Investigation and Implementation of Low-Cost UAV Sensor Systems for Constructing Object and Topographical 3D Models in Near-Real-Time

EECE MEng4 FYP – Final Report

Yousef Amar
(1095307)

2014-04-14

Abstract

Traditional terrain mapping techniques are laborious and often expensive, sometimes to an impractical degree. In recent times however, inexpensive UAVs have become increasingly accessible yet the hardware remains costly. The aim of this project is to investigate and develop systems for turning real world physical terrain and structures into high fidelity virtual 3D models as easily and cheaply as possible, with the aid of UAV-mounted sensors, for potential use in such fields as exploration and heritage.

Contents

1 Executive Summary

This report investigates exiting UAV mapping technology, and details the progress and results on the endeavour to build a low-cost UAV sensor systems for constructing object and topographical 3D Models as well as the discussion thereof. It also describes all hardware and software implementations in great detail, and their testing and validation. Finally, the results and outcomes of this project are analysed and their implications explored.

1.1 Mission Statement

The goal is to contribute to a unique UK Digital Heritage Project designed to show future/potential government sponsors what can be achieved with UAV technologies in support of mapping areas of important rural and maritime heritage in the south west of the country.

1.2 Aims and objectives

The aim of this project is to investigate, design, and implement systems that can convert raw sensor data from a UAV, such as IMU, LIDAR, and GPS readings, into a coherent and accurate 3D model. This conversion should be done in real-time with as small a latency as possible and the resulting model should update accordingly as the data increases.

1.3 Research challenges

This project is an exercise in software and hardware design, sensor fusion, and the processing and validation of information, as well as making the best of limited options. It involves solving academically formidable problems, translating the theory into practice, and subsequently analysing, evaluating, and validating the results. It also combines existing technologies to create something new that is greater than the sum of its parts.

2 Introduction

In an age of information, communication, and technology, the physical world is the sole limiting factor to the rate of increase in data and the betterment of technology. Collecting data from the real world loses efficiency very rapidly when confronted with problems such as cost, time, labour, logistics, legality, and even just the laws of physics. It is no surprise that there is a severe demand for cheap solutions in data acquisition and sensor technology especially in times of dire economic climate such as these.

The title of this final year MEng project is *Investigation and Implementation of Low-Cost UAV Sensor Systems for Constructing Object and Topographical 3D Models in Near-Real-Time*. A most important keyword here is *low-cost*. As is explored further in this report, terrain mapping has indeed been done before however traditional terrain mapping techniques are laborious and often expensive, sometimes to an impractical degree. Ground-based mapping takes remarkably long and, depending on the setting, can be wholly impossible. Getting a decent model with high fidelity using an aerial system requires a specialised, full-sized helicopter with bespoke hardware that can only be manufactured when backed by military funding.

2.1 Background

In recent times, inexpensive UAVs in the form of remotely controlled quad- or hexacopters have become increasingly accessible. Having access to a method of virtualising the topography and metadata of terrain cheaply through the use of such UAVs would undoubtedly be of great benefit especially in areas such as defence, construction, exploration, heritage, surveillance, investigation, and reconnaissance, amongst many others.

However, the sensor hardware that can be used to scan terrain using such systems is unfortunately still developing and the quality and feasibility of certain solutions, such as those based on LIDAR, is still strongly correlated to price. There is hope yet though, since if one shifts the context of the problem from a hardware one to a software one, it becomes solvable given the existing constraints, which is what this project aims to do.

This project was conducted as a final year, MEng individual project. Although this project is only worth 40 credits towards the MEng degree rather than the usual 60, the endeavours it elicits are just as valuable.

At the beginning, the final objective of this project was decided on after careful consideration on what existing areas of active research could benefit. This was done after looking at a number of limitations in a series of email exchanges with this project's supervisor. Eventually the demand for something such as a stabilised on-board LIDAR sensor, with the aim of scanning real-world data that can then be converted into VR/AR images, was recognised. The obvious potential safety and legality issues that may arise are also discussed separately in this report. Subsequently, once it was decided that this would indeed be a unique and challenging project worthy of an MEng FYP, the final project specifications were determined. They can be listed as follows (as typed verbatim on the project specification form):

2.2 Project Specification

2.2.1 Expected Outcomes

The aim of this project is to design and implement a system that can convert raw sensor data from a UAV, such as IMU, LIDAR, and GPS readings, into a coherent and accurate 3D model. This conversion should be done in real-time with as small a latency as possible and the resulting model should update accordingly as the data increases. This project is an exercise in software design, sensor fusion, and the processing of information, as well as making the best of limited options.

2.2.2 Fall-back and Rebuild Position

If all else fails, the minimum to be delivered is a system capable of the following:

- To render a 3D model of terrain data obtained independently (i.e. not in real-time) from a stationary point.
- To demonstrate core functionality and feasibility in a lab setting.
- To demonstrate theoretical feasibility of any algorithms that can be used to provide a model of higher fidelity.
- All with the same conditions and hardware.

2.2.3 Enhancement Position

If all expected outcomes are achieved, further achievable enhancement objectives are:

- To enhance the topological model by adding textures obtained from an on-board camera or satellite imagery corresponding to recorded GPS coordinates.
- To enhance the topological model with terrain metadata such as a thermal map built using on-board thermal sensors for potential use in such functions such as detecting gas leakages in a civilian setting, or camouflaged units in a military setting.

It is important to realise however that the above specifications ceased to match the research conducted and work undertaken about halfway through the first term of the time period. Instead, the project has expanded and evolved quite significantly after more information was gathered and as time passed on.

The entire process of what is attempted to be achieved here from a top-level view can be summarised with the aid of the following diagram.

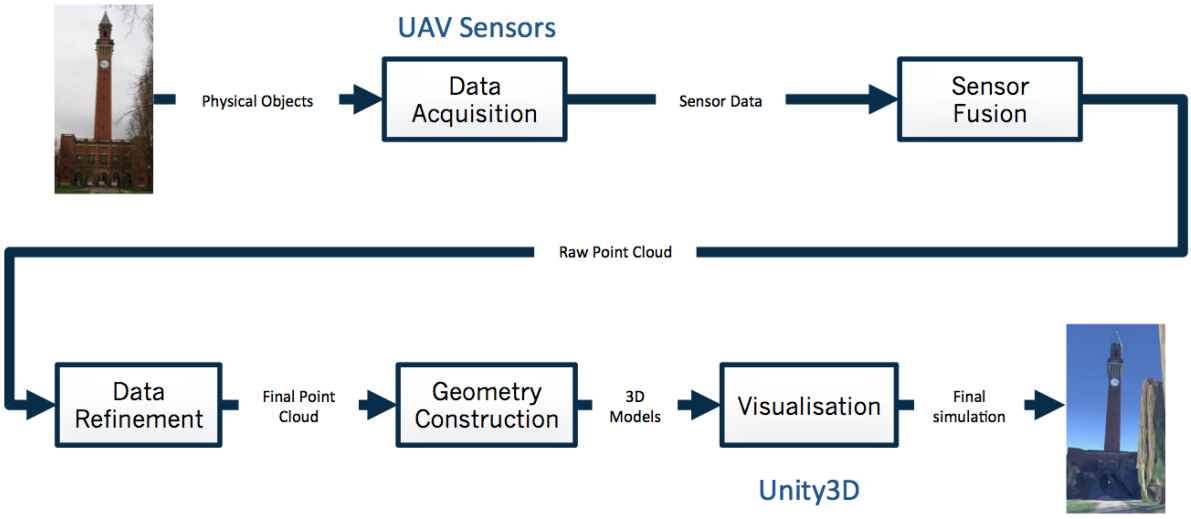


Figure 1: Graphical Representation of the Project Pipeline

Ideally this would all be in real-time. More data would stream from the UAV as fast as possible essentially forming a large data pipeline. The difficulty herein lies on a higher level; how can the final model be made to be as true to reality as possible, without nuking the budget? The answer is in the details. The entire process must be as optimised as possible. It has to reach a point where the simulation can be immediately recognised as a depiction of the real deal and look like it was modelled by hand.

Along with all the required tasks described formerly, the algorithms researched and devised must be applied in such a way that the final result is greater than the sum of its parts. Each process in the above flow chart is a challenge in itself.

To summarise, data would be obtained through the sensors on the UAV, fused and cleaned on a standard PC in any programming or scripting language or even in the visualisation stage itself as discussed earlier then visualised using native OpenGL (in

C/C++ using SDL or even Java using JOGL/LWJGL). Ideally however, the final model should be visualised in Unity3D which can be coded from scratch or with the aid of packages such as PointCloud SDK.

One does not need to be an expert in the field to realise that this endeavour is something that would take much more than the efforts of a single undergraduate MEng student over the course of a single academic year; indeed due to the unprecedented nature of this process, it is more suitable to be carried out by a PhD-level research group.

As such, the initial goal of the project was to simply investigate and implement various algorithms for converting point clouds — such as those obtained from a LIDAR sensor — into a 3D mesh using Unity3D (i.e. the bottom half of the 1 pipeline).

The aims of the project have since expanded to the upper half as well, for reasons explained later. Furthermore, the scope of the project has expanded yet again to include additional solutions and combinations thereof instead of one single solution. These are all discussed in section 4.3 on page 30.

2.3 Summary of Supervisor Meeting Minutes and Discussion

Throughout the meetings with this project's supervisor, Professor Bob Stone, a range of matters were discussed (see logbook for detailed meeting forms). Apart from discussing progress since the last meeting, other plans and possibilities were discussed with both Professor Bob Stone and sometimes one of his PhD students, Chris Bibb, who's work involves piloting a hexacopter with which this system could potentially be integrated with. Since defining the specification the meetings were used for recapitulation of past progress and advice and briefing on future plans. These include but are not limited to:

- Discussion of paperwork such as:
  - Project specification form
  - Ethics forms
  - Hazard and risk assessment forms
  - Plagiarism and extenuating circumstances forms
- Procurement of components from third party firms as well as the university and budget options
- Demonstration of prototypes and subsystems that would aid the completion of the project
- Potential sites and structures for testing and verification such as:
  - H.M.S. Anne's hull remnants
  - The Purton Ships' Graveyard
  - The Longstone Manor ruins at Burrator Reservoir
  - Areas of important rural and maritime heritage in general in the west/south west of the country
- Advice on and review of reports and presentations
- Discussion of next steps, direction, and future plans
- Discussion of context and real world applications of work

3 Analysis

Before starting to work on building a solution and completing the project, the problems must be analysed and the requirements completely understood. These are inferred from the MEng FYP handbook as well as the project specification.

3.1 Requirements Analysis

3.1.1 General

- All activity must be legal and should not bring the university into disrepute.
- Costs must not exceed the 100 budget limit.
  - Fortunately, a lot of existing hardware can be utilised and the costs for software needed are very small.
- All risks must be properly assessed.

3.1.2 Project Specific

- Hardware should be procured cheaply or built from scratch if need be.
  - Manufacturers should be contacted directly and pressed for economical deals.
  - Parts can be solicited from the department especially due to the departments history of running UAV projects for MEng3 groups and retaining the parts on conclusion.
- Raw sensor data should be converted into a coherent and accurate 3D model.
  - Different sensor data should be fused for higher accuracy.
  - Readings should be parsed from into a format that can be processed to be displayed in 3D visualisation software (e.g. Unity3D).
- The model should update as the data increases.

- A timing-critical code should be very fast on the order of milliseconds.
- Metadata should be collected and overlaid.
- Visual and thermal data should be collected by a camera (or queried from online satellite imagery) and a thermal sensors respectively, and virtualised.

In order to explore how the goals of this project can be accomplished, literature appropriate to the purpose of this project must be researched.

3.2 Literature Review and Summary of Research

The University of Birmingham is already in possession of a hexacopter with a gimbal rig in place that would help stabilised any sensors and payload that rely on being immune to sudden surges in motion and the resulting noise that would distort the recorded data. Therefore, at first, research has been conducted under the assumption that data from on-board IMU and GPS sensors can already be obtained and that a LIDAR sensor of reasonable weight can be mounted onto the UAV with relative ease. Current prototypes however work as complete standalone solutions that do not rely on on-board electronics and sensors.

3.2.1 Physical World to Point Cloud

The first stage of the process to be researched is depicted on the top half of the pipeline diagram, figure 1, on page 5. For this a number of methodologies were researched.

LIDAR

To begin with, LIDAR sensors were investigated. LIDAR systems work on relatively simple principles similar to echolocation but with sound; rapid pulses of laser are shined onto an object and the time taken for the light signal to reflect off of said object and return to a detector is measured. Depending on the time it takes for the light to return, the distance between the emitter and the object along the line of the beam can be determined to an extremely high resolution (LiDAR-UK, 2008).

Of course, the speed of light warrants the use of very fast devices to detect the time difference between emission and detection, which is one of the reasons commercial LIDAR sensors are so expensive. The measured distance in essence gives sort of a polar coordinate that can be converted into a Cartesian coordinate relative to the position of the emitter. To calculate the distance, the time taken for the roundtrip is simply multiplied by the speed of light and then divided by two. In other words:

$$x = \frac{ct}{2} - \epsilon$$

There exists a small error that is less than or equal to the inverse of the detector's frame-rate plus a negligible error due to the laser source being offset from the centre of the detector.

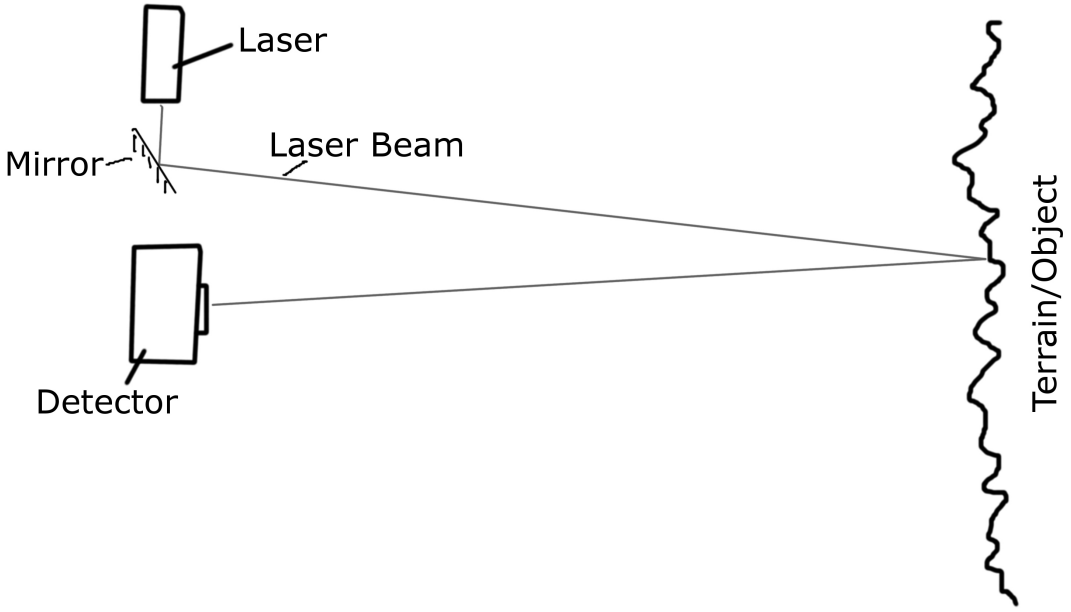


Figure 2: Digitisation of Logbook Sketch of LIDAR

Once distances can be accurately measures, systems of mirrors or motors can be added for "scanning". Depending on the functionality required, different methods exist for doing so. Objects can be "swept" for a series of coordinates, if the angle of the laser beam is rapidly changed over time, using a small pivoting mirror for example, and the orientation of the mirror is also recorded. Doing this allows you to scan physical 3D objects as a flat-bed scanner would do by moving the emitter on a plane parallel to the object. A more complex system might emit a 2D grid of laser beams in one frame essentially taking "snapshots" of the object or terrain and then combining them.

Finally, other systems, mainly concerned with scanning objects around the same height as the emitter, might scan in circles outwards like a lighthouse would do. The final point map is then constructed as radar would and objects are scanned by moving

the emitter on the Y-axis. This is especially useful for indoor navigation (Achtelik et al., 2009) as well but still very effective outdoors (Phoenix Aerial Systems, 2012).

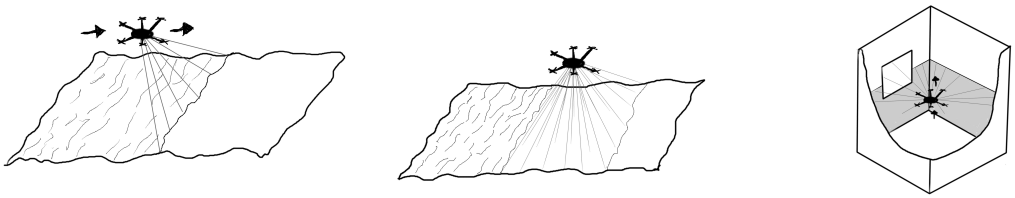


Figure 3: Digitisation of Logbook Sketch of Scanning Mechanisms

Knowing the position relative to the object is also required. This is usually done with the aid of GPS which yields positions relative to world space. These positions are used to offset the point cloud data on a frame to frame basis to calculate the world-space point cloud coordinates. GPS is however known to be very inaccurate especially civilian GPS (GPS.gov, 2013). This problem is usually solved using expensive hardware however, and unless the point clouds built for this project are satisfactory, computer vision techniques will be used to further refine the data as is expanded upon further in this report.

Furthermore, it is typical for IMU data to be used to filter out noise from the LIDAR measurements (Vallet, 2007). It is expected that wind and other external factors beyond our control may distort the data and skew or shift data mid-scan. Individual points can thus be corrected by recording these turbulences and cleaning the data by filtering them out by oftentimes only using simple transformations.

Additionally, occluded areas of terrain would have to be "in-filled". This can be done at a late post-processing stage after the point cloud has already been converted into geometry. Ideally, this should be done automatically. A common way of filling in missing geometry is Singular Spectrum Analysis (SSA) or Multi-channel SSA, which has been tried and tested (Kondrashov and Ghil, 2006), and doing so was explored for the purposes of this project. It may simply be enough to perform cosine, cubic, or even just bilinear interpolation to fill in the gaps.

To obtain the final point cloud there are a number of processes that may or may not have to be carried out. When collecting the data, the series of measured distances must be combined to form the final point cloud. Some LIDAR sensors do this internally

already, presenting the user with a point cloud from the get-go. Once all point cloud "frames" are pre-processed with the aid of the other sensors and data, they must be stitched together to form one large point cloud. Image stitching is a science to itself in the field of Computer Vision and is usually accomplished using existing libraries such as the popular OpenCV library.

For point clouds, there exists a well-known, cross-platform library appropriately called Point Cloud Library (PCL) developed internationally by a large number of specialists, corporations, and universities. It contains code for algorithms such as those required for stitching as well as a host of other useful functions that are actually pretty exciting and would enhance the final output significantly. The use of such libraries in this project is to be abundant, as is done in industry, since there is no reason to reinvent the wheel.

For reasons discussed in section 4.3 on page 30, the use of an actual LIDAR sensor on a project of this magnitude is all but infeasible. For this reason, other methods were also researched. One of these methods is stereoscopic matching.

Stereoscopic Matching

Stereoscopic matching is a concept with many tried and tested techniques and algorithms. It is based on the same concepts that explain perceiving depth with normal human binocular vision. The most prevalent dataset used for research on stereo vision algorithms is that provided by the University of Tsukuba's Computer Vision Laboratory (CVLAB, 2012). Figure 4 shows processing of the classic Tsukuba example images (left and right cameras respectively) into a disparity map using OpenCV (code for doing this is discussed in 4.3.3 on page 40)

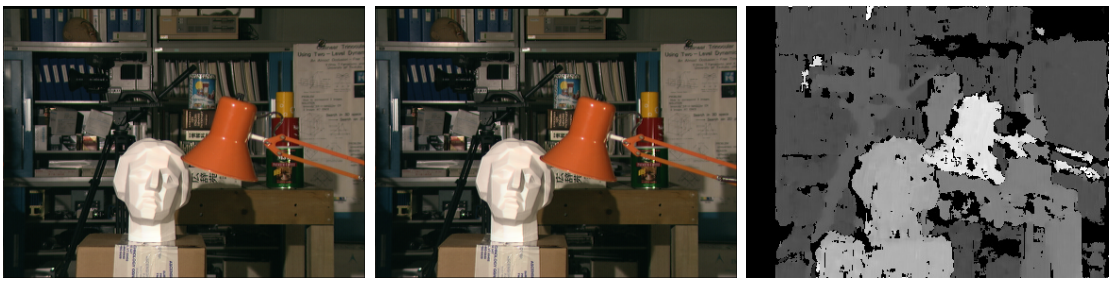


Figure 4: Example of Tsukuba Stereo Disparity Mapping

As is evident from the above disparity map, figuring out depth through stereo matching algorithms is not always very precise and often subject to a lot of noise that is only very difficult to figure out. Stereo matching is often used to isolate objects from their backgrounds rather than build 3D models, however the latter is likewise possible especially for objects where coarseness is not an issue such as terrain.

For the purpose of this project, two common algorithms are investigated for stereo matching: the Sum of Absolute Differences (SAD) and the Sum of Squared Differences (SSD) which is also known as Squared Deviations and is related to the Least Squares algorithm discussed in the *Point Cloud to Mesh* section. These can be expressed mathematically as follows (Scharstein and Szeliski, 2002):

$$\sum_{(i,j)\in W} |I_1(i,j) - I_2(x+i, y+j)|$$

Figure 5: Sum of Absolute Differences (SAD)

$$\sum_{(i,j)\in W} \big(I_1(i,j) - I_2(x+i, y+j)\big)^2$$

Figure 6: Sum of Squared Differences (SSD)

Here, i and j are the column and row index respectively where W is a window of pixels of a pre-defined size. I_1 and I_2 are the two images (left and right) and x and y are the positions, in pixels, of the window W as it iterates through the images. For each position of the window on an image, the SAD or SSD is calculated and the result used to determine how far away any given part of of the image is. There are a lot of parameters that can be tweaked and modified for instance the SAD/SSD window size.

For example, if you convert an image to greyscale and treat the colour values as numbers from, usually, 0 (black) to 255 (white), then it is easier to describe SAD/SSD as a series of steps.

To simplify, assume there is a single image (I) and a template (T) and you wanted to match a template to it with a size equal to W (as opposed to matching two sub-images within two stereoscopic images). Both image matrices are populated with random

numbers in the range $[0 - 255]$.

$$T = \begin{bmatrix} 84 & 200 & 16 \\ 56 & 238 & 11 \\ 251 & 206 & 203 \end{bmatrix} \qquad I = \begin{bmatrix} 160 & 40 & 250 & 27 & 114 \\ 81 & 94 & 14 & 100 & 37 \\ 225 & 52 & 148 & 137 & 111 \end{bmatrix}$$

Here, the template T can be compared against the image I in three different places. These are the 3x3 sub-matrices that encapsulate 160 & 148, 40 & 137, and 250 & 111 respectively. For example, to calculate the SAD values for the three 3x3 windows in I, one would iterate through those windows. To illustrate:

$$I_1 = \begin{bmatrix} 160 & 40 & 250 \\ 81 & 94 & 14 \\ 225 & 52 & 148 \end{bmatrix} \qquad I_2 = \begin{bmatrix} 40 & 250 & 27 \\ 94 & 14 & 100 \\ 52 & 148 & 137 \end{bmatrix} \qquad I_3 = \begin{bmatrix} 250 & 27 & 114 \\ 14 & 100 & 37 \\ 148 & 137 & 111 \end{bmatrix}$$

Then $|T - I_n|$ would give:

$$I_1' = \begin{bmatrix} 76 & 160 & 234 \\ 25 & 144 & 3 \\ 26 & 154 & 55 \end{bmatrix} \qquad I_2' = \begin{bmatrix} 44 & 50 & 11 \\ 38 & 224 & 89 \\ 199 & 58 & 66 \end{bmatrix} \qquad I_3' = \begin{bmatrix} 166 & 173 & 98 \\ 42 & 138 & 26 \\ 103 & 69 & 92 \end{bmatrix}$$

The sums of all the values within these three matrices are 877, 779, and 907 respectively. Meaning the middle section of the image, I_2 is the most similar, or least different, to the template. Thus the middle of the image is determined to be the closest match to the template. Some noise and false matches can be filtered out by setting a threshold above with a SAD value can be classified as a true match, i.e. a value of 779 could be ignored and replaced for the disparity map.

This process can then be expanded to two large stereoscopic images from slightly different angles with moving windows. On finding the best match between windows, a disparity map can be built using the normalised relative distances between two matching

windows from both images. Visually, the disparity map would simply look like an image where the whiter pixels are nearest to the cameras and the black pixels furthest.

The black border edges that are visible in figure 4 on page 13 are a result of areas that one camera sees but not another and would be cut out of the resulting images. The final disparity map only contains values for parts of the view visible in both images. Incidentally the size of the black area could be used to estimate camera distances and angles, however there are no well known, documented cases of academic research on this having been done to date.

Structure from Motion

Another optical, image processing based technique for building point clouds from objects is commonly referred to as Structure from Motion (SfM). SfM part of the field of photogrammetry which many industrial or commercial applications utilise such as Apple Maps. Like stereoscopic matching, it has the advantage that the collected data gives you colour/texture information for "free", unlike LIDAR.

This method bears a number of other similarities to stereoscopic matching too. Instead of matching pixels from two images from slightly different perspectives, features from a large number of images are matched and through identifying these features, camera positions are inferred and a point cloud is reconstructed.

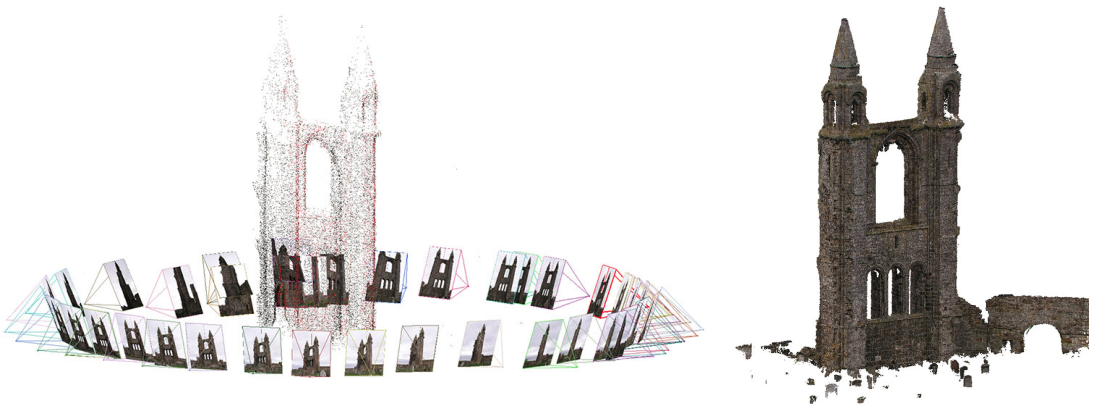


Figure 7: Dense Point Cloud Reconstruction through SfM (Source: nframes.com)

Photogrammetry is a science in itself with a myriad of research and development

behind it. Nowadays, tools can be used to make the process more efficient such as *VisualSFM* or the *Python Photogrammetry ToolBox*. This project explores the use of one particular algorithm known as *Scale-invariant feature transform* or SIFT, published by David Lowe (1999), (2004) and is a cornerstone for the relatively new field of photogrammetry; cited tens of thousands of times. This algorithm is used to detect features within an image and this is what makes it a different paradigm to stereoscopic matching.

In a nutshell, and this only scratches the surface, the SIFT algorithm works as follows:

1. Apply a Gaussian blur to an input image to reduce noise
2. Identify local feature points that are relatively immune to perspective distortions and transformations (translation, rotation, scaling) and lighting variations
    (a) Select points that contrast their background significantly (fixed or dynamic threshold) by processing histograms
    (b) Store feature points in an array of vectors
3. Match vectors across images
    (a) E.g. via the Random Sample Consensus (RANSAC) method (Fischler and Bolles, 1981), often used for filtering outliers out of a dataset
4. Use the change in positions of vectors across images to calculate feature and camera positions

The result of this process through common tools is a point cloud, that is often in PLY format, which can be processed with other software such as MeshLab — a program that can be used to modify and process point clouds as well as 3D geometry — and through MeshLab, be exported in any common format such as OBJ, or better yet, LAS.

3.2.2 Point Cloud to Mesh

After getting the final point cloud data, things become significantly easier. There are a whole range of algorithms that can be used to build 3D models. First and foremost,

objects will most likely need to be identified and isolated from the rest of the point cloud. The nice thing about working with point cloud data is that simple clustering algorithms such as K-Means clustering (MacQueen, 1967) can work wonders.

A number of centroids are selected using methods such as agglomerative or divisive clustering and then through the K-Means algorithm, these centroids are moved iteratively, by finding the average position of the points closest to any given centroid (essentially dividing the points up as a Voronoi diagram would). The centroids are set to the new means and the process is repeated until the positions of the centroid no longer change. Finally, any given sample is assigned to the cluster that the nearest centroid to it defines.

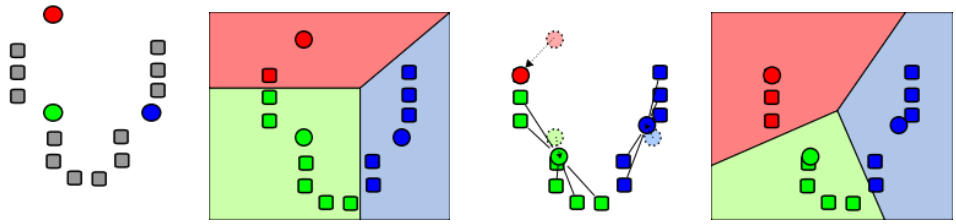


Figure 8: Demonstration of K-Means Clustering; Source: Wikipedia, K-Means Clustering

This works for as many dimensions as necessary and helps isolate objects within the point map such as trees or buildings so that they do not have to be selected manually. Alternatively PCL, mentioned before, contains methods of segmentation that can be repurposed to fit our needs. At the same time, point metadata can help classify certain points. For example, if you also had corresponding colour images of the structure recorded, each point can be assigned a colour and points with a similar colour are more likely to belong to the same object.

Then there are a few ways that the resulting separated point clouds can be rendered as geometry. Either they can be sorted then put directly into a Vertex Buffer Object (VBO) and sent off to the GPU, effectively treating each point as a vertex, which although naïve, would be very easy. Even graphics card that only support legacy OpenGL would be able to render these primitives fairly easy without the need for added tessellation.

The two modes in the blue boxes in figure 9 for instance, could be used to render the shapes automatically with the only provision being that the vertices are sorted into the correct order which is a trivial task. There are however other, perhaps more elegant,

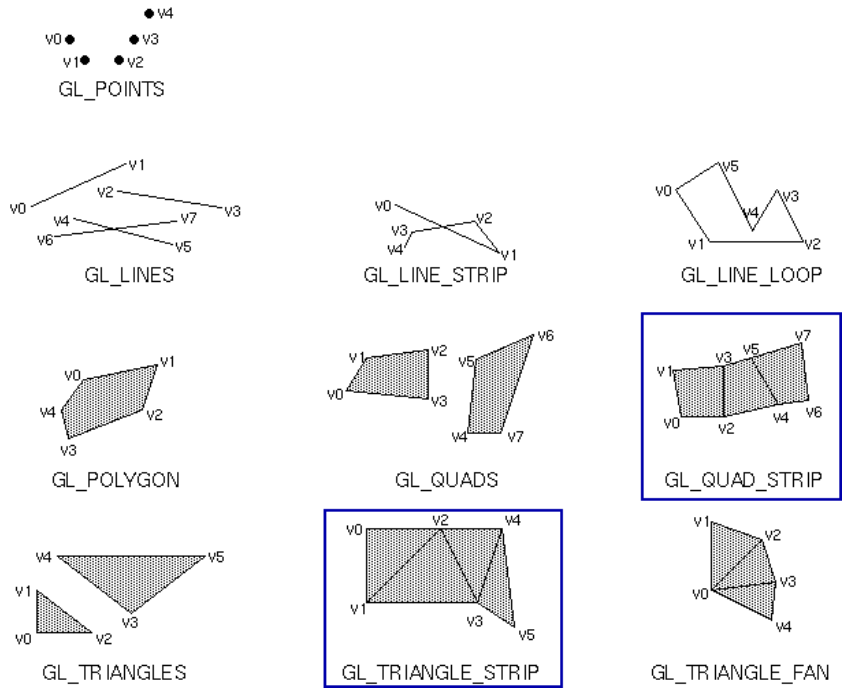


Figure 9: Geometric Primitive Types; Source: http://fly.cc.fer.hr/

algorithms for building the geometry that may even increase the quality of the final model.

A technique intended to be used, as is described later on in this report, was devised specifically for this project. Essentially, each point in the point cloud of any given object is treated as a very tiny pseudo-voxel. Imagine then a rubber balloon is stretched over these tiny voxels, building a mesh shaped around them. The balloons stretchiness is then relaxed to fit snugly around the object.

This kind of implementation has already been done before many times but in a different context and to solve different problems. The most common one is mesh smoothing (Bade, Haase, and Preim, 2006) for normal geometry using surface relaxation algorithms. Unlike isocontouring, where you simply repeatedly chop corners off of a shape, or marching cubes (Lorensen and Cline, 1987) where you store the smoothed shape of all possible configurations of voxels in a look-up table, these surface relaxation algorithms start by surrounding all voxels on an object with a single loop of vectors — or in the case of 3D graphics, a mesh — then progressively relaxing the mesh until it fits onto the voxels. It can then be smoothed even further depending on the kind of object (e.g. organic vs.

mechanical) preferably using Bézier or Cubic Hermite splines for added visual appeal. It does not matter if the shape is hollow because that part will not be seen by the user anyway.

Then there are libraries like PCL which support greedy algorithms like fast triangulation of unsorted point cloud data. This works by "growing" meshed from each point. A point will look for a number of nearest local neighbours to connect to and neighbours. The results are generally quite ugly though and end with an excessive number of triangles. The final model is not very smooth and transitions between areas of high and low point densities are immediately noticeable.

Other algorithms that were found include for example Delaunay Triangulation (Delaunay, 1934) which is an algorithm which helps avoid skinny triangles and can make for efficient tessellation.

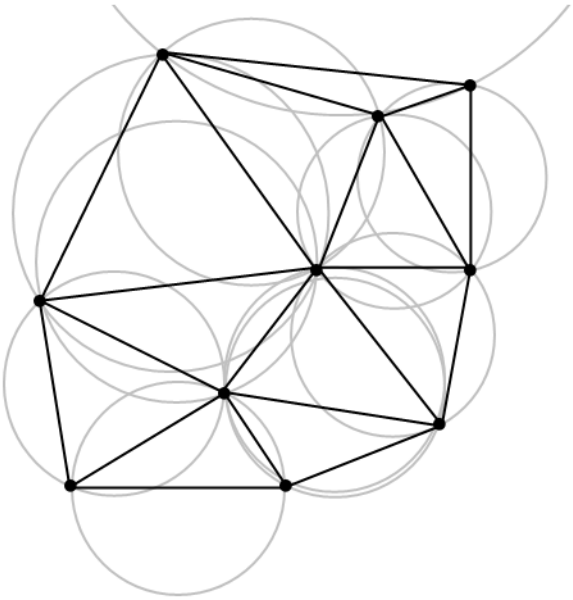


Figure 10: A Delaunay triangulation in the plane with circumcircles shown; Source: Wikipedia, Delaunay triangulation

Most CAD software have an option built-in for converting points to meshes and there are a number of other techniques that can be used to build meshes from points, many of which were explored throughout the course if this project.

A number of potential algorithms with differing complexities and feasibilities were ultimately listed as potential candidates. These barely scratch the surface of the sea of

other papers on completely unique algorithms out there but the following were the ones deemed most worthy.

Marching Cubes/Diamonds (Lorensen and Cline, 1987)

The famous marching cubes algorithm, or its cousin that circumvented patent laws in the past, marching diamonds, uses predefined geometry that is rendered depending on an occupied cell and its surrounded neighbours. For example, a cell that is completely surrounded would be invisible while one that is surrounded everywhere except on top would look like a flat horizontal plane that connects to its neighbours etc.

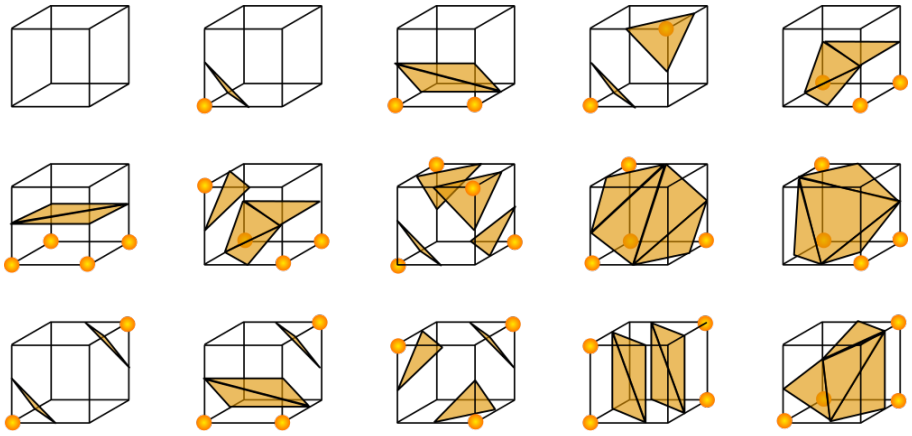


Figure 11: The original 15 cube configurations; Source: Wikipedia, Marching cubes

Surface Mesh Relaxation

As already discussed, surface mesh relaxation, as I like to call it, is where an existing mesh of a pre-defined or dynamically allocated size is taken and procedurally "shrunk" to specified degree on top of a cluster of points such that none of the points protrude. This is akin to shrink wrapping something.

Voronoi (1908) Cells and Delaunay (1934) Triangulation

As similarly discussed formerly, cells in a Voronoi diagram have borders that are equidistant between adjacent points while Delaunay triangulation is like the inverse of

that and provides you with optimal triangles rather than polygons. Some examples of algorithms for obtaining these are:

- Fortune's algorithm, a sweep line algorithm for generating a Voronoi diagram
- The Bowyer-Watson algorithm, an incremental algorithm for computing Delaunay triangulation
- Lloyd's algorithm, AKA Voronoi iteration, which is very closely related to K-means clustering discussed above

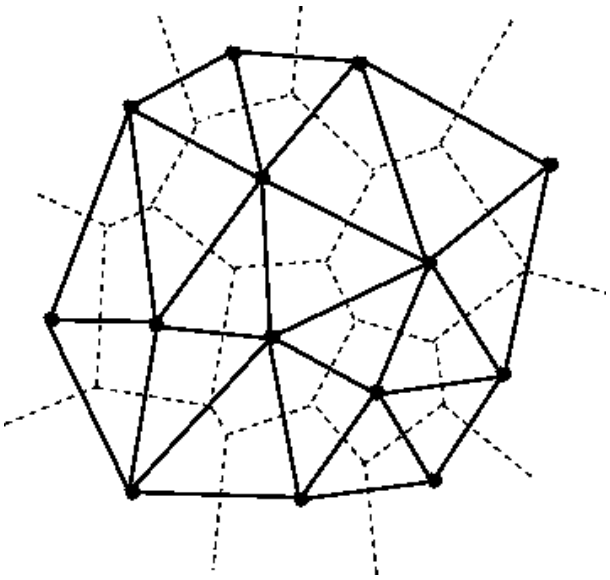


Figure 12: Delaunay triangulation, on top of a Voronoi diagram; Source: http://www.comp.lancs.ac.uk/ kristof/research/notes/voronoi/

Moving Least Squares (Hartley and Zisserman, 2000)

A very common algorithm for this kind of application that is a bit like least squares but for a morphing mesh. It can be specifically optimised for this purpose and can be used to increase the resolution of the final model. This is best used in conjunction with Multi-channel SSA as discussed before and is similar to advances interpolation techniques. This algorithm is used with some very impressive applications to do things such as iteratively grow a dynamic mesh from a "seed" triangle (Karkanis and Stewart, 2001) or even change

triangle sizes procedurally depending on the level of detail required and how feature rich and area is (Scheidegger, Fleishman, and Silva, 2005).

In addition to mesh construction, research was conducted in alternatives for the infamously expensive LIDAR sensor. It was discovered that something strikingly similar to the method developed (discussed in detail in the progress section) is already in use (Deyle, 2009). Range-finding in this manner has certain software requirements however. For one, one needs to use computer vision approaches to detect a laser pointer on an image, which is fortunately not too difficult (Meško and Toth, 2013), as is further explained in the next section of this report.

4 Procedure

The project was carried out in a relatively systematic manner. This project was an exercise in software design, sensor fusion, and the processing of information, as well as making the best of limited options. The main effort was concentrated on a process of data conversions that forms the backbone of this project (see figure 1 on page 5).

Once working solutions were obtained, the testing phase was a simple iterative cycle of deployment, analysis, and refinement until a satisfactory point had been reached. Different techniques and parameters were also tested out and analysed to compare and contrast until a final deliverable worthy of the MEng FYP title had been produced.

4.1 Planning

Over the course of the project, priorities had changed somewhat after it became clearer what can and cannot be done. As mentioned in the Project Specification section of this report (section 2.2 on page 4), the original plan was to build a system that can convert point clouds — the likes of which would be obtained from LIDAR sensors — into geometry. This was completed very early on and as a result, issues on a higher level of the process were tackled. The goals of the project shifted to designing a complete system for constructing 3D geometry from real world objects and terrain. The project has since evolved to incorporate the design of *multiple* systems with the purpose of reaching this goal and evaluating and comparing them.

Originally, other techniques and methods — for example those from the field of Orthomosaics discussed above — were merely part of the contingency plan if the original goals were not met. Instead, these alternatives were incorporated into the project.

For the most part, the development process was split into two parts; simulation in software and implementation in hardware. As far as software is concerned, no paperwork of procurement was required. For hardware, available resources and options were explored early on.

Dave Checkley, an EECE lab technician was also spoken to extensively about available resources. It turns out that the 3rd year project that was carried out a couple of years ago doing something along the lines of mapping terrain in The Vale was only using cameras and primitive techniques. Dave did not have any components left from those projects that could be of any use in particular however.

Most other hardware resources, as are discussed in the Solutions section (section 4.3 on page 30), were either already available, cheaply procured, or recycled.

As is depicted in the Gantt charts on page 27, development on both hardware and software was simultaneous and independent. Once the hardware was complete, all that needed to be done for integration is to replace the dummy data from the internet and the virtual scene with data acquired from UAV flights to generate the models and simulations from.

Subsequently, when all was said and done, and everything had been tested and declared secure, flights were carried out to obtain the data this project was conceived for. This was the testing and verification phase of the project. This was where the aforementioned testing sites came into play.

Figure 13: Flight of the Hexacopter over a Half-buried Ship in the Purton Ships' Graveyard

4.1.1 Project Management

When it comes to procurement, fortunately no special items or facilities were needed. The hardware required is generic and can be ordered very quickly through the university or bought from university stores. So far the only devices bought were generic prototyping components such as batteries and stripboards; everything else was either freely available or "recycled". Software used to aid development is freely accessible and any other code can be written from scratch. As such no significant costs were incurred that warrant a strict budgeting plan.

All milestones for the project were fortunately kept, meaning that the timing of the project progressed as planned. Some general deadlines were set provisionally on a week to week basis. It was intended to complete all practical work by the end of the first term and

a half. This left the second half of the second term free to work on any unfinished work, enhancements, and documentation. Simple Gantt charts were created to help explain the scheduling:

ID	Task Name	Start	Finish	Duration
1	Research	14/10/2013	13/12/2013	9w
2	Procure Hardware	14/10/2013	25/10/2013	2w
3	Hardware Integration	28/10/2013	01/11/2013	1w
4	Parse LAS into Point Cloud	14/10/2013	18/10/2013	1w
5	Build Geometry from Point Cloud	21/10/2013	01/11/2013	2w
6	Optimise Geometry Construction	04/11/2013	08/11/2013	1w
7	Implement Real-time System	11/11/2013	22/11/2013	2w
8	Collect and Overlay Metadata	25/11/2013	06/12/2013	2w
9	Field Testing	25/11/2013	29/11/2013	1w
10	Refinement and Optimisation	02/12/2013	13/12/2013	2w
11	Write Report	14/10/2013	13/12/2013	9w

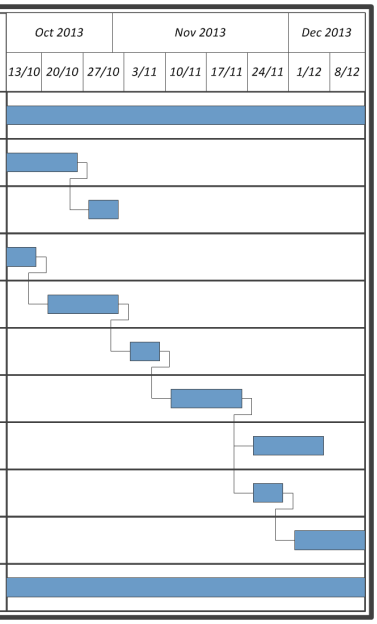
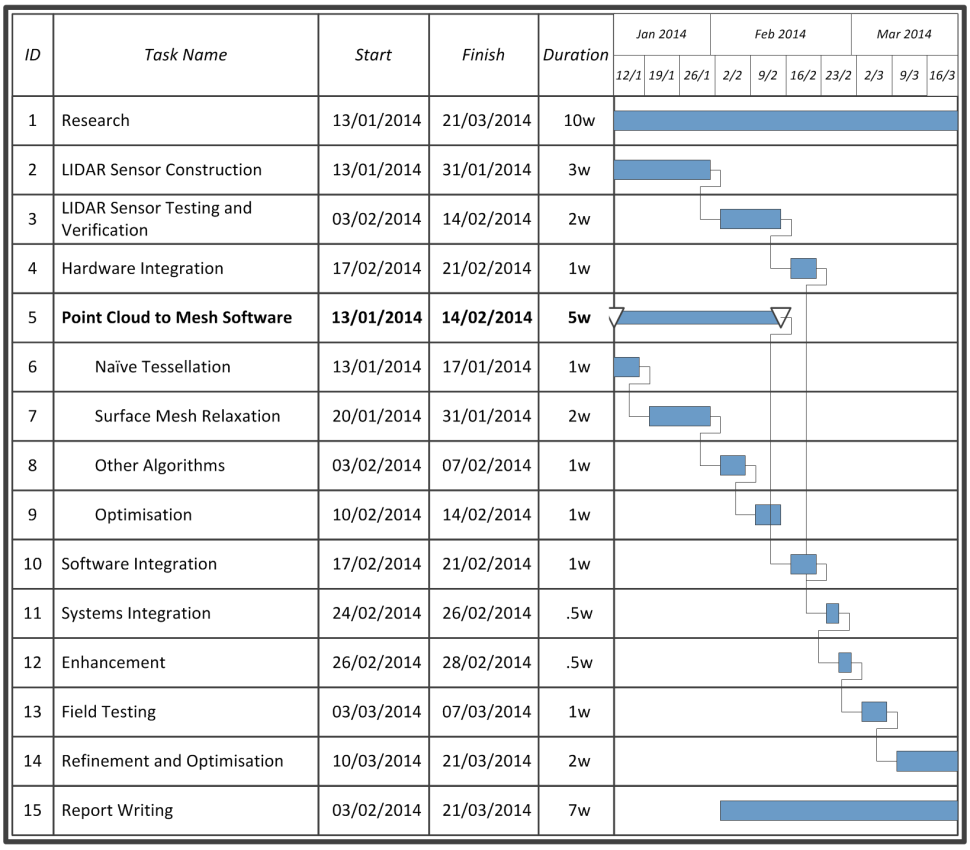


Figure 14: Term 1 Gantt Chart

As more information became available in the second term the plan was slightly refined to entail any additional work that needed to be done. This was also dependent on any feedback received from stakeholders such that the objectives can be reassessed and any necessary course correction can be performed.



ID	Task Name	Start	Finish	Duration
1	Research	13/01/2014	21/03/2014	10w
2	LIDAR Sensor Construction	13/01/2014	31/01/2014	3w
3	LIDAR Sensor Testing and Verification	03/02/2014	14/02/2014	2w
4	Hardware Integration	17/02/2014	21/02/2014	1w
5	**Point Cloud to Mesh Software**	**13/01/2014**	**14/02/2014**	**5w**
6	Naïve Tessellation	13/01/2014	17/01/2014	1w
7	Surface Mesh Relaxation	20/01/2014	31/01/2014	2w
8	Other Algorithms	03/02/2014	07/02/2014	1w
9	Optimisation	10/02/2014	14/02/2014	1w
10	Software Integration	17/02/2014	21/02/2014	1w
11	Systems Integration	24/02/2014	26/02/2014	.5w
12	Enhancement	26/02/2014	28/02/2014	.5w
13	Field Testing	03/03/2014	07/03/2014	1w
14	Refinement and Optimisation	10/03/2014	21/03/2014	2w
15	Report Writing	03/02/2014	21/03/2014	7w

Figure 15: Term 2 Gantt Chart

4.2 Health and Safety

Naturally a project like this has some obvious health and safety considerations that need to be addressed.

4.2.1 Risk assessment

From the beginning, no activity with any potential danger was planned to take place. However, before the testing stage, recommendations and risk and hazard statements were be made and discussed with Andy Dunn. According to safety guidelines and regulations set by the CAA on UK regulations for unmanned flight, since the UAV is not autonomous and as long as the craft is below 2.5kg, a lot of the restrictions do not apply.

Fortunately, a lot of the ways for dealing with potential risks were able to be replicated from a previous project which involved the flight of an autonomous quadcopter. These include simple actions such as setting a perimeter and clearing the area underneath which

the UAV will fly to make sure it cannot crash-land on anybody or anything. Another precaution can be surrounding the rotor blades with a Styrofoam frame or a similar light material to prevent them from cutting anybody; they can slice through flesh like butter.

The class of lasers used in most LIDAR sensors, as well as another standard class 1 laser used for this project, and at the distance they are used, are harmless to human eyes. These were however also included in the risk assessment. Since this is mostly a software project though, and flights are preformed by this project's supervisor and his research group, no other significant risks required attention other than the standard risks of using the labs and the equipment therein.

4.2.2 Ethics Review

Once again there was no intention to use human participants in experiments or evaluation trials. At no stage was it decided to do so and as such the ethics did not need to be gone through in more detail then.

4.3 Solutions

All hardware subsystems required were built from scratch from individual components. The reason for this is that several manufacturers of LIDAR related devices have been considered or corresponded with and most have returned excessive quotes. Some of these companies include *AutonomouStuff* and *Copter Craft* to name a few. Useful information has been exchanged via email, however some of the prices, without shipping and taxes, are for example:

- ERC-2KIT for £263
- Ibeo LUX for £12,207
- HDL-32E for £18,000+

Consequently, right off the bat, the use of commercial LIDAR sensors were determined unworkable. The ERC-2KIT would have been a bargain, except it was all but a glorified range finder as opposed to a proper scanning LIDAR sensor. As such, alternative solutions had to be developed.

4.3.1 Ultrasonic POC

Implementation

A relatively simple circuit was first prototyped and then soldered onto a strip-board as a proof of concept. This was done using "recycled" components from Dave Checkley's office after working through some extra paperwork. It behaves like a hand-held 3D scanner but rather than using laser, it uses an ultrasonic sensor that is easily replaced. Since it only has a range of up to 4 meters, and that at a pretty low resolution an accuracy, it can be described as "Virtual Spray Paint". A top-level circuit diagram was also created.

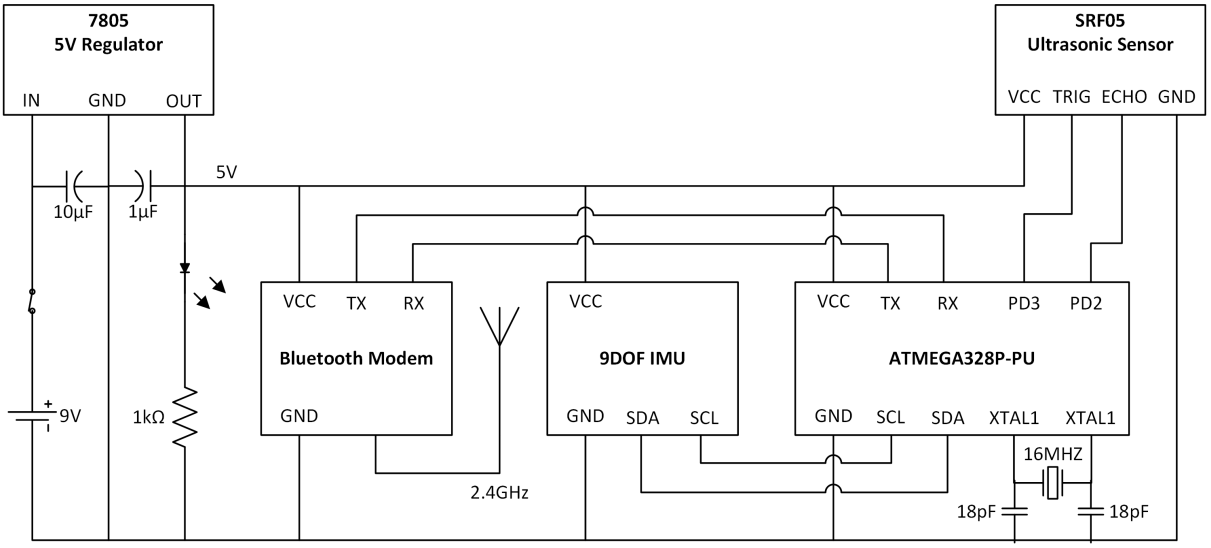


Figure 16: Simple Circuit Diagram for the Virtual Spray-paint Can

Essentially, by combining an ultrasonic sensor with a bluetooth modem, an IMU (that contains an accelerometer, gyroscope, and magnetometer), and a microcontroller, a system that proves the feasibility of this method was constructed. The IMU tracks orientation and the ultrasonic sensor measures distance to the nearest object at that orientation and that information is sent to a laptop for processing via bluetooth. In essence, LIDAR and Sonar/Acoustic Imaging systems are interchangeable; LIDAR is fundamentally the light equivalent of echolocation after all.

Code for a Unity3D project (see appendix A.3 on page 66 for code listings) was written that would take IMU readings and distance measures sent over a Bluetooth connection (non-stop and delimited with commas) and continuously update the virtual depiction of

the circuit as well as build a point cloud of scanned data.

Results and Verification

This worked relatively well however, due to the nature of ultrasound and the fact that the resolution was 8-bit (0m-4m = 0-255), did not produce realistic point clouds. Furthermore, as the IMU was also cheap relatively speaking, the scanner position had to be locked in place as it would otherwise unsalvageably drift from accelerometer inaccuracy and noise. The resolution of the data relative to the local y axis of the board at any give point was 63.75 points per metre. However, as the accuracy of the orientation was much more accurate, the angular resolution — and hence the horizontal resolution relative to the local coordinate system of the board — at any give point was 100 points per degree as the angles were read to two decimal places.

Experimentally, a heightmap of data scanned with this device was however limited by the 63.75 points per metre vertical resolution. Consequently point clouds would look very step-like and noise would be clearly visible.

In other words, this system proves that combining a rangefinding sensor with an IMU sensor is indeed a viable alternative to the unreasonably expensive LIDAR sensors. However it is not quite enough yet.

4.3.2 Laser Rangefinder

Instead the ultrasonic sensor can be replaced by a more long-range solution. Traditional rangefinders work on the same principle at LIDAR sensor and are hence expensive and difficult to integrate with bespoke solutions. Instead, an alternative system for rangefinding was developed that puts the weight on software instead of hardware.

Simulation

A scene was developed in Unity3D, this time one that would simulate a LIDAR-esque rangefinder sensor and output similar data for processing. The virtual scanner scene uses generic terrain and simple 3D models of ships that were partially buried in an attempt to recreate something that could pass for a ship graveyard like the one in Purton.

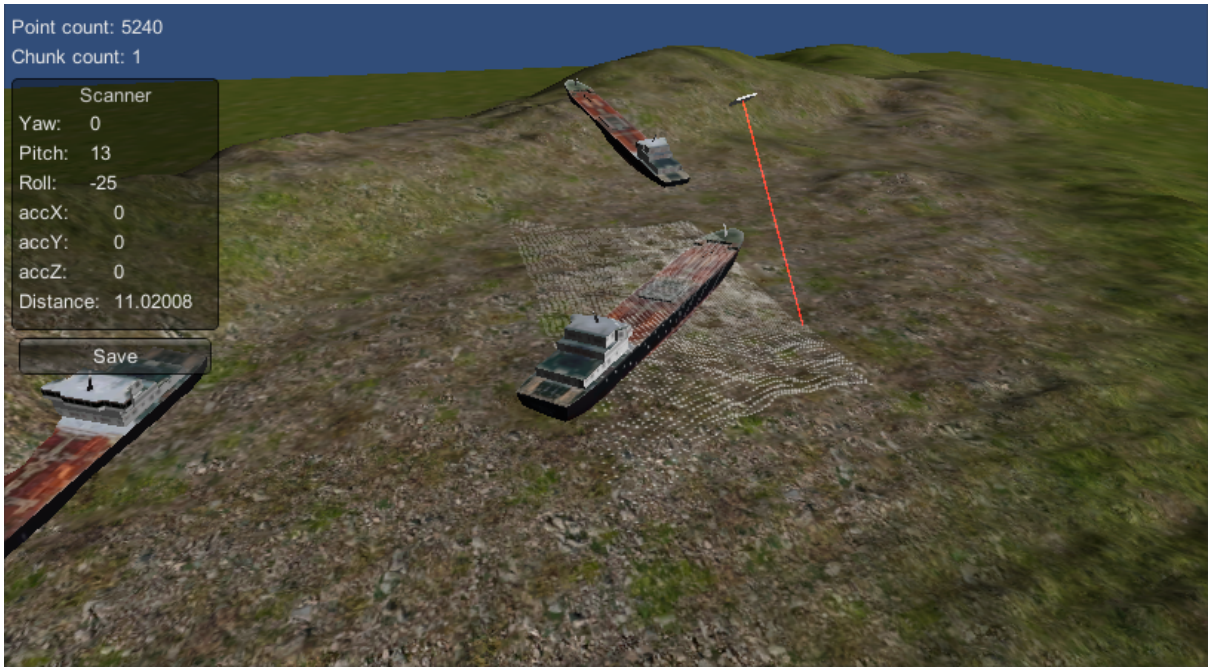


Figure 17: Virtual Scanner Unity3D Scene Scanning Test Ships

A virtual scanner was also added to the scene that sweeps over the terrain with a "laser" in a grid-like manner by rotating and using ray casting to simulate real-world range-finding (albeit to a much higher accuracy). Thus points are created at an angular resolution of 1° per point. These points are then written to a file in the CSV format; one point per line where floating point coordinates are separated by commas that can then

easily be parsed in the visualiser discussed in section 4.3.5 on page 50.

Implementation

From this, an alternative to traditional LIDAR was designed and prototyped. As previously mentioned, commercial LIDAR sensors and range-finders use expensive detectors that can measure the time it takes for light, literally the fastest thing in existence, to rebound off of an object. In order to overcome this cost barrier, a different solution was developed that relies on software rather than hardware.

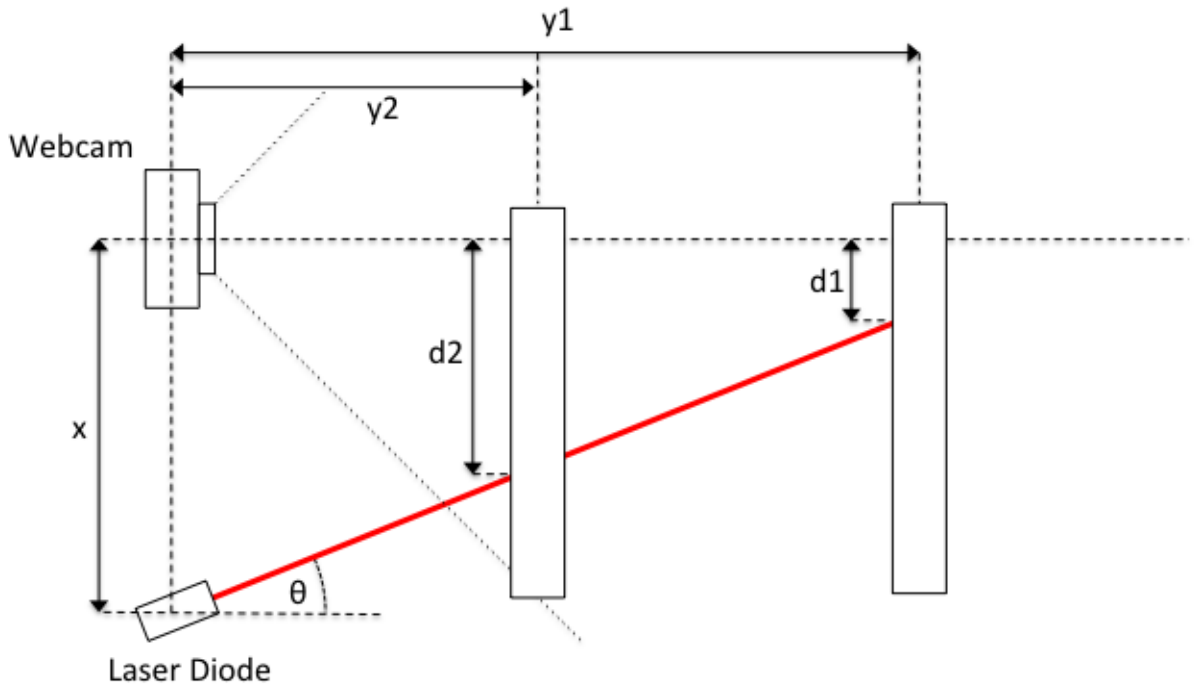


Figure 18: An Alternative to LIDAR

First you begin by attaching a normal class 1 laser pointer a fixed and know distance, x, from a cheap webcam with a reasonable resolution. In this case, a small Logitech webcam and a normal commercial laser diode were used. These components were in this case recycled and came at no cost. Depending on how far away an object is (y_1 and y_2), and knowing the angle of incidence θ — which incidentally can be adjusted in real time with the aid of a servo in order to focus the limited resolution of different average object distances for mode accurate models — the laser pointer will appear at a different height

on the webcam image (d_1 and d_2). Using this information and some simple geometry, the distance to the object being measured can be determined.

This technique is cheap but not without complications. The immediately most obvious question is how can you detect the position of a laser pointer on an image? This problem was solved using some C++ and by learning to use the powerful computer vision library OpenCV. The answer is quite simple and can be explained in a number of simple steps:

1. Get webcam frame
2. Convert to HSV (hue, saturation, value) colour model
3. For each pixel near the centre of the image:
   (a) Binary threshold registering only pixels with:
      i. A red hue (values around 0°/180° with a tolerance of 5°)
      ii. Minimum saturation
      iii. Maximum value
   (b) Count the number of pixels passing the threshold
   (c) Add their coordinates to a summation of all xs and ys respectively
4. Calculate the mean pixel coordinate for all passing pixels using the summation and pixel count
5. Output value; that's where the laser pointer is

For testing a GUI version of this program (rather than pure command-line interface) was created where all thresholds and tolerances have been made into adjustable sliders and the processed image and laser pointer location are displayed alongside the original webcam frame grab.

Once the behaviour was validated a final iteration of the program was developed (see appendix C.2 on page 96 for code listings) that output pixel values continuously that correspond to where the detected laser point is located.

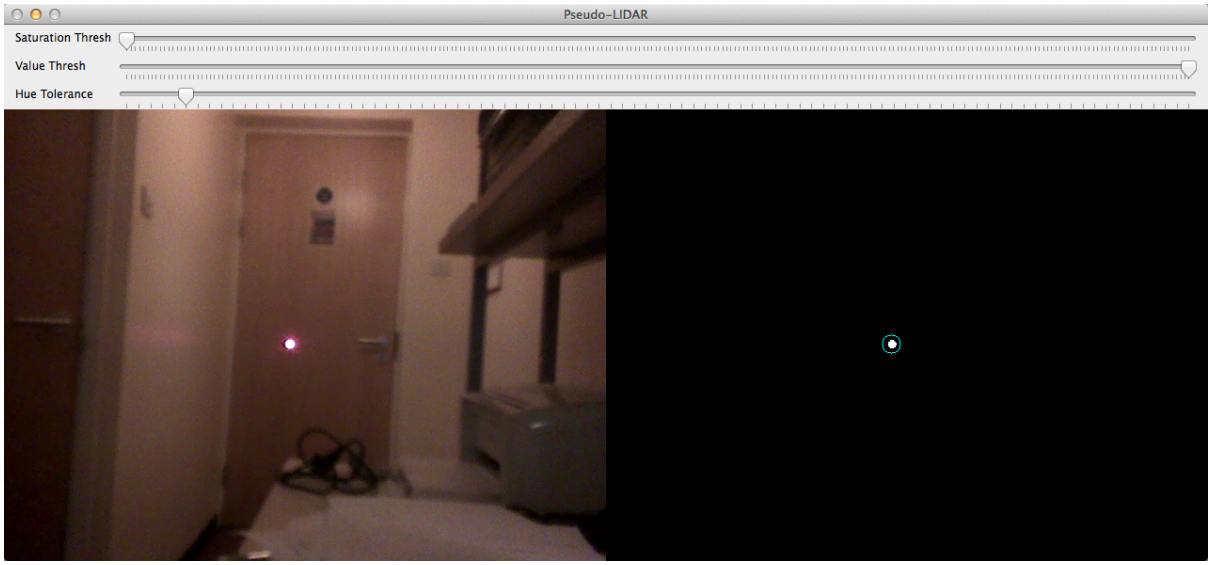


Figure 19: Screenshot of Laser Pointer Detector GUI Program

Using this, another problem was identified. Noise would often distort the calculated location and the program was unable to distinguish between a small laser point and other bright lights. Techniques were explored for remedying this issue such as flood-fill algorithms for removing blobs that do not fit a certain size or shape as well as computer vision techniques for filtering noise. Automatically adjusting lighting levels counteracted the issue however the problem still persists.

Another explored remedy involved flashing the laser at a frequency that matches exactly half of the frame rate of the camera using a transistor and a bit of extra C code. As a result of doing this, the camera would output two images per laser pulse; one with and one without the laser in it without the image changing much otherwise. With this data, it would theoretically become trivial to determine which pixels are part of the laser pointer and which are just noise or other sources of bright white or reddish light. All that would be needed to be done is to find out which pixels suddenly pass the threshold between images buy simply XORing them.

Practically, this was not the case. Not only was the frame rate now halved, but improvements to detection were still not ideal as noise became an issue especially when the camera was in motion. Moving laser-like lights would be misclassified as the laser point.

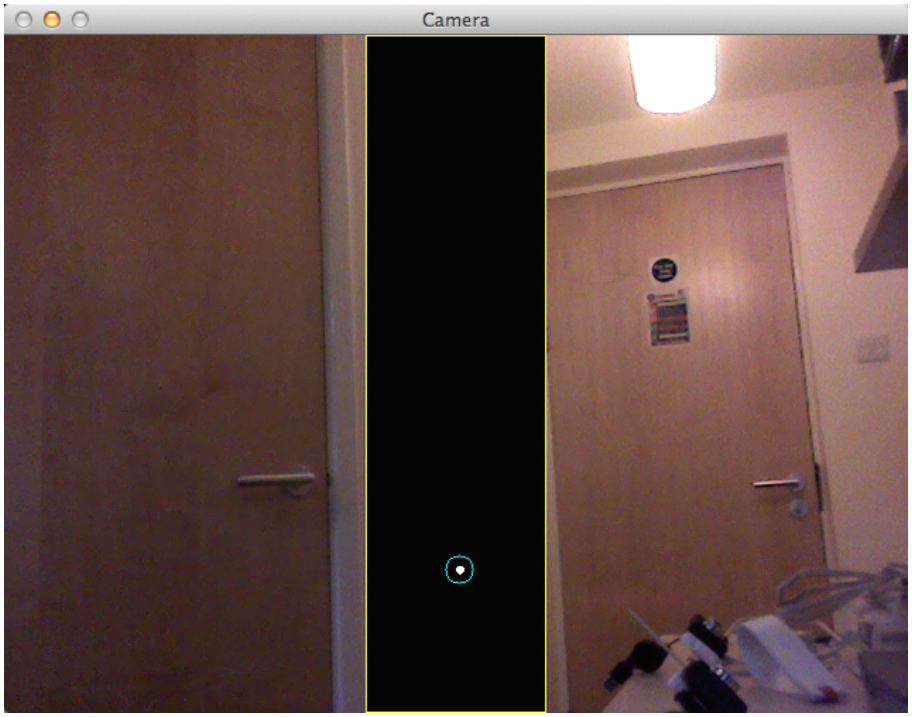


Figure 20: Screenshot of Laser Rangefinder GUI Program

Another potential issue is distance calibration. The hard way to do so would be to find out the field of view of the webcam and the optical characteristics of its lens and calculate its theoretical projection matrix as a virtual camera would have in computer graphics. Using that information, provided it has been deemed accurate through experimental means, the pixel coordinates of the laser pointer can be accurately mapped out to a distance which may not necessarily be a linear mapping depending on the nature of the lens.

An easier method is to use inverse problem solving. A one time calibration process specific to the camera being used would be carried out where distances are manually mapped out to pixel coordinates. The data gathered from this process can be used to plot a graph and from there, finding a best fit curve is trivial. This equation can then be hard-coded into the device so that it outputs accurate distances.

Additionally, a small mirror — or indeed the laser diode itself — can be mounted onto one or more servos or motors and be made to sweep any given area. Everything else can stay the same and a series of point cloud coordinates are hence determined in software. This was ultimately decided against however as it would increase the number

of misclassifications and was deemed not worth the extra data.

This system has the added advantage that textures for the mesh are automatically provided. These are the images that remain "unpolluted" by the laser pointer and can be sampled at regular intervals, stitched together, and mapped onto the constructed mesh.

The system initially ran on a *Raspberry Pi* – a credit card sized computer – that ran Arch Linux, and streamed images over IP on a local area network, for example over a secure WiFi connection, at a rate of about one image per second. These images can be called using the Raspberry Pi's IP address and a predetermined port number and viewed with anything that can handle the MJPEG format such as OpenCV or the internet browser FireFox.

Since this was determined too a *UDOO* board was used instead of a *Raspberry Pi*. A *UDOO* board is approximately as powerful as 4 *Raspberry Pi*s and an *Arduino Mega* microcontroller. It can also support a tiny HD camera and also runs Linux meaning it can run OpenCV. This enabled the processing to be done as fast as the frame rate of the camera attached (30 frames per second or 30 distance measurements per second) which were then sent to the laptop over WiFi.

Results and Verification

The system did indeed stream values correctly and could be coupled to the original IMU setup. To test this, with the aid of a tape measure, 50 samples were taken for every 10cm, averaged, then plotted (figure 21).

There are a number of things that are quite revealing about this graph. The most obvious is that detecting the laser point at long ranges is unreliable; the values begin to default to 0 as the system fails to detect a laser point at those positions. This can be random but is more likely at longer distances. This does not introduce noise, but instead has the effect that there are fewer points in the final point cloud for areas that are far away from the UAV.

The second observation is the shape of the curve; it begins to rise a small distance away from the camera then slowly plateaus. This is not so much due to the lens characteristics

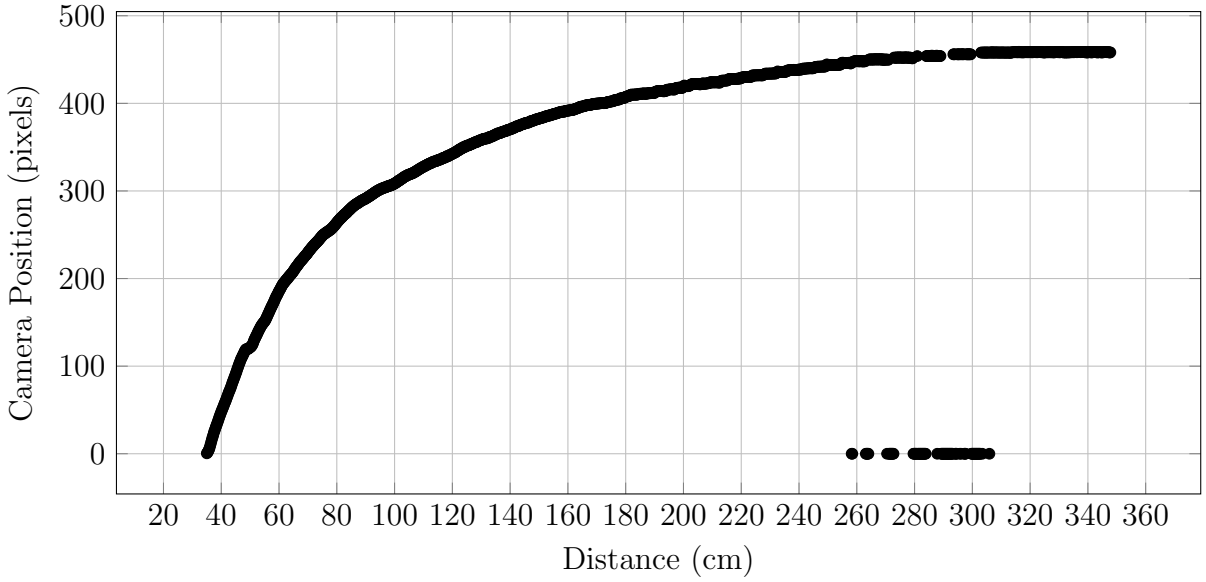


Figure 21: Plot of Laser Detection Position vs Distance

of the camera, but instead mainly due to the nature of the universe. Objects get smaller the further away you go and it becomes evident from figure 18 on page 34 that although it is easy to get clear, discreet distance measures for objects that are close, it is not for objects that are further away. For example, an object at 1m and an object at 2m would be easy to differentiate while an object at 100m and one at 200m might both register at 150m for example, which is one huge limitation of this method.

This is also not a question of changing the laser diode's angle of incidence (θ); the only effect that has is stretching out the same curve over a longer range. This problem would not occur with true LIDAR since the same curve would be relatively linear. The only remedy for this system would be to increase the distance between the camera and the laser diode (x), however doing this is impractical as space is limited on a UAV.

Clearly this was a problem. For this reason, yet other outside the box solutions were explored to help reach the goals of this project.

4.3.3 Stereoscopic Matching

One such solution is that of stereoscopic matching. As discussed in the literature review, this method is not novel. However, it has not ever been applied in the context of UAV mapping before, only in areas such as 3D entertainment (cinema/video games) and the like.

Simulation

To begin with, the idea was tried in simulations on scenes created with Unity3D. This was done with code written in Python and using OpenCV (see appendix for fully documented code listings). The following are three examples of disparity maps with SAD window sizes of 3x3, 9x9, and 15x15 respectively.

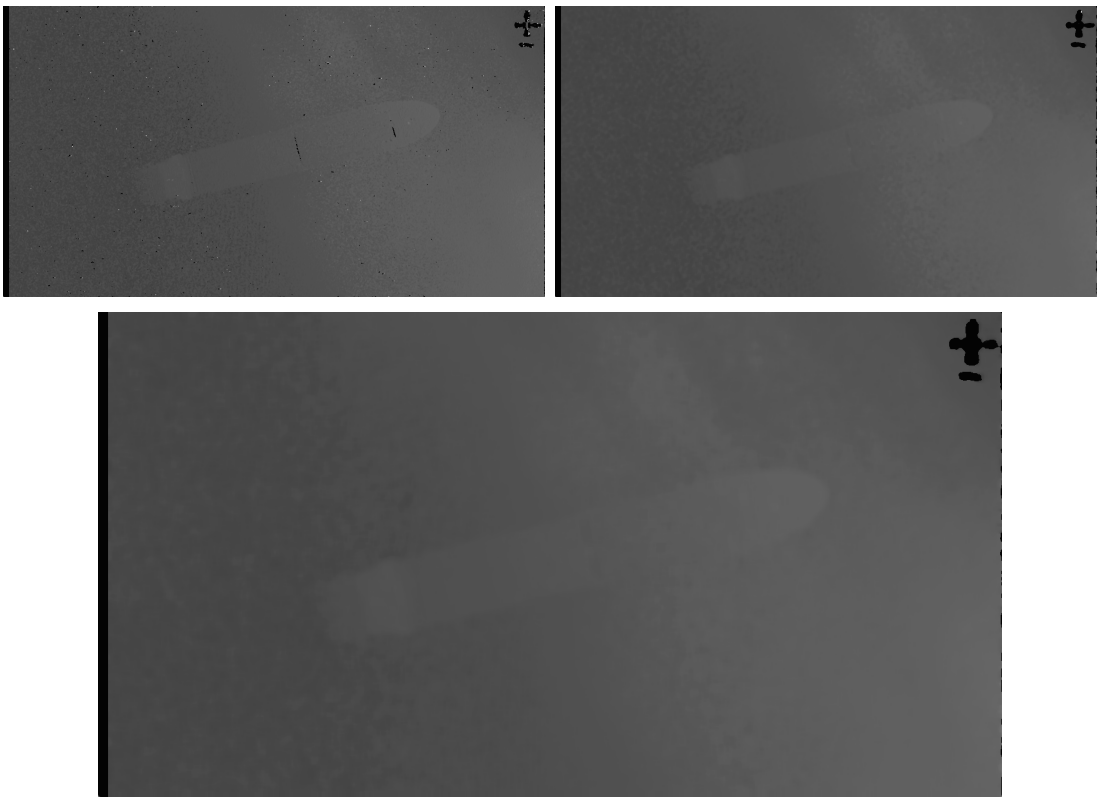


Figure 22: Disparity Maps of Simulated Scene

Notice the white, noisy spots in the first image. This is due to the smaller SAD window size (3x3 pixels). At the same time, the third image (15x15 pixels) has less defining features and looks blurred. To contrast the results of altering window sizes as

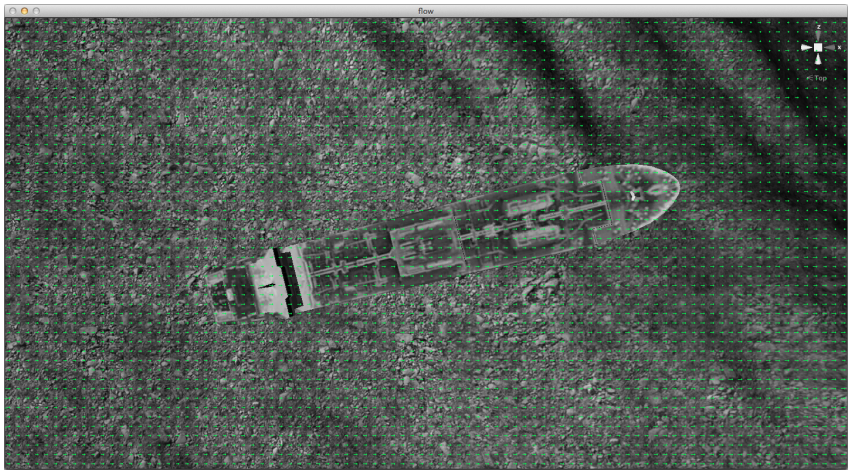


Figure 23: Farnebäck Optical Flow on Simulated Scene

well as to determine if it would be indeed useful to construct point clouds from disparity maps, another Unity3D scene was created for this purpose (see appendix C.4 on page 101 for code listings). Using the three disparity maps as inputs, the following are screenshots of the resulting point clouds.

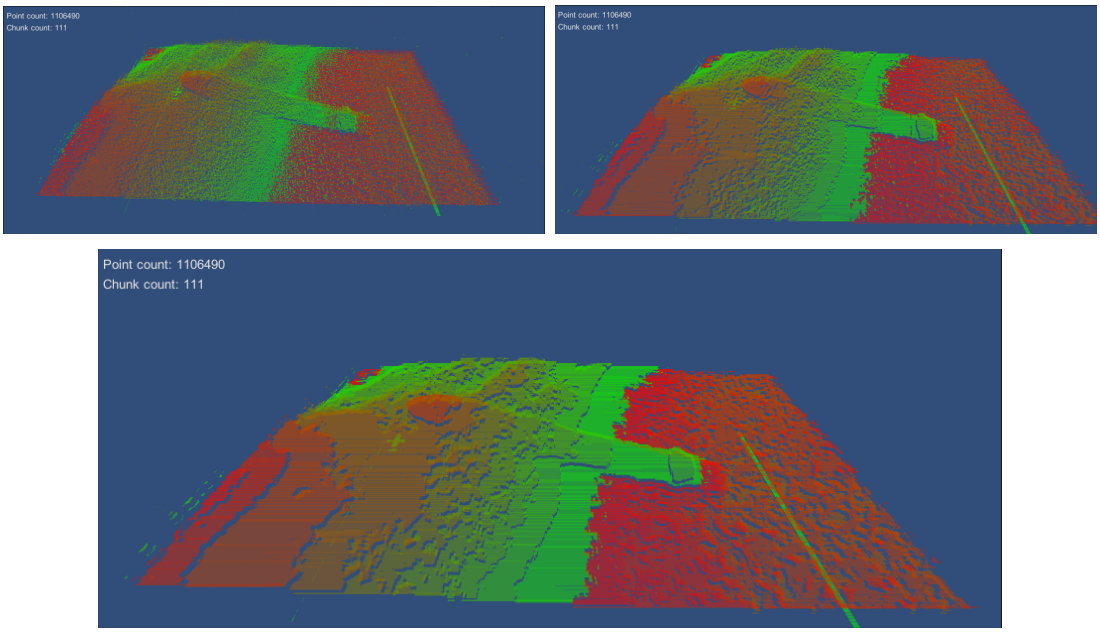


Figure 24: Point Clouds from Disparity Maps of Simulated Scene

Here it becomes clear that the noise in first point cloud (3x3 SAD window) – although giving the optical effect of smoothness – would be impossible to turn into geometry without noise filtration. At the same time, the third point cloud (15x15 SAD window) has the problem of low vertical resolution; points clamp to "steps" and the point cloud

would need to be smoothed through interpolation or averaging out point heights with their neighbouring points.

Also, completely black pixels would need to be ignored as they would otherwise become points that are at the lowest possible position and distort the final geometry (for example, the black bar on the side). This does however prove the viability of real world disparity maps being converted to point clouds as successful.

Implementation (Single Camera)

One hardware possibility that was theoretically possible yet had never been done before in an academic, industrial, or any other setting, was to use a *single* camera to capture the images used for stereo matching. The idea was that a single camera could substitute the use of two cameras by taking multiple frames over time and calculating their relative positions to each other, thus acting as if multiple cameras had recorded these frames.

After automatic calibration and rectification (by taking pictures of a printed chequerboard from different perspectives) to correct for lens distortion, a program was written that does just that (see appendix C.6 on page 106 for code listings) The process can be describes as follows:

1. Get initial camera frame
2. Repeat:
    (a) Store previous camera frame
    (b) Get new camera frame
    (c) Calculate mean optical flow through the Farnebäck algorithm (see code comments) to determine by how many pixels relative to the old frame the new frame has shifted along the x and z axes
    (d) Compute disparity map using the semi-global block matching algorithm
    (e) Output disparity map

Unlike the laser system, processing was done on a laptop instead of onboard the UAV when testing. This was because the program was extensively processor-intensive and some parts needed to run on the GPU. At the same time, doing this cuts down on the amount of data that needs to be transmitted from the UAV since only images would need to be sent instead of images (for colour) as well as disparity maps (for depth).

Implementing things this way turned out to be complex but ultimately successful. Test flights were carried out with the Parrot AR Drone; a commercially available quadcopter that was previously owned (i.e. not part of the budget). The different parts of the system can be depicted as follows.

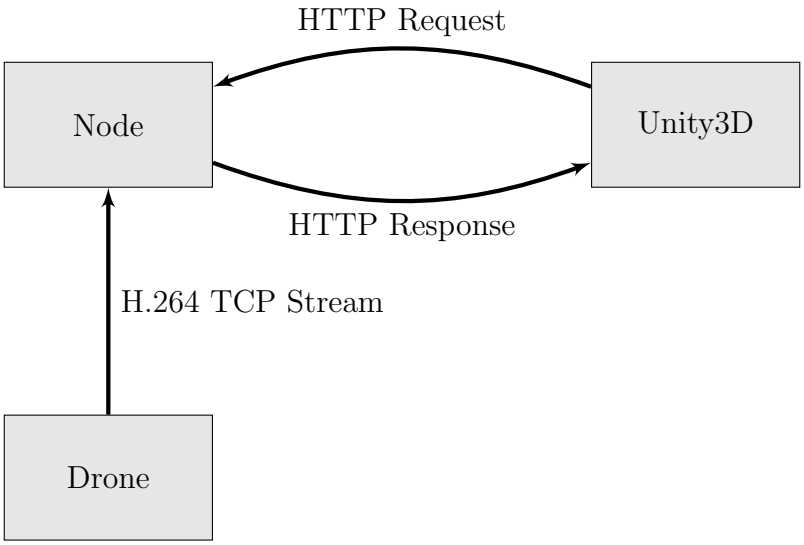


Figure 25: Top-level System Elements

Here, images from a downwards facing camera on the drone are streamed in H.264 format to the laptop over WiFi. The laptop is running a web server written in JavaScript using NodeJS (see appendix C.7 on page 108 for code listings). This server processes the video stream and converts it to PNG images. These images are then served over HTTP on the same network. The Unity3D scene polls these images repeatedly when the next one is needed and they can now also be applied as textures to geometry within the scene.

Results and Verification (Single Camera)

A test flight was conducted over the Vale and a point cloud was constructed. The results in practice were however nothing like in theory. Figure 26 shows a visualisation of the resulting point cloud normally and inverted.

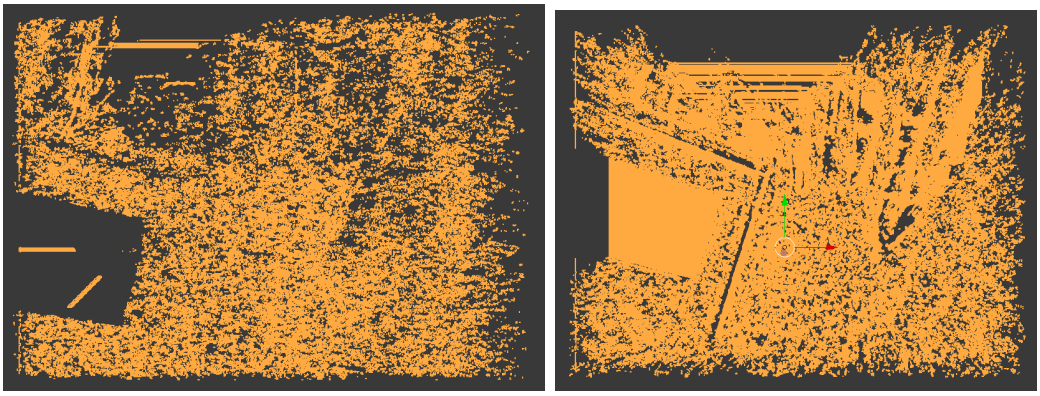


Figure 26: The Vale Point Cloud Visualised in Blender

It became evident that in practice stereo matching with a single camera is all but impossible especially with on a UAV that is outdoors and affected by wind. Buildings and paths are just about discernible in the point clouds, as they are generally flat uniform areas, however grass and trees at completely drowned out by noise. Consequently, a two-camera solution was investigated.

Implementation (Dual Camera)

Stereo matching is traditionally done with two cameras the distance between human eyes apart to emulate human sight. In this case however, the objects are being viewed from a high altitude so to improve performance, two cheap cameras with a very small field of view (no lens) were mounted on a rig 50cms apart; a reasonable diameter for a commercial UAV.

To test this, a program was written (see appendix C.5 on page 103 for code listings) that works on the same basis as the single camera solution however taking frames from video feeds from two cameras simultaneously as input.

Results and Verification (Dual Camera)

To test the system, disparity maps were created with it and evaluated. The parameters were tuned accordingly, however the results were still not at a usable standard. In figure 27, closer areas are indeed whiter than those further away, however one would not be able to infer the original environment from the disparity map alone wither by looking at it or by constructing geometry from it. There is simple too much inevitable noise.

In figure 28 the system collapses as the treetops, elongated and thin objects, are matched incorrectly with the ones adjacent to them and the resulting disparity map becomes pure high frequency noise.

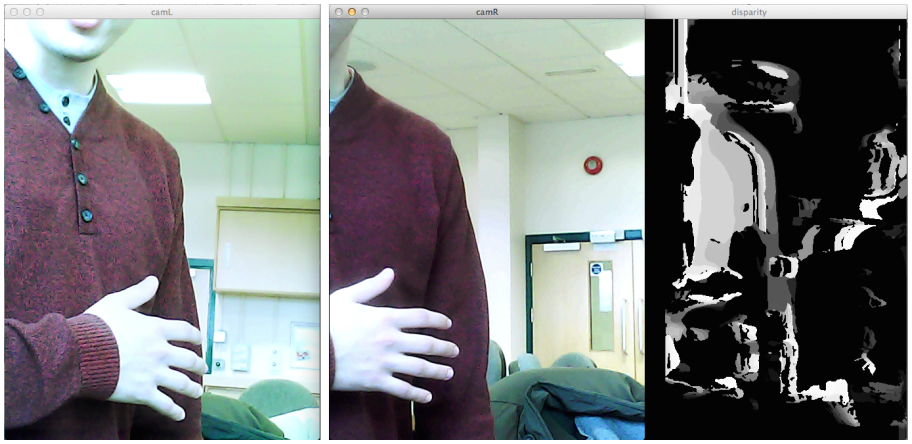


Figure 27: Example of Disparity Map of Indoor Environment

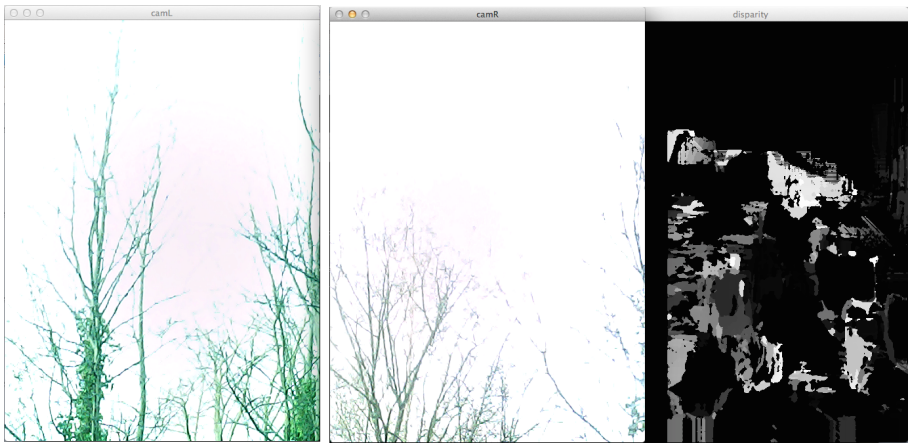


Figure 28: Example of Disparity Map of Outdoor Environment

The vertical resolution of the resulting point cloud is limited to a range of $[0 - 255]$ for any given pair of frames. Points at the 0 position correspond to darker pixels in

the disparity map (farther away) while points at the 255 position correspond to lighter pixels in the disparity map (closer). The physical range must be predefined, however the tradeoff becomes one of range vs resolution. In an effort to create high fidelity point clouds yet, a final possible method is investigated.

4.3.4 Structure from Motion

As hinted on earlier, one of the sites that Professor Bob Stone and his research group surveyed with a UAV was the wreck of the 17^{th} century naval ship, the H.M.S. Anne, and what remains of her hull.

Figure 29: Photos of the H.M.S. Anne Remains and Surveying thereof

Aerial footage was recorded of the wreck and photogrammetry methods applied to the footage with a number of tools. There were a number of steps that needed to be taken before a point cloud could be constructed from the footage however.

First and foremost, the camera of the DJI Phantom 2 quadcopter with which the footage was recorded has a near-fisheye lens and warped the image significantly. SfM algorithms would not work on that footage because it would be as if the terrain itself was warping as the camera panned.

Fortunately, Adobe Systems released a lens profile for that particular camera so removing lens distortion from the video footage was a trivial task (at the cost of losing some information at the edges of the frames). To do this a tool named "ProDRENALIN" was used.

Once the footage was ready, individual frames were exported from the footage into JPEG images. This was done using a command-line tool called "mplayer" and later a GUI tool called "MPEG Streamclip". Best results were obtained at a frame rate of 2 frames per second; at the speed of the quadcopter, a faster frame rate would not have made much of a difference in fidelity but would take exponentially longer to process. Figure 30 depicts samples of the frames that would be used as inputs for feature matching.

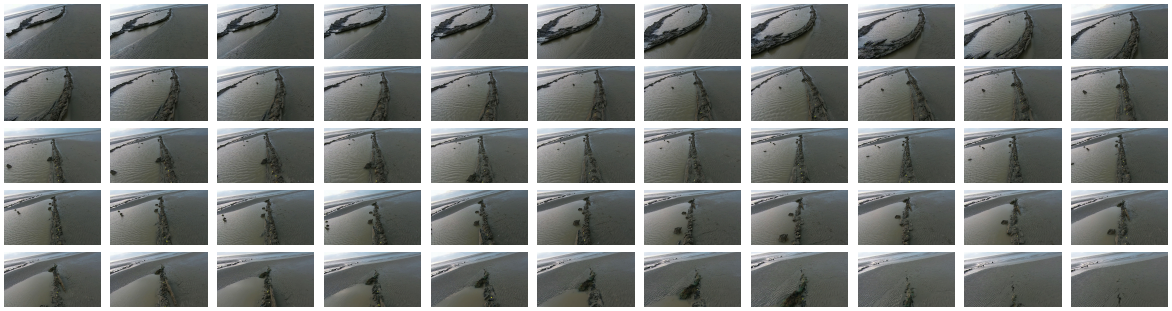


Figure 30: Processed Frames from a Segment of H.M.S. Anne Aerial Footage

These frames are then subjected to pairwise matching and sequence matched – with 4 neighbouring frames being matched at any given point – on the GPU using a tool called VisualSFM. This is done using the SIFT algorithm discussed in the literature review. A sparse 3D reconstruction is then run, likewise on the GPU, and camera positions as well as coloured feature positions are calculated and visualised.

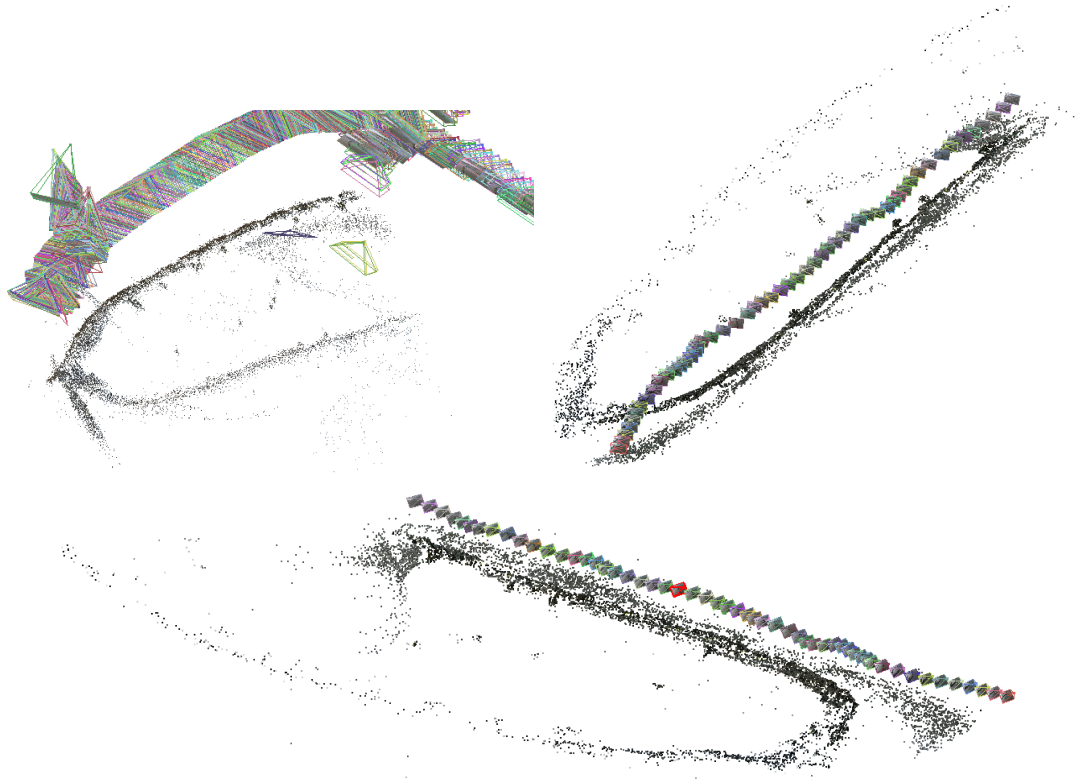


Figure 31: Point Clouds and Camera Locations through SfM

Figure 31 shows examples of the point clouds constructed and the paths that the quadcopter was calculated to have taken. The first is one where features were matched for 30 frames a second, and the other two are 2 frames per second for only the flight segment

along the left hull shown in figure 30. This reconstruction was particularly successful as the hull had yellow archaeological markers on it that were easy to automatically match across frames.

The point cloud built from this particular footage was not the most suitable for automatic mesh reconstruction as it was developed from scenery that included water that changes appearance constantly. It is however incredibly useful for modelling the scenery as it can provide a template to start from that is truer to reality than the human eye additionally corrupted through lens distortion. These point clouds were converted to a format that is trivial to process with tools such as MeshLab discussed in the literature review and opens doors in the 3D modelling domain.

4.3.5 Other Unity3D Scenes

The following are a number of other scenes created from scratch in Unity3D that were not previously discussed (such as the *Virtual 3D Scanner*, the *Physical 3D Scanner*, the *Disparity Visualiser*, and the *AR Drone Stream Processor* as well as the lesser miscellaneous ones).

LAS v1.3 Parser

Before hardware was procured, the software side of the project was focused on. The very fist task that was completed was writing a LAS file parser and visualiser for Unity3D. Recall that LAS is the most popular — and most complicated — file format for point clouds. It is no surprise that this is also the format most commercially available LIDAR sensors output. Doing this, as well as the visualiser created that is discussed later, means that the "Fallback and Rebuild Position" requirements in the project specifications were completed right off the bat.

The LAS file format specification is quite convoluted, however only the parts that were relevant to this project were implemented in C# for Unity3D (see appendix A.1 on page 57 for code listings). Only LAS version 1.3 was supported as this is the most prevalent version; if the file is not LAS v1.3 compatible, the program quits. To simulate the real-time streaming of data, a file stream is not closed and the point cloud is populated continuously while the file is still open until there are no more points left.

Although LAS files could be streamed over serial communication, it is more efficient to define a bespoke custom format in most cases, which is what was later done, not to mention that some information in the file header can only be obtained once all the data that is part of that file is collected making its design unsuitable for real-time applications. Additionally, the order of points in a LAS file can be arbitrary which means that converting its point cloud into a mesh will only work using certain algorithms that do not assume the vertices to be in a certain order. They cannot be sorted in real-time since you would need all the other points too to sort against.

The point cloud builder was adapted to use particles from geometric shapes to visualise

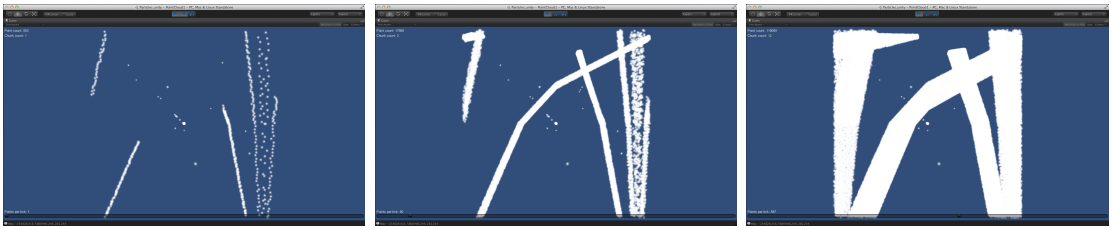


Figure 32: An Example Airport LAS File Streaming to the Unity3D Scene

the vertices. Doing this increased performance 100-fold for obvious reasons. Furthermore particles are divided into chunks of 1000 particles per chunk which boosted the number of particles that can be rendered by another order of magnitude. These optimisations have all been documented in more detail in the code appended to this report. As a direct consequence, millions of particles can be rendered simultaneously relatively lag-free on an average PC.

Tessellator and Visualiser

The visualiser is yet another scene. This scene reads in a CSV file and adds them all to the scene while at the same time dynamically colouring them such that the point highest on the Y-axis is the reddest and the point lowest on the Y-axis is the greenest. Once that is done, a mesh is constructed using an algorithm developed specifically for this project.

This method's name was coined as "Naïve Tessellation" by me in allusion to its simplicity. The idea is to build triangles procedurally on a grid of vertices. It assumes that your vertices are indeed in the form of a grid with a fixed number of rows and columns and in a specific order. These restrictions are outweighed by the fact that it is very fast and can take advantage of OpenGL geometric primitive types.

This method, discussed in part in the literature review under section 3.2.2 on page 17, operates under the assumption that the point cloud data is in the form of a grid of a specified width and depth. Each vertex is connect to its adjacent vertices as a normal grid mesh would and the resulting cells are tessellated as quads. An additional step is recalculating vertex normals for smooth shading which was also done (see appendix A.5 on page 72 for code listings) in order to make the lighting look realistic which is especially

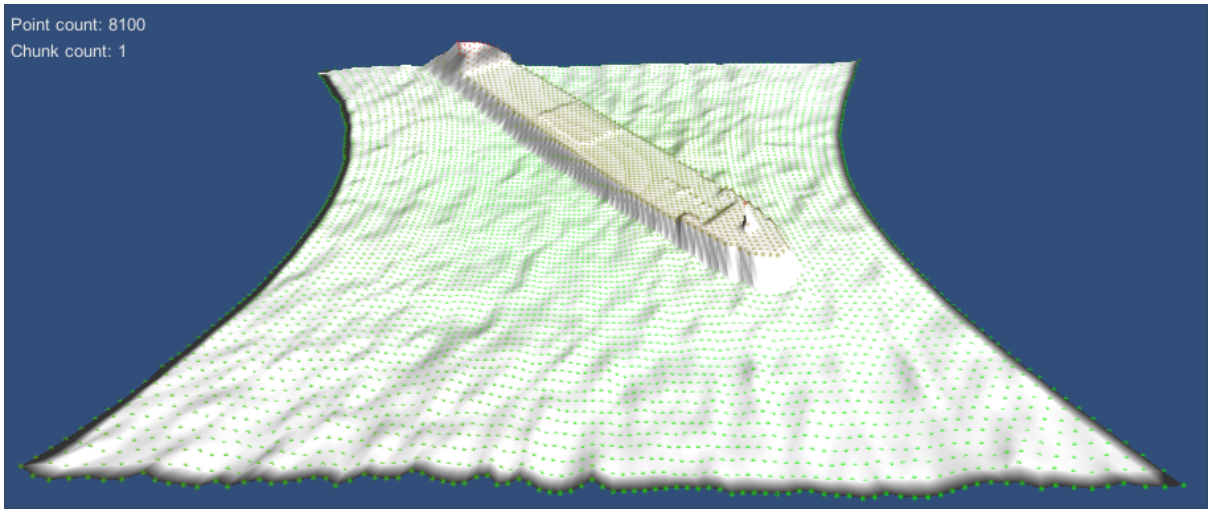


Figure 33: The Simulation Point Cloud Visualised through Naïve Tessellation

vital when your mesh is untextured in order for the features to be seen. A side effect of this is that the edges are dark due to the sudden falloff with no other vertices to compare against, as can be seen in figure 33. Parts of the code written for this scene was also used to visualising the disparity maps.

5 Conclusion

5.1 Summary and Critical Review

To recapitulate, a number of hardware and software systems were researched, designed, developed, simulated, implemented, tested, and evaluated. Software solutions depended largely on their hardware counterparts and as such could not be simply interchanged an compared. Hardware solutions came from different paradigms and likewise each hat their merits and downfalls. To summarise, figure 34 tabulates the most prominent of these.

	Pros	Cons
LIDAR	-High level of detail -Volume of data	-No inherent colour -Extremely expensive
Ultrasonic	-Very cheap -Accessible	-Range $< \approx$4m -No inherent colour
Laser + Camera	-Cheap -Weight on software	-Unreliable -Difficult to calibrate
Stereo (One Camera)	-Very cheap -No calibration	-Prone to noise -Prone to error
Stereo (Two Cameras)	-Cheap -Accurate	-Fixed range -Prone to noise
Structure from Motion	-Very accurate -Very cheap	-Laborious -Very slow

Figure 34: A Summary of Solution Pros and Cons

In terms of empirical resolution, SfM scores highest simply due to its nature. The point cloud density is unlimited. Meanwhile, ultrasonic and stereo are limited to ranges —fixed and variable respectively — with a maximum of 255 steps, as discussed earlier. The laser solution has a logarithmic resolution relative to range and as this cannot be accurately compared to the other solutions. Each solution has its limitations, from a perspective of fidelity however, SfM results are truest to reality.

5.2 Routes to Exploitation and Commercialisation

The outcomes of this project have already produced tangible benefits. One such benefit was discussed earlier on how the constructed point clouds could be used to aid the modelling of the H.M.S. Anne; a project currently under active development.

Other outcomes of this project, such as the tessellation program/algorithm developed for constructing 3D meshes from point clouds could also be used in commercial applications — especially involved with the use of Unity3D — for automatically building 3D models from real world data. Such functionality, even through 3rd party packages/plugins is currently non-existent.

Furthermore, specific parts of the work done on this project are hopefully to be published. The application of handled technology to the particular problem of object/terrain mapping is novel and unprecedented. Any industrial organisation with the intention of performing such mapping would undoubtedly want to cut their costs. Additionally, demand is evident; not too long ago Dr Karl Harrison — a lecturer in Forensic Archaeology — made contact on being interested in "the ability to check case anecdotes against virtual simulacra" to quote directly. This would be a perfect direct application of the outcomes of this project among many others!

Ultimately, this project continues to have some very exciting prospects. The goals and requirements were met as much as possible, yet there is still a lot that could be done in addition. Moreover, the academic content of this project made for a distinct learning experience.

References

Achtelik, M. et al. (2009). *Autonomous Navigation and Exploration of a Quadrotor Helicopter in GPS-denied Indoor Environments.*

Bade, R., J. Haase, and B. Preim (2006). *Comparison of Fundamental Mesh Smoothing Algorithms for Medical Surface Models.*

CVLAB (2012). *New Tsukuba Stereo Dataset.* URL: `http://www.cvlab.cs.tsukuba.ac.jp/` (visited on 12/08/2013).

Delaunay, B. (1934). "Sur la sphère vide, Izvestiya Akademii Nauk SSSR." In: *Otdelenie Matematicheskikh i Estestvennykh Nauk*, pp. 793–800.

Deyle, Travis (2009). *Ultra Low-Cost Laser Rangefinders Actualized by Neato Robotics.* URL: `http://www.hizook.com/blog/2009/12/20/ultra-low-cost-laser-rangefinders-actualized-neato-robotics` (visited on 12/06/2013).

Fischler, Martin A and Robert C Bolles (1981). "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Communications of the ACM* 24.6, pp. 381–395.

GPS.gov (2013). *GPS Accuracy.* URL: `http://www.gps.gov/systems/gps/performance/accuracy/` (visited on 10/05/2013).

Hartley, Richard and Andrew Zisserman (2000). *Multiple View Geometry in Computer Vision*, p. 312.

Karkanis, Tasso and A James Stewart (2001). "Curvature-dependent triangulation of implicit surfaces". In: *Computer Graphics and Applications, IEEE* 21.2, pp. 60–69.

Kondrashov, D. and M. Ghil (2006). "Spatio-temporal filling of missing points in geophysical data sets." In: *Nonlinear Processes in Geophysics*, pp. 151–159.

LiDAR-UK (2008). *How does LiDAR work?* URL: `http://www.fermentas.com/techinfo/nucleicacids/maplambda.htm` (visited on 10/04/2013).

Lorensen, W. E. and H. E. Cline (1987). *Marching Cubes: A high resolution 3D surface construction algorithm. Computer Graphics.*

Lowe, David G (1999). "Object recognition from local scale-invariant features". In: *Computer vision, 1999. The proceedings of the seventh IEEE international conference on.* Vol. 2. Ieee, pp. 1150–1157.

Lowe, David G (2004). "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2, pp. 91–110.

MacQueen, J., ed. (1967). *Some methods for classification and analysis of multivariate observations.* Vol. 5: *Weather Modification. Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability.* Berkeley. University of California Press, p. 14.

Meško, Matej and Štefan Toth (2013). "LASER SPOT DETECTION". In: *Journal of Information, Control and Management Systems* 11.1.

Phoenix Aerial Systems (2012). *Phoenix Aerial Systems.* URL: `http://www.phoenix-aerial.com/` (visited on 10/05/2013).

Scharstein, Daniel and Richard Szeliski (2002). "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms". In: *International journal of computer vision* 47.1-3, pp. 7–42.

Scheidegger, Carlos E., Shachar Fleishman, and Cláudio T. Silva (2005). "Triangulating Point Set Surfaces with Bounded Error". In: *Eurographics Symposium on Geometry Processing.*

Vallet, J. (2007). *GPS/IMU and LiDAR integration to aerial photogrammetry: Development and practical experiences with Helimap System®.*

Voronoi, Georgy (1908). "Nouvelles applications des paramètres continus à la théorie des formes quadratiques". In: *Journal für die Reine und Angewandte Mathematik* 133, pp. 97–178.

Appendices

A Referenced Unity3D Scripts

A.1 LASBuilder.cs

```
 using UnityEngine;
using System;
using System.IO;
using System.Collections.Generic;

public class LASBuilder : MonoBehaviour {

  public Transform pointCloudChunkPrefab;
  public bool allAtOnce = false;
  public int pointsPerTick = 1;

  private BinaryReader br;
  private ushort pdrLen;
  private uint prCount;
  double scaleX, scaleY, scaleZ;
  double offsetX, offsetY, offsetZ;

  private int pointID = 0, chunkCount = 0;
  private Vector3[] points;
  private ParticleSystem currentChunk;

  Vector3 velocity = new Vector3(0, 0, 0);
  float size = 100;
  float lifetime = float.PositiveInfinity;
  Color32 color = new Color32(255, 255, 255, 255);

  private List<Vector3> LoadPointCloud (string fileName) {
    // TODO: Do all of this in LASReader.

    TextAsset asset = Resources.Load(fileName) as TextAsset;
```

```
Stream s = new MemoryStream(asset.bytes);
br = new BinaryReader(s);

string fileSig = new String(br.ReadChars(4));

if (!fileSig.Equals("LASF")) {
  Debug.LogError("File is not of type 'LAS'.");
  return null;
}

// Skip some public header block information for now
for (int i = 0; i < 20; i++)
  br.ReadByte();

string version = br.ReadByte().ToString()+"."+br.ReadByte().ToString();
if (!version.Equals("1.3")) {
  Debug.LogWarning("LAS version "+version+" is not yet supported;
following 1.3 specs.");
  //return;
}

Debug.Log("System Identifier: " + new String(br.ReadChars(32)));
Debug.Log("Generating Software: " + new String(br.ReadChars(32)));
Debug.Log("File Creation Day of Year: " + br.ReadUInt16());
Debug.Log("File Creation Year: " + br.ReadUInt16());

ushort headerSize = br.ReadUInt16();
Debug.Log("Header Size: "+headerSize);

uint offsetToPointData = br.ReadUInt32();
Debug.Log("Offset to Point Data: "+offsetToPointData);

uint vlrCount = br.ReadUInt32();

// TODO: Support all formats, or at least the one specific to the
LIDAR sensor we get.
string pdfID = br.ReadByte().ToString();
```

```
Debug.Log("Point Data Format ID: " + pdfID);

pdrLen = br.ReadUInt16();
Debug.Log("Point Data Record Length: "+pdrLen);

prCount = br.ReadUInt32();
Debug.Log("Number of Point Records: "+prCount);

// Skip some public header block information for now
for (int i = 0; i < 20; i++)
  br.ReadByte();

scaleX = br.ReadDouble();
scaleY = br.ReadDouble();
scaleZ = br.ReadDouble();

offsetX = br.ReadDouble();
offsetY = br.ReadDouble()*0;
offsetZ = br.ReadDouble();

double maxX = br.ReadDouble();
double minX = br.ReadDouble();
double maxY = br.ReadDouble();
double minY = br.ReadDouble();
double maxZ = br.ReadDouble();
double minZ = br.ReadDouble();

Debug.Log("Scale: "+scaleX+", "+scaleY+", "+scaleZ);
Debug.Log("Offset: "+offsetX+", "+offsetY+", "+offsetZ);
Debug.Log("Min: "+minX+", "+minY+", "+minZ);
Debug.Log("Max: "+maxX+", "+maxY+", "+maxZ);

// Skip some public header block information for now
for (int i = 220; i < headerSize; i++)
  br.ReadByte();

// Skip variable length record header for now
```

```
    for (int i = 0; i < 54; i++)
      br.ReadByte();

    // Skip straight to point data records
    //for (int i = 0; i < offsetToPointData-220; i++)
    //  br.ReadByte();

    return null;
}

void OnGUI () {
    GUI.Label (new Rect (5, 5, 1000, 20), "Point count: "+pointID);
    GUI.Label (new Rect (5, 25, 1000, 20), "Chunk count: "+chunkCount);
    GUI.Label (new Rect (5, Screen.height-45, 1000, 20), "Points per tick:
  "+pointsPerTick);
      pointsPerTick = (int) GUI.HorizontalScrollbar (new Rect (5,
   Screen.height-25, Screen.width-10, 20), pointsPerTick, 1, 0, 1000);
}

void Start () {
    LoadPointCloud("test1");

    if (!allAtOnce)
      return;

    foreach (Vector3 point in points) {
      currentChunk.Emit(point, velocity, size, lifetime, color);
    }
}

// Update is called once per frame
void Update () {
    if (Input.GetKeyDown(KeyCode.Escape))
      Application.LoadLevel(0);

    if (allAtOnce || pointID >= prCount)
      return;
```

```
    int until = pointID + pointsPerTick;
    float x, y, z;
    //Debug.Log (pointID+", "+prCount);
    while (pointID < until &&  pointID < prCount) {
      x = (float)((float)br.ReadInt32() * scaleX + offsetX)/1000;
      y = (float)((float)br.ReadInt32() * scaleY + offsetY)/1000;
      z = (float)((float)br.ReadInt32() * scaleZ + offsetZ)/1000;

      // Skip other information for now
      for (int j = 0; j < pdrLen-12; j++)
        br.ReadByte();

      if (pointID++%10000 == 0) {
        currentChunk = ((Transform)
    Instantiate(pointCloudChunkPrefab)).particleSystem;
        chunkCount++;
      }

      currentChunk.Emit(new Vector3(x, y, z), velocity, size, lifetime,
    color);
    }
  }
}
```

A.2 DisparityLoader.cs

```
 using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.IO;

/**
 * Note: This will load an entire file all at once at start.
 */
public class DisparityLoader : MonoBehaviour {

  public Transform pointCloudChunkPrefab;
  public Texture2D disparityMap;

  private ParticleSystem currentChunk;
  private int pointCount = 0, chunkCount = 0;
  private List<Vector3> points = new List<Vector3>();

  private Vector3 velocity = new Vector3(0, 0, 0);
  private float size = 0.2F;
  private float lifetime = float.PositiveInfinity;
  private Color32 color = new Color32(255, 255, 255, 255);

  void SpawnParticle (Vector3 position) {
    points.Add(position);

    if (pointCount++%10000 == 0) {
      currentChunk = ((Transform)
   Instantiate(pointCloudChunkPrefab)).particleSystem;
      chunkCount++;
    }

    // TODO: Dynamically compute color range based on min and max height.
    byte col = (byte)((position.y)*25.5F);
    color = new Color32(col, (byte)(255-col), 0, 255);
    currentChunk.Emit(position, velocity, size, lifetime, color);
```

```
}

void OnGUI () {
  GUI.Label(new Rect (5, 5, 150, 25), "Point count: "+pointCount);
  GUI.Label(new Rect (5, 25, 150, 25), "Chunk count: "+chunkCount);
}

void Start () {

  int width = disparityMap.width;
  int height = disparityMap.height;

  Color pixel;

  for (int row = 0; row < width; row++) {
    for (int col = 0; col < height; col++) {
      pixel = disparityMap.GetPixel(row, col);
      if (pixel.a == 0.0F)
        continue;
      SpawnParticle(new Vector3(((float)col)/10.0F, 100.0F*pixel.r,
 ((float)row)/10.0F));
    }
  }

  Mesh mesh = new Mesh();
  GetComponent<MeshFilter>().mesh = mesh;
      mesh.vertices = points.ToArray();
  Vector3[] normals = new Vector3[points.Count];
  int pointID = 0;
  Vector3 normal, fwd, back, left, right, centre;
  for (int row = 0; row < width-1; row++) {
    for (int col = 0; col < height-1; col++) {
      if (row == 0 && col == 0) continue;
      pointID = (row*width)+col;
      centre = mesh.vertices[pointID];
      fwd = pointID-width >= 0 ? mesh.vertices[pointID-width] : centre;
```

```
        back = pointID+width <= width*height-1 ?
    mesh.vertices[pointID+width] : centre;
        left = pointID-1 >= 0 ? mesh.vertices[pointID-1] : centre;
        right = pointID+1 >= 0 ? mesh.vertices[pointID+1] : centre;
        normal = Vector3.Cross(back - fwd, left - right);
        normal.Normalize();
        normals[pointID] = normal;
      }
    }
    mesh.normals = normals;
    Vector2[] uv = new Vector2[points.Count];
    for (int i = 0; i < uv.Length-4; i+=4) {
      uv[i] = new Vector2(0, 0);
      uv[i+1] = new Vector2(1, 0);
      uv[i+2] = new Vector2(0, 1);
      uv[i+3] = new Vector2(1, 1);
    }
    mesh.uv = uv;
    int[] tris = new int[points.Count*6];
    pointID = 0;
    int triID = 0;
    for (int row = 0; row < height-1; row++) {
      for (int col = 0; col < width-1; col++) {
        pointID = (row*width)+col;
        triID = (row*(width*6))+(col*6);
        if (row != 0 || col != 0) {
          tris[triID] = pointID+width;
          tris[triID+1] = pointID;
          tris[triID+2] = pointID+1;
        }
        tris[triID+3] = pointID+width+1;
        tris[triID+4] = pointID+width;
        tris[triID+5] = pointID+1;
      }
    }
    mesh.triangles = tris;
  }
```

```
  void Update () {
    if (Input.GetKeyDown(KeyCode.Escape))
      Application.LoadLevel(0);
  }
}
```

A.3 ScannerPhysical.cs

```
 using UnityEngine;
using System.Collections;
using System.IO.Ports;

public class ScannerPhysical : MonoBehaviour {

  public Transform pointCloudChunkPrefab;

  private ParticleSystem currentChunk;
  private int pointCount = 0, chunkCount = 0;

  private Vector3 velocity = new Vector3(0, 0, 0);
  private float size = 0.1F;
  private float lifetime = float.PositiveInfinity;
  private Color32 color = new Color32(255, 255, 255, 255);

  private SerialPort stream;
  private bool connected = false;
  float yaw = 0.0F, pitch = 0.0F, roll = 0.0F;
  float dX = 0.0F, dY = 0.0F, dZ = 0.0F;
  int dist = 0;

  void SpawnParticle (Vector3 position) {
    if (pointCount++%10000 == 0) {
      currentChunk = ((Transform)
   Instantiate(pointCloudChunkPrefab)).particleSystem;
      chunkCount++;
    }

    currentChunk.Emit(position, velocity, size, lifetime, color);
  }

  void OnGUI () {
    GUI.Label(new Rect (5, 5, 150, 25), "Point count: "+pointCount);
    GUI.Label(new Rect (5, 25, 150, 25), "Chunk count: "+chunkCount);
```

```
  GUI.Box(new Rect(5, 50, 150, 240), "Scanner");

  GUI.Label(new Rect (10, 70, 150, 25), "Yaw:    \t"+yaw);
  GUI.Label(new Rect (10, 90, 150, 25), "Pitch: \t"+pitch);
  GUI.Label(new Rect (10, 110, 150, 25), "Roll:   \t"+roll);
  GUI.Label(new Rect (10, 130, 150, 25), "accX:\t\t"+dX);
  GUI.Label(new Rect (10, 150, 150, 25), "accY:\t\t"+dY);
  GUI.Label(new Rect (10, 170, 150, 25), "accZ:\t\t"+dZ);
  GUI.Label(new Rect (10, 190, 150, 25), "Distance: \t"+dist);

  if (connected) {
    GUI.Label(new Rect (10, 230, 150, 25), "Scanner connected.");
    if(GUI.Button(new Rect (15, 255, 130, 25), "Disconnect")) {
      stream.Close();
      connected = false;
    }
  } else {
    GUI.Label(new Rect (10, 230, 150, 25), "Scanner disconnected.");
    if(GUI.Button(new Rect (15, 255, 130, 25), "Connect")) {
      stream.Open();
      connected = true;
    }
  }
}

void Start () {
  //stream = new SerialPort("/dev/tty.usbmodem1421", 57600);
  stream = new SerialPort("/dev/tty.Base-SPP", 57600);
}

void Update () {
  if (Input.GetKeyDown(KeyCode.Escape))
    Application.LoadLevel(0);

  if (connected) {
    string line = "";
```

```
      while (stream.BytesToRead > 0)
        line = stream.ReadLine();

      if (line.Equals("")) return;

      // Note: Format yaw,pitch,roll,distance.
      string[] data = line.Split(',');
      if (data.Length >= 7) {
        try {
          yaw = float.Parse(data[0]);
          pitch = float.Parse(data[1]);
          roll = float.Parse(data[2]);
          dX = float.Parse(data[3]);
          dY = float.Parse(data[4]);
          dZ = float.Parse(data[5]);
          dist = int.Parse(data[6]);
        } catch (System.Exception ex) {}
      }

      //transform.rotation = Quaternion.Slerp(transform.rotation,
    Quaternion.Euler(pitch,yaw,roll), Time.deltaTime*10);
      transform.rotation = Quaternion.Euler(pitch,yaw,roll);

      transform.Translate(new Vector3(0.00F*dX, 0.00F*dY, 0.00F*dZ));
      transform.position = new Vector3(transform.position.x, 0,
    transform.position.z);

      SpawnParticle(transform.TransformPoint(0, -dist, 0));

      stream.BaseStream.Flush();
    }
  }

  void OnApplicationQuit () {
    stream.Close();
  }
}
```

A.4 ScannerVirtual.cs

```
 using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.IO;

public class ScannerVirtual : MonoBehaviour {

  public Transform pointCloudChunkPrefab;
  public float scanIntervalS = 0.1F;
  public int pointsPerTick = 10;

  private ParticleSystem currentChunk;
  private int pointCount = 0, chunkCount = 0;
  private List<Vector3> points = new List<Vector3>();

  private RaycastHit hit;
  private float yaw = 0.0F, pitch = -45.0F, roll = -45.0F;
  private float dX = 0.0F, dY = 0.0F, dZ = 0.0F;
  private float dist = 0;

  private Vector3 velocity = new Vector3(0, 0, 0);
  private float size = 0.2F;
  private float lifetime = float.PositiveInfinity;
  private Color32 color = new Color32(255, 255, 255, 255);

  void SpawnParticle (Vector3 position) {
    points.Add(position);

    if (pointCount++%10000 == 0) {
      currentChunk = ((Transform)
   Instantiate(pointCloudChunkPrefab)).particleSystem;
      chunkCount++;
    }

    currentChunk.Emit(position, velocity, size, lifetime, color);
```

```
}

void OnGUI () {
  GUI.Label(new Rect (5, 5, 150, 25), "Point count: "+pointCount);
  GUI.Label(new Rect (5, 25, 150, 25), "Chunk count: "+chunkCount);

  GUI.Box(new Rect(5, 50, 140, 170), "Scanner");

  GUI.Label(new Rect (10, 70, 150, 25), "Yaw:    \t"+yaw);
  GUI.Label(new Rect (10, 90, 150, 25), "Pitch: \t"+pitch);
  GUI.Label(new Rect (10, 110, 150, 25), "Roll:   \t"+roll);
  GUI.Label(new Rect (10, 130, 150, 25), "accX:\t\t"+dX);
  GUI.Label(new Rect (10, 150, 150, 25), "accY:\t\t"+dY);
  GUI.Label(new Rect (10, 170, 150, 25), "accZ:\t\t"+dZ);
  GUI.Label(new Rect (10, 190, 150, 25), "Distance: \t"+dist);

  if(GUI.Button(new Rect (10, 225, 130, 25), "Save")) {
    StreamWriter sw =
 File.CreateText(Application.dataPath+"/Resources/points.txt");
    foreach (Vector3 point in points)
      sw.WriteLine("{0},{1},{2}", point.x, point.y, point.z);
    sw.Close();
  }
}

void Start () {
}

private float scanTimer = 0;

void Update () {
  if (Input.GetKeyDown(KeyCode.Escape))
    Application.LoadLevel(0);

  if (pitch >= 45)
    return;
```

```
    scanTimer -= Time.deltaTime;
    if (scanTimer > 0)
      return;

    scanTimer += scanIntervalS;

    for (int i = 0; i < pointsPerTick; i++) {
      if (Physics.Raycast(transform.position,
    transform.TransformDirection(Vector3.down), out hit)) {
        dist = hit.distance;
        SpawnParticle(transform.TransformPoint(0, -dist*10, 0));
      }

      roll++;
      if (roll >= 45) {
        roll = -45.0F;
        pitch++;
      }
      transform.rotation = Quaternion.Euler(pitch, yaw, roll);
    }
  }
}
```

A.5 CustomLoader.cs

```
 using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.IO;

/**
 * Note: This will load an entire file all at once at start.
 */
public class CustomLoader : MonoBehaviour {

  public Transform pointCloudChunkPrefab;

  private ParticleSystem currentChunk;
  private int pointCount = 0, chunkCount = 0;
  private List<Vector3> points = new List<Vector3>();

  private Vector3 velocity = new Vector3(0, 0, 0);
  private float size = 0.2F;
  private float lifetime = float.PositiveInfinity;
  private Color32 color = new Color32(255, 255, 255, 255);

  void SpawnParticle (Vector3 position) {
    points.Add(position);

    if (pointCount++%10000 == 0) {
      currentChunk = ((Transform)
   Instantiate(pointCloudChunkPrefab)).particleSystem;
      chunkCount++;
    }

    // TODO: Dynamically compute color range based on min and max height.
    byte col = (byte)((position.y+10.0F)*100.0F);
    color = new Color32(col, (byte)(255-col), 0, 255);
    currentChunk.Emit(position, velocity, size, lifetime, color);
  }
```

```
void OnGUI () {
  GUI.Label(new Rect (5, 5, 150, 25), "Point count: "+pointCount);
  GUI.Label(new Rect (5, 25, 150, 25), "Chunk count: "+chunkCount);
}

void Start () {
  string file = Application.dataPath+"/Resources/points.txt";
  if (!File.Exists(file)) {
    Debug.LogError("File not found: "+file +".");
    return;
  }

  StreamReader sr = File.OpenText(file);
  string line = sr.ReadLine();
  string[] data;
  while (line != null) {
    data = line.Split(',');

    SpawnParticle(new Vector3(float.Parse(data[0]),
 float.Parse(data[1]), float.Parse(data[2])));
    line = sr.ReadLine();
  }
  Mesh mesh = new Mesh();
  GetComponent<MeshFilter>().mesh = mesh;
      mesh.vertices = points.ToArray();
  Vector3[] normals = new Vector3[points.Count];
  int pointID = 0;
  Vector3 normal, fwd, back, left, right, centre;
  for (int row = 0; row < 89; row++) {
    for (int col = 0; col < 89; col++) {
      if (row == 0 && col == 0) continue;
      pointID = (row*90)+col;
      centre = mesh.vertices[pointID];
      fwd = pointID-90 >= 0 ? mesh.vertices[pointID-90] : centre;
      back = pointID+90 <= 90*90-1 ? mesh.vertices[pointID+90] : centre;
      left = pointID-1 >= 0 ? mesh.vertices[pointID-1] : centre;
```

```
        right = pointID+1 >= 0 ? mesh.vertices[pointID+1] : centre;
        normal = Vector3.Cross(back - fwd, left - right);
        normal.Normalize();
        normals[pointID] = normal;
      }
    }
    mesh.normals = normals;
    Vector2[] uv = new Vector2[points.Count];
    for (int i = 0; i < uv.Length-4; i+=4) {
      uv[i] = new Vector2(0, 0);
      uv[i+1] = new Vector2(1, 0);
      uv[i+2] = new Vector2(0, 1);
      uv[i+3] = new Vector2(1, 1);
    }
    mesh.uv = uv;
    int[] tris = new int[points.Count*6];
    pointID = 0;
    int triID = 0;
    for (int row = 0; row < 89; row++) {
      for (int col = 0; col < 89; col++) {
        pointID = (row*90)+col;
        triID = (row*(90*6))+(col*6);
        if (row != 0 || col != 0) {
          tris[triID] = pointID+90;
          tris[triID+1] = pointID;
          tris[triID+2] = pointID+1;
        }
        tris[triID+3] = pointID+91;
        tris[triID+4] = pointID+90;
        tris[triID+5] = pointID+1;
      }
    }
    mesh.triangles = tris;
  }

  void Update () {
    if (Input.GetKeyDown(KeyCode.Escape))
```

```
        Application.LoadLevel(0);
    }
}
```

A.6 LASBuilderOld.cs

```
 using UnityEngine;
using System;
using System.IO;
using System.Collections.Generic;

public class LASBuilderOld : MonoBehaviour {

  public Transform pointCloudChunkPrefab;
  public Transform pointCounter;
  public bool allAtOnce = false;
  public int pointsPerTick = 1;

  private int pointID = 0;
  private Vector3[] points;
  private ParticleSystem currentChunk;

  Vector3 velocity = new Vector3(0, 0, 0);
  float size = 100;
  float lifetime = float.PositiveInfinity;
  Color32 color = new Color32(255, 255, 255, 255);

  private List<Vector3> LoadPointCloud (string fileName) {
    // TODO: Do all of this in LASReader.

    TextAsset asset = Resources.Load(fileName) as TextAsset;
    Stream s = new MemoryStream(asset.bytes);
    BinaryReader br = new BinaryReader(s);

    string fileSig = new String(br.ReadChars(4));

    if (!fileSig.Equals("LASF")) {
      Debug.LogError("File is not of type 'LAS'.");
      return null;
    }
```

```
// Skip some public header block information for now
for (int i = 0; i < 20; i++)
  br.ReadByte();

string version = br.ReadByte().ToString()+"."+br.ReadByte().ToString();
if (!version.Equals("1.3")) {
  Debug.LogWarning("LAS version "+version+" is not yet supported;
following 1.3 specs.");
  //return;
}

Debug.Log("System Identifier: " + new String(br.ReadChars(32)));
Debug.Log("Generating Software: " + new String(br.ReadChars(32)));
Debug.Log("File Creation Day of Year: " + br.ReadUInt16());
Debug.Log("File Creation Year: " + br.ReadUInt16());

ushort headerSize = br.ReadUInt16();
Debug.Log("Header Size: "+headerSize);

uint offsetToPointData = br.ReadUInt32();
Debug.Log("Offset to Point Data: "+offsetToPointData);

uint vlrCount = br.ReadUInt32();

// TODO: Support all formats, or at least the one specific to the
LIDAR sensor we get.
string pdfID = br.ReadByte().ToString();
Debug.Log("Point Data Format ID: " + pdfID);

ushort pdrLen = br.ReadUInt16();
Debug.Log("Point Data Record Length: "+pdrLen);

uint prCount = br.ReadUInt32();
Debug.Log("Number of Point Records: "+prCount);

// Skip some public header block information for now
for (int i = 0; i < 20; i++)
```

```
    br.ReadByte();

double scaleX = br.ReadDouble();
double scaleY = br.ReadDouble();
double scaleZ = br.ReadDouble();

double offsetX = br.ReadDouble();
double offsetY = br.ReadDouble()*0;
double offsetZ = br.ReadDouble();

double maxX = br.ReadDouble();
double minX = br.ReadDouble();
double maxY = br.ReadDouble();
double minY = br.ReadDouble();
double maxZ = br.ReadDouble();
double minZ = br.ReadDouble();

Debug.Log("Scale: "+scaleX+", "+scaleY+", "+scaleZ);
Debug.Log("Offset: "+offsetX+", "+offsetY+", "+offsetZ);
Debug.Log("Min: "+minX+", "+minY+", "+minZ);
Debug.Log("Max: "+maxX+", "+maxY+", "+maxZ);

// Skip some public header block information for now
for (int i = 220; i < headerSize; i++)
    br.ReadByte();

// Skip variable length record header for now
for (int i = 0; i < 54; i++)
    br.ReadByte();

// Skip straight to point data records
//for (int i = 0; i < offsetToPointData-220; i++)
//    br.ReadByte();

List<Vector3> points = new List<Vector3>();

for (int i = 0; i < 50000; i++) {
```

```
      float x = (float)((float)br.ReadInt32() * scaleX + offsetX)/1000;
      float y = (float)((float)br.ReadInt32() * scaleY + offsetY)/1000;
      float z = (float)((float)br.ReadInt32() * scaleZ + offsetZ)/1000;

      // Skip other information for now
      for (int j = 0; j < pdrLen-12; j++)
        br.ReadByte();

      //Debug.Log(new Vector3(x, y, z));
      //Instantiate(point, new Vector3(x, y, z), Quaternion.identity);
      points.Add(new Vector3(x, y, z));
      }

    return points;
}

void Start () {
    points = LoadPointCloud("test1").ToArray();
    currentChunk = ((Transform)
   Instantiate(pointCloudChunkPrefab)).particleSystem;

    if (!allAtOnce)
      return;

    foreach (Vector3 point in points) {
      currentChunk.Emit(point, velocity, size, lifetime, color);
    }

    pointCounter.guiText.text = points.Length.ToString();
}

// Update is called once per frame
void Update () {
    if (allAtOnce || points == null || pointID >= points.Length)
      return;

    int until = pointID + pointsPerTick;
```

```
    while (pointID < until &&  pointID < points.Length)
      currentChunk.Emit(points[pointID++], velocity, size, lifetime,
    color);

    pointCounter.guiText.text = pointID.ToString();
  }
}
```

A.7 LASReader.cs

```
 using UnityEngine;
using System;
using System.IO;

public class LASReader : MonoBehaviour {

  public Transform point;

  void Start () {
    TextAsset asset = Resources.Load("test1") as TextAsset;
    Stream s = new MemoryStream(asset.bytes);
    BinaryReader br = new BinaryReader(s);

    string fileSig = new String(br.ReadChars(4));

    if (!fileSig.Equals("LASF")) {
      Debug.LogError("File is not of type 'LAS'.");
      return;
    }

    // Skip some public header block information for now
    for (int i = 0; i < 20; i++)
      br.ReadByte();

    string version = br.ReadByte().ToString()+"."+br.ReadByte().ToString();
    if (!version.Equals("1.3")) {
      Debug.LogWarning("LAS version "+version+" is not yet supported;
   following 1.3 specs.");
      //return;
    }

    Debug.Log("System Identifier: " + new String(br.ReadChars(32)));
    Debug.Log("Generating Software: " + new String(br.ReadChars(32)));
    Debug.Log("File Creation Day of Year: " + br.ReadUInt16());
    Debug.Log("File Creation Year: " + br.ReadUInt16());
```

```
ushort headerSize = br.ReadUInt16();
Debug.Log("Header Size: "+headerSize);

// Skip some public header block information for now
for (int i = 0; i < 35; i++)
  br.ReadByte();

double scaleX = br.ReadDouble();
double scaleY = br.ReadDouble();
double scaleZ = br.ReadDouble();

double offsetX = br.ReadDouble();
double offsetY = br.ReadDouble()*0;
double offsetZ = br.ReadDouble();

double maxX = br.ReadDouble();
double minX = br.ReadDouble();
double maxY = br.ReadDouble();
double minY = br.ReadDouble();
double maxZ = br.ReadDouble();
double minZ = br.ReadDouble();

Debug.Log("Scale: "+scaleX+", "+scaleY+", "+scaleZ);
Debug.Log("Offset: "+offsetX+", "+offsetY+", "+offsetZ);
Debug.Log("Min: "+minX+", "+minY+", "+minZ);
Debug.Log("Max: "+maxX+", "+maxY+", "+maxZ);

// Skip some public header block information for now
for (int i = 220; i < headerSize; i++)
  br.ReadByte();

// Skip variable length record header for now
for (int i = 0; i < 54; i++)
  br.ReadByte();

int pos = 0;
```

```
    int length = (int)br.BaseStream.Length;
      while (pos < length && pos < 50000) {
      float x = (float)((float)br.ReadInt32() * scaleX + offsetX)/100;
      float y = (float)((float)br.ReadInt32() * scaleY + offsetY)/100;
      float z = (float)((float)br.ReadInt32() * scaleZ + offsetZ)/100;

      // Skip other information for now
      for (int i = 0; i < 8; i++)
        br.ReadByte();

      Instantiate(point, new Vector3(x, y, z), Quaternion.identity);
      //Debug.Log(new Vector3(x, y, z));

      pos += sizeof(int) * 3 + sizeof(byte) * 8;
      }
  }

  void Update () {
    if (Input.GetKeyDown(KeyCode.Escape))
      Application.LoadLevel(0);
  }
}
```

A.8 DroneStream.cs

```
 using UnityEngine;
using System.Collections;

public class DroneStream : MonoBehaviour {

  IEnumerator UpdateTex () {
    WWW www = new WWW("http://localhost:8080");
    yield return www;
        renderer.material.mainTexture = www.texture;
    }

  void Update () {
    StartCoroutine(UpdateTex());
  }
}
```

A.9 CamStreamSock.cs

```
 using UnityEngine;
using System;
using System.IO;
using System.Net.Sockets;



public class CamStreamSock : MonoBehaviour {

  private NetworkStream network;
  private StreamReader reader;

  void Start() {
    TcpClient socket = new TcpClient();
    socket.Connect("localhost", 8082);
    network = socket.GetStream();
    reader = new StreamReader(network);
    }

  void Update() {
    Debug.Log(reader.ReadLine());
  }

  void OnApplicationQuit () {
    network.Close();
  }
}
```

A.10 MainMenu.cs

```
 using UnityEngine;
using System.Collections;

public class MainMenu : MonoBehaviour {


  void OnGUI () {
    if(GUI.Button(new Rect((Screen.width-180)/2, (Screen.height-40)/2 -
   100, 180, 40), "LAS File Reader Cubes"))
      Application.LoadLevel(1);
    if(GUI.Button(new Rect((Screen.width-180)/2, (Screen.height-40)/2 -
   50, 180, 40), "LAS File Reader Particles"))
      Application.LoadLevel(2);
    if(GUI.Button(new Rect((Screen.width-180)/2, (Screen.height-40)/2,
   180, 40), "Physical Scanner"))
      Application.LoadLevel(3);
    if(GUI.Button(new Rect((Screen.width-180)/2, (Screen.height-40)/2 +
   50, 180, 40), "Virtual Scanner"))
      Application.LoadLevel(4);
    if(GUI.Button(new Rect((Screen.width-180)/2, (Screen.height-40)/2 +
   100, 180, 40), "Visualiser"))
      Application.LoadLevel(5);
  }
}
```

B Referenced Microcontroller Code

B.1 Z_Main.ino

```
#include <TimerOne.h>

float sampleTimeMS = 20.0F;
boolean readNewAngles = false;
int dist = 0;

void interruptLoop() {
  /* Flag that new angle readings are needed for the next interrupt call */
  readNewAngles = true;
}

void setup() {
  pc.begin(57600);
  pc.println("Hello, world!");

  /* Setup subsystems */
  imuSetup();
  usSetup();
  batterySetup();

  resetAngleOffsets();
  resetRefAngles();

  imuLoop();

  /* Initialise Timer1 interrupt */
  Timer1.initialize((int) (sampleTimeMS * 1000));
  Timer1.attachInterrupt(interruptLoop);
}

void loop() {
  readSerialInput();
```

```
  /* If reading new angles is scheduled, read from IMU and calculate
    orientation */
  if (readNewAngles) {
    imuLoop();
    dist = getDistanceCM();

    pc.print(angles[YAW]-offsets[YAW]);
    pc.print(",");
    pc.print(angles[PITCH]-offsets[PITCH]);
    pc.print(",");
    pc.print(angles[ROLL]-offsets[ROLL]);
    pc.print(",");
    pc.print(accel[X]);
    pc.print(",");
    pc.print(accel[Y]);
    pc.print(",");
    pc.print(accel[Z]);
    pc.print(",");
    pc.println(dist);

    /* Clear schedule flag */
    readNewAngles = false;
  }
}
```

B.2 P_Angles.ino

```
float angles[3], offsets[3], refAngles[3];
int avCounter;

void resetAngleOffsets() {
  offsets[YAW]   = 0.0F;
  offsets[PITCH] = 0.0F;
  offsets[ROLL]  = 0.0F;//12.0F;
}

void resetRefAngles() {
  refAngles[YAW]   = 0.0F;
  refAngles[PITCH] = 0.0F;
  refAngles[ROLL]  = 0.0F;
}

/* Returns false if still calibrating, else true. */
boolean calibrate() {
  if (avCounter <= 0)
    return true;

  //Else, calibrate by finding the average of 5 angle readings and
   updating the angle offsets.
  //NB: All of this is pure software calibration. Hardware calibration is
   also possible but less flexible.
  avCounter--;
  offsets[YAW] += angles[YAW];
  offsets[PITCH] += angles[PITCH];
  offsets[ROLL] += angles[ROLL];
  if (avCounter <= 0) {
    offsets[YAW] /= 5.0F;
    offsets[PITCH] /= 5.0F;
    offsets[ROLL] /= 5.0F;
    pc.println("Orienation calibrated.");
  } else {
```

```
    /* Delay between angle readings to obtain more temorally spread out
    samples and as a result, a more useful mean. */
    delay(200);
  }
  return false;
}

void setAngles(float yaw, float pitch, float roll) {
  angles[YAW] = yaw;
  angles[PITCH] = pitch;
  angles[ROLL] = roll;
}
```

B.3 A_Ultrasound.ino

```
#define ECHOPIN 2
#define TRIGPIN 3

void usSetup() {
  pinMode(ECHOPIN, INPUT);
  pinMode(TRIGPIN, OUTPUT);
}

int getDistanceCM() {
  digitalWrite(TRIGPIN, LOW);
  delayMicroseconds(2);
  digitalWrite(TRIGPIN, HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIGPIN, LOW);
  return pulseIn(ECHOPIN, HIGH)/58;
}
```

C Referenced Miscellaneous Program Source Code

C.1 laser-detector.cpp

```
//
//  main.cpp
//  lidar
//
//  Created by Yousef Amar on 30.11.13.
//  Copyright (c) 2013 Amar.io. All rights reserved.
//

#include <stdlib.h>
#include <stdio.h>
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace cv;

char* windowName = "Pseudo-LIDAR";

int main(int argc, char** argv) {
    int threshSat = 0, threshVal = 255, toleranceHue = 5, row, col,
   pixelCount, hue, objectID;
    Point mean = Point(0, 0);
    Mat srcGS, srcHSV, temp;
    vector<Mat> channels(3);
    Vec3b hsv, pixel;
    Scalar cols[] = {
        Scalar(0, 255, 0),
        Scalar(255, 0, 0),
        Scalar(255, 255, 0),
        Scalar(255, 0, 255),
        Scalar(0, 255, 255),
        Scalar(0, 0, 255),
        Scalar(255, 255, 255)
```

```
};

// NOTE: CV_CAP_ANY will select built-in webcam; 1 is hardcoded in to
get the logitech webcam.
CvCapture* capture = cvCaptureFromCAM(1);

namedWindow(windowName, CV_WINDOW_AUTOSIZE);

createTrackbar("Hue Tolerance", windowName, &toleranceHue, 90);
createTrackbar("Saturation Thresh", windowName, &threshSat, 255);
createTrackbar("Value Thresh", windowName, &threshVal, 255);

Mat src = cvQueryFrame(capture);
Mat dst(src.rows, src.cols, CV_8UC3);
//Mat temp(src.rows, src.cols, CV_8UC3);
Mat fin(src.rows, 2*src.cols, CV_8UC3);

char key;
do {
    // Create image frames from capture
    src = cvQueryFrame(capture);

    //cvtColor(src, srcGS, CV_RGB2GRAY);
    //threshold(srcGS, dst, thresh, 255, THRESH_BINARY);

    // Only threshold on the red channel (r = bright, gb = dark) since
laser is red
    //split(src, channels);
    //threshold(channels[2], dst, thresh, 255, THRESH_BINARY);

    cvtColor(src, srcHSV, CV_BGR2HSV);

    pixelCount = 0, mean.x = 0, mean.y = 0;
    for(row = 0; row < srcHSV.rows; row++) {
        for(col = 0; col < srcHSV.cols; col ++) {
            hsv = srcHSV.at<Vec3b>(row, col);
            hue = (hsv[0]+90)%180;
```

```
            if (hue < 90 + toleranceHue && hue > 90 - toleranceHue &&
hsv[1] >= threshSat && hsv[2] >= threshVal) {
                dst.data[dst.step[0]*row + dst.step[1]* col + 0] = 255;
                dst.data[dst.step[0]*row + dst.step[1]* col + 1] = 255;
                dst.data[dst.step[0]*row + dst.step[1]* col + 2] = 255;
                mean.x += col;
                mean.y += row;
                pixelCount++;
            } else {
                dst.data[dst.step[0]*row + dst.step[1]* col + 0] = 0;
                dst.data[dst.step[0]*row + dst.step[1]* col + 1] = 0;
                dst.data[dst.step[0]*row + dst.step[1]* col + 2] = 0;
            }
        }
    }

    /*
    objectID = 0;
    for(row = 0; row < dst.rows; row++) {
        for(col = 0; col < dst.cols; col ++) {
            pixel = dst.at<Vec3b>(row, col);
            if (pixel[0] == 255 && pixel[1] == 255 && pixel[2] == 255)
                floodFill(dst, Point(col, row),
cols[objectID++]);//Scalar(rand()%256, rand()%256, rand()%256));
        }
    }
    */

    if (pixelCount) {
        mean.x /= pixelCount;
        mean.y /= pixelCount;
        circle(dst, mean, 10, Scalar(255, 255, 0), 1, 8);
    }

    fin.adjustROI(0, 0, 0, -src.cols);
    src.copyTo(fin);
    fin.adjustROI(0, 0, -src.cols, src.cols);
```

```
        dst.copyTo(fin);
        fin.adjustROI(0, 0, src.cols, 0);
        imshow(windowName, fin);

        key = cvWaitKey(10);
        // Break out of loop if Esc key is pressed
    } while (char(key) != 27);

    // Clean up
    cvReleaseCapture(&capture);
    cvDestroyWindow(windowName);

    return 0;
}
```

C.2 laser-rangefinder.cpp

```
#include <iostream>
#include <string>
#include <strstream>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

int main(int argc, char** argv) {
  // NOTE: CV_CAP_ANY will select built-in webcam; 1 is hardcoded in to
   get the logitech webcam.
  VideoCapture cap(1);
  if(!cap.isOpened())
    return -1;

  int threshSat = 0, threshVal = 255, toleranceHue = 5, row, col,
   pixelCount, hue, objectID;
  Point mean = Point(0, 0);
  Vec3b hsv;

  Mat frame, frameSub, frameSubHSV;
  cap >> frame;
  Mat dst(frame.rows, frame.cols, CV_8UC3);

  Mat structElem = getStructuringElement(MORPH_RECT, Size(5, 5));

  Size dims = frame.size();

  Rect aoi((dims.width*2)/5, 0, (dims.width)/5, dims.height);

  char key;
  float position = 0.0F;
```

```
do {
  cap >> frame;

  frameSub = frame(aoi);
  cvtColor(frameSub, frameSub, CV_BGR2HSV);

  pixelCount = 0, mean.x = 0, mean.y = 0;
  for(row = 0; row < frameSub.rows; row++) {
    for(col = 0; col < frameSub.cols; col ++) {
      hsv = frameSub.at<Vec3b>(row, col);
      hue = (hsv[0]+90)%180;
      if (hue < 90 + toleranceHue && hue > 90 - toleranceHue && hsv[1]
 >= threshSat && hsv[2] >= threshVal) {
        frameSub.data[frameSub.step[0]*row + frameSub.step[1]* col + 0]
 = 255;
        frameSub.data[frameSub.step[0]*row + frameSub.step[1]* col + 1]
 = 255;
        frameSub.data[frameSub.step[0]*row + frameSub.step[1]* col + 2]
 = 255;
        mean.x += col;
        mean.y += row;
        pixelCount++;
      } else {
        frameSub.data[frameSub.step[0]*row + frameSub.step[1]* col + 0]
 = 0;
        frameSub.data[frameSub.step[0]*row + frameSub.step[1]* col + 1]
 = 0;
        frameSub.data[frameSub.step[0]*row + frameSub.step[1]* col + 2]
 = 0;
      }
    }
  }

  //morphologyEx(frameSub, frameSub, MORPH_OPEN, structElem);

  /*
  if (char(key) == 'a') {
```

```
      // Left key pressed
      position -= 0.10F;
      printf("Pos: %.2f\n", position);
    } else if (char(key) == 'd') {
      // Right key pressed
      position += 0.10F;
      printf("Pos: %.2f\n", position);
    }
    */
    if (pixelCount) {
      //if (char(key) == 32)
        printf("%f\n", ((float)mean.y)/pixelCount);
      mean.x /= pixelCount;
      mean.y /= pixelCount;
      mean.x += aoi.x;
      circle(frame, mean, 10, Scalar(255, 255, 0), 1, 8);
    } else {
      printf("No point detected.\n");
    }
    rectangle(frame, aoi, Scalar(0, 255, 255));
    imshow("Camera", frame);

    key = cvWaitKey(10);
    // Break out of loop if Esc key is pressed
  } while (char(key) != 27);

  destroyAllWindows();

  return 0;
}
```

C.3 udoo-blink.py

```
import numpy as np
import cv2

cap = cv2.VideoCapture(3)
cap.set(3,640)
cap.set(4,480)

f = open("/sys/class/gpio/gpio21/direction",'w')
f.write('out')
f.close()
f0 = open("/sys/class/gpio/gpio21/value",'w')
f1 = open("/sys/class/gpio/gpio21/value",'w')

laser = True

while True:

    f1.write('1')
    f1.flush()

    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    src1 = cv2.threshold(gray, 250, 255, cv2.THRESH_BINARY)[1]

    f0.write('0')
    f0.flush()

    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    src2 = cv2.threshold(gray, 250, 255, cv2.THRESH_BINARY)[1]

    dst = cv2.bitwise_xor(src1, src2)

    # Display the resulting frame
    cv2.imshow('Pseudo-LIDAR', dst)
```

```
    if cv2.waitKey(1) & 0xFF == 27:
        break

# When everything done, release the capture
cap.release()
cv2.destroyAllWindows()
```

C.4 stereo-sim.py

```
# Some code based off of official samples from the OpenCV repository at
    https://github.com/Itseez/opencv/

import sys
sys.path.append('/usr/local/lib/python2.7/site-packages')

import numpy as np
import cv2

def draw_flow(img, flow, step=16):
  h, w = img.shape[:2]
  y, x = np.mgrid[step/2:h:step, step/2:w:step].reshape(2, -1)
  fx, fy = flow[y, x].T
  lines = np.vstack([x, y, x+fx, y+fy]).T.reshape(-1, 2, 2)
  lines = np.int32(lines + 0.5)

  print np.mean(fx)
  print np.mean(fy)



  vis = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
  cv2.polylines(vis, lines, 0, (0, 255, 0))
  for (x1, y1), (x2, y2) in lines:
    cv2.circle(vis, (x1, y1), 1, (0, 255, 0), -1)
  return vis

print 'Loading images...'
#imgL = cv2.imread('tsuleft.png')
#imgR = cv2.imread('tsuright.png')
imgL = cv2.imread('shipL.png')
imgR = cv2.imread('shipR.png')

grayL = cv2.cvtColor(imgL, cv2.COLOR_BGR2GRAY)
grayR = cv2.cvtColor(imgR, cv2.COLOR_BGR2GRAY)
```

```
stereo = cv2.StereoSGBM(minDisparity = 0, numDisparities = 16,
    SADWindowSize = 3)

print 'Computing disparity...'
disp = stereo.compute(imgL, imgR).astype(np.float32) / 16.0

flow = cv2.calcOpticalFlowFarneback(grayL, grayR, 0.5, 1, 10, 15, 5, 2,
    cv2.OPTFLOW_FARNEBACK_GAUSSIAN)

#cv2.imshow('left', imgL)
#cv2.imshow('right', imgR)
cv2.imshow('flow', draw_flow(grayR, flow))
cv2.imshow('disparity', (disp-0)/16)

cv2.waitKey()
cv2.destroyAllWindows()
```

C.5 stereo-dual.py

```
import sys
sys.path.append('/usr/local/lib/python2.7/site-packages')

import numpy as np
import cv2

#min_disp = 0
#num_disp = 16

#stereo = cv2.StereoSGBM(minDisparity = min_disp, numDisparities =
    num_disp, SADWindowSize = 3)

window_size = 32
min_disp = 16
num_disp = 112-min_disp
# NOTE: These parameters are based off of the "stereo_match.py" sample
    from the official OpenCV repository at https://github.com/Itseez/opencv/
stereo = cv2.StereoSGBM(minDisparity = min_disp,
  numDisparities = num_disp,
  SADWindowSize = window_size,
  uniquenessRatio = 10,
  speckleWindowSize = 1000,
  speckleRange = 320,
  disp12MaxDiff = 1,
  P1 = 8*3*window_size**2,
  P2 = 32*3*window_size**2,
  fullDP = False
)

captureL = cv2.VideoCapture(0)
captureR = cv2.VideoCapture(2)

ret, frameL = captureL.read()
ret, frameR = captureR.read()
```

```
while True:
  ret, frameL = captureL.read()
  ret, frameR = captureR.read()

  frameL = cv2.flip(frameL, 1)
  frameL = cv2.transpose(frameL)

  frameR = cv2.transpose(frameR)
  frameR = cv2.flip(frameR, 1)

  grayL = cv2.cvtColor(frameL, cv2.COLOR_BGR2GRAY)
  grayR = cv2.cvtColor(frameR, cv2.COLOR_BGR2GRAY)

  disp = stereo.compute(frameL, frameR).astype(np.float32) / 16.0

  cv2.imshow('camL', frameL)
  cv2.imshow('camR', frameR)

  cv2.imshow('disparity', (disp-min_disp)/num_disp)

  key = cv2.waitKey(10)
  if key == 27:
    break

# ply_header = '''ply
# format ascii 1.0
# element vertex %(vert_num)d
# property float x
# property float y
# property float z
# property uchar red
# property uchar green
# property uchar blue
# end_header
# '''

# def write_ply(fn, verts, colors):
```

```
#     verts = verts.reshape(-1, 3)
#     colors = colors.reshape(-1, 3)
#     verts = np.hstack([verts, colors])
#     with open(fn, 'w') as f:
#       f.write(ply_header % dict(vert_num=len(verts)))
#         np.savetxt(f, verts, '%f %f %f %d %d %d')

# print 'generating 3d point cloud...',
# h, w = frame.shape[:2]
# # guess for focal length
# f = 0.8*w
# Q = np.float32([[1, 0, 0, -0.5*w],
#         [0,-1, 0,  0.5*h], # turn points 180 deg around x-axis,
#         [0, 0, 0,     -f], # so that y-axis looks up
#         [0, 0, 1,      0]])
# points = cv2.reprojectImageTo3D(disp, Q)
# colors = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
# mask = disp > disp.min()
# out_points = points[mask]
# out_colors = colors[mask]
# out_fn = 'out2.ply'
# write_ply('out2.ply', out_points, out_colors)
# print '%s saved' % 'out.ply'

cv2.destroyAllWindows()
```

C.6 stereo-single.py

```
# Some code based off of official samples from the OpenCV repository at
    https://github.com/Itseez/opencv/

import sys
sys.path.append('/usr/local/lib/python2.7/site-packages')

import numpy as np
import cv2
import socket

def draw_flow(img, flow, step=16):
  h, w = img.shape[:2]
  y, x = np.mgrid[step/2:h:step, step/2:w:step].reshape(2, -1)
  fx, fy = flow[y, x].T
  lines = np.vstack([x, y, x+fx, y+fy]).T.reshape(-1, 2, 2)
  lines = np.int32(lines + 0.5)

  #print np.mean(fx)
  #print np.mean(fy)

  vis = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
  cv2.polylines(vis, lines, 0, (0, 255, 0))
  for (x1, y1), (x2, y2) in lines:
    cv2.circle(vis, (x1, y1), 1, (0, 255, 0), -1)
  return vis

#sock = socket.socket()
#sock.bind(('localhost', 8082))
#sock.listen(5)
#conn, addr = sock.accept()
#print 'New connection: ', addr

capture = cv2.VideoCapture(1)

ret, framePrev = capture.read()
```

```
ret, frame = capture.read()

frameCnt = 0

while True:
  framePrev = frame
  ret, frame = capture.read()

  gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
  grayPrev = cv2.cvtColor(framePrev, cv2.COLOR_BGR2GRAY)

  min_disp = 0
  num_disp = 16
  stereo = cv2.StereoSGBM(minDisparity = min_disp, numDisparities =
    num_disp, SADWindowSize = 3)

  disp = stereo.compute(frame, framePrev).astype(np.float32) / 16.0

  flow = cv2.calcOpticalFlowFarneback(gray, grayPrev, 0.5, 1, 10, 15, 5,
    2, cv2.OPTFLOW_FARNEBACK_GAUSSIAN)

  cv2.imshow('flow', draw_flow(grayPrev, flow))
  cv2.imshow('disparity', (disp-min_disp)/num_disp)

  #cv2.imshow('cam', frame)

  #conn.send(disp)
  #conn.send('\n');

  key = cv2.waitKey(10)
  if key == 27:
    break

#conn.close()
cv2.destroyAllWindows()
```

C.7 drone-video.js

```
var arDrone = require('ar-drone');
var http = require('http');
var fs = require('fs');

var client = arDrone.createClient();

//client.config('video:video_channel', 0);

var tcpVideoStream = client.getVideoStream();

//client.on('navdata', console.log);

client.disableEmergency();

var server = http.createServer(function(req, res) {
  if (req.url === '/jsfeat.min.js')
    fs.createReadStream('jsfeat.min.js').pipe(res);
  else
    fs.createReadStream('index.html').pipe(res);
}).listen(8080);

require('dronestream').listen(server, {tcpVideoStream: tcpVideoStream});
```

C.8 drone-video.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Terrain Scanner</title>
  <script src="/dronestream/nodecopter-client.js"></script>
  <script>
    var canvasDrone, canvasProc, contextProc, width, height, imageData,
    rgbSize, rgbaData, ws;

    function init() {
      new NodecopterStream(document.getElementById('droneStream'));
      canvasDrone = document.getElementsByTagName('canvas')[0];
      //canvasProc = document.getElementById('procCanvas');

      //contextProc = canvasProc.getContext('2d');

      width = canvasDrone.width;
      height = canvasDrone.height;

      //canvasProc.width = width;
      //canvasProc.height = height;

      //imageData = contextProc.createImageData(width, height);

      rgbSize = width * height * 4;

      rgbaData = new Uint8Array(rgbSize);

      ws = new WebSocket('ws://localhost:8081');
    }
    function onFrame(gl) {
      gl.readPixels(rgbaData);

      ws.send(rgbaData);
```

```
        //for (i = rgbSize; i--;)
        //  imageData.data[i] = rgbaData[i];

        //contextProc.putImageData(imageData, 0, 0);
      }
    </script>
</head>
<body onload="init();">
    <div id="droneStream"></div>
    <canvas id="procCanvas"></canvas>
</body>
</html>
```

C.9 drone-images.js

```
var arDrone = require('ar-drone');
var http = require('http');
var fs = require('fs');

var client = arDrone.createClient();

//client.config('video:video_channel', 0);

var pngStream = arDrone.createClient().getPngStream();

//client.on('navdata', console.log);

client.disableEmergency();

var server = http.createServer(function(req, res) {
  //fs.createReadStream('index.html').pipe(res);

  if (!lastPng) {
    res.writeHead(503);
    res.end('Did not receive any png data yet.');
    return;
  }

  res.writeHead(200, {'Content-Type': 'image/png'});
  res.end(lastPng);
}).listen(8080);

var lastPng;
pngStream
  .on('error', console.log)
  .on('data', function(pngBuffer) {
    lastPng = pngBuffer;
  });
```

C.10 drone-images.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Terrain Scanner</title>
  <script src="/dronestream/nodecopter-client.js"></script>
  <script>
    var canvasDrone, canvasProc, contextProc, width, height, imageData,
    rgbSize, rgbaData, ws;

    function init() {
      new NodecopterStream(document.getElementById('droneStream'));
      canvasDrone = document.getElementsByTagName('canvas')[0];
      canvasProc = document.getElementById('procCanvas');

      contextProc = canvasProc.getContext('2d');

      width = canvasDrone.width;
      height = canvasDrone.height;

      canvasProc.width = width;
      canvasProc.height = height;

      imageData = contextProc.createImageData(width, height);

      rgbSize = width * height * 4;

      rgbaData = new Uint8Array(rgbSize);

      ws = new WebSocket('ws://localhost:8081');
    }
    function onFrame(gl) {
      gl.readPixels(rgbaData);

      ws.send(rgbaData);
```

```
      for (i = rgbSize; i--;)
        imageData.data[i] = rgbaData[i];

      contextProc.putImageData(imageData, 0, 0);
    }
  </script>
</head>
<body onload="init();">
  <div id="droneStream"></div>
  <canvas id="procCanvas"></canvas>
</body>
</html>
```