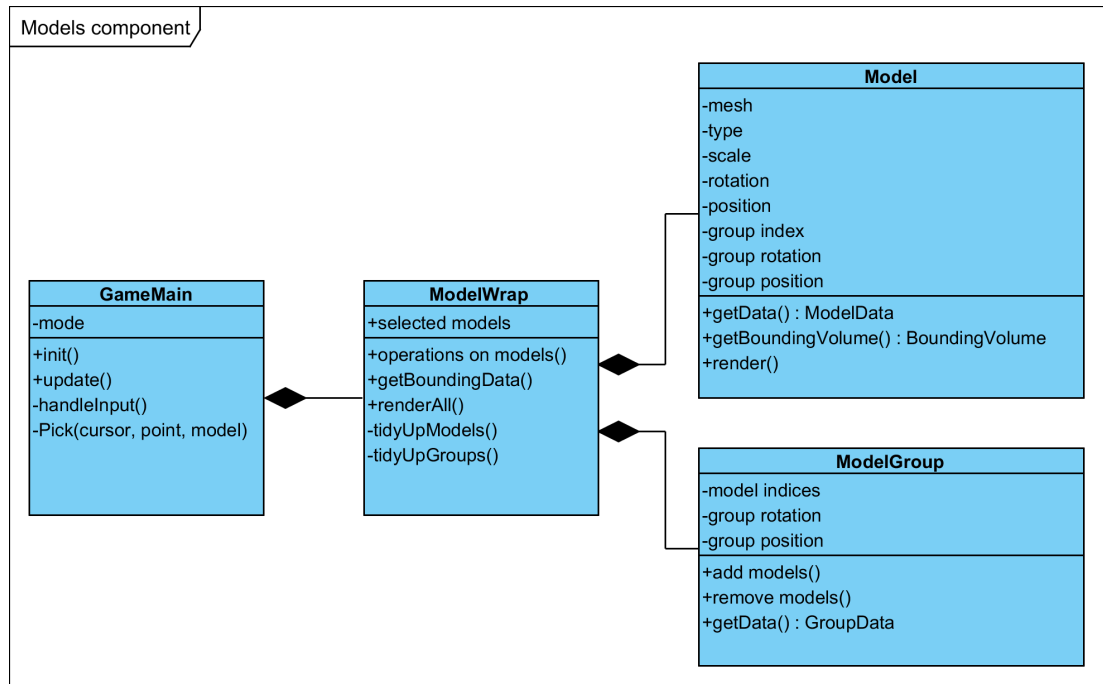


Figure 3.3: Models component class diagram

### 3.2.1 Model class

The **Model** class provides functionality for the 3D model loading and drawing. The class contains the following data:

- **model mesh:** the data loaded from the **X file**. This defines the shape of the model. Multiple models with the same mesh do not share the data — each model has its own copy of the mesh. The model also carries its type, which is tightly connected with the mesh. It serves as an identifier, and it

determines which mesh should be loaded when a new object is being added, for example.

- **model transformation data:** consist of the model transformation vectors and their respective matrices. The transformation vectors define the *scale*, *rotation* and *position* of the model.
- **group transformation data:** Also consist of transformation vectors and their matrices. However, they define the transformation of the whole group. These are represented by the *groupPosition* and *groupRotation* vectors and they are set only if the model belongs to a group.
- **group identifier:** an index into the group container in **ModelWrap** class, also signifying whether or not the model belongs into a group.

The *render* method of **Model** class uses the model transformation data and, if applicable, group transformation data to transform the mesh from model space to world space. It is then presented to the DirectX device, which renders it on the screen. The class also provides a method to highlight the model. In this case, a specific *scale* vector is created and substitutes the original *scale* vector. The transformations are applied to the mesh in the exact same way as during standard rendering, only using this new *scale* vector. Then it is presented to the DirectX device in wire-frame mode and with black colour. This results in a highlighting overlay as seen in the image below. Model highlighting gives us the ability to show the user which models are currently selected.

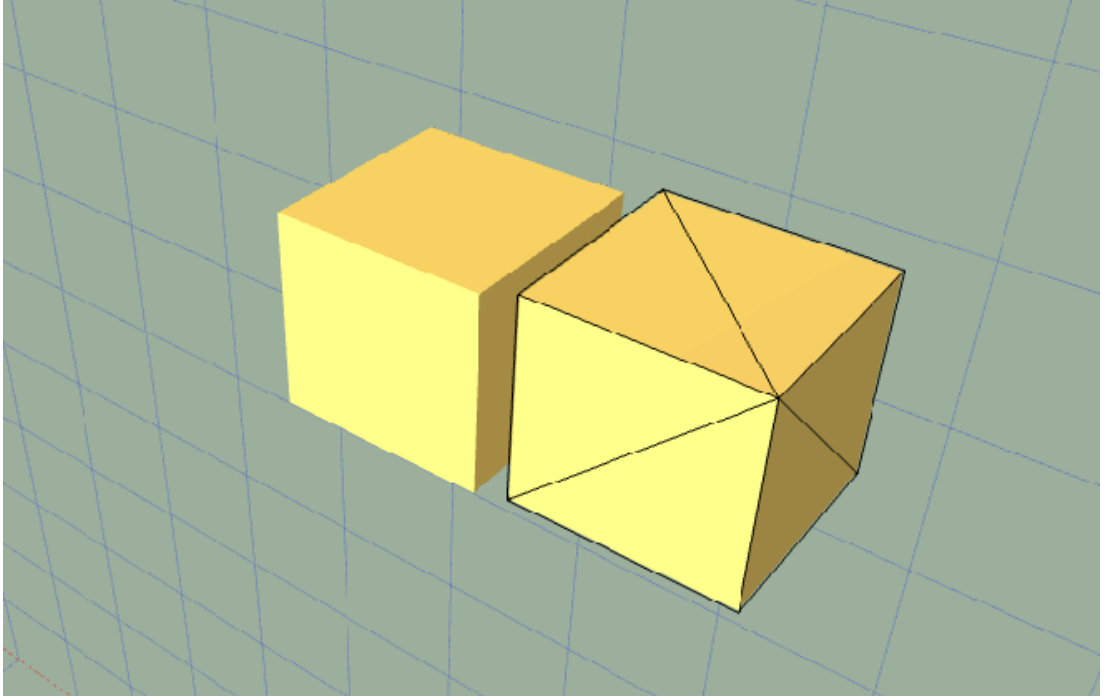


Figure 3.4: Selected model is highlighted

The models represent objects which are in the role of obstacles in the project. This is why we must be able to transfer information about models into the environment representation of the path-finding component. To do that, the **Model**

class implements a method which computes a bounding volume of the model. This is a volume which we can describe using a mathematical equation and the entire mesh of the model fits inside it. The fit should be tight. We examine this in more detail in the chapter about path-planning, see chapter 4.1.1.

### 3.2.2 ModelGroup class

This class is only a basic **STL vector** of model indices (into the **ModelWrap** group container), and allows us to work with more models at once using group operations. It also holds the group transformation data — *groupPosition* and *groupRotation* vectors. Group scaling is not supported, because it is hard to determine how this operation should behave. Moreover, while group rotation is a very useful operation, the usefulness of group scaling is questionable. It is worth noting that the class does not perform any computing, it only holds the data and implements methods the **ModelWrap** class can use to retrieve and set the data.

### 3.2.3 Order of transformations

The transformations are performed in this order: model scaling, model rotation, model translation, group rotation, and finally group translation.

Each transformation is represented by a matrix and they are multiplied to obtain the resulting **world matrix**. Each of the transformations is represented by a vector, only the rotation transformations are represented by quaternions. Quaternions are four-dimensional vectors which can represent rotation by an arbitrary angle around an arbitrary axis.

The order in which transformations are performed influences the resulting transformation. The result should be natural, predictable and anticipated by the user.

The model mesh is created in such a way that the centre of the model is identical with the origin of the model's coordinate system (model space). If the world matrix is set to *identity matrix*, the model is then positioned at the world coordinate system origin. Considering the order of transformations, this means that the scaling transformation behaves as if the object was inflated. Thus, scaling does not influence other transformations and is performed first.

The centre of the rotation transformation is in the world coordinate system origin, so it behaves naturally, too. Rotation must be performed after scaling. Otherwise the object scale would change with rotation, which is not desired.

Translation places the object centre at a given point. We must perform rotation and scaling before translation. Otherwise, the resulting transformation would be unnatural, as translation would influence the results of both scaling and rotation. For example, when the user places an object in the environment and then decides to rotate it, he expects it to rotate around its own centre, not around the coordinate system origin.

Group transformations allow for a common transformation of several models at once. A group usually represents a complex object like a chair. However, it is only possible to transform individual models. Thus, the group transformation effect is achieved by placing the models into positions relative to the group centre and then applying group transformations to individual models. The entire group

then rotates naturally around its centre. For the same reason as before, group translation comes after group rotation. Group scaling is not supported.

### 3.2.4 ModelWrap class

**ModelWrap** encapsulates models and groups in containers. It contains an **STL vector** of models and an **STL vector** of groups. In addition, it also has an **STL vector** of currently selected models. It implements the methods for object and group manipulation, which are called as a result of pressing buttons in the application context menu. The **ModelWrap** creates an abstraction layer over all the models and groups to make accessing them from the **GameMain** class easier. For example, instead of "asking" each model to render itself, we "ask" **ModelWrap** to render all the models for us.

**ModelWrap** also implements the logic of creating groups and maintaining the data after a model or group is deleted. Furthermore, it is able to exchange necessary data with other components, like with the **Map component** to save and load the map.

### 3.2.5 Creating a group - maintaining data consistency

The operation of creating a group must fulfil one condition. It must not change the resulting transformations of objects. In other words, when the user creates a group, he cannot be surprised by unpredictable object movement or rotation.

Let us assume that the group is to be created from an arbitrary number of existing groups and single objects (objects that do not belong to any group). A simple algorithm follows:

1. Objects belonging to the selected groups are transformed, so that their transformations are no longer relative to the centre of their group.
2. A **target group** is chosen. It is the selected group which has the minimal index in the **ModelWrap** group container.
3. All selected objects are added to the **target group**.
4. The centre of the target group is computed as the average of positions of all its models.
5. All the selected groups except of the target group are destroyed and deleted from the **ModelWrap** group container.
6. The *group identifiers* of all models are updated.

### 3.2.6 Destroying a group - maintaining data consistency

This operations turns a group of models into single models. The models are not deleted from the scene. Maintaining data consistency when destroying a group is much simpler than when creating one. This operation must also satisfy the condition that it must have no effect on the resulting image.

Step 1 of the algorithm above is applied. Then the selected group is destroyed and deleted from the **ModelWrap** container. Finally, *group identifiers* of all models are updated.

### 3.3 Map component

In this section we will talk about the map file structure and classes that work with the files. This is needed so that the user can save what he has created. The map file is a simple list of map parameters, objects with their parameters, and groups. The data saved about a model are its type and transformation data. A group is saved as a list of model indices and its group transformations, just as it is represented inside the program.

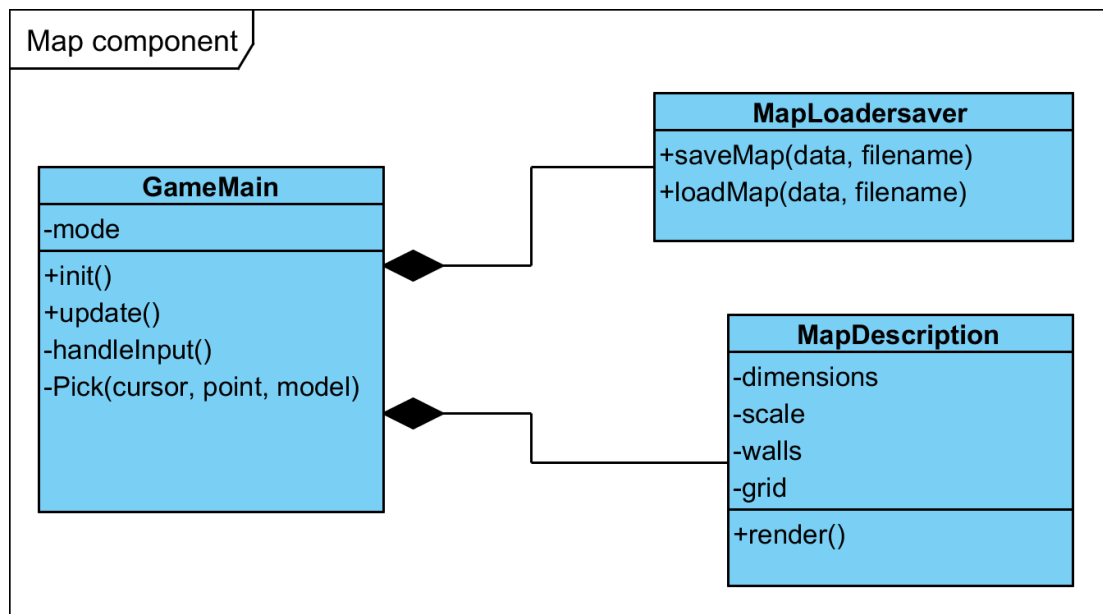


Figure 3.5: Map component class diagram

#### 3.3.1 File structure

The purpose of the file is to store all the necessary information to recreate the modelled environment. It must contain map parameters, a list of models with their parameters and a list of groups and their parameters. We use special tags to separate the different sections of the file. These tags begin with the '#' character. First section is the map parameters section, which begins with **#OPTIONS**, is followed by a list of parameters and their values, and finally ends with **#ENDOOPTIONS**. There are four map parameters: length, width, height and scale. Their structure is **@parameter.value**, where value is in centimetres (a positive integer).

The map parameters are followed by a list of models, which begin with **#MODELS.count**. Count signifies the number of models that follow. Each model is then on a separate line. Each line contains one integer and three sets of floating point numbers and integers separated by spaces. In a set, floating

point numbers form DirectX vectors and quaternions, and integer numbers form equivalent values in centimetres and degrees.

The first integer represents the model type. The first set represents the model *position*: three floating point numbers as the position vector, and three integers as the position in centimetres. The second set represents the model rotation, four floating point numbers for the *rotation* quaternion and three integers for angles representing rotation around the coordinate system axes. Finally, the last set is the model *scale* - three floating point numbers and three integers in a similar fashion to the model *position*.

Then the groups follow. They begin in a similar fashion like models, with **#GROUPS\_count**. Again each group is on one line and it contains an arbitrary number of integers (model indices) and two sets of floating point numbers and integers separated by spaces. The first number on the line is the number of models in this group. The sets represent the group *position* and *rotation* like in the case of a model.

The data in the file are always correct. The floating point values and integer values match, however, in the case of models belonging into groups, the floating point data are relative to the group while integer data are absolute. Code 3.1 shows a sample map file.

```
#OPTIONS
@length 400
@width 400
@height 250
@scale 10
#ENDOPTIONS

#MODELS 3
1 -0.56027 0.00000 -3.13398 68 87 158 0.00000 0.00000 0.00000
  1.00000 0 0 90 1.00000 1.00000 1.00000 10 10 10
3 0.56027 0.00000 3.13398 80 87 96 0.00000 0.00000 0.00000
  1.00000 0 0 90 1.00000 1.00000 1.00000 10 10 10
2 10.00000 5.00000 -10.00000 100 50 100 0.00000 0.00000 0.00000
  1.00000 0 0 0 1.00000 1.00000 1.00000 10 10 10

#GROUPS 1
2 0 1 7.40761 8.73325 -12.79970 74 87 127 0.00000 0.00000 0.70711
  0.70711 0 0 90
```

Code 3.1: A sample map file

### 3.3.2 Classes of the Map component

The component comprises two classes. Class **MapLoaderSaver** implements method for file manipulation — saving the map into a file and loading a map from a file.

The class **MapDescription** holds information about the map — dimensions of the environment and *scale*. *Scale* is an important parameter. Not only does *scale* influence the size objects appear on the screen, but it also provides a connection between the application units (dimensionless floating point number) and real-world length (centimetres). This connection is important for the quadcopter navigation.

## 3.4 User interface implementation

The user interface is implemented using DirectInput [11] and simulated buttons rendered on the screen. We use DirectInput in immediate mode to obtain the state of mouse and keyboard at every frame. This state is then processed by an input-handling method. This method checks the state of pre-defined keys (and mouse buttons) and reacts accordingly. We also implement undo and redo functionality, for which the Command pattern (see[13]) is used. The command pattern creates an abstraction layer over our operations, encapsulating them into commands. The commands are stored in a stack-like data structure with an index pointing at its top (the last valid command). Undo and redo operations then translate into decrementing and incrementing this index by 1, respectively.

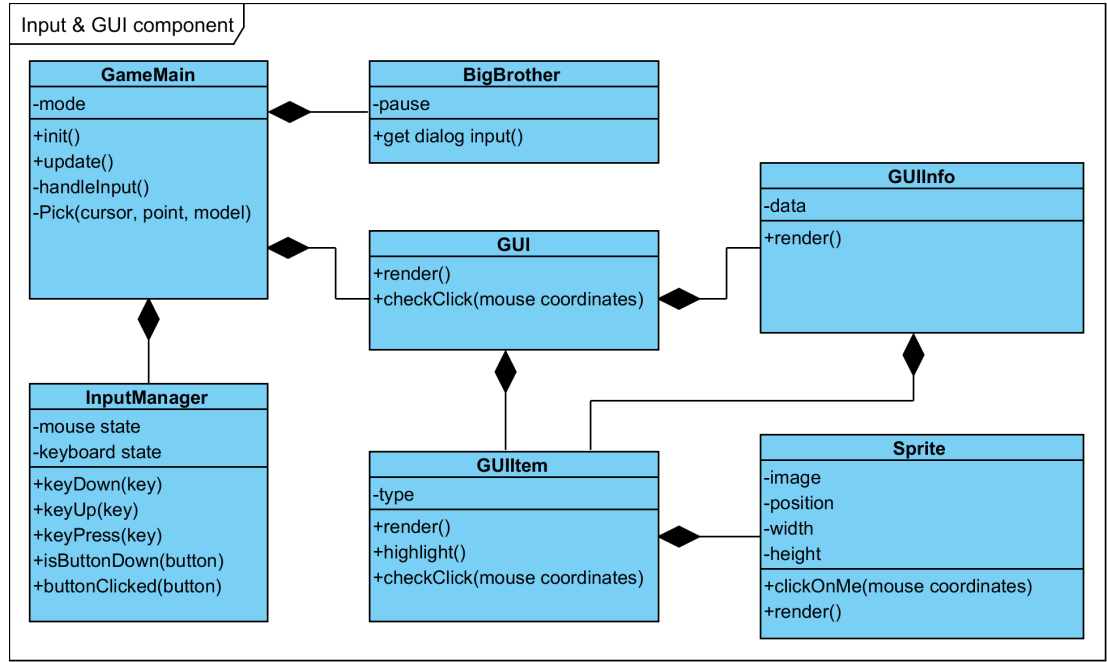


Figure 3.6: User interface component class diagram

### 3.4.1 Layout and interaction

The user interface is split into three elements. The first is the menu bar on top of the screen. This contains buttons dedicated to program control, like saving the created map, running path planning and drone control, and exiting to menu. The second element is the object bar on the bottom of the screen. This contains buttons that let the user place objects into the world. The third element is a context-dependent menu that pops up every time the right mouse button is clicked. This menu contains buttons which allow the user to work with the objects within the world.

The user interface elements are described in more detail in the attached user guide. The guide explains all the button functions and lists all used keyboard keys.

### 3.4.2 Basic classes of the GUI component

The **GUIItem** class is the class implementing buttons. Each button has an image and a unique identifier.

The **GUIInfo** class implements an info panel with a close button. This is used to display information about an object to the user upon request.

The **GUI** class is a wrapper class around the **GUIItem** and **GUIInfo** objects, holding them in containers and providing methods at a higher level of abstraction. It also manages the cursor.

### 3.4.3 Sprite class

The **Sprite** class provides a layer over the DirectX sprite objects and related methods. Sprites are 2D images integrated into a scene and can also be used to draw elements of the user interface. For more information about sprites, see [10]. We use sprites to render the images of buttons in the application. A sprite holds image data with its position and dimensions. Together, they define the area the image occupies. **Sprite** then implements the *clickOnMe* method. Given mouse pointer coordinates, this method determines whether or not the pointer is within the area defined by the sprite. It is only called when the left mouse button is pressed. The purpose of this method is to identify a button that should be highlighted or was clicked. It is called by methods of the **GUI** class, which encapsulates buttons in containers.

The call order is as follows: The input handler (**GameMain**) calls the *checkClick* method of **GUI** class, which then calls the *checkClick* method of each **GUIItem**, which results to a call to the *clickOnMe* method of **Sprite**.

The use of DirectInput proved to have a major disadvantage. The system cursor is not available in the application (a custom cursor is used), thus the application window cannot be moved nor resized with the cursor. It would only be possible using the ALT + SPACEBAR key combination, which most users are not familiar with. Unfortunately, we were unable to solve this problem. Thus, we decided to disable window movement and resizing.

### 3.4.4 BigBrother class

Many of the user inputs are quite complex and Windows dialogs are used to obtain them. One example of such input is setting object dimensions. This requires three parameters to be specified and a dialog window allows us to retrieve them all at once. When a dialog is initiated, our application has to be paused and can only resume after the dialog is closed. That is due to Windows dialogs naturally running in a separate thread. The application is paused to ensure that it does not process any inputs that belong to the dialog window. **BigBrother** lets us pause our application, and it also retrieves the data input by the user in the dialog. When an action that uses a dialog is selected, **GameMain** sets the pause tag in **BigBrother**. While this tag is set, new frames are not rendered. Then **GameMain** calls the *get* function of **BigBrother** with a parameter signifying what dialog should be created. Different operations require different dialogs, so it is necessary to specify, which dialog should be created. **BigBrother** then creates the dialog and stores the data, if some are obtained.



### 3.4.5 How it works

The `DirectInput` interfaces we use are encapsulated within the **InputManager** class. Every frame, **InputManager** obtains the keyboard and mouse state. The keyboard state consists of key states (up or down) and the mouse state consists of button states (again up or down) and mouse movement parameters. Then the input must be handled. The *handleInput* method of class **GameMain** goes over all the controls our program uses, checks their state and performs actions accordingly. Some actions are performed immediately (like camera movement), for undo-able actions a command has to be created first.

Buttons are simulated as a class **GUIItem**. Each button has a unique identifier and an image as an instance of the **Sprite** class. These classes are very similar. One could put the type variable directly into the **Sprite** class but we wanted to separate button functionality from image drawing functionality. A button also implements the *checkClick* method, which only calls the *clickOnMe* method of the sprite to determine whether or not this button was clicked. Furthermore, it implements the *highlight* method that draws a frame around the image area when the mouse pointer is within the image area. This way we can give feedback to the user about which button he is going to click.

All the **GUIItem** elements are encapsulated within the class **GUI**. This class holds all the buttons in an (**STL vector**). Moreover, it manages the cursor. As we stated before, it implements its own *checkClick* method which iterates over all the **GUIItem** instances and returns the identifier of the one which returned true in its *checkClick* method. If none of the items return true, a special identifier is returned, signifying that the mouse pointer was outside of the user interface (no button was clicked). The **GUI** class also manages the already so many times mentioned context menu, creating it and destroying it when needed. The context menu consists of buttons, which we have already described.

### 3.4.6 Undo and redo - the Command pattern

Undo and redo are functions every editor should have. If a user makes a mistake, we want him to be able to fix it as easily as possible, and so we have implemented undo and redo functionality. For that, we used the Command pattern.

The Command pattern creates an abstraction layer over the operations, encapsulating them within commands. Commands store information about the operation, so that it can be undone or executed again. The Command pattern consists of three types of objects: The *invoker*, the *receiver*, and *commands*. There also is a fourth essential element, the *command stack*.

The invoker is responsible for command creation. In our application, the *handleInput* method of **GameMain** class acts as the invoker. Every time the user chooses an operation that is undo-able it creates a new command. Then the invoker "asks" the command to execute itself and puts it on the command stack. Since the invoker is the input-handling method, it is also responsible for calling *undo* and *redo* functions of the command stack object.

There is one class for each type of command, and they are all inherited from the base abstract **Command** class with pure virtual *Execute* and *Undo* methods. Every command stores information about the models and values it is modifying.

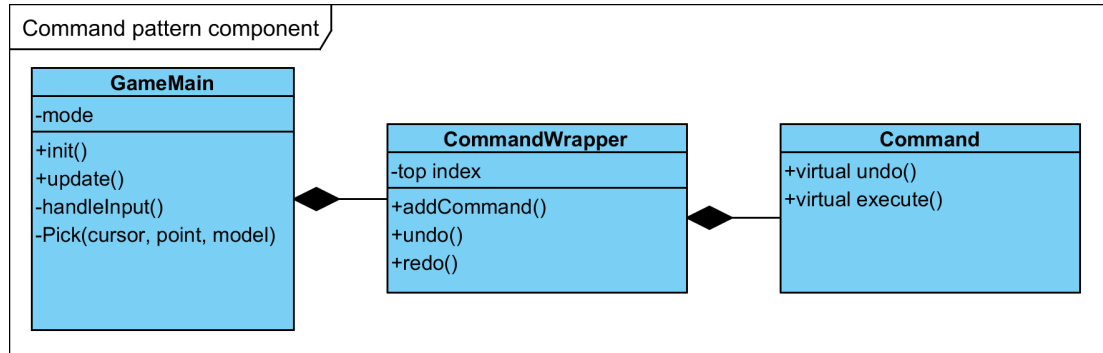


Figure 3.7: Command pattern class diagram

Every command has its own receiver, and the *Execute* and *Undo* methods call corresponding methods of the receiver. In our application, only the **ModelWrap** class acts in the role of the receiver, because only the operations on models are undo-able actions and all operations are implemented within **ModelWrap**.

The command stack is, in fact, not a stack at all. It only simulates stack-like behaviour. Since we also support redo, the command cannot be deleted from the stack when *Undo* is called. That is why the command stack is an **STL vector** of commands with an index that is pointing to the last valid command. When *Redo* is called, this index is increased by 1 and *Execute* is called on the command it points to. When *Undo* is called, the *Undo* method of the command is called, and the index is decreased by 1. When a new command is added, all the commands with a higher index than the last valid command are deleted. In other words, all the undone commands are forgotten, and a new one is inserted at the end. This way we keep the command history "aligned in a line", we do not create a tree of branching histories.

Most commands only store information about the change that happened — the affected models and the original and new parameters. However, the group and ungroup commands are a little more complicated. They store the complete original state of groups. Storing information about the change is not enough in this case, as it is not possible to reconstruct the object groups using that information. The entire state of groups is a memento, thus these commands also combine a small part of the Memento pattern.

### 3.5 GameMain component

This component provides the connection between all the other components. It also connects the components to the application window. It comprises of only a single class — **GameMain**. It is also the base class of our application. If we took away all the other components, our application would still work. However, it would not do anything.

The *update* method we mentioned in section 2.1.1 is implemented within this component. Our application can run in two modes: menu mode and editing mode. Each of the modes has its own special update function responding to different inputs and performing different operations. Depending on which mode the application is in, the corresponding update function is called by the *update*

method.

The menu update function only handles the three main menu buttons, while the editor update function handles a multitude of inputs, be it buttons in the user interface or keyboard keys.

This class also implements picking, which was explained in section 2.1.4. Picking provides the means to select objects and move them using the mouse. The implementation of picking iterates over all the objects within the environment.

## 4. Path planning

We want to navigate a quadrocopter from one location to another. For that, we need to be able to find a path between these locations. In order to do that, we need a way to represent the continuous space in our program. Obviously, that would be complicated at the least. That is why space is represented as a set of discrete units. There will also be objects - obstacles in our environment, so we need a way to represent those, too. We also need an algorithm, which will use our environment representation to find the path from a start location to a goal location. Most path-finding algorithms work on graphs, where edges define the neighbour relationship between vertices. If we manage to define when two units of our environment are neighbours, we can use such algorithms.

### 4.1 Environment representation

Naturally, since we are navigating a quadrotor aircraft, we are working with three-dimensional space. We also need to take into account the freedom of movement of the aircraft. If we were to navigate an object with its movement constrained to a plane (like a wheeled or legged robot), we would consider different options than for an aircraft. To navigate a quadrocopter, we need to represent the complete space because it can potentially go anywhere within our environment. We would also like to treat all the dimensions equally, that is, give the aircraft equal options whether it is moving horizontally or vertically. The condition stems from the quadrocopter itself. It can move in all directions equally, albeit vertical movement usually happens at a lower speed than horizontal movement.

There are many options how one can partition continuous space. In the 2D case, the most basic approach is to create a rectangular grid. The vertices of the graph can then be the midpoints of rectangles or rectangle vertices. In computer games, navigation meshes are often used. Navigation mesh defines an accessible area (surface) as a mesh of polygons. However, navigation meshes are limited to flat surfaces and are not suitable for 3D space representation. An example of a navigation mesh can be seen in figure 4.1. The white rectangles in the image are obstacles. Other possibilities include quadtrees, which divide 2D space into a grid of rectangles of various sizes. This can save space and speed up the search. More possibilities are described, for example, in [20], Chapter 3.3.

In case of 3D space, the situation is a little more complicated. Most of the 2D solutions can be extended to 3D by introducing flight levels, essentially partitioning 3D space into a set of 2D sub-spaces. Grids can also be used to partition the space into a set of units that fill the space without gaps. An interesting option is the use of visibility graphs. An example of a (complex) 2D visibility graph can be seen in figure 4.2. The vertices of the graph are vertices of obstacles and two vertices are connected, when they have line of sight. However, it is not possible to find truly shortest paths on visibility graphs in 3D space because the truly shortest paths do not have to be constrained to the edges of the graph, according to [21]. The article further states, that finding truly shortest paths in 3D environments with polyhedral obstacles is NP-hard, so this approach is unsuitable for us.

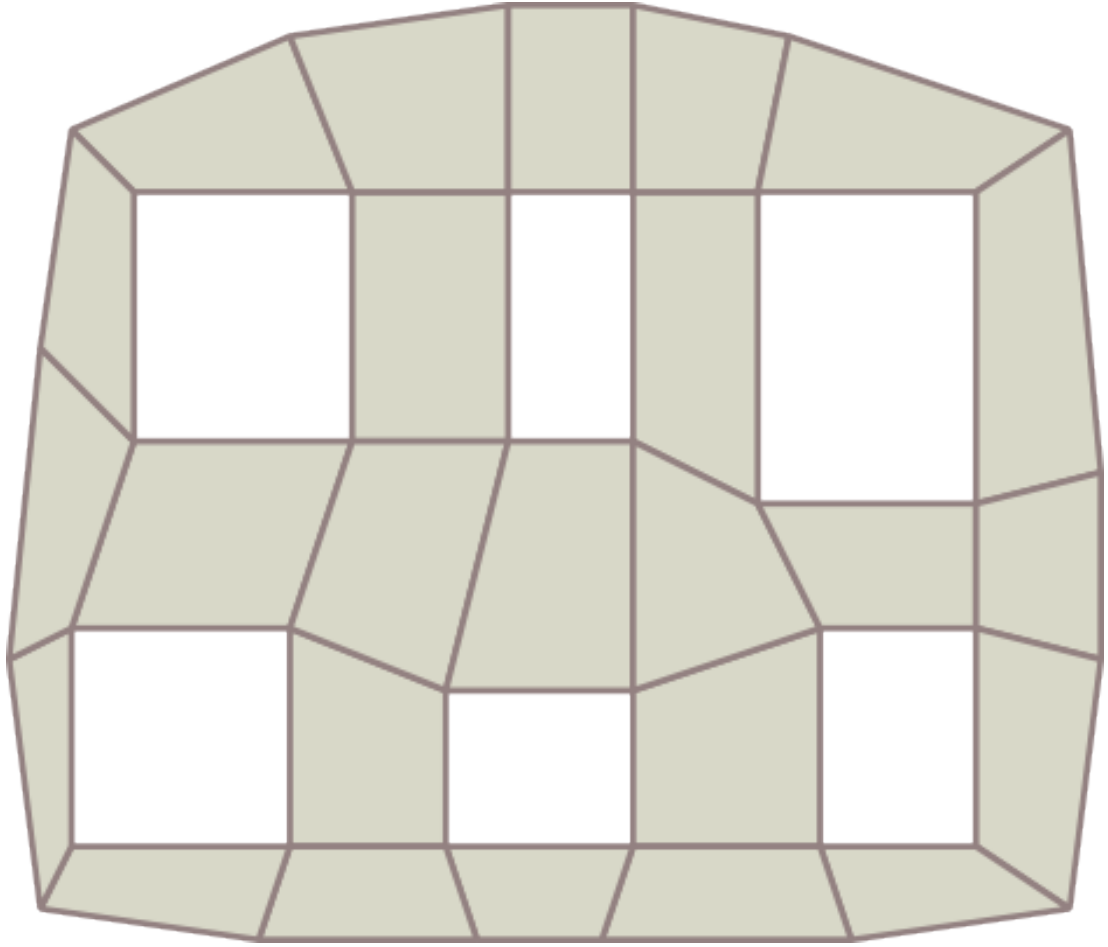


Figure 4.1: A Navigation mesh  
Source: Stanford University, [22]

It is most common to partition the 3D space into cubic grids. This way, reasonable time complexity and short paths can be found. Using this representation, the paths are approximately 13% longer when they are constrained to graph edges than true shortest paths (not constrained to graph edges), according to [21]. This gives us a good reason to choose an algorithm, which does not constrain the paths to graph edges.

### Final choice of representation

In the end, we were considering two options. Partitioning the environment into a cubic grid and using flight levels to partition 3D space into a set of 2D spaces, which we can then partition into grids of 2D units.

We chose the cubic grid approach. That is because we wanted to treat all the dimensions in the same way, and the cube is the only 3D body, that is regular and can fill 3D space without gaps using only translation [16]. These properties exactly dictate the arrangement of the cubes. Furthermore, this approach is very natural and simple. There are other 3D bodies that fulfil our space-filling condition, but they are not regular. So they do not satisfy our condition to treat all the dimensions equally. Furthermore, they can be significantly more complicated in shape than cubes. An example is a hexagonal prism or a truncated

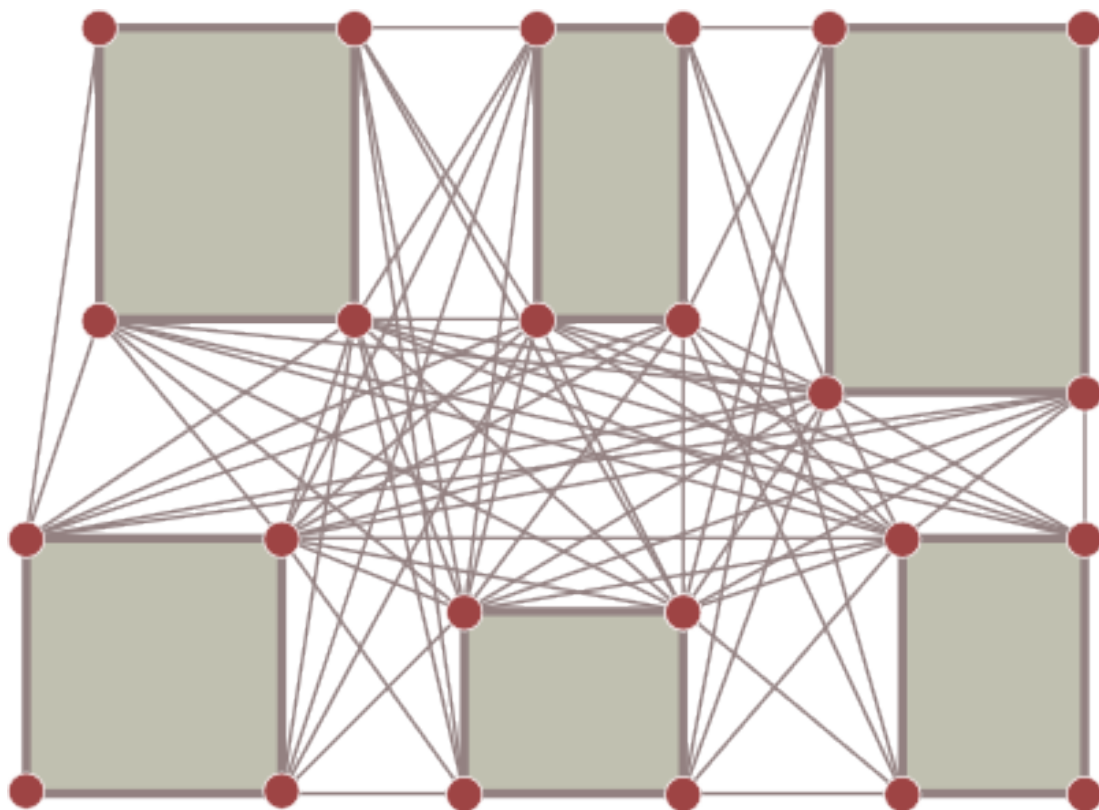


Figure 4.2: A 2D visibility graph  
Source: Stanford University, [22]

octahedron.

Now when we have our environment represented as a set of units, we can represent obstacles as sets of units, too. These units are blocked, so the quadcopter cannot pass through them, and has to fly around. We only need to be able to compute the set of blocked units for each obstacle.

We then define the neighbour relationship, in which two units (cubes) are neighbours when they share a face, an edge or a vertex. This results in a regular 26-neighbour grid of cubes. Every cube is a vertex with its center as coordinates and it is connected to 26 other vertices. Now that we know what is a vertex and when two vertices are neighbours, we have defined a graph.

The second approach is to divide the space into flight levels, and each flight level into polygons like squares or hexagons. This, however, does not fulfil our condition to treat all the dimensions equally. Flight levels are separated differently from one another than cells within one level are, even though there can be some correlation. Using flight levels presents many questions. What should be the correlation between flight levels and cells within one level? We would also have to define when two cells from subsequent flight levels are neighbours. Should it be only the cells with the same horizontal position or also the in-flight-level neighbours of those cells?

Furthermore, if we used squares and equidistant flight levels with distance same as the length of the square side, we might obtain the same result as when partitioning space into a grid of cubes (depending on how the neighbour relationship is defined), only in a more complicated way.

### 4.1.1 Determining blocked cells

Since we represent the environment as a 26-neighbour grid, we need a way to transform the obstacles in the world into a set of blocked cells. There are many ways this can be done, perhaps the first that comes to mind is computing whether the cell(cube) intersects the object. This is quite complicated, as the object can be arbitrarily rotated and may have a complicated mesh (geometric shape). Thus, it is better to encapsulate each object within a bounding volume of simple shape.

The easiest bounding volume to use is a bounding sphere. Then all the cells that intersect the sphere are blocked. Even though this is very efficient and simple, we chose a slightly more complicated approach. Bounding spheres have one major problem — they do not fit tightly around objects of elongated shape. That is why we decided to use bounding ellipsoids. For some objects (a cube for example) they do not offer any benefits over bounding spheres. But in the case of a general rectangular cuboid, an ellipsoid gives a tighter fit. Actually, the more elongated the shape, the bigger the difference between a bounding ellipsoid and a bounding sphere. From our point of view, bounding ellipsoids are the best compromise between simplicity and tight fit. Needless to say, the fit is not perfect. Figure 4.3 presents a comparison between a bounding sphere and a bounding ellipsoid encapsulating the same object.

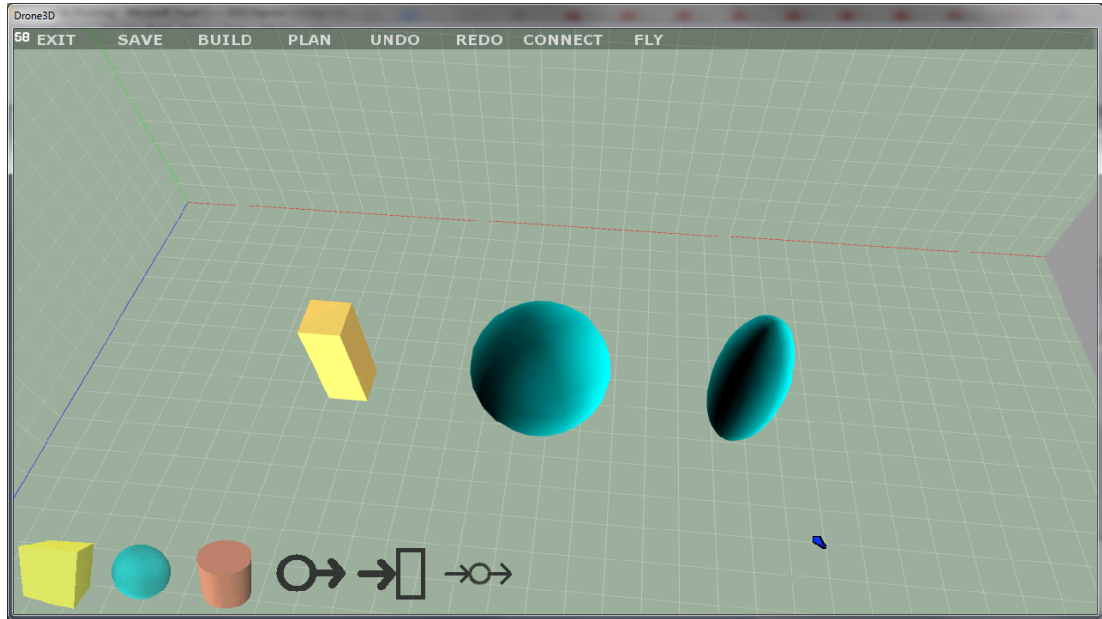


Figure 4.3: A bounding sphere and a bounding ellipsoid

We would like to mention one more way of enclosing geometric bodies. That is using bounding boxes. Bounding boxes would be nearly perfect for our application if we decided to forbid object rotation. Then we could achieve a perfect fit for rectangular cuboids and a very close fit for spheres and cylinders. And considering our implementation, computing the set of cells that fall within the bounding box and should thus be blocked would be very simple. However, when working with arbitrarily rotated objects, non-rotated bounding boxes actually give a worse fit (block more cells unnecessarily) than bounding spheres. Rotated bounding boxes could provide a perfect fit but the calculations would then be complicated and

the reason for using any bounding volume is to make the calculations simpler.

### 4.1.2 Computing bounding ellipsoids

We support three types of basic objects in our application: a cube, a sphere and a cylinder. By scaling, these objects can be generalized into a rectangular cuboid, an ellipsoid and an elliptical cylinder (a cylinder with elliptical base). The ellipsoid is given by its position and three semi-axes. A semi axis is represented by a direction vector and length.

Assume that the bounding ellipsoid of an object with position  $p$ , scaling vector  $s$  and rotation matrix  $M_R$  is being computed. The bounding sphere radius of the base (unscaled) object is  $r$ . The parameters of the bounding ellipsoid must be computed: semi-axes  $x, y, z$ , lengths of the semi-axes  $a, b, c$  and the position of the ellipsoid  $p_e$ . The calculation is performed by the following algorithm:

1. Position of the ellipsoid is set  $p_e = p$ .
2. Lengths of the ellipsoid semi-axes are set. The scaling vector  $s$  of the object is used. The vector has three components  $s_1, s_2, s_3$ . The ellipsoid parameters are set:  $a = s_1 * r$ ,  $b = s_2 * r$ ,  $c = s_3 * r$ .
3. Semi-axes  $x, y, z$  of the ellipsoid are initialized.  $x = (1, 0, 0)$ ,  $y = (0, 1, 0)$ ,  $z = (0, 0, 1)$ .
4. The semi-axes are now rotated using the object rotation matrix  $M_R$ . The rotation quaternion of the object is also saved within the data of the bounding ellipsoid. It is used in the calculation described in chapter 4.1.4.

If the basic object is a sphere, we do not need to compute the radius of the bounding sphere, we can simply use the scale of the sphere. Otherwise we simply iterate over the vertices of the model mesh and find the vertex furthest from the object centre. The distance between this vertex and the object center gives the bounding sphere radius  $R$ . The same applies to computing the radius  $r$  of the bounding sphere of the unscaled object, the calculation is only performed using un-transformed vertices of the object.

The resulting ellipsoid encapsulates the object and also creates a slight margin around it.

### 4.1.3 Quadrocopter in the environment

Now that we have the objects (obstacles) represented in our environment, we also need to represent the aircraft. The aircraft is an object with certain dimensions, and we want it to avoid the obstacles. Thus, we must ensure no collisions happen. Again, using some kind of a bounding volume can simplify the situation. We could encapsulate the quadrocopter within a bounding sphere, however, we did not take this approach. Essentially, we "flip" the idea.

We can either store information about the aircraft dimensions in the environment representation or we can compute collisions in the path-planning algorithm. We chose the former for multiple reasons. Firstly, this approach is much simpler. We only need to enlarge the bounding ellipsoids of all objects. Secondly, this



approach does not slow our path-planning algorithm down. Collision avoidance during path-planning most certainly would slow the algorithm down. It could potentially have a big impact on the speed of the algorithm.

Computing collisions in our representation would mean computing intersections of spheres and ellipsoids. This can be transformed into computing the distance between the ellipsoid and the sphere center (computing coordinates of the point on the ellipsoid closest to the sphere center). However, the calculation would be quite complicated.

Needless to say, the way we enlarge the ellipsoids does not give the same result. What we would need is the outside offset of the bounding ellipsoid. Let us define the outside offset of an object as a surface on which every point is at a given distance from the object surface, and is outside of the object. An offset can also contain points within the object but these are not interesting for us. However, we only prolong each of the ellipsoid's semi-axes, adding the quadcopter bounding sphere radius to their lengths. This does not produce the same result as the offset of the ellipsoid. The outside offset of an ellipsoid is not an ellipsoid (because the offset of an ellipse is not an ellipse). The semi-axes are of the same length as of the enlarged ellipsoid but the curvature is different. Fortunately, the difference between them is not too big. We also do not expect our aircraft to be able to follow the path with perfect precision so we can afford to have some deviation in obstacle representation. There is enough margin around the objects that no collision should happen.

This way, the path-planning algorithm is simplified, as it searches for paths for a point mass. Point mass is an infinitely small physical object. We then use a path for a point mass to navigate a real object with dimensions. The path does not collide with the enlarged obstacles, and the enlarged ellipsoids provide enough of a margin around the bounding ellipsoids that the aircraft does not collide with the obstacles. We should also point out that the bounding ellipsoids (not enlarged) already create a margin around the real obstacles, only lowering the chance of collision.

#### 4.1.4 From a bounding ellipsoid to blocked cells

An obstacle is represented by its bounding ellipsoid. We then enlarge this ellipsoid to take the quadcopter dimensions into account.

Now we need to transform this representation into a set of blocked vertices (cubes) in our environment representation. We could define a blocked cube this way:

**A cube is blocked when the bounding sphere of this cube is intersecting the bounding ellipsoid of an obstacle.**

This would, however, result in a complicated calculation. We would have to compute the distance between a point and the surface of the ellipsoid.

We use a simpler definition:

**A cube is blocked, when its centre is within the bounding ellipsoid.**

We can compensate for the error (in comparison to the previous definition) by further enlarging the ellipsoid, lengthening its semi-axes by a small constant. However, we do not do this as the ellipsoid is already enlarged enough by the dimensions of the quadcopter. More importantly, this definition then makes the

calculation very simple, as we only use the equation of the ellipsoid to calculate whether the cube centre falls within the ellipsoid or not. We do not have to compute an intersection of a cube (or sphere) and an ellipsoid, which is very complicated. This approach is very efficient.

Assume an ellipsoid with semi-axes  $x, y$ , and  $z$  of lengths  $a, b$  and  $c$ . The cube has a midpoint  $p$  and the quadcopter bounding radius is  $r$ . The calculation for a single cell follows a simple algorithm:

1. Transform  $p$  into the coordinate system given by the ellipsoid semi-axes  $x, y, z$ . Point  $p$  is translated and rotated using the ellipsoid position and orientation parameters and has coordinates  $a_p, b_p, c_p$  in the ellipsoid coordinate system.
2. Follow formula 4.1. If  $f \leq 1.0$ , the cube is blocked. Otherwise the cube is not blocked.

$$f = \left( \frac{a_p}{a+r} \right)^2 + \left( \frac{b_p}{b+r} \right)^2 + \left( \frac{c_p}{c+r} \right)^2 \quad (4.1)$$

## 4.2 A survey of path-finding algorithms

Most modern path-finding algorithms are based on Dijkstra's algorithm with some adjustments. Here we will look at the widely-used A\*, and also some more sophisticated path-finding algorithms, which are based on A\*. The most important thing we had to keep in mind, was that in the end, we will be navigating a quadcopter. These have very high freedom of movement, but can also be quite unstable when frequently or rapidly changing direction. That is why we would like our algorithm to find paths with few direction changes, while keeping path lengths reasonably close to optimal and direction changes not too sharp. That would allow our quadrotor to fly in a smooth and steady manner.

### 4.2.1 Dijkstra algorithm

Dijkstra's algorithm is a well-known path-finding algorithm. It works on graphs with weighted edges that do not contain negative cycles — cycles on which the sum of edge weights is negative. The graph representing our environment does not contain any negative edges, thus it does not contain negative cycles. The algorithm has one starting vertex and a goal vertex. Each vertex has a value, which represents the length of the shortest path from the start vertex to this vertex; and can have three states:

- **unseen** - a vertex the algorithm has not visited yet, its value is infinity (we assume there is no path from *start* vertex to this vertex).
- **open** - a vertex that the algorithm has already visited but its value is not yet final. A better path to this vertex (one with lower value) can still be found.

- **closed** - a vertex that will not be visited again. Its value is final, it can not change and the value is equal to the minimal length of the path from starting vertex to this vertex.

In Dijkstra's algorithm the vertex value is often called *g-value*.

Each vertex has a parent in this algorithm. The parent  $p$  of a vertex  $s$  is the predecessor of  $s$  on the path from *start* vertex to  $s$ . When a path from the *start* vertex to the *goal* vertex is found, we only know its length. That is why vertices must contain information about parents, which can then be used to reconstruct the path.

The open vertices are maintained in a data structure ordered by their value. In every step, the vertex  $s$  with the minimal value is taken from the *open* data structure and is expanded. During expansion, a new value is computed for all neighbours  $n$  of  $s$  which are not blocked nor closed. If such a neighbour is in the *open* data structure, its value is updated. The  $s$  vertex is closed.

As can be seen, time complexity depends on what data structure is used to hold the open vertices and how fast the state of a vertex can be found. The data structure containing open vertices performs three operations: minimum extraction, vertex update and vertex add. The efficiency of these operations has a great impact on the efficiency of the algorithm. For the open data structure a priority queue or a heap is most usually used, and it is best when the vertex state can be determined in constant time.

Algorithm 1 presents the Dijkstra algorithm in pseudocode. The helping functions are presented together in algorithm 2. The algorithms examined in further chapters are very similar.

---

**Algorithm 1** Dijkstra algorithm

---

```
function DIJKSTRA(start, goal)
   $g(start) \leftarrow 0$ 
   $parent(start) \leftarrow start$ 
   $open.Insert(start, Value(start))$ 
   $closed \leftarrow \emptyset$ 
  while  $open \neq \emptyset$  do
     $s \leftarrow open.Pop()$ 
     $[SetNode(s)]$  ▷ only for Lazy Theta*
    if  $s = goal$  then
       $solution \leftarrow ExtractSolution(s)$ 
      return "path found"
    end if
     $closed \leftarrow closed \cup \{s\}$ 
    for all  $n \in neighbours(s)$  do
      if  $n \notin closed$  then
        if  $n \notin open$  then
           $g(n) \leftarrow \infty$ 
           $parent(n) \leftarrow null$ 
        end if
         $UpdateVertex(s, n)$ 
      end if
    end for
  end while
  return "no path found"
end function

function VALUE(s)
  return  $g(s)$ 
end function
```

---

---

**Algorithm 2** Dijkstra help functions

---

```
function EXTRACTSOLUTION( $s$ )
  while  $\text{parent}(s) \neq s$  do
     $\text{solution.Add}(s)$ 
     $s \leftarrow \text{parent}(s)$ 
  end while
   $\text{solution.Add}(s)$ 
  return  $\text{reverse}(\text{solution})$ 
end function

function UPDATEVERTEX( $s, n$ )
   $g_{old} \leftarrow g(n)$ 
   $\text{ComputeCost}(s, n)$ 
  if  $g(n) < g_{old}$  then
    if  $n \in \text{open}$  then
       $\text{open.Remove}(n)$ 
    end if
     $\text{open.Insert}(n, \text{Value}(n))$ 
  end if
end function

function COMPUTECOST( $s, n$ )
  if  $g(s) + \text{cost}(s, n) < g(n)$  then
     $\text{parent}(n) \leftarrow s$ 
     $g(n) \leftarrow g(s) + \text{cost}(s, n)$ 
  end if
end function
```

---

### 4.2.2 A\*

Dijkstra's algorithm explores a lot of vertices which do not lead to the goal. Adding a heuristic function estimating the path cost from the current vertex to the goal vertex, focusing the search towards the goal, can help find the path much sooner.

A\* is such a modification of the Dijkstra algorithm. This changes the order in which the algorithm expands vertices; and thus helps find the shortest path sooner and also expand fewer vertices. Most often the euclidean distance between the vertex and the *goal* vertex is used as the heuristic function. So in the expansion phase, the algorithm prefers vertices situated along the line from the start vertex to the goal vertex before other vertices. This heuristic is admissible because it is underestimating — the actual length of the path from a vertex  $s$  to the goal vertex is at least as much as the heuristic estimate of its length. Using admissible heuristics guarantees that A\* finds the shortest path. The proof is presented in [18] and consists of a few statements:

- A\* must terminate: On finite graphs, this holds true. For A\* not to terminate, it would have to keep adding vertices into *open* infinitely. However, every vertex is added into *open* at most once and at each step one vertex is removed from *open*. Thus, A\* terminates. Note: In our application the graph is always finite.

- A\* terminates in the shortest path: A\* terminates, when the goal vertex is expanded. Since only the vertex with minimal *f-value* in *open* can be expanded and it is the goal vertex ( $f\text{-value}(\text{goal}) = g\text{-value}(\text{goal})$  for admissible heuristics), the found path is the shortest path.
- In every step, there is a vertex in *open*, which is on the shortest path to goal and A\* has found the shortest path to this vertex. Finally, it also satisfies the condition that the heuristic estimate of path length from start to goal through this vertex is lower than the actual length of the shortest path from start to goal. This holds true thanks to the heuristic function being admissible (underestimating).

So the vertices in *open* are now ordered by their  $f\text{-value} = g\text{-value} + h\text{-value}$  instead of only *g-value*, where *h-value* is the heuristic estimate.

The most important property of A\* is, however, that it constrains the path to graph edges. In other words, the parent (predecessor on the path) of a vertex can only be its neighbour because they have to be directly connected with an edge. Unfortunately, this makes it unsuitable for our project, because it introduces many changes in direction of the aircraft. To solve this problem, the found path could be post-smoothed. However, a direct any-angle path-finding approach gives better results according to [15].

### 4.2.3 Non-Admissible heuristics in A\*

This is quite a complicated topic. Using non-admissible heuristics with A\* path-planning can have many effects. Usually a non-admissible heuristic is intended to speed up the search by lowering the number of expanded vertices. A non-admissible heuristic is overestimating — it overestimates the cost of the path from current vertex to *goal* vertex. The effect of such overestimation is that the search is more focused towards the *goal*. This can also help with obstacle avoidance, finding a path around the obstacle faster. Usually, however, if the search needs to backtrack, the effect may be negative, resulting in longer run-times.

The biggest problem of non-admissible heuristics is determining the factor of overestimation. It is very hard to determine by how much the heuristic should overestimate and it is usually performed experimentally and tailored to the specific application.

Using a non-admissible heuristic with A\* means that the algorithm is no longer guaranteed to find the shortest path. It will find some path, it might find it faster than with admissible heuristic, but the path may be longer than the path found with an admissible heuristic.

Non-admissible heuristics can also be used in more advanced algorithms derived from A\*. We, however, do not take advantage of their features. In our algorithm, we implement euclidean distance heuristic, as it is simple and we think using non-admissible heuristics might have undesired effects on our paths, which could potentially make quadrocopter navigation more complicated.

### 4.2.4 Theta\*

Theta\* is an any-angle path-finding algorithm based on A\*. This is achieved by dropping the A\* constraint, that the parent of a vertex can only be its neighbour.

Instead, for an arbitrary vertex  $s$ , its parent can be any vertex which has line of sight on  $s$ . In our representation, two vertices have line of sight when the line connecting the centres of the cubes they represent does not intersect any blocked cubes. Detailed information about Theta\* can be found in [15].

The main advantage of Theta\* over A\* is that it performs any-angle path-planning and thus finds truly shortest paths on cubic grids (our representation). A\* constrains paths to graph edges, thus it cannot always find the truly shortest path.

In the expansion phase, Theta\* considers two paths. Let us assume that the algorithm is expanding vertex  $s$  which has parent  $p$  (which may or may not be a neighbour of  $s$ ). It explores each neighbour  $n$  of  $s$  and performs a line of sight check between  $p$  and  $n$ . Based on the result of the check, it chooses one of the two following paths:

- **direct path from  $p$  to  $n$**  if the check is positive. In this case  $p$  has line of sight on  $n$ , thus a detour through  $s$  is unnecessary and we can take the direct path. Due to triangle inequality this path is shorter than the path through  $s$ . This path allows Theta\* to find any-angle paths and is not considered by A\*.
- **path through  $s$  from  $p$  to  $n$**  if the check is negative. In this case  $p$  does not have line of sight on  $n$ , thus the path through  $s$  must be chosen. We can be certain this path is correct. The vertex  $p$  is the parent of  $s$ , thus it must have line of sight on  $s$ . Otherwise it would not have been chosen as the parent of  $s$  by the algorithm. Vertex  $n$  is a neighbour of  $s$  and is not blocked (the algorithm does not explore blocked vertices), thus  $s$  also has line of sight on  $n$ . So the path is not blocked.

In our 3D environment represented as 26-neighbour grids, performing line of sight checks at the expansion phase is quite expensive. For every vertex inserted into *open*, we could potentially perform the same check several times. In order to speed up our algorithm, we would like to avoid this, and ideally perform a check only once per expanded vertex. Fortunately, Lazy Theta\* makes it possible.

#### 4.2.5 Lazy Theta\*

*Lazy Theta\** is a modification of Theta\*, which postpones line of sight checks until they are absolutely necessary. The algorithm is described closely in [19] and [21]. In Theta\*, the check for a vertex is performed before it is inserted into *open*. Instead of performing the check at this point, *Lazy Theta\** assumes it would be positive and postpones it until the vertex is being extracted from *open*. When  $s$  is extracted from *open*, it has a parent  $p$ , and a line of sight check between  $p$  and  $s$  is performed. When the check is positive, the assumption was correct. However, if it is negative, a new parent for  $s$  has to be found. The algorithm now iterates over all neighbours  $n$  of  $s$ , which are closed. Then the one with shortest path length from the *start* vertex to itself (minimal *g-value*) is chosen as the new parent of  $s$ .

The algorithm does not add much complexity over standard Theta\* and it significantly lowers the number of line of sight checks. Thanks to this it can run noticeably faster. We determined it to be the best algorithm to navigate

a quadcopter in 3D space using our representation. It implements any-angle path planning and can run reasonably fast on 3D spaces, so it can find relatively smooth paths our quadcopter should be able to follow.

As *Lazy Theta\** is the algorithm we implement, we present the pseudocode in algorithm 3. Only the parts different from the Dijkstra algorithm presented in algorithms 1 and 2 are shown.

---

**Algorithm 3** Lazy Theta\* algorithm

---

```

function VALUE( $s$ )
  return  $g(s) + h(s)$ 
end function

function SETNODE( $s$ )
  if  $\neg \text{LineOfSight}(\text{parent}(s), s)$  then
     $\text{parent}(s) \leftarrow \arg \min_{n \in \text{neighbours}(s) \cap \text{closed}} (g(n) + \text{cost}(s, n))$ 
     $g(s) \leftarrow \min_{n \in \text{neighbours}(s) \cap \text{closed}} (g(n) + \text{cost}(s, n))$ 
  end if
end function

function COMPUTECOST( $s, n$ )
  if  $g(\text{parent}(s)) + \text{cost}(\text{parent}(s), n) < g(n)$  then
     $\text{parent}(n) \leftarrow \text{parent}(s)$ 
     $g(n) \leftarrow g(\text{parent}(s)) + \text{cost}(\text{parent}(s), n)$ 
  end if
end function

```

---

## 4.3 Implementation

We have already discussed the theoretical aspect of path planning and chosen the path-planning algorithm. In this section we discuss the implementation. Figure 4.4 shows the class diagram of this component.

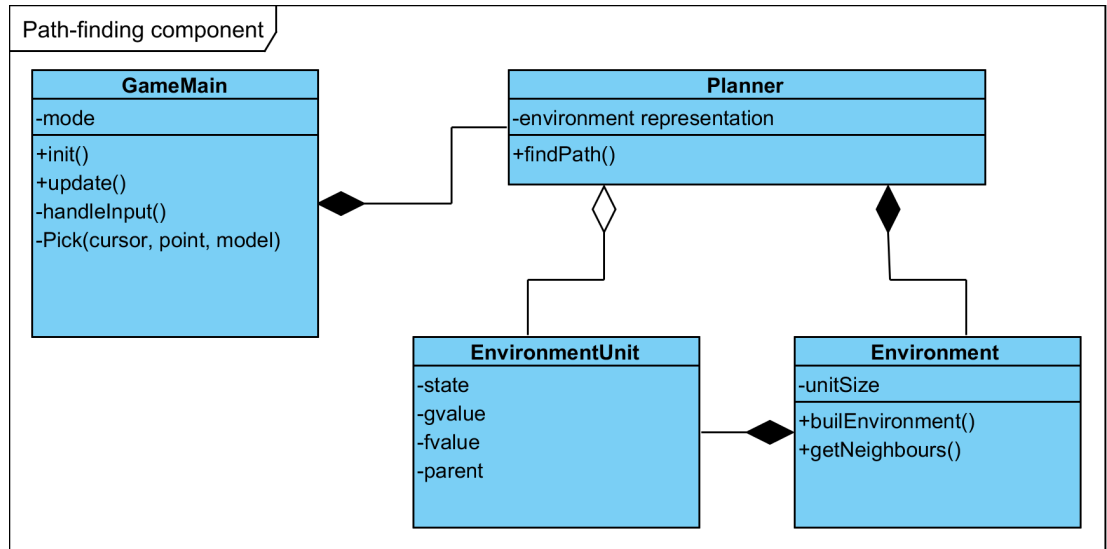


Figure 4.4: Path-finding component class diagram



### 4.3.1 Environment representation

Our chosen representation of regular 26-neighbour grids proved to be very easy to implement. At first we were experimenting with a memory-saving approach, only providing a method that computes all the neighbours of a vertex. This was a step in the wrong direction, because it made our algorithm very slow. When vertex  $s$  was being expanded, for every neighbour of  $s$ , the algorithm had to search through the *open* and *closed* data structures to determine its state. The information was only accessible that way. That was, of course, very inefficient. It is a much better approach to actually store information about vertices in memory. Then we can determine the state and values of the vertex in constant time, unlike before, when we had to search through all closed and open nodes.

We use the **EnvironmentUnit** class to represent vertices. Multiple properties are stored in the class: the *position* of the vertex, the *state*, path-planning algorithm values (*g-value* and *f-value*) and finally a link to the *parent* of this vertex. The class takes up only 36 bytes of space, so we can represent relatively large environments. Even with 1 million units, the memory consumption is quite negligible on computers of today, equipped with gigabytes of memory.

The environment is represented as a three-dimensional **STL vector space** — **STL vector** containing **STL vectors** which contain **STL vectors** filled with environment units. The blocked units are represented as an **STL vector** containing identifiers of units. An identifier of a unit is a set of three indices into the *space* data structure, one index per dimension. We implement identifiers as a class **UnitIndex**.

### 4.3.2 Blocking cells

First, we enlarge the bounding ellipsoid to take the quadcopter dimensions into account. Then we select the set of candidates for blocking. We transform the ellipsoid position into an index to the environment representation *space* structure. We take the length of the greatest semi-axis of the ellipsoid and determine the number of candidates in each direction. Then the candidates for blocking are arranged in a cube. Finally, the calculation follows according to the definition presented in 4.1.4.

In figures 4.5 and 4.6 we present an example of blocking cells. We use a sphere with radius 10 cm and a cube 10 cm in size and scale parameter of 10 cm. The margin around the objects already includes quadcopter dimensions.

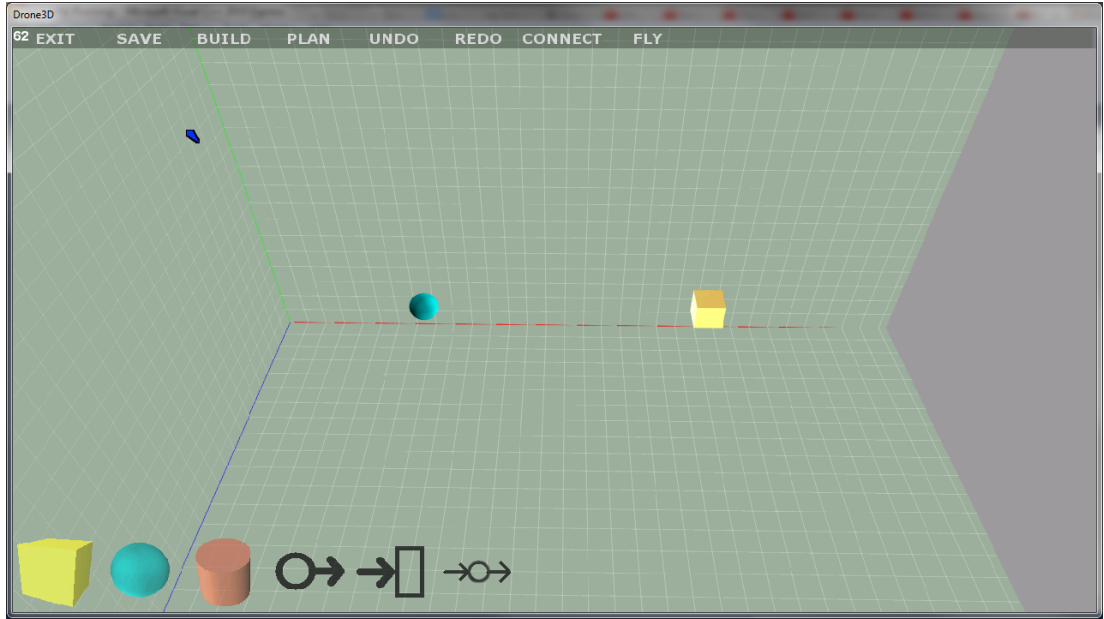


Figure 4.5: A 10 cm sphere and a 10 cm cube in the environment

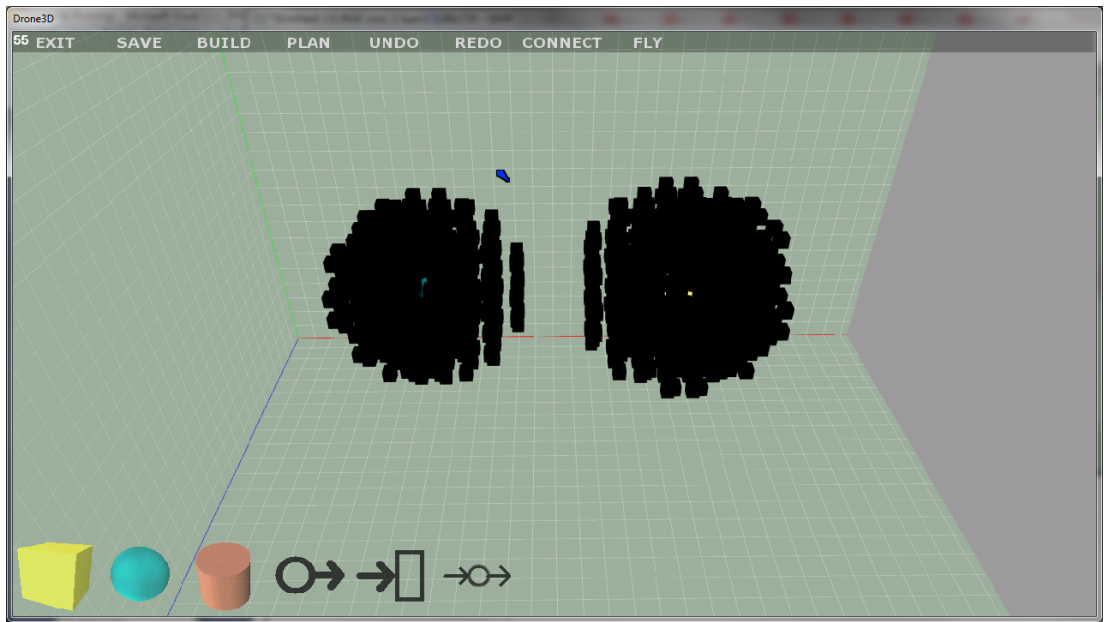


Figure 4.6: Vertices blocked by the sphere and the cube (shown in black)

### 4.3.3 Lazy Theta\* implementation

Like the rest of our application, our algorithm is implemented in C++ utilizing the C++ Standard Library. We have the entire environment stored in memory, and thanks to that we can determine the state and values of any vertex in constant time. Our algorithm also needs to maintain the *open* data structure. We do not need a data structure for closed vertices because we already store this information in the vertex itself. However, we keep it in the implementation as the size of the *closed* "list" lets us monitor algorithm progress. The *open* set is an **STL vector** of unit identifiers paired with their *f-value*. The elements of *open* are instances of **EnvUnitID** class.

**EnvUnitID** class holds an environment unit *identifier* (class **UnitIndex**) and *f-value*. These are all the data necessary for the *open* structure to perform its role. The *identifier* provides a link to the environment representation so that we can determine the state or neighbours of this vertex. The *f-value* is used to order the vertices within the *open* **STL vector**.

However, using only the **STL vector** would result in slow performance of *open* data structure operations. Thus the **STL vector** only serves as an underlying container for C++ **Standard Library heap algorithms**. The library implements four algorithms encapsulating heap operations: *push\_heap*, *pop\_heap*, *make\_heap* and *sort\_heap*. They can be used on multiple kinds of underlying containers, however, we use a simple **STL vector**.

We do not use *sort\_heap*, as it is a sorting algorithm which takes advantage of the heap structure to put the elements in the container into ascending order quickly. We do not require this functionality. The *make\_heap* algorithm rearranges the elements within the underlying container, so that they form a heap. The *push\_heap* algorithm adds one element into an existing correctly formed heap. The *pop\_heap* algorithm extracts the top (**maximum**) element from an existing heap. Since we require the minimum element, this presents a problem, though it is very easy to solve (see the following paragraph). The C++ Standard Library heap algorithms implement exactly the operations required by our algorithm. Using these algorithms to implement path-planning algorithms is also suggested in [20].

The C++ Standard Library heap algorithms by default construct a heap with the maximum element on the top. However, we need the minimum element there. Fortunately, they take an inequality comparison function as a parameter, so we decided to take advantage of that. We implemented a comparison function which returns the opposite of what the inequality operator on **EnvUnitID** would. So if we compare **EnvUnitID** *a* and *b* using our function and comparison by inequality operator  $a < b$  holds true, our comparison function returns false and vice versa. This way we obtain a heap with the minimum element on top, thus we can extract it quickly using the *pop\_heap* algorithm.

#### 4.3.4 Adapting the algorithm to quadcopter navigation

This adaptation is very simple and does not influence the algorithm directly. We only normalize the start and goal locations by adjusting their height to 1 m above the start and goal locations set in the program to account for the aircraft takeoff. The aircraft is programmed by the manufacturer to automatically hover around 1 m above the surface after takeoff.

## 5. Plan execution

In this chapter we will introduce our aircraft, the Parrot AR.Drone, in detail and examine its flight characteristics. Then we will describe the network interface and how our program communicates with the drone. Finally, we will present the navigation routine and the experiments we performed and their results.

### 5.1 Parrot AR.Drone

The AR.Drone is a quadcopter aircraft — essentially a helicopter with four propellers. The propellers are all aligned in one plane and their orientation is fixed. The aircraft can move in all directions. To describe the movements, we can split the four rotors into six pairs: *front*, *rear*, *left*, *right* and diagonal *clockwise* and *counter-clockwise* pairs.

Movement forward and backwards is performed by changing the pitch angle of the aircraft, which is done by adjusting the speed of the *front* and *rear* rotor pairs. Left and right movement is performed by changing the roll angle, which corresponds to adjusting the speed of *left* and *right* pairs of rotors. Movement up and down corresponds to adjusting the speed of all four rotors equally. Finally, turning left and right is performed by changing the yaw angle, which corresponds to adjusting the speed of *clockwise* and *counter-clockwise* pairs of rotors. These rotors are paired-up diagonally, which results in the quadcopter turning in place (the centre of rotation is in the middle of the aircraft).

The Parrot AR.Drone currently comes in two versions. The second version is more advanced, features a wider array of sensors and overall has better flight characteristics. For our project we were using the AR.Drone version 1.

#### 5.1.1 Sensors on the Parrot AR.Drone

The first version of the AR.Drone is equipped with a 3-axis accelerometer, 2-axis gyroscope and 1-axis yaw gyroscope. These sensors allow the drone to fly in a relatively stable way and measure its orientation. Furthermore, it is equipped with an ultrasound altimeter with range up to 6 meters which helps the drone maintain altitude. It also features two cameras. One facing forward and one facing down. The camera facing down is used to aid in stabilization. Image from both of the cameras can be streamed onto the connected device.

The second version of the drone features a greatly improved frontal camera (with 1280x720 resolution). It also adds more sensors — a pressure sensor for more accurate altitude measurements and a magnetometer compass which allows the use of a special control mode with some mobile devices. The second version is an overall improvement over the first one.

Finally, both versions of the drone feature a Wi-Fi hotspot. To control the aircraft, a computer or a mobile device must be connected to this hotspot. There are three communication channels. On the first one, the drone receives control commands. The other two channels are used to send data from the drone to the controlling device; one channel for navigation data and one for video.

### 5.1.2 Flying with the Parrot AR.Drone

Flying with the drone is quite easy and even children can manage it. However, precise control is very hard to achieve. Control with a smartphone or a tablet is usually quite imprecise and is mostly intended as a toy. More precision can be achieved by using a computer joystick or a keyboard to control the aircraft. Nevertheless, precise control is still problematic. A big part of this imprecision stems from the characteristics of the aircraft itself.

We found out that the aircraft has a tendency to drift after taking off. Our unit usually moves a couple of metres backwards. We found out from our colleagues that they have similar problems, only their drone drifts to the right. This makes navigation by a program even more complicated.

When moving, the aircraft has a certain momentum. For a human, this is quite easy to compensate for. However, it is much harder to simulate in a program. The AR.Drone is also very light, making it susceptible to wind and even the turbulence it itself creates. It is capable of flying outdoors but even a slight breeze can influence the trajectory. A stronger wind usually causes the quadcopter to lose control and/or flip over. Indoors, the aircraft creates a lot of turbulence which can "upset" it a lot, especially when close to objects or walls. This turbulence can completely change the heading of the aircraft and it makes flying in cluttered and confined areas nearly impossible.

## 5.2 Drone communication

The following diagram shows the structure of the drone control component:

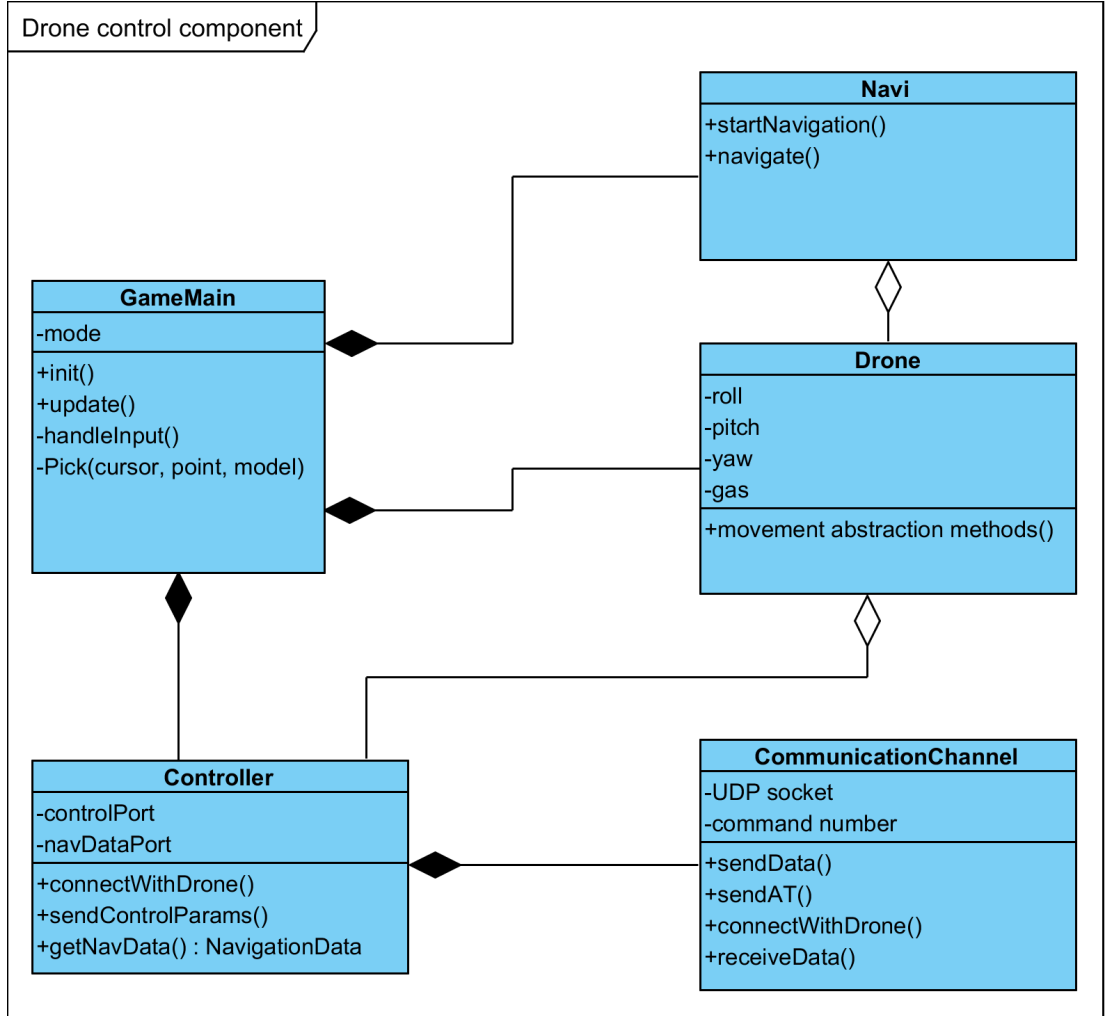


Figure 5.1: Drone control component class diagram

We implement all the classes ourselves, however, the **CommunicationChannel** and **Controller** classes are inspired by a library created by *Pongsak Suwanpong*, see [23].

The **CommunicationChannel** class implements network communication: creating the channel and sending and receiving data on the channel. It also implements a specific method of sending the control commands to the drone,. The method which obtains a command in a specific format and must encode it into a stream of bytes to send it as a UDP (User Datagram Protocol) packet.

The **Controller** class then has two channels — a control channel and a data channel. The control channel is used to send commands to the aircraft and the data channel is used to receive the navigation data from the drone. We do not use the cameras, thus a video data channel is not implemented. The **Controller** also implements commands which encapsulate the raw commands processed by the **CommunicationChannel** class.

The **Drone** class creates yet another abstraction layer over the commands,

encapsulating the **Controller** commands into more natural commands like *move-Forward*.

The **Navi** class implements the navigation routine. When navigation is started, it receives the desired path and then proceeds to send instructions to the aircraft in order to guide it along the path.

### 5.2.1 Structure of commands

The Parrot AR.Drone is controlled by **AT commands**, which are essentially text strings. These commands are sent within UDP packets [24]. According to the AR.Drone Developer Guide [25], the commands should be sent at least every 30 ms for smooth drone movement. Our application sends commands even more often than that. An important feature to keep in mind is that the drone will consider the connection to be lost if a delay between commands is more than 2 seconds. If this happens, the drone will set its "watch dog" and stop sending data to the controller.

The command syntax is described in sufficient detail in the AR.Drone Developer Guide [25]. In simple terms, a command consists of a header and arguments. The header signifies the type of the command (like *set orientation*) and the parameters provide the values to be set. The header always starts with the **AT\*** character sequence.

These are the most important commands:

- **AT\*REF**: These are the takeoff, landing and emergency stop commands, depending on the parameter.
- **AT\*PCMD**: This command is used to set the drone orientation angles (roll, pitch and yaw) and gaz (motor speed).
- **AT\*FTRIM**: After receiving this command, the drone will reset its gyroscopes and other sensors, and set current orientation as default. Thus, it should always be placed on a flat surface before this command is sent.
- **AT\*COMWDG**: This command instructs the drone to reset its communication "watch dog". This command is helpful to counter the 2 second delay described above. It can be used to keep the connection alive as well as re-initiate the data streams with the help of other commands.

### 5.2.2 Incoming data streams

We only implement and use the navigation data stream. The video stream is unnecessary for our application, as we do not use the cameras.

Both data streams are described in the AR.Drone developer guide [25]. Unfortunately, the navigation data stream is not described in enough detail, rather refers to the Parrot SDK source code.

The navigation data stream packet consists of a header part and option blocks. The header part consists of the header itself, followed by drone state, sequence number and flags. Then the *options* follow, and each *option* consists of fixed-length ID and block length values (always 16-bit unsigned integers), followed by the data.

Table 5.1: Structure of AR.Drone navigation data

item	type	unit
ID	16-bit unsigned integer	
length	16-bit unsigned integer	
control state	unsigned integer	
battery level	unsigned integer	
pitch angle	float	millidegrees
roll angle	float	millidegrees
yaw angle	float	millidegrees
altitude	integer	centimetres
forward speed	float	mm/s
left/right speed	float	mm/s
vertical speed	float	mm/s

It is possible to receive the data in two modes: demo (minimal data) and debug. We use the demo mode and we are only interested in the *option* names *navdata\_demo\_t*, which contains data about drone orientation and speed.

All the values of *navdata\_demo\_t* are 32-bit floating point numbers and integers (except of the option ID and length) and the structure is as follows:

### 5.2.3 Network interface

Since we have decided to use DirectX for 3D rendering, we knew our application would be designed for the Windows operating system. Thus, we could use WIN32 API to implement the network communication. Network programming requires sockets, which are implemented in Winsock.

We also had other options to implement the network communication. We could have used the official API by Parrot, however we were not satisfied with it. It is quite bloated and cumbersome. It has many features completely unnecessary for our application. Furthermore, the documentation is sometimes missing crucial information and does not go into enough detail.

Another option is to use a library or a program created by someone else. We found a nice library implementing AR.Drone communication channels [23] in C++, however, it was using the `commonc++` library to implement socket communication. The `commonc++` library is very big and would not bring us any benefits. Thus, we decided to implement our own solution. We took many ideas from [23], especially since some important parts are not described in the Parrot technical documentation in enough detail (e.g. the exact structure of navigation data sent by the drone to the controller).

The main advantage of implementing a custom solution is that it is very lightweight and thanks to using Winsock it does not require additional libraries (Winsock is a part of WIN32 API).



## 5.3 Navigation routine

The navigation routine is implemented within the class **Navi**. In this section, we will explain how the navigation works.

To make navigation easier, we limit the maximum speed and maximum altitude of the aircraft. It is possible to send commands setting maximum altitude and maximum tilt angles of the drone. Maximum altitude is set at 2m and the tilt angle limits are 20 degrees. This way we limit the speed of the drone, making both navigation and user control easier.

First, the navigation has to be started. The **Navi** class receives the *path* the drone should follow and its *starting direction* vector. The path is a list of points, in which two successive points form a line segment (a part of the path). Each point (except of the first and last) is then a location where the drone will turn. This way we divide the path into straight **segments** and **turning locations**. The starting direction vector is important for initial heading adjustment.

From the path data, the heading of each segment is computed. Then the initial heading correction angle is computed as the difference between the heading of the first segment and the initial direction heading.

After the initialization is finished, navigation can start. We divide the process into several phases:

**Ready:** This is the initial phase of the navigation, into which it is put after it is started. The *"watch dog" reset* command is sent to the drone to make sure navigation data are being received. Then, the *takeoff* command is sent, and we move on to the second phase.

**Hover:** The second phase lasts for five seconds and gives the aircraft time to stabilize after takeoff. The navigation routine tries to compensate for the drone movement, as it has a tendency to drift after taking off, see 5.1.2.

**Setup:** During this phase, the drone adjusts its heading to match the heading of the first path segment. Then, navigation can proceed to the next phase.

**Flight:** In this phase, the navigation actually happens. We distinguish the last segment from the other segments and the navigation is different in the two cases. At all times, the navigation routine is checking if the drone heading is correct and if the drone has not drifted too far to the left or right of the path, and performs adjustments accordingly. We examine this phase in detail below, as it is the bulk of the navigation routine.

**Done:** This phase is the second part of the last segment navigation. It is entered when the navigation routine determines the drone has enough momentum to drift to the *goal* and the *hover* command is sent repeatedly. The phase exits when the forward speed of the drone drops below 100 mm/s.

**Landing:** This phase follows after the drone speed drops to negligible values during the **Done** phase. The *land* command is repeatedly sent to the drone until altitude drops below 10 cm. Then we can be sure the drone is actually landing. We implemented this because we encountered problems with landing. Sometimes the drone would refuse to land if only one command was sent.

**Landed:** This is the last phase and marks the end of navigation. Commands are no longer sent to the drone.

During all phases of navigation, a logging system is active and writes information about navigation into a log file. The phase and performed adjustments are logged, along with navigation data. Each piece of information written into the log has a time stamp.

### 5.3.1 Hover phase

During this phase, we implement a "controlled takeoff". The navigation routine sends commands to the aircraft to attempt to counter the drift of the aircraft. The goal is to perform a takeoff as close to a perfect vertical takeoff (without horizontal movement) as possible.

Navigation data from the drone are obtained and the horizontal speed values from the data are used to compute the orientation of the drone to counter the speed. The goal is to keep speed at minimal values. The calculation follows these rules:

- drone *pitch* angle scales linearly with the value of the drone forward speed, up to 1000 mm/s. Speeds over this value result in *pitch* angle being set to the maximum or minimum, depending on direction.
- drone roll angle scales linearly with negated value of the drone left/right speed, following the same rules as the *pitch* angle. Negating the value is necessary because negative left/right speed values represent movement to the left and negative *roll* angle moves the aircraft to the left.

Introducing "controlled takeoff" helped improve our takeoffs, however, it did not completely solve the problem. Sometimes the aircraft moves backwards a lot (over 1 m). We believe this may be caused by the fact that our application does not receive new navigation data as often as we expected. The AR.Drone Developer guide states, that the drone should send data with a time interval of less than 5 ms. Examining our flight logs, however, we found that our program does not receive navigation data in such a short interval, rather around every 50 ms. Unfortunately, we were unable to solve this issue.

While flying the aircraft ourselves, we also noticed that it seems to ignore commands for a short period after the takeoff command is sent. This makes it very hard to compensate for the unstable takeoff.

### 5.3.2 Flight phase

The flight phase is when the drone is actually moving. We use the obtained navigation data from the drone to compute position and heading changes, from which we determine what commands we should send to the drone. This approach mimics a PID controller (Proportional-Integral-Derivative controller) but is not a true PID controller. It only considers the proportional part, so it is only a P (proportional) controller. The integral and derivative parts were not implemented.

### Last segment

Let us examine the special case first—navigation in the last segment. During this segment, we compute the remaining distance the drone has to fly in order to reach the goal. We also maintain the correct heading (in the same way as in **Setup** phase) and short perpendicular (left/right) distance from the path.

We use the drone forward velocity together with the remaining distance to calculate, whether we should switch to the **Done** phase or not. We do this using basic physics formulae for uniformly decelerated motion. Since the drone is moving at low speeds, we can consider the motion to be uniformly decelerated. Thus, the deceleration parameter is constant and it is unnecessary to dynamically change this parameter.

$$v = v_0 - at; \quad s = v_0 t - \frac{at^2}{2}$$

where

- $t$  : time interval
- $v$  : final velocity
- $v_0$  : original velocity
- $a$  : deceleration
- $s$  : distance

In this case, the final velocity  $v$  is 0 m/s, because the aircraft is supposed to land. We use the Windows high resolution timer (method *QueryPerformanceCounter*, see [26]) to compute the time interval  $t$  between the last drone command and current command. We compute the distance  $s$  as the distance between the current drone position and the goal. The current velocity  $v_0$  is obtained from the drone navigation data. The forward velocity is used.

The deceleration parameter  $a$  is unknown and to determine it exactly would be too complicated. Either we would have to take measurements in a wind tunnel, which was not available to us; or derive the parameter from the aircraft characteristics, which would be extremely complicated. Thus, we only estimated this parameter by running test with the aircraft. Initially, we estimated the deceleration value at  $0.5 \text{ m/s}^2$ . After examining the log files of our further test flights we concluded that our estimate was very close to the real value. However, during the first measured test flights, we noticed the aircraft "overshoots" the goal most of the time. By lowering the parameter to  $0.45 \text{ m/s}^2$ , we were able to obtain much better results.

Using the current velocity  $v_0$  and deceleration  $a$ , we compute the expected time  $t_e$  it would take the quadcopter to stop (forward velocity drops to 0 m/s). We then use the second equation to compute the expected distance  $d$ , which is the distance the quadcopter would travel if we switched to **Done** phase at this moment. If expected distance  $d \geq s$  (remaining distance), we know the aircraft has enough momentum and navigation is switched to the **Done** phase. Otherwise the navigation remains in the **Flight** phase.

## Standard segment

In case of a standard segment, we also maintain the correct heading and short perpendicular distance from the path. This is necessary for our navigation routine to recognize the turning locations. A turning location is located at each path point.

We measure time the same way as in case of the last segment, and we also obtain current velocity data from the drone. We use these to update the current position of the drone, which in turn allows us to compute the distance from the drone to the next turning location.

If the drone is close to the next turning location, the navigation switches to the next segment. Whether the drone is close is determined by computing the distance between the drone position and turning location in the program. If the distance is lower than 20 cm, the drone is considered close to the turning location. Thus, the navigation routine will start adjusting the drone heading to match the next segment heading, and the drone will turn.

### 5.3.3 Computing the change of drone parameters

#### Position change

The position change of the drone is computed using the speed values obtained from the received navigation data and the time interval between the last and current command.

Speeds in all directions are considered. The drone position is maintained as a point in the 3D environment. Position change is a vector which is rotated around the vertical axis by the drone *yaw* angle and is then added to the drone position to compute the new position of the drone.

#### Heading change

Drone heading calculation is performed by comparing the yaw angle of the drone from received navigation data with the heading of the segment and turning the drone accordingly. At the beginning of navigation (at the start of **Hover** phase), the headings of all segments are "normalized". The *yaw* angle of the drone is added to all the segment headings because the initial *yaw* angle of the drone may not be 0 degrees.

#### Maintaining short distance from the path

The horizontal distance between the current drone position and the segment line is computed. This calculation also determines on which side of the segment line the drone is located (left or right). If this distance exceeds 10 cm, the navigation routine tries to compensate and bring the drone closer to the segment line by sending appropriate commands.

## 6. Experiments

We performed several experiments with the aircraft, testing the navigation routine and path-planning algorithm in various scenarios.

The test were performed in a closed room approximately 17 m by 8 m large and 3 m tall. The room was nearly empty, except of a few small objects and two big columns (95 cm x 125 cm) on one side. A ventilation system was running in the room, and it was not possible to switch it off completely. Thus, near the vents of the system, airflow was relatively significant. However, there was very little airflow in the room overall. The size and low airflow made the room a suitable location for our experiments. Needless to say, in some places, the airflow from the vents impacted the aircraft significantly. In figure 6.1 a photo of the room is shown.



Figure 6.1: Photo of the room where we performed the experiments

Our aircraft was not in perfect condition. It has been used in some experiments before, where it took minor damage. Some of the components were also quite worn, like the propellers. However, the aircraft was still capable of flight and easy to control for a human. While creating the navigation routine, we had a few crashes. Despite our best efforts, sometimes we were unable to prevent them. One of the crash has unfortunately caused minor damage to the rear left motor. After this incident, the motor would sometimes not spin up during takeoff, upon which we have to reset the drone.

Unfortunately, we were not able to track the aircraft in flight because such equipment was not available to us. We were only able to measure the distance

between the goal and the point where the aircraft actually landed. This can give us an idea about the overall accumulated error during the entire flight. However, using these measurements, we cannot identify problematic aspects of the navigation routine easily. Neither can we objectively measure how closely the aircraft has followed the path. We can only make observations and gather data from the flight log.

## 6.1 Straight flight 3 metres

In this test, we set the path for the drone to be only a straight line 3 m long at the same altitude. The drone should take off, fly 3 metres and land.

During the test, the aircraft was able to land relatively close to the goal most of the time. In some cases, the flight was very unsuccessful due to outside factors and we do not include these in the measurements.

We organized the measurements into a convenient table:

Table 6.1: Straight flight

flight	difference in direction [cm]	difference perpendicular to direction [cm]
1	65 B	30 R
2	30 F	30 R
3	250 F	170 R
4	0	75 R
5	30 F	10 L
6	140 F	130 R
7	75 B	30 L
8	45 F	75 L
9	20 F	60 R
10	60 F	35 L

F, B, R and L stand for *forward*, *backwards*, *right* and *left*, respectively.

The table shows that the results are relatively inconsistent. While most of the time the aircraft does indeed land close to the goal, very bad results are not especially rare. The average of the values is quite good, however, it is not a valuable piece of information in this case, due to the inconsistency of results.

During the test, we observed that it is very dependant on the quality of the takeoff. Even after introducing "controlled takeoff" in the **Hover** phase of navigation, sometimes the aircraft moves more than 1 m backwards and to the left, when taking off. Part of this problem may stem from the airflow within the room as we described earlier. Also the problem with receiving navigation data certainly plays a role. Moreover, when landing, the aircraft often drifts in some direction due to outside influence. Then it lands in a different location to that it was in when it started landing. During landing, we do not control the aircraft. Unfortunately, with out tools it is not possible to measure where the aircraft was in the air when it entered **Landing** mode. We also experienced issues with landing, when the aircraft would refuse to shut down its engines and bounce

off the ground several times before doing so. We cannot explain this behaviour. Fortunately, it is very rare.

The results are close to what we expected. While we were hoping for more consistent results, we are glad that we achieved results as good as this. However, this is the easiest test and more demanding tests are ahead of us.

## 6.2 30 degree turn

In this test, we prepared a simple turn. The aircraft would fly straight 1 m and then turn 30 degrees to the right and fly straight for 115 cm before landing.

At first, we were performing this test with the left/right drift compensation turned on. So our navigation routine tried to make sure the aircraft stays close to the set path. However, the results we obtained were unsatisfactory. Due to the problematic takeoff, the aircraft would move to the left of the path, thus detect that it needs to compensate for this drift. Unfortunately, the method was not working very well and it caused our aircraft to fly the the right side excessively.

We concluded that this part of the navigation routine should be improved. To still be able to perform the test, we decided to turn this feature off. We obtained surprisingly good results, which we show in the following table.

Table 6.2: Flight with 30 degree turn

<b>flight</b>	<b>difference in direction [cm]</b>	<b>difference perpendicular to direction [cm]</b>
1	5 F	20 R
2	20 F	10 L
3	40 F	60 R
4	10 B	50 R
5	30 F	10 L
6	50 F	60 R
7	0	15 L
8	15 F	10 R
9	50 F	45 L
10	30 B	0

F, B, R and L stand for *forward*, *backwards*, *right* and *left*, respectively.

This is a better result than we obtained when flying straight. This is quite surprising and perhaps it is only a coincidence.

We observed that the aircraft performed the turn very close the the location it should and it turned nearly precisely 30 degrees every time.

## 6.3 90 degree turn

We performed this test in the same conditions as the tests before. The drone would fly 1 m straight, then turn 90 degrees to the right, fly 2 m straight and land.

When we began this test, the results looked very promising. We expected the results of this test to be worse than in the case of the **30 degree turn** test, which proved to be the case after a few flights. During the testing, we encountered some issues. Sometimes our aircraft behaved in an inexplicable way, performing actions completely different from the commands we were sending. The aircraft rolled to the right when no rolling instructions were sent. This may have been caused by a problem in communication between the computer and the drone. Otherwise we cannot explain such behaviour.

In one instance, the rear left engine of the drone did not start. So we restarted the drone by pulling out the battery and plugging it back in. When we plugged it in, a tiny explosion resounded and one of the engine boards started smoking. Nothing else happened and after plugging the battery in again the drone was flying normally. We were lucky that the drone was not severely damaged and we could continue our experiments.

Again, we put the results in a table:

Table 6.3: Flight with 90 degree turn

<b>flight</b>	<b>difference in direction [cm]</b>	<b>difference perpendicular to direction [cm]</b>
1	15 B	30 L
2	45 F	60 R
3	0	110 L
4	15 B	30 L
5	20 F	40 R
6	55 F	20 L
7	150 F	50 L
8	80 F	35 R
9	20 F	20 R
10	55 B	20 L

F, B, R and L stand for *forward*, *backwards*, *right* and *left*, respectively.

While the values in the table do not look bad, we had many unsuccessful flights, partly due to the technical issues we experienced. The shortcomings of the navigation routine are also partly responsible for unsuccessful flights. Moreover, we were unable to explain certain errors.

We observed that the aircraft started turning slightly late. Then it got to the left side of the path and landed on the left side of the target, which can also be seen in the results in the table. This is caused by imperfections in our program.

On multiple occasions, the aircraft also turned too much and did not correct its heading later like during other flights. We examined the flight logs of our navigation routine and found out that the routine had detected this. It also kept sending the right instructions to the AR.Drone to correct its heading. Despite the fact that we could not find any error in the navigation or the flight logs, the aircraft simply did not turn. The only way we could explain this is that a problem in communication occurred. However, that should not have been the case as the aircraft did respond to landing instructions later. We tested the aircraft



by controlling it manually and it had no problems turning to either side. Thus, we are unable to explain this behaviour.

## 6.4 Zig-zag

In this experiment, we defined a more complicated path. The aircraft would fly 1 m forward; turn 30 degrees to the right, fly 115 cm straight; turn 60 degrees to the left and fly 2 m straight before finally turning 30 degrees to the right and flying another 115 cm and landing.

Unfortunately, we did not manage to obtain any data from this test. The aircraft simply was not capable of following the path all the way. The best result we obtained was completing the second turn (60 degrees to the left). While the heading of the aircraft was correct, it was not in the correct position and missed the last turning location.

We knew the navigation routine would have big problems with this test, however, we did not expect results this bad.

## 6.5 Path from algorithm

In the tests before, we did not test the ability of our navigation routine to change the altitude of the aircraft. Thus, we decided to set up this test to also include altitude changes.

We modelled a scenario in which the quadcopter had to fly 3 m forward. In the middle between the start and goal locations we placed an obstacle 65 cm tall, 40 cm long and 70 cm wide. We modelled this in the editor and also added "fake" obstacles to both sides of the real obstacle to force the path-planning algorithm to find a path which leads over the obstacle. With only the real obstacle modelled, the algorithm found a path around the obstacle which was not what we wanted to test. The model is shown in figure 6.2. The figure also shows the found path. We did not model the entire room where we performed the experiments, only a part of it relevant to the experiment.

The aircraft flew over the obstacle without crashing every time. However, it usually got thrown off course by its own turbulence hitting the obstacle. This is due to the characteristics of the aircraft and is very hard to counter.

We could not objectively measure the altitude changes and observing them was difficult as they were minimal.

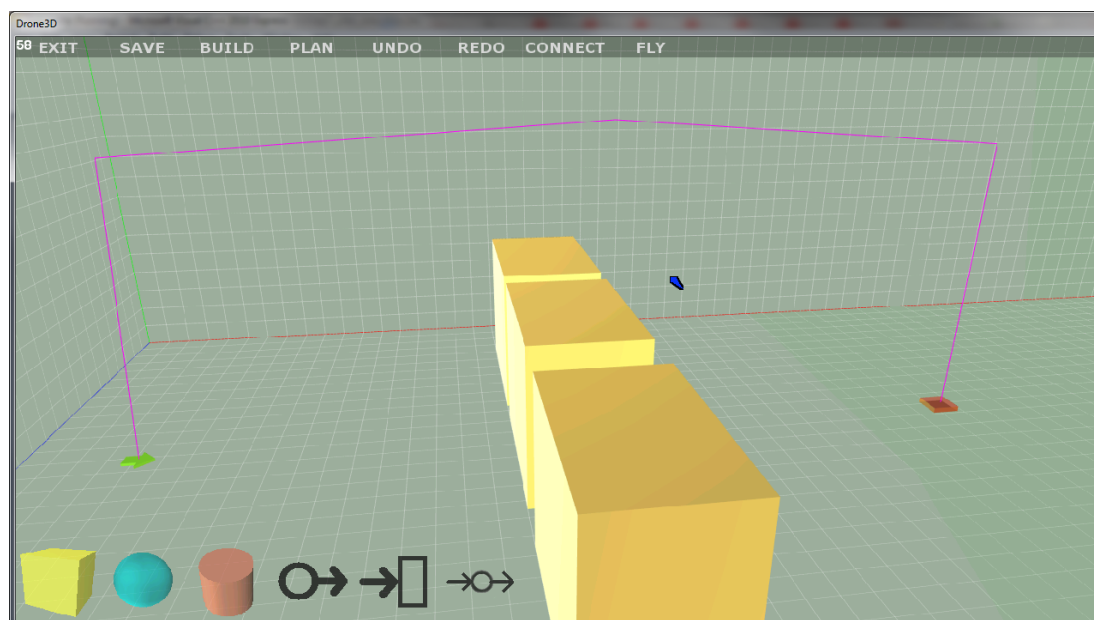


Figure 6.2: Model of the experiment in the editor

# Conclusion

At the beginning, we set out clear goals that we wanted to accomplish. Now is the time to evaluate our work—whether or not we accomplished our goals.

We had three main goals:

- Implementing a 3D environment editor
- Implementing a path-planning system
- Extending the editor to be able to navigate a quadrocopter using a path from the path-planning system

We succeeded at implementing the 3D environment editor. The editor has a wide array of functions and allows for creation of environments with simple objects. Simple objects can be placed into the environment directly, and more complex objects can also be modelled by grouping multiple simple objects together.

The editor implements advanced but necessary features like undo and redo. It also implements very easy to use functions to manipulate the objects in the environment. Thanks to this, modelling a room with objects is simple and fast.

Implementing the editor was also our first venture into the world of DirectX and 3D graphics. While we certainly did not take full advantage of the power of DirectX due to implementing only simple visuals, we obtained experience and learned about the principles of DirectX and game programming.

Using DirectInput and DirectX to implement the user interface of our application was perhaps not the best choice. All the components work very well, however, it took a lot of effort and development time. Using a higher level library might have saved us time that we could have then devoted to other parts of the project.

We have also successfully implemented a path-planning system. We examined multiple options of representing the modelled environment and an array of path-planning algorithms. We chose a simple yet efficient way of representing the environment—a grid of cubes. For the path-planning algorithm, we chose the modern and advanced *Lazy Theta\** algorithm.

The algorithm does indeed find paths suitable for a quadrocopter. Unfortunately, we were unable to completely verify this due to the shortcomings of the quadrocopter navigation. However, examining the paths found by the algorithm, we notice that they usually do not contain very sharp turns. Thus, we are confident a quadrocopter would be able to follow such a path, provided a quality navigation system is available.

Our implementation of the algorithm runs reasonably fast and consumes very little memory. However, we are certain that improvements are possible, especially to optimize the running-time of the algorithm. That could be done either by implementing more advanced data structures or experimenting with heuristic functions, which we already touched on.

We also implemented quadcopter navigation, however, we are not particularly satisfied with it. We expected this to be problematic and the results roughly match our expectations; however, we were hoping for more. Unfortunately, our hopes did not come true.

The results of the navigation are inconsistent even in very simple tests and complex tests result in failure most of the time. In simple tests, the aircraft sometimes lands very close to the target and it is common for it to not follow the path properly. In some cases this is caused by technical issues, in other cases by the shortcomings of our navigation routine. In most cases, however, the aircraft should fly in such a manner that it should be able to detect a marked landing zone if such functionality was available.

In more complex tests, our navigation routine usually fails. This makes us conclude that it is by far not ready for any real-world application. We have made our first experiments and gained experience that we can use in the future. We learned that a simple P (proportional) controller is not enough for the task.

We believe it is possible to improve our navigation routine. Perhaps not to obtain better results in terms of landing close to the target but mostly to make it much more reliable and capable of flying along more complex paths. Extending our navigation routine to use a true PID or even a PI (Proportional-Integral) controller might improve the quality of our controller drastically. Further improvements are possible by solving the problem with navigation data receiving, as the application is capable of sending instructions in a shorter interval than the interval in which it receives new navigation data from the drone.

A pleasant side effect of implementing quadcopter navigation is that we also implemented the option to control the aircraft manually. It is very simple to use and provides a fun experience.

Our project proved to be much bigger and more challenging than we had originally anticipated. The quadcopter navigation is far from perfect and leaves a lot of room for improvement. However, we believe we created a foundation, upon which we and others will hopefully be able to build in the future. At the very least, we gained valuable experience which we share and can benefit from in our future projects or use to help other projects.

# Shortcomings and possible extensions

We are aware of the fact that our project is not perfect and has several shortcomings. In this short chapter, we name the problems we consider to be the most significant. Then we look at the possibilities of future improvements and extensions of our project.

Perhaps the biggest shortcoming of our project is the low quality of the quadcopter navigation component. We believe that the navigation routine could be drastically improved by implementing a true PID or PI controller and solving other issues, as stated in the Conclusion.

We are also quite dissatisfied with some flaws of the editor. Implementing the user interface in DirectX with DirectInput was complicated and took a lot of time. Furthermore, because of using DirectInput, it is not possible to move or resize the application window. We were unable to solve this problem. This does not impact the functionality of the application, however, it impairs the user experience.

Looking at the implementation of the path-planning algorithm, we see potential for optimization, especially in terms of running time. This could be achieved by implementing more advanced data structures. More advanced heuristic functions might also improve the running time of the algorithm.

Our application could be easily adapted to use a different quadcopter controller. Furthermore, if our controller was improved, it might be possible to join our project together with other projects to perform more complex tasks. Overall, there are many possibilities of extending our application, only limited by one's imagination.

# Bibliography

- [1] 3DS Max by Autodesk  
<http://www.autodesk.com/products/3ds-max/overview>
- [2] SketchUp by Trimble Navigation, originally Google Sketchup  
<http://www.sketchup.com/>
- [3] Blender  
<http://www.blender.org/>
- [4] CryEngine  
<http://cryengine.com/>
- [5] Unreal Engine 4  
<https://www.unrealengine.com/>
- [6] ardrone\_autonomy by AutonomyLab  
[https://github.com/AutonomyLab/ardrone\\_autonomy](https://github.com/AutonomyLab/ardrone_autonomy)
- [7] DirectX  
<http://msdn.microsoft.com/en-us/library/windows/desktop/bb219837>
- [8] OpenGL  
<http://www.opengl.org/>
- [9] OGRE  
<http://www.ogre3d.org/>
- [10] Sprite (computer graphics)  
[http://en.wikipedia.org/wiki/Sprite\\_\(computer\\_graphics\)](http://en.wikipedia.org/wiki/Sprite_(computer_graphics))
- [11] DirectInput  
<http://msdn.microsoft.com/en-us/library/windows/desktop/ee416842>
- [12] Microsoft Developer Network, Viewports and Clipping (Direct3D 9) <http://msdn.microsoft.com/en-us/library/windows/desktop/bb206341>
- [13] Command pattern  
[http://en.wikipedia.org/wiki/Command\\_pattern](http://en.wikipedia.org/wiki/Command_pattern)
- [14] Memento pattern  
[http://en.wikipedia.org/wiki/Memento\\_pattern](http://en.wikipedia.org/wiki/Memento_pattern)
- [15] DANIEL, K., NASH, A., KOENIG, S., AND FELNER, A. *Theta\*: Any-Angle Path Planning on Grids*. Journal of Artificial Intelligence Research 39 (2010) 533-579
- [16] BOURKE, Paul. Polyhedra that fill space (without holes) using only translation.  
<http://paulbourke.net/geometry/spacefill/>

- [17] Polygon mesh  
[http://en.wikipedia.org/wiki/Polygon\\_mesh](http://en.wikipedia.org/wiki/Polygon_mesh)
- [18] NILSSON, NILS J.. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998
- [19] CHRPA, Lukáš, OSBORNE, Hugh. *Towards a Trajectory Planning Concept: Augmenting Path Planning Methods by Considering Speed Limit Constraints* Journal of Intelligent and Robotic Systems, 2013
- [20] DELOURA, Mark, et.al. *Game Programming Gems*. Charles river Media, 2000
- [21] NASH, A., KOENIG, S., TOVEY, C. *Lazy Theta\*: Any-Angle Path Planning and Path Length Analysis in 3D*. Third Annual Synopsium on Combinatorial Search, 2010
- [22] PATEL, Amit. *Amit's Thoughts on Pathfinding, Map representations*.  
<http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html>
- [23] SUVANPONG, Pongsak. C++ classes for controlling AR-Drone.  
<http://www.ccs.neu.edu/home/psksvp/AR-Drone.html>
- [24] User Datagram Protocol  
[http://en.wikipedia.org/wiki/User\\_Datagram\\_Protocol](http://en.wikipedia.org/wiki/User_Datagram_Protocol)
- [25] ARDrone API, Developer Guide  
<https://projects.ardrone.org/wiki/ardrone-api>
- [26] QueryPerformanceCounter function, Microsoft Developer Network  
<http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904>

# List of Tables

5.1	Structure of AR.Drone navigation data . . . . .	52
6.1	Straight flight . . . . .	58
6.2	Flight with 30 degree turn . . . . .	59
6.3	Flight with 90 degree turn . . . . .	60



# List of Abbreviations

API - Application Programming Interface

WIN32 API - Windows API

STL - Standard Template Library

UDP - User Datagram Protocol

PID controller - Proportional-Integral-Derivative controller

PI controller - Proportional-Integral controller

# Attachments

## 1. Disc contents

The attached Disc contains the following files:

- *user\_guide.pdf* – The user guide
- *thesis.pdf* – This text
- *setup.exe* – Program installer
- *\source* – Program source code
- *class\_diagram.pdf* – The class diagram

The user guide describes the features of the user interface and functions of buttons and keys. It also guides the user through the installation process.

The installer (file *setup.exe*) unpacks the application and necessary redistributable components. For convenience, it also contains the user guide.

## 2. User guide

Drone3D is a simple application providing a 3D environment editor combined with path-planning and Parrot AR.Drone controlling functionality.

The editor allows for the modelling a room, in which obstacles can be placed. A start location and a goal location for the quadrocopter can also be defined. A model of an environment is often called a map.

Then, the path-planning algorithm can be run, which will find a path between the start and goal locations. Alternatively, a custom path can be defined using waypoints. However, the obstacles are ignored when waypoints are used.

Finally, the AR.Drone navigation can be run, which will use the path to navigate a real AR.Drone. Manual control of the quadrocopter is also possible and during program navigation, it is possible to take over control of the drone.

Combining all the features together, experiments with the Ar.Drone can be performed. For example, an obstacle course can be created in a room and modeled in the application. Then, a start location and a goal location can be defined, which will correspond to locations in the room. Then the application can be used to navigate the quadrocopter from the start to the goal.

### About the guide

This user guide serves as a simple and concise explanation of the program functions, and how to use them.

The application contains many buttons and uses multiple keys on the keyboard. In the guide they are distinguished from standard text by brackets: [button].

# Installation

Installing the application is very simple. However, it also requires DirectX 9 and Visual C++ 2010 redistributable components to be installed on your system.

The application supports the Microsoft Windows Vista, 7 and 8 operating systems.

The installation procedure is as follows:

1. Run *setup.exe*
2. In the setup, choose a destination folder for the application. Default is *C:\Drone3D*
3. Let the setup process finish
4. Open the folder, where you have installed the application
5. Run the DirectX redistributable component setup (file *dxwebsetup.exe*). This requires internet connection.
6. Run the Visual C++ redistributable component setup(file *vc redistrib\_x86.exe*)
7. You are now finished and can run the application (file *Drone3D.exe*)

In step 2 of the installation process, it is recommended to choose a non-protected folder on Windows 7 and 8 operating systems. Otherwise the application must be run in administrator mode to be able to save maps.

# Main menu

When the application is run, the main menu is presented. A frames per second counter is displayed in the top left of the screen.

Figure 6.3: Main menu



The main menu has three items:

**NEW MAP** Opens a dialog which starts the creation of a new map

**LOAD MAP** Opens a dialog which can load a saved map from a file

**EXIT** Exits the application

## New map dialog

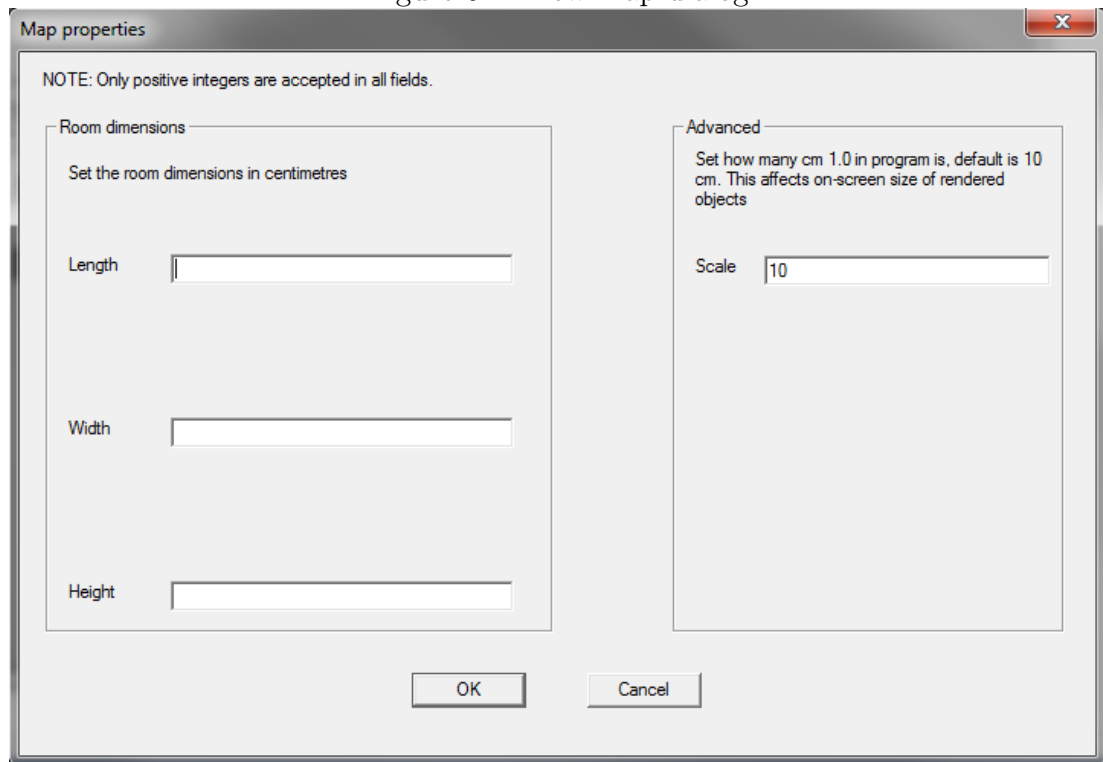
The new map dialog can be seen in the image above. The dialog requires the dimensions of the map to be set in centimetres. You can model an environment of any size, however, we recommend rooms from 5 m by 5 m to 10 m by 10 m for quadcopter applications.

The *scale* parameter can also be set. However, this is an advanced parameter and it is not recommended to change it. The default value is 10. *Scale* provides a connection between the program and the real world, and connects the 1.0 value in the program to the given value in centimeters. By default,  $1.0 = 10$  cm

## Load map dialog

This option runs the standard Windows dialog to open files. By default, the the **Maps** folder in the application directory is open.

Figure 6.4: New map dialog



The image shows a 'Map properties' dialog box with a title bar and a close button. Inside, a note states: 'NOTE: Only positive integers are accepted in all fields.' The dialog is divided into two main sections: 'Room dimensions' and 'Advanced'. The 'Room dimensions' section has the instruction 'Set the room dimensions in centimetres' and three input fields labeled 'Length', 'Width', and 'Height'. The 'Advanced' section has the instruction 'Set how many cm 1.0 in program is, default is 10 cm. This affects on-screen size of rendered objects' and a 'Scale' input field containing the value '10'. At the bottom are 'OK' and 'Cancel' buttons.

Map properties

NOTE: Only positive integers are accepted in all fields.

**Room dimensions**

Set the room dimensions in centimetres

Length

Width

Height

**Advanced**

Set how many cm 1.0 in program is, default is 10 cm. This affects on-screen size of rendered objects

Scale

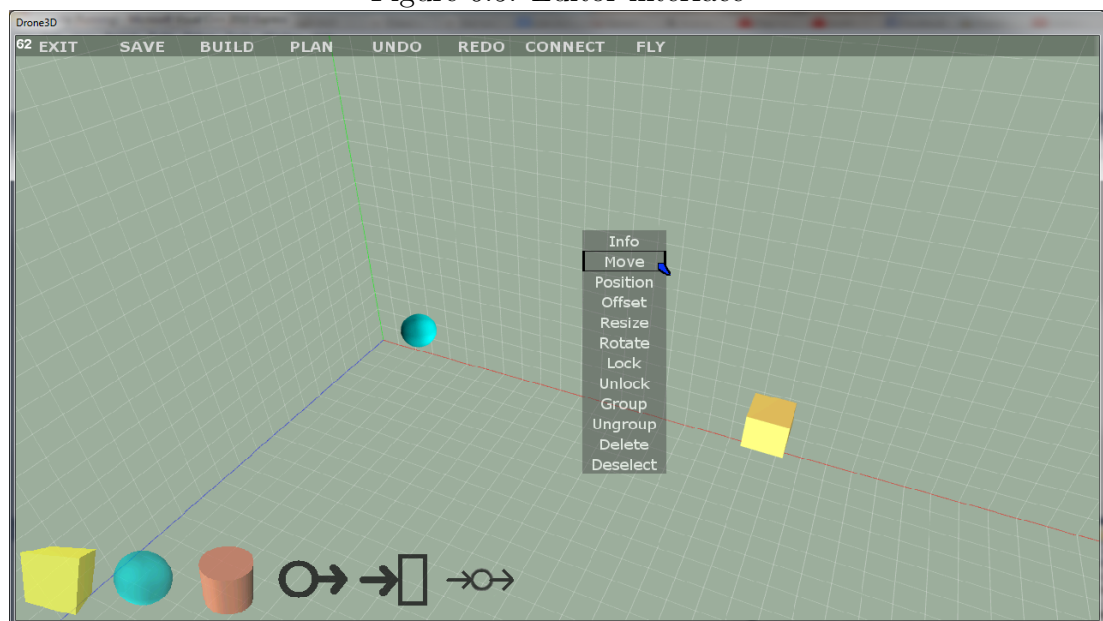
OK Cancel

# Map editor

After creating a new map or loading a map from a file, the editor is shown. The user interface has three parts:

- A **menu bar** on the top of the screen
- An **object bar** on the bottom of the screen
- A **context menu** with operations

Figure 6.5: Editor interface



The **menu bar** contains buttons that let the user navigate in the application and run specific parts of the program like path-planning.

The **object bar** contains buttons that let the user place objects in the environment.

The **context menu** contains buttons that let the user perform operations on objects. It is only displayed when the [right mouse button] is clicked.

## The environment

The environment is a box. The boundaries are represented by the drawn planes. A regular grid is also displayed, with the distance between lines equal to 1.0 in program, which corresponds to the *scale* parameter, which is set when a new map is created.

The coordinate axes are highlighted, the *X* axis is red, the *Y* axis is green, and the *Z* axis is blue. This is to make orientation easier.

## Camera movement

The camera is moved using the classical WASD combo. Hold the [W] key to move the camera forward. The [A] key moves the camera to the left, [S] key backward, and [D] key to the right. You can also use the [R] and [F] keys to move the camera up and down, respectively.

Please note that the direction the camera is moving depends on the camera orientation. So if the camera is looking down, holding the [R] key will actually not move it up in the environment but the camera will move in the direction of its own *up vector*.

The camera can be rotated by holding down the [SHIFT] key and moving the mouse.

## Menu bar functions

Figure 6.6: Menu bar



The menu bar contains multiple buttons with the following functions:

**[EXIT]:** By pressing this button, the editor exits into the main menu. If unsaved changes were made to the map, the user will be prompted to save the map. Either the file name can be set and the button  pressed to save, or the  button to ignore the changes.

**[SAVE]:** This button opens a dialog which lets the user input the name of the map file. Then, the map will be saved in that file in the *Maps* folder in the application directory.

**[BUILD]:** By pressing this button, the environment building is initiated. This transforms the created environment into a space that the path-planning algorithm can then search for a path. As a minimum, start and end points are required.

**[PLAN]:** Pressing this button will start the path-planning algorithm, if the environment has been built. Otherwise it will pop-up a message box saying that the environment must be built first.

**[UNDO]:** If applicable, pressing this button will undo an action in the editor (e.g. moving an object). If the user makes a mistake, he can easily correct it this way.

**[REDO]:** Redo is the inverse of undo. Pressing this button will redo an undone action, if there is any.

**[CONNECT]:** After pressing this button, the application will connect to the parrot AR.Drone. Please note that the computer must be connected to the drone via Wi-Fi, otherwise the program will not be able to establish connection.



[FLY]: After pressing this button, the AR.Drone navigation will be started if a path is available. The path can be either found by the path-planning algorithm or input as waypoints.

## Path-planning

Before the path-planning algorithm can be run, the environment must be built by pressing the [BUILD] button. Then, black cubes will be displayed in place of blocked environment units. This feature can be toggled on and off by pressing the [Z] key.

It is expected that at least 150 centimetres of free space is above both the quadcopter *start* and *goal* locations. The quadcopter automatically takes off into an altitude of 1 metre, thus the algorithm is designed to take this into account.

When the path-planning is run, the application will show a dialog showing the number of expanded units (representing progress) and a button that lets you close the dialog and a button to stop the algorithm.

The algorithm runs in a separate thread, so that the user can interact with the application while it is running. In some cases, it may take several seconds (possibly even minutes) for the algorithm to finish.

## Undo and redo

Undo and redo are functions which can make work with any editor much easier, and thus Drone3D also provides them. If the user makes a mistake, he can correct it easily using those functions.

If the parameters of a model are changed, this change can be undone by pressing the [UNDO] button. In case the button is pressed accidentally, the [REDO] button can be pressed to re-do the change. When the user reverts multiple actions and then performs a new action, the undone actions will be forgotten and it will not be possible to redo them.

Consider an example: The user adds a new model and sets its position. Then he rotates it, and finally makes it twice as big. However, the result did not turn out the way he expected. So he decides to undo the scaling and rotation operations. Then he performs a new scaling operation. At this point, the rotation and first scaling operations are forgotten, and it is not possible to redo them.

## Mouse modes

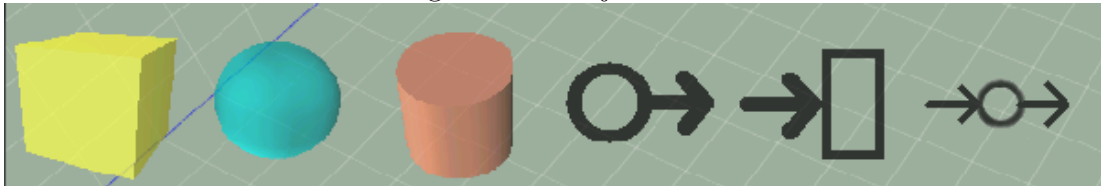
The mouse works in various modes in the application. The default and most common mode is *object selection*. To deselect individual objects, the mouse switches to *deselect* mode. When an object is being moved using the mouse, the mouse is in *movement* mode. Finally, when an object is being added, the mouse is in *object adding* mode.

## Object bar functions

The object bar contains buttons that let the user place objects in the environment. When he clicks a button in the object bar, the application enters *object adding* mode. Each button represents a different object.

After a button in the object bar is clicked, the selected type of object can be placed by clicking in the environment, either at an existing object or one of the three border planes. A preview of the object is displayed, showing the position where it will be placed when the user clicks to confirm. The [ESCAPE] key can be pressed to exit *object adding* mode.

Figure 6.7: Object bar



The object bar contains the following buttons from left to right:

[CUBE]: Place a cube in the environment

[SPHERE]: Place a sphere in the environment

[CYLINDER]: Place a cylinder in the environment

[START]: Set the starting location in the environment

[GOAL]: Set the goal location in the environment

[WAYPOINT]: Place a waypoint in the environment

## Selecting objects

When the mouse is in *object selection* mode, the user can select objects by clicking them with the [left mouse button]. Objects can also be grouped, and when an object in a group is clicked, the entire group is automatically selected.

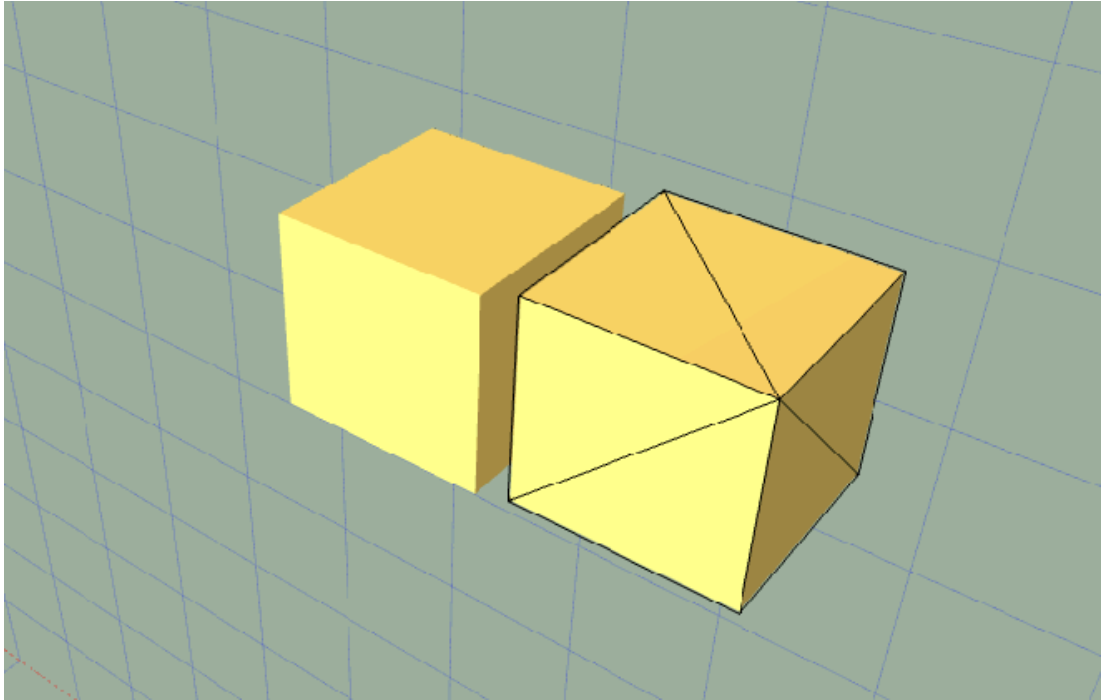
Currently selected objects are highlighted, as can be seen in the following image showing one cube and a selected cube.

If necessary, deselect individual objects can be deselected by holding down the [CTRL] key and clicking them with the [left mouse button]. All objects can be deselected using the [Deselect] button in the context menu, described in the following chapter.

## Context menu functions

The context menu is displayed after the [right mouse button] is clicked. The items it contains depend on the type of selected objects. It is not possible to

Figure 6.8: Selected object highlighting



manipulate with multiple types of objects at once. There are four types of objects: a **standard** object, **start**, **goal** and **waypoint**.

The standard context menu looks like this:

[**Info**]: Displays information about the object.

[**Move**]: Switches the mouse to *movement* mode, then you can move the selected objects with the mouse.

[**Position**]: Opens a dialog which lets you set object position in centimeters. This operation can be only be performed on a single object or a single group.

[**Offset**]: Opens a dialog which lets you set the object offset in centimeters. This is very similar to setting object position. However, offset is the distance between the displayed border planes and the object. Object rotation influences the result of this operation.

[**Resize**]: Opens a dalog, in which the user sets the object size in centimeters. Previous object rotation or size are not taken into account.

[**Rotate**]: Opens a dialog, which lets the user set the object rotation around the  $X$ ,  $Y$  and  $Z$  axes. Previous settings are not taken into account.

[**Lock**]: Locks the object in position.

[**Unlock**]: Unlocks the object for manipulation.

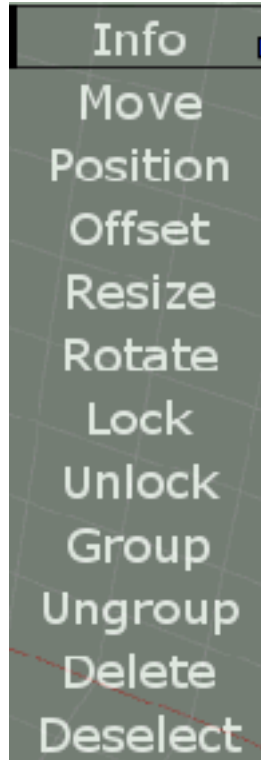
[**Group**]: Creates a group from the selected objects and/or groups.

[**Ungroup**]: Disbands the selected groups into individual objects.

[**Delete**]: Deletes the selected objects or groups.

[**Deselect**]: Deselects all selected objects and groups.

Figure 6.9: Context menu



The **start** and **goal** objects support only some of those operations. The unsupported operations are not shown in their context menus.

The **start** object introduces one new operation, that is [**Direction**] which lets the user set the direction of the drone. This is expected to correspond to the direction the Parrot AR.Drone is facing, when you put it on the ground and initiate navigation.

## Drone control

The application also allows manual control of the drone. After it is connected to the drone, press [T] key to take off. The drone will take off and start hovering in position. The IJKL keys and arrow keys are used to control the aircraft. The [I] key will cause the drone to move forward. The [J] key moves the drone to the left, the [K] key backward and the [L] key to the right.

The [LEFT ARROW] and [RIGHT ARROW] keys turn the drone left and right, while the [UP ARROW] and [DOWN ARROW] keys are for vertical movement. To land, press the [G] key.

Sometimes, the drone sensors need to be reset. To do this, place the aircraft on a flat surface and press the [M] key while connected to the drone to send the flat trim command. The aircraft should reset its sensors.

If no command is sent for over 2 seconds, the drone will stop sending navigation data to the computer. To re-initiate this process, press the [V] key and then the [X] key. However, the navigation data are not important for manual control and the application ensures that the data are being received when it is in control. To verify whether or not the application is truly connected to the drone, press the [B] key. The drone LEDs should blink.

When the application is in control of the drone, control can be aborted and a landing forced by pressing the [SPACEBAR] key. The user can take over control by pressing [ENTER].

## List of used keyboard keys

key	function
[W]	Move the camera forward
[A]	Move the camera to the left
[S]	Move the camera backwards
[D]	Move the camera to the right
[R]	Move the camera up
[F]	Move the camera down
[LEFT SHIFT]	Enable camera rotation
[ESCAPE]	Cancel object adding
[LEFT CTRL]	Switch mouse to deselecting mode
[O]	Undo
[P]	Redo
[Z]	Toggle display of blocked environment units on or off
[T]	Quadrocopter take off
[G]	Quadrocopter land
[I]	Quadrocopter forward
[J]	Quadrocopter roll left
[K]	Quadrocopter backwards
[L]	Quadrocopter roll right
[UP ARROW]	Quadrocopter increase altitude
[LEFT ARROW]	Quadrocopter turn left
[DOWN ARROW]	Quadrocopter decrease altitude
[RIGHT ARROW]	Quadrocopter turn right
[B]	Quadrocopter blink LEDs
[N]	Disconnect from the quadrocopter
[M]	Send the flat trim instruction to the quadrocopter
[V]	Send the watch dog reset command to the quadrocopter
[X]	Request navigation data from the quadrocopter
[SPACEBAR]	Abort quadrocopter navigation and force a landing
[ENTER]	Take over control of the quadrocopter