Historical Stereogram Depth Finding

Group Members

James Deal

Reia Drucker

Daniel Foster

Jordan Richards

Sponsor

Dr. Amy Giroux, Associate Director for Center of Humanities and Digital Research, UCF

National Cemetery Administration

Table of Contents

Executive Summary

The goal of this project, which was sponsored by the National Cemetery Administration, is to use computer vision and machine learning techniques to measure distances and angles between objects in historical stereographic Civil War era photographs of St. Augustine National Cemetery. The data will then be used in order to properly identify the closest modern day gravestones that the historical gravestones in the image are closest to, in order to place the appropriate name on the modern day gravestone.

The reason this approach is necessary is due to the fact that the St. Augustine National Cemetery is unable to use other techniques that may disturb the gravesite to solve this problem due to federal regulations. For example, using radar or some other technique that may disturb the gravesite is not permitted. Therefore, using computer vision and machine learning techniques to solve the problem was deemed the only acceptable approach.

A local executable will be developed that will accept these historical images as input, then the camera's parameters will be estimated via a machine learning model. The parameters that are the most important to predict are the focal length and field of view. After this, predictions will be made about the 3D layout of the scene captured in those images via a depth finding algorithm. After this is done a user will be able to manually plot points and vectors on locations of interest in the image, and the software will automatically calculate important data, such as magnitude of vectors and angle between two vectors, and provide that information to the user. Afterwards, this data can be used in order to map the location of a historical gravestone to the closest modern day gravestone.

This project is significant as it will help bring peace to those soldiers whose tombstones are currently unlabeled. Not only that, it will bring closure to their families and descendants as well. It is also important as it properly preserves a piece of American history. Moreover, it will demonstrate that modern computer vision techniques can be used to learn more about historical photographs.

The results of this project could be refined in order to explore techniques for increasing accuracy of other computer vision techniques in historical images, as most computer vision research is concerned with gleaning information from modern photographs and videos.

Statements of Motivation

General Project Statement of Motivation

Motivation for this project comes mainly from the desire to solve the real-world problem given as the basis for the project, which is a problem never solved in a similar fashion as the solutions devised in this document, and the great opportunities to develop greater understanding and experience in the fields required to solve this problem (mainly Robot Vision and Machine Learning). This project introduces a problem that, after extensive research into ways of solving the problem, seems to be a very fresh issue and has not been definitively solved yet. That concept is an intriguing one, and even though our experience in the associated fields may be limited, it provides an incredible amount of room for learning through developing potential solutions and research into the topics.

Hopefully by the end of the project, these motivations will be fulfilled, and a fully-functional application will be developed for use in solving not only this problem in particular, but any problem that shares the general scope of this project. When meeting for the first few times and with the professors of the class, it was stated multiple times by the group that our main goal is providing the sponsor with a working and usable product. If that goal can be achieved by the end of the project, which we are confident is possible, then an extra, bonus goal involves the application of our solution to other, similar issues.

James Deal Statement of Motivation

Part of my motivation for selecting this project came down to how intrinsically rewarding and unique the end result would turn out. In contrast to the vast majority of projects pitched to the students in senior design, this project offered an opportunity to solve a truly unique and unsolved problem that truly lined up with principles and values I've held for a long time. While most other projects essentially boiled down to building a web app for a company to build its own social network or add on to an existing social network, this project's final product would be something special and distinct from the others. The reason behind that, in my opinion, is how the specifications from the project's sponsor appear to be structured: the mathematics, calculations and algorithms must be functional and precise, and the aesthetics of the application seem to be second to that.

This focus on functionality really appealed to me as well, as building a working product to make these calculations sounds far more engaging and challenging than building a social media web app. Though the projects that involve making a web application or website may be easier (either technically or due to the fact that UCF's main computer science curriculum seems to be focused on the development web applications) and more formulaic than a project like this, I don't find those types of projects nearly as engaging as developing a solution to a legitimately unsolved problem that has real-world impacts.

The other main reason behind my picking this project came from the learning opportunities it provided. The areas of robotic vision and machine learning are incredibly interesting fields of computer science, and they're some areas I've gotten a foundational education in from courses at UCF, though to a very limited extent. Developing further understanding and experience in these fields sounded engaging and rewarding all the way through to the end, and that's something I find important when deciding on a project you'll be working on for eight months.

Reia Drucker Statement of Motivation

My motivation for selecting this project is that it was one of the few projects that really caught my eye. When the project was first pitched, I was interested right away as machine learning and computer vision are the topics in computer science that interest me more than any others. Being able to leverage data and math to make predictions was a concept that excited me ever since I first learned about it. It's something I'm fairly passionate about, so much so that I intend to pursue a Masters in the field after I graduate from UCF.

This seemed like a really good project to force myself to dive deeper into the field and spend more time doing some independent learning at UCF before I graduate. At UCF I had taken classes like Artificial Intelligence, Robot Vision, and Computer Graphics and after taking them I realized how much I still need to learn. Hopefully in undertaking this project I can further my knowledge of computer vision and machine learning and better prepare myself for graduate school. I think that without this project, and senior design in general, that I would have graduated without learning things that I could have learned even if I did not come to UCF. However, this senior design project is something that I think I would only be able to experience having learned computer science at UCF.

I also didn't want to work on something that had already been done before or was a solved problem. While this project is a bit ambitious, the thought of just building a database backed web app instead didn't seem particularly interesting to me. My goal in getting a computer science degree was always to solve problems that hadn't been solved before, since that is what I believe science is all about. I think there just isn't anything remarkable about doing something that has been done before. What made me want to go to school in the first place was so that I could learn the skills I would need to solve problems. I never wanted to learn computer science just to make websites or mobile apps and working on this project was something that was exactly like what I imagined I'd be doing when I first started programming at eleven years old. I've always wanted to solve problems and an unsolved problem that had already been attempted, was something that I couldn't pass up the opportunity to work on.

The other reason I selected this project was due to the fact that I have friends whose loved ones have passed while serving in the armed forces and the thought of them not being able to properly honor those they have lost due to government regulations bothered me. There may not be anyone alive who remembers the people whose names will be placed on the modern gravestones, but I felt compelled to amend this oversight regardless.

Daniel Foster Statement of Motivation

There were several reasons which compelled me to pick this project. One reason which motivated me to pick this project was its uniqueness. It presented a problem I could easily understand—measuring distances and angles in photographs for which we do not have the camera properties—and whose solution I could not even imagine but wanted to try to figure out, which was a perfect combination. Even with the fear of knowing a previous team had attempted and failed to solve this problem, I felt confident that we would find a solution.

Though there were a few other appealing projects, most of them either seemed like basic database portals or social media applications, or were affiliated with organizations whom I did not want to support. Helping the National Cemetery Administration is a worthy cause, as well as one I had not considered to be necessary before hearing about this project. I intend to work in education after I graduate, so my only goal for Senior Design was to choose a problem I thought would be interesting to solve, regardless of how it would relate to my field or future. By picking this project, I hope to work on a real-world issue that will be not just successful in the scope of this project but will aid with similar stereogram issues in the future.

I am also motivated by the chance to hone my skills in Python and in interface design. I have only really started using Python for class and for personal projects in the last year, and so when we agreed that the machine learning, computer vision, and user interface aspects of the project would be completed in Python, I was incredibly excited. User interface design is also one of my main programming interests, so I was looking forward to get working on a project that both had a visual design and component and did not really have a template that we had to build off of, like there would be with other projects such as the ones for building social media websites.

In summary, it was a combination of the clarity of the problem and the open-endedness of the solution that appealed to me most. I was excited to pick such a unique and unsolved project and work with others to find a solution.

Jordan Richards Statement of Motivation

The most exciting thing about this project is how open-ended it is. The presentation posed an interesting problem, and simply asked us to solve it. While a few suggestions have been made as to what approach to take, the solution we come up with will largely be our own. Additionally, another team had already failed to come up with an adequate solution. Regardless of how our team fares, this is a great opportunity to learn a lot about linear algebra and computer vision. I have a fairly strong background in linear algebra, and some experience developing computer vision applications, but the ideas that we will need to leverage to write our solution lie just outside of my comfort zone; an ideal area for research.

This project is also a great opportunity to gain more Python experience. Most of the large projects I've worked on in the past have utilized C++ or Java, so this is a great opportunity to learn about more Python patterns and tools. While I have some Python experience in mainly dev-ops applications, Python is definitely a very different beast when it comes to production code. Being an interpreted language, I will need to learn new tools for code analysis to ensure that I can write clean and correct code.

Another area of motivation for me is the strict deadline. A lot of projects I have worked on in the past stagnated due to a lack of an imposed structure. The time constraints on this project lie in a great sweet spot: long enough to flesh out a complicated idea, while also tight enough to prevent loss of focus and motivation part of the way through. In addition to the time constraints, there is also a lot riding on this project in terms of graduation and the future of my career as a whole, so it’s a lot easier to stay on topic.

All in all, there are several ways in which this project presents a challenge. From the constrained time limit to the complexity of the project itself, it would be hard not to be engaged. I’m looking forward to seeing this through to the end.

Project Goals / Objectives

The goal of this project is to produce a system capable of accurately measuring depth from a stereogram image to within a foot without knowing the camera's specifications. The two aspects of this system, predicting the camera specifications, and construction of a depth map of the image are the main goals of this project. This system will be accompanied by a user interface that will allow the user to upload a stereogram image, and then plot points and vectors on that image at a location of interest. It must also allow the user to be able to compute the angle between two vectors plotted using those points.

The priority of the project is to solve this specific problem, but a general solution that could be applied to solve other similar problems is ideal. This is relatively likely, as nothing about the depth finding algorithm constrains its ability to function only for the set of images we will be working with. The machine learning model will also likely use a generalized dataset to predict camera parameters due to the fact that it will likely be impossible to find a dataset specifically tailored to this problem.

Another goal of this project is to keep the interface very simple so that it can be easily used by those without much technical experience.

Function of the Project

In general, we will provide a program where the user can upload stereogram images. The program will then estimate the focal length of the camera used, and use it to generate a depth mapping or point cloud. From this screen, the user can specify known parameters in the image to tweak the focal length estimation, such as the length of an edge or an angle. After the user has an accurate measure of the focal length, they can use the program to determine the distance between 2 points in the scene, or the angle between 2 vectors.

This will probably require a GUI, where the user can draw on the depth map or point cloud and enter measurements. The program should also display the measurements the user has requested. Optionally, this display could also have camera controls, allowing the user to pan around the point cloud (if possible).

Design Criteria and Constraints

- Final product must be user friendly.
- Final product must be accessible as an executable file.
- Final product must also be adjustable to user input.
- Final Product should run on Windows or Linux
- Project must be completed by the four team members listed above by the Senior Design showcase at the end of the Spring semester in 2022.
- Measurements by the depth-finding algorithm should be accurate to within a foot (with modern images).
- Development process needs to stay within budget, meaning it should have little to no expenses.
- Finding a good dataset to train a model on may be difficult.

Broader Impacts

Completion of this project will have a large impact on the field of historical stereoscopic analysis. The scope of this project will focus on getting accurate measurements of just one specific stereogram image, but if that is successful, then there is no reason that it cannot be applied to many more. The results of this project may enable a greater ability for historians to glean information from old stereogram photographs. It will especially be useful to the National Cemetery Administration, as the depth-finding algorithm is trained primarily on images of cemeteries, making it optimal for problems in the same field.

The direct results of this project will also involve bringing closure to the families whose loved ones have passed. The software that we develop could also be adapted for use in many different applications where photos could be used to measure distances and angles between different objects in an image. Although our program will be optimized for solving the issue of the stereogram photo, it can be generalized for many other uses. For example it could be used in construction, reconnaissance, engineering, or medicine to enable quick and precise measurements of objects of interest in a relatively accessible and efficient manner. The machine learning model to predict properties of a camera could be used in forensics applications to trace the types of cameras used for certain photographs. This could help aid in criminal investigations of child pornography as determining the type of camera used may help law enforcement track down a suspect.

Legal, Ethical, and Privacy Issues

Since this software is being developed by students of a public university, it makes sense for the project to be open source and accessible to the public. Considering we have no legal obligation to provide our work exclusively to the National Cemetery Administration, and also considering that they are a government organization whose goal is to serve the public, this made the most sense to us. Therefore the project will be released under the GPL v3.0 License. This license ensures that any work done on this project and any derivatives of it must remain free and open source, which we believe to be the best for this project.

Since we are processing images that could potentially be sensitive, the program will not store any information that can identify the image or the user. The software will also have no link to any database, nor query the web in any way. What a user does with this program will stay exclusively on their computer.

Another concern is that any machine learning algorithm developed to make predictions on a camera's specifications could be used to trace the model of camera used for any photograph, raising privacy concerns. However, considering that by publishing a photo you give up an expectation of privacy to the data contained within, it is not of concern for us. Additionally, there is not much we could do to prevent this from happening in the first place as the ability to predict camera specifications is a core function of this project.

Budget and Financing

The budget for this project is basically $0.00. We have been told that reimbursement for small expenses may be possible but generally we want to avoid spending money on this project as it has the potential to be out of pocket.

Luckily there are a few free resources that we can use to train our model, such as Google Colab. Training our model on our own PCs is also possible. Outside of the machine learning aspects of this project, there shouldn’t be very many computationally expensive tasks.

Specifications and Requirements

Need to Have

- Generalized model to work on many different types of images and multiple stereogram settings
- Users will need to upload their stereogram images as right and left
- Users should be able to draw a vector on an uploaded image and input a known length
- Angles between vectors should be able to be calculated
- Multiple points and vector displayed at the same time
- One image will be displayed
- Z-axis required to be oriented into the screen
- Zoom feature for point plotting
- Depth finding accurate to within one foot
- Separate data output for predicted and adjusted lengths
- Estimation of depth needs to be adjusted for user input

Nice to Have

- Change of basis support to remap any point to origin
- Save point and vector data to a file and load data from file
- Unit conversion between metric and imperial
- Multiple coordinate systems for vector data

Signed off on 9/27/21 by sponsor

Ideas to Explore / Potential Solutions

The ideas we are exploring mostly have to do with what went wrong with the previous project, as their idea of a model to make predictions about the camera's specifications and then using those predictions to run a depth finding algorithm was basically what we had thought of as well. The main issue with the previous group's project was the incredibly inaccurate focal length prediction model, giving results with accuracies as low as 10%. There could be several sources of this issue, and more information about possible explanations behind the severe inaccuracy of their model can be found in the Focal Length Prediction Model section of this document.

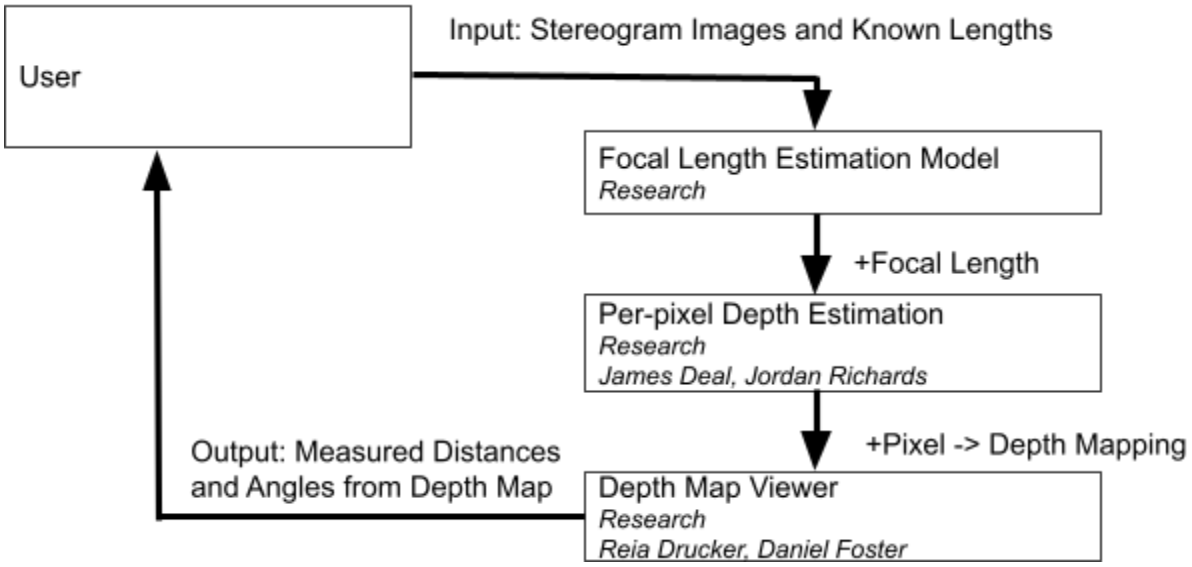
Right now, one idea is that the training images may have all been from calibrated cameras which might not have led to good predictions for such an old camera which would have been physical in nature and thus uncalibrated. Ideally the model the previous group designed will be effective with some changes and we will be spending most of our time tweaking that model for as high an accuracy as we can. Potentially we could end up distorting calibrated images in a way that allows them to more closely match the images we want the model to make predictions on. This would introduce noise and distortions present in the target image to the training and validation data that would most likely throw off the previous group's model when making its predictions based on "cleaner" training data. Another possibility is that we could run the photos through an edge detection algorithm as that is really the only relevant data when finding the edge to edge distance.

Another potential solution is to use the fact that our image has some parallel lines in it that we could use to estimate a vanishing point and find the focal length of the cameras that way. This solution would be much more specific to this problem and most likely doesn't generalize as well, but it avoids having to use machine learning in the case that we just can't produce a good data set for the model to train on.

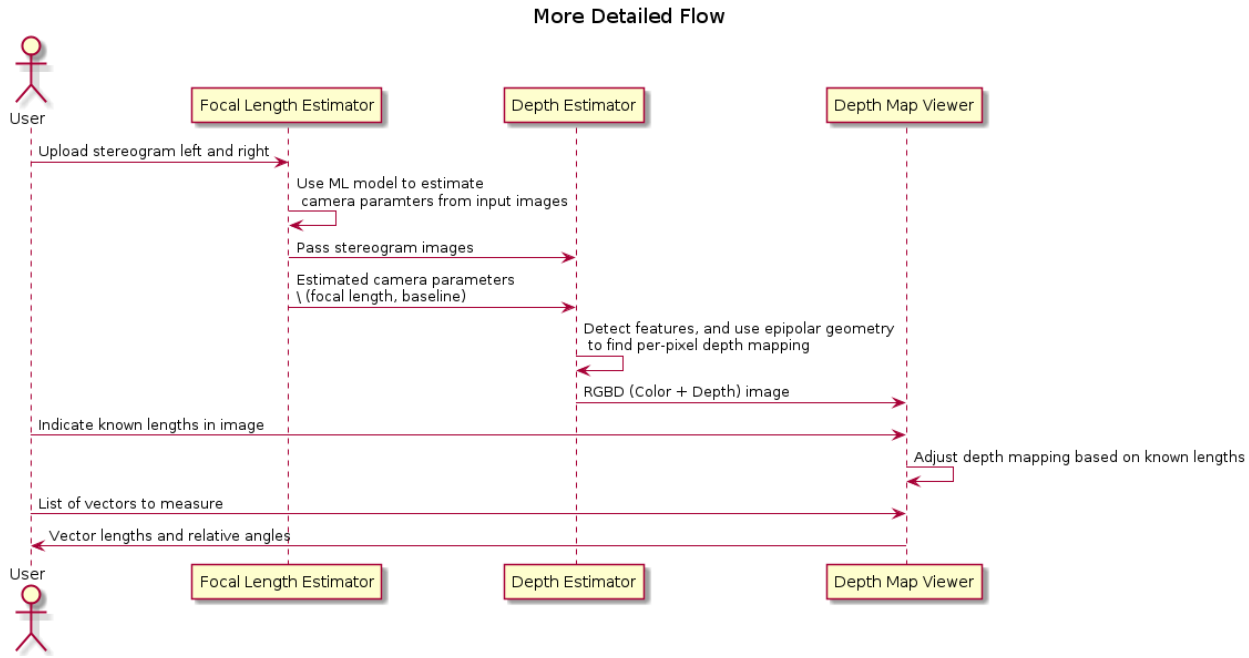
We might also try guessing the focal length of the camera and using it to produce a depth map, and then we can use the known characteristics of our scene (the length of the pyramid, angles, etc.) in order to search for the real focal length. This would also remove the need for a machine learning model, which would take a lot of time developing, training, validating and reworking throughout the course of the project in favor of a more pure, math-based solution to this problem.

This method of finding focal length would be about as generalizable as the use of parallel lines to find a vanishing point, and finding a focal length from there. It would still be hard to apply this method to images without a certain amount of significant geometric data present, such as flat landscape images or images where distances and real-world geometric measurements are unknown. For the purposes of the targeted photograph this project is based around however, it could work.

Block Diagrams / Figures



Project flow diagram.



Use case diagram.

Project Milestones (with dates)

- **September 30**
  - Initial project proposal completed, 3 pages per person.
  - Allocate team roles.

- **October 11**
  - Select GUI and Computer Vision program.
  - Complete general GUI layout design.

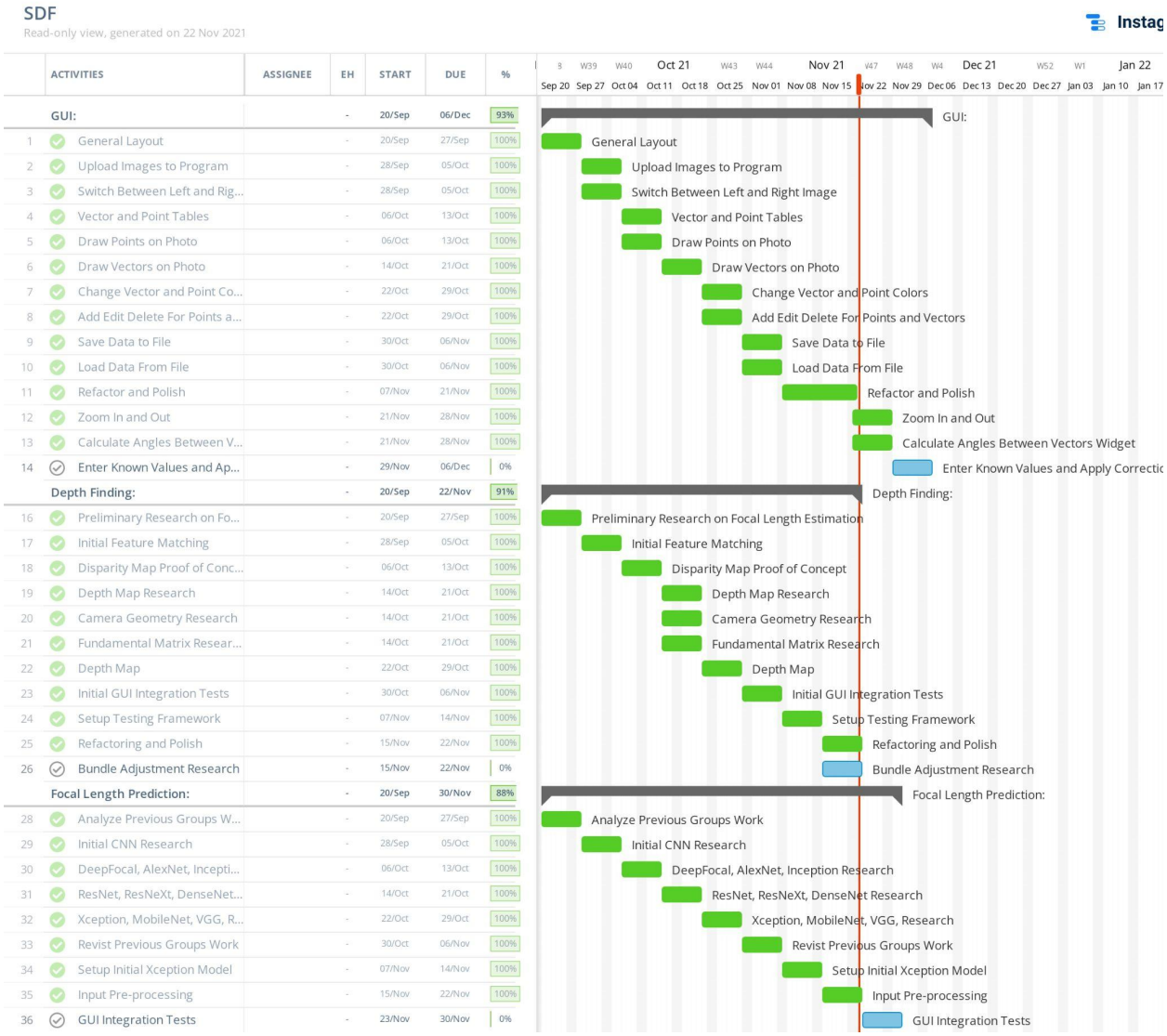
- **October 18**
  - 15 pages per person completed.
  - Upload images to the GUI via File Explorer.

- **November 8**
  - 25 pages per person completed.
  - Complete point and vector classes.

- **November 29**
  - Functional GUI implementation without import capabilities. Includes being able to zoom in and out on an image and draw on the image. Also point and vector table updates based on user input.
  - Functional depth-finding algorithm implementation.

- **December 6**
  - 30 pages per person completed, finished documentation.
  - Functional GUI implementation, including ability to import a dummy model into the program.

Gantt Chart



Gantt Chart Progress as of 11/21/21

Much of the development after this first semester will involve tuning the accuracy of our model in order to increase performance. Nothing can really be fully planned out yet since we do not yet know what issues we will need to address.

Division of Labor

It was decided that each member of the group should have a primary and secondary focus to make sure that each section has at least two team members working on it. The reason for this is mostly so that even if one member is having issues there will be at least one other member who will be able to help them out. Each team member's primary and secondary focus are listed below along with some details about what exactly they will be doing for the first semester of senior design.

Note that every team member is involved in the GUI for the first semester as that is the main section we want completed by the end of SD1. At the end of SD1, more focus will be given to the depth finding and machine learning components of the project and any work on the GUI will be of secondary concern as it will be mostly complete at that time.

James Deal

Primary: Machine Learning Model (Research / Prototyping)
Secondary: GUI (Testing / user experience)

Reia Drucker

Primary: GUI (PhotoDisplayer / Math Support Classes)
Secondary: Depth Finding Algorithm / Project Management (Verification of accuracy)

Daniel Foster

Primary: GUI (Data Management / handling user input)
Secondary: Machine Learning Model (Research Oversight)

Jordan Richards

Primary: Depth Finding Algorithm (Research / Prototyping)
Secondary: GUI (Testing / user experience)

Design Philosophy

This project was brought to us with very loose guidelines and with a focus above all else of providing results. Given the fact that a previous group had already attempted this project and had been unsuccessful, along with the fact that the problem presented was quite challenging, the clear choice of design philosophy for this project was to use iterative design.

Iterative design is the process by which a product is repeatedly proposed, prototyped, tested and reevaluated prior to working product release. Iterative design operates on the following principle: It is unrealistic to create an ideal product on the first try. We knew that going into this project there would be much to learn and much to do, so we felt that the best approach would be to prototype as quickly as possible and see what changes we would have to make to our design.

The GUI and depth finding algorithm were chosen as the best places to start and are planned to be nearly fully functional at the end of Senior Design 1. This approach gives us the maximum amount of time to find solutions to the problem of not knowing the focal length, as if the machine learning approach does not work well enough we will have time to try other approaches. With the GUI and depth finding algorithm in place we will be prepared to try many different solutions quickly and iterate until we find the ideal solution.

As part of this rapid design process we have been consulting with our sponsor in order to receive feedback about our design on a bi-weekly basis. This approach has proven to be quite effective, as very little time has been misallocated during development because of this. Every iteration and new feature is discussed with our sponsor in the planning step, as well as following the implementation of the first version in order to receive any feedback and adjust the final version accordingly.

So far we have not had to make any adjustments to the project, as our conversations with the sponsor have ensured that we have a clear direction for every single feature we are working on. The sponsor has also been very supportive of this decision and so far has been very pleased with our progress.

Depth Finding Algorithm

Technical Objectives

Inputs

The primary input is the pair of input images, forming a stereogram. As these are scans of historical photos, these are expected to be somewhat noisy. In our case, our target historical photo is an image depicting a gravesite with a few pyramids, shown in fig. 1.

The position of the photo centers is also expected to vary between the two images. The images in this case are not cropped, so there will be additional noise around the edges. These images might have some degree of additional distortion, which we will not know ahead of time.

Figure 1: Our target historical stereogram, featuring left and right frames taken simultaneously by 2 cameras. Source [1].

From the focal length estimator, we expect to receive a rough estimate of the focal length. This estimate is expected to have some error, so further refinement might be required to achieve a useful depth estimate.

The user will also need to specify some ground truth data for the image. This data will provide a pair of points in one of the images, as well as a required scale. In the case of our target historical photos, this will likely be the length of the pyramid's base. Ideally, this will be used to establish the scale of the output depth map.

We don’t expect to have accurate camera intrinsics. For instance, parameters such as distortion will need to be estimated in case they are not negligible. Additionally, we won’t have extrinsic parameters for our cameras. In other words, we won’t know the relative rotation and translation of the right camera from the left. We may need to come up with some method of estimating this.

Output

The primary output of the depth estimation algorithm will be a dense mapping from image coordinates to 3D world coordinates. This will likely be relative to one of the camera’s reference frames, but we should be able to transform these coordinates to be relative to a landmark in the image, such as the base of a pyramid.

Acceptance Criteria

Our acceptance criteria will be based on experimental accuracy from a test data set, within some reasonable distance to the camera. While having good performance will improve the user experience, accuracy is far more important. We expect the accuracy of depth information to be within 30 cm (approximately one foot) of reality for any object visible in the image, provided that the furthest object is as close as the last visible pyramid in the image.

Since we don’t have ground-truth data for the historical image, we will need to base this estimate on accuracy for a test data set. For the test data set, we can estimate the expected percentage of error for depth estimates provided by this algorithm by comparing it to ground truth test data. We will then assume this expected accuracy is probably accurate to our target historical stereogram.

Related Work

Our problem is a variant of structure from motion [2]. We have multiple frames, between which some objects have moved relative to the camera. What makes our problem unique is that we don’t have the intrinsic or extrinsic parameters of the camera, and we only have 2 images to work with.

Camera Calibration

A few solutions already exist that attempt to find camera parameters based on the structure of an image.

fSpy is a tool for finding camera focal length [3]. It takes in a user-inputted set of pairs of parallel lines in the image, which should be perpendicular. Given enough of these pairs, it is able to determine the focal length. However, fSpy relies on the scene having enough good perpendicular lines to find the focal length. While this is typical of a cityscape, this isn't exactly the case for our target historical photo. Additionally, this method doesn't account for distortion, so we would need some way of un-distorting our photo before considering this method.

The camera calibration system in [4] is built for video frames taken by traffic cameras. Similar to [3] it uses parallel lines to solve for camera parameters. However, it uses the motion of cars between frames as one set of parallel lines. If our cameras had the same rotation and the only motion between our left and right frames was translation, we might be able to use the same method to extract lines. However, our camera setup likely includes some rotation between the camera frames, so the motion of objects would no longer be parallel. This method also relies on undistorted frames.

Computer Vision Libraries

Many tools and libraries exist to perform the different parts of the structure-from-motion pipeline. However, most pre-made solutions are made to solve the problem with slightly different constraints (e.g. many frames, little noise, negligible distortion). As this is the case, we will be using general purpose computer vision libraries which provide the tools we need to set up our own pipeline.

One solid choice of library is OpenMVG [5]. OpenMVG provides tools for multiple view geometry calculations, including methods for structure from motion and PnP (perspective-n-points is a related problem for finding camera pose relative to a camera of known dimensions). The only issue is that OpenMVG does not provide Python bindings, which means we would need to write our own wrapper in order to use it for this project.

OpenCV [6] is a mature general purpose computer vision library. While OpenCV does not provide many features specifically related to structure from motion, it does provide many of the tools we need to build most of the parts of our pipeline. OpenCV also has Python bindings, allowing us to avoid wrapping / distributing native binaries ourselves. The only downside is that OpenCV doesn't provide specific methods for structure from motion, such as bundle adjustment, so we will need to compose these ourselves from other more basic computer-vision building blocks.

In addition to a computer vision library, we will also use a combination of a few math libraries. Numpy [7] provides tools for linear algebra and other mathematical operations. SciPy [8] provides a least-squares optimizer which will be useful for implementing some methods that OpenCV does not provide.

Modeling The Problem

Now that we have a basic understanding of the tools we are working with, we can start analyzing the problem.

Pinhole Camera Model

We will start by defining a good model for the camera. [9] describes a simplified model of the camera, known as the pinhole camera model. This model is made up of a few important parts, as shown in fig. 2.

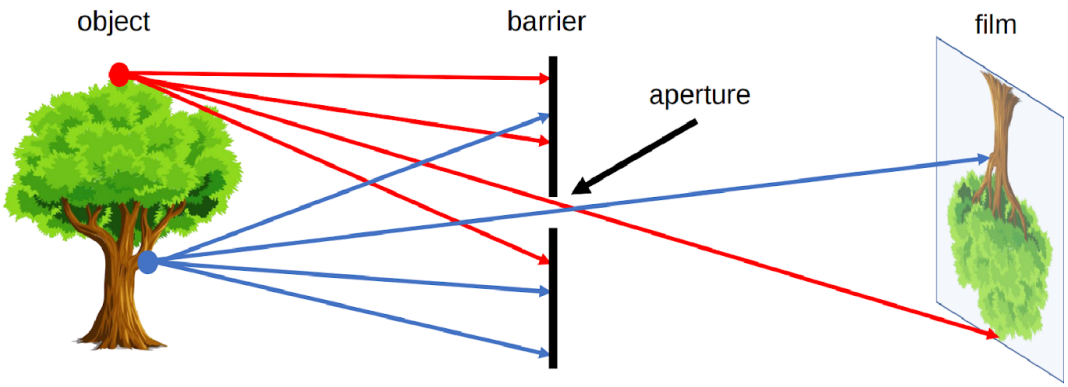


Figure 2: Source: [9]. Diagram of a pinhole camera.

In a pinhole camera, the light from our scene is directed through an *aperture* and exposed on the other side. Optimally, this aperture would be a single point, but in a physical camera this would be impossible. In a normal camera, this would be the focal point of the lens, where all the incoming light beams intersect. We will use this as the origin of the camera's reference coordinate system.

The plane where our image will be projected is known as the *image plane*. We will call the 2D coordinates mapping the intersection of incoming light beams and the image plane *image space*. When we are dealing with a digital image, the units of these coordinates are often measured in pixels.

The *focal length* of the camera is the distance from the aperture to this image plane. It is useful for the focal length to share the units of the image space coordinates, so we will use pixels.

The line normal to the aperture passing through the origin of the camera is the *optical axis*. The intersection of the optical axis and the aperture is called the *principal point*. Another way to see the principal point is the projection of the origin in image space.
As aforementioned, creating a useful pinhole camera is difficult, as we would like the aperture to be a single point, which is impossible with a physical camera. In order to make re-focusing the light on a single point possible, most cameras use lenses instead. These lenses don't necessarily have fixed focal length throughout, possibly due to errors in manufacturing. These variations in focal length cause distortion. The most common form of distortion is radial distortion, where the focal length varies as we move away from the center of the lens. fig. 3 shows several examples of radial distortion. If the distortion is significant, we will need to account for it before we can get accurate measurements.

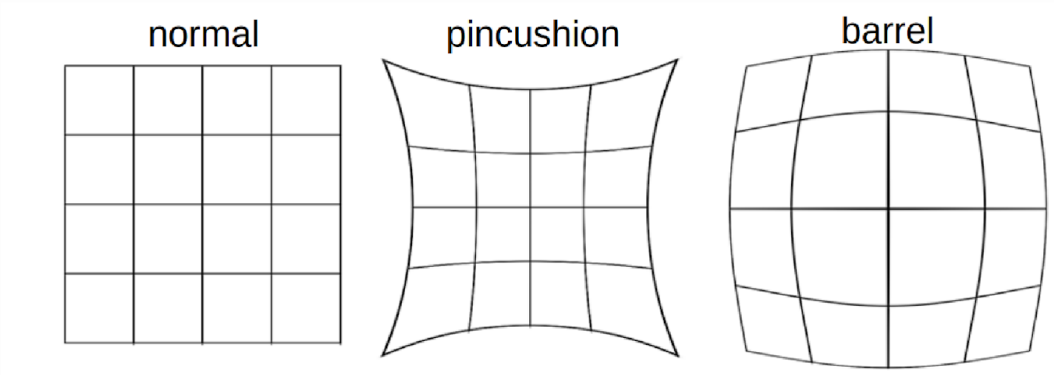


Figure 3: Source [9]. Examples of the 2 types of radial distortion, caused by variations in focal length of the lens.

Now that we have a reasonable model of the camera, it is important to understand how points in 3D world coordinates are projected to 2D image space. This transformation can be split into 2 main steps each defined by a set of parameters. The extrinsic parameters of the camera encode the pose information, including the rotation and translation of the camera in world coordinates. This is especially important if you have multiple cameras, like we do in our stereogram setup. For instance, we can use the left camera's reference frame as our world coordinates, making the pose of the right camera relative to the first camera. The intrinsic parameters describe the projection from 3D coordinates in our camera's reference system to image space. These parameters include the center of our image and the focal length.

The Math of Projection

Now that we have a basic understanding of the camera model, we need a way to represent these transformations using linear operations. [10] describes a new coordinate system that we can use to describe the type of transformation we need: homogeneous coordinates. We will focus on the 2D case for the purpose of example, but the ideas represented here can easily be extended to 3D.

A point in Cartesian coordinates can be represented as a vector [*x y*], the distance along the coordinate axes from the origin. The same point can be written in homogeneous coordinates as a vector [*xZ yZ Z*], where *Z* is a non-zero real number. Z acts as a normalization factor: scaling our entire vector by any non-zero scaling factor will yield the same point, as shown in fig. 4.

$$\begin{bmatrix} xZ \\ yZ \\ Z \end{bmatrix} = Z \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \propto \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 4: The normalization factor makes points represented in homogeneous coordinates invariant to scale.

If our normalization factor is zero, our point in homogeneous coordinates no longer maps back to a finite point in Cartesian coordinates. A vector of the form [*x y* 0] represents a point in the direction [*x y*] infinitely far from the origin. In higher dimensions, a normalization factor at 0 yields lines and planes at infinity.

Lines are described using the same format as points. Given a line described by vector λ = [*a b c*], and a point described by vector *p* = [*xZ yZ Z*], if $\lambda \cdot p = 0$ the point p is on line λ. fig. 5 shows how we can find the equation of a line written in homogeneous coordinates.

$$\begin{bmatrix} xZ \\ yZ \\ Z \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = 0$$

$$axZ + byZ + cZ = 0$$

$$y = -(a/b)x - c/b$$

Figure 5: The equation of a line from its homogeneous coordinates representation.

The slope of lines of the form [*a b c*] is $-a/b$, and the y-intercept is at $-c/b$. Lines are also invariant to scaling operations, so $\lambda \propto s\lambda$. The intersection of 2 lines is the cross

product. If our lines are parallel, this intersection will be a point at infinity in the direction of the lines, as shown in fig. 6.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \times \begin{bmatrix} a \\ b \\ d \end{bmatrix} = \begin{bmatrix} b*d - b*c \\ a*c - a*d \\ a*b - a*b \end{bmatrix} \propto \begin{bmatrix} b \\ -a \\ 0 \end{bmatrix}$$

Figure 6: The intersection of parallel lines.

Linear transformations in homogeneous coordinates can be modeled as matrices, just like with Cartesian coordinates. As homogeneous coordinates add a normalization factor, these matrices have additional degrees of freedom and can represent more types of transformations. The most general form of transformation allowed by homogeneous transformation matrices is projection.

In fig. 7, *A* is a 2x2 matrix. This matrix can be seen as an affine transformation in Cartesian coordinates, potentially including scaling, rotation, or skew. *t* is our translation vector, a fixed value that will be added to each coordinate after the transformation represented by A is applied. *s* is a normalization factor. Our vectors will be scaled down by this factor after the other operations have been applied. *b* is a skew vector. It allows us to break parallelism, lines that are parallel before our transformation no longer need to be parallel after. Breaking parallelism allows us to project images with non-orthographic perspectives.

$$P = \begin{bmatrix} A_{11} & A_{12} & t_1 \\ A_{21} & A_{22} & t_2 \\ b_1 & b_2 & s \end{bmatrix} = \begin{bmatrix} A & t \\ b & s \end{bmatrix}$$

Figure 7: Breakdown of a projection matrix into a few key components.

With homogeneous coordinates and projection matrices we can now map from 3D coordinates to the camera's image space using a linear transformation. The most basic building block of the projection matrix in the pinhole camera model is the camera intrinsic matrix, *K*. *K* maps from 3D directions in our camera's reference space in Cartesian coordinates, to 2D homogeneous coordinates in image space. Note that *K* will not capture any information about the position of our camera in world coordinates, it assumes that the vectors it transforms are from the camera's origin to a point in 3D space.

The illustration in fig. 8 shows how points in 3D are projected to image space. A few similar triangles are present in this image, and we can use these to establish a relationship between the parameters of our camera, the point's coordinates in our camera's reference frame, and the coordinates in image space. fig. 9 shows what this relationship would look like in Cartesian coordinates. We need to divide by *z* to get the intended result, making this relationship non-linear. However homogeneous coordinates allow us to rewrite this relationship as a linear transformation, as shown in fig. 10.

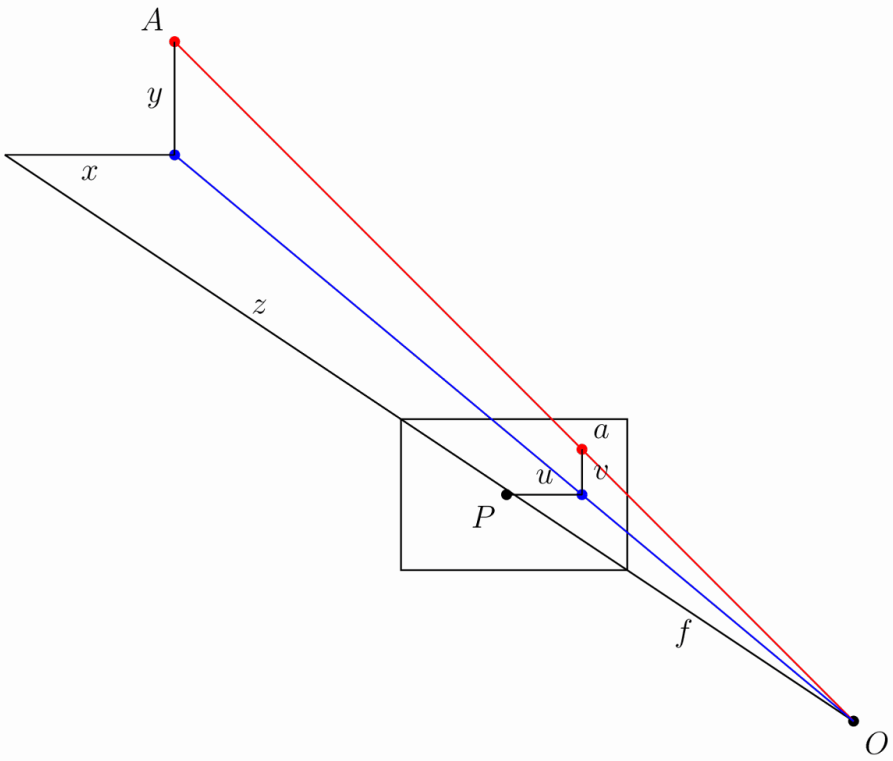


Figure 8: Diagram of projection. *O* is the aperture. *f* is the focal length of the lens. *P* is the principal point. *A* is the point in 3D space, *a* is the same point on the image plane. [*x y z*] specifies the location of *A* in the camera's reference frame. [*u v*] specifies the location of *a* in image coordinates.

$$\begin{bmatrix} u - c_x \\ v - c_y \end{bmatrix} / f = \begin{bmatrix} x \\ y \end{bmatrix} / z$$

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f * x/z + p_x \\ f * y/z + p_y \end{bmatrix}$$

Figure 9: Solving for our image coordinates given details about our camera, using Cartesian coordinates. Here the center of the image is [c_x c_y].

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \propto \begin{bmatrix} uz \\ vz \\ z \end{bmatrix} = \begin{bmatrix} f * x + c_x * z \\ f * y + c_y * z \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad a = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \quad A = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$a \propto KA$$

Figure 10: Converting our projection equation to homogeneous coordinates and defining the camera intrinsic matrix K.

This matrix *K* has a few key parts. [c_x c_y] represents the location of the principal point. f_x and f_y are the focal length of the image along the coordinate axes. If $f_x = f_y$, the image has square pixels. This is a useful assumption that will simplify the problem of solving for *K*.

As K maps from 3D to 2D coordinates, there has to be some loss of information in the process. As K is a transformation on homographic coordinates, the scale of the input does not affect the output. Therefore, all points in the same direction from the camera's origin are mapped to the same point in image space. K is a bijective mapping between 3D directions and 2D points in image space.

In 3D, the intersection of 2 parallel lines will be a point at infinity in the same direction as those lines. Projecting those lines to image space, we will find that they are no longer parallel (provided they are visible in the image). The intersection of these 2 parallel lines in image space is known as a vanishing point. By inverting K, we can map this vanishing point to a direction in 3D space in the same direction as our lines. This is useful as it allows us to find the angle of a pair of parallel lines in our image. Provided we can find 2 sets of parallel lines that lie on the same plane, we can use this technique to find the normal vector for a plane in our image as well.

The intrinsic projection matrix does not include a mapping from world coordinates to camera coordinates. This transformation might include translation, so a matrix including this camera pose information would need to map from 3D homogeneous world coordinates to 2D homogeneous image coordinates.

Typically, this pose information would be represented as a rotation matrix and translation vector for the camera's reference system relative to world coordinates. This

means we need to apply the inverse of these operations to map the points from world coordinates to camera coordinates: $R_c = R_p^{-1}$. $T_c = -R_cT_p$, where R_c, T_c is the camera's pose information, and R_p, T_p is the pose of points in the camera's reference system relative to the points in world coordinates.

If our camera is not rotated or translated relative to the camera coordinate system, we can write the final projection matrix as $P = K[I\ 0]$ We can incorporate the pose of our points, R_p, T_p, as $P = K[R_pT_p]$.

Determining Depth

Looking at the model we have so far, we can see that as points move away from the camera's origin along the Z axis, their projections move towards the principal point in 3D space. This has the effect of shrinking objects as they move further from the camera, proportional to their distance from the camera. Using this information, if we know the length of an object parallel to the camera plane, as well as the focal length, we can determine the distance of the object to the camera. However in cases where we don't have the necessary information – such as the object's measurements – we won't be able to determine the depth with just the information from a single camera.

If we don't know an object's rotation relative to the camera, it might be difficult to determine its length parallel to the camera plane. In cases where we know an object's exact dimensions, there are few methods that we can use to determine its relative rotation. If our object has appropriate structure we can find its rotation by finding vanishing points. This is usually only the case for simple objects.

In the general case, this class of problems is known as Perspective-n-Points or PnP. OpenCV provides a number of solutions for this problem, as described in [11] and [12]. As input these functions take in a list of 3D points describing features of our object, as well as a list of 2D points specifying the corresponding features in image space. In order to generate a unique solution, these functions need at least 4 points.

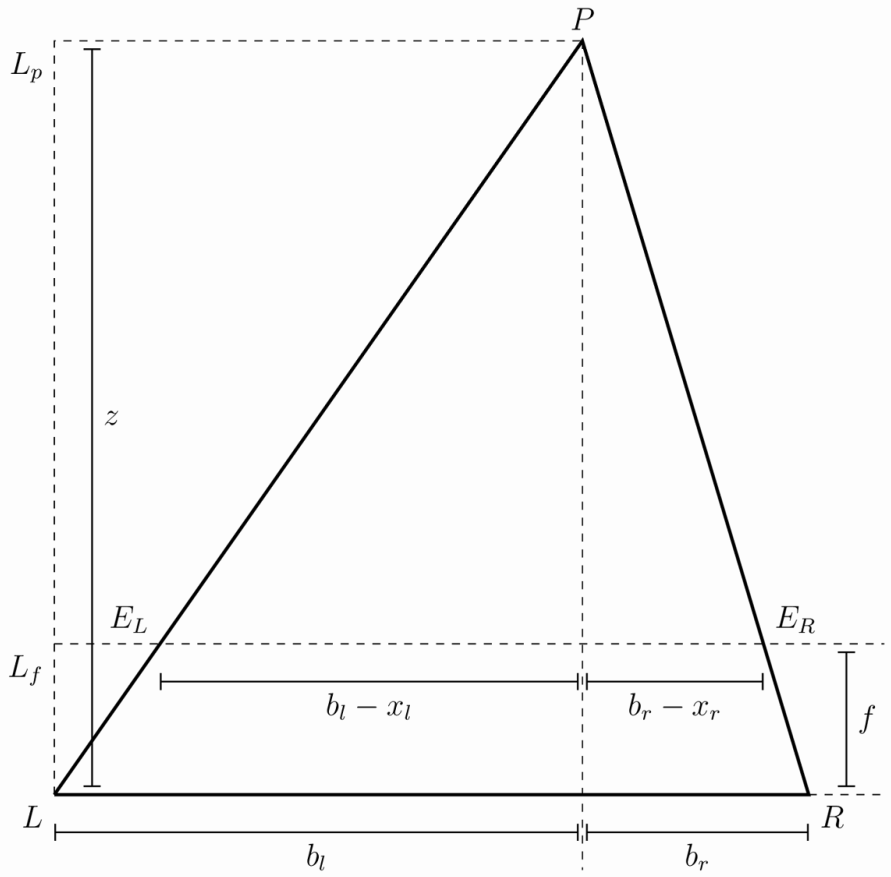
With 2 or more frames, we can use the extra information provided by the second camera to triangulate points in 3D, provided we know the relative pose of the other camera, and that we can identify the same point in both images. The first step of triangulation would be to use the camera intrinsic matrix to find the directions from each camera to our target point. Using each camera's pose information, we can draw lines passing through the camera's origin in the directions obtained from the previous step. The intersection of these lines should be the location of the point in world coordinates.

To account for error, instead of finding the intersection of the lines, we can find the optimal point using least squares optimization. This method requires us to find matches using a global matcher, which might result in a larger number of possible false positive matches. In other words, we can filter matches based on their quality, but if we set the threshold too high our set of matches will be too sparse and we will not have depth information for most of the image. If we set the threshold too low, we will probably end up with many inaccurate matches. If we can constrain our matches based on the geometry of our scene, we might be able to filter out some of these false matches without sacrificing the overall quality of matches.

[13] describes the geometry of a 2 camera setup, or epipolar geometry. In a typical stereogram camera setup, the camera centers are separated by a fixed distance. The segment between the two cameras' centers is known as the *baseline*. The intersection of the baseline with a camera's image plane is known as an *epipole*. If the camera plane and the baseline are parallel, the *epipole* is a point at infinity, though this is typically not the case.

For some 3D point P visible in both cameras' images, we can define a few more structures. The *epipolar plane* is the plane formed by the 2 camera's centers and the point P. It is important to notice that this plane contains the baseline and the epipoles. The *epipolar line* is the intersection of the epipolar plane and a camera plane. For any choice of P, this line will still contain the epipoles.

If our cameras' image planes are parallel to the baseline, we know the focal length and the baseline, and we have a matching pair of points from each image, we can calculate the depth to that point using a more simple method than triangulation. In fig. 11, we can see that $\triangle PLR$ and $\triangle PE_LE_R$ are similar. We can use the similar triangles in the diagram to find b_l (the distance along the baseline between camera L and our point P), as shown in fig. 12.



- L, R: left and right cameras.
- P: a point visible in both cameras.
- b_l, b_r: distance along the baseline from L and R to P
- x_l, x_r: the distance along the x-axis of each camera to the projection of P
- E_L, E_R: endpoints of the epipolar line intersecting the projection lines for P
- f: focal length
- z: depth

Figure 11: Diagram of an epipolar setup where both cameras image planes are parallel to the baseline. *L* and *R* are the left and right cameras. *P* is some point visible to both cameras. b_l and b_r are the distance along the baseline to P from camera's *L* and *R* respectively. x_l and x_r is the distance along the x-axis of each camera to the projection of *P*. E_L and E_R are the endpoints of the epipolar line intersecting the projection line for P. *f* is the focal length, and *z* is the depth.

$$\frac{b_l - x_l}{b_l} = \frac{b_r - x_r}{b_r} \tag{1}$$

$$b_l b_r - x_l b_r = b_r b_l - x_r b_l \tag{2}$$

$$x_l b_r = x_r b_l \tag{3}$$

$$b_l + b_r = b \tag{4}$$

$$x_l b_l + x_l b_r = x_l b \tag{5}$$

$$x_l b_l + x_r b_l = x_l b \tag{6}$$

$$(x_l + x_r) b_l = x_l b \tag{7}$$

$$b_l = \frac{x_l b}{x_l + x_r} \tag{8}$$

Figure 12: Solving for distance along the baseline.

We can label the value $x_l + x_r$ the disparity d. Looking now at similar triangles $\triangle LL_zP$ and $\triangle LL_fE_l$, we can now solve for the depth z, as shown in fig. 13.

$$\frac{z}{b_l} = \frac{f}{x_l} \tag{9}$$

$$z = \frac{f b_l}{x_l} = \frac{f b}{d} \tag{10}$$

Figure 13: Solving for depth from disparity.

As this math depends on our cameras' image planes being parallel to the baseline, we will need to define a transformation to *rectify* the images of cameras that don't adhere to this constraint. If we can find a set of matching points in our images, we can use our points to determine the required transformation to rectify our images.

First we must introduce the concept of the essential matrix. A 3D point P has corresponding points in the image space of each camera, p and p'. If we know p and we want to find p', we know that it must lie on the epipolar line for P. To find the epipolar line, we must know the relative position of our cameras defined by rotation matrix R and translation vector T (in our first camera's reference system). We assume for simplicity that our cameras have a focal length of 1 and centers at [0 0], so $K = K' = I$. Based on this, the projection of p' on camera 1's image plane is $Rp' + T$.

We know that T (the translation vector between the cameras) is our baseline, and therefore must be on the epipolar plane. We also know that $Rp' + T$ must also be on the epipolar plane. We can find a vector normal to the epipolar plane using the cross

product: $T \times (Rp' + T) = T \times Rp'$. For any point on the plane, the dot product with this normal vector should be 0. We know that p is on the epipolar plane as well, so $p \cdot (T \times Rp')$. For 3D vectors we can rewrite our cross product as matrix multiplication: $p^T(T_\times R)p'$. From this we can derive our definition of the essential matrix, $E = T_\times R$. For any point in our first image p, it's corresponding matching point must lie on the line $(p^T E)$.

This definition of the essential matrix assumed that $K = K' = I$, so we can factor this back into the equation to get our fundamental matrix: $F = K^{-T}EK'^{-1}$. This new matrix performs the same function as the essential matrix, but for cameras with varying focal length and center. F has a few interesting properties, it only has 8 degrees-of-freedom since its scale doesn't matter, and it has rank 2 since it maps from points to lines.
Without knowledge of our camera's intrinsic parameters, we might still be able to find the fundamental matrix assuming we can find a sufficient number of independent epipolar lines. Given 2 matching points in our images, $p = [u\ v\ 1]$ and $p' = [u'\ v'\ 1]$, fig. 14 shows how we can construct a linear constraint.

$$p^T F p' = 0$$

$$\begin{bmatrix} u & v & 1 \end{bmatrix} \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix} \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = 0$$

$$\begin{bmatrix} (u * F_{11} + v * F_{21} + F_{31}) & (u * F_{12} + v * F_{22} + F_{32}) & (u * F_{13} + v * F_{23} + F_{33}) \end{bmatrix} \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = 0$$

$$\begin{bmatrix} u'u & v'u & uu'v & v'v & vu' & v' & 1 \end{bmatrix} \begin{bmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{bmatrix} = w \cdot f = 0$$

Figure 14: Constructing a linear constraint on the fundamental matrix from a matching pair of features.

With more matches we can build up a matrix W from row vectors w_i, such that $Wf = 0$. We only need 8 matching points to fully define F (since it only has 8 degrees of freedom), but we can use more to deal with potential noise in our matches, finding F

with the smallest mean squared error. Our resulting estimate for F might have rank 3, while the real F has rank 2, but we can fix this using SVD, as shown in fig. 15.

$$\hat{F} = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} V$$

$$F = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} V$$

Figure 15: Correcting the rank of our estimate for the fundamental matrix using SVD.

While using more than 8 matches to estimate F will be more resistant to noise, we might need to employ other methods to deal with outliers in our matches. One strategy is to use RANSAC: select random samples from our set of matches and find the one that produces the least outliers. Another strategy is to find F that minimizes the median of squared error rather than mean squared error.

Given the camera intrinsic matrix, *K*, we can recover the essential matrix using our estimate of the fundamental matrix. Alternatively, we can use a similar algorithm, the 5 point algorithm, to solve for the essential matrix directly. Once we have the essential matrix, [14] describes a method we can use to recover the pose information of our cameras.

After we have estimates for our camera's parameters, we can refine them using a process known as bundle adjustment [15]. This process uses the parameters for each camera to triangulate our matched points, and then reproject them back to image space. The sum of distances between the reprojected points and the actual points is known as reprojection error. We can then use a least squares minimizer to minimize this reprojection error. This process finds better estimates for the camera's intrinsic parameters (e.g. focal length, distortion) and extrinsic parameters (e.g. pose). However, as the reprojection error function is not linear, the minimizer is not guaranteed to find the best possible solution. In order for bundle adjustment to succeed, we need a reasonable initial estimate for our camera parameters.

Now that we have camera parameters we can create a transformation to align our image planes with the baseline. [16] describes a method for computing this

transformation from the fundamental matrix. This algorithm breaks down the transformation into 3 component parts: projection, similarity, and shearing. The goal of the projection transform is to move the epipoles to a point at infinity. The goal of the similarity transform is to rotate the epipoles so they are aligned with the x-axis. Finally, the goal of the shearing transform is to preserve the aspect ratio of our images. Without the shearing transform, the resulting transformation might squash our image.

In our newly rectified image, we now know that our matches need to be approximately aligned horizontally before we consider them. This constraint allows us to switch to a different kind of feature matcher: a stereo block matcher. In a stereo block matcher, we generate a feature vector for the block around each pixel, and then we find the best match in the same row. As not all of these matches will be accurate, the disparity map is then smoothed to make the output less noisy. The stereo block matcher is able to do this efficiently by only checking for horizontally allowed matches. If we know that points that we care about in our image must be within a certain range of depths, we can use this to constrain where we look for matches as well. By using a sliding window to search for matches instead of checking the entire column, a semi-global block matcher is both more efficient and typically has less noise.

Feature matching in a block matcher still relies on matching images based on their visual appearance. This means that large flat, untextured surfaces, such as walls, often produce inaccurate disparity measurements. Additionally, at the edges of objects, where blocks include parts of the background, the block matcher might also produce inaccurate results, usually resulting in halo around objects known as *speckle*. We can deal with speckle and untextured surfaces using smoothing or filtering, but this will often still result in large areas of inaccurate disparity information.

Revisiting Related Work

We now have all the necessary parts to build a structure-from-motion pipeline. Before we continue, however, it is useful to review the aforementioned work on camera calibration and structure from motion.

fSpy uses the geometry of the image to calibrate the camera. The user needs to find the vanishing points corresponding to 3 pairwise perpendicular directions. They can do this by marking sets of parallel lines on the scene. fSpy uses these lines to calculate the vanishing points. The dot product of the directions of each pair of vanishing points should be zero, as they are perpendicular, as shown in fig. 16.

$$d_i = K^{-1} p_i$$

$$d_1 \cdot d_2 = 0$$
$$(K^{-1} p_1)^T (K^{-1} p_2) = 0$$
$$p_1 (K K^T)^{-1} p_2 = 0$$

Figure 16: Constructing a linear constraint on K from perpendicular vanishing points.

These constraints form a system of linear equations, where our coefficients are the coordinates of the vanishing points, and our unknowns are the parts of the camera intrinsic matrix. Note that these constraints form a linear constraint not on *K*, but on $(KK^T)^{-1}$. This is fine, however, as we can use Cholesky decomposition to recover *K* from this value, according to [9].

With 3 points we get 3 equations, but the camera intrinsic matrix can have up to 5 unknowns including the 2 focal length values, the principal point, and skew parameter. By assuming our image has zero skew and square pixels, we can reduce the number of unknowns to 3.

The calibration technique in [4] works similarly to fSpy. Unlike fSpy, [4] works with more than 2 lines for each vanishing point. In order to find the closest point to a set of lines, they use a special dual space known as *DiamondSpace* [17]. This dual space simplifies the process of finding the most likely vanishing point, and makes it simpler to group lines into perpendicular groups. For the first vanishing point, they assume their video frames will have cars moving in parallel lines, following the lanes. They first filter only moving features in the video, and then match the features between frames to find several motion vectors throughout the video. They then convert the set of lines to diamond space in order to locate the first vanishing point. For the second vanishing point they also rely on the geometry of moving vehicles. They find edges along the cars that remain parallel as the car moves. By converting these lines to their dual in diamond Space, they can filter out the edges that aren't perpendicular to the first vanishing point. After this they find the vanishing point as they did for the first set. Instead of finding a third vanishing point in their scene, they reduce the number of unknowns in the camera intrinsic matrix to just one, by assuming the principal point of each frame is the center of the image. By setting up the same equation as fSpy for the first two vanishing points, they can solve the linear equation for the focal length.

As we mentioned before, both of these techniques rely on having good geometry in the images, which our target historical stereogram does not have. Even the "motion" vectors between our camera frames will likely not be parallel as the cameras' pose likely includes some relative rotation. This process doesn't work for images with significant distortion, and it is unclear whether or not the cameras that took our target images would satisfy this constraint. If we try to account for distortion in this process, it would no longer be a linear system of equations. Bundle adjustment attempts to solve a similar problem for a nonlinear system of equations, but it requires us to have a reasonable initial guess in order to avoid falling into a local minima.

Our computer vision library of choice will be OpenCV as it is a popular and mature computer vision library that provides Python bindings. It includes several high level functions that we can use to build our pipeline. OpenCV provides a wide range of feature matchers to choose from, such as ORB [18], KAZE [19], and SIFT [20], each offering a similar interface but different performance. Of all these feature matchers, ORB seems to have the best trade-off between number of features generated and the accuracy of the generated matches. Another important feature of ORB is that it's feature vectors are invariant under some rotation, which we might have depending on how our cameras are set up.

Once we have enough matches from both images, we can start matching them. As performance is not a consideration for our application, as we only have 2 frames to match, we can use the brute force matcher (`BFMatcher`). `BFMatcher` tries every pairing of points from our images, and returns only the top matches by comparing the feature vectors. In the case of ORB, feature vectors are interpreted as binary strings and not directional vectors, so our distance metric of choice will be the Hamming distance: the number of bits that are different between our vectors.

`findEssentialMat()` estimates the essential matrix using the 5 point algorithm. It requires us to have some initial guess for our camera's intrinsic parameters, including the focal length and the principal point.

Additionally, it depends on us having a high number of features to generate a reasonable pose estimate. In the presence of outliers in our match info, the 5 point algorithm might generate an inaccurate estimate. To deal with this, OpenCV can use one of 2 methods to filter the input points: *RANSAC* or *LMEDS*. LMEDS, or least medians, attempts to find the essential matrix minimizing the median of error for all matches. The median is less susceptible to outliers than the mean, and is overall a better representation of the center of the population of our points. RANSAC is a general

method for filtering noisy inputs when optimizing linear functions. It works by selecting random samples of the input points, generating an essential matrix using the 5 point algorithm for the sample, and then finding the number of matches from our total that lie within some threshold error from that estimate. In practice, LMEDS appears to be more stable for our use case.

`recoverPose()` takes in an essential matrix and breaks it down into its component pose information. It outputs a rotation matrix and a translation vector. The pose corresponding to a particular essential matrix is not unique, and there can be up to 4 different combinations. `recoverPose()` takes in our set of matching points and uses the pose estimate to triangulate their locations. It then finds the pose in which most of the points are triangulated to be in front of the camera.

OpenCV does not provide a method for bundle adjustment, but it does provide several methods for us to implement bundle adjustment ourselves. Typically bundle adjustment uses reprojection error as the cost function, minimizing the difference between re-projected triangulated points and the corresponding points in each image. The initial guess for the best possible triangulated points can be found using `triangulatePoints()` a function that takes in each of our camera's intrinsic matrix and the essential matrix and outputs the triangulated points. This function does not account for distortion, so we will need to call `undistortPoints()` with our guess for the distortion parameters before we call this function.

To reproject these points back to image space, we can use `projectPoints()` which takes in our camera's parameters and outputs the projected points. Bundle adjustment also requires us to be able to calculate the Jacobian of our cost function, and we could calculate it directly if OpenCV provided a Jacobian for *projectPoints()*. Unfortunately while `projectPoints()` does output a Jacobian, it does not include the partial derivatives relative to the input set of points, which we would need. Therefore, in order to obtain the Jacobian we will need to estimate using the finite difference method.

We might also consider using an alternative cost function instead of reprojection error, such as the one described in [21]. This function uses our camera intrinsic matrix and poses information to lines in world coordinates. The error is then the minimum distance between these lines. This avoids triangulating our points and then reprojecting them, but still finds a similar cost.

For the actual bundle adjustment implementation, we need a least square minimizer such as the popular Levenberg-Marquardt algorithm [22]. Fortunately, SciPy offers a

Python implementation: *scipy.optimize.least_squares*. This function can also optionally approximate the Jacobian using the finite difference method for us. It might be useful to consider calculating the Jacobian directly for improved accuracy.

After we have an accurate estimate for the pose of our cameras we would need to generate a rectification transform. `stereoRectify()` takes in our camera's parameters and outputs this transformation for each of our cameras. After we have the transformations, we can actually perform the rectification using `warpPerspective()` a function that transforms an image using a projection matrix.

`StereoSGBM` implements the semi-global block matching algorithm. In addition to the input stereogram, `StereoSGBM` also includes several hyperparameters we can fine-tune to get a better output disparity map. The function of each parameter is described in [23]. The parameter `num Disparities` controls the size of the sliding window, i.e. how far to look to the left for a match for each block. The `speckleWindow`, `minDisparity`, and `speckleRange` parameters control the process that filters out speckles.

Speckle is a type of error that can occur along the edge of objects when the sliding window includes foreground and background elements. The process discards matches for pixels where the minimum disparity in the window specified is less than the minimum. Finally, the smoothing parameters `P1` and `P2` control how smooth the outputted disparity map should be. This can help fill in holes in the disparity map, at the cost of accuracy. As an alternative to generating disparity information for each pixel in our image using a block matcher, [24] uses a neural network to extend sparse depth information. This method would use the depth from our triangulated points, as well as the image data in order to predict the depth data for the rest of the image. The primary advantage of this solution is that it doesn't rely on finding matches in sparsely textured areas in order to determine depth, potentially resulting in less areas of high inaccuracy.

Our Pipeline

Now that we understand all the tools at our disposal, we can use them to construct a solution. fig. 17 breaks our pipeline down into several concrete steps, each building on the input generated from the last until we have our desired output.

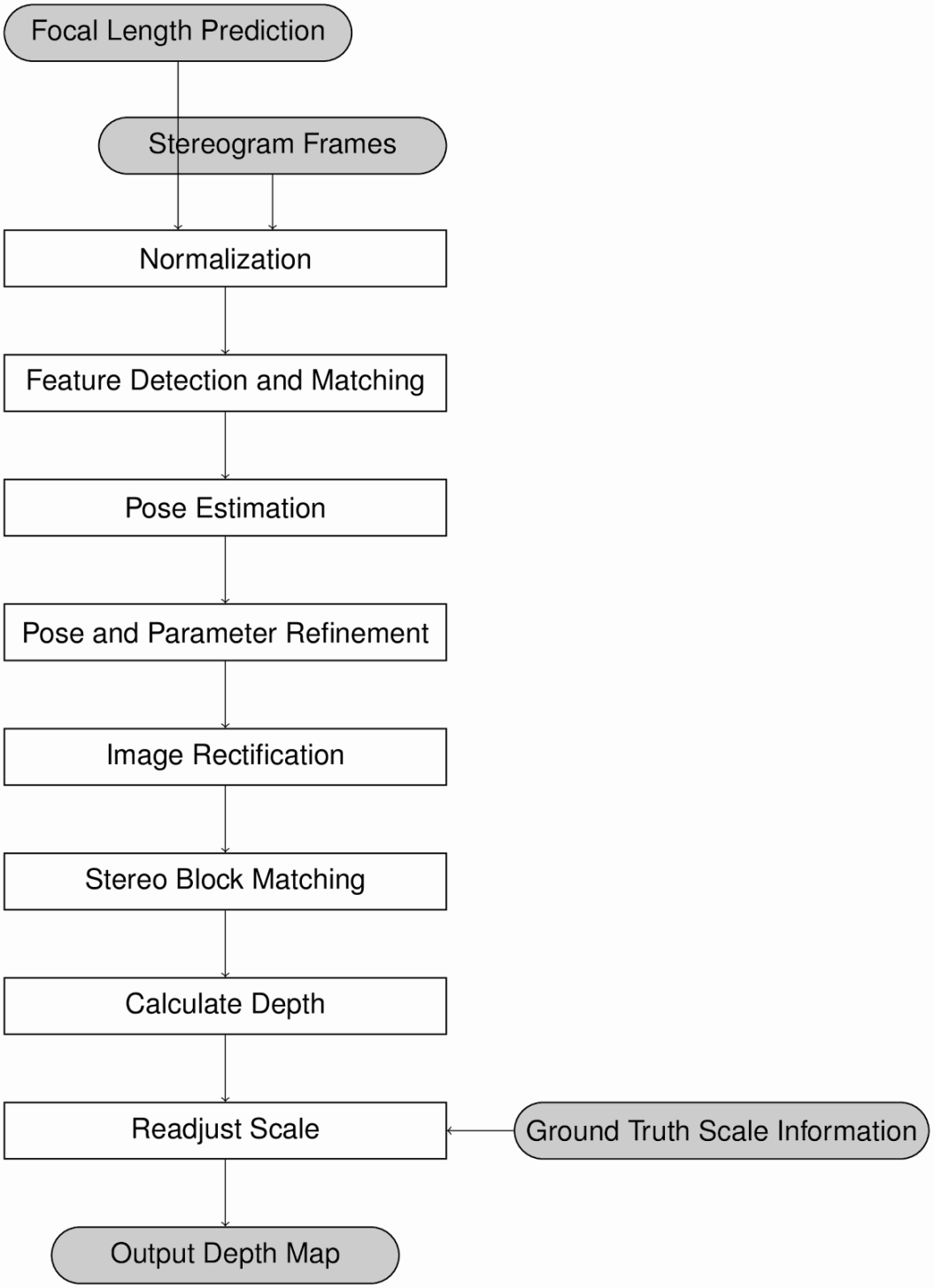


Figure 17: Our depth estimation pipeline.

Normalization

Before we do anything it is important that we reduce the scale of our image down to a reasonable size. This will help speed up the process, as well as blur noisy areas of our image.

Feature Detection and Matching

The first step of our pipeline is to find features. We start by using one of OpenCV's feature detectors on each of our image frames. We will likely use either ORB or AKAZE as they are fast, produce many features, and the feature vectors are invariant under rotation. After we have 2 sets of features, we can use a brute force matcher to find the top 2 matches for each point in the left image to points in the right image. We then apply the ratio test described in [20] to filter out points where the first best match is not unambiguously the best. The output of this phase is an $n \times 2 \times 2$ tensor of points, where n is the number of points. The second dimension selects between our 2 images, and the last dimension selects the x or y coordinate. A diagram of such a set of points is shown in fig. 18.

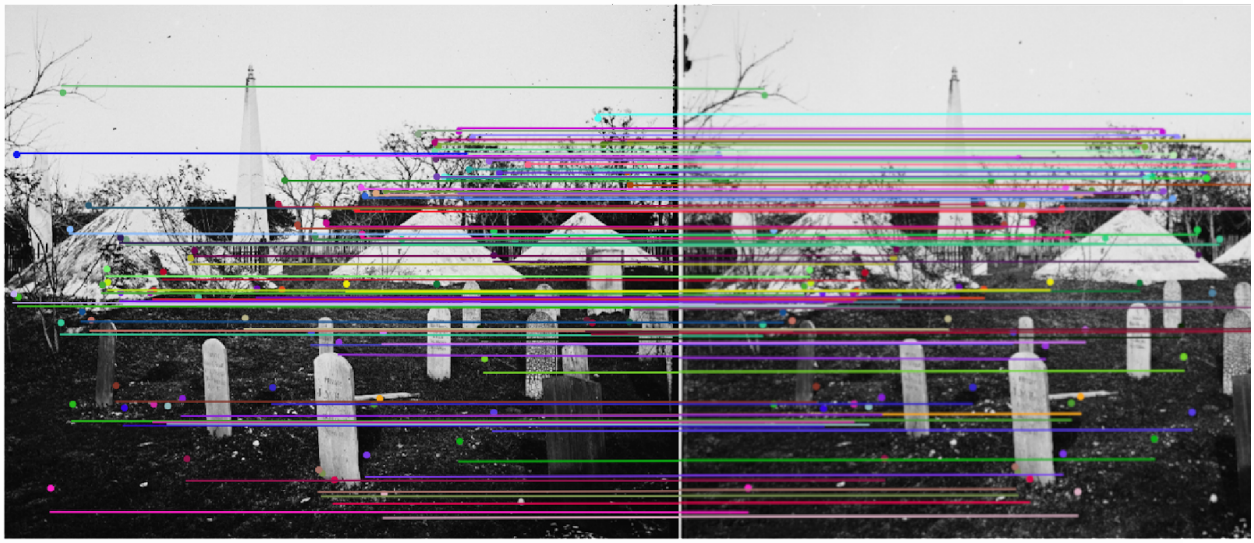


Figure 18: A sample showing feature matching on our target image, based on an early prototype of the feature matching step. The lines match features on the left to their best matches on the right. The longer the line, the further the closer the corresponding point is to the camera.

Initial Pose Estimation

Once we have our matching points, we can use them to estimate the pose of our stereo setup. To do this we will use `findEssentialMat()` passing it our set of matches from the

previous step and our initial estimates for the camera parameters. The focal length will come from the focal length predictor, and we will assume initially our image has no distortion or skew, and that the principal point is just the center of the image. While this initial estimate will most likely be inaccurate due to inaccuracy in these assumptions, our goal is just to get a good starting point for bundle adjustment to start optimizing. The output of this phase is the essential matrix, encoding the rotation and translation of our second camera relative to the first.

Pose and Camera Parameter Refinement

Now that we have initial guesses for both of our camera's intrinsic and extrinsic parameters, we can use bundle adjustment to hopefully improve their accuracy. The output of the previous step was the essential matrix,so we will need to first decompose it into a rotation matrix and translation vector before we can actually use it, using *recoverPose()*. Once we have all of our input parameters in a usable format, we can define our cost function, based on [21]. After that we can pass our cost function to `scipy.optimize.least_squares()`. This will use the Levenberg-Marquardt minimizer, determining the Jacobian using the finite difference method. The output of this function will be our refined camera parameters, including predicted distortion.

Image Rectification

After we retrieve our refined pose from the previous step, we can use it to rectify our images. Before we apply a rectification transformation, we can use the distortion values found by bundle adjustment to call `undistort()` on our input images. Next we can pass our camera parameters to *stereoRectify()*. This function will generate a good rectification transformation for us, preserving the aspect ratio of our images. The output of this step will be our newly rectified images.

Stereo Block Matching

Now that our images are rectified, we can use the `StereoSGBM` algorithm to generate a disparity map for our image. We can fine tune some of the parameters based on the matches we generated in the feature matching step. For instance, the value of `minDisparities` and `numDisparities` can be set based on the distribution of lengths between matching points. Other parameters will likely need to be set based on the image scale or other properties we might be unable to automate, so some user input may be necessary in this step. The output of this step will be a disparity map that we can use to compute depth, such as the one shown in fig. 19.

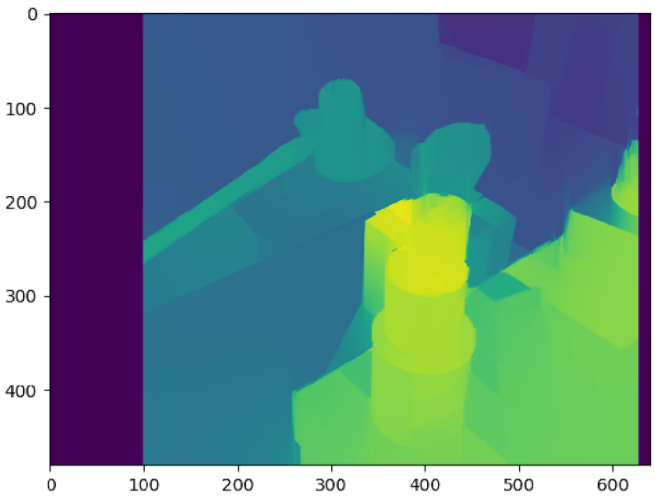


Figure 19: A sample showing the disparity result for a random image from the New Tsukuba dataset, generated by an early version of the depth algorithm. Colder / blue regions have lower disparity and are further from the camera.

Calculating Depth and Rescaling

Once we have the disparity information, we can convert it to a depth map using the relation we established in the mathematical model section above: $z = fb/d$. The baseline in these equations determines the scale of our scene, so we will need user input to fine tune it. Our initial estimate for the baseline can be 1 for simplicity. After we display the depth map to the user, they can provide ground truth length data by marking a line in the image and indicating it's length. The depth map will need to be rescaled based on the ratio of the real length to our estimate.

Testing

Datasets

In order to benchmark our pipeline, we first need a dataset. One option is the New Tsukuba Dataset [25]. Tsukuba is a popular stereo test dataset consisting of 1800 frames of stereo video, developed by the University of Tsukuba. This video is rendered from a 3D scene, based on a similar scene in a previous dataset. The image in fig. 20 is an example of what a frame in the Tsukuba dataset looks like. The fact that this dataset is simulated means that it does not have noisy images or validation data. This is not necessarily representative of our target stereogram, where there is plenty of noise. Additionally, the stereo camera setup is already rectified, therefore we cannot use it to test our pose estimation. There is also no distortion, and all the video frames come from a single scene with similar structure throughout. Due to all of these factors, Tsukuba is a

good sanity check dataset, but using it as a benchmark to predict our performance on our target image is probably a bad idea.

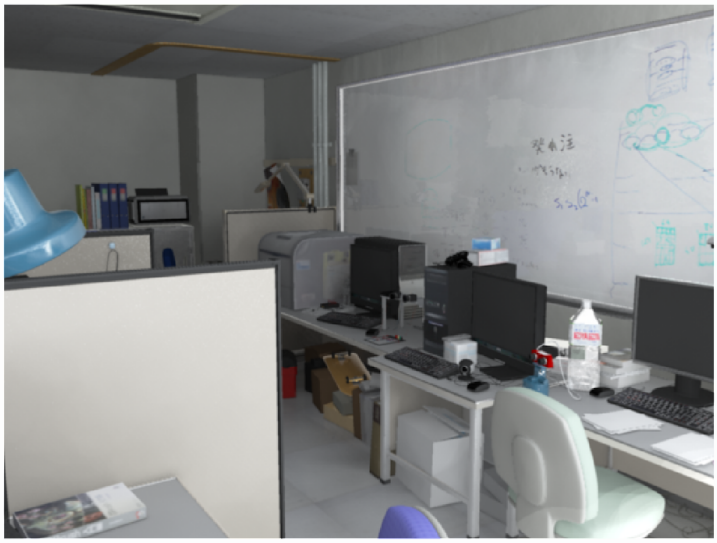


Figure 20: A sample frame from the Tsukuba dataset.

Another popular option is the KITTI dataset [26]. Unlike Tsukuba's dataset, the data here was generated while testing autonomous vehicles in the real world. The image in fig. 21 shows a sample of a more rural environment from the KITTI dataset. The sensor data was taken in a variety of different road environments. This ranges from more urban streets with several visible people and buildings, to quieter rural areas with trees and more complex terrain. This means that the images have the potential for noise and distortion, but this comes at the cost of accuracy for our ground truth data.

The ground truth data for KITTI was generated by a few other sensors onboard the autonomous vehicle platform. This includes a LIDAR sensor to generate depth data, as well as an accelerometer to determine changes in the vehicle's position. The cameras will need to be rectified and undistorted, but the camera parameters are likely to be relatively stable throughout the dataset. This makes it a good sanity check for our pose estimation, but not a good benchmark in this regard. Overall, KITTI is a lot more well rounded then Tsukuba, but might not give us a good idea of how our solution will behave in situations with strange camera pose or distortion.

Figure 21: A sample frame from the KITTI dataset. This is from the residential course of images, and features a very different scene from what you find in the Tsukuba dataset.

If we need to, we might even be able to build our own dataset to benchmark our pipeline. This will probably need to be simulated due to resource constraints. However, we can vary camera parameters in ways that the other datasets do not. We can also generate noise to see how our model behaves in that situation as well. This simulation would depend on having a scriptable render pipeline. 3D modeling programs like blender could allow us to render the data we need, but it will likely involve a lot of work developing the tools to do so.

Benchmarks

Regardless of what dataset we choose, we will need to have a metric to compare our results to the validation data and estimate our accuracy. KITTI's strategy involves checking each pixel against the validation data, and counting the number of pixels in our disparity map within 3% error. We can tune this parameter based on the kind of performance we would want from our solution. In our case, we want the model to be accurate to the last visible pyramid within 30cm, so once we have that measurement it will be fairly straightforward to set an error threshold.

Development Timeline

September

- Proof-of-concept feature matching completed
- Epipolar geometry research, focused on finding methods of estimating the fundamental matrix for features.
- Completed proof of concept for fundamental matrix estimation, rectification, and block matching.

October

- Refactor the existing code into a more stable form
- Begin researching details of math mapping disparity to depth based on camera pose.
- Initial pass at the design document, including rough outline without normalization or bundle adjustment.

November

- Validated the ability of the frontend to make measurements based on output depth map
- Initial benchmarking setup with the Tsukuba dataset. Does not test rectification.
- Start developing prototype implementation of bundle adjustment

December

- Working bundle adjustment prototype.

January

- Experiment with depth completion networks

February

- Finish implementing resizing based on user input, marking 1.0.

March

- Complete first phase of benchmarking and tuning.
- Stable output, frontend integration.

Conclusions

To review, our proposed solution is a pipeline that takes in some initial estimates for camera parameters, as well as the input stereogram and generates a depth map for each pixel. Each step in the pipeline either builds new structures we can use in later steps, or refines the results of a previous stage. By building off the existing wealth of knowledge in the field of structure from motion, including several helpful libraries, we can rapidly prototype and tune the different parts of the pipeline.

Our pipeline attempts to address several possible sources of error, from noise to distortion, to inaccuracy in our initial estimates of camera parameters. By normalizing our images we can reduce the effect of noise on our final results. Our estimates of the camera pose can be refined using techniques such as bundle adjustment and least medians optimization, allowing us to filter out outliers and hopefully end up with more accurate output down the line. Errors in our features matches resulting from untextured areas of the image, or speckle, or noise can be reduced by tuning the smoothing parameters.

By running benchmarks, we can keep track of how changing certain hyperparameters affects our solution's behavior. From this information, we can attempt to tune our parameters to increase the potential accuracy of our pipeline for our target scene. Even though we don't have ground truth image data for our actual target, we can still use the datasets we do have to predict the accuracy in our target image and tune our parameters accordingly.

User Interface

Research

We determined fairly quickly that we wanted to develop the user interface portion in Python. This was mainly so that it would be simpler to synergize with the machine learning and computer vision portions of the project which would also be completed in Python.

We then sought out a library that would facilitate quick development of a simple yet functional user interface. The main two libraries that stood out for developing a user interface in Python in our research were PyQt5 and Tkinter. Both libraries are fairly similar in their design, meaning they both are based around combining predefined widgets together to create an application. These include tools like buttons, checkboxes, labels, file dialogs, image canvases, etc. This style of user interface development seemed like a good fit for this project as being able to create a new widget, such as a Photo Displayer to plot points on an uploaded image, to fit our needs was an option, and many simple but necessary widgets were already included in both libraries. After a bit more research, we determined that PyQt5 would be the best choice.

This was for a few reasons. For one, some of our group members had previously worked with the C++ version of PyQt5, which largely includes many of the same objects and widgets as the Python version. Furthermore, PyQt5 provides what is called the QtDesigner module, which allows much of the user interface to be constructed by simply dragging and dropping components into place. This development process serves to remove having to write a large amount of boilerplate code and leaves the GUI team with the sole responsibility of determining how clicking on different parts of the user interface should cause it to respond. By enabling fast and efficient prototypes by being able to transform an interface mockup directly into code, we could focus the development process on more important aspects such as ensuring accurate model predictions and depth finding.

Another benefit in choosing PyQt5 over Tkinter was that the tutorials and documentation were more readily available to us with PyQt5, which allowed for us to learn how to use the library much faster than if we had chosen to use Tkinter. All of these advantages indicated that using the PyQt5 library would be the correct choice for how we wanted to design the interface. We needed to make a rough layout as quickly and easily as possible in order to get feedback from Dr. Giroux and figure out what tweaks the design

needed in the future. With its visual designer and robust documentation, PyQt5 streamlined this initial design process and proved to be the correct choice.

An additional thing we felt was worth researching regarding the GUI was what the best method for saving and loading data. Previously, members of the group had saved data in a java project using object serialization and deserialization. Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. We figured python would also have support for such a feature and indeed it does.

The pickle module is included in python and provides support for such features. Initially it was thought it would be best to serialize the entire GUI and load files as an entirely new window, similar to behavior that you would see in applications such as Microsoft Word. However, a few tests revealed that because of the nature of the PyQt widgets, instances of those classes were not possible to serialize via the pickle module. In an attempt to find a workaround PyQt's own QDataStream class was looked at but ended up only providing serialization support for lower level widgets. Later, an extension for the pickle module known as dill was attempted as a last resort but that also failed.

Finally, we ended up deciding that it would be best just to use pickle to serialize only the data we truly needed to save. These include but are not limited to the point list, vector list, and file paths for the images. Custom functions for restoring the state of the application from this data would need to be written which will add some complexity to the project, but it was deemed a worthwhile trade off for the smaller file size.

Time permitting, it would most likely be better to try to find a solution to the inability to serialize the entire GUI object, but in the interest of time it was deemed that only serializing necessary data would be the better choice in order to deliver the project in a timely manner.

In order to help users to manage their data more easily it was also decided that a custom file extension was to be used in order to store the data for the GUI. Truthfully it is just a simple byte stream and any file extension could be used as long as it didn't conflict with any existing ones. The file extension we decided on was ".SDFDATA" as there was no way for it to be confused with a file of any other type. Regardless, safeguards are in place to ensure that the only type of file the GUI will attempt to load is of our custom file type.

Installing PyQt5

To install PyQt5 on a Windows device, run the following command in the Windows command prompt:

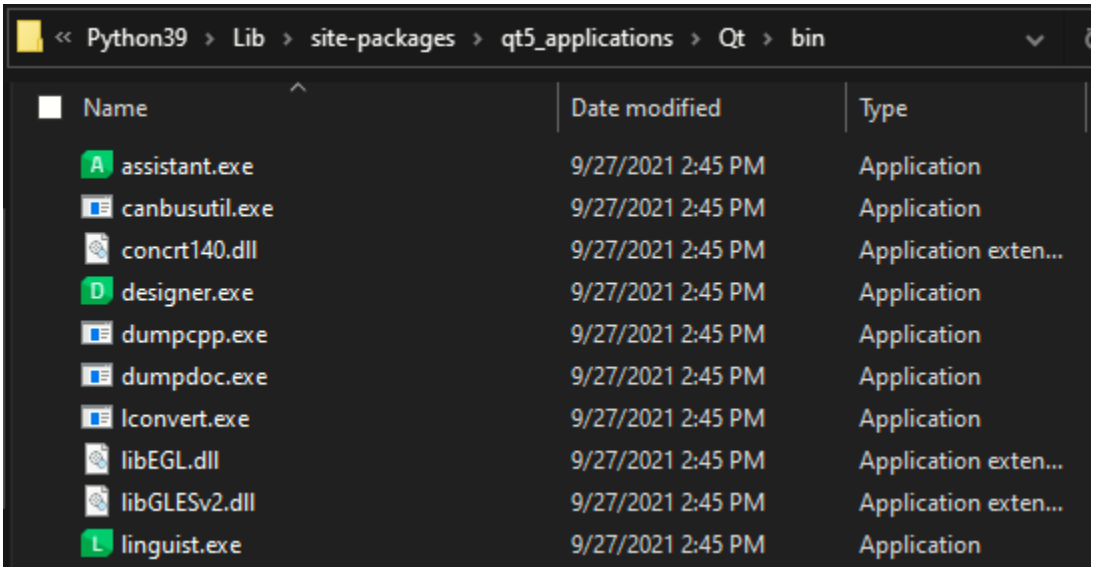
pip install pyqt5

This will install the PyQt5 library on a Windows device and allow Python programs using the PyQt5 library to be written. However, this initial installation does not include the QtDesigner module, a tool which factored heavily into our decision to use PyQt5 for this project. To obtain these tools on a Windows device, run the following command in the Windows command prompt:

pip install pyqt5-tools

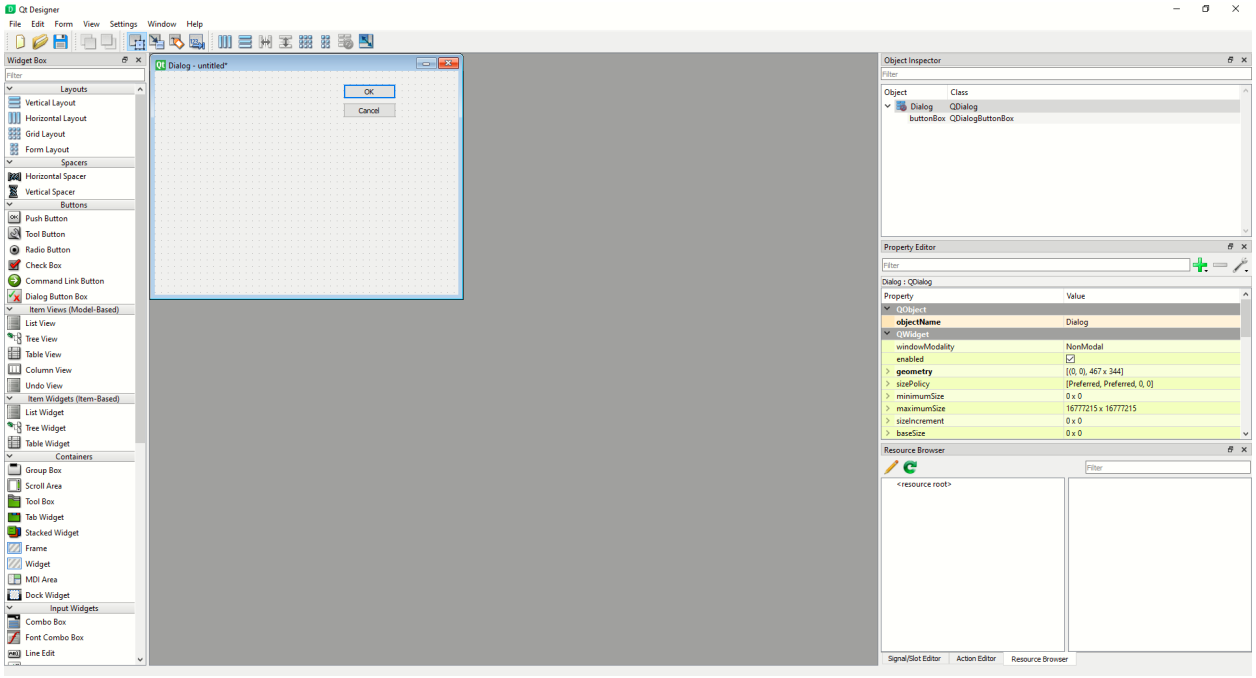
After that installation is complete, go to the local directory:

C:\Users\"yourUserName"\AppData\Local\Programs\Python\Python39\Lib\site-packages\qt5_applications\Qt\bin



designer.exe should be one of the files in this directory.

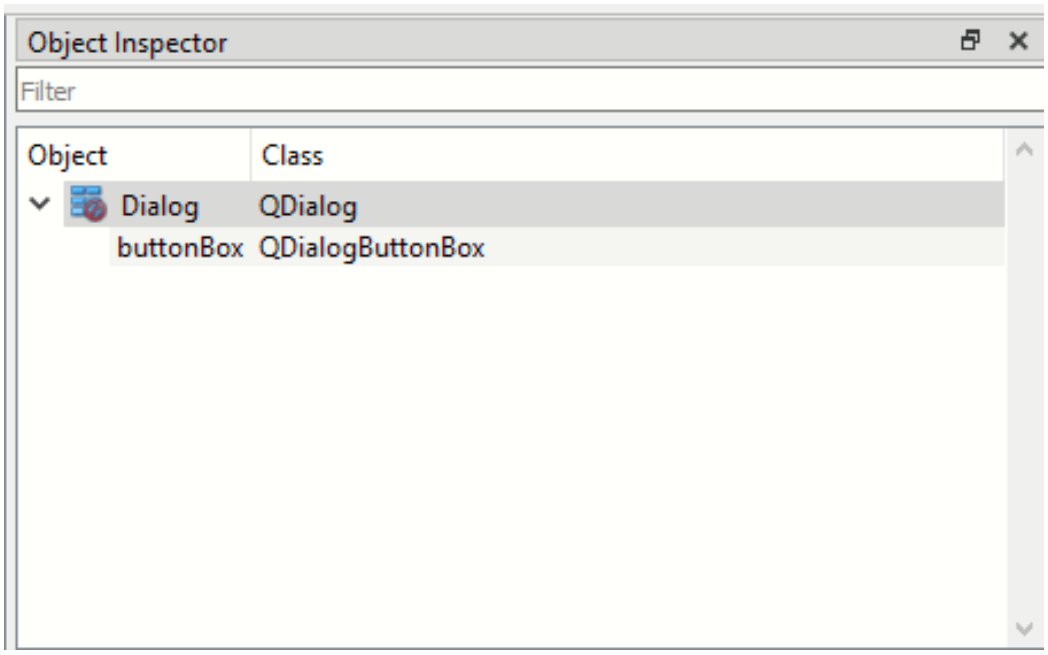
The QtDesigner module, named “designer.exe”, will likely be found at this address. Afterwards, the following window will be displayed when clicking on the application.



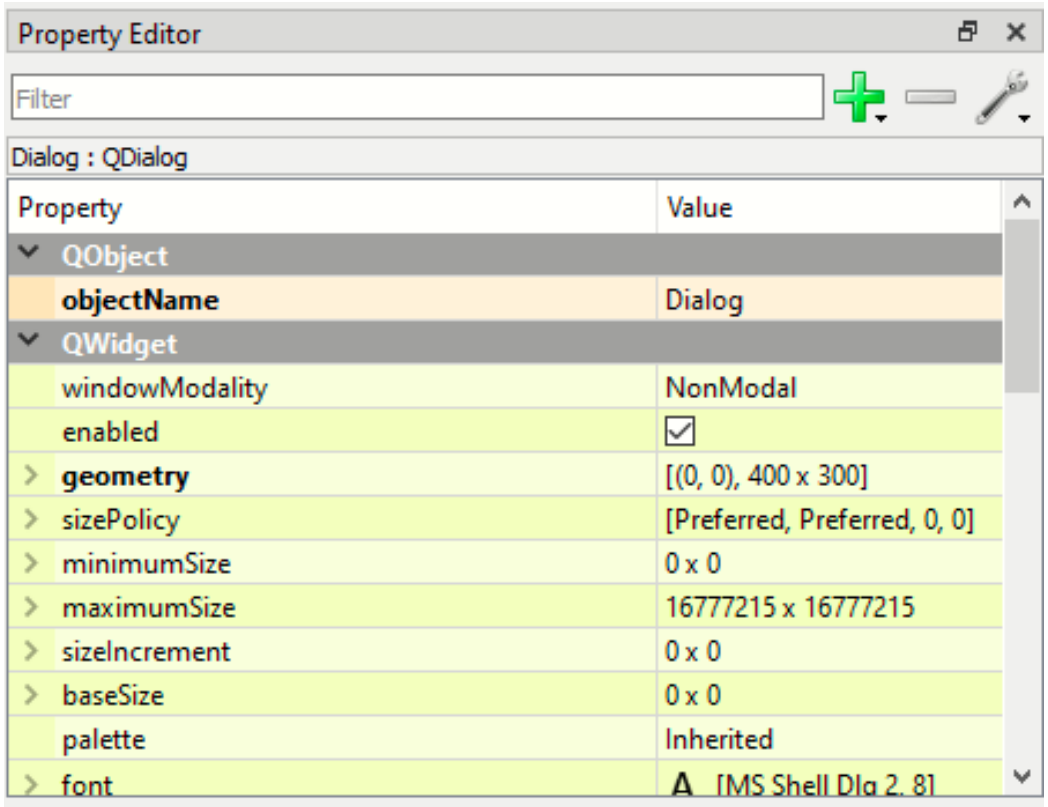
QtDesigner application.

The center panel on the screen is a preview of the window being designed. The left panel contains many of the different elements that can be dragged into the window. Included are various layouts, spacers, buttons, item widgets, containers, input widgets, and display widgets. The right-side panels provide greater property inspection for the objects inside of the window. There is an Object Inspector which provides a broad overview of all the elements in the window, and a Property Editor which displays all the property fields of a selected object.

The QtDesigner program is incredibly useful for developing an interface prototype quickly. Its drag-and-drop interface makes it easy to make a rough design sketch without a steep learning curve. But as we develop the project, we will spend more time writing code using the PyQt5 library than using this application because we will need to hardcode certain properties and functionality for custom widgets such as the Photo Displayer, which will be explained in greater depth below.



Object inspector for QtDesigner.

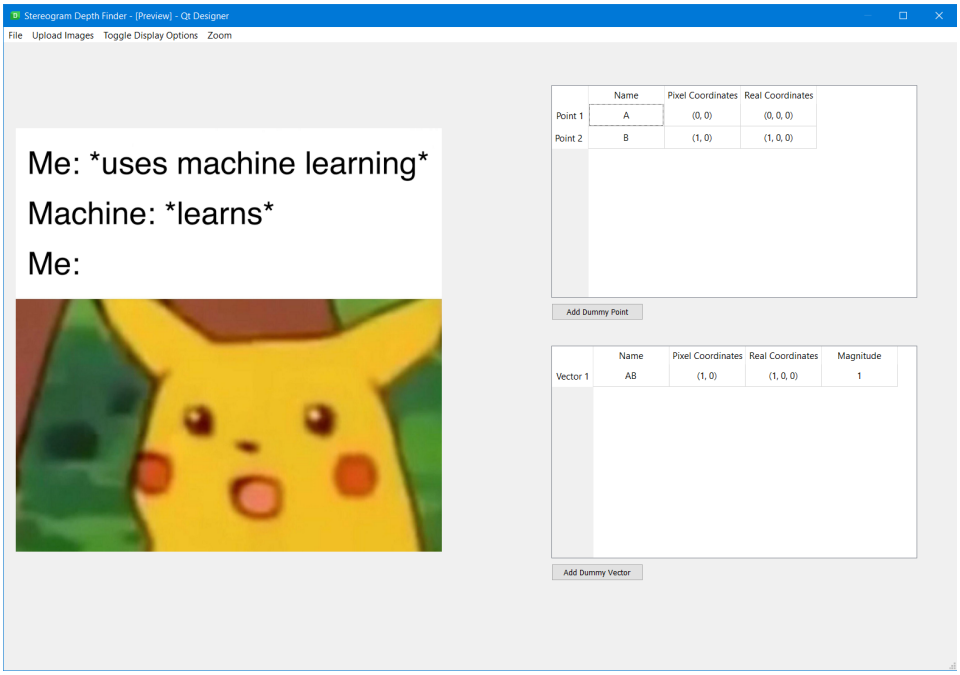


Property editor for a single object in QtDesigner.

Initial Design

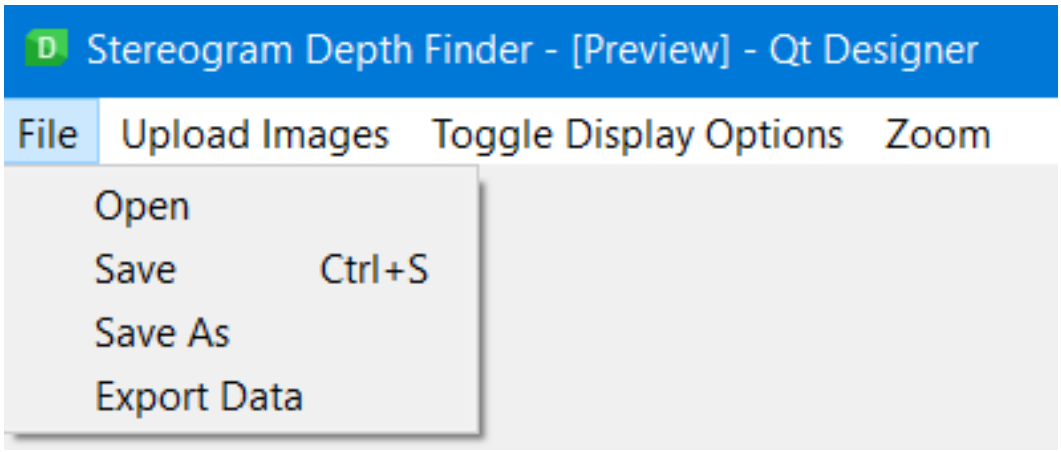
In designing the user interface, we determined that a simple barebones interface would suffice for this project as most of the important work is in ensuring accuracy of both the depth-finding algorithm and the focal length prediction model. We began with four initial menu bar items: The basic file item for opening and saving data, an item for uploading the left and right stereogram images, an item for toggling various display options, and an item for handling various zoom options. The initial design was quickly put together, with emphasis on proof of design concepts over aesthetic polish or finalizing the layout of the user interface. As a result, certain elements such as the dialog windows for entering new points and vectors and the photoDisplayer were not initially designed with a larger idea of how the interface should look in mind. However, this process worked for us because it allowed us to very quickly understand which parts of the user interface would be very easy to program and thus would not need as much attention in the future, such as the File Explorer upload functionality, and those which were not as simple as we initially thought and would need more attention, such as zoom capabilities. The idea was to quickly build a prototype and iterate from there as we came up with new ideas.

A menubar for selecting different actions, a section to display the image and allow the user to place vectors directly on the image, and a section to display the information related to the vectors were the core components of our design and an early mock up can be seen below:

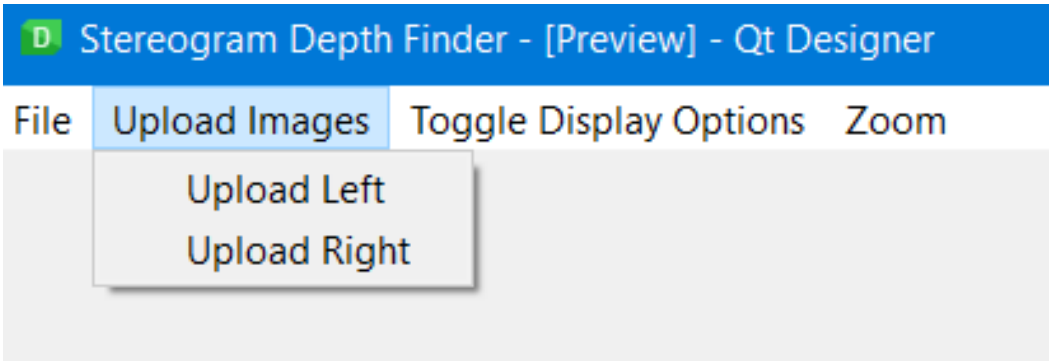


Early mockup of the GUI.

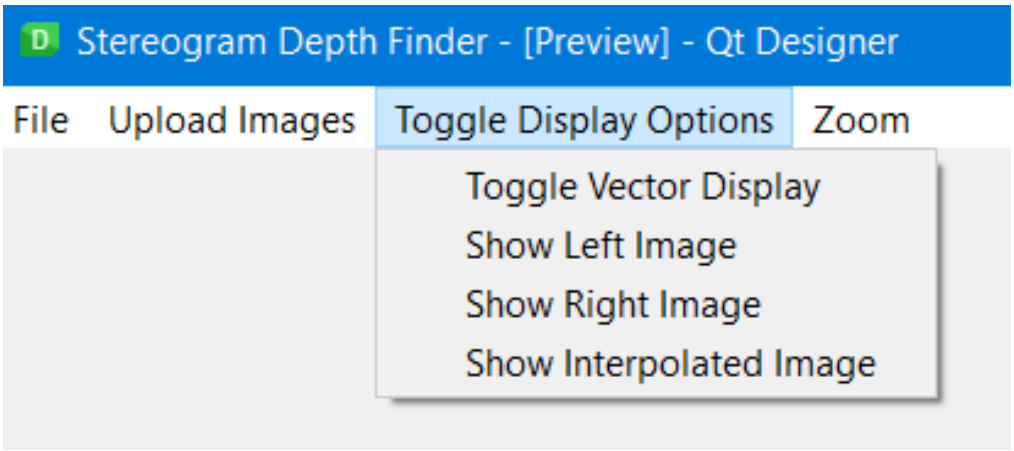
The options under each component of the menubar are as follows:



File options.



Upload options.



Display options.

This is the initial state for the Photo Displayer widget in the user interface.

Initial Photo Displayer state.

Without any uploaded images, the Photo Displayer widget defaults to a blue rectangle. For testing purposes, all the written functionality for the widget still works on this blue rectangle, in order to debug point and vector plotting. When the user clicks on any part of the blue rectangle, a red point is plotted on the displayer. For example:

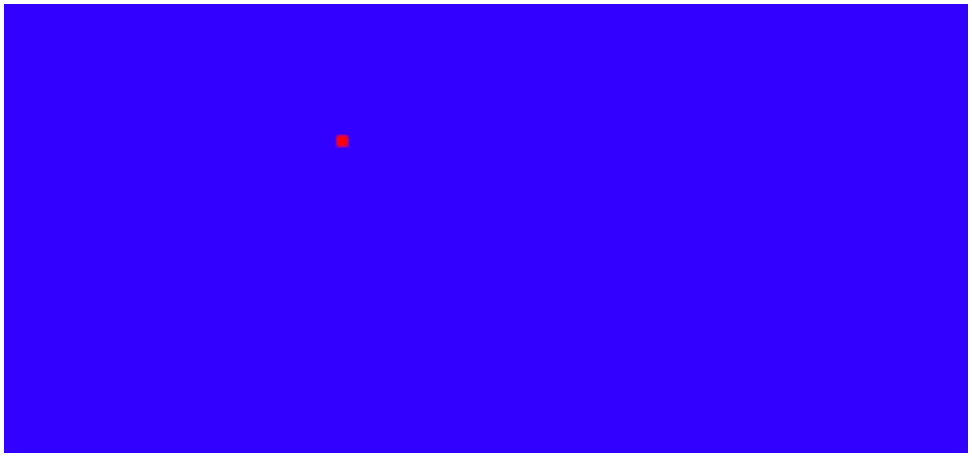


Photo Displayer with one point plotted.

When a second point is plotted on screen, the Photo Displayer draws a black line between the two points. This shows the vector relation between the two points, which will be entered into the appropriate tables on the right side of the interface.

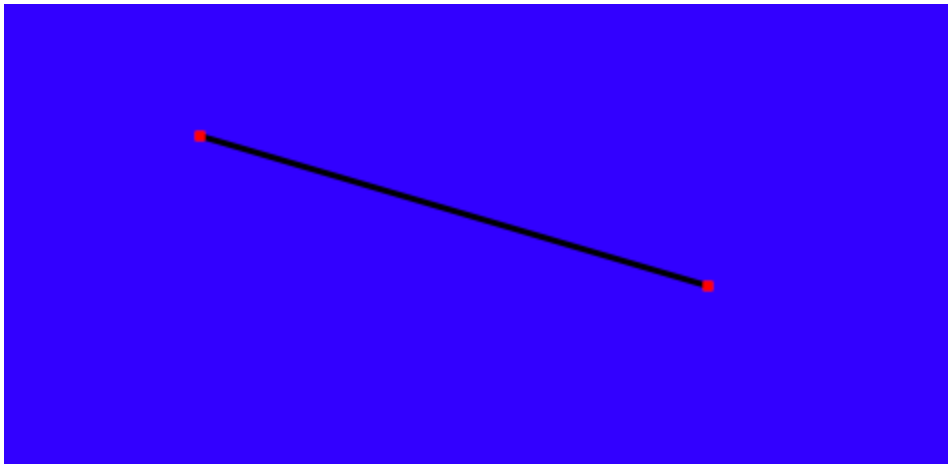


Photo Displayer with a vector plotted.

The next image demonstrates how the Photo Displayer widget plots points and vectors after an image is uploaded to the program via File Explorer.

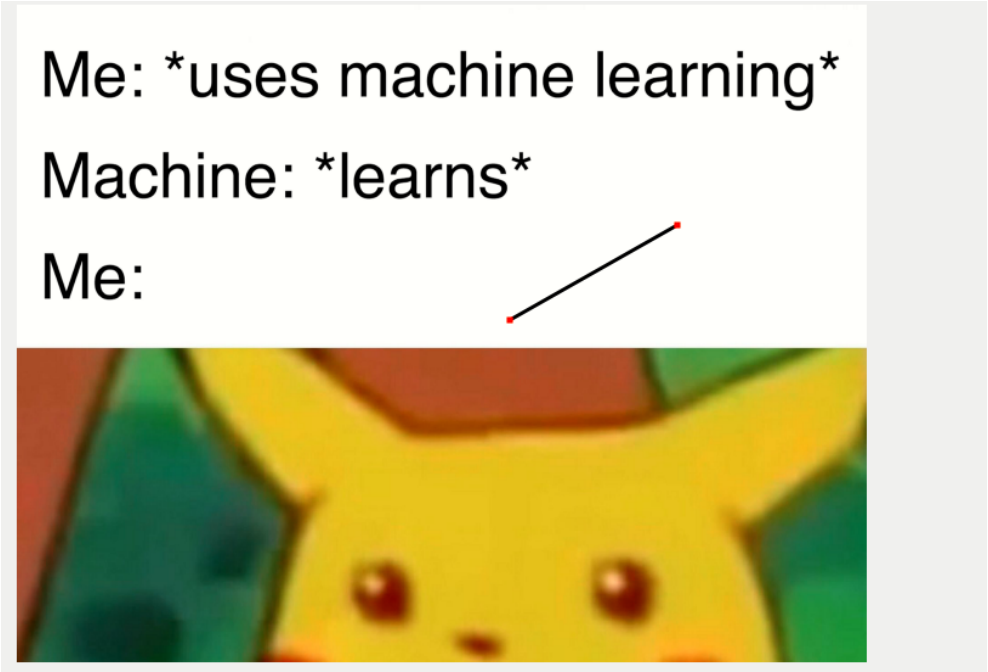
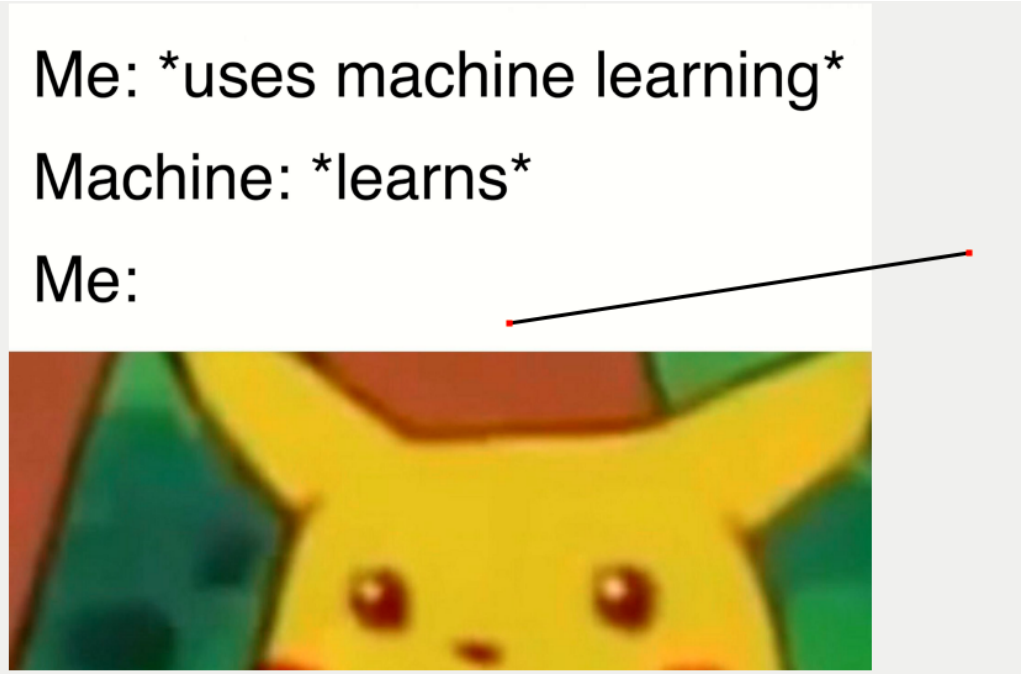


Photo Displayer with an image and vector.

One issue with the initial design of the Photo Displayer is that the size of the draw area did not adjust to the size of the uploaded image. This test image has a different aspect ratio and smaller width from the original blue rectangle, but as demonstrated in the screenshot below, a point could still be plotted outside of the image if the original area of the Photo Displayer contained that point.

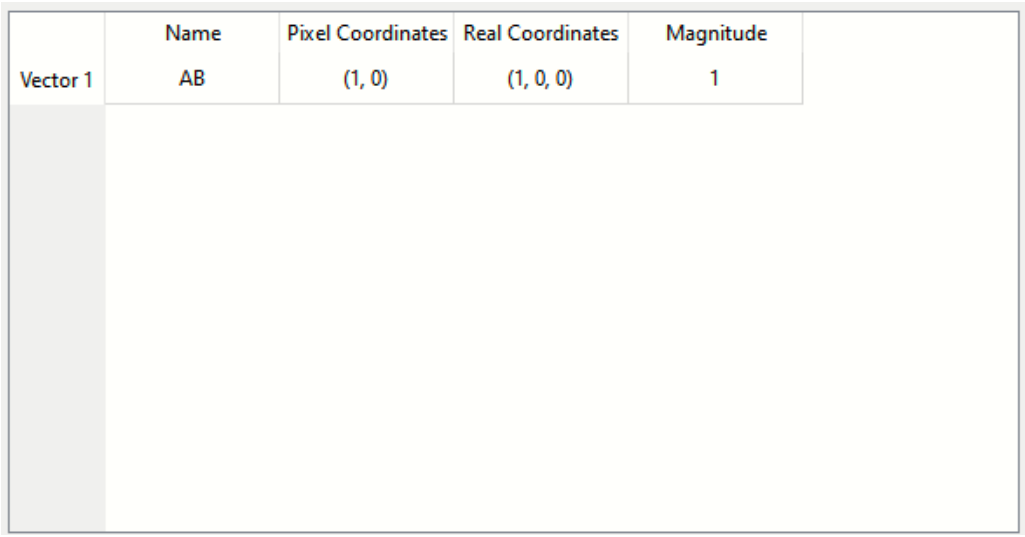


Bug showcasing plotting outside the image

This initial design cannot demonstrate the eventual zoom capabilities of the Photo Displayer. That function proved to be more difficult to execute than at first anticipated. The Photo Displayer needs to not only zoom in on the uploaded images, but also maintain the correct coordinates of the plotted points relative to the entire image, not just the section visible on the zoom. The next two images are initial designs for the point and table widgets:

	Name	Pixel Coordinates	Real Coordinates
Point 1	A	(0, 0)	(0, 0, 0)
Point 2	B	(1, 0)	(1, 0, 0)

Point table mockup.

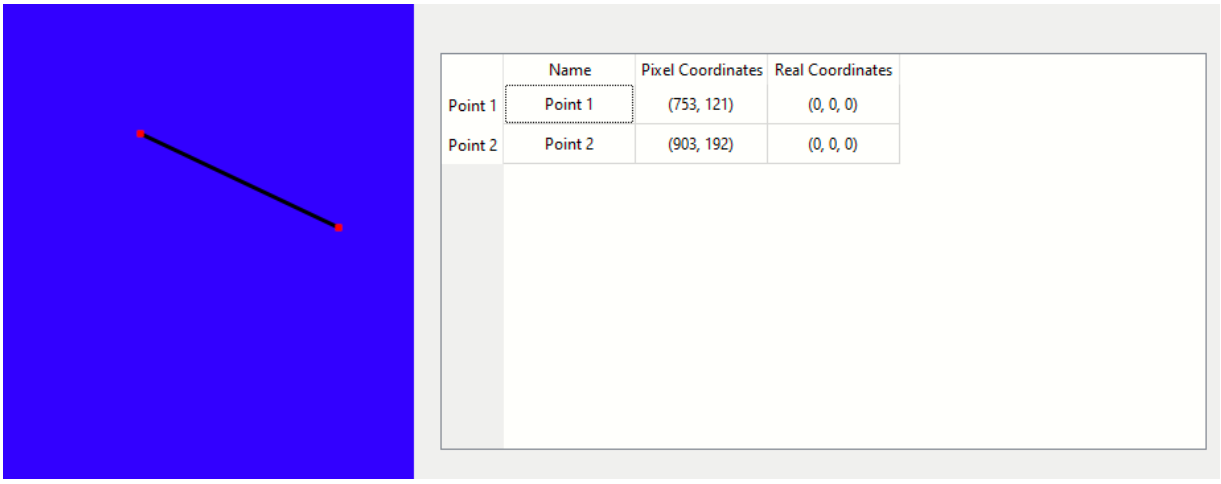


	Name	Pixel Coordinates	Real Coordinates	Magnitude
Vector 1	AB	(1, 0)	(1, 0, 0)	1

Vector table mockup.

The tables were initially filled with some placeholder values in order to test formatting correctness. For each point, the table displays its name, its pixel coordinate value, and its real coordinate value. The vector table displays the same properties for the vectors with an additional column for displaying the magnitude of the vector.

The following three images showcase the link between the Photo Displayer and the point and vector tables.

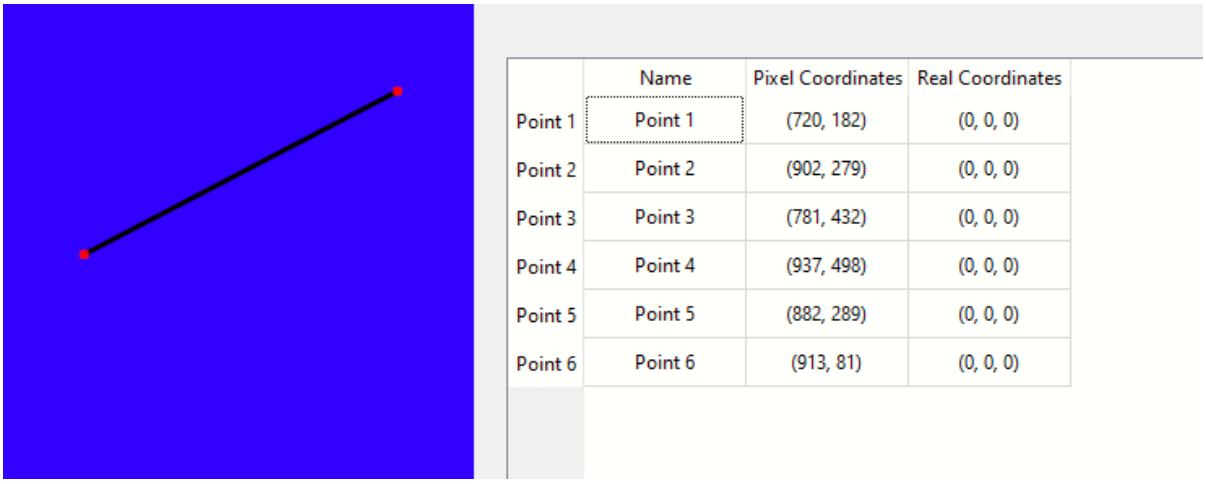


	Name	Pixel Coordinates	Real Coordinates
Point 1	Point 1	(753, 121)	(0, 0, 0)
Point 2	Point 2	(903, 192)	(0, 0, 0)

Showcasing of linked Photo Displayer and point table.

Whenever a point is plotted on the Photo Displayer with a mouse click, that point is appended to an array storing every plotted point and then added as a row to the point

table. In the initial design of the Photo Displayer, the first plotted point never changed. This meant that there are only ever two points displayed at any given time, with “Point 1” always remaining on display. But each plotted point was stored in an array and in the point table, so it remained in the program, as seen in the following image. Additionally, the “Real Coordinates” column entry for each point is automatically entered as (0, 0, 0). This is just because the real coordinates for each point are not yet being calculated when the points are plotted, so that column for the time being functions as a placeholder.



	Name	Pixel Coordinates	Real Coordinates
Point 1	Point 1	(720, 182)	(0, 0, 0)
Point 2	Point 2	(902, 279)	(0, 0, 0)
Point 3	Point 3	(781, 432)	(0, 0, 0)
Point 4	Point 4	(937, 498)	(0, 0, 0)
Point 5	Point 5	(882, 289)	(0, 0, 0)
Point 6	Point 6	(913, 81)	(0, 0, 0)

Another example of the Photo Displayer and table link.

The relationship between the point table widget and the Photo Displayer changed in a few ways throughout development. The automatic naming convention for points (names given to points that were not manually named by the user) was originally the same as that for the vertical header, as seen in the above screenshot: “Point” + the index of the point in relation to the rest of the list. The naming convention is now alphabetical, where the first point is named “A,” the second point is named “B,” etc. Once the uppercase letters are exhausted, the points are named “a,” “b,” etc.

Instead of only displaying two points at a time, with the first plotted point remaining one of the two in the Photo Displayer widget, every point remains in the Photo Displayer widget as they are created. Adding a point through the table will also draw that point to the screen. The Photo Displayer redraw function is triggered whenever the user makes a change to the widget by adding/removing/editing a point. Before, the redraw consisted of changing the second point being drawn to the screen and drawing a new vector line between the first and second points. Now, vectors are handled by the vector table widget and its corresponding vector array. This means that vectors are not drawn

automatically as the user adds points to the Photo Displayer, only when the user also adds a vector object to the vector list through the Photo Displayer.

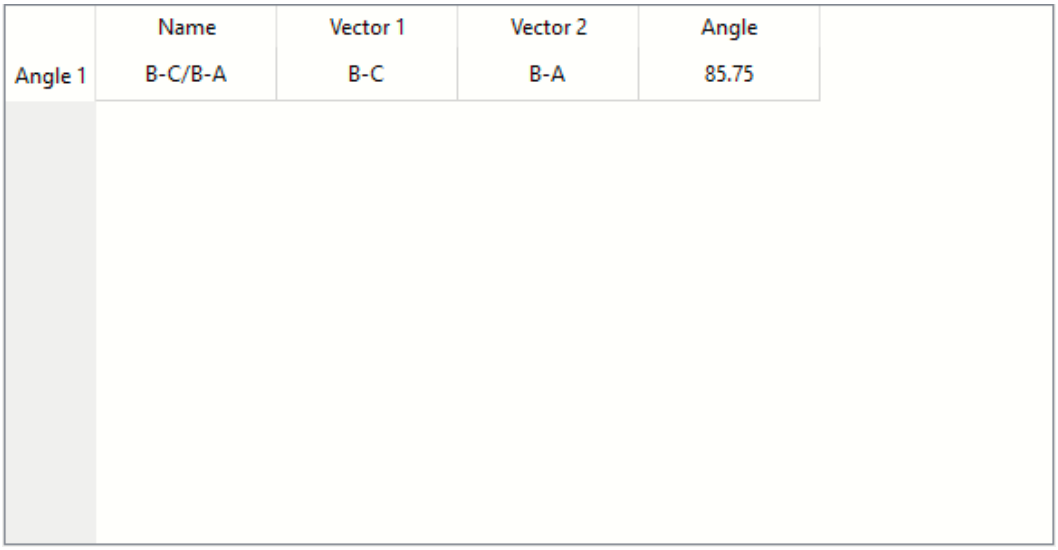
The next image shows the vector table when filled out after plotting on the Photo Displayer.

	Name	Pixel Coordinates	Real Coordinates
Vector 1	Point 1Point 2	(182, 97)	(0, 0, 0)
Vector 2	Point 1Point 3	(61, 250)	(0, 0, 0)
Vector 3	Point 1Point 4	(217, 316)	(0, 0, 0)
Vector 4	Point 1Point 5	(162, 107)	(0, 0, 0)
Vector 5	Point 1Point 6	(193, -101)	(0, 0, 0)

Initial vector table naming convention.

The automatic naming convention for vectors is to combine the names of the two points used to form the vector. However, as vectors are now only added through dialog windows by the user, the user has the option to give custom names to every vector as they are created. For the initial design of the vector table widget, analogously to the point table widget, there are true values put in for the “Real Coordinates” column, and instead each vector defaults to (0, 0, 0) for this section.

The following image shows the initial design of the angle table widget.



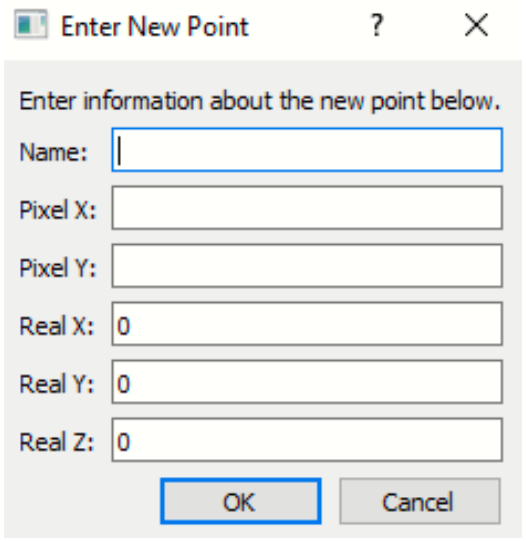
	Name	Vector 1	Vector 2	Angle
Angle 1	B-C/B-A	B-C	B-A	85.75

Angle table mockup.

The angle table displays the name of the angle object, the two vectors which form the angle, and the value of the angle in degrees. The automatic naming convention for new angles is the name of the first vector followed by a “/” followed by the name of the second vector. The angle object stores the full value of the angle float, but in the table widget the value is rounded up to the second decimal place for formatting reasons. The angle table widget does not need to display complex information about angles, in contrast to the vector and point table widgets which need to condense array and magnitude information into String-readable formats, because there is no further complex information about angles to show. With the angle table widget, everything from angle objects is displayed.

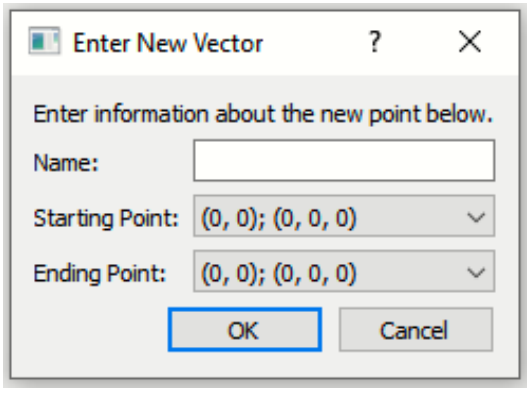
The only connection between the Photo Displayer and the angle table widget is that the Photo Displayer stores an array of angle objects that are also listed in the table, but there is no visual component to the angles in the Photo Displayer. Therefore, this image summarizes the full utility of the angle table widget without connection to a computer vision algorithm.

The initial design of the dialog window for adding points, vectors, and angles is shown below:



Dialog window for adding a point to Photo Displayer.

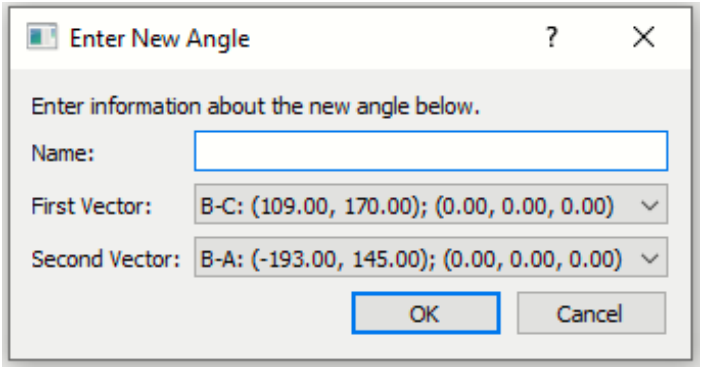
The Point Dialog window was initially planned out to take the above values as input as well as another textbox input row for the real coordinates. However, it quickly became clear that the real coordinates would be derived from the depth-finding algorithm and not from user input, so that row was deleted. The design of this dialog window has evolved the most as the overall function of the interface was developed, because entering points will primarily come from plotting on the Photo Displayer.



Dialog window for adding a vector to Photo Displayer.

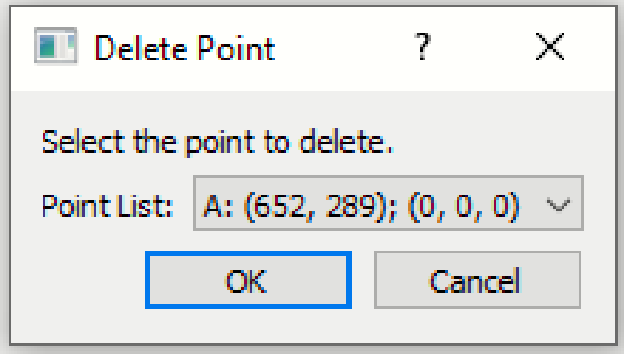
The Vector Dialog window was initially planned out to take text input for the points, similar to the Point Dialog, so the user could write any points to create a new vector object. But after looking at the other initial designs, it became clear that the user only

needs to select points that were already stored in the program, either in the point table or from the chosen points off the image itself. Besides this, all the dialog window needs is a text box for filling in the name so that the vector will be identifiable in the vector table. The dialog window will be used in tandem with vector line plotting on the Photo Displayer.



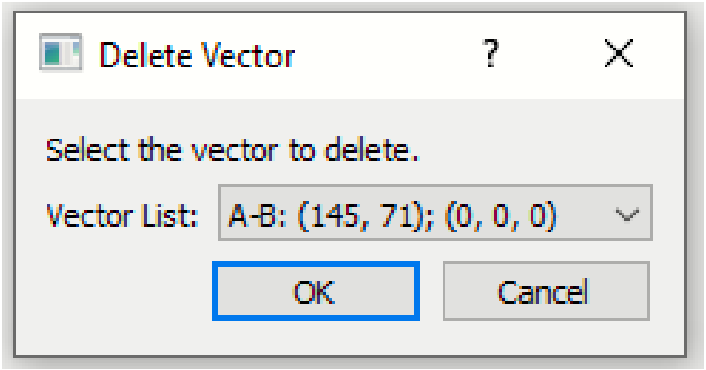
Dialog window for adding angle to Photo Displayer.

The Angle Dialog window has a very similar design to the Vector Dialog window. There is a field for entering a name, and following that two drop-down menus which both store all the vectors in the Photo Displayer. Once two distinct vectors are chosen, then the user can press the accept button and add the angle to the array of angles in the Photo Displayer.



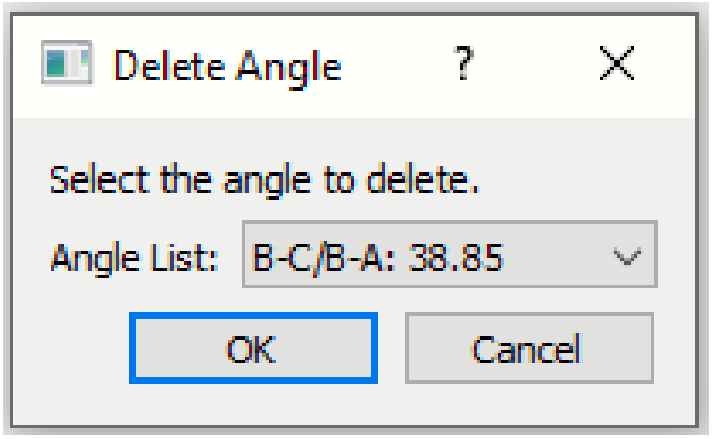
Dialog window for deleting point from Photo Displayer.

The dialog window for point deletion is simpler than the one for point adding, as all that's needed is a drop-down menu that lists all of the points in the Photo Displayer, i.e. stored in the points array of the Photo Displayer object. Each item in the QComboBox lists the name of the point, followed by the pixel coordinates and then the real coordinates. Having this descriptive information for each item is important so that the user can identify which point they are selecting as opposed to just seeing a name.



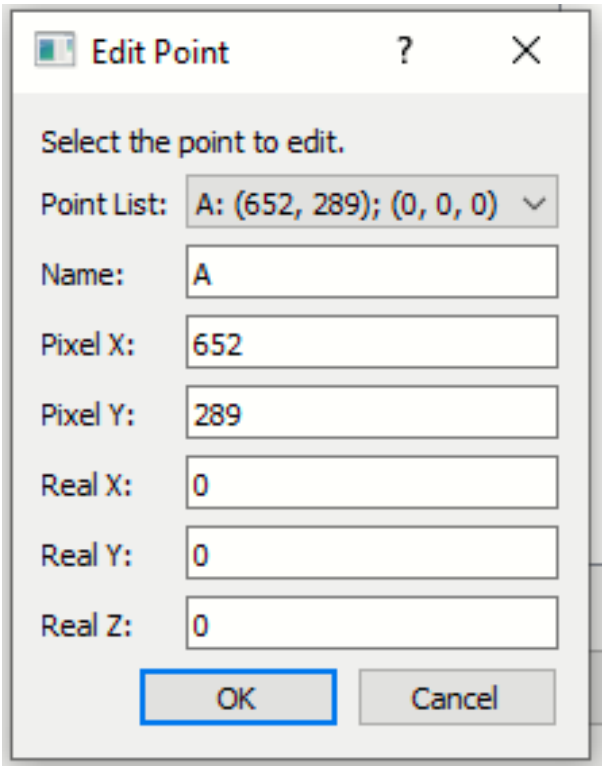
Dialog window for deleting vectors from Photo Displayer.

This dialog window for vector deletion has the same function and layout as the one for point deletion. Each item in the QComboBox represents a vector in the Photo Displayer's vector array, and each item lists the name of the vector, followed by the pixel coordinates and the real coordinates of the vector.



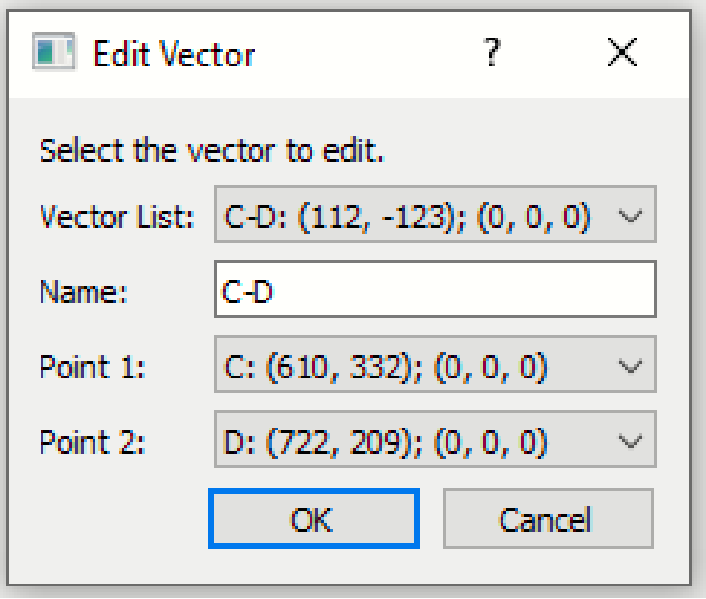
Dialog window for deleting angle from Photo Displayer.

The same layout and design principles apply for the dialog window for angle deletion. Each item in the QComboBox represents an angle in the Photo Displayer's angle array, and each item lists the name of the angle followed by the value of the angle in degrees. This value is displayed up to two decimal places in the dialog window for formatting purposes only; the full float value of the angle resides in the properties of that instance.



Dialog window for editing point in Photo Displayer.

In terms of layout and function design, the editing dialog windows are basically a combination of the adding dialog window and the deleting dialog window for that respective object. There is a single QComboBox that contains every point stored in the Photo Displayer's point array. This drop-down menu contains the same information about each point as did the one in the dialog window for deleting points. Whenever one of the items is selected from the drop-down menu, all of the text fields below are automatically filled with their respective properties from that object in the array. In the above screenshot of the dialog window, none of the QLineEdit fields for name, pixel coordinates, or real coordinates were filled in manually by the user, though the user can then edit those values and press "Ok" to modify that object with those new values. Like with adding a point through the pointDialog window, there are checks to make sure that the new values of the point do not equal those of an already existing point or exist outside the bounds of the image. Therefore, there is no way for the user to plot an invalid point using the interface of this program.



Dialog window for editing vectors in Photo Displayer.

The dialog window for editing vectors follows the same layout and function principles as the dialog window for editing points. There is a single QComboBox that contains every vector stored in the Photo Displayer's vector array. This drop-down menu contains the same information about each vector as did the one in the dialog window for deleting vectors. As opposed to filling text fields, as in the dialog window for editing points, this dialog window contains two additional QComboBox objects which each store all of the points in the Photo Displayer's point array. Whenever one of the vectors is selected from the first drop-down menu, the two points that comprise that vector are automatically selected in the bottom two drop-down menus. The user can then select other points to form that vector or change the name of the vector and press "Ok" to save those changes.

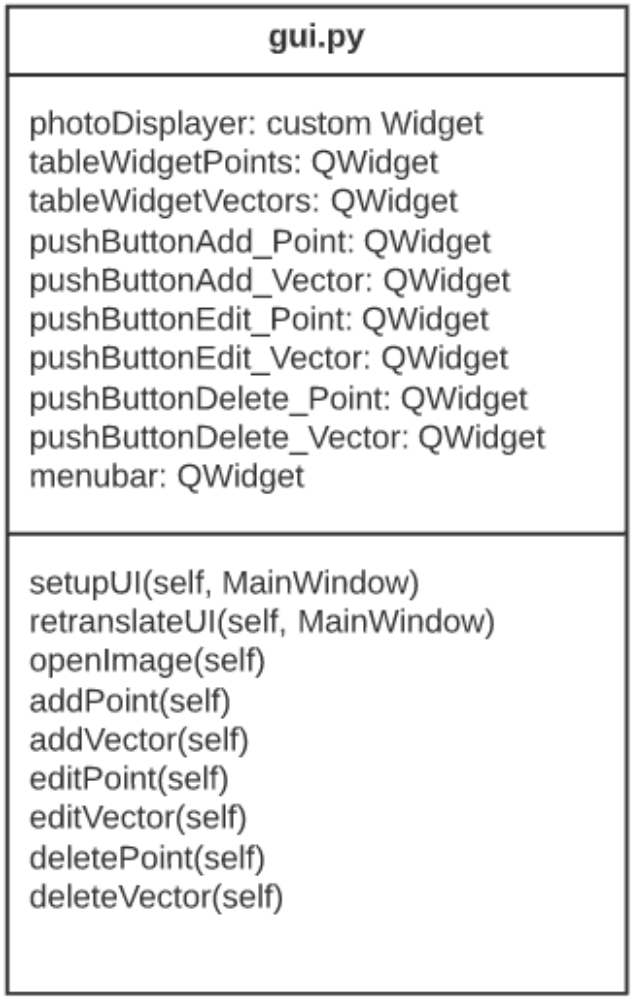
Like with the dialog window for adding vectors, there are a few conditions that will prevent the dialog window from accepting the edit. If the same two points are chosen to modify the vector, the dialog window will display an error message telling the user to select two distinct points. In addition, if the vector based on those two points already exists inside of the Photo Displayer's vector array, the program will display an error message telling the user to select different points to form a unique vector. There is no risk of the user forming a new vector if no points exist, because only launches if the point and vector arrays are not empty.

Feature Set Summary

- Save data (points and vectors plotted as well as color to render them in + selected images)
- Load data (see above)
- Change vector color
- Change point color
- Zoom in
- Zoom out
- Reset zoom level
- Switch between left and right images
- Add points via left mouse click on image
- Add points via data entry
- Add vectors via data entry
- Edit plotted points via data entry
- Edit plotted vectors via data entry
- Correct depth predictions based on user inputted data
- Automatic update to vectors when a point changes
- Delete points via combobox
- Delete vectors via combobox
- Compute angle between vectors
- Compute vector magnitude
- Antialiasing of drawn points and vectors

Implementation Details

GUI Class



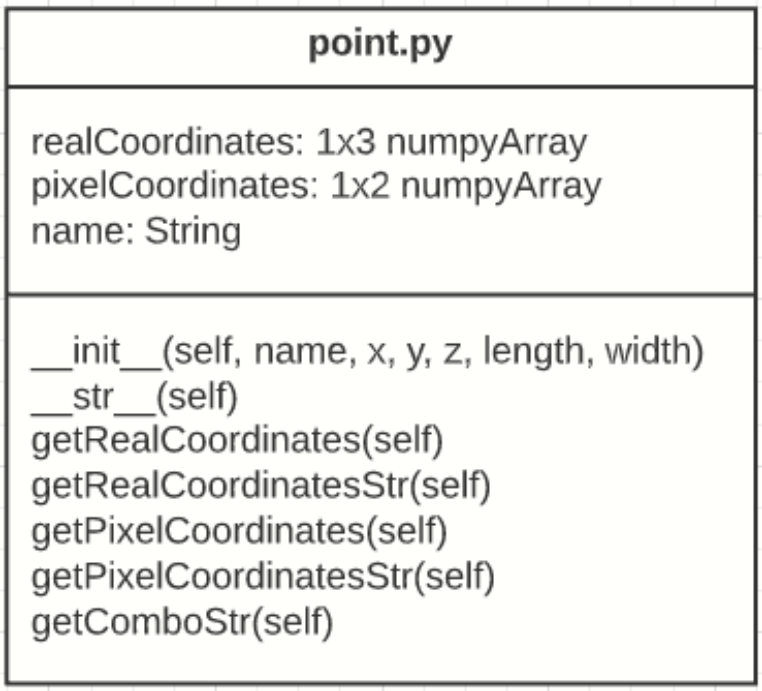
Ui_MainWindow class diagram.

The GUI is made up of a few parts. A main UI class that handles what is displayed on screen and handles different events, and a point and vector class used to handle the data that GUI will need to process. The GUI contains 3 main sections, the first displays the image for the user to click on and plot points. The other two sections show a collection of the points and vectors in table form. The data the GUI must store include the image paths, list of points, list of vectors, predicted focal length and depth map results. When designing a GUI with PyQt5 it is important to note that only the most common GUI elements, known as widgets, such as tables, buttons, combo boxes, etc., are natively supported in the library. In order to build a more complex widget, such as

the photo displayer that will allow the user to plot points, a new custom widget must be created. The detailed implementations of each class are listed below:

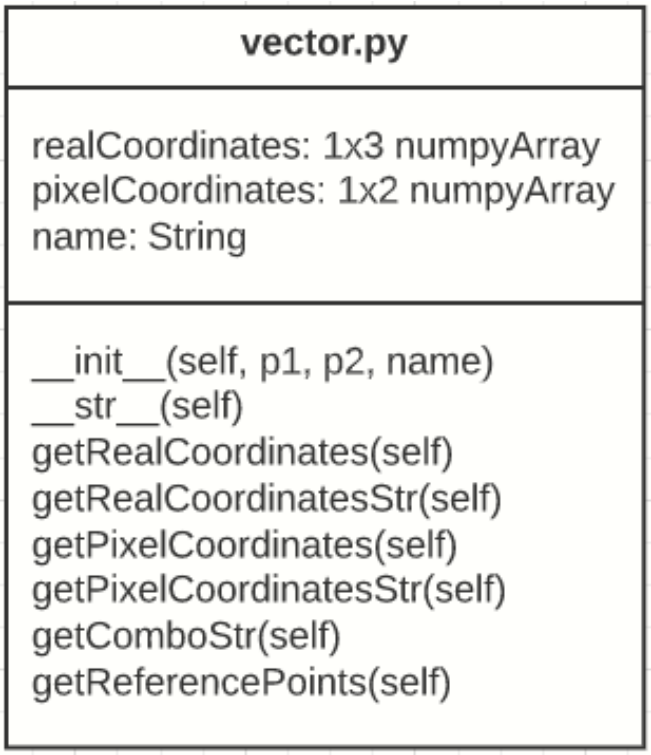
As you can see the GUI class is made up mostly of widgets along with some functions to handle the logic associated with each button press. The GUI class is simply made up of its constituent parts, and has some functions that change the state based on user input.

Points, Vectors, and Angles



Point class diagram.

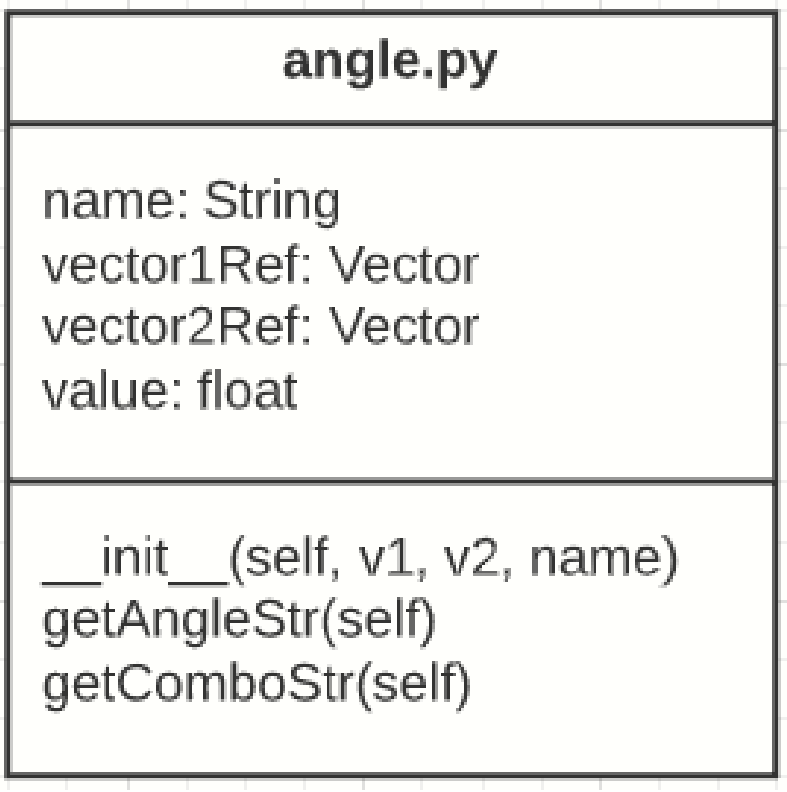
The point class constructor takes in six parameters: The name of the point; the real coordinates represented by x, y and z; and the pixel coordinates of the point represented by length and width. The real coordinates are stored in the 1x3 numpy array realCoordinates as [x,y,z], and the pixel coordinates are stored in the 1x3 numpy array pixelCoordinates as [x,y].



Vector class diagram.

These are essentially just wrapper classes for numpy array objects. They are convenient containers for working with both the real and pixel coordinates that we need to perform mathematical operations on. The reason for having these two classes rather than just working with pairs of numpy arrays is that if a transformation needs to be performed that needs to update both the pixel and real coordinates at the same time a single function can be used as designed within the vector or point class.

The point and vector classes each have three functions which return String objects containing properties stored in those classes. For instance, Both classes share the getRealCoordinatesStr(self) and getPixelCoordinatesStr(self) functions, which are what's called to display point/vector properties in their respective table widgets. They also share the getComboStr(self), which combines the two aforementioned functions along with the object's name into a single String object that is called in the QComboBox objects in dialog windows. For example, the two QComboBox objects in the vectorDialog window list the returned String from the point's getComboStr(self) function call.

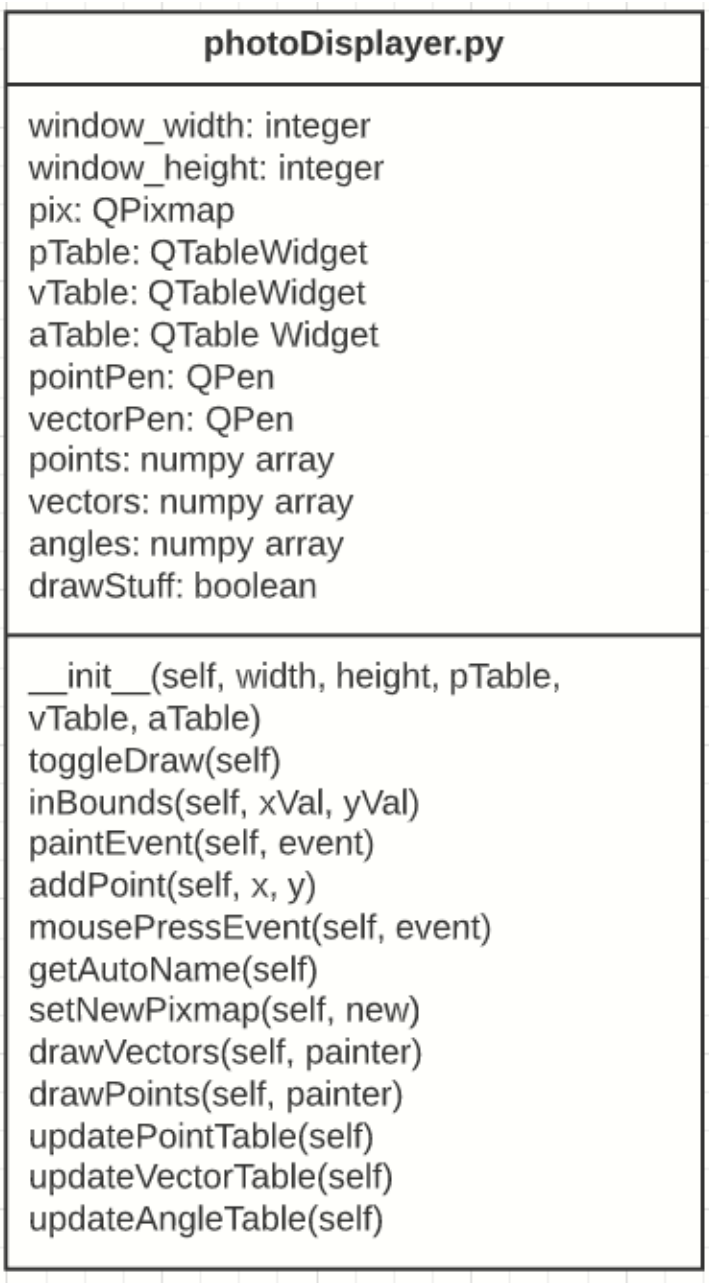


Angle class diagram.

The angle class, similar to the point and vector classes, is another convenient wrapper class that simply contains two vector objects and facilitates easy management of the data contained within. It also contains functions that allow the data to be converted into strings to show in the angle table.

For the constructor, the angle class takes in three parameters: The two vector objects which form the angle, and the name of the angle.

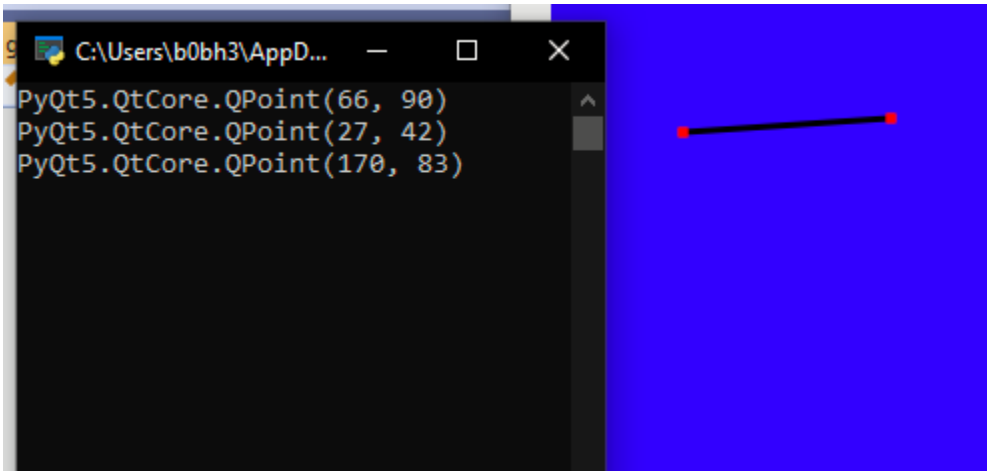
PhotoDisplayer Class



PhotoDisplayer class diagram.

The photoDisplayer class is perhaps the only truly complex part of the GUI as it is a custom widget that both displays an image and allows the user to plot points via clicking on the image. It is the main point of user data input and provides a very intuitive way for the user to select points of interest that they would like to use for their calculations.

The following image shows a test run on the Photo Displayer as it outputs the pixel coordinates of each plotted point.



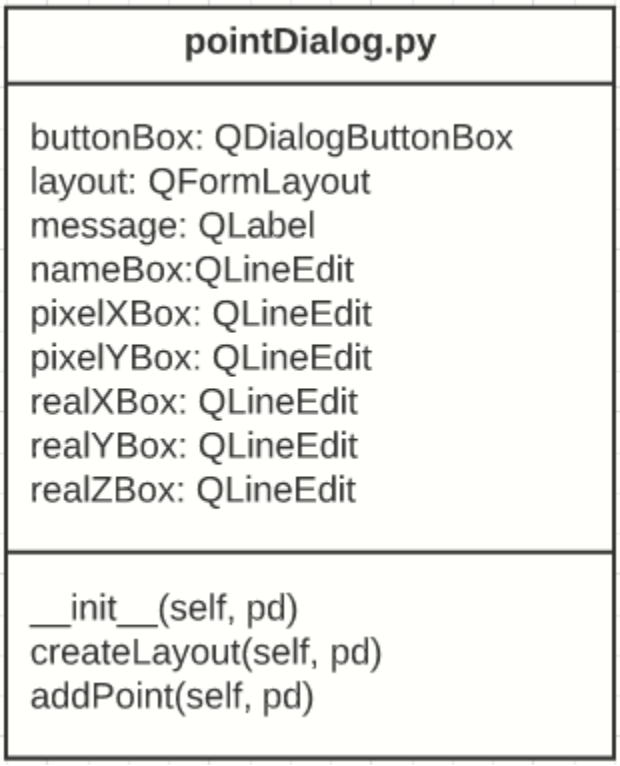
Point plotting data in terminal.

As seen in the above screenshot, when a point is plotted on the photo displayer, a QPoint object is created with the absolute coordinates of the point in the image. From that QPoint creation, that point is added to the point table in the user interface.

It is important to note that the photoDisplayer class stores three numpy arrays, for points, vectors and angles, for the entire user interface. It was necessary to have these three arrays, and it was only a matter of where they should be stored. Ultimately, it made the most sense to house them in the photoDisplayer class, mainly for two reasons: Because it already has access to the properties of points that were just plotted, and because those points need to be immediately added to the point table widget (and the vector and angle table widgets when necessary). Therefore, it was the simplest option to initialize and modify these arrays in the photoDisplayer class itself, rather than passing them in through the main GUI class.

The point array and point table will then be updated at the same time whenever a new point is plotted on the Photo Displayer. This is not, however, the same with the vector array and vector table. We do not want the plotting of two points on the Photo Displayer to automatically add a vector to that vector table because two points plotted consecutively does not by default mean that those points are meant to be endpoints of a new vector. Therefore, the Photo Displayer will not deal with all functionality relating to the point and vector arrays, and some of those duties will be allocated to other widgets that will be explained below.

There are various other functions in the Photo Displayer. The three update functions are called whenever one of the three numpy arrays does not match correctly with their corresponding table widgets. This is always the case when a point, dialog, or angle object is added to their respective array through one of the respective dialog windows. The inBounds(self, xVal, yVal) function is used to catch an inappropriate mouse event. It checks if a click from the mouse position xVal and yVal is within the confines of the image. The two drawing functions iterate through the point and vector arrays and draw them to the photoDisplayer widget if the drawStuff boolean is set to True.



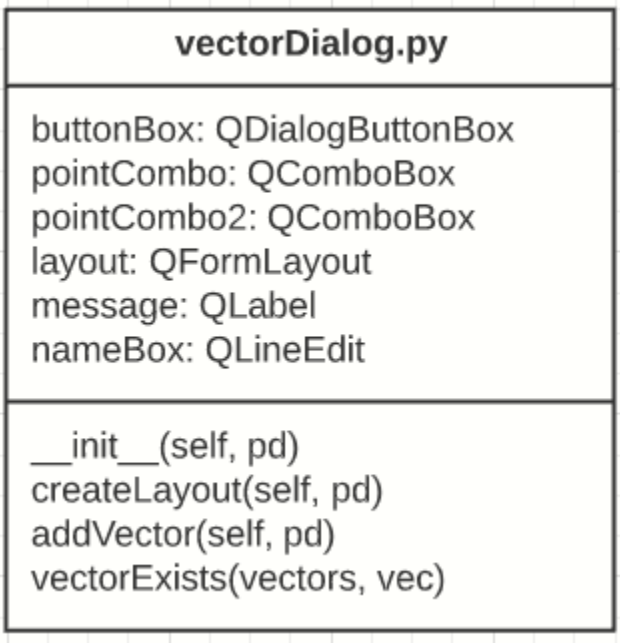
pointDialog class diagram.

To better organize the various dialog windows we would create, we decided to use a common format that would be applied to all of them. All the dialog functions contain at least three of the same functions: The initialization function __init__(self), the layout initialization function createLayout(self), and the function specific to the purpose of that dialog window, such as addPoint(self), addVector(self), etc.

For the dialog window specific to adding points to the Photo Displayer, the unique widgets mainly consist of QLineEdit objects for entering the numerical properties of a new point. The three QLineEdit objects which represent each property of the point's real coordinates are filled with 0s by default. In the development stage of the program, there

was no computer vision integration to actually determine what the real coordinates could be, so while these fields acted essentially as dummy properties, they were filled with 0s. The two text fields representing the pixel coordinates are not automatically filled and need to be filled in order for the form to accept correctly.

In the addPoint(self, pd) function, the input in each QLineEdit field is cast to a type int variable and passed into the parameters of a new Point constructor. Then, the point is added to the point array inside the photoDisplayer object pd. The updatePointTable(self) function is then called on pd to update the point table widget and repaint if necessary.



vectorDialog class diagram.

The vectorDialog window is slightly more complicated than the pointDialog window as it includes two drop-down menus containing points from the point table. When submitted, the class will create a new instance of the Vector class with the information from the chosen points and add that vector to the new vector table.

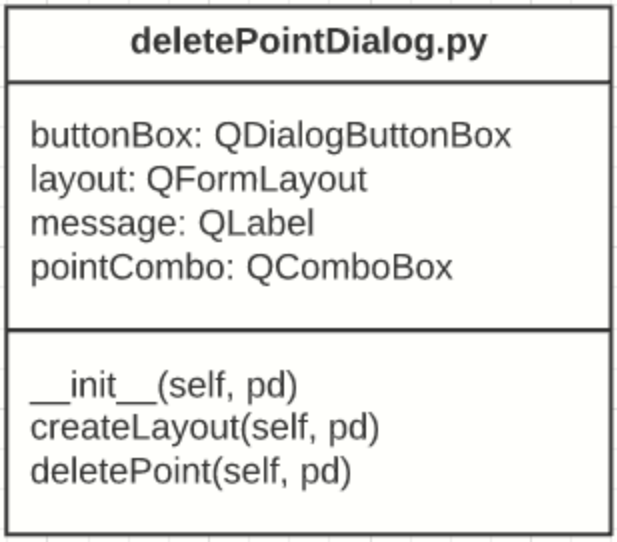
The vectorDialog window will be more prominent in the user interface's use than the pointDialog window. This is because, generally, points will automatically be added to the Point Table widget through the custom Photo Displayer widget. Therefore, a user is less likely to rely on the pointDialog to add points to their respective table. However, we do not want the plotting of two points on the Photo Displayer to automatically add a vector to that vector table. Two points plotted consecutively does not automatically mean that those points are meant to be endpoints of a new vector. Therefore, the vectorDialog

window will be more commonly used because it is needed to choose which two points should be connected into a vector and added to the vector table.

The vectorDialog has a few conditional statements that will stop the add action from proceeding if they are not all met. If the same point is selected as the start and endpoint of the vector, an error message is displayed in the dialog window. A vector also cannot be added to the program if the drawStuff boolean inside of photoDisplayer object pd is set to False. Lastly, the vectorDialog contains a static method vectorExists(vectors, vec), which checks whether a vector already exists inside of an array of vectors. If the newly formed vector already exists inside of the vector array of pd, i.e. the same points in the same order were selected to make the vector, then that vector will not be added to the Photo Displayer. If all of these conditions are met, then the new vector will be appended to the vector array, and the vectorDialog will call the appropriate update functions.

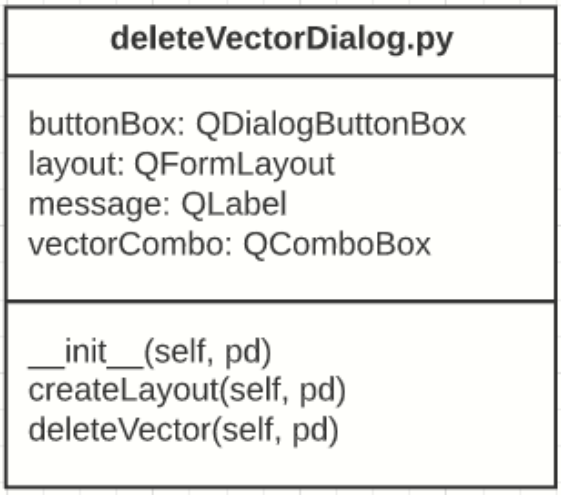
angleDialog class diagram.

The angleDialog class works with vectors very similarly to how the vectorDialog class works with points. There are two QComboBox items from which the user chooses the two vectors they want to form the new angle object. The angleDialog window alerts the user if they tried to use the same vector twice to form the angle as well as if the angle already exists in the angle array of pd via the angleExists(angles, angle) function.



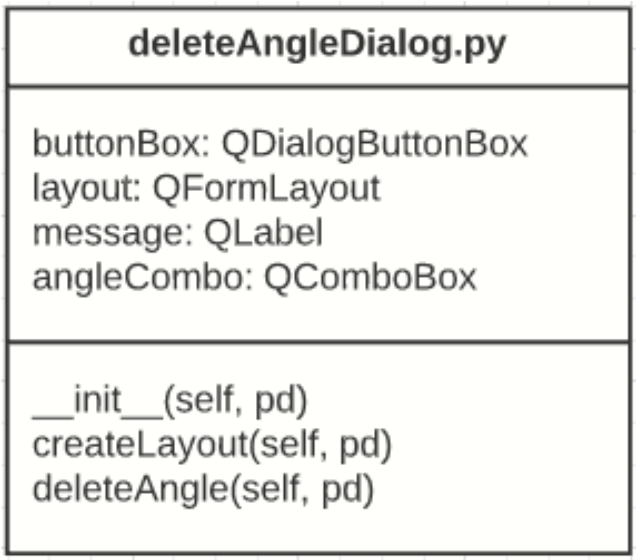
deletePoint dialog class diagram.

The deleting dialog windows have the same design principles as the adding ones. There are three main functions: One to initialize the window, one to initialize the form layout of the window, and another to delete the object from its respective array in the photoDisplayer object pd.



deleteVectorDialog class diagram.

The deleteVectorDialog class has the exact same design as the deletePointDialog class. The only difference is that the QComboBox displays information from the vector array of pd rather than the point array.



deleteAngleDialog class diagram.

The deleteAngleDialog class follows the same principle. Now, the angle array of pd is accessed instead.

The delete dialogs follow the same format: The form consists of a single QComboBox with a list of every point/vector/angle stored in the Photo Displayer pd. Once a point, vector or angle is selected, the user can press the accept button. Then, the chosen point, vector or angle is deleted from the list. and the corresponding table widget and is updated to reflect the change. For points and vectors specifically, the Photo Displayer is also redrawn to reflect the change.

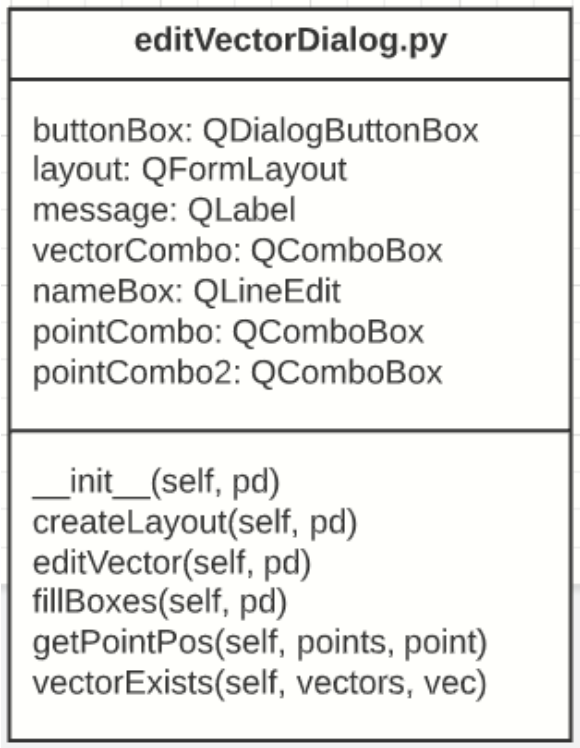
In a previous design of the delete dialog windows, the deleting function checked if the corresponding array was empty and stopped the update from going forward if it was. Now, the dialogs are called from gui.py, with a conditional statement checking if those arrays are empty before opening the correct dialog window, so this check was eliminated from the specific dialog windows.

editPointDialog class diagram.

In terms of layout and function design, the editing dialog windows are basically a combination of the adding dialog window and the deleting dialog window for that respective object. The editPointDialog class thus is a combination of the pointDialog class and the deletePointDialog class. There is a QComboBox object that stores information on every point in the photoDisplayer object pd. There are six QLineEdit objects to edit the properties of the chosen point.

There is an additional function, fillBoxes(self, pd), which is connected to an index change in pointCombo; whenever a new point is selected in the QComboBox, that point's properties are filled into all of the QLineEdit objects.

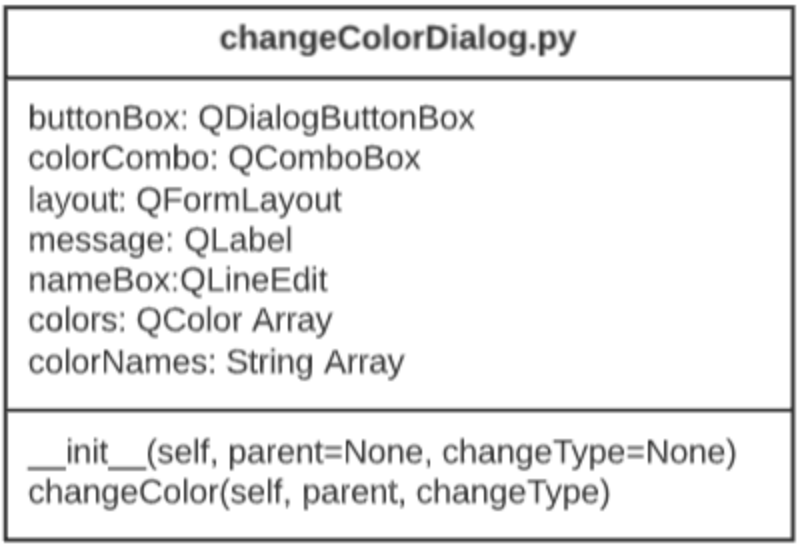
The editPoint(self, pd) function contains the same checks as the addPoint(self, pd) function in the pointDialog class.



editVectorDialog class diagram.

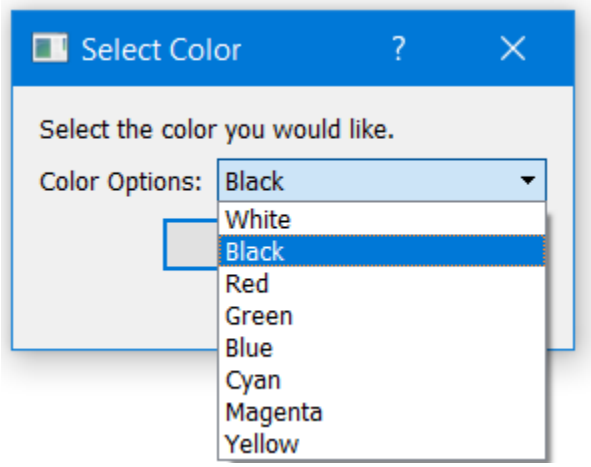
The editVectorDialog class was designed similarly to the editPointDialog class. There is the class initialization function, the layout creation function, and the function for editing the vector which is called when the accept button is clicked. There is also a fillBoxes(self, pd) function which fills in the two pointCombo objects with all of the points in photoDisplayer pd, with their current index being set to the index of the points which make up the vector in the point array.

There are two additional functions in the editVectorDialog class. The getPointPos(self, points, point) function takes in a point array and a point object and returns the index of that point in the point array. This is necessary to set the current index of the pointCombo objects in the fillBoxes(self, pd) function. There is also the vectorExists(self, vectors, vec) function, which is the same as the one of the same name in the vectorDialog class.The editVector(self, pd) function has the same checks as the addVector(self, pd) function, so the same check if the newly added/edited vector already exists in the vector array is required.



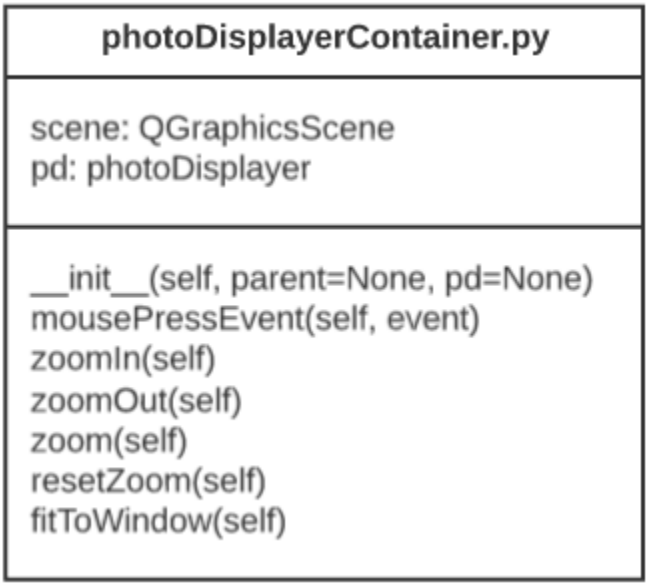
changeColorDialog class diagram.

Similar to the previous pointDialog and vectorDialog classes, the change color dialog provides the ability for the color of either the points or vectors to be changed in order to allow for more visual contrast with the image. The colors supported are white, black, red, green, blue, cyan, magenta, and yellow. We felt this gave a wide range of colors but in the future more colors could be implemented as well as a per vector or point color system to allow the user to use different colors for each part of the image. This feature was determined to not be essential to the project and was therefore left out at this time. The design of this class took advantage of the fact that the color selection for points and vectors is identical so what is changed is based only on an integer variable known as changeType so that the same dialog window could be reused for changing both the point and vector colors.



changeColor Dialog options

PhotoDisplayerContainer Class



photoDisplayerContainer class diagram

In order to implement zoom functionality it was determined that the best solution was to implement a container for the photoDisplayer, as the PyQt class the photoDisplayer was based on does not support those advanced features. The photoDisplayerContainer class is based on the QGraphicsScene provided by PyQt and provides native support for zoom. Implementing zoom in this way allowed for us to not have to redesign the photoDisplayer class, as it could simply be a part of a QGraphicsScene and have zoom functionality that way. This allowed for us to save a lot of time not having to redevelop a large section of the GUI.

The main challenge in developing this class was in ensuring that points added by the user clicking on the container have their coordinates translated appropriately to the corresponding pixel on the image. This however, was not as difficult as initially planned as the built-in mapToScene() function supported by PyQt only needed minimal adjustment to be configured to return integer values exclusively. The reason float values could not be used was simply due to the fact that an image has a discrete number of pixels and therefore the location of any point must also adhere to that. For example it would not be possible to get the depth for pixel (100.5, 0.25) so that point needs to be converted into (100, 0), so that the appropriate depth value can be referenced.

Underlying Structure

The underlying structure of the GUI is handled by the PyQt library but an understanding of how PyQt handles things is important to have regardless. A PyQt application is made up of widgets, things like buttons, combo boxes, labels, e.t.c, and has an event loop that is constantly running and checking if an event has occurred. Examples of events that PyQt is checking for include things like mouse clicks, keyboard input, window resizing, minimizing the application, e.t.c. If an event occurs it is handled by the event handler and the state of the application changes based on the event. How the application responds to these events is decided by the programmer by extending the event handling functions in a widget to allow them to perform a certain task. This event loop is generally handled behind the scenes but it is important to understand how PyQt functions in order to properly design custom widgets. It is best to not mess with any of the functions core to the event loop as small changes can lead to massive bugs that can affect all widgets in an application.

The widget class in PyQt is the parent class for every type of widget and interacts with the event loop and other widgets through signals and slots. Signals are notifications emitted by widgets when something happens, such as a user pressing a key on their keyboard or left clicking with their mouse. Many signals are initiated by user action, but this is not a rule. Signals can also be configured to give additional context to events as well. Slots are the part of a widget that receives a signal, but in Python any function in an application can be used as a slot by simply calling that function when an event occurs. The fact that many simple widgets have predefined slots is very convenient as it allows for simpler widgets to be linked directly together. Therefore the need to write many simple functions is eliminated and the code looks much cleaner.

Another important thing to keep in mind is that all widgets without parent widgets are rendered in a separate window, so if a custom widget needs to be a part of a larger widget then it is important that the constructor allows for the designation of a parent object. When a custom widget doesn't deviate too much from the parent's, the parent's super function can be called to handle an event regularly and custom code can follow or precede that. This helps to save time as some changes can be very minor but necessary in order to extend the functionality of a GUI.

A link to the documentation for PyQt for further clarification on the underlying structure of PyQT can be found at the following link: https://doc.qt.io/qtforpython/ [27]

Development Timeline

- **October 1**
  - Complete general GUI layout design.

- **October 8**
  - Complete point and vector classes.
  - Upload images to the GUI via File Explorer.

- **October 15**
  - Proper image scaling for uploaded images and switching between them
  - Ability to add dummy points and vectors to the table

- **October 22**
  - Ability to click on a point on the image and make a point
  - Zoom in and out functionality

- **October 29**
  - Ability to combine two points into a vector

- **November 5**
  - Functional depth-finding algorithm implementation.
  - Link systems so that vectors can access and calculate things based on the depth

- **November 12**
  - Vector and point display toggle
  - Begin work on saving and loading files

- **November 19**
  - Finish up save and loading of files
  - Import of dummy ML model and link it to the existing images

Expected Technical Challenges

The technical challenges related to the GUI can be broken down into two main parts. The first of which is implementation of image scaling and zoom, since it is vital that any image transformations functions for the GUI have to be invertible transformations in order to preserve the integrity of the data. It is possible that the way the image will be displayed to the user will be negatively affected by the design of the focal length prediction model or depth finding algorithm.

For example, if the resolution of an image needs to be reduced in order to reduce computation time for either the focal length prediction model or depth finding algorithm the options for displaying the image are either to simply display the scaled down image, which will most likely not look very good to the user, or to display the full resolution image and handle the conversion of what pixels are selected by the user to the scaled down pixel data that is actually processed. Ideally neither of these things will need to happen as hopefully the focal length prediction model and depth finding algorithm can be designed to work with higher resolution images. However, if during testing it is deemed that a lower resolution is required for performance reasons it is important for the GUI to be designed in advance to be able to handle this issue.

The other technical challenge is file/data management. While not a must have requirement it is likely to be one of the more challenging aspects to implement. Ideally all of the data used by the GUI can be saved to a file and accessed later when the program is launched again. It is very likely that JSON will be used over XML as XML does not have the array support that will likely be used in order to store depth map and pixel map information. The main issue is that when reading from a file many different checks need to be in place in order to avoid aberrant behavior if data is null or corrupted in some way.

Another approach to file/data management that may serve us well if it is not important to be able to export our data to other software is to simply serialize the objects we are concerned with and reload them later. This has the benefit of keeping file sizes extremely small and allowing for any custom save and load functions to avoid having to parse a JSON or XML file.

Testing Methodology

Since the GUI is made up of many different widgets that need to work together we have decided to employ a combination of unit and integration testing. Widgets will be tested individually at first and then once deemed to be fit for deployment they will be integrated into the GUI as a whole. At this point the entire system will be tested to ensure that the widget behaves as expected and does not cause any unexpected behaviors in other widgets.

Given the scope of the project much of the testing will be conducted manually as the system is not overly complex. Automatic testing was considered, but was deemed unnecessary for this part of the project. Much of the integration testing would have to be performed manually anyways and that was the main incentive for sticking exclusively with manual testing.

Some examples of what manual testing has enabled us to correct during our development so far are as follows:

- Crash when trying to add a vector when no points exist yet
- Crash when trying to delete a vector from an empty vector list
- Vectors not being deleted from the vector table when the points they reference are deleted in the point table
- Angles not being deleted from the angle table when the vectors they reference are deleted in the vector table
- Add Point and Add Vector buttons being able to add a point or vector to the list that the photoDisplayer references even when the toggle for displaying points and vectors is disabled
- Point and vector plotting being allowed outside of the image but within the photoDisplayer
- PhotoDisplayerContainer cropping the image to fit the initial size of the container rather than displaying the whole image if it was larger than the container
- Allowing duplicate points
- Allowing duplicate vectors
- Allowing decimal precision point plotting when zoomed in on an image when only integer precision will work

We will continue to perform manual testing going into the second half of the project as it has so far been very successful in finding bugs and allowing us to quickly correct them. However, once the depth finding algorithm is linked some automatic testing may be added as well to help ensure accuracy.

Focal Length Prediction Model

Overview

In order for the measurement of depth, distances and angles of an image to be possible, the focal length of the camera that took the photograph must be known. In the case of the stereogram image of the gravesite our project is based upon, this information is an unknown variable. There are a few methods in obtaining this information, each of which containing its own hurdles and required technical knowledge. One of these methods is by using a machine learning algorithm to accurately predict the focal length of our stereogram image (either both, or one of the two halves individually as it can be reasonably assumed that both lenses on the same camera would have roughly the same focal length) trained on images of known focal lengths. From this prediction (or range of predictions, depending on the exact implementation at the development stage), the output of the model, being the focal length, would be used in the measurements done in the depth-finding algorithm and user interface. These predictions would need to be precise enough to provide measurements from the other parts of the program within the one foot accuracy requirement given to us by Dr. Giroux, so the implementation and training of the model, if that is the chosen method, is an extremely important part of the process.

The model must be trained on images with known focal lengths that can be included in the training, validation and testing datasets. The model can be trained on modern images, which will provide a far greater amount of data for model training, but may not have the same accuracy for images of the same time period which the gravesite stereogram was taken due to distortions and noise. According to Dr. Giroux, if the model is trained on modern images, the model would need to be as accurate as possible as a proof-of-concept before moving on to predicting focal lengths of the gravesite image and others similar in age and condition.

Technical Objectives

- Research Convolutional Neural Networks and their application to machine learning.
- Investigate languages and libraries to build the model in.
- Determine the best pre-built network to adapt to the estimation problem, if any.
- Implement a neural network in the chosen language.
- Train and test the model’s performance on chosen datasets and rework the model if necessary.

- Pass the output of the model to the depth-finding algorithm in order to obtain real-world measurements.

Technical Goals

- Learn specifics about convolutional neural networks and their application to machine learning.
- Learn about implementation of different convolutional neural networks and their differences.
- Learn about the implementation of pre-built neural networks on available libraries.
- Develop, implement, train and test a neural network that creates predictions that lead to real-world measurements accurate to within one foot.

Technical Specifications

- The model will be trained on a dataset of images with the target parameters known and used as the labels for training and validation data.
- The model will be implemented and exported in a way such that another component of the project can run the model to predict the target parameter on a given image (such as the user interface or the depth-finding algorithm).
- The model will output the parameter in a way that is usable by other components of the project.

Technical Requirements

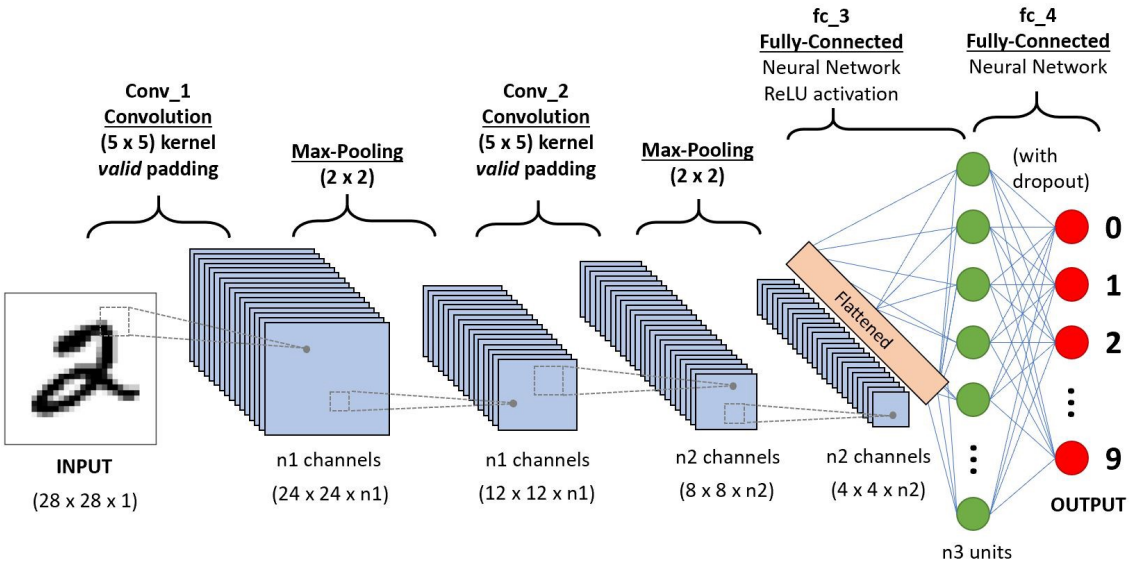
- The model must create predictions that lead to real-world measurements accurate to within one foot.
- The model must be compatible with the other components of the project that require predictions from the focal length prediction model such that it can be called to make a prediction on a given image from other parts of the project and output that prediction in a compatible format.

Machine Learning Research

The first decision of the process is what kind of neural network should be used to provide the basis of the model. This decision comes fairly easily as a Convolutional Neural Network (CNN) is designed specifically for image and video processing, most commonly used in Object Classification, but can be modified to serve other purposes such as scene recognition, image segmentation, or various regression (real value prediction) problems.

These neural networks consist of several layers of differing types and orders, specifically several variations in order of Convolutional, Pooling and Fully-Connected layers. Convolutional layers involve the use of a Kernel (or Filter), which is a two-dimensional matrix (typically of size 3x3, but can be larger sizes e.g. 5x5 or 7x7) consisting of weights to be applied to the image or feature map (the output of a convolutional layer) via dot product. These weights are the variables being learned in the neural network. After the kernel is done processing through the entire feature map, the neural network applies a Rectified Linear Unit (ReLU) function to the resulting feature map.

Pooling layers are designed to reduce the size of the input. Similar to how a kernel will go across the entire image and modify it according to the weights of the kernel, a pooling layer will process the entire image and aggregate the pixels by maximum value or average value instead of performing a dot product. Fully-Connected Layers perform the actual classification task from the outputs of the previous layers. Stacking these layers together make up the CNN, and the more layers in the network leads to the network being able to recognize more features, though stacking too many layers (increasing the depth of the model) leads to issues like the vanishing gradient. An example of these layers put together into a CNN can be seen below:



Example CNN Structure. Source: [28]

The kernels in these Convolutional Neural Networks can be visualized as a two-dimensional matrix of Weights to be applied to the input of the convolutional layer. These weights are the variable being learned by the neural network after each epoch (iteration in training - the number of epochs can be set to arbitrary numbers but generally as high as it takes to start seeing very small changes to the results) when undergoing training and validation testing. Every kernel used to traverse over a feature map in a CNN learns a feature from the image or feature map. To do so, after what is known as the Forward Phase of an epoch (where the image is passed entirely through the neural network and a prediction is obtained at the end), the Backward Phase is entered, and this goes back through the neural networks and applies changes to the weights of each filter based on the Error or Loss, which is the difference between the predicted value and the ground truth value.

In addition to the Convolutional Neural Network being the network of choice, since both the user interface and the depth finding algorithms are being written in Python and we elected to not develop a web-based application in favor of a strictly client-side approach, the model would be written in Python as well. To do so would involve the use of Keras, a machine learning API written for python that greatly streamlines the process of developing a machine learning model. The Keras library includes several algorithms already packaged within the library, making their implementations far simpler. The list of all models (except for the EfficientNet variations, their accuracies were not listed in the table but there is more information on them as described later, so all but the base

EfficientNet model are excluded to show that it is still included in the Keras library) and some quantitative data about the models are listed below (from the Keras API documentation):

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	0.790	0.945	22,910,480	126	109.42	8.06
VGG16	528	0.713	0.901	138,357,544	23	69.50	4.16
VGG19	549	0.713	0.900	143,667,240	26	84.75	4.38
ResNet50	98	0.749	0.921	25,636,712	-	58.20	4.55
ResNet101	171	0.764	0.928	44,707,176	-	89.59	5.19
ResNet152	232	0.766	0.931	60,419,944	-	127.43	6.54
ResNet50V2	98	0.760	0.930	25,613,800	-	45.63	4.42
ResNet101V2	171	0.772	0.938	44,675,560	-	72.73	5.43
ResNet152V2	232	0.780	0.942	60,380,648	-	107.50	6.64
InceptionV3	92	0.779	0.937	23,851,784	159	42.25	6.86
InceptionResNetV2	215	0.803	0.953	55,873,736	572	130.19	10.02
MobileNet	16	0.704	0.895	4,253,864	88	22.60	3.44
MobileNetV2	14	0.713	0.901	3,538,984	88	25.90	3.83
DenseNet121	33	0.750	0.923	8,062,504	121	77.14	5.38
DenseNet169	57	0.762	0.932	14,307,880	169	96.40	6.28
DenseNet201	80	0.773	0.936	20,242,984	201	127.24	6.67
NASNetMobile	23	0.744	0.919	5,326,716	-	27.04	6.70
NASNetLarge	343	0.825	0.960	88,949,818	-	344.51	19.96
EfficientNetB0	29	-	-	5,330,571	-	46.00	4.91

Keras Pre-Trained Model Applications. Source: [29]

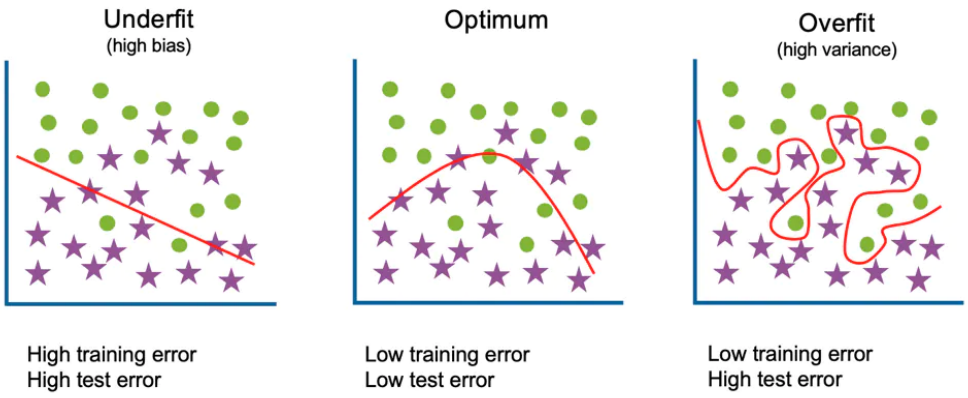
Model Architecture Research

AlexNet

AlexNet itself was created in 2012 and won that year's ILSVRC (ImageNet Large Scale Visual Recognition Challenge, a competition involving the ImageNet dataset where models must compete for the highest possible accuracy) by a wide margin with a top-5 error (the percent of images where the correct label is not in the model's top 5 predictions) of 15.3%. This network was considered a pioneer when it was released, providing a much deeper network for higher accuracy and using the ReLU function instead of the more commonly used Sigmoid function to avoid the vanishing gradient issue in machine learning (over time, while training a model with activation functions such as Sigmoid, the change in outputs over the epochs will greatly reduce, causing the "vanishing gradient"). This model was chosen by the DeepFocal team for its easy adaptability to their problem, though that decision could have come at the cost of accuracy. More about this issue can be read about in the *DeepFocal and Its Issues* section of this document. Notably, AlexNet is not included in the Keras library by default. Even though this would be an issue if we were to decide on using AlexNet for our model, it is surpassed in performance by other models as seen in the rest of this section.

Inception

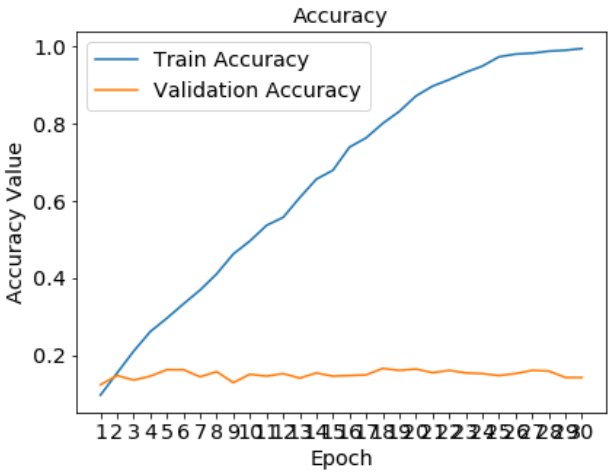
Inception is a neural network originally titled GoogLeNet (as a reference to the earlier LeNet, which was later surpassed in accuracy and performance by the similarly structured AlexNet) in 2014 that was built with one of the issues AlexNet introduced as the core concept of its design. As layers are added to a neural network, overfitting becomes an increasingly pressing issue and the model's accuracy suffers.



Overfitting vs. Underfitting. Source: [30]

Above is a visualization of overfitting, where the model performs very well with training data, but has a high error on testing data (or validation data, as some papers refer to it). Inception's design philosophy involves expanding the model in width rather than depth. Multiple convolutions of various sizes are done on the same layer and concatenated at the end, making the model much shallower, and much wider as a result. The first iteration of the Inception model (InceptionV1 or GoogLeNet) used 1x1, 3x3 and 5x5 convolutions on the same level and concatenated the results, resulting in a 6.67% top-5 error rate in the 2014 ILSVRC, achieving a lower error rate than AlexNet by more than half. Further iterations were released, InceptionV2 and InceptionV3, the latter being even further optimized by through more computationally efficient matrix operations (using 1x3 and 3x1 convolutions instead of 3x3 in some areas) and using overlapping 3x3 convolutions instead of the far more expensive 5x5 convolutions. These optimizations and further improvements to accuracy resulted in a top-5 error rate of 3.58%, cutting the first iteration's error rate nearly in half.

While Inception's design philosophy revolves around reducing overfitting resulting from stacking too many layers in the model, it is still vulnerable to overfitting from other sources. This is an issue the previous Stereogram Depth Analysis group ran into, where their retrained Inception model was very accurate on the training data they supplied, but a poor performance on the test data resulting in a textbook example of overfitting in a neural network as shown by graphs included in their model's file in their project's github repository:

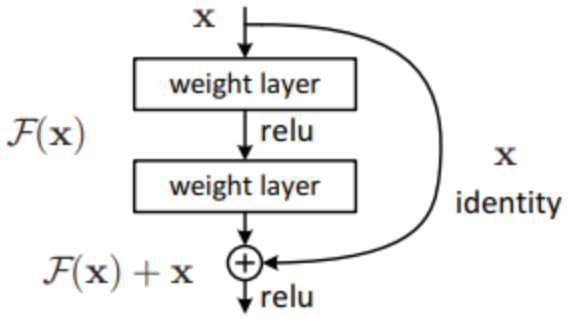


Previous Group Inception Model Performance. Source: [31]

As evidenced by the results of the previous group's efforts, the choice in pre-trained neural networks alone will not make the model produce acceptable results. More parameters need to be adjusted in order to make a model with accurate predictions.

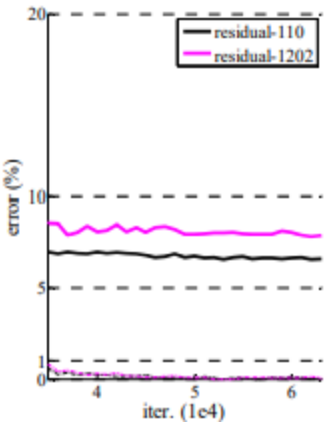
ResNet

ResNet was developed in 2015 for a similar purpose as the Inception network, and won the 2015 ILSVRC with a top-5 error rate of 3.57%. After AlexNet won the 2012 ILSVRC by such a considerable margin, researchers sought to improve the performance of that network, and one area of opportunity was by reducing the impact of the vanishing gradient problem, which is caused by the activation functions after each layer causing the parameters of the layers to become so small that, at a certain point, they virtually do not change at all and the learning of the network stagnates for every epoch afterwards. The way ResNet, or Residual Networks, handle this vanishing gradient is by using Skip Connections between layers. These Skip Connections are placed between every few convolution layers and add the input of the layer to the output of the layer after the convolution operations and normalizations, as shown below:



Residual Block. Source: [32]

These skip connections proved incredibly effective at allowing for increasingly deep networks, and the authors of ResNet even experimented with a network of 1202 layers. While the training error was similarly small, the testing error was still higher than the 101 layer residual network, though not by much [32]. The authors argue that this disparity is due to overfitting still having an impact on the model with such a large depth, even with the skip connections. A visualization of the error disparity between the 101 layer network and 1202 layer network can be seen below, with the bold lines being testing error and dashed lines being training error:

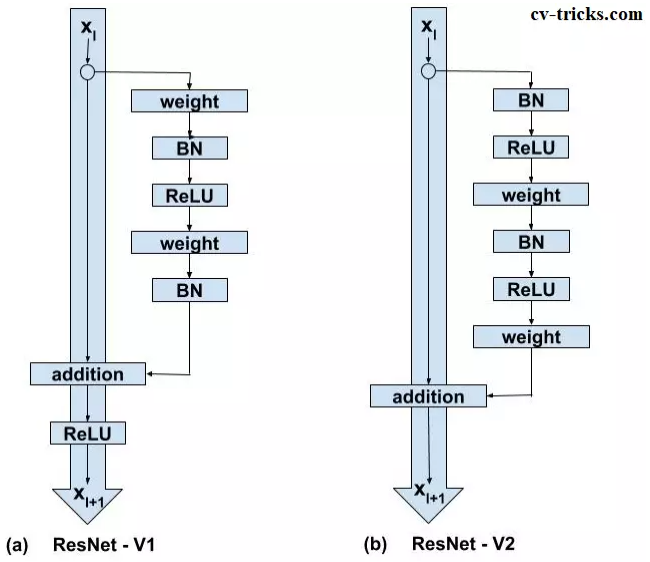


ResNet110 vs ResNet1202 Source: [32]

This design philosophy of using skip connections to build residual blocks in the model led to the creation of several variations on the formula, including ResNeXt and DenseNet. Included in the Keras library are the ResNet50, ResNet101 and ResNet152 models, and each one is implemented with both ResNet v1 and ResNet v2. The numbers 50, 101 and 152 represent the number of layers in their respective networks, offering deeper and deeper options for building the network.

The difference between ResNet v1 and ResNet v2 is that layers in ResNet v1 undergo post-activation, while layers in ResNet v2 undergo pre-activation. In post-activation, the non-linearity (ReLU activation function) is applied after the addition between the resulting feature map from the convolution and the input image. In pre-activation, the final non-linearity function is removed, and a batch normalization and ReLU activation are applied before the first application of the weights in the layer. This allows the output of the layer to be sent directly into the next layer as is, and allows the calculated error at the end of the network to be applied much easier than with ResNet v1.

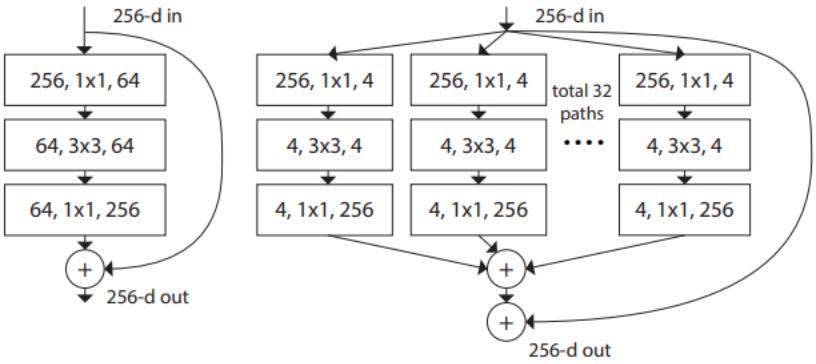
A visualization of the difference between post-activation (left) and pre-activation (right) can be seen below:



ResNetV1 and ResNet V2. Source: [33]

ResNeXt

ResNeXt combined the same general idea of multiple convolution/pooling operations on the same layer seen in Inception networks and the skip connections of Residual Networks to reduce the amount of parameters in designing networks, inspired by the simplicity of VGG networks and ResNet. Similarly to the Inception networks, the input of a layer is split, transformed by a set of filters, and concatenated at the end. Similarly to the Residual Networks, the output of the layer (or in the case of ResNeXt, the output of each transformation in the set) is concatenated with the input in order to reduce the impact of the vanishing gradient problem with deeper networks. An example of a ResNet block (left) and a ResNeXt block (right) can be seen below:



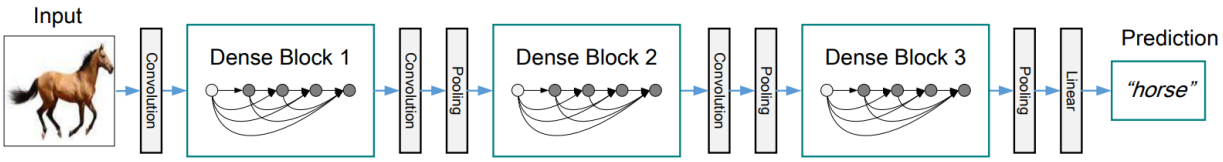
ResNet and ResNeXt. Source: [34]

ResNeXt introduces its own new dimension called cardinality, or the size of the set of transformations, that (according to the authors of ResNeXt through experimentation) can further impact the accuracy of the model when width and depth have little impact on the performance. ResNeXt would be a considerable option for a machine learning model given its simplicity and its self-reported ease of adapting to new tasks. The Keras API does not include ResNeXt by default, though there exist builds of the network for Keras available for download.

DenseNet

DenseNet (or Densely Connected Network) was developed in 2017 in an effort to simplify the connections between layers in the ResNet architecture. The way this is achieved is by giving every layer access to the feature maps of every preceding layer by concatenating the output of each layer instead of summing them like how ResNets operate, reducing the required number of parameters and increasing the amount of information each layer has access to, making models much easier to train. Being connected to all prior feature maps also gives the current layer in the model far more data to use in training the weights for that layer.

With this amount of information per layer however comes the issue of very high input sizes per layer. In order to combat this problem, the developers of DenseNet introduced Bottleneck Layers, which perform a 1x1 convolution before the 3x3 convolutions for each dense block. Having a more efficient approach to using parameters also had the effect of reducing overfitting the model to the training data, which may help with the issues the previous group had with the machine learning model. An example of how DenseNets operate showing the concatenations can be seen below:

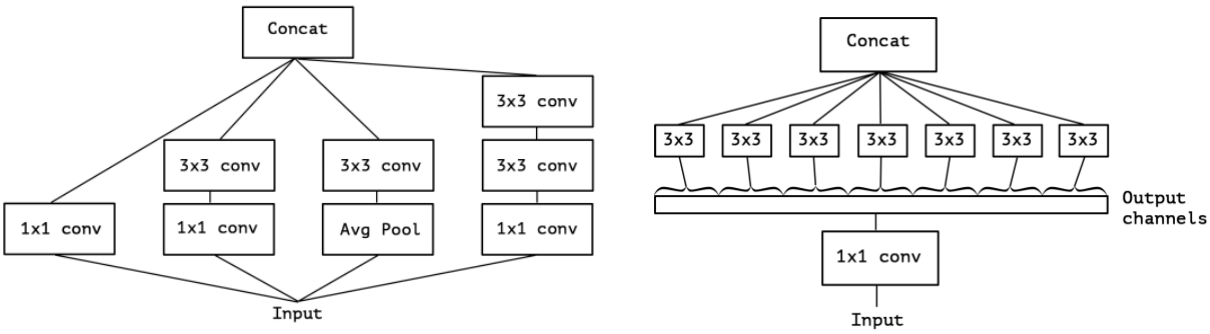


DenseNet Structure. Source: [35]

Keras comes with multiple versions of DenseNet pre-built with the library. DenseNets 121, 169 and 201 are available right out of the box, but more variations can be built from scratch using different libraries.

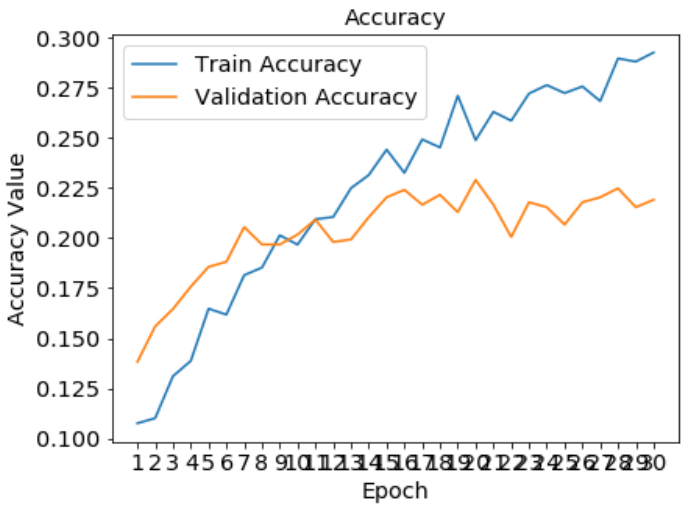
Xception

Xception (or Extreme Inception) takes the Inception model and provides a slight twist to the design. Instead of the standard Inception modules that build the network that includes multiple channels of 1x1 convolutions followed by 3x3 convolutions, the Xception module built from depth-wise separable convolutions performs the 3x3 convolutions on each depth first, then after all depthwise convolutions are complete, a 1x1 convolution across each channel is done to reduce the number of output channels. There is also a more impactful difference in the fact that there is no non-linearity function (ReLU) after the first convolution operation. The difference in structure between the two can be seen below, taken from the Xception research paper from Google, with the standard Inception module (left) and the new “extreme” version that makes up the Xception network (right):



Standard Inception Module vs. Extreme Inception Module. Source: [36]

These modifications resulted in a higher performance on the ImageNet dataset than both ResNet and Inception, even with virtually the same parameter count. This would be a formidable option when designing the focal length prediction model, especially compared to the Inception model as Xception’s very slight modification to the order of operations of Inception makes it almost a direct improvement to its predecessor. However, this network still resulted in a lower-than-needed performance for the previous stereogram depth analysis group, as they also used Xception. If the graph of the model’s accuracy (shown below) is accurate, it seems that their model was underfit, as both the training accuracy and the validation accuracy were both low:



Previous Project Xception Model Analysis. Source: [31]

This indicates that their model was either not trained to recognize a sufficient amount of features, too much regularization in the model, or another unknown issue. If the model was trained for a longer period of time on more data, this could have alleviated the issue of underfitting if that was the issue. In these circumstances, it does not appear that the choice in the network was responsible for the underperformance in the model.

MobileNet

MobileNet was developed specifically with mobile devices in mind by Google to create a very lightweight and streamlined neural network. It uses the same Depthwise Separable Convolution layers that Xception introduced as its primary layer, which was shown to be an increase in model accuracy. Because MobileNet is so lightweight and efficient with its parameter count at the cost of some of its accuracy, it makes a perfect choice for being run on websites or mobile devices where computational power is severely limited, compared to a dedicated graphics processor or CPU that models are usually run on. Since the application we are designing is not web-based, nor would we want to sacrifice accuracy for running it solely on a web page if it were, this network would most likely not be a top choice given that we are not limited by running solely on a webpage or mobile device.

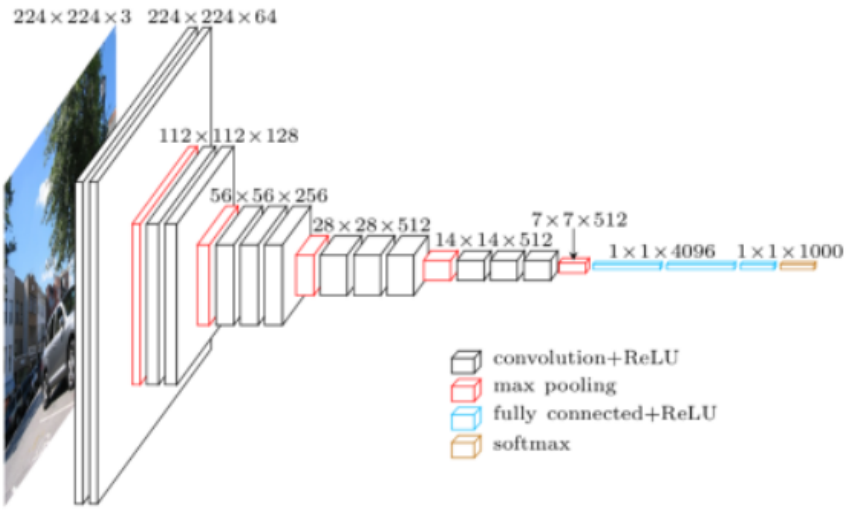
However, it does introduce the concept of Depthwise Separable Convolutions, which would go on to be used by networks like EfficientNet due to their computationally efficient nature as opposed to standard convolutional operations. The depthwise separable convolutions take standard convolutions and break them into two parts - the depthwise convolution and the pointwise convolution. The depthwise convolution takes the kernel (or filter) of a layer and applies it to each channel of the layer's input. The

pointwise convolution then goes across the output of that depthwise convolution and applies a 1x1 convolution to the resulting features, combining them. According to the authors of MobileNet, the depthwise separable convolutions are eight to nine times more efficient than standard convolutions, though this does still come at some performance cost [37].

VGGNet

VGG16 and VGG19 are both variations of the VGG network that came in second place for the 2014 ILSVRC, behind GoogLeNet (Inception network's first version). 2014 was the first year any network achieved an error rate of sub-10%, and VGGNet's iterations would go on to inspire paradigms adopted by other networks developed in the future. Its use of only 3x3 convolutions instead of the much larger and more expensive 5x5, 7x7, etc. convolutions greatly increased efficiency without the loss of accuracy.

However, this was still relatively early in the world of image classification, and before innovations such as ResNet's skip connections would allow for far deeper networks, so the 16 layer and 19 layer iterations (where the numbers in VGG16 and VGG19 come from) were considered extremely deep at the time. In addition, the training process is known to be slow and inefficient when compared to other networks, as VGG initially used pre-training, where versions of the network with fewer layers are trained and then used as initializations for larger versions. This means whole networks must be trained one after another, making the process very slow. A visualization of VGGNet's structure can be seen below:



VGG Architecture. Source: [39]

Since VGGNet's development, far more efficient and accurate models have been published, so this specific network, while still used today, would probably not be a top choice for the network. However, knowing the innovations VGG introduced is a great insight to how its successors operate.

EfficientNet

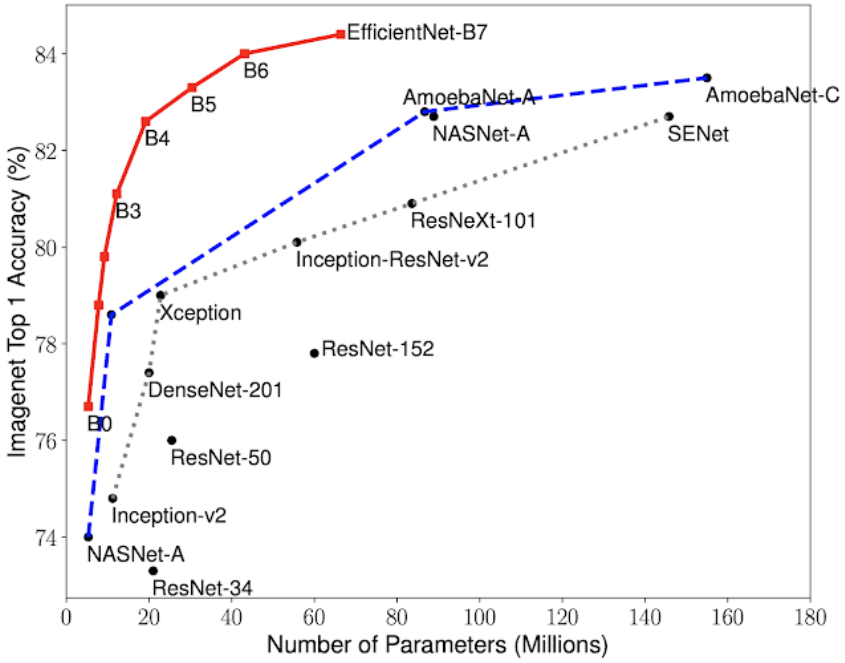
EfficientNet is an architecture developed by Google researchers in 2019, making it one of the most recent innovations in the field. This architecture's design and concept revolves around the scaling of the model and how it affects performance. With some other networks (such as VGG16 and VGG19, and ResNets 50, 101 and 152), the scaling from the baseline model is highly arbitrary - the amount of scaling between versions of the model being chosen through trial-and-error while experimenting with the network's performance, or by some other means. This arbitrary scaling is also generally in one dimension, be it width, depth or resolution.

With EfficientNet, this scaling is done mathematically through a form of hyperparameter optimization (hyperparameters are variables which affect the resulting predictions of a model that are not learned by the model through training, e.g. learning rate or batch sizes) known as a grid search in order to find scaling coefficients for the width, depth and resolution of the entire network in relation to the resources available to the network. The scaling coefficient is then applied to the entire network uniformly based on the resources available in all dimensions. These scalings resulted in each version of EfficientNet: B0, B1, B2, B3, B4, B5, B6 and B7. The number for each network represents its position in size compared to the other members of the EfficientNet family, with B1 being larger than B0, B2 being larger than B1, and so on.

The baseline EfficientNetB0 model came from Google's own neural architecture search framework known as AutoML MNAS, which automates the design of neural networks with surprisingly accurate results. This baseline uses the same inverted bottleneck convolution layers that MobileNet uses to boost its parameter efficiency, and part of its structure can be seen below:

EfficientNet Base Model Structure. Source: [40]

EfficientNet was tested on several datasets, primarily the ImageNet dataset, where, for example, EfficientNetB4 has a 82.6% Top-1 Accuracy, while ResNet50 has a Top-1 Accuracy of 76.3%, even though EfficientNetB4 uses fewer parameters than ResNet50. The substantially high accuracies in other datasets show that EfficientNet can transfer between sets formidably. An example of EfficientNet's performance compared to other models was given by Google and can be seen below:

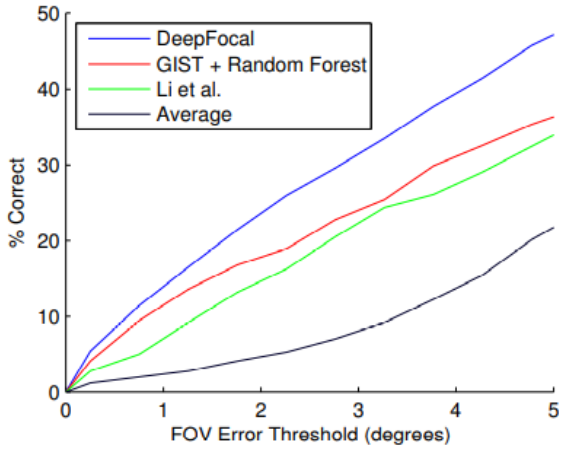


EfficientNet Performance Comparison. Source: [40]

EfficientNet would be a formidable option as a pre-trained model to use as a basis for our focal length prediction model. Its use of more efficient convolution methods would reduce training time for networks of similar sizes, as well as the accurate base model used for the scaling providing great performance in the datasets used. It also acts as a sort of proof-of-concept of transfer learning's place in training a model on different datasets, as outlined in the EfficientNet documentation.

DeepFocal and Its Issues

Before going about creating an entire neural network from scratch, it seemed prudent to try to find any existing research and/or documentation on predicting focal length from an image using a neural network. A group of computer science researchers from the University of Kentucky released a paper on their method of predicting focal length in an image using a modified version of the AlexNet architecture known as DeepFocal, and this research seemed to also be what the previous Stereogram Depth Analyzer group's work was mostly based upon. This document provides great high-level insight on their implementation of the altered AlexNet framework as well as visualizations on the, admittedly low, accuracy of the model as shown below:



DeepFocal Performance. Source: [41]

This graph includes other "baseline methods" of prediction that proved even less accurate than the DeepFocal network, which was only able to achieve a roughly 30% correct prediction rate within a 3 degree margin of error. Since other parts of this program of calculating the distances in the gravesite image are dependent upon the focal length and the accuracy of any prediction pertaining to the focal length, in addition to the accuracy requirement given by Dr. Giroux being within one foot, this exact implementation would not do on its own. Though the University of Kentucky's research would be a good insight, other methods would need to be implemented.

This inaccuracy was believed to have come from the foundational architecture from which everything else in their model was built upon, AlexNet. According to the University of Kentucky's documentation on DeepFocal, AlexNet seemed to be an easily adaptable choice for what was required, as the only change necessary was changing the final fully-connected layer to one with only one output node representing the focal length of the image. Also worthy of note, the DeepFocal documentation was published

in 2015, and in the last six years, more recent innovations in machine learning architectures have come about, producing more efficient and, in some cases, accurate algorithms for image classification. For example, in the very same year of DeepFocal's documentation publishing, the ResNet and InceptionV2 network algorithms were released, providing two different approaches to improve accuracy in predictions. This led to more research being done on newer algorithms and how they might compare to AlexNet.

In addition to these limitations of AlexNet, it is not included by default in the Keras library. There are ways of easily implementing versions of AlexNet with the Keras library, but the far more recent innovations in the field of convolutional neural networks makes it an obsolete option.

Building onto the possibility of the pre-trained model possibly being the root of the inaccuracies shown in the DeepFocal documentation, the type of output from the model could also have led to the inaccurate prediction results. DeepFocal's single modification to the AlexNet architecture was to change the last fully-connected layer of the model into a single node which corresponds to the horizontal field-of-view of the image. This makes the architecture designed around object classification perform a regression task (instead of probabilities of different categories like in Classification problems, Regression problems involve predicting real values).

If this is a contributor to the issue, keeping the existing architecture as a classifier rather than regression could help. However, this limits the precision of the output of the model, since there are a finite amount of nodes in classification models, whereas regression models have much less of a limitation in that regard due to it predicting real values instead of preset categories.

Given that the model itself is designed properly and can achieve high accuracy with regression rather than classification, regression should be used instead of classification, due to the focal length of the camera being wildly variable, and we don't know anything about the camera used to take the photograph this project is based around. If classification is used with an acceptable amount of categories (representing possible focal lengths or horizontal field-of-view angles as used in the DeepFocal documentation), the probability for each category can be shown in the output of the model, and if regression is used, there are methods of showing confidence intervals for the prediction, which can be used to give a certain range of real values for the depth finding algorithm in search of the real-world measurements.

The inaccuracy could also have come from the use of transfer learning instead of completely retraining the model from scratch. Transfer Learning involves taking the weights already obtained from datasets (in DeepFocal's case, the weights were obtained from a hybrid of both CaffeNet and Places, two pre-trained neural networks based on the AlexNet architecture and both used for the task of classification, one for objects and one for scenes) and using them for a different task. The authors of DeepFocal reference a paper written about transfer learning, saying it is quantifiably better to train a model with pre-trained weights as opposed to entirely randomly initialized weights, so the likelihood of the network's inaccuracies resulting from the use of transfer learning is low.

In DeepFocal's case, they use these AlexNet variations as a feature extractor, meaning the only change made to the architecture itself in order to adapt the network to this new problem is the final output layer. This is one method of transfer learning also known as feature extraction, where a base pre-trained model is used in its entirety, except for the final output layer, which is changed to fit a new target parameter, or new dataset.

Nevertheless, it could be worth trying to train a model from scratch instead of using weights from existing models if there is enough time remaining in the semester to allow it, and if all other solutions come up ineffective. This would be significantly slower than transfer learning, as the most common way of training a model from scratch is to randomly initialize the weights for each filter in the model and train from there. The choice in model will have an effect on the amount of time this traditional training method will take - EfficientNet's shallower variations would be faster than networks such as Xception, as seen in EfficientNet's comparison graph. Due to this limitation, it would be an absolute last resort given the limited timeframe this project takes place in.

FocaLens

When researching for datasets to use in the training of our machine learning model, the previous group's design document was analyzed to help in the search for any datasets that might prove beneficial. The mention of the FocaLens dataset stood out, so more research was done on that experiment.

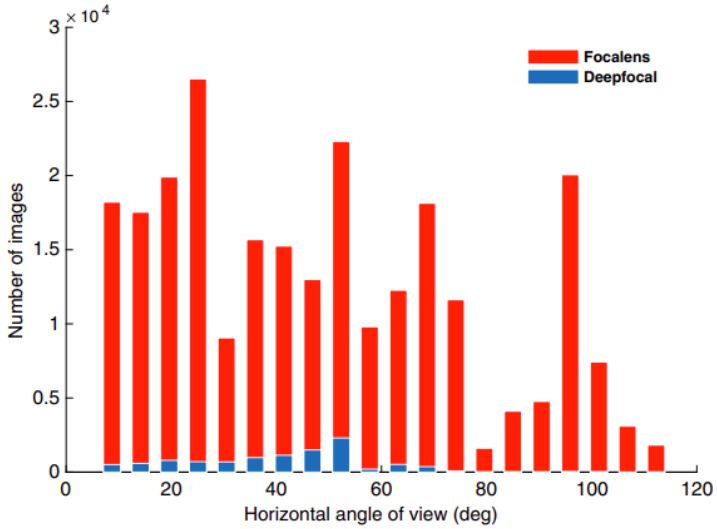
Similar to DeepFocal, FocaLens was an experiment in focal length prediction in images through machine learning. FocaLens also references the work in DeepFocal several times, as FocaLens is a sort of development and improvement on DeepFocal's findings. The basis for FocaLens's philosophy is that DeepFocal's dataset was far too small to give more accurate results in the network, and that the use of a standard convolutional network did not account for details in images that could aid in the search for the focal length, such as object density and distribution.

For the dataset, the authors created their own containing over 200,000 images in total, and each image is labeled with the focal length of the image. In order to create this dataset, the authors queried the Flickr database to obtain over one million total images with keywords pertaining to four image categories (city, landscape, indoor and portrait) and commonly used focal length measurements. From there, they use the formula also used in DeepFocal to map the sensor width and focal length of a camera to horizontal field-of-view:

$H_{\Theta} = 2arctan(\frac{w}{2f})$, H = Horizontal field of view, w = sensor width, f = focal length Source: [41,42].

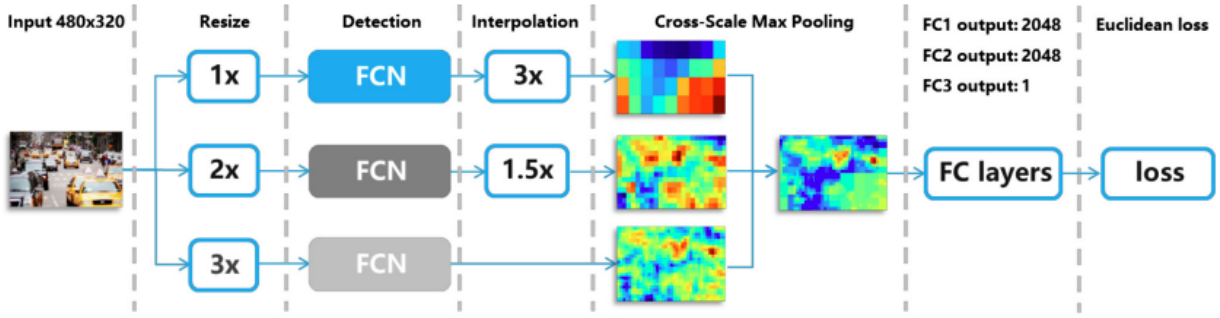
This further filters the query results in order to obtain images only taken by DSLR cameras by certain brands in order to avoid having to manually input the sensor widths of every camera model from every brand. After this filter, several screening filters are applied to the results of the previous filter, including checks for inconsistent aspect ratios as a result of cropping and checks to ensure the image actually matches its classification as all images in the Flickr database are manually classified.

These queries and screenings resulted in the 200,000+ images in the dataset, which, at the time of the FocaLens documentation's writing, was (and possibly still is) the largest existing dataset for focal length prediction. This is also substantially larger than the dataset used for DeepFocal, as can be seen in the graph from FocaLens's documentation on the comparison between the two datasets:



Field of View Distribution for DeepFocal vs. FocaLens Datasets. Source: [42]

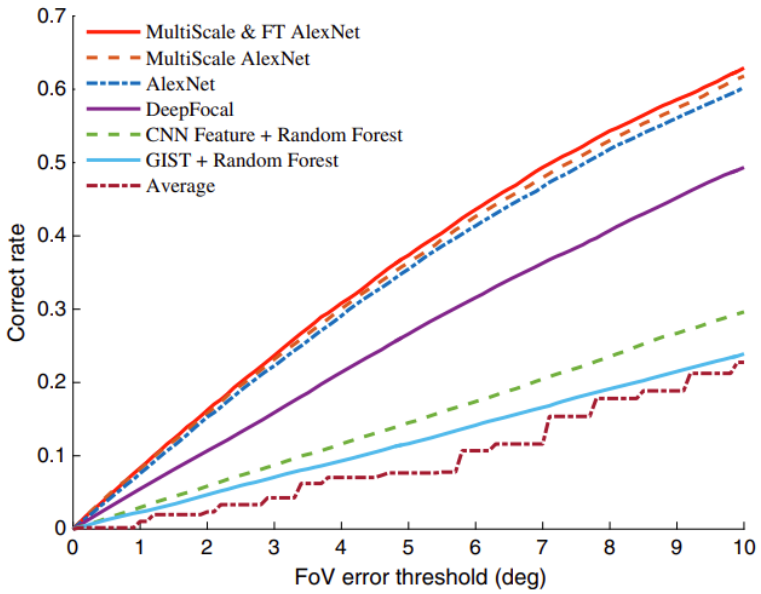
For the actual neural network, the authors of FocaLens introduced the idea that object density and distribution in the image can help in the prediction of focal length. In order to implement this into a neural network, they propose a network with three branches, each with a different input size. From the resulting “heat maps” (representing object density and distribution in the image) from the convolution layers in the individual branches, the size of each heat map is interpolated to match the size of the largest heat map (the input images all have the same aspect ratio as a result of the dataset screening and are set to a 480x320 size prior to being sent through the network) and max-pooled together. Directly after the “cross-dimensional max-pooling layer”, the output layer makes its prediction. A visualization of the proposed structure can be seen below:



FocaLens Neural Network Proposed Structure Source: [42]

The combination of the vastly larger dataset used to train the model and the introduction of object density and distribution in the image led to better performance of the FocaLens

model than the DeepFocal model. However, the accuracy is still only slightly better than the DeepFocal model with a 4-5 degree margin of error for the field of view, with FocaLens's multi-scale model giving about 40% accuracy at a 5 degree error threshold, compared to DeepFocal's roughly 25% accuracy at the same threshold. A visualization of these error rates can be seen below from the FocaLens source documentation:



FocaLens vs. DeepFocal Accuracies. Source: [42]

Although the FocaLens modifications to database size and network structure did improve the accuracy of the predictions, it remains less than desirable for the accuracy required to predict our targeted photograph. This paper does introduce a couple considerations for the construction of our model, however. The dataset acquired by the FocaLens team could be used to train our model, as it is more robust

Dataset Research

The dataset used to train a neural network is one of the most important aspects of the model making process. A well designed base model without an equally or more solid dataset will still return poor results with the analytics and prediction results. The datasets must be substantially large for training purposes, consistent in its content, and be labeled for the proper target parameter, which in our case is the focal length of the camera that took the picture.

Initially, the idea of building the dataset using 3D modelling was discussed amongst members of the group. Being able to use a free modeling software such as Blender (which is widely used for artistic and experimental purposes) to create and arrange objects in an entirely controlled area with multiple cameras seemed ideal for the purposes of this problem. However, the idea was sidelined in favor of real-life images in order to give a far more diverse set of data for training purposes. Replicating the variance, noise and distortion of real life images for tens or hundreds of thousands of samples by hand or learning how to do so programmatically would take significant effort on its own, as well as significant amounts of time to adjust, process and render each image.

The issue of finding sufficient existing datasets then arose. As mentioned previously in the FocaLens section of model architecture research, this experiment was briefly discussed in the previous Stereogram Depth Analysis group's documentation, and seemed to more heavily favor the DeepFocal experiment. This dataset seems to only be available in a limited capacity on Figshare, and even searching through the Google dataset search engine and the Kaggle collection returned only the limited dataset or nothing at all, respectively.

Each image in this limited version of the FocaLens dataset is reduced to a 420x380 pixel resolution, which is the required input size for their neural network. This is a negligible issue, however, seeing as the default input size for the Xception Keras pre-trained model application is 299x299 pixels and only has a minimum input size of 71x71 with no maximum input size listed, and the EfficientNet model in Keras has input sizes scaling from 224x224 to 600x600 on the networks EfficientNetB0 to B7, respectively. No matter what dataset is used, resizing and padding will occur to fit the required dimensions. When downloaded and the compressed folders are extracted, the total size of the dataset comes to 6.71 GB, with a total of 232,768 images within the dataset across the four categories.

The README file included with the dataset explains the use of the dataset. All images are located in the same folder, regardless of category, and each assigned an ID as a name for the file. There are eight text files that categorize the data into all four categories (city, scene, portrait and indoors) and into test or training datasets. These files contain only the ID of the image (which can be converted to the file path of the image by adding ".jpg" to the ID) and the horizontal field of view in radians.

Something to consider when using this dataset is that FocaLens and DeepFocal were created to predict the focal length of images with any category present in the ImageNet labels. In the context of our targeted photograph, the scope of the dataset does not need to be nearly as vast as the ones used in FocaLens and DeepFocal. One possibility of narrowing down the dataset to more relevant images would be taking the publicly available limited version of the FocaLens dataset and narrowing it down to any images of cemeteries, graves, obelisks, monoliths, etc. given that our targeted photograph is one of an outdoor scene with three pyramids and several graves.

The issue of what images were used by the authors of FocaLens then arises from this proposition. Given the fact that there are over 200,000 images in the dataset, manually screening every individual image to check if it somewhat contains relevant geometric cues in an outdoor setting would take an exorbitant amount of time. One way of getting around this would be using another neural network to go through the images in the dataset and pick out which ones correspond to the keywords we want, or build a dataset in the same way as the FocaLens team, but add more and change other keywords in the initial queries to search through the proper categories on the website.

The former suggestion of using another neural network to scan through the images in the existing dataset from FocaLens leads into another issue, however. The models for Keras can be initialized either with random weights (untrained), or with weights trained from the ImageNet dataset. The ImageNet dataset does not contain labels for things like "gravesite", "grave", "cemetery", etc. When experimenting with predicting labels of different images of gravesites online using the Xception network with the ImageNet weights, the most common predictions came out as "picket fence" or "obelisk". This means a new network will need to be trained, though this new network could use the ImageNet weights for initialization to utilize the benefit of transfer learning.

In addition to this limitation, the FocaLens dataset uses labels pertaining to the horizontal field of view in radians, instead of the actual focal length of the camera that captured the image. According to the mathematical formula used in FocaLens and DeepFocal to find this horizontal field of view, we would need the sensor width of the

camera as well as the focal length, which are two parameters that we don't have when considering the target photograph. This completely invalidates the use of the FocaLens dataset for our model, since the target parameter we need for the Depth Finding Algorithm is the focal length, not the horizontal field of view. However, this does not invalidate the process by which the FocaLens team obtained the images that made up their dataset.

With regards to the latter suggestion of creating a new dataset from images we collect from the Flickr repository, given the documentation for FocaLens and their detailed steps in acquiring and screening all the data that went into the set, we can get images of grave sites and cemeteries in a similar fashion. This would require significant setup and time for use of the Flickr API, getting the images matching our queries (which would likely number in the hundreds of thousands, taking considerable amounts of space), then programmatically screening through the gathered images to guarantee consistency in the same way as the FocaLens dataset's construction.

Despite it taking considerable amounts of time, this would be the best way of creating a dataset tailored specifically to our requirements. An alternative to this would be using the datasets the previous Stereogram Depth Analysis project group used, though this could lead to more inaccurate results if the dataset was a contributing factor to their model's poor performance.

With any data acquired, the images in the dataset must be configured to fit the neural network's input parameters. For Keras, input sizes vary between the pretrained models, and images must be square (1:1 aspect ratio). For images that are not square, the dimension with the greatest size can be resized to fit the required input size, and the same factor used to resize that dimension will be applied to the other dimension. The other dimension will need padding to ensure a 1:1 aspect ratio. The image cannot be cropped and must be resized by other means, since the act of cropping to resize affects the horizontal field of view, thus impacting the focal length measurement.

Research Summary

Considering the above information on the possible pre-built neural networks implemented in the Keras library, the Xception model appears to be the best fit for this project. Its low parameter count and high performance relative to the amount of parameters gives us a more accurate base model while also keeping training and validation times lower than other networks with a higher parameter count.

Using this model could also see benefits from taking the previous stereogram depth analysis group's implementation of the network and training it on more data, since (as seen in their graph of the Xception model analysis) their implementation appeared to suffer from underfitting. If the model structure itself was adequate, then the issues with inaccuracy should be resolved with more data used in the training phase. This would rely on the datasets we choose to train the network on, as it would need to be larger by a certain degree in order to fix the problem of underfitting resulting from lack of data, if that was the issue.

If that does not solve the issue, then several options need to be experimented with: a different dataset, a different pre-trained model in the Keras library, or we must consider an entirely different approach to finding the focal length of the image. If a different pre-trained model is experimented on, the EfficientNet model also shows promise in the same ways the Xception model shines. Its low amount of parameters and high performance on multiple datasets will be useful if we adapt the model to our dataset properly. The use of more efficient convolutions in the network as a base for the model would also improve training times for the larger dataset that would be gathered for the purpose of training and validating the focal length prediction model.

For the dataset, some kind of image scraper to access the images acquired from a query in the Flickr API would work for the purposes of gathering large amounts of data. Queries ensuring that focal length is included in the data as well as certain input size and aspect ratio criteria similar to the FocaLens queries should be used. Given that the previous Stereogram Depth Analysis group's image scraper was successful, modifying that software to further enhance the data being acquired can work for our purposes.

Initial Design

In order to aid the development of the user interface and depth finding algorithm while the machine learning model was still in the research phase, a model which would take in an image and just output any number within a range should be created. This dummy model would be just the Xception neural network trained on any dataset (or just keeping the pre-trained ImageNet weights as included in the Keras library) with the final output layer changed to output a single number representing the focal length of the image.

For the dummy:

- Uses a pre-trained Xception model available in Keras.
- Can use default input image resolution.
- Final output layer replaced to output a single integer to use as a fake focal length prediction value.
- Model file exported and able to be imported by the User Interface and used to “predict” an integer from the input.

For the official model:

- Modify image data to fit input parameters.
- Train model on collected dataset.
- Edit training parameters and layer data to affect accuracy.
- Export the model and ensure the user interface can use it to predict the target parameter from an image.

Implementation Details

The overall build plan for the model is very similar to the standard design process for any neural network:

- Preprocess dataset images.
    - Resize images in the dataset to fit the required input size for the chosen pre-trained neural network (Xception was chosen, though others such as EfficientNet can be experimented on if Xception and its results do not produce adequate results). This step will need to be repeated whenever the input size parameter changes, such as when the pre-trained model is changed.
    - Split images into training data and testing data sections. Test data must reflect the features of the training data to accurately validate the model's performance. 75% - 80% training data and 25% - 20% testing data should be a sufficient split. This split can be handled by Keras when training the model.
- Modify model structure and parameters.
    - The Xception model will be used as a base, and from there, frozen layers will be unfrozen in order to train the model on the new database. The final layers will also be changed to fit the target parameter of focal length.
    - Based on validation analytics, parameters will be adjusted and the model will be retrained again.
- Train and test the model.
    - After model parameters are tweaked, use built-in Keras functionality and data visualization libraries such as Matplotlib to show model analysis. Repeat previous steps based on training and testing results.
- Predict focal lengths.
    - Once the model reaches sufficient accuracies, it can be used to predict focal lengths. Its accuracy should match the accuracy observed in the training and testing statistics on images that are similar to the dataset used in the design process.
- Export the model.
    - In order to be used by the user interface, the model must be exported at the end of training and testing. Both the model itself and its output must be in a usable format and integrate with the user interface after exporting.

In order to implement the focal length prediction model, the following technologies will be used:

- Python
    - A high-level object-oriented programming language designed to expedite development processes with a simple syntax, standard library with significant amounts of native functionality, and capacity for importing libraries created by other developers.
    - This language is being used for the development of the User Interface and Depth Finding Algorithm components of the project, and though it's not required to use the same language to develop the model, the functionality included with several Python libraries specifically for machine learning makes the language ideal.
- Google Colab
    - Free service from Google that allows for execution of Python notebooks, which combine code and documentation for readability and being able to run parts of software in isolation and in different orders. This service utilizes Google hardware and does not require the user's computer to be set up for machine learning and hardware acceleration, which is a convoluted and potentially volatile process and varies heavily depending on hardware.
    - The base service is free, though there are cheap paid subscription options to increase access to higher quality hardware and longer use times. It is unlikely that these will be issues that prevent the development of the model, though if it does then discussion of the low monthly fee can happen.
- Tensorflow
    - Machine learning library developed for languages including Python.
    - Includes the Keras library, which is what will primarily be used in the implementation of the machine learning model.

Testing Methodology

Testing the model is a self-explanatory process. Tests for accuracy and model performance come with the process of neural network development, and if the model does not give adequate accuracies, several parameters should be checked and/or adjusted. Changing the model should start at the low level and ascend from there, going from adjusting training parameters such as learning rate and batch sizes, to the structure of the model itself, to the dataset and its possibility of causing error, to the overall method of the solution by using a machine learning model.

Testing the model's performance comes down to the model's analytics, which Keras (and other Python libraries if Keras functionality is limited from what is required) facilitate access to during the training and validation stages of the model's construction. Given sufficient training and testing on a model's current structure, if performance is still sub-par, reworks and adjustment is necessary.

The integration with the user interface also must be tested. When the model changes and must be exported again, it must be tested with the user interface to ensure that it still works as intended with the front end.

Development Timeline

- **October 1**
  - Obtain, analyze, possibly diagnose previous group's work.
  - Set up local environments.

- **October 6**
  - Initial Machine Learning Research
  - Convolutional Neural Networks research

- **October 13**
  - Research into DeepFocal, AlexNet and Inception

- **October 15**
  - Research into ResNet, ResNeXt, DenseNet

- **October 25**
  - Research into Xception, MobileNet, VGG

- **November 8**
  - Revisit previous group's work for Dataset research.
  - Continue Neural Network research

- **November 19**
  - Set up Google Colab file with pre-trained Xception model
  - Resize network to fit MNIST dataset

- **November 25**
  - FocaLens dataset/neural network structure research

- **December 6**
  - Finalize research
  - Develop dummy model and export for use in User Interface

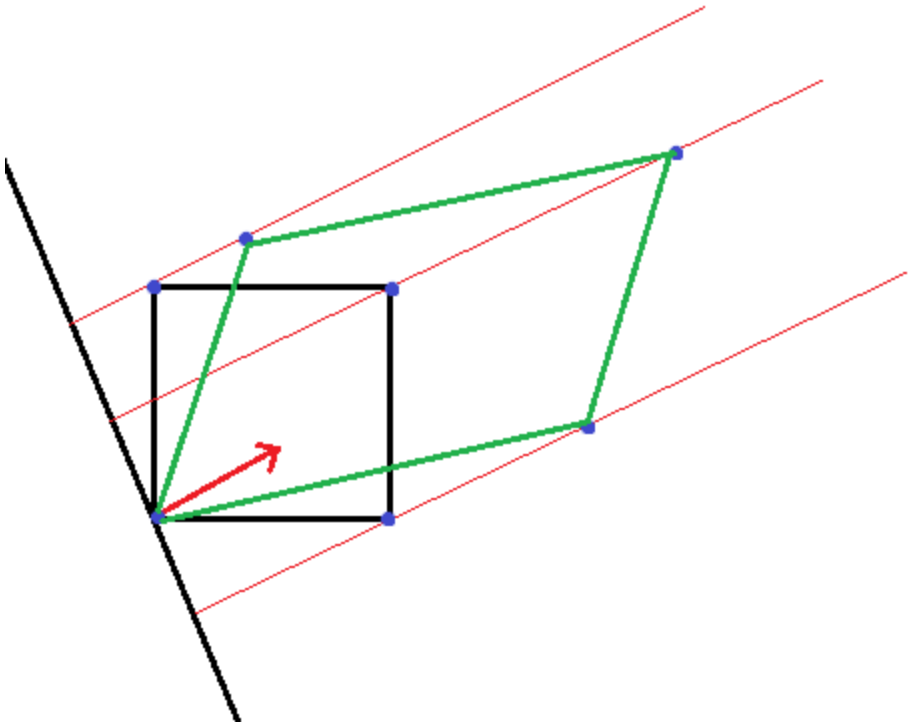
Possible Issues and Plans to Address Them

Depth Finding

Issues arising in the prediction of the depth map have the potential to make much of the data we produce unusable. Currently we are operating under the assumption that our model will produce results close to what is the ground truth of the image. However, due to the fact that the photographs we are using are from an older analog camera there are many sources of error that may present challenges in producing an accurate depth map. The first of which is noise in the image being processed. Feature matching, a step in finding the depth map, is a process that is somewhat sensitive to noise in the right and left images. If it is determined that noise in the images is affecting the accuracy of the depth map too heavily, it is advisable to preprocess the images in order to reduce the noise. Edge detection or smoothing techniques have the potential to increase the accuracy of the depth map as smoothing techniques are explicitly designed to reduce the amount of noise in an image and edge detection would discard noisy parts of the images and preserve only the edges. In our case discarding this data is fine, as we are generally interested in finding distances between the edges of two objects. In this case the base of the pyramid and the gravestones.

Another technique that will be central in combating any error in the depth map will be the ability for the user to enter known lengths in the images to correct the magnitude associated with a plotted vector. Say for example, our model predicts that the base of the pyramid has a side length of 10 meters, but in reality it has a side length of 8 meters. When the user goes to correct the magnitude of that vector we can obviously correct that vector by reducing the magnitude but we can also use that information to correct the entire depth map. By taking the projection of all other points onto that vector and correcting them by the same amount the original vector is, we can improve the accuracy of the depth map in its entirety. The geometric intuition is as follows. Say we had a box that was meant to hold some sticks but the box is too small when we attempt to place the first stick in. Instead of increasing the length of the box only where the stick is supposed to be, we can simply stretch the entire box to match the stick. This is essentially what we are doing when we correct the depth map.

Mathematically we can describe what we are doing as a scaling transformation in the direction of our original vector. As shown in the picture below, all points are scaled by a factor of two in the direction of the vector normal to the plane shown in red.

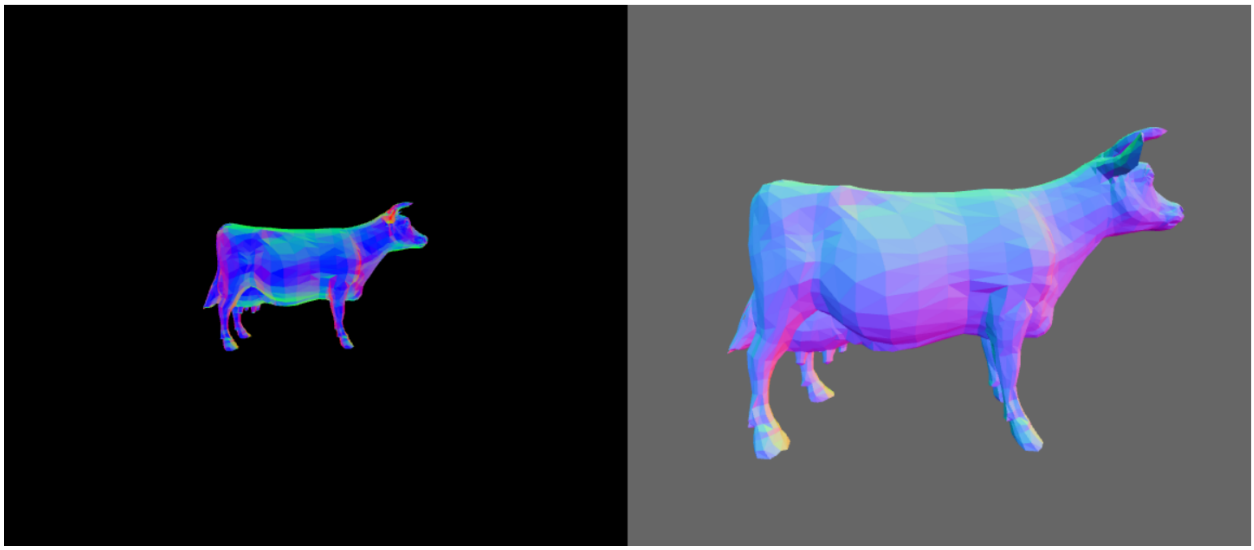


2x scaling of a square in direction of the normal vector

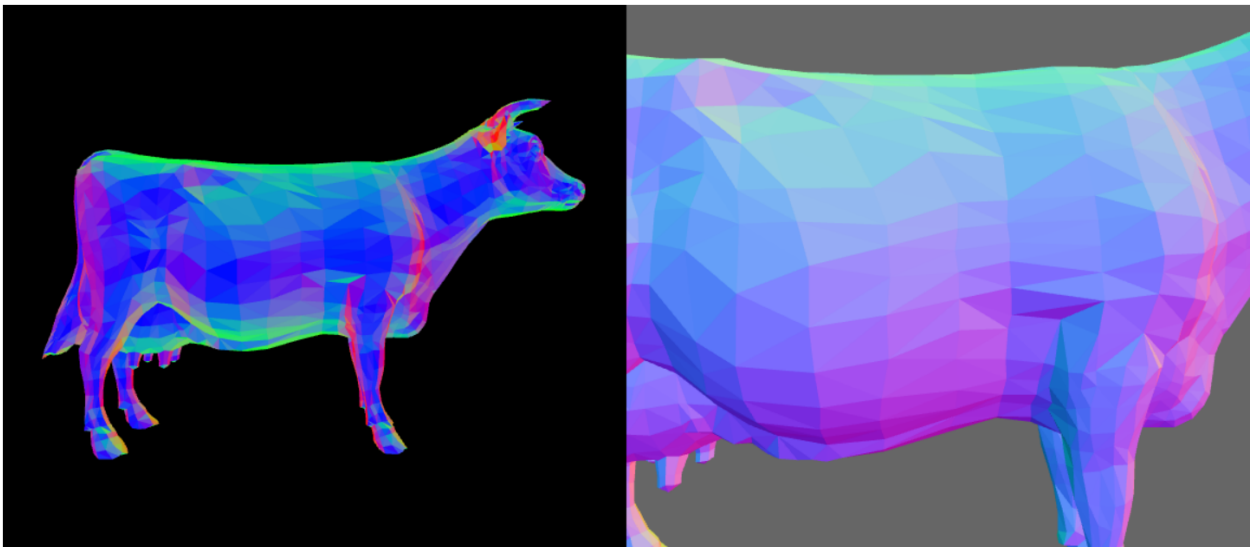
This is essentially what we will be doing except in three dimensions rather than two. This transformation can be represented by a matrix and efficiently computed due to the vectors we are using being numpy arrays which support matrix multiplication operations natively. The more information we know in the image the more accurate we can make our predictions. Ideally we know the lengths of many vectors in the image that are close to perpendicular as that will provide the largest amount of error correction.

Focal Length Prediction Model

One concern about the focal length prediction model based on the research we have conducted, is that it may not be possible to accurately estimate the focal length of this camera to a sufficient degree. Research in focal length prediction is fairly limited as the focal length of modern cameras is generally given and thus there is no need to make a prediction for it. It is also generally a very awkward problem to solve as the size of an object in an image is dependent on distance from the camera as well as the focal length. For reference, see below a render of two cows with the cow on the right being twice as large as the one on the left. Also note that field of view and focal length are proportional to each other. The 1st image is a render of the cows with a 90 degree field of view, and the second is with a 45 degree field of view. As you can see the cow on the right and cow on the left when rendered with a different field of view are very close to the same size.



90 degree FOV render



45 degree FOV render

It is for this reason that the machine learning model has been left to be developed until later in the lifespan of the project as it may be adequate to have an expert make a guess on what the focal length of the camera is and apply the error correction techniques outlined in the previous section. Other possible issues regarding the focal length prediction model include the lack of readily available training data, as again the problem is not particularly well studied as obtaining the focal length of a modern camera is usually as trivial as googling the model of the camera. The current idea to solve the issue of a lack of training data is to generate our own from a pre-existing dataset, but the challenge involved in that may make it impractical for the scope of this project.

Regardless of the challenges with this approach, we felt it best to devote time into researching this problem fully in order to find possible solutions before ruling it out, as a model with the capability to accurately predict focal length could prove extremely valuable if feasible.

Project Summary and Conclusion

In summary this project has three distinct components, a GUI, depth finding algorithm, and focal length prediction model. As of the writing of this document, the GUI is almost completely finished and fully functional in order to facilitate focus almost exclusively on the depth finding algorithm and focal length prediction model during Senior Design 2. The depth finding algorithm has also already reached a state where most of the work that is left to be done is just in tuning the accuracy of the algorithm. The focal length prediction model has at this point been researched to an extent where we feel comfortable in proceeding with developing and testing prototypes. We are confident in the design of our solution, and given how rapid work on the project has progressed up to this point, we feel comfortable in saying that the nine men whose names are visible in the historical photograph will be on the appropriate gravestones by the time this project is complete.

References

[1] "Saint Augustine National Cemetery." https://www.loc.gov/pictures/item/2018671309/; The Library of Congress.

[2] T. Mathworks, *Structure from motion overview*. https://www.mathworks.com/help/vision/ug/structure-from-motion.html.

[3] P. Gantelius, *fSpy: basics*. Stockholm, Sweden: fspy.io/basics.

[4] J. Sochor, "Traffic analysis from video," Master's thesis, Brno University of Technology, 2014.

[5] P. Moulon, *openMVG: "Open multiple view geometry"*. http://imagine.enpc.fr/~moulonp/openMVG/.

[6] *OpenCV*. Opencv.org.

[7] *NumPy*. Numpy.org.

[8] *SciPy*. Scipy.org.

[9] K. Hata and S. Savarese, "CS2321A course notes 1: Camera models." https://web.stanford.edu/class/cs231a/course_notes/01-camera-models.pdf; Stanford Vision; Learning Lab.

[10] K. Hata and S. Savarese, "CS2321A course notes 2: Single view metrology." https://web.stanford.edu/class/cs231a/course_notes/02-single-view-metrology.pdf; Stanford Vision; Learning Lab.

[11] X.-S. Gao, X.-R. Hou, J. Tang, and H.-F. Cheng, "Complete solution classification for the perspective-three-point problem," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2003.

[12] V. Lepetit, F. Moreno-Noguer, and P. Fua, "EPnP: An accurate o(n) solution to the PnP problem," *International Journal of Computer Vision volume*, 2009.

[13] K. Hata and S. Savarese, “CS2321A course notes 3: Epipolar geometry.” https://web.stanford.edu/class/cs231a/course_notes/03-epipolar-geometry.pdf; Stanford Vision; Learning Lab.

[14] T. Opsahl, “Lecture 7.3: Pose from epipolar geometry.” https://www.uio.no/studier/emner/matnat/its/nedlagte-emner/UNIK4690/v16/forelesninger/lecture_7_3-pose-from-epipolar-geometry.pdf.

[15] B. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon, “Bundle adjustment — a modern synthesis,” *Lecture Notes in Computer Science*, 2002.

[16] C. Loop and Z. Zhang, “Computing rectifying homographies for stereo vision,” *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1999.

[17] M. Dubská and A. Herout, “Real projective plane mapping for detection of orthogonal vanishing points,” *British Machine Vision Conference*, 2013.

[18] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, “ORB: An efficient alternative to SIFT or SURF,” *International Conference on Computer Vision*. 2011.

[19] P. F. Alcantarilla, A. Bartoli, and Andrew J. Davison, “KAZE features,” *Lecture Notes in Computer Science*, 2012.

[20] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision*, 2004.

[21] D. E. Schinstock, C. Lewis, and C. Buckley, “AN ALTERNATIVE COST FUNCTION TO BUNDLE ADJUSTMENT USED FOR AERIAL PHOTOGRAPHY FROM UAVS,” *American Society for Photogrammetry and Remote Sensing Annual Conference*, 2009.

[22] J. J. Moré, “The levenberg-marquardt algorithm: Implementation and theory,” *Lecture Notes in Mathematics*, 2006.

[23] T. Mathworks, “Stereo disparity using semi-global block matching.” https://ww2.mathworks.cn/help/visionhdl/ug/stereoscopic-disparity.html.

[24] A. Wong and S. Soatto, “Unsupervised depth completion with calibrated backprojection layers,” *International Conference on Computer Vision*, 2021.

[25] U. of Tsukuba Computer Vision Laboratory, *New tsukuba stereo dataset*. https://home.cvlab.cs.tsukuba.ac.jp/dataset, 2012.

[26] A. Geiger, *The KITTI vision benchmark suite: Stereo evaluation*. http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo, 2015.

[27] The Qt Company Ltd., Qt for Python, Accessed on: 12/5/21. Available: https://doc.qt.io/qtforpython/, 2021

[28] P. Ratan, “What is the Convolutional Neural Network Architecture?” Accessed on: 12/5/2021. Available: https://www.analyticsvidhya.com/blog/2020/10/what-is-the-convolutional-neural-network-architecture/, 2020

[29] F. Chollet, “Keras API Reference”, Accessed on: 12/5/2021. Available: https://keras.io/api/applications/, 2021.

[30] IBM, “Overfitting”, Accessed on: 12/5/21. Available: https://www.ibm.com/cloud/learn/overfitting, 2021.

[31] C. Dowlatram, S. Calderon, A. Santamaria, T. Whitaker (12/3/2019). *Stereoscopic Image Analysis* [Python source code]. Available: https://github.com/cdowlatram/Stereoscopic-Image-Analysis.

[32] K. He, X. Zhang, S. Ren, en J. Sun, “Deep Residual Learning for Image Recognition”, *arXiv [cs.CV]*. 2015.

[33] A. Sachan, “Detailed Guide to Understand and Implement ResNets”, Accessed on: 12/5/2021. Available: https://cv-tricks.com/keras/understand-implement-resnets/.

[34] S. Xie, R. Girshick, P. Dollár, Z. Tu, en K. He, “Aggregated Residual Transformations for Deep Neural Networks”, *arXiv [cs.CV]*. 2017.

[35] G. Huang, Z. Liu, L. van der Maaten, en K. Q. Weinberger, “Densely Connected Convolutional Networks”, *arXiv [cs.CV]*. 2018.

[36] F. Chollet, “Xception: Deep Learning With Depthwise Separable Convolutions”, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[37] A. Howard *et al.*, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”, 04 2017.

[38] K. Simonyan en A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, *arXiv [cs.CV]*. 2015.

[39] Davi Frossard, “VGG in TensorFlow · Davi Frossard,” *Toronto.edu*, 2016. Available: https://www.cs.toronto.edu/~frossard/post/vgg16/.

[40] M. Tan en Q. V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”, *arXiv [cs.LG]*. 2020.

[41] S. Workman, C. Greenwell, M. Zhai, R. Baltenberger, en N. Jacobs, “DeepFocal: A Method for Direct Focal Length Estimation”, in *International Conference on Image Processing (ICIP)*, 2015.

[42] H. Yan, Y. Zhang, S. Zhang, S. Zhao, en L. Zhang, “Focal length estimation guided with object distribution on FocaLens dataset”, *Journal of Electronic Imaging*, vol 26, bl 033018, 06 2017.