Bachelor Thesis

Google Loon Control Systems

Author: Stefan Witold Mertens (s113420)
Supervisor: Jan Madsen

30.6.2014

I would like to thank Jan Madsen
for his helpful guidance throughout the project.

Contents

1 Abstract

The primary purpose of this study is to simulate a number of balloons moving in the stratosphere, similar to Google's *Project Loon*. A basic de-centralized steering algorithm is developed for these balloons, and evaluated using the simulation. Experiments are carried out by manipulation of the balloons' parameters, and their influence on balloon behaviour is observed.

2 Introduction

Project Loon is an ambitious project being developed by Google X, a facility run by Google which is dedicated to major technological innovation. The idea is a world wide network of balloons travelling in the stratosphere, above air traffic and weather, designed to connect the 4 billion people without internet to the world, especially in remote rural regions. The project is not only called loon as an abbreviation, it is, similar to many Google X projects, very difficult, costly to realise and might seem crazy at first. Nevertheless, many of these projects become real products one day, as e.g. the driver-less car or Google Glass.
The project is undeniable massive, because a global network would require many thousands balloons at once. To verify if such a project is even possible, simple flight tests are conducted, but these cannot say much about the behaviour of many balloons at once. To examine large scale questions like "Can the balloons spread out evenly in stratosphere winds, or will they not be able to control their flight?", simulation is a useful and cheap tool. This thesis aims at creating a simulated environment and a flight control system to be able to answer exactly that question.

3 Modelling the problem

3.1 Introduction

Translating this problem into a correct software model is a critical part of this thesis, as a correct model is a prerequisite for a correct simulation, control system and steering algorithm. It has to be noted that a completely realistic model is out of scope for this thesis. Simplifications and abstractions are made everywhere, and some variables have to be omitted all together.

Similar to Google's simulation seen in an "Ask Away" video [1], the scope for this simulation is an arbitrary 1000 km^2. No terrain is taken into account and every point in the area is weighted equally in regards to coverage priority. Balloons are spawned randomly into that area, and will try to spread out evenly to provide the largest possible internet coverage area. Balloons will only change their altitude to reach different stratosphere wind layers with different wind directions in order to navigate.

3.2 The wind model

The viability of the whole Loon project cannot and will not be determined by this control system's implementation alone - it depends heavily on conditions in the real world, most importantly air movements in the stratosphere.

The only *real* data used in the wind model is facts about average wind speed and direction, for a few reasons: General visualizations of stratosphere winds are found easily on the internet [2], but most of that data is for a specific pressure level only, and does not provide any information about the wind layers in the stratosphere, i.e. it is not detailed enough for this simulation. The American NOAA [3] provides wind data like this, but only in complex and very specific or big data sets. Going through hundreds of gigabytes of data is not a viable option for this thesis. For these reasons, wind has to be randomly generated and simulated. This does have one big advantage when compared to the use of real data: Many different wind patterns and extreme values can be tested in quick succession.

Assumptions

A few assumptions have to be made in order to implement realistic simulated wind:

1. Wind in the stratosphere can be divided into vertical layers, each mostly independent with their own wind velocities and directions
2. The average wind direction in each of these layers deviates from the others enough for the balloons to navigate
3. Balloons do not get stuck, they will always have a chance to move away from their current position.

Note that (3.) is mainly assumed for convenience, as the situation might occur in the real world. In that situation, the balloon control would be powerless no matter what, and balloons getting stuck because of very specific wind situations is a variable this simulation does not need in order to analyse a control algorithm.

3.2.1 Parameters

For this simulation, seven wind layers are randomly generated. These wind layers expand over one kilometre horizontally, and there is no interpolation between layers. This means that a balloon at 19.1 km height will have the exact same speed and direction as a balloon at the same GPS coordinates but at 19.9 km height. This is gross simplification, but it does not actually influence the balloons' control algorithm, which allows for an arbitrary number of wind layers or subdivisions. The 1 km for a wind layer is an arbitrary value and would probably vary a lot in reality, but the only dependant parameters are balloon ascent and descent speed, which are easy to modify.

The hard edges problem A problem with this choice is a situation which will occur regularly while balloons are navigating the wind: A balloon will be able to change between wind layers very quickly by just staying on the edge of one and moving to another in virtually no time. To circumvent this problem, balloons will always stay in the middle of their wind layer while not navigating to another, so their distance to the layer above and below are the same.
This only weakens the problem, because a balloon still may hover between two layers if the steering algorithm wants the balloon to alternate between them and

they are next to each other. There are ways to solve the problem, but each has their disadvantages:

- The balloons steering could be restricted, in a way which forces them to move to the middle of one layer before choosing a new target altitude. This would solve the problem, but make the steering algorithm much less flexible. A balloon should be able to change altitude at any time, not only at set intervals.
- The wind field could be interpolated between layers on the vertical axis. This would also solve the problem by just removing the hard edges between wind layers and replacing them with soft transitions. The problem with this approach is the even more unrealistic representation of real wind, and the fact that balloons would be able to freely choose their direction. This defeats the purpose of having specific wind layers.

It is clear that each of these solutions has their problems. An ideal solution would probably have either very accurate real wind data or define a transition gradient between layers, or even simulate turbulence. These possibilities are too complex for this simple simulation, hence the wind is simulated with hard edges and the aforementioned problem might occur. It only occurs in specific situations and does not have a huge impact on balloon behaviour, and is thus deemed acceptable for this simulation.

3.2.2 General properties

Taking a closer look at stratosphere winds (See [2]) will reveal the fact that stratosphere winds mostly move in a similar direction in any 1000 km^2 area, with a few exceptions. It is assumed the chosen area will be void of these exceptions and will not have extreme changes in wind velocity or direction over the simulation's area.

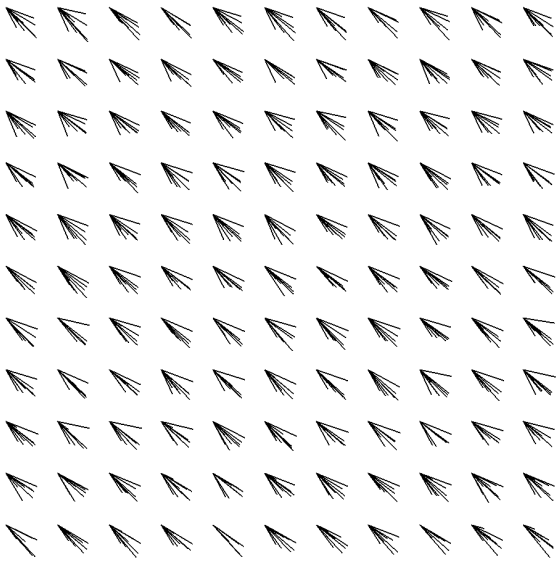
Each wind layer contains a 100 $km \cdot 100$ km spaced grid of vectors, which show the winds velocity and direction on the horizontal plane. Collectively, all these vectors in every layer shall be defined as *the wind field*. Note the fact that this does not comply with the mathematical definition of a vector field, because it does not define a vector for every point in the space. Instead, horizontal bilinear interpolation is used to find wind vectors in between the grid points, read more about that in section 5.6.

3.2.3 Wind terminology

The simulation is designed around two types of wind fields:

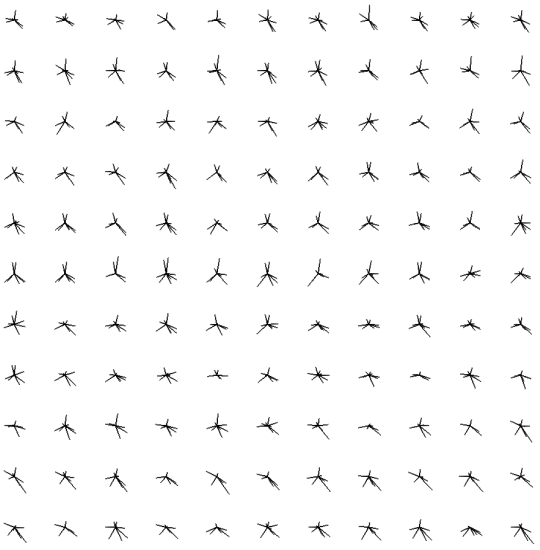
Absolute wind An *absolute wind field* resembles real wind data. Every wind vector points in a roughly similar direction, with a variance of about $\pm 45°$. Their velocities vary by about 50%. The average direction is set to south-east by default to ensure movement on two axes. See Figure 1 for a visual example. If not specified otherwise, *wind field* refers to this type.

Figure 1: Example absolute wind field, view is from top. All 7 wind layers' vectors are drawn on top of each other.



Relative wind A *relative wind field* is made to resemble an absolute wind field, where the average of all vectors is subtracted from every vector, thereby only showing deviations from the average wind instead of absolute wind vectors. This wind field is useful to test the steering algorithm in a very diffuse wind field. The steering algorithm will also translate an absolute wind field to a relative one in order to determine movement relative to other balloons, see subsection 5.6.
Such a wind field is illustrated in Figure 2.

Figure 2: Example relative wind field, view is from top. All 7 wind layers' vectors are drawn on top of each other.



3.3 The Balloon Model

The balloons are modelled as points in 3-dimensional space with no volume or weight, thus they cannot collide or directly influence each other directly in any way. Real balloons are not big when compared to the average distance between balloons, so a collision will be very unlikely, and even if one occurs, it is not clear what would happen. It would be possible to assume both balloons have a chance to crash when they collide, but the added randomization would not benefit a consistent and controlled simulation.
Balloons may be removed or added at any point during the simulation, which can be used to simulate crashing balloons or the launch of additional balloons.

3.3.1 Parameters

The simulated balloons have the following parameters:

Coverage Radius This is the the radius in which a balloon can provide internet coverage to the ground. The default value for this is 20 km, as stated in the Google Loon FAQ [4].

Communication Radius This is the radius in which a balloon can communicate and share internet connectivity with other balloons horizontally. The default value for this is 80 km, which is just an approximation, although mentioned by Google. It is not clear whether or not this value is actually relevant, because Google also mentions ground stations every 100 km, which are connected to the internet and can relay that connections to balloons [5].
For the simulation it is assumed that not every balloon is connected to such a ground station, and, as a compromise, a balloon that does not have any other balloon within its communication radius will be assumed disconnected. In that case, the balloon's coverage radius will not count towards internet coverage.

Detection Radius This is the radius in which a balloon can detect other balloons in its vicinity. This value is not meant to be the longest possible detection distance, but the distance in which the steering algorithm will consider other balloons.
In reality, this value could always be at least equal to or greater than the communication distance, because a balloon with a network connection to another will always be able to just send its position via that connection. It is assumed that this distance may be far greater, because even a weak or very slow connection should be enough to communicate simple coordinates. This parameter is not directly a balloon's property, but a setting for a balloon's steering.

Ascent and Descent Rate This is how fast a balloon can rise and fall in altitude. For simplicity, if not mentioned otherwise, these speeds are assumed to be identical. These speeds are assumed to be around 1 to 10 m/s, which corresponds to general data about balloon ascent and descent speeds [6, 7, 8]. The way balloons navigate the wind field is to adjust their altitude, because wind direction and velocity will be different at each altitude. It is clear that if a balloon cannot ascend or descend fast enough, it may not be able to reach a specific altitude before it has to go in a new

direction and change altitude again. Therefore, ascent and descent rate is a critical parameter for balloon steering behaviour.

Others There are many other parameters which could be relevant for an accurate simulation, such as energy consumption and regeneration. They are intentionally omitted from the simulation because they are either: non-influential on balloon steering in most situations, unknown, or not relevant enough to justify the added complexity in the steering algorithm or the simulation in general.

This simulation serves mostly to answer the question: "Is it possible to spread balloons out evenly and provide stable coverage?". The balloon's temperature or size are good examples for parameters that would be essential to know in a real test, but are neglectable in this simulation.

3.4 Summary and simplifications

The choice of scope for this simulation is an 1000 km^2 area with no relation to a real geographical area. If a balloon leaves this area on one side, it will be reset to the opposite side of the simulation. This effectively simulates an area that is part of a bigger world with more of these areas and balloons, and enables a continuous simulation. Also see subsection 7.3 for an explanation as to why this is a good idea. In general, a balloon will fly around the world a few times in its 100 day lifespan [9]. Therefore, there is no way of directly steering a balloon in all directions, only smaller deviations from the general wind direction are possible. In general, wind data is assumed not to change over the course of a simulation, but a good steering algorithm must allow for changing wind situations.
Other possible scopes are the whole globe, or an area with variable or not rectangular boundaries. Both these possibilities make it significantly harder to implement the simulation while not providing any big advantages, therefore the best scope for the simulation is the square area.

4 Design

4.1 Program design

The simulation program itself is designed in what vaguely resembles the classic model-view-controller (MVC)[10] software model. The the definitions and responsibilities are a bit different for this program, because the program is not a classic business application, but a simulation and a graphical representation of said simulation, with minimal user input and no data persistence.

Model For the simulation software, the model is defined as the wind and balloon object classes, with each their parameters, which in turn closely resemble those defined in the modelling section above. The simulation will use instances of these objects, and they each contain code which allows them to render to the graphical output. These objects each have their own *Render* and *Update* functions, which the simulation and the renderer will call in order to simulate and draw the objects. Thus, the simulation and the renderer can use the same objects; the simulation will

simply pass the list of objects on to the renderer, which then will call the *Render* function of each object.

Simulation In a classic MVC model this would be a *Controller*, but the original definition does not carry over to this program very well, as mentioned. Therefore, the controller is simply replaced by what essentially is only one class: *Simulation*. This class has wind data and an array of balloon objects, and, as the name suggests, steers the simulation. It does this by providing application logic such as deleting and adding balloons, while also updating every balloon object in every simulation cycle. This class, together with the model is completely independent of any graphical output and could be run on its own with the same results.

View The view corresponds nicely to the original MVC idea, because it is just a user interface which renders the simulation, but does not know anything about simulation logic. This GUI consists of a form with an embedded OpenTk canvas (See section 5.1), which is used to actually render the balloons in 3D. Even though the balloons are truly rendered in 3D, the camera will stay in fixed position, therefore the graphical output will resemble a 2-dimensional implementation. The camera could be rotated freely, but in practice this is more confusing than helpful, and it might be difficult to reset the camera to a neutral position. Also, a side view does not actually provide any useful information the top view would not. See subsubsection 4.2.1 for more details.

Summary Each of these parts are as loosely coupled as possible, making changes in one mostly independent of other code, e.g. the simulation could run completely on it's own, and changes to the simulation will not require changes in the view.

4.2 User interface

The user interface is divided into two windows, seen in Figure 3 and Figure 6. These are:

4.2.1 Graphics output

The black area in Figure 3 is the rendered 3D output, where all the balloons are displayed while navigating the wind field. The output is actually 3D, but with a fixed camera. The has a few advantages, as it is easy to draw elements in the right order, or see which balloon is further up if two are very close.

Orthographic vs. perspective projection The human eye naturally perceives the world in a perspective projection. On the other hand, an orthographic projection will project 3-dimensional objects on a 2-dimensional plane in parallel[11]. That way, every balloon will have the same size, no matter how far away or close to the edges it is.

In practice there is no big difference between the two projections in this simulation, because the camera is high up and parallel to the ground plane. The orthographic projection has one big advantage, which makes it the choice for the simulation: It will accurately show additional graphics like the simulation border in relation to the

Figure 3: The main user interface

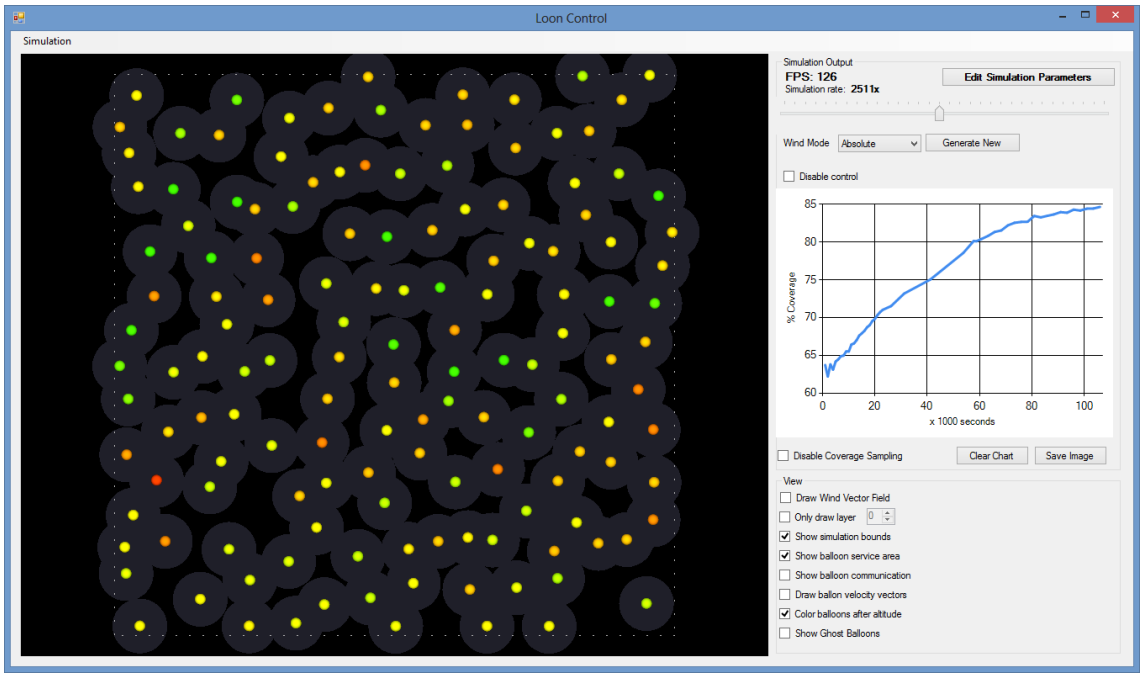
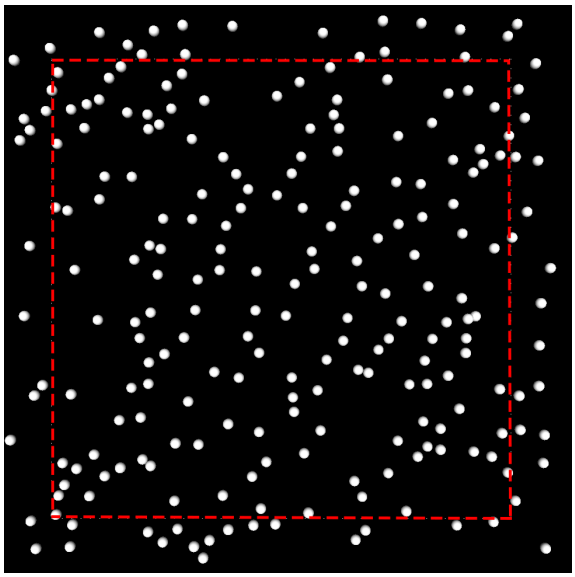


Figure 4: A perspective view of the simulation, the stippled line indicates the simulation boundaries drawn at ground level.

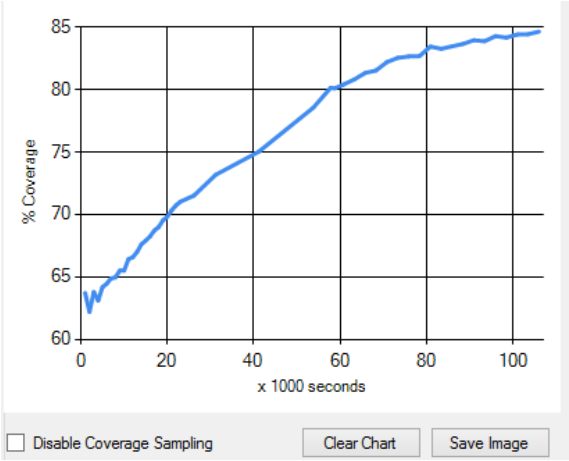


balloons, even if they are not truly 3-dimensional and/or not drawn at the same height as the balloons. Figure 4 shows the problem: the simulation border is drawn at ground level, and every balloon's coordinates are actually inside that border, but because they are higher up it is difficult to see whether or not they actually are inside. To accurately show the simulation bounds they would have to be drawn as a 3-dimensional box instead, and more visual cues would have to be added, like balloon shadows or a moving camera.
In summary, an orthographic perspective shows everything just as well and is less confusing, because the graphical output does not actually benefit from being shown in a true 3-dimensional perspective.

4.2.2 Basic simulation settings, graphics settings and coverage graph

Simulation settings The first few GUI elements in the panel to the right on the main window (see Figure 3) control the basic simulation parameters, such as the time scale. The simulation can be paused altogether, or sped up to about 60000 times real time. Using the button and drop-down menu below, different wind fields can be generated, either pointing in any direction (relative) or mostly pointing in similar directions with small variations (absolute). Absolute is the default setting, because such a wind field will most resemble a real situation. The *Generate New* button will generate new wind vectors at any time, even when the simulation is paused or running. This is useful to manually simulate a changing wind field. In the same section, there is a check box which allows to disable balloon steering altogether. This enables the user to compare controlled and uncontrolled balloons, and to see how balloons behave when they stay at the same altitude and just fly with the wind.

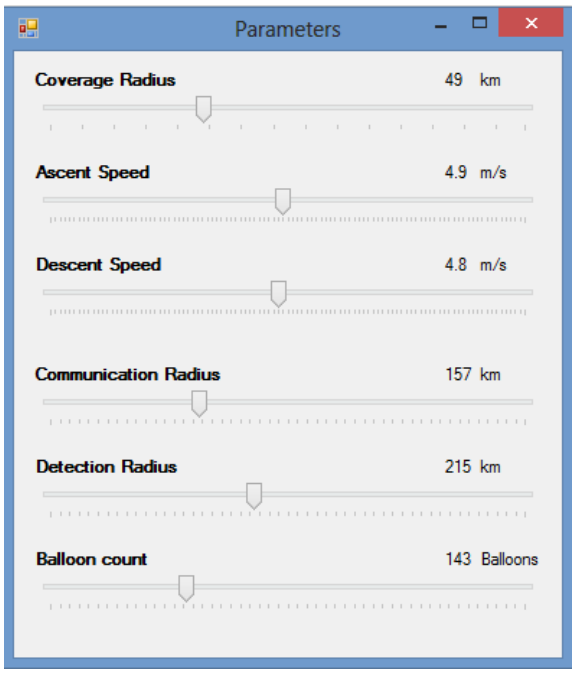
Figure 5: Simulation time step and coverage graph



Coverage chart Figure 5 shows the panel's next section, the coverage chart. One of the most useful metrics to determine how effective a steering algorithm is, is how much internet coverage balloons provide to the ground. The simulation includes a simple coverage calculation, which the program will run once every real time second and display in the chart, also see subsection 5.7. Together with the parameter window described below, it makes it easy to see how different parameters or just balloon steering influence coverage over time.
Two small buttons allow the user to clear and restart the chart or to save the chart as an image. Additionally, coverage sampling may be disabled to pause the chart, or if CPU performance is an issue.

Display settings This section is the panel's bottom part. Multiple check boxes allow to enable and disable different graphic helpers for the graphics output, such as displaying the balloons' coverage area, drawing the wind vectors or colouring balloons depending on their altitude. These settings are very useful in order to analyse the simulation or find potential problems.

Figure 6: The parameters window



4.2.3 Parameters Window

The parameters window is opened by the *Edit simulation parameters* button and can be seen in Figure 6. It is separated from the main window for two reasons:

1. To save screen space when it is not used
2. The simulation parameter sliders are the only GUI element which directly influences balloons' parameters and their steering algorithm, and it makes sense to visually separate them from the GUI's other controls.

Arguably the GUI's most important part, this is small window which has sliders for most of the balloons' important parameters. These can be changed at any time, even while the simulation is running, which is very useful to see any parameter's influence on the simulation. These settings are saved and are persistent, i.e. if the user closes the program and opens it up again, the settings will stay the same.

5 Implementation

5.1 Software and Frameworks used

Programming language C# is the language of choice for the implementation, first of all because of prior experience with the language and the *.NET Framework*. The biggest disadvantage of this choice is the loss of multi-platform support when using the .NET Framework. A multi platform version is possible through the mono framework[12], but was omitted for this simulation because multi-platform support is no priority.

.NET and Windows Forms With C# comes the .NET Framework, and with the .NET Framework comes *Windows Forms*, which is Microsoft's graphical interface API for .NET. An alternative exists in the form of the *Windows Presentation Foundation (WPF)*, but Windows Forms was chosen because of the extremely rapid GUI development in Microsoft Visual Studio. Windows Forms allows GUI development in literal minutes and helps shift development focus to the program's more important parts.

OpenGL and OpenTk The graphics are a important part of the program and do not only have to be good-looking and clear, but also fast. There are two big modern graphic standards - *DirectX* and *OpenGL*. OpenGL has one big advantage, as it provides multi platform support, and even though the simulation does not provide that, multi platform support would be easy to add with OpenGL and difficult with DirectX later on. Seeing no other big difference, OpenGL is used in this program. OpenGL cannot be used directly in C#, a wrapper framework is necessary. Luckily, OpenTk [13] is an excellent wrapper for OpenGL in C#, and is thus the final choice for a graphics framework.

5.2 Render and update loop

Just like in most computer games, the separation between graphics and application logic requires a main application loop which basically runs as fast as possible and calls two essential functions as often as possible: one updates every model and calculates the next simulation step, the other draws the next graphics frame to the screen. In the program, the main form actually creates both the renderer and the simulation and keeps updating them, this is adapted from an excellent tutorial on OpenTk inside Windows Forms[14].

Time independent simulation The renderer measures the amount of milliseconds it takes to render the last frame and reports that number back to the simulation, which will then multiply balloons' velocity by that amount. This effectively drives the simulation forward δ seconds, where δ is the number of milliseconds the last frame took to render. Because the default time unit in the simulation is seconds, the default simulation rate is 1000 times real speed, the amount of milliseconds in a second. This technique makes the simulation run at the same speed, no matter how slow the graphics may render, and the simulation will be accurate even if the graphics lag behind.

The simulation speed slider in the GUI simply multiplies δ to reach the desired simulation step. The simulation step is $1000ms/\delta$ and indicates how often the simulation will update per second. This can be problematic if velocities become so large that balloons move too much per simulation step, which will make movement less precise and also multiply rounding errors. For accurate simulation results, the user should choose the smallest acceptable time multiplier.

Simulation update step Each simulation update does the following:

1. Check if the number of balloons in the simulation is the same as the one set in the parameter window, and adjust accordingly.

2. For each balloon, call the balloon's update function which will

   (a) Read balloon parameters from settings

   (b) Check if the balloon is outside the simulation bounds, and reset it to the other side if it is

   (c) Find closest balloons and adjust vertical velocity according to the steering algorithm (No acceleration)

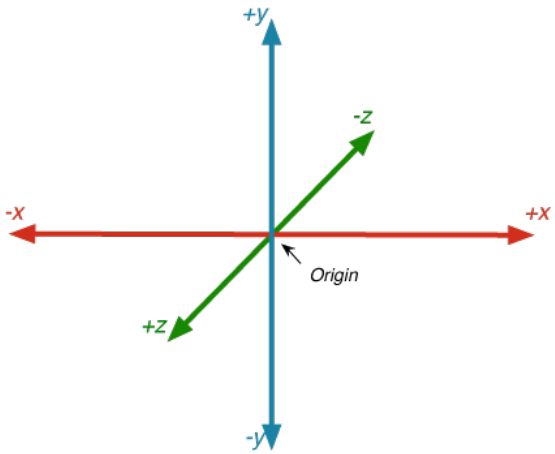
   (d) Simply add $velocity \cdot \delta$ to position to find the new position

2 (c) is where the actual balloon control has to determine which altitude is ideal for the balloon. It is detailed in subsection 5.6.

5.3 The coordinate system

Before the model is implemented, it is important to decide on common implementation parameters, such as the coordinate system. The default cartesian coordinate system in mathematics is a right-handed one with the z-axis pointing upwards. On the other hand, *OpenGL* and physics mostly use similar coordinates, but with the y-axis pointing upwards by default[15]. Every coordinate in the program is in these *OpenGL* coordinates. It would be easy to change this to z-axis upwards, one would just have to rotate all coordinates, as the coordinate systems are symmetrical. This is just a naming convention and does not change any calculations.

Figure 7: OpenGL coordinates, the coordinate system used in the simulation. Source: `http://wiki.stupeflix.com/doku.php?id=geometricanimations`



5.4 Rendering implementation

Rendering the simulation in 3-dimensional space is very practical, because the render logic may share or have proportional coordinates to the simulation.
The rendering code is neither particularly interesting or relevant to this thesis, most graphics code is organized in the *OpenGlRenderer* class, which paints all graphic elements according to the program's settings. The balloons each have their own render methods, and because they are rendered as 3-dimensional spheres, drawing them takes the longest time of all objects in the scene. To improve this time, their

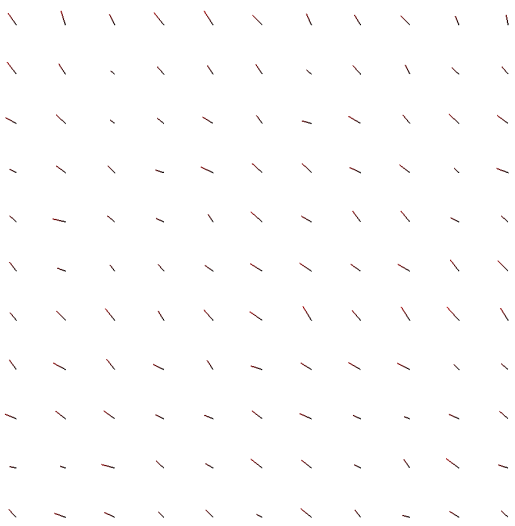
models are stored in a *Vertex Buffer Object*, which directly writes 3D vertices to the graphics card, speeding up the rendering significantly.

5.5 Wind generation

The generation algorithm ensures that the wind vectors are similar within a layer, and deviate from the layers above and below. This leads to a plausible vector field of wind. This wind vector field could also be read from real-world data if the simulation should be expanded in the future. These are the steps for wind generation:

1. One completely random vector is generated per layer, the *start vector*.
2. For each vector in that layer:
   (a) Set the vector to the layers start vector plus a random deviation
   (b) Add a little random deviation to the start vector

Figure 8: Generated wind vectors for a specific layer



This ensures a wind layer will look similar to Figure 8. This does not guarantee a vector in every direction at every point on the grid, which probably corresponds to real winds in the stratosphere. Because there are seven layers, the probability is fairly high that not all the vectors point in similar directions, and the balloons can successfully navigate. It also resembles the wind vectors shown by Google in an "Ask Away" video [16]. In conclusion, it provides a good base for a wind model and is good enough for this simulation. If these vectors were read from real weather data for more sophisticated simulations, the balloon's steering algorithm could remain the same, see subsection 5.6.

5.6 Balloon Control

This is the most essential part of the implementation, because it controls how balloons navigate the wind.

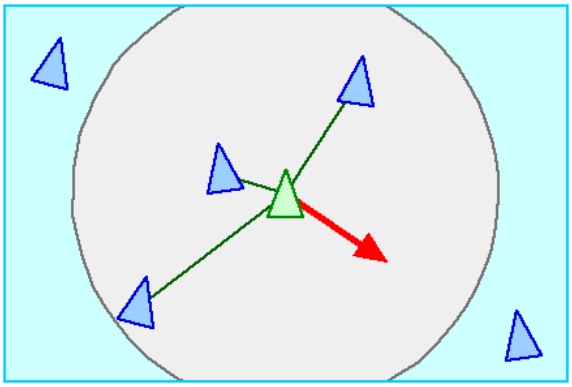
As mentioned, balloons have to control themselves, there must not be any central control scheme. When presented with the problem of having to spread out objects evenly in a wind field, flocks of birds or fish immediately come to mind. The intended balloon behaviour does indeed resemble their behaviour, and this behaviour is called *flocking*. Further research on the topic [17] [18] shows that flocking behaviour normally is divided into 3 behaviour patterns:

- Separation: entities steer to avoid each other
- Alignment: entities steer to go into roughly similar directions
- Cohesion: entities steer to stay together

The purpose of this steering algorithm clearly prioritizes separation, and alignment will be provided by the wind field. Balloons which have no other balloons in their communication range will try to steer towards other balloons, thereby cohesion is also provided. Assuming every point in the area is equally important, balloons which repel each other will actually work fairly well for even area coverage. It is important to remember that there is no immediate control over the balloons, they cannot fly in arbitrary directions. Instead, they have to choose the best wind layer for their intended direction, as detailed in the section below.

Separation The basic separation algorithm is adapted from one of the biggest influences on the steering algorithm and one of the best sources about steering behaviours for autonomous characters, a paper by Craig W. Reynolds [19]. The separation algorithm for the balloons resembles Figure 9, which is Figure 14 in the aforementioned paper:

Figure 9: Separation algorithm illustrated



1. A balloon, called B, will find all other balloons in its communication radius
2. Create a new vector *target* and initialize it to $\vec{O}$
3. For every balloon in that radius:
   (a) Substract its position from B's position to find the vector between them.
   (b) Divide that vector by r^2, where r is the distance between B and the other balloon

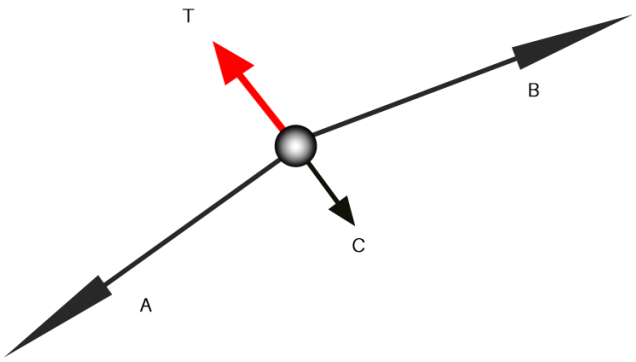
(c) Add the resulting vector to *target*.

4. Normalize *target*

Now the balloon knows which direction it wants to take. But the wind field restricts the balloon's movement into only certain directions. The amount of wind layers determines how many directions the balloon has to choose from. After calculating *target*, a balloon will find wind directions in all layers at its current position. Then, the average of these directions is subtracted from each one and they are normalized. After that, the vector pointing in the most similar direction to *target* can simply be found by adding each normalized wind vector to *target* and then finding the resulting vector's length. This length will be between 0 (vector pointing at $target^{-1}$) and 2 (vector = *target*). The largest result determines which wind layer the balloon wants to move to, and it will start moving either up or down to reach that layer, or stay if it already is at the best altitude.

The velocity problem This method provides reliable results, but has a problem if wind speed (wind vector length) is very varied. Figure 10 shows the problem visually. Imagine a wind field with only three layers. The balloon can choose between the vectors A, B and C, and T is the direction it actually wants to go.

Figure 10: The wind velocity variance problem illustrated



Because the wind vectors are averaged and normalized, the steering algorithm does not consider their magnitude at all. This means that in the situation shown in Figure 10, the balloon will actually choose A, because its direction is most similar to T. Arguably, C would be a better choice, because wind C will probably let the balloon stay closer to the position it wants to be.
In practice, this situation rarely happens, because the generated wind vectors do not have the extreme differences in velocity that create this problem. It is assumed that this will be similar in real life wind scenarios, and therefore will not be an issue when real wind data is used for the simulation or balloon control.

Potential improvements Another problem with this approach is the fact that close wind layers are not prioritized, which may lead to balloons going through many

changes in altitude just to find a direction that is only a little bit better than their current layer. As mentioned in subsection 3.3, energy consumption is not considered in this simulation, but that does not change the fact that this behaviour is not always ideal. Ascent and descent speed, which obviously are a part of this simulation, can be problematic too: While adjusting its altitude, a balloon is still driven by the winds, and might reach a position where the target layer it found before is not the best any more.
For a very accurate simulation and an optimal steering algorithm, balloons would have to predict an ideal position for a given time, and then find the optimal path through the full 3-dimensional wind field. This solution is much more complex and will be prone to other errors and might have difficulties with constantly changing wind fields. For these reasons, and because this simulation's goal is to build a *simple* steering algorithm, the original method is used in the simulation.

Cohesion If the balloons just repel each other and only a few balloons are in the area, the balloons will drift away from each other over time and lose their connection. This is clearly not the intended behaviour. The solution to this problem is fairly simple: If balloons get to the edge of their communication radius to another balloon or have no connection at all, the balloon's target vector is simply inverted on the horizontal plane, making the balloon try to get back towards other balloons. The minimum distance for this behaviour is an estimated percentage, which is one of the few parameters in the simulation which could be changed continuously, but instead are left at a fixed value that works well: it is set to 90% communication distance.

Wind at a balloon's exact position As seen in Figure 8 and described before, not every point in the simulation area has a wind vector, only certain grid points do, by default every 100 km. To find wind vectors in between these grid points, *bilinear interpolation* is necessary. This is rather easy, because the OpenTk framework *Vector2* class has a built-in linear interpolation function. Bilinear interpolation between four vectors is just linear interpolation between each pair of vectors, and then interpolation between the two results. This results in an accurate interpolation between the four wind vectors, and since every point in the simulation is surrounded by four wind grid points, accurate wind interpolation can be found for any balloon's position. Especially for wind vector fields, bilinear interpolation is not the best interpolation method [20], but it is sufficient for this simplified simulation.

5.7 Coverage sampling

To find out how much of the simulation area is covered by the balloons in regard to internet connection, the area inside every balloons coverage range has to be considered. This means that the area of many potentially overlapping circles has to be calculated, which makes this a non-trivial problem [21]. It is actually faster and easier to implement this search by sampling, which means to divide the simulation area in a large number of points (samples), and checking for each of these points whether or not they are covered by any balloon. Then, dividing the number of covered samples by those uncovered will give a good approximation of the covered area.

$$\frac{samples\ inside}{total\ samples} \cdot 100\ \% = \%\ area\ covered$$

Without having to implement any of the complex algorithms mentioned in [21], this sampling can still easily be verified because of the graphical output and a little help from any program that can count pixels of a specific color. Just taking a screen shot of the graphical representation with coverage area display enabled, loading it into such a program and counting the amount of black pixels (the background color), then using this formula

$$(1 - \frac{black\ pixels}{total\ pixels}) \cdot 100\ \% = \%\ area\ covered$$

will also output a very accurate estimate of the covered area. Experiments show that sampling is almost as accurate as this method, thereby confirming a viable solution to the problem. Alas, sampling is a very slow algorithm, because it has to find the nearest neighbour in the array of balloons for every sample. Therefore, the sampling function is run in a different thread, to avoid it slowing down the simulation or graphics. Because both the main simulation thread and the sampling thread will try to access the complete array of balloons at various times and sometimes simultaneously, the array of balloons has to be thread safe, i.e. only one thread at a time may access it. In C# this is very easy to implement compared to other languages: an empty object can serve as a lock object, and critical sections just lock and unlock this object, making the other critical sections wait if the object is in use. One might think that this mechanic would also stop the program while it is sampling, but the sampling calculation is first in the update loop and the frame will still be rendered, which might take as much time as the calculation. Therefore, unless the sampling starts exactly when the simulation update function is called, a whole frame may be updated and displayed before the simulation thread has to wait for sampling to finish.
This does not sound like much, but in practice it means the difference between the graphics stuttering or running smoothly.

6 Experiments

In the finished simulation, various parameters can be changed to simulate and analyse different scenarios and the impact of different parameters on balloon behaviour. Experiments are carried out for a few different scenarios; these do not account for every possible situation and parameter, but are a few especially interesting scenarios, or parameter changes to confirm the assumptions made about balloon behaviour in earlier sections, such as the influence of ascent and descent rate.

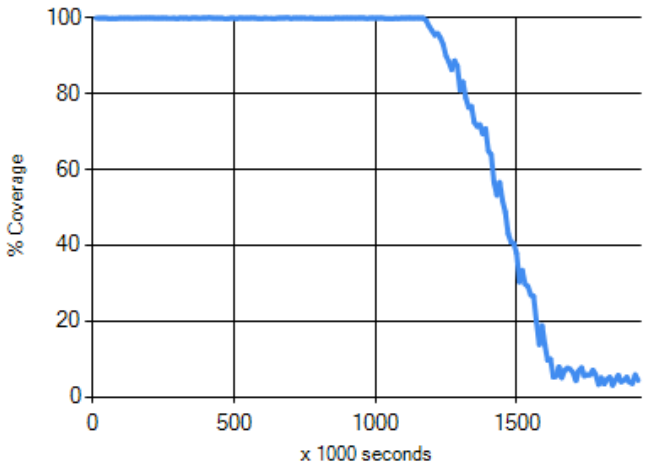
Does the algorithm work? This question should be answered by now, but it is good practice to confirm the most basic assumptions if possible, and it is reassuring to see that the steering is in fact able to spread the balloons out evenly in varying wind conditions, and provides consistently more ground coverage than uncontrolled balloons.
To carry out this simulation, somewhat realistic parameters are chosen (the default values mentioned in subsection 3.3), and control will be switched off and on in different wind fields.

Result The result is very satisfactory, the balloon steering algorithm provides stable coverage and good stability when compared to uncontrolled balloons. No wind vector field seems to be a particular problem, which confirms the assumption made earlier: the steering algorithm will work in any generated wind field. There is a video showing this experiment [22].

How does communication distance influence coverage? Communication distance is the maximum distance a balloon can share internet connections with other balloons. As described in subsection 5.6, a balloon with no or close to no connection to other balloons will actually fly towards other balloons to regain connection. Therefore, balloons with a smaller communication distance will tend to clump together, and as long as the communication distance is large enough it will not influence coverage. To carry out this simulation, 250 balloons with a large coverage area (50 km) will be used. Their communication will be gradually reduced until it is 1 km.

Result The result is easy to predict: Coverage will stay roughly the same until balloons are forced to clump together in order to communicate, at which point their communication areas will start to overlap, and thus reduce coverage. And in fact, as in seen in Figure 11, this is exactly what happens. In the graph, coverage is reduced evenly from 400 km to 1 km. 100 km communication range is at about 1200k simulation seconds, at which point the coverage starts to drop because of overlapping coverage areas.

Figure 11: Effect of gradually decreasing coverage range



There is a video that shows this experiment [23].

How does ascent and descent speed influence coverage? This is the most interesting experiment to carry out, not only because it directly affects balloon steering, but also because there are two parameters which have to be manipulated. To carry out the experiment, both ascent and descent rate is decreased until it is 0. Then only ascent speed will be increased, then only descent, and finally both.

Result The result is as expected: A too low ascent or descent speed will result in balloons losing control, because they are unable to change layer quickly enough. Setting either speed to 0 will result in balloons only rising or falling in altitude. Setting one to low and one to high is almost the same as setting both to low, it is clear that the slower speed is the limiting factor, because balloons will have to rise and fall constantly to keep up a good spread. There is a video showing this experiment [24].

Does the steering work in a wind field where winds may go in every direction, or in changing wind fields? One of the benefits the relatively simple control algorithm has, is the fact that both these scenarios should work. A simple simulation is conducted, where wind fields are changed, set to a relative wind field, and changed back to an absolute wind field. After that, the wind field will be generated (changed) a few times every second to simulate changing wind manually.

Result The result is as expected: Changing the wind has no significant influence on area coverage. There is a video showing this experiment [25].

7 Optimization and problem solving

7.1 Introduction

Most of the problems in this simulation stem from the simple model definitions and omitted variables. Some of them may be close to a realistic depiction of real balloons in real wind, while others would require very complex or difficult solutions and thus cannot be solved within the scope of this thesis.

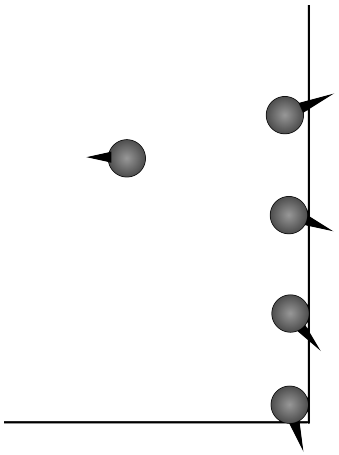
7.2 The knowledge problem

There are a few problems with the assumptions in the simulation: the biggest one is the need for accurate wind information: A balloon can only choose which wind layer to go to if it knows the winds direction in every layer above and below. In a real-world scenario this depends greatly on how much the wind directions vary within each wind layer, if they do not change much over time, other balloons in close proximity and on different altitudes could report their velocity and the balloon could interpolate between those vectors.
Another method would be simple trial and error: If a balloon can rise and fall fast enough, it should be able to test the wind in different layers and then choose the right layer accordingly, while also relaying the found information to other balloons. Both these methods depend greatly on real world conditions, and cannot be evaluated without real wind data and in-depth knowledge of conditions in the stratosphere and the balloons' exact parameters. Therefore, balloons which have complete knowledge of the wind field are the best compromise for this simulation, because real balloons will not have to rely solely on information from other balloons either. Thousands of weather balloons and real time models would also provide data, making it difficult to determine how much a real balloon will actually know about the wind field. This is an example of a problem that is not optimized to completely simulate a real situation.

7.3 The edge problem

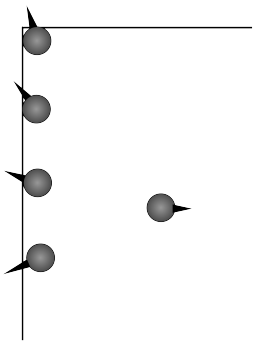
One of the biggest solved problems in this simulation is what shall be called the *edge problem*. One big difference to a completely realistic scenario is the fact that the simulation has boundaries, beyond which balloons cannot move. The choice to reset balloons to the opposite side of the simulation area helps with the problem, but does not solve it.

Figure 12: Balloons on the simulation area's edge (Top view)



The problem is illustrated in Figure 12, where four balloons are almost outside the simulation border, with one balloon being further inside. All four balloons at the edge want to get away from each other as well as from the balloon to the left, leading them to steer into directions as indicated in the picture. Because they are aligned on one axis, and other than that only detect the one balloon to the left, they will steer to the right, towards the simulation boundary. Once there, they will be reset to the opposite side, leading to a situation as in Figure 13. If they would not be

Figure 13: Balloons on the simulation area's opposite edge (Top view)



reset to the simulations other side, they would simply stick to the simulation border forever. If there is some balloon inside the simulation area similar to the one before, the same problem will occur and the balloons will steer back to the border. Because the simulation area is filled with balloons, these two situations almost always occur. This leads to:

1. Most balloons staying very close to the simulation area's boundary

2. Balloons quickly moving through the boundaries and back again continuously

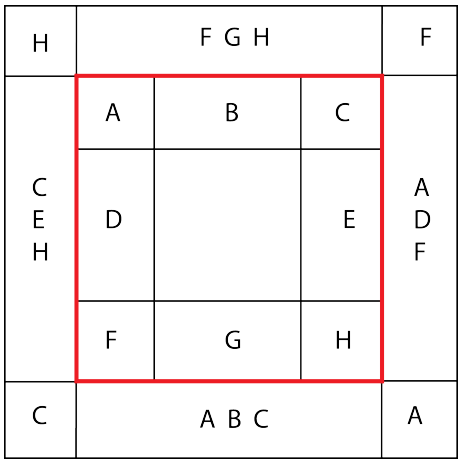
This situation does not always happen because there might not be a wind vector that steers balloons back towards the border, but it does, and enough balloons are present, the problem becomes noticeable. Evidently, a larger communication range leads to more balloons inside the area and less outside being detected, worsening the problem. A too large detection range may even lead to worse coverage because of this, because balloons would consider a lot of the empty space outside. This is not acceptable in such a simulation, and a good solution to this problem is crucial.

The core problem is the fact that the balloons move on what actually is a toroidal shape, which can be described as a rectangle, in which opposite sides are attached to each other, which is exactly how the simulation behaves. See [26] for a visualization of this.

The balloons move on this torus, but to the control algorithm it is still a square, so to solve the edge problem, the control algorithm for a balloon will have to consider balloons close to an opposite edge of the simulation, too.

Ghost balloons Implementing this is fairly simple by just creating so called *ghost balloons*, which are representations of balloons on the simulation's other side. These are just copies of "real" balloons, and do not have any parameters other than a position. The simulation will not actively steer these balloons, as they are just copies. See Figure 14 to see how these ghost balloons are created. Once they are in

Figure 14: Balloons are copied outside the simulation to fill the space outside the boundaries, the red area represents the simulation area, and two areas with the same letter will contain the same balloons, just translated on the horizontal plane, so that their relative positions within each rectangle will remain the same. The red area shows the simulation bounds, balloons outside will be ghosts



the simulation, they influence the real balloons' steering, because the real balloons no longer detect empty space outside the simulation boundaries. This is consistent with the simulation's general concept, where the simulated area is just a small window into a big world with many more balloons. In this scenario, the adjacent squares just happen to have the same pattern of balloons in them. It is easy to verify the consistency of the concept by enabling the *Show Ghost Balloons* option in the view settings. If one now focuses on a ghost balloon it will be obvious that the positions correlate to those on the other side. This is easy to confirm visually: as soon as a

ghost touches the simulation border, it will be replaced by a real balloon, which in turn will be replaced by a ghost on the other side. See [27] for a video which shows this effect.

The ghost balloons only exist to influence the balloon's steering, therefore only ghost balloons within communication radius to the border are necessary. This concept still leads to a large increase in balloon count, because every balloon may spawn up to two ghost balloons. Imagine for example a balloon in section A of Figure 14. It will be copied to the two sections at the bottom containing the letter A. This is necessary, because all 8 rectangles surrounding the simulation square have to be filled with ghost balloons in order to avoid any empty spaces.

7.4 Performance

Even though ghost balloons have no property except a position, they are included in every real balloon's closest neighbour search and will further slow down the simulation, to the point where a primitive algorithm which just measures the distance from every balloon to every other balloon in $O(n^2)$ time is too slow.

A better performing algorithm or data structure is clearly needed, and will also speed up every other calculation involving balloons, e.g. coverage area sampling.

Algorithm and data structure Binary space partitioning is a concept which allows dividing a multidimensional space into smaller subsections. These sections are then divided again and again, until every partition meets certain criteria. This will allow very fast access to searches such as *nearest neighbour search*, which is exactly what the simulation needs. In this case, the space is divided until every section only contains one element, making a *k-dimensional tree* (kd-tree for short) the best data structure for the purpose. See [28] for a visual explanation on how these trees are built and searched.

Many kd-tree implementations exist, and many are more optimized than this thesis' scope would allow in reasonable development time. To not reinvent the wheel, an existing solution is used from [29]. This solution additionally uses a custom optimized data structure to store the tree, making it one of the fastest implementations of the algorithm.

To fill this tree, a simpler version of the balloon object is used, which only has two properties: A position and an identification number. All balloons and ghost balloons are in the same tree, but ghost balloons will all have the same id as the balloon they originate from, which opens up for very useful possibilities, such as connections over the simulations border. This again is more realistic than it may seem, because the simulation area is thought surrounded by other areas containing balloons, which the balloons absolutely should connect to if they are close enough.

7.5 Further potential improvements

Even though the simulation serves its purpose very well, there is always room for improvement. E.g. a more intricate solution would allow setting and parameters changes set in files, which the simulation could run without any graphical output in order to obtain more accurate results in less time, and make experiments repeatable. The renderer is not optimized for speed either, the amount of OpenGL render calls

could be reduced drastically by gathering similar calls into one. These two and every other potential improvement add little benefit for sometimes large costs, such as increased source code complexity, which leads to more difficult code maintenance. The simulation is imperfect, but good enough to serve its purpose very well. Therefore, these and other potential improvements are deemed not necessary for a correct and working solution.

8 Conclusion

The simulation is a simplified model of the real problem, but is accurate enough to draw conclusions about the real world problem. Experiments confirm that the implemented control algorithm indeed works as intended. It is therefore deemed a success.
It is clear that autonomous balloons are able to provide stable coverage to the ground, as long as their parameters are not set to extreme values. But even those extreme values allow for useful observations, which might translate over to real scenarios, and certainly would be helpful to anyone working on balloon control for Project Loon. The findings in this simulation are consistent with Project Loon's own simulation of the problem [16], confirming the fact that the simulation's results would be similar if real wind data were used instead of a randomly generated wind field.

9 References

If not specified otherwise, websites are referenced as of 30.6.2014 12:00.

References

[1] Ask Away: How Does Project Loon Use Wind Data?
https://www.youtube.com/watch?v=gaY_1_2UBas&index=5&list=PLi7C1_I60LN7Z_CEMhl2w0pKVlwzlwMrs

[2] Visualization of Stratosphere winds http://earth.nullschool.net/#current/wind/isobaric/70hPa/orthographic=-18.30,-9.32,439

[3] National Oceanic and Atmospheric Administration http://www.noaa.gov/

[4] Google Loon Team mentions range
http://www.google.com/loon/faq/#tab=technology

[5] Article about Loon mentioning ground stations http://bigstory.ap.org/article/google-begins-launching-internet-beaming-balloons

[6] A Hydrogen-Filled Weather Balloon Flight into Near Space
http://bovineaerospace.wordpress.com/2013/06/02/a-hydrogen-filled-weather-balloon-flight-into-near-space/

[7] Balloon Rise Rate and Bursting Altitude
http://championship.endeavours.org/2014/Resources/Tech/Balloon%20Rise%20Rate%20and%20Bursting%20Altitude.pdf

[8] Radiosondes
http://www.radiopassioni.it/pdf/materialirsonde/Radiosondes.pdf

[9] Project Loon - 500,000 kilometers in flight
https://plus.google.com/103068231639729844333/posts/bWn1HJmyqLk

[10] Model-View-Controller: A Design Pattern for Software
https://ist.berkeley.edu/as-ag/pub/pdf/mvc-seminar.pdf

[11] Donald H. House: Orthographic and Perspective Projection http://people.cs.clemson.edu/~dhouse/courses/405/notes/projections.pdf

[12] The mono framework http://www.mono-project.com/Main_Page

[13] OpenTK Framework http://www.opentk.com/project/opentk

[14] The Open Toolkit Manual, Chapter 2: Introduction to OpenTk
http://www.opentk.com/book/export/html/105

[15] OpenGL Programming Guide - Chapter 3
http://www.glprogramming.com/red/chapter03.html#name2

[16] Ask Away: How can balloons provide stable coverage?
http://www.youtube.com/watch?v=mjyLynnQuC4

[17] Herbert G. Tanner: Stable Flocking of Mobile Agents, Part I: Fixed Topology
http://www.seas.upenn.edu/~jadbabai/papers/boids_smooth.pdf

[18] Craig W. Reynolds: Flocks, Herds, and Schools: A Distributed Behavioral Model http://www.cs.toronto.edu/~dt/siggraph97-course/cwr87/

[19] Craig W. Reynolds: Steering Behaviors For Autonomous Characters
http://www.red3d.com/cwr/steer/gdc99/

[20] Joseph T. Schaefer and Charles A. Doswell: On the Interpolation of a Vector Field http://www.spc.noaa.gov/publications/schaefer/interpol.pdf

[21] Federico Librino, Marco Levorato and Michele Zorzi: An Algorithmic Solution for Computing Circle Intersection Areas and its Applications to Wireless Communications http://arxiv.org/pdf/1204.3569.pdf

[22] Balloon Control influencing coverage
https://www.youtube.com/watch?v=DCRKjBQ_WBY

[23] Influence of varying balloon communication distance
https://www.youtube.com/watch?v=X0zfc_3G3O4

[24] Ascent and descent speed changes
https://www.youtube.com/watch?v=4b-QcN4cFas

[25] Balloons navigating changing wind fields
https://www.youtube.com/watch?v=sW5ZjAkWqZw

[26] Wikipedia: Torus http://en.wikipedia.org/wiki/Torus#Flat_torus

[27] Ghost Balloons in the Simulation
https://www.youtube.com/watch?v=S64GKUNp5zk

[28] Slides on K-D-Tree https://courses.cs.washington.edu/courses/cse373/02au/lectures/lecture22l.pdf

[29] KD-Tree Implementation used in the simulation.
https://code.google.com/p/kd-sharp/

10 Appendix

10.1 Source code

Source files generated by the IDE are not included. The complete source code (including project files and compiled binaries) can be downloaded at `http://bit.ly/1nXqVFp`.

10.1.1 Wind.cs

```
using OpenTK;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LoonControl.Model
{
    class Wind
    {
        public static Vector2[, ,] generateRandomWindField(int x, int z,
            Vector2 average, int layers=8)
        {
            Vector2[, ,] result = new Vector2[x+1, layers, z+1];

            //Generate completely random data
            Vector2 temp = Vector2.Zero;//This vector stores the initial
                wind point for every layer
            Random r = new Random();

            //Wind speeds should go from 1 to 10 m/s
            //So the max vector should be about (8,8), and average wind is
                about (4,4)

            for (int _y = 0; _y < layers; _y++){

                for (int _z = 0; _z <= z; _z++){

                    for (int _x = 0; _x <= x; _x++)

                    {

                        if (_x == 0 && _z == 0)
                        {
                            temp = new Vector2((float)(r.NextDouble()-0.5f)
                                * 4f, (float)(r.NextDouble()-0.5f) * 4f);//
                                -2 .. +2
                            result[_x, _y, _z] = temp+average;

                        }
                        else
                        {
                            result[_x, _y, _z] = average + temp + (new
                                Vector2((float)(r.NextDouble()-0.5f) * 0.2f,
                                (float)(r.NextDouble()-0.5f) * 2f));//
                                Deviate a little from temp
                            //Change temp a little bit
                            temp = temp + (new Vector2((float)(r.NextDouble
                                () - 0.5f) * 0.1f, (float)(r.NextDouble() -
                                0.5f) * 0.5f));
```

```
                        }
                    }
                }

            }

                return result;
        }
    }
}
```

10.1.2 Balloon.cs

```
using KDTree;
using LoonControl.Control;
using LoonControl.Model;
using LoonControl.Util;
using OpenTK;
using OpenTK.Graphics.OpenGL;
using System;
using System.Collections.Generic;

namespace LoonControl
{
    internal class Balloon
    {
        public static bool bound = false;
        public static VertexBufferManager vbo;
        private static Random r = new Random();

        private Vector3 position;//private field for Position property

        public Vector3 Position { get { return position; } set { position =
            value; } }

        public Vector3 velocity;//private field for Velocity property

        public Vector3 Velocity { get { return velocity; } set { velocity =
            value; } }

        //Wayfinding variables and settings

        public Vector2 target = Vector2.Zero;

        public List<SimpleBallon> connectedBalloons = new List<SimpleBallon
            >();

        public bool Connected { get; set; }

        private float communicationDistance = 20; //Private field for
            CommunicationDistance property

        private float ascendVelocity = 1f;

        private float descendVelocity = 1f;

        public float CommunicationDistance { get { return
            communicationDistance; } set { communicationDistance = value; } }
```

```
private float coverageDistance = 20;//Private field for
    CoverageDistance property

public float CoverageDistance { get { return coverageDistance; } set
     { coverageDistance = value; } }

private float detectionDistance = 100;//Private field for
    DetectionDistance property

public float DetectionDistance { get { return detectionDistance; }
    set { detectionDistance = value; } }

public Balloon(Vector3 pos)
{
     Position = pos;
     if (!bound)
     {
          vbo = new VertexBufferManager();
          SphereMesh s = new SphereMesh(24);
          vbo.setMesh(s.getVertices(), s.getElements());
          bound = true;
     }
}

public void Render()
{
     GL.MatrixMode(MatrixMode.Modelview);
     GL.PushMatrix();
     GL.Translate(Position);
     vbo.Render();
     GL.PopMatrix();
}

public void Update(double delta, Vector2[, ,] wind, WrapStyle wrap,
    bool controlDisabled, ref KDTree<SimpleBallon> tree, Vector2
    averageWind)
{
     //Re-read settings
     CommunicationDistance = Properties.Settings.Default.
         CommunicationRadius / 10f;//Setting is in km, simulation in
         km/10
     CoverageDistance = Properties.Settings.Default.CoverageRadius /
         10f;
     DetectionDistance = Properties.Settings.Default.DetectionRadius
         / 10f;//We need the detection distance to be squared

     //Vertical axis is in km, no worries here
     ascendVelocity = Properties.Settings.Default.AscendSpeed;
     descendVelocity = Properties.Settings.Default.DescentSpeed;

    //Target direction in 2D coordinates
     target = Vector2.Zero;
     //Clamp position

     clamp(wrap);

     //Calculate the interpolated vector at the current position
     Vector2 mean = VectorMath.CalculateLerpVector(position, wind);
```

```
velocity.X = mean.X;
velocity.Z = mean.Y;

//Reset flag and connection list
connectedBalloons.Clear();
Connected = false;

//Find all balloons in detection distance or Communication
   Distance
double findDistance = Math.Max(communicationDistance,
   DetectionDistance);
var pIter = tree.NearestNeighbors(new double[] { position.X,
   position.Z }, 36, Math.Pow(findDistance,2));
double closest = 1000;
while (pIter.MoveNext())
{
    double distance = Math.Sqrt(pIter.CurrentDistance);
    // Get the ellipse.
    if (distance > 0)
    {
        if (distance <= DetectionDistance)
        {
            //Our y axis is flipped when compared to standard
               coordinates, therefore y=y*-1
            target += (new Vector2(position.X - pIter.Current.
               pos.X, (position.Z - pIter.Current.pos.Y))) / ((
               float)Math.Pow(distance, 2));
        }

        if (distance <= CommunicationDistance)
        {
            Connected = true;
            if (distance < closest)
            {
                closest = distance;
            }
            connectedBalloons.Add(pIter.Current);
        }

    }
}

if ( closest > CommunicationDistance * 0.9 || connectedBalloons.
   Count == 0)//Last 10 percent of communication
{
    //We want to away back to detected balloons
    target *= -1; //Inverse vector

}
target.Normalize();
//Find the most fitting wind vector
//First, calculate the interpolated vector for every layer
double closestProduct = -2;
int closestLayer = -1;
Vector2 temp;
for (int i = 0; i < wind.GetLength(1); i++)
{
```

```
            temp = VectorMath.CalculateLerpVector(new Vector3(position.X
                , 17 + i, position.Z), wind);
            temp = temp - averageWind;
            temp.Normalize();
            if ((temp + target).LengthFast > closestProduct)
            {
                closestLayer = 17 + i;
                closestProduct = (temp + target).LengthFast;
            }
        }
        // Use the line below for immediate translation
        // this.position.Y = closestLayer;
        if (closestLayer + 0.5f < (int)position.Y)
        {
            velocity.Y = -descendVelocity;
        }
        else if ((closestLayer + 0.5).Equals(position.Y))
        {
            velocity.Y = 0;
        }
        else
        {
            velocity.Y = ascendVelocity;
        }
        //If control is disabled, override vertical velocity to 0
        if (controlDisabled)
        {
            velocity.Y = 0;
        }

        //Position += new Vector3(target.X, 0, target.Y) * 0.05f ;
        //Finally apply the transformation, calculate (0.0001f * delta)
            first to improve rounding
        Position += velocity * (0.0001f * (float)delta); //Velocity is m
            /s, Position Grid is in unit 10 km.
    }

    private void clamp(WrapStyle wrap)
    {
        if (position.Y < 17)
        {
            position.Y = 17;
        }
        else if (position.Y > 24)
        {
            position.Y = 24;
        }

        if (position.X > 100)
        {
            if (wrap == WrapStyle.Wrap)
                position.X = position.X % 100;
            else
                position.X = 100;
        }
        else if (position.X < 0)
        {
            if (wrap == WrapStyle.Wrap)
```

```
            position.X = 100;
        else
            position.X = 0;
    }
    if (position.Z > 100)
    {
        if (wrap == WrapStyle.Wrap)
            position.Z = position.Z % 100;
        else
            position.Z = 100;
    }
    else if (position.Z < 0)
    {
        if (wrap == WrapStyle.Wrap)
            position.Z = 100;
        else
            position.Z = 0;
    }
}

public static double CalculateMeanDistance(Balloon[] balloons)
{
    double average = 0;
    for (int i = 0; i < balloons.Length; i++)
    {
        double localAverage = 0;
        for (int j = 0; j < balloons.Length; j++)
        {
            if (i != j) //Dont compare to self
                localAverage += (balloons[i].Position - balloons[j].
                    Position).Length;
            if (j >= balloons.Length - 1)//Last element
                localAverage /= j;
        }
        average += localAverage;//Add calculated average for local
            balloon to total average

        if (i >= balloons.Length - 1)//Last element
            average /= i;
    }
    return average;
}
public static List<Balloon> GenerateRandomBalloons(int p)
{
    Random r = new Random();
    List<Balloon> result = new List<Balloon>();
    for (int i = 0; i < p; i++)
    {
        result.Add(new Balloon(new Vector3(1 + r.Next(99), 18 + r.
            Next(6), 1 + r.Next(98))));
    }
    return result;
}

private Vector2 GetClosestEdgePoint(Vector2 point){
    Vector2 result = new Vector2();
    int side = 0; //0 = left, 1 = up, 2 = down, 3 = right
```

```
Vector2 northeast = new Vector2(0, 0);
Vector2 southeast = new Vector2(0, 100);
Vector2 northwest = new Vector2(100, 0);
Vector2 southwest = new Vector2(100, 100);

double tempDistance = 0;
double newDistance = 0; //Used to avoid calculating distance
    twice

tempDistance = (VectorMath.PointToLineDistance(northeast,
    southeast, point));//LEFT

// UP
newDistance = (VectorMath.PointToLineDistance(northeast,
    northwest, point));
if (newDistance < tempDistance)
{
     side = 1;
     tempDistance = newDistance;
}
//DOWN
newDistance = (VectorMath.PointToLineDistance(southeast,
    southwest, point));
if (newDistance < tempDistance)
{
     side = 2;
     tempDistance = newDistance;
}
//RIGHT
newDistance = (VectorMath.PointToLineDistance(northwest,
    southwest, point));
if (newDistance < tempDistance)
{
     side = 3;
     tempDistance = newDistance;
}

//Now get the right point on the border and return it
switch (side)
{    case 0://Left
     {
          result = new Vector2(-0.1f, point.Y);
          break;
     }
case 1://Up
     {
          result = new Vector2(point.X, -0.1f);
          break;
     }
case 2://Down
     {
          result = new Vector2(point.X, 100.1f);
          break;
     }
case 3://Right
     {
          result = new Vector2(100.1f, point.Y);
          break;
```

```
                }
                default:
                Console.WriteLine("Could not find closest edge!");
                    break;
            }

            return result;
        }


    }
    struct SimpleBallon
    {
        public int id;
        public Vector2 pos;

        public SimpleBallon(int id, Vector2 pos)
        {
            this.id = id;
            this.pos = pos;
        }
    }
}
```

10.1.3 MainForm.cs

```
using LoonControl.Model;
using OpenTK.Graphics.OpenGL;
using OpenTK.Input;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;
using LoonControl.Control;

namespace LoonControl
{
    public partial class MainForm : Form
    {
        private bool loaded;
        private OpenGLRenderer renderer;
        private Simulation sim;

        //Simulation stuff
        private double simulationRate = 1;

        //Stopwatch for debugging and framerate calculation
        Stopwatch sw = new Stopwatch();
        double totalSimulationTime = 0;

        ParametersForm parameters;
```

```
private bool chartEnabled = true;

double accumulator = 0;
int idleCounter = 0;


public MainForm()
{
    InitializeComponent();

}

private void Form1_Load(object sender, EventArgs e)
{
    windComboBox.SelectedIndex = 0;
}

private void glControl1_Load(object sender, EventArgs e)
{
    loaded = true;
    renderer = new OpenGLRenderer(glControl1);
    sim = new Simulation(renderer);
    sim.Initialize();
    Application.Idle += Application_Idle;
    sw.Start();

}
private double CalculateDelta()
{
    sw.Stop();
    double delta = sw.Elapsed.TotalMilliseconds;
    sw.Reset();
    sw.Start();
    return delta;
}

void Application_Idle(object sender, EventArgs e)
{
    while (glControl1.IsIdle)
    {
        double milliseconds = CalculateDelta();
        CalcFPS(milliseconds);
        processKeys(milliseconds);
        sim.Update(milliseconds*simulationRate);
        renderer.Render();
    }

}

private void updateAverageDistance(double delta)
{
   totalSimulationTime += delta;
   if (chartEnabled)//Every second
   {
       new Thread(sample).Start();

   }
}
```

```
 private void sample()
 {
     float percentage =  sim.SampleAreaCoverage();
     try
     {
         chart1.BeginInvoke(new Action(delegate() { chart1.Series[0].
            Points.AddXY(Math.Round(totalSimulationTime,2),
            percentage); }));
     }
     catch (Exception e)
     {
         Thread.CurrentThread.Abort();//Apllication may have been
            closed
     }
 }
 private void CalcFPS(double milliseconds)
 {
     idleCounter++;
     accumulator += milliseconds;
     if (accumulator > 1000)
     {
         frameRateLabel.Text = "FPS: "+idleCounter.ToString();
         accumulator -= 1000;
         idleCounter = 0; // don't forget to reset the counter!
         updateAverageDistance(simulationRate);
     }
 }

 private void glControl1_Paint(object sender, PaintEventArgs e)
 {
     if (!loaded) // Play nice
         return;
     renderer.Render();
 }

 private void glControl1_Resize(object sender, EventArgs e)
 {
     if (loaded)
     renderer.Resize();
 }

 private void glControl1_KeyPress(object sender, KeyPressEventArgs e)
 {
 }

protected void processKeys(double delta){

    if (Keyboard.IsKeyDown(Keys.OemMinus))
     {
         renderer.Scroll(true,delta);
     }
     if (Keyboard.IsKeyDown(Keys.Oemplus))
     {
         renderer.Scroll(false, delta);
     }

    /*
```

```
        if (Keyboard.IsKeyDown(Keys.Left))
        {
            renderer.RotateCamera(0, 0.001f*delta);
        }
        if (Keyboard.IsKeyDown(Keys.Right))
        {
            renderer.RotateCamera(0f, -0.001f * delta);
        }
        if (Keyboard.IsKeyDown(Keys.Up))
        {
            renderer.RotateCamera(-0.001f * delta, 0);
        }
        if (Keyboard.IsKeyDown(Keys.Down))
        {
            renderer.RotateCamera(0.001f * delta, 0);
        }*/

    }

    private void restartToolStripMenuItem_Click(object sender, EventArgs
        e)
    {
        chart1.Series[0].Points.Clear();
        totalSimulationTime = 0;
        sim.Restart();
    }

    private void trackBar3_ValueChanged(object sender, EventArgs e)
    {
        if (trackBar3.Value == -11)//Pause
        {
            label4.Text = "Paused";
            simulationRate = 0;
        }
        else
        {
            float factor = trackBar3.Value / 10f;
            label4.Text = ((int)(Math.Pow(10,factor) * 1000)).ToString("D
                ") + "x";
            simulationRate = Math.Pow(10, factor);
        }

    }

    private void windCheckBox_CheckedChanged(object sender, EventArgs e)
    {
        renderer.DrawWindField = windCheckBox.Checked;
        if (!windCheckBox.Checked)
        {
            layerCheckBox.Checked = false;
        }
    }

    private void layerCheckBox_CheckedChanged(object sender, EventArgs e)
    {
        layerUpDown.Enabled = layerCheckBox.Checked;
        renderer.OnlyOneLayer = layerCheckBox.Checked;
        renderer.SpecificLayer = (int)layerUpDown.Value;
```

```
    if (layerCheckBox.Checked)
    {
        windCheckBox.Checked = true;
    }
}

private void layerUpDown_ValueChanged(object sender, EventArgs e)
{
    renderer.SpecificLayer = (int)layerUpDown.Value;
}

private void drawBorderCheckBox_CheckedChanged(object sender,
   EventArgs e)
{
    renderer.DrawBorder = drawBorderCheckBox.Checked;
}

private void velocityCheckBox_CheckedChanged(object sender, EventArgs
    e)
{
    renderer.DrawBalloonVelocity = velocityCheckBox.Checked;
}

private void addBalloonButton_Click(object sender, EventArgs e)
{
    sim.AddBalloon();
}

private void deleteBalloonButton_Click(object sender, EventArgs e)
{
    sim.DeleteBalloon();
}

private void colorCheckBox_CheckedChanged(object sender, EventArgs e)
{
    renderer.ColorBalloons =  colorCheckBox.Checked;
}

private void communicaionCheckBox_CheckedChanged(object sender,
   EventArgs e)
{
    renderer.ShowConnections = connectionCheckBox.Checked;
}

private void ServiceCheckbox_CheckedChanged(object sender, EventArgs
   e)
{
    renderer.ShowServiceRadius = serviceCheckbox.Checked;
}

private void chart1_Click(object sender, EventArgs e)
{

}

private void parametersButton_Click(object sender, EventArgs e)
{
    if (parameters == null || parameters.IsDisposed)
```

```
        {
            parameters = new ParametersForm();
        }
        parameters.Show();
        parameters.BringToFront();
    }

    private void samplingCheckBox_CheckedChanged(object sender, EventArgs
        e)
    {
        chartEnabled = !samplingCheckBox.Checked;
    }

    private void backgroundWorker_DoWork(object sender, DoWorkEventArgs e
        )
    {

    }

    private void clearButton_Click(object sender, EventArgs e)
    {
        chart1.Series[0].Points.Clear();
        totalSimulationTime = 0;
    }
    private void controlCheckBox_CheckedChanged(object sender, EventArgs
        e)
    {
        sim.ControlDisabled = controlCheckBox.Checked;
    }

    private void saveImageToolStripMenuItem_Click(object sender,
        EventArgs e)
    {
        //Not implemented yet
    }

    private void ghostCheckBox_CheckedChanged(object sender, EventArgs e)
    {
        renderer.DrawGhosts = ghostCheckBox.Checked;
    }

    private void newWindButton_Click(object sender, EventArgs e)
    {
        sim.newWindField(windComboBox.SelectedIndex == 0);
    }

    private void windComboBox_SelectedIndexChanged(object sender,
        EventArgs e)
    {

    }

    private void windButton_Click(object sender, EventArgs e)
    {
        sim.newWindField(windComboBox.SelectedIndex == 0);
    }
```

```
        private void controlCheckBox_CheckedChanged_1(object sender,
            EventArgs e)
        {
             sim.ControlDisabled = controlCheckBox.Checked;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            SaveFileDialog f =  new SaveFileDialog();
            f.Filter = "Image |*.png";
            DialogResult r = f.ShowDialog();
            if (r == DialogResult.OK)
            {
                chart1.SaveImage(f.FileName, System.Windows.Forms.
                    DataVisualization.Charting.ChartImageFormat.Png);
            }
        }

    }
}
```

10.1.4 ParametersForm.cs

```
using System;
using System.Windows.Forms;

namespace LoonControl
{
    public partial class ParametersForm : Form
    {
        public ParametersForm()
        {
            InitializeComponent();
        }

        private void coverageTrackBar_Scroll(object sender, EventArgs e)
        {
            coverageLabel.Text = coverageTrackBar.Value.ToString();
            Properties.Settings.Default.CoverageRadius = coverageTrackBar.
                Value;
            Properties.Settings.Default.Save();

        }

        private void descendTrackBar_Scroll(object sender, EventArgs e)
        {
            descendLabel.Text = (descendTrackBar.Value / 10f).ToString();
            Properties.Settings.Default.DescentSpeed = descendTrackBar.Value
                /10f;
            Properties.Settings.Default.Save();
        }

        private void ascendTrackBar_Scroll(object sender, EventArgs e)
        {
            ascendLabel.Text = (ascendTrackBar.Value/10f).ToString();
            Properties.Settings.Default.AscendSpeed = ascendTrackBar.Value
                /10f;
            Properties.Settings.Default.Save();
        }
```

```
        private void startCountTrackBar_Scroll(object sender, EventArgs e)
        {
            startCountLabel.Text = startCountTrackBar.Value.ToString();
            Properties.Settings.Default.StartCount = startCountTrackBar.
                Value;
            Properties.Settings.Default.Save();
        }

        private void detectionTrackBar_Scroll(object sender, EventArgs e)
        {
            detectionLabel.Text = detectionTrackBar.Value.ToString();
            Properties.Settings.Default.DetectionRadius = detectionTrackBar.
                Value;
            Properties.Settings.Default.Save();
        }

        private void communicationTrackBar_Scroll(object sender, EventArgs e
            )
        {
            communicationLabel.Text = communicationTrackBar.Value.ToString()
                ;
            Properties.Settings.Default.CommunicationRadius =
                communicationTrackBar.Value;
            Properties.Settings.Default.Save();
        }

        private void ParametersForm_Load(object sender, EventArgs e)
        {
            //Initialize the bars to the values from settings
            coverageTrackBar.Value = Properties.Settings.Default.
                CoverageRadius;
            coverageLabel.Text = coverageTrackBar.Value.ToString();

            communicationTrackBar.Value = Properties.Settings.Default.
                CommunicationRadius;
            communicationLabel.Text = communicationTrackBar.Value.ToString()
                ;

            detectionTrackBar.Value = Properties.Settings.Default.
                DetectionRadius;
            detectionLabel.Text = detectionTrackBar.Value.ToString();

            ascendTrackBar.Value = (int)( Properties.Settings.Default.
                AscendSpeed*10);
            ascendLabel.Text = (ascendTrackBar.Value / 10f).ToString();

            descendTrackBar.Value = (int)(Properties.Settings.Default.
                DescentSpeed*10);
            descendLabel.Text = (descendTrackBar.Value / 10f).ToString();

            startCountTrackBar.Value = Properties.Settings.Default.
                StartCount;
            startCountLabel.Text = startCountTrackBar.Value.ToString();
        }
    }
}
```

10.1.5 Simulation.cs

```
using KDTree;
using LoonControl.Model;
using OpenTK;
using System;
using System.Collections.Generic;

namespace LoonControl.Control
{
    internal class Simulation
    {
        //The renderer object
        private OpenGLRenderer renderer;

        //Enable toggling on and off balloon control
        public bool ControlDisabled { get; set; }

        //All balloon objects in the scene
        public List<Balloon> Balloons { get; set; }

        //The wind vector Property
        public Vector2[, ,] WindField { get; set; }

        //Border behaviour
        public WrapStyle Wrap { get; set; }

        public double CoverageRadius { get; set; }

        private KDTree<SimpleBallon> tree;

        public List<SimpleBallon> ghostBalloons = new List<SimpleBallon>();

        //Used to lock the KDTree
        private readonly object treelock = new object();

        public Vector2 WindAverage = new Vector2(1,1);

        public Simulation(OpenGLRenderer renderer)
        {
            this.renderer = renderer;
        }

        public void Initialize()
        {
            tree = new KDTree<SimpleBallon>(2);
            Balloons = new List<Balloon>();
            Restart();
            renderer.setSimulation(this);
            renderer.Initialize();
        }

        public void Restart()
        {
            Balloons.Clear();
            Balloons = new List<Balloon>();
            Balloons = Balloon.GenerateRandomBalloons(Properties.Settings.
                Default.StartCount);
```

```
        newWindField(true);//Absolute by default
    }

    internal void Update(double delta)
    {
        if (Properties.Settings.Default.StartCount < Balloons.Count)//
            Balloons should to be removed
        {
            for (int i = 0; i < (Balloons.Count - Properties.Settings.
                Default.StartCount); i++)
            {
                Balloons.RemoveAt(0);
            }
        }
        else if (Properties.Settings.Default.StartCount > Balloons.Count
            )
        {
            Balloons.AddRange(Balloon.GenerateRandomBalloons(Properties.
                Settings.Default.StartCount - Balloons.Count));
        }
        updateKDTree();
        for (int i = 0; i < Balloons.Count; i++)
        {
            Balloons[i].Update(delta, WindField, WrapStyle.Wrap,
                ControlDisabled, ref tree, WindAverage);
        }
    }

    private void updateKDTree()
    {
        ghostBalloons.Clear();
        float range = Math.Min(Properties.Settings.Default.
            DetectionRadius/10f,50);

        lock (treelock)
        {
            tree = new KDTree<SimpleBallon>(2);
            for (int i = 0; i < Balloons.Count; i++)
            {
                tree.AddPoint(new double[] { Balloons[i].Position.X,
                    Balloons[i].Position.Z }, new SimpleBallon(i, new
                    Vector2(Balloons[i].Position.X, Balloons[i].Position.
                    Z)));
                addOverlap(Balloons[i], range, i );
            }

        }
    }

    private void addOverlap(Balloon balloon, float range, int id)
    {
        //Left -> Right
        if (balloon.Position.X < range)
        {
            tree.AddPoint(new double[] { balloon.Position.X + 100,
                balloon.Position.Z }, new SimpleBallon(id, new Vector2(
                balloon.Position.X + 100, balloon.Position.Z)));
```

```
        ghostBalloons.Add(new SimpleBallon(id, new Vector2(balloon.
            Position.X + 100, balloon.Position.Z)));
    }
    //Right -> Left
    else if (balloon.Position.X > 100 - range)
    {
        tree.AddPoint(new double[] { balloon.Position.X - 100,
            balloon.Position.Z }, new SimpleBallon(id, new Vector2(
            balloon.Position.X - 100, balloon.Position.Z)));
        ghostBalloons.Add(new SimpleBallon(id, new Vector2(balloon.
            Position.X - 100, balloon.Position.Z)));
    }
    //Top -> Bottom
    if (balloon.Position.Z < range)
    {
        tree.AddPoint(new double[] { balloon.Position.X, balloon.
            Position.Z+100 }, new SimpleBallon(id, new Vector2(
            balloon.Position.X, balloon.Position.Z+100)));
        ghostBalloons.Add(new SimpleBallon(id, new Vector2(balloon.
            Position.X , balloon.Position.Z+100)));
    }
    //Bottom -> Top
    else if (balloon.Position.Z > 100 - range)
    {
        tree.AddPoint(new double[] { balloon.Position.X, balloon.
            Position.Z - 100 }, new SimpleBallon(id, new Vector2(
            balloon.Position.X, balloon.Position.Z - 100)));
        ghostBalloons.Add(new SimpleBallon(id, new Vector2(balloon.
            Position.X , balloon.Position.Z-100)));
    }

    //Left & Top
    if (balloon.Position.X < range && balloon.Position.Z < range )
    {
        tree.AddPoint(new double[] { balloon.Position.X + 100,
            balloon.Position.Z + 100}, new SimpleBallon(id, new
            Vector2(balloon.Position.X + 100, balloon.Position.Z+100)
            ));
        ghostBalloons.Add(new SimpleBallon(id, new Vector2(balloon.
            Position.X + 100, balloon.Position.Z+100)));
    }
    //Right & Bottom
    else if (balloon.Position.X > 100 - range && balloon.Position.Z
        > 100 - range)
    {
        tree.AddPoint(new double[] { balloon.Position.X - 100,
            balloon.Position.Z - 100 }, new SimpleBallon(id, new
            Vector2(balloon.Position.X - 100, balloon.Position.Z -
            100)));
        ghostBalloons.Add(new SimpleBallon(id, new Vector2(balloon.
            Position.X - 100, balloon.Position.Z - 100)));
    }
    //Right & Top
    else if (balloon.Position.Z < range && balloon.Position.X > 100
        - range)
    {
        tree.AddPoint(new double[] { balloon.Position.X-100, balloon
            .Position.Z + 100 }, new SimpleBallon(id, new Vector2(
```

```
            balloon.Position.X-100, balloon.Position.Z + 100)));
        ghostBalloons.Add(new SimpleBallon(id, new Vector2(balloon.
            Position.X-100, balloon.Position.Z + 100)));
    }
    //Left & Bottom
    else if (balloon.Position.Z > 100 - range && balloon.Position.X
        < range )
    {
        tree.AddPoint(new double[] { balloon.Position.X+100, balloon
            .Position.Z - 100 }, new SimpleBallon(id, new Vector2(
            balloon.Position.X+100, balloon.Position.Z - 100)));
        ghostBalloons.Add(new SimpleBallon(id, new Vector2(balloon.
            Position.X+100, balloon.Position.Z - 100)));
    }

}


internal void AddBalloon()
{
    Balloons.AddRange(Balloon.GenerateRandomBalloons(1));
}

internal void DeleteBalloon()
{
    if (Balloons.Count > 0)
    {
        Balloons.RemoveAt(0);
    }
}

public float SampleAreaCoverage(int samplecount = 100)
{
    //Re-read communication radius, might have been changed.
    CoverageRadius = Math.Pow(Properties.Settings.Default.
        CoverageRadius / 10f, 2);//Unit for setting is km, simulation
         is 1/10 km

    int inside = 0;//Number of samples inside any balloon's radius

    float step = 100f / samplecount;
    int counter = 0;

    lock (treelock)
    {
        for (double x = 0; x < 100; x += step)
        {
            for (double z = 0; z < 100; z += step)
            {
                var pIter = tree.NearestNeighbors(new double[] { x,
                    z }, 1, CoverageRadius);
                if (pIter.MoveNext())
                {
                    counter++;
                    try
                    {
                        if (Balloons[pIter.Current.id].Connected)
                            inside++;
```

```
                            }
                            catch (ArgumentOutOfRangeException e)
                            {
                                // Balloons[pIter.Current.id] might have been
                                    deleted, ignore the exception
                                //All other iterations are probably useless
                                    too, just return.
                                return ((float)inside / counter) * 100;
                            }
                        }
                    }
                }

            }


            return ((float)inside / (samplecount * samplecount)) * 100;
        }

        internal void newWindField(bool absolute)
        {
            if (absolute)
            {
                WindAverage = new Vector2(4, 4);
            }
            else
            {
                WindAverage = new Vector2(0, 0);
            }
            WindField = Wind.generateRandomWindField(10, 10, WindAverage);
        }
    }

    internal enum WrapStyle
    {
        None,
        Wrap
    }
}
```

10.1.6 OpenGLRenderer.cs

```
using LoonControl.Model;
using LoonControl.Util;
using LoonControl.Control;
using OpenTK;
using OpenTK.Graphics.OpenGL;
using OpenTK.Input;
using System;

namespace LoonControl
{
    internal class OpenGLRenderer
    {
        //The Simulation
        private Simulation sim;

        //Camera position and target
        // private Matrix4 cameraRotation;
```

```
private Vector3 cameraPosition;
private Vector3 cameraTarget;

private OpenTK.GLControl glControl;

private Matrix4 projection;

//used to calculate the amount of rotation of the camera
private float totalCamPitch = -90f;

private float totalCamYaw = 0f;

private Matrix4 view;

private Matrix4 world = Matrix4.CreateTranslation(0, 0, 0);

//Settings
public bool DrawWindField { get; set; }

public int SpecificLayer { get; set; }

public bool OnlyOneLayer { get; set; }

public bool DrawBalloonVelocity { get; set; }

public bool DrawBorder { get; set; }

public bool DrawBalloonRadius { get; set; }

public bool ColorBalloons { get; set; }

public bool ShowConnections { get; set; }

public bool DrawGhosts { get; set; }

public bool ShowServiceRadius { get; set; }

private bool perspective = false;
public bool Perspective { get { return perspective; } set {
   perspective = value; Resize(); } }

private float ZoomLevel = 0.15f;

public OpenGLRenderer(OpenTK.GLControl glControl)
{
    this.glControl = glControl;
}

public void Initialize()
{
    cameraPosition = new Vector3(50, 150, 50.1f);
    cameraTarget = new Vector3(50, 0, 50);

    Resize();

    GL.Enable(EnableCap.Light0);
    GL.Enable(EnableCap.Light1);
```

```
}

public void Render()
{
    GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.
       DepthBufferBit);//No need to clear color

    //Set model view according to camera
    GL.MatrixMode(MatrixMode.Modelview);
    view = Matrix4.LookAt(cameraPosition, cameraTarget, Vector3.
       UnitY);//Y = up

    GL.LoadMatrix(ref view);

    //Enable lighting
    GL.Enable(EnableCap.DepthTest);
    GL.Enable(EnableCap.Lighting);
    GL.Color3(1f, 1f, 1f);
    GL.Enable(EnableCap.ColorMaterial);
    //Paint balloon
    foreach (Balloon balloon in sim.Balloons)
    {
        if (ColorBalloons)
        {
            float layer = ((balloon.Position.Y) - 17) / 8f; //float
               layer = 0..1.0

            float red = 0;
            float green = 0;

            if (layer < 0.5)
            {       //first, green stays at 100%, red raises to 100%
                green = 1.0f;
                red = 2 * layer;
            }

            else if (0.5 <= layer)
            {       //then red stays at 100%, green decays
                red = 1.0f;
                green = (float)(1.0 - 2 * (layer - 0.5));
            }
            GL.Color3(red, green, 0f);
            //GL.Color3(1f,0f,1f);
        }
        else
        {
            GL.Color3(1f, 1f, 1f);
        }
        balloon.Render();
        if (DrawBalloonVelocity)
        {
            DrawBalloonDebugVelocity(balloon);

        }

        if (ShowServiceRadius)
        {
            GL.Color3(0.6f, 0.6f, 0.8f);
```

```
                GL.Begin(BeginMode.TriangleFan);
                drawCircle(balloon.CoverageDistance, new Vector2(balloon
                    .Position.X, balloon.Position.Z));
                GL.End();
            }

        }
        if (ShowConnections)
        {
            GL.Disable(EnableCap.Lighting);
             GL.Color3(1f, 1f, 0.9f);
             GL.Begin(BeginMode.Lines);
             foreach (Balloon b1 in sim.Balloons){//Seperate loops
                allows all draws in one call
                  foreach (SimpleBallon b2 in b1.connectedBalloons){
                      drawCommunicationLine(b1.Position.X, b1.Position.Z
                          , b2.pos.X, b2.pos.Y);
                  }
             }
            GL.End();
        }
        //Paint wind field over balloons
        if (DrawWindField)
        {
            PaintWindField(0);
        }
        //Paint area border
        if (DrawBorder)
        {
            DrawAreaBorder(new Vector3(100, 0, 100));
        }
        if (DrawGhosts)
        {
            DrawGhostBalloons();
        }
        glControl.SwapBuffers();
    }

    private void DrawGhostBalloons()
    {
        GL.Disable(EnableCap.Lighting);
        GL.Color3(0.8, 0.9, 0.9);
        GL.Begin(BeginMode.Lines);
        foreach (SimpleBallon b in sim.ghostBalloons)
        {
                GL.Vertex3(b.pos.X - 1, 0, b.pos.Y - 1);
                GL.Vertex3(b.pos.X + 1, 0, b.pos.Y + 1);

                GL.Vertex3(b.pos.X - 1, 0, b.pos.Y + 1);
                GL.Vertex3(b.pos.X + 1, 0, b.pos.Y - 1);
        }
        GL.End();
    }

    private static void PaintGradientBackground()
    {
        //Push projection matrix to prepare for background painting
        GL.MatrixMode(MatrixMode.Projection);
```

```
        GL.PushMatrix();
        GL.LoadIdentity();
        //Disable the depth test and lighting too
        GL.Disable(EnableCap.DepthTest);
        GL.Disable(EnableCap.Lighting);
        //Paint the background
        GL.MatrixMode(MatrixMode.Modelview);
        GL.LoadIdentity();
        GL.Begin(BeginMode.Quads);
        //red color
        GL.Color3(0.7, 0.7, 1);
        GL.Vertex2(-1.0, -1.0);
        GL.Vertex2(1.0, -1.0);
        //blue color
        GL.Color3(0.4, 0.4, 1.0);
        GL.Vertex2(1.0, 1.0);
        GL.Vertex2(-1.0, 1.0);
        GL.End();
        //Restore the projection matrix and depth settings
        GL.MatrixMode(MatrixMode.Projection);
        GL.PopMatrix();
        GL.Enable(EnableCap.DepthTest);
        GL.Enable(EnableCap.Lighting);
    }

    public void PaintWindField(int level)
    {
        GL.Disable(EnableCap.Lighting);

        GL.LineWidth(2.0f);

        GL.Begin(BeginMode.Lines);

        for (int x = 0; x < sim.WindField.GetLength(0); x++)
        {
            for (int z = 0; z < sim.WindField.GetLength(2); z++)
            {
                if (OnlyOneLayer)
                {
                    int y = SpecificLayer;
                    GL.Color3(1f, 1f, 1f);
                    GL.Vertex3(x * 10, 17f + y, z * 10);
                    GL.Color3(1f - sim.WindField[x, y, z].Length, 1f, 1f
                        );
                    GL.Vertex3((x * 10) + sim.WindField[x, y, z].X, 17f
                        + y, (z * 10) + sim.WindField[x, y, z].Y);
                    /* //Uncomment this to show interpolated vectors
                    GL.Color3(1, 0f, 1f);
                    if (x < 9 && z < 9)
                    {
                        Vector2 inter = VectorMath.CalculateLerpVector(
                            new Vector3(10 * x + 5, 17 + y, 10 * z + 5),
                            sim.WindField);
                        inter *= 5;
                        drawCommunicationLine(5 + x * 10, 5 + z * 10, 5
                            + x * 10 + inter.X, 5 + z * 10 + inter.Y);
                    }*/
                }
```

```
                else
                {
                    for (int y = 0; y < sim.WindField.GetLength(1); y++)
                    {
                        GL.Color3(1f, 1f, 1f);
                        GL.Vertex3(x * 10, 17f + y, z * 10);
                        GL.Color3(1f - sim.WindField[x, y, z].Length, 1f
                            , 1f);
                        GL.Vertex3((x * 10) + sim.WindField[x, y, z].X,
                            17f + y, (z * 10) + sim.WindField[x, y, z].Y)
                            ;
                    }
                }
            }
        }

        GL.End();
    }

    public void Resize()
    {
        int w = glControl.Width;
        int h = glControl.Height;
        GL.MatrixMode(MatrixMode.Projection);
        if (Perspective)
            projection = Matrix4.CreatePerspectiveFieldOfView((float)
                Math.PI / 4, w / (float)h, 1f, 1024.0f);
        else
            projection = Matrix4.CreateOrthographic(ZoomLevel * ((float)
                w), ZoomLevel * ((float)h), 1f, 1024.0f);

        GL.LoadMatrix(ref projection);

        GL.Viewport(0, 0, w, h); // Use all of the glControl painting
            area
    }

    internal void RotateCamera(double pitch, double yaw)
    {
        totalCamPitch += (float)pitch;
        totalCamYaw += (float)yaw;

        //Set maximum pith to 90 degrees,  google gimbal lock if you
            want to know why.

        if (totalCamPitch > 1)
        {
            totalCamPitch = 1;
        }
        if (totalCamPitch < -1)
        {
            totalCamPitch = -1;
        }
    }

    internal void Scroll(bool direction, double delta)
    {
        if (!Perspective)
```

```
        {
            if (direction)
            {
                ZoomLevel = ZoomLevel * (float)(1 + (0.002f * delta));
            }
            else
            {
                ZoomLevel = ZoomLevel * (float)(1 - (0.002f * delta));
            }
            Resize();
        }
        else
        {
            if (direction)
            {
                cameraPosition.Y++;
            }
            else
            {
                cameraPosition.Y--;
            }
        }
    }

    private void DrawBalloonDebugVelocity(Balloon b)
    {
        GL.Color3(1f, 1f, 1f);
        GL.Disable(EnableCap.Lighting);
        GL.Begin(BeginMode.Lines);
        GL.Vertex3(b.Position.X, b.Position.Y, b.Position.Z);
        GL.Vertex3(b.Position.X + b.Velocity.X, b.Position.Y, b.Position
            .Z + b.Velocity.Z);

        //Draw intented direction
        /*
        GL.Color3(1f, 1f, 1f);
        Vector2 targetVector = b.target * 10;
        GL.Vertex3(b.Position.X, b.Position.Y, b.Position.Z);
        GL.Color3(1f, 0.1f, 0.1f);
        GL.Vertex3(b.Position.X + targetVector.X, b.Position.Y, b.
            Position.Z + targetVector.Y);
        */
        GL.End();
        GL.Enable(EnableCap.Lighting);
    }

    private void DrawAreaBorder(Vector3 max)
    {
        GL.Disable(EnableCap.Lighting);
        GL.LineStipple(0, 3);
        GL.Enable(EnableCap.LineStipple);
        GL.Begin(BeginMode.Lines);
        GL.Color3(0.7f, 0.7f, 0.7f);

        GL.Vertex3(0, 0, 0);
        GL.Vertex3(max.X, 0, 0);

        GL.Vertex3(max.X, 0, 0);
```

```
            GL.Vertex3(max.X, 0, max.Z);

            GL.Vertex3(max.X, 0, max.Z);
            GL.Vertex3(0, 0, max.Z);

            GL.Vertex3(0, 0, max.Z);
            GL.Vertex3(0, 0, 0);

            GL.End();
            GL.Disable(EnableCap.LineStipple);
        }

        private void drawCircle(float radius, Vector2 position)
        {
            for (int i = 0; i < 360; i+=10)
            {
                double degInRad = i * 3.1416 / 180;
                //Draw under everything else
                GL.Vertex3(position.X + Math.Cos(degInRad) * radius, -1,
                    position.Y + Math.Sin(degInRad) * radius);
            }
        }

        internal void setSimulation(Simulation simulation)
        {
            this.sim = simulation;
        }

        private void drawCommunicationLine(float x, float z, float x1, float
            z1)
        {
            //Check the distance again to avoid strang communication lines
            if (VectorMath.DistanceSqrd(x, z, x1, z1) < 5000)
            {
                //Draw line above everything
                GL.Vertex3(x, 30, z);
                GL.Vertex3(x1, 30, z1);
            }

        }


    }
}
```

10.1.7 VertexBufferManager.cs

```
using OpenTK.Graphics.OpenGL;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LoonControl
{
    class VertexBufferManager
    {
        uint vbo_vertex;
        uint vbo_index;
```

```
uint vbo_normals;
uint vbo_color;

float[] vertices;
float[] normals;
ushort[] indices;

public void setMesh(float[] vertices, float[] normals, ushort[]
    indices)
{
    this.vertices = vertices;
    this.normals = normals;
    this.indices = indices;
    genAndBindBuffers();
}
public void setMesh(Vertex[] vertices, ushort[] indices)
{
    List<float> TempVertices = new List<float>();
    List<float> TempNormals = new List<float>();
    foreach (Vertex v in vertices)
    {
        TempVertices.Add(v.Position.X);
        TempVertices.Add(v.Position.Y);
        TempVertices.Add(v.Position.Z);

        TempNormals.Add(v.Normal.X);
        TempNormals.Add(v.Normal.Y);
        TempNormals.Add(v.Normal.Z);
    }
    this.vertices = TempVertices.ToArray();
    this.normals = TempNormals.ToArray();
    this.indices = indices;
    genAndBindBuffers();
}
public void Render()
{
    //VBO implementation

    GL.BindBuffer(BufferTarget.ArrayBuffer, vbo_normals);
    GL.NormalPointer(NormalPointerType.Float, 0, IntPtr.Zero);
    GL.EnableClientState(ArrayCap.NormalArray);

    GL.EnableClientState(ArrayCap.VertexArray);
    GL.BindBuffer(BufferTarget.ArrayBuffer, vbo_vertex);
    GL.VertexPointer(3, VertexPointerType.Float, 0, IntPtr.Zero);


    GL.BindBuffer(BufferTarget.ElementArrayBuffer, vbo_index);
    //This is the line that actually draws stuff
    GL.DrawElements(BeginMode.Triangles, this.indices.Length,
        DrawElementsType.UnsignedShort, IntPtr.Zero);

    GL.DisableClientState(ArrayCap.VertexArray);
    GL.DisableClientState(ArrayCap.NormalArray);
    GL.DisableClientState(ArrayCap.ColorArray);

}
```

```
        private void genAndBindBuffers()
        {
            GL.GenBuffers(1, out this.vbo_vertex);
            GL.BindBuffer(BufferTarget.ArrayBuffer, this.vbo_vertex);
            GL.BufferData(BufferTarget.ArrayBuffer, new IntPtr((sizeof(float
                ) * this.vertices.Length)), this.vertices, BufferUsageHint.
                DynamicDraw);
            GL.BindBuffer(BufferTarget.ArrayBuffer, 0);

            GL.GenBuffers(1, out this.vbo_normals);
            GL.BindBuffer(BufferTarget.ArrayBuffer, this.vbo_normals);
            GL.BufferData(BufferTarget.ArrayBuffer, new IntPtr((sizeof(float
                ) * this.normals.Length)), this.normals, BufferUsageHint.
                StaticDraw);
            GL.BindBuffer(BufferTarget.ArrayBuffer, 0);

            GL.GenBuffers(1, out this.vbo_index);
            GL.BindBuffer(BufferTarget.ElementArrayBuffer, this.vbo_index);
            GL.BufferData(BufferTarget.ElementArrayBuffer, new IntPtr((
                sizeof(ushort) * this.indices.Length)), this.indices,
                BufferUsageHint.StaticDraw);
            GL.BindBuffer(BufferTarget.ElementArrayBuffer, 0);

            //Check for errors
            ErrorCode glErr = GL.GetError();
            if (glErr != ErrorCode.NoError)
                Console.WriteLine(glErr);
        }
    }
}
```

10.1.8 SphereMesh.cs

```
using OpenTK;
using OpenTK.Graphics.OpenGL;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LoonControl
{
    //Adapted from http://www.opentk.com/node/1800
    class SphereMesh
    {
        Vertex[] sphereVertices;
        ushort[] sphereElements;

        public static Vertex[] CalculateVertices2(float radius, float height
            , byte segments, byte rings)
        {
            var data = new Vertex[segments * rings];

            int i = 0;

            for (double y = 0; y < rings; y++)
            {
                double phi = (y / (rings - 1)) * Math.PI; //was /2
                for (double x = 0; x < segments; x++)
```

```
            {
                double theta = (x / (segments - 1)) * 2 * Math.PI;

                Vector3 v = new Vector3()
                {
                    X = (float)(radius * Math.Sin(phi) * Math.Cos(theta)
                        ),
                    Y = (float)(height * Math.Cos(phi)),
                    Z = (float)(radius * Math.Sin(phi) * Math.Sin(theta)
                        ),
                };
                Vector3 n = Vector3.Normalize(v);
                Vector2 uv = new Vector2()
                {
                    X = (float)(x / (segments - 1)),
                    Y = (float)(y / (rings - 1))
                };
                data[i] = new Vertex() { Position = v, Normal = n,
                    TexCoord = uv };
                i++;
            }

        }

        return data;
    }
    public Vertex[] getVertices()
    {
        return sphereVertices;
    }
    public ushort[] getElements()
    {
        return sphereElements;
    }
    public SphereMesh(byte subdivisions)
    {
        sphereVertices = CalculateVertices2(1f, 1f, subdivisions,
            subdivisions);
        sphereElements = CalculateElements(1f, 1f, subdivisions,
            subdivisions);
    }
    public static ushort[] CalculateElements(float radius, float height,
        byte segments, byte rings)
    {
        var num_vertices = segments * rings;
        var data = new ushort[num_vertices * 6];

        ushort i = 0;

        for (byte y = 0; y < rings - 1; y++)
        {
            for (byte x = 0; x < segments - 1; x++)
            {
                data[i++] = (ushort)((y + 0) * segments + x);
                data[i++] = (ushort)((y + 1) * segments + x);
                data[i++] = (ushort)((y + 1) * segments + x + 1);

                data[i++] = (ushort)((y + 1) * segments + x + 1);
```

```
                    data[i++] = (ushort)((y + 0) * segments + x + 1);
                    data[i++] = (ushort)((y + 0) * segments + x);
                }
            }

            // Verify that we don't access any vertices out of bounds:
            foreach (int index in data)
                if (index >= segments * rings)
                    throw new IndexOutOfRangeException();

            return data;
        }
    }
}
```

10.1.9 Vertex.cs

```
using OpenTK;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LoonControl
{
    public struct Vertex
    { // mimic InterleavedArrayFormat.T2fN3fV3f
        public Vector2 TexCoord;
        public Vector3 Normal;
        public Vector3 Position;
    }
}
```

10.1.10 VectorMath.cs

```
using OpenTK;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LoonControl.Util
{
    class VectorMath
    {

        //  0 <----u----> 1
        //  a ----------- b    0
        //  |             |   /|\
        //  |             |    |
        //  |             |    v
        //  |  *(u,v)     |    |
        //  |             |   \|/
        //  d------------ c    1
        public static Vector2 QuadLerp(Vector2 a, Vector2 b, Vector2 c,
           Vector2 d, float u, float v)
        {
            Vector2 abu = Vector2.Lerp(a, b, u);
            Vector2 dcu = Vector2.Lerp(d, c, u);
```

```
            return Vector2.Lerp(abu, dcu, v);
        }

        public static double Distance(float x1, float y1, float x2, float y2
            )
        {
            return Math.Sqrt(Math.Pow(x1 - x2, 2) + Math.Pow(y1 - y2, 2));
        }

        public static double DistanceSqrd(float x1, float y1, float x2,
            float y2)
        {
            return (Math.Pow(x1 - x2, 2) + Math.Pow(y1 - y2, 2));
        }

        public static Vector2 CalculateLerpVector(Vector3 targetposition,
            Vector2[, ,] wind)
        {
            //0,0
            Vector2 va = wind[(int)(Math.Floor(targetposition.X / 10)), (int
                )(targetposition.Y - 17), (int)Math.Floor((targetposition.Z /
                 10))];
            //10,0
            Vector2 vb = wind[(int)(Math.Ceiling(targetposition.X / 10)), (
                int)(targetposition.Y - 17), (int)Math.Floor((targetposition.
                Z / 10))];
            //0,10
            Vector2 vc = wind[(int)(Math.Floor(targetposition.X / 10)), (int
                )(targetposition.Y - 17), (int)Math.Ceiling((targetposition.Z
                 / 10))];
            //10,10
            Vector2 vd = wind[(int)(Math.Ceiling(targetposition.X / 10)), (
                int)(targetposition.Y - 17), (int)Math.Ceiling((
                targetposition.Z / 10))];

            return Util.VectorMath.QuadLerp(va, vb, vc, vd, (targetposition.
                X % 10) / 10f, (float)(targetposition.Z % 10) / 10f);
        }

        public static double PointToLineDistance(Vector2 lineStart, Vector2
            lineEnd, Vector2 point)
        {

            return Math.Abs((lineEnd.X - lineStart.X) * (lineStart.Y - point
                .Y) - (lineStart.X - point.X) * (lineEnd.Y - lineStart.Y)) /
                     Math.Sqrt(Math.Pow(lineEnd.X - lineStart.X, 2) + Math.
                         Pow(lineEnd.Y - lineStart.Y, 2));
        }

    }
}
```

10.1.11 Keyboard.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
```

```
using System.Windows.Forms;

namespace LoonControl
{
    public abstract class Keyboard
    {
        [Flags]
        private enum KeyStates
        {
            None = 0,
            Down = 1,
            Toggled = 2
        }

        [DllImport("user32.dll", CharSet = CharSet.Auto, ExactSpelling =
           true)]
        private static extern short GetKeyState(int keyCode);

        private static KeyStates GetKeyState(Keys key)
        {
            KeyStates state = KeyStates.None;

            short retVal = GetKeyState((int)key);

            //If the high-order bit is 1, the key is down
            //otherwise, it is up.
            if ((retVal & 0x8000) == 0x8000)
                state |= KeyStates.Down;

            //If the low-order bit is 1, the key is toggled.
            if ((retVal & 1) == 1)
                state |= KeyStates.Toggled;

            return state;
        }

        public static bool IsKeyDown(Keys key)
        {
            return KeyStates.Down == (GetKeyState(key) & KeyStates.Down);
        }

        public static bool IsKeyToggled(Keys key)
        {
            return KeyStates.Toggled == (GetKeyState(key) & KeyStates.
               Toggled);
        }
    }
}
```

10.1.12 Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace LoonControl
{
    static class Program
```

```
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new MainForm());
        }
    }
}
```