The Pennsylvania State University

The Graduate School

Department of Aerospace Engineering

COMPUTER VISION AND TARGET LOCALIZATION ALGORITHMS FOR AUTONOMOUS UNMANNED AERIAL VEHICLES

A Thesis in

Aerospace Engineering

by

James Alton Ross

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2008

The thesis of James Alton Ross was reviewed and approved* by the following:

Lyle N. Long
Distinguished Professor of Aerospace Engineering
Thesis Advisor

Joseph F. Horn
Associate Professor of Aerospace Engineering

George A. Lesieutre
Professor of Aerospace Engineering
Head of the Department of Aerospace Engineering

*Signatures are on file in the Graduate School

ABSTRACT

The Unmanned Aerial Vehicle (UAV) field is currently experiencing exponential growth in both military and civilian applications. An increase in the number of UAVs operating at once using more complex behavior has shown shortfalls that will require more automation in the future. This thesis discusses autonomous computer vision identification and target localization of ground targets from a UAV. The research is being pursued at the Pennsylvania State University and Applied Research Lab (ARL). Flight test results of the algorithms developed will also be presented.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Dr. Lyle Long and Dr. Joseph Horn for their academic guidance and advice. Al Niessner, Brian Geiger, and Gregory Sinsley were very helpful with, quite literally, getting this work off of the ground. I would also like to thank the Penn State Applied Research Laboratory for funding this project and the personal financial support.

Chapter 1

Introduction to Unmanned Aerial Vehicles

In general, an Unmanned Aerial Vehicle (UAV) is an unpiloted aircraft best suited for “dull, dirty, or dangerous” assignments. Dull missions are those where pilots would normally be required to stay alert in rather long and tedious legs of flight. Dirty missions include those where a pilot might be exposed to poisonous chemicals or radiation. Dangerous assignments refer to missions that might endanger the life of a pilot. UAVs have been used as weapons platforms for combat or training and, more recently, are used for real-time reconnaissance. All UAVs have some method of automated flight control known as an autopilot. The methods of control have varied over the years through technological advancement.

Typically, a modern autopilot will contain an inertial measurement unit (IMU) for attitude determination and a Global Positioning System (GPS) for position determination and waypoint navigation. The IMU usually consists of micro electro-mechanical three-axis accelerometers and three-axis gyroscopes. The noisy sensor data is usually combined in software using a Kalman filter to more accurately determine the pitch, yaw, roll, elevation, latitude, and longitude of the aircraft. Then, using a software model of the aircraft and flight controls, the software determines the appropriate control surface deflections and throttle settings for the desired behavior.

Additional sensor inputs include barometric pressure sensors, ranging sensors, magnetometers, and thermopiles. The pressure sensors are used to determine altitude and airspeed. Ranging sensors, including radar, LIDAR, and ultrasound, are used to

determine distances to targets or terrain. These sensors are important for automated takeoffs and landings. Magnetometers are used to measure the earth's magnetic field to help determine the attitude of the aircraft. Arrays of infrared (IR) thermopiles have been used in place of an expensive IMU to stabilize aircraft pitch and roll as demonstrated by the open-source hardware and software Paparazzi autopilot [**1**].

Modern UAVs usually have wireless controls for human operators on the ground to take manual control of the aircraft. These range from small handheld R/C transmitters to mobile trailer-sized ground control stations. There is usually a continuously connected wireless data link to provide real-time aircraft telemetry data and other sensor measurements. Images and video from onboard cameras are usually transmitted to the ground station for viewing or processing.

As the number of UAVs flying simultaneously increases, coupled with additional onboard sensory input and flight capabilities, the operator workload grows. To deal with this problem, the U.S. Air Force has removed pilots from their operations in the airspace over the Middle East and placed them behind UAV controls in the US. This is only a short-term solution, however. The long-term solution to the increasing operator workload is to create more automated and intelligent platforms. The military has a keen interest in real-time, autonomous target identification and precise target localization [**2**]. The work presented in this thesis demonstrates a UAV autonomously identifying and localizing a target on the ground.

1.1 History of UAVs

For nearly as long as humans have been able to fly, there has been an effort to automate flight. In general, a UAV is flown remotely or uses automated flight systems, and comes in a large range of aerial platforms. There are three general technologies that were required to meet the definition of a modern UAV: automated stabilized flight, wireless remote controls, and automated navigation. Militaries around the world have had much influence in the development of UAV technology. There is a great deal of technological overlap between modern UAVs and cruise missiles. Cruise missiles or aerial torpedoes are only a subset of UAVs and were, for much of their early history, the primary function of UAVs. Unmanned target drones were developed later for target practice by aerial, ground, and sea based systems. Decoys, the obvious practical extension to target drones, were developed to help their manned mission counterparts to penetrate enemy airspace. Air-reconnaissance vehicles, those UAVs with onboard camera systems, have been used to identify and monitor enemy activity. In terms of numbers of UAVs, reconnaissance is the most common application in the US today. More recently, armed UAVs are carrying out missions traditionally performed by bomber or fighter pilots. UAVs are also being used for remote monitoring in research.

1.1.1 Early UAVs

The earliest UAV, demonstrated by Laurence Sperry in 1914, used a crude three-axis gyroscopic stabilizer for attitude control [**3**]. While this was a remarkable technical achievement, it never went into production. His father, Elmer Sperry, and Peter Hewitt

convinced the U.S. Navy of the practical applications of a new autopilot design and obtained funding for the production of an aerial torpedo in 1916 using Navy-furnished N-9 seaplanes (Figure **1**).

Figure **1**: U.S. Navy Curtiss N-9 Seaplane [**4**]

It was to combine both gyro-stabilized flight controls adapted from naval battleships and newly developed electronic radio controls, however the radio controls were never used in the project. The gyros controlled servo-motors which adjusted the elevator and ailerons. Additional input from an aneroid barometer controlled the altitude of the aircraft. There was an additional mechanism that roughly counted distance traveled. Once a set count was reached, an additional mechanism would either drop a bomb or aim the ordinance-laden aircraft downward. Elmer Sperry demonstrated these technological capabilities in September of 1917.

In October, Sperry commissioned the Curtiss Aeroplane and Motor Company to deliver six special planes for production of a flying bomb in just 30 days. The plane was to weigh 500 pounds and be able to carry an explosive payload of 1000 pounds. The design was unproven and after a few attempted flight tests it was realized that more

needed to be known about the flight characteristics. Dubbed the "Curtiss-Sperry Flying Bomb" (Figure **2**), the UAV was never utilized in a military situation and was abandoned after disappointing early results. The US Navy continued development on the Sperry's work after the end of WWI. More research work was focused on radio control (RC) at what would later be called the Naval Research Laboratory.

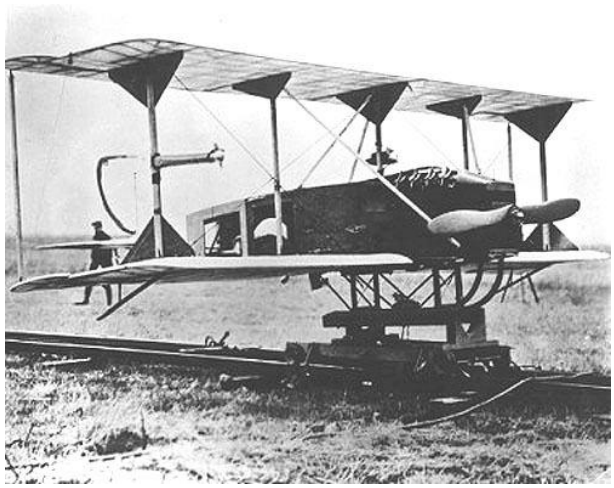


Figure **2**: A Curtiss/Sperry Flying Bomb on a Catapult Rail Launch [**5**]

The US Army also had an interest in cruise missiles near the end of WWI. The Kettering Bug was the result. It used a barometer to climb to an altitude and maintain level flight in addition to a gyroscope to maintain a heading. One of the early flights on October 1, 1918 was impressive enough to convince the Army to purchase 75 additional Bugs before 1919. It had climbed to approximately 12,000 feet and flew for an estimated 100 miles before crashing. Modifications were made to improve the stability and, three weeks later, the Bug took off, flew to 200 feet altitude and 1500 feet downrange before diving onto the intended target. It managed to strike only a few feet away. After the

armistice to WWI, development of the Kettering Bug was cancelled. Only 36 had been built at that point, but it had become the first production UAV [**6**].

1.1.2 WWII Era UAVs

With the exception of the addition of radio control, the fundamentals of autopilot design changed very little until close to WWII. Radio-controlled aircraft were developed by both the US and British for use as target drones in the 1930s. In 1931, the British Fairey Aviation Company Limited developed the RC Fairey "Queen" target drone based on the Fairey IIIF. Four years later, the British began mass production began on another RC target drone based on the de Havilland Tiger Moth biplane trainer (Figure **3**) dubbed "DH.82H Queen Bee"[**7**]. A few hundred of these drones were built and used as targets for British warships.

Figure **3**: DH.82 Queen Bee Based on de Havilland Tiger Moth Biplane [**8**]

The US Navy and Army had a need for target drones in the early part of WWII. The "OQ-2" (Figure **4**) was a the final result of a scale RC airplane initially developed

and demonstrated by Reginald Denny at his model airplane business, Radioplane. The OQ-2 was manufactured by the thousands in 1941 for training aircraft gunners. It was an inexpensive little aircraft with a wingspan of 3.7 meters and a takeoff weight of 47 kilograms. It could be landed with RC controls or a parachute. The Radioplane company was eventually purchased by Northrop [**7**]. The US military also attempted to modify B-17 and B-24 bombers with radio controls during the war, however this wasn't very successful.

Figure **4**: Radioplane OQ-2 Developed by Reginald Denny [**7**]

The German Vergeltungswaffe 1 (V-1), developed during WWII, used a gyrocompass and pendulum for attitude control. A vane anemometer was used to determine the traveled distance. A powerful pulse jet engine was used to yield its high speed, earning it the notorious "buzz bomb" name. There were no RC controls for this cruise missile and it was only accurate enough for area bombing. Approximately 10,000 V-1s were launched during the war, primarily towards the British. While inaccurate, enough of the missiles hit their intended targets to cause the Allies to devote a lot of resources to defend against them [**6**].

1.1.3 Cold War Era UAVs

During the Cold War, a number of radio controlled UAVs with autopilots were developed as targets, decoys, research platforms, and reconnaissance. Also during this period, advancements were made in cruise missiles for heavy nuclear payloads. The development of the inertial navigation system (INS) added another important piece to modern UAV autopilot design. An INS is used to determine position, velocity, heading, and orientation by continually integrating measurements from accelerometers and gyroscopes. After initialization, velocity in the inertial frame is calculated by integrating the acceleration and position is calculated by integrating velocity. An accurate INS will have a high frequency of measurements and low noise. Similarly, angular velocities and orientations are calculated by integrating gyroscope accelerations. Errors add up over time and must be corrected.

In the early 1960s, Rudolf Kalman developed a continuously updating filter that takes a series of intermittent and noisy measurements from a dynamic system and estimates the state of that system [**9**] [**10**]. The Kalman filter utilizes knowledge about the dynamics of the system and control surfaces and couples them with the measurements to determine a more accurate state after a period of time. This advanced control theory and lead to improvements in autopilot design. Towards the end of the Cold War era, GPS was completed and operational. This enabled highly accurate positioning with coupled GPS-INS devices.

The primary use of UAVs during the Cold War was for reconnaissance. Manned airplanes like the U-2 spy plane were putting pilots at risk of being shot down and killed

by surface-to-air missiles. Low-observable reconnaissance drones were of interest to the US Air Force. Ryan Aeronautical had developed the Firebee drone in late 1950s and then worked on increasing the range and lowering its radar signature. A U-2 was shot down on May 1, 1960 and the Soviet Union used the captured pilot for political gain. An embarrassment to the US, Ryan Aeronautical was awarded a contract to develop their Firebee further for operational use in long range reconnaissance missions shortly after the incident. The simple autopilot was based on a timer-programmer, a gyrocompass, and an altimeter. The aircraft could be launched from the air, would travel in a specified direction for a set amount of time and then turn around and return. A parachute was deployed at the end of the mission for recovery. The development program stalled a few times with changing presidential administration, but after several versions of the airplane, the Ryan Model 147B "Lightening Bug" reached operational status in July 1963. The Model 147B had a large wingspan (8.2 meters) and could fly at altitudes up to 19 kilometers [**11**].

Many reconnaissance missions were flown over southern China and North Vietnam starting on October 11, 1964. Several different models with varying engines, airframes, and radio controls were built for flying different kinds of missions at various altitudes and speeds. Many Lightening Bugs were shot down, but the US media remained unconcerned because there was no crew. This demonstrated one of the biggest advantages of using drones instead of manned spy planes. There were 3,435 missions flown with the various Lightening Bug models, of which 578 were lost on missions through 1975. More than half of those lost had been shot down by China, North Korea, or North Vietnam [**11**].

1.1.4 Modern Military UAVs

The Global Positioning System and the miniaturization of the receivers are responsible for significant technological improvement in autopilot design and performance. The exponential growth in processing power of microprocessors has also been a huge factor in increasing the capabilities of UAVs. A recent Pentagon report stated that the operational use of UAVs has doubled in the nine months before October 2007 [**12**]. The acronym UAS, meaning unmanned aircraft system, has been used more recently by the military in recognition that there are many systems used in the deployment of UAVs, including communication systems and ground control stations.

In terms of numbers of UAVs in service, the RQ-11 Pathfinder Raven (Figure **5**) is the most produced with 10,000 aircraft planned for delivery in 2007. Developed by AeroVironment, Inc., this small electric (4-5 lb) aircraft has an endurance of 90 minutes and a ceiling of 14,000 feet. The aircraft is hand launched (because it doesn't have any landing gear) and is usually undamaged after crash landing. The UAS is used to support battalion level maneuvers and smaller with situational awareness and force protection mostly for the US Army and Marines. The typical payload consists of two fixed cameras; one forward looking and the other side looking. These can be visible or infrared cameras depending on the mission requirements. According to a project manager at AeroVironment, the video cameras have a super high resolution of five megapixels. Three UAVs can be used with one base station and must maintain continuous line-of-sight (LOS). They can operate up to six nautical miles from the base station and be flown remotely or autonomously with the onboard autopilot [**2**].

Figure **5**: RQ-11 Pathfinder Raven [**2**]

The MQ-1 Predator (Figure **6**), developed by General Atomics Aeronautical Systems, Inc., is another popular UAS that has logged 170,000 hours as of July 2006. It began service in 1994 and has served as a real-time surveillance platform in Bosnia, Kosovo, Afghanistan, and Iraq. The 55-foot wingspan Predator has a gross weight of 2250 pounds, is powered by a 115 horsepower Rotax 914F engine, and has a radius of 500 nautical miles. It has gimbaled visible and infrared cameras. The US Air Force added Hellfire missiles and a laser designator to the aircraft in 2001 to be able to attack ground targets. It uses a continuous satellite data connection for remote control and transmitting sensor data measurements. Thus, the operational range is only limited by the amount of fuel it can carry. As of 2007, 170 Predators are planned to be built for the US Air Force, Navy, and Army [**2**].

Figure **6**: MQ-1 Predator with Hellfire Missiles

Now that flight stability, remote control/communication, and waypoint navigation technologies have sufficiently advanced, the Pentagon would like improvements in the electronic payloads onboard UAVs. Securing communications and vehicle command links is an important step. Additional technologies will be needed for a UAV to assess its near-field of view for other aircraft, which can be several kilometers around it. Autonomous and continuous classification of objects such as ground vehicles and humans is also important for the future. Standards are needed with data links and sensor data flow. The Department of Homeland Security has also specified functional capabilities that UASs will provide in the future that other systems cannot provide as well. These include visual/nonvisual monitoring, target geolocation, communications interception, tactical situational awareness, pursuit management, intelligence support, visible security systems, and enforcement operations [**2**].

1.1.5 Modern Civilian Uses

The military has sought UAVs for nearly a century, while the civilians have only relatively recently considered their applications for agricultural, scientific and entertainment purposes. With the advent of smaller, less expensive, and more powerful electronic control technologies, autopilot technology has become available to a much larger market. Improvements and lower costs in microelectronic inertial measurement units (IMUs) have brought autopilot capabilities to the masses.

Civilian use of UAVs has grown dramatically in the past decade. Many radio control hobbyists have begun developing their own autopilots for under $1,000 using commercial off-the-shelf (COTS) components [**13**]. The Paparazzi autopilot is a very inexpensive open source and open hardware design [**1**]. Quite a number of commercially available autopilots exist. Table **1** contains a partial list of those autopilots. There is a large price and functionality range. All of the autopilots in table **1** are designed to primarily work with scale RC airplanes using standard servo connections to command the aircraft control surfaces. Autopilot development is thus vehicle independent and is flexible enough to work within a variety of airframes.

Table **1**: Commercially Available Autopilots

Autopilot	Company	Size (mm)	Weight (g)	Website
TGE	Continental Controls & Design	30x30x18	13	www.continentalctrls.com
MP2028	MicroPilot	100x40x15	28	www.micropilot.com
PICOPILOT	UNAV	51x25x33	48	www.u-nav.com
Piccolo Plus	Cloud Cap Technology	142x76x61	212	www.cloudcaptech.com
Kestrel	Procerus Technologies	51x35x12	17	www.procerusuav.com
BTA AS-07G	Maxx Products	102x51x19	85	www.maxxprod.com
Tiny2.11	Paparazzi UAV	71x40x12	n/a	www.onefastdaddy.com
MNAV	Crossbow	57x45x22	33	www.xbow.com

There are also several complete UAV systems commercially available. In general, they're marketed to law enforcement, researchers, farmers, and enthusiasts. Procerus Technologies, developer of the Kestrel autopilot, has also developed the Unicorn UAV Test Platform. The Unicorn is an electric foam wing that can carry two fixed cameras or a small gimbaled camera. There is a base station for waypoint control and viewing transmitted video. The CropCam is a UAV targeted at consumers in the agriculture market. It uses pre-programmed waypoints to fly a pattern over an area which it will photograph with a Pentax still camera. The images are later stitched together to create mosaics of the flight area for analysis. The Dragonflyer Stabilized Aerial Video System is a gyro-stabilized RC quad-rotor helicopter that wirelessly transmits NTSC video from an onboard camera. The Dragonflyer is primarily for enthusiasts. A company called Cyber Defense Systems has developed a few complex UAVs targeted at the law enforcement market.

1.2 Imaging and Surveillance Systems

Most modern UAVs have optical cameras, while some use infrared (IR), thermal, or radar-based surveillance systems. Many camera systems transmit video and images wirelessly to a base station on the ground. Many UAVs have gimbaled camera systems and some of the gimbals contain their own IMU to maintain orientation to some coordinate system. The cameras may also have the ability to change the optical zoom to focus on a narrower field of view.

Commercially available wireless cameras are relatively inexpensive and transmit a low resolution analog NTSC video signal using a variety of commercial transmitters and receivers. Transmitters are limited by FCC regulations to one watt in most situations, which limits the range to a few miles. There can be some video degradation with LOS or long range transmission. Receivers on the ground can be connected to computers or video recording devices.

The MQ-1 Predator (Figure **6**) uses the AN/AAS-52 Multi-Spectral Targeting System (MST) built by Raytheon for target acquisition and laser designation (Figure **7**). This 18 inch diameter, 130 pound gimbaled system contains a forward-looking infrared (FLIR) camera, and TV cameras (both IR and color). It has the capability for ultra-narrow fields of view for long-range surveillance. In the case of the Predator, the video is transmitted back to the ground control station using a satellite data connection.

Figure **7**: AN/AAS-52 Multi-Spectral Targeting System by Raytheon [**14**]

Cloud Cap Technology has a family of small (4.4 in. dia.), inertially stabilized gimbaled cameras called TASE. They have both color and IR video sensors with zoom capability and NTSC output for wireless transmission to a ground station [**15**]. These cameras can be specified to lock on to a specific point on the ground or stay fixed on a specified vector in the earth inertial frame.

Micro UAV makes less expensive gimbaled cameras (Figure **8**) that are used with the Proceres Kestrel autopilot OnPoint Targeting technology. There is no inertial stabilization, so the camera view bounces around during flight. To compensate, the OnPoint system uses some basic video stabilization to keep the view from moving around so much. It's coupled with some basic object tracking computer vision processing where a user specifies an object to follow on the video stream. The vision processing tries to keep a bounding box around that object while calculating its coordinates based on the state of the UAV and the orientation of the camera. The camera automatically adjusts the pan and tilt to keep the object in view. The UAV then goes into an orbit mode and circles around the target after an initial position estimate. After a single revolution around the target, Proceres claims to have a target localization estimate accurate to 3-5 meters [**16**].

Figure **8**: Micro UAV NTSC Camera and Gimbaled System

1.3 Onboard Intelligence

While not required for autopilot design, there have been demonstrations of artificial intelligence (AI), using fuzzy logic, applied to flight controls in UAVs [**17**]. Onboard computer artificial intelligence is a growing part in autonomous vehicle research, particularly computer vision algorithms and sensor data fusion. One functioning example of this is the unmanned ground vehicle (UGV) developed at Carnegie Mellon called "Crusher". The sensory input from onboard stereo cameras and LIDAR systems is combined to identify and avoid obstacles in its path.

The ARL/PSU Intelligent Controller (IC) architecture has been integrated into the UAV flight computers. It has been simultaneously flight tested on two UAVs to show that they were capable of collaborating on a mission. The IC is comprised of two main parts: perception and response. The perception module receives various sensor inputs and processes the information using AI algorithms. The appropriate behavior is determined and passed on to the response module. Once the appropriate behavior is chosen, it executes the action. A graphical depiction of this process can be found in figure **9**. Additional information about the IC may be found in the papers by Weiss [**18**] and Sinsley, et al [**19**].

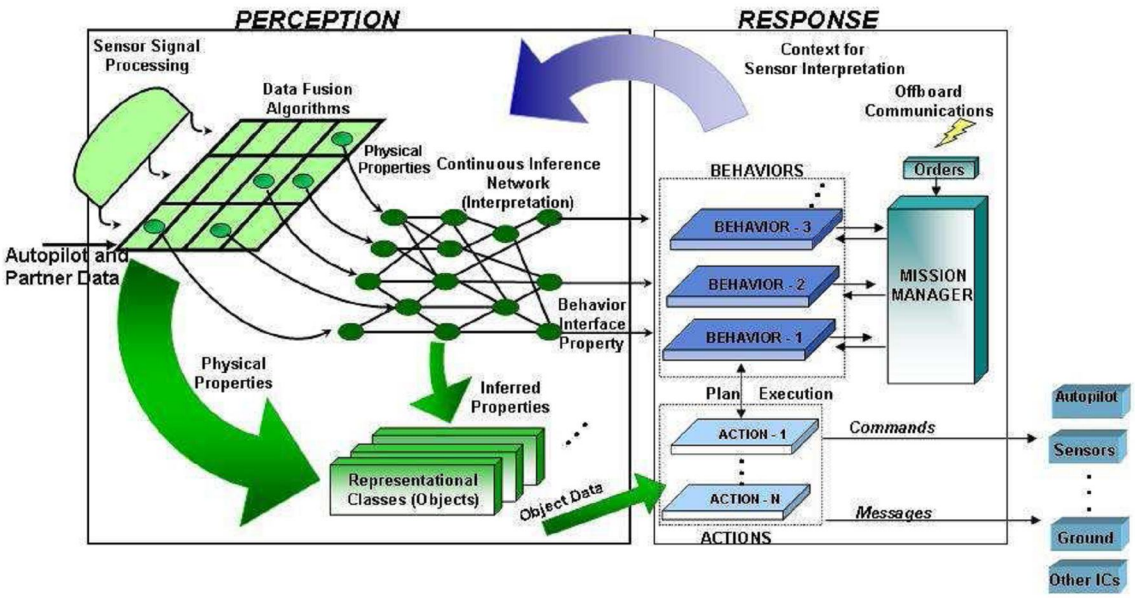


Figure **9**: Graphical Depiction of ARL/PSU IC Architecture [**19**]

1.4 Contributions from this Thesis

The US DoD has specified in the Unmanned Systems Roadmap [**2**] that there is a shortfall in UAV computer vision systems that can identify and geolocate military targets in real-time. This thesis will discuss the research work performed in autonomously identifying two red balls on the ground and determining their geographical location in real-time. It will discuss the computational methods used for identifying the target and some aspects of other methods. The algorithm for determining the target position in a single image frame will also be explained. In the future, more advanced computer vision algorithms, executed on more powerful computers, will need to be able to autonomously identify more complex objects.

Chapter 2

UAV Platforms at ARL/PSU

2.1 SIG Kadet Senior

The SIG Kadet Senior aircraft is a trainer class model airplane modified from an Almost-Ready-to-Fly (ARF) kit. The modified aircraft the current workhorse of the UAV team (Figure **10**). It has a relatively large open area inside the fuselage for the electronics. The SIG Kadet Senior is rather stable and can fly at low speeds (approximately 20 knots stall speed for the heavier UAV). In total, there are four aircraft, two of which have Piccolo Plus autopilots and onboard computers. Table **2** lists the aircraft technical specifications.

Figure **10**: Modified SIG Kadet Senior UAVs

Table **2**: Modified SIG Kadet Senior Technical Specifications

Wingspan	80	inches
Length	64.75	inches
Wing Area	1180	sq. inches
Empty Weight	6.5	pounds
Gross Weight	~14	pounds
Wing Loading	1.7	pounds/ft^2
Engine	0.91	in^3 (4-stroke)
Servos	6	(4 channels)

There have been several modifications to the basic radio control (RC) airplane kit to create the UAVs. A removable pitot-static tube and mount have been modified to fit inside the wing. Rubber tubing was run from the mount to the autopilot in the fuselage. Heavier landing gear was added so the aircraft could support the larger payload. The standard 0.40 cid glow engine was upgraded to an OS FS-91 Surpass four stroke engine (Figure **11**) to provide higher thrust. A Zinger 12-6-10 variable pitch wooden propeller is used for higher thrust. Three mounts were added for the GPS receiver antenna, Piccolo Plus autopilot 900 Mhz transceiver antenna, and 2.4 Ghz long-range wireless network antenna for the onboard computer (Figure **12**). Mounts for the onboard computer and autopilot were also built and placed inside the fuselage (Figure **12**). The rudder and elevator servos were moved back to the tail so they were out of the way in the main fuselage area. An additional servo controls the nose gear instead of being coupled with the rudder. The 24 ounce fuel tank was upgraded to 32 ounces. Power switches were mounted inside the fuselage with a master switch on the outside. More information on this airborne system is available in Miller, et al [**20**].

Figure **11**: OS FS-91 Surpass Four Stroke Engine [**21**]

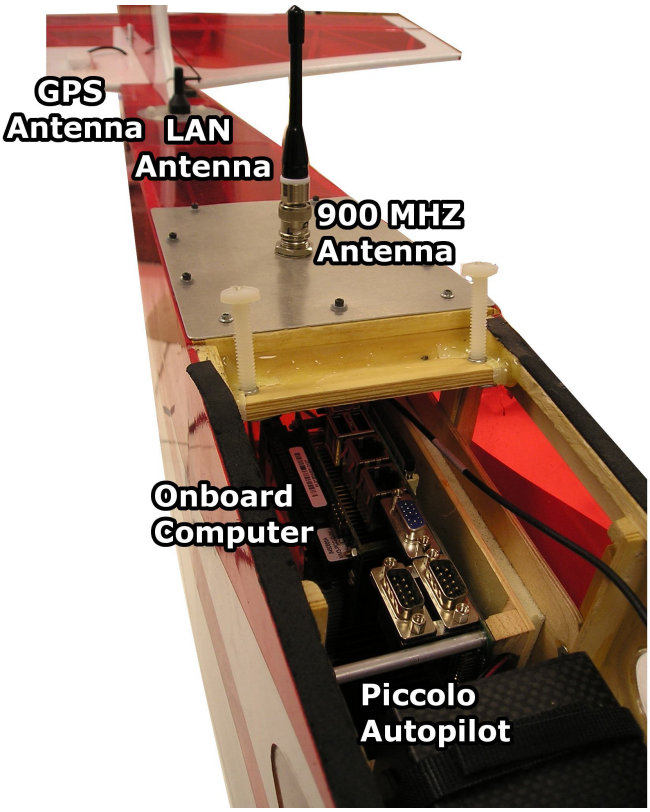


Figure **12**: Modifications to SIG Kadet Senior ARF Kit

2.2 BTE Super Flyin' King

The SIG Kadet Senior has been a good platform, but has limited the use of some larger gimbaled cameras and other sensors. Additionally, there have been incidents when the Kadet Senior has had an engine stall during flight. Engine vibration has also caused distortion in still images and digital video. Two Bruce Thorpe Engineering (BTE) Super Flyin' Kings (SFKs) are currently under construction at the ARL/PSU UAV lab. The SFK is a large balsa and plywood RC kit plane with an optional twin engine configuration. This larger plane features a greater payload capacity (20 pounds), electric propulsion, and a smoother, more stable platform. The larger fuselage will be able to carry sensors and payload that were too large for the Kadet Senior airframes. To overcome the stalling engine issue, each SFK will have electric motors in the twin engine configuration. The electric motors should also decrease vibration for sharper, less distorted digital images and video.

Construction has taken a few months of part-time work, but the SFK airframe is mostly completed at this time. Figure **13** is a comparison photo to display the relative size between the SIG Kadet Senior and the BTE Super Flyin' King.

Figure **13**: Super Flyin' King and Kadet Senior Comparison

The SFK will fly with two AXI Gold 5345/16 brushless outrunner motors for propulsion. Six 4 cell 5,000 mAh "eXtreme" 14.8V lithium polymer batteries will be used for each motor. Two sets of three battery packs connected in series will be connected in parallel for an effective 10,000 mAh 44.4 V battery in each wing. Each motor has a Phoenix HV-85 electronic speed controller (ESC) that is capable of providing 85 amps at 50 volts. So the total output of each motor can be up to approximately 4 kW. At full throttle, this would yield a flight time of about 7 minutes. Fortunately, the SFK will not need the full motor output except during takeoffs. It can fly at slow speeds and has flaps for increased lift. This configuration is hoped to yield a flight time of at least an hour. Initial motor runs have measured the thrust from two different propellers using the battery, ESC, and motor configuration stated above. The Zinger 18x10 two-blade propeller yielded 15 pounds of thrust at full power and NX 20x8 two-blade propeller yielded 19 pounds of thrust. Technical specifications of the SFK airframe appear in table **3**. An additional three-view diagram appears in figure **14**.

Table **3**: Modified Super Flyin' King Technical Specifications

Wingspan	132	Inches
Length	95	Inches
Wing Area	3380	sq. inches
Empty Weight	~36	Pounds
Gross Weight	~44	Pounds
Wing Loading	1.9	pounds/ft2
Servos	6	(5 channels)

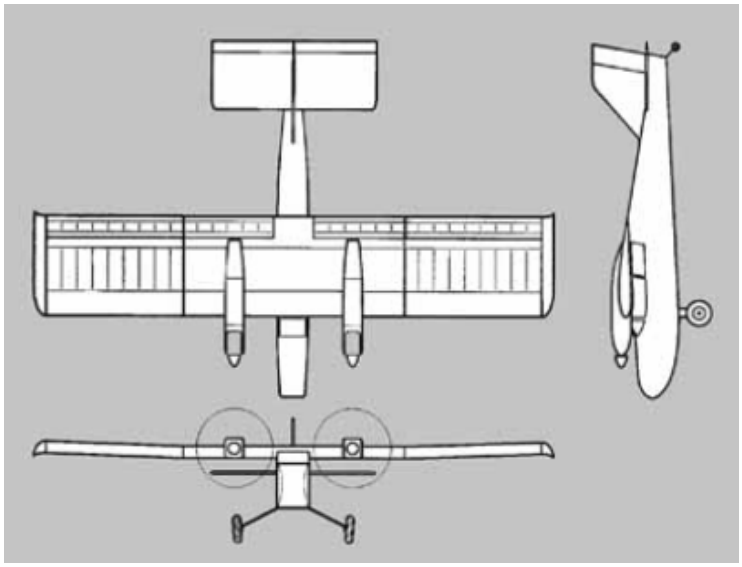


Figure **14**: Three-view Super Flyin' King

Chapter 3

Electronic UAV Equipment

The ARL/PSU airborne platforms are equipped with a lot of standard RC electronics in addition to a Cloud Cap Technology (CCT) Piccolo Plus autopilot, an Ampro ReadyBoard 800 Single Board Computer (SBC), and either a Logitech QuickCam Ultra Vision USB Webcam or a digital still camera. Figure **15** is a general diagram of the electronics onboard the SIG Kadet Senior UAVs.

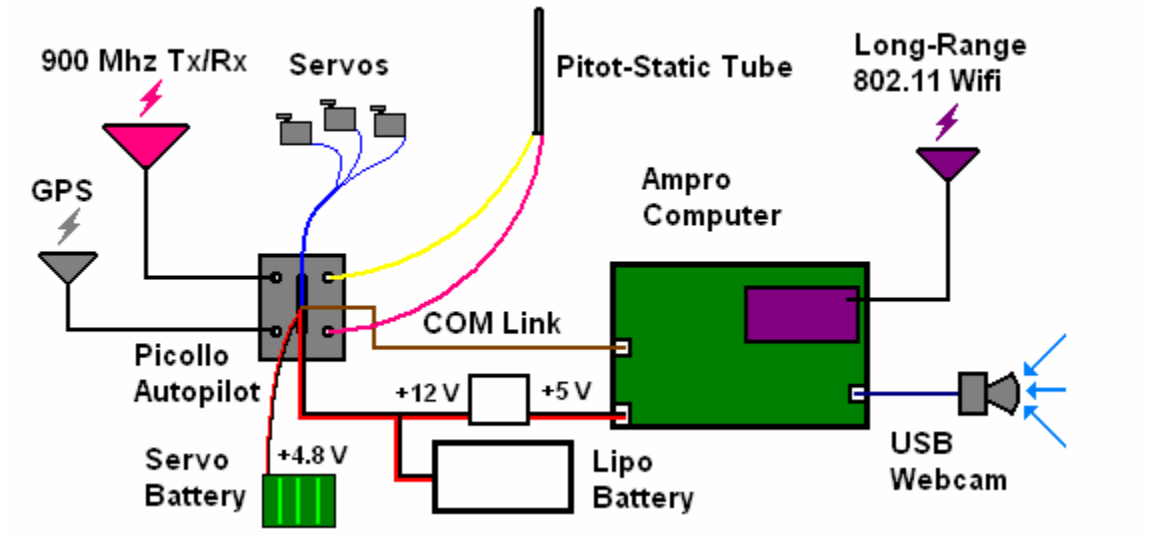


Figure **15**: Electronics Hardware Diagram

3.1 Onboard Computers

Each UAV contains an Ampro ReadyBoard 800 SBC (Figure **16**). This is a small (4.5x6.5”) computer with a 1.4 GHz Low Voltage Pentium M738 CPU (512 kB L2 cache). It contains 1Gb PC2700 DDR333 SODIMM DRAM and a 4Gb Compact Flash (CF) memory card for storage. It runs a modified version of the Windows XP operating system. An additional 802.11b Long Range Wireless LAN PC card is mounted on the board using the PC-104 connection. This LAN connection has been very reliable during flight tests and has never been lost at the ground station.

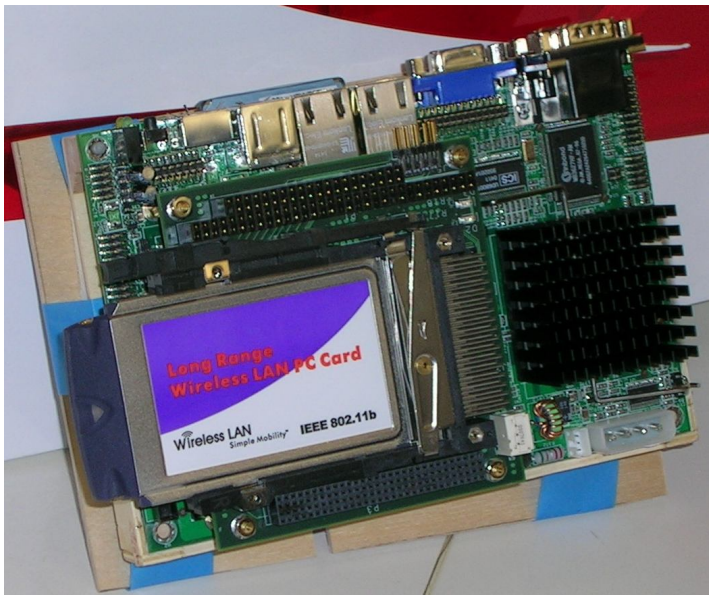


Figure **16**: Ampro ReadyBoard 800 Single Board Computer

The onboard computer can be turned on and off with a custom circuit wired into the Piccolo Plus autopilot and the power source. It’s as simple as toggling the “Enable/Disable Lights” button on the CCT Operator Interface. After giving the onboard computer a couple minutes to boot during flight, software is executed on the onboard

computer by logging in using the Windows Remote Desktop Connection software. One of the four serial ports is connected to the Piccolo Plus autopilot to receive telemetry data and send commands.

3.2 Cameras

The onboard computer has four USB 2.0 ports for peripherals. One of these is used for the Logitech QuickCam Ultra Vision webcam (Figure **17**). This fixed focus 1.3 megapixel CMOS camera has a rather wide horizontal field of view for a camera at 75 degrees. The vertical field of view is 54 degrees. It is powered by the USB connection so there is no need for an external power source. The video resolution is 640x480 pixels and looks sharper than the previous Logitech QuickCam Fusion model used onboard. There have been improvements with newer models but the Ultra Vision has been sufficient so far. Higher resolutions are better for picking out finer details from the aerial photography; however, this would slow down image processing codes since the onboard computer has a relatively slow 1.4 Ghz single core processor.

Figure **17**: Logitech QuickCam Ultra Vision [**22**]

The QuickCam Ultra Vision is modified from its manufactured form so that it can be mounted in the UAV. This consisted of taking apart the external casing and removing the monitor mounts so that only the internal circuit board and USB connection remained. Then a small box for the electronics was built out of plywood so that it could be fit squarely. The plywood mount on the underside of the UAV has an additional layer of foam rubber to dampen some of the vibration from the engine. Photographs of the camera mount appear in figures **18** and **19**. A square hole was cut out in the bottom of the mount and a piece of Plexiglas was added to keep dirt and engine exhaust off of the camera lens.

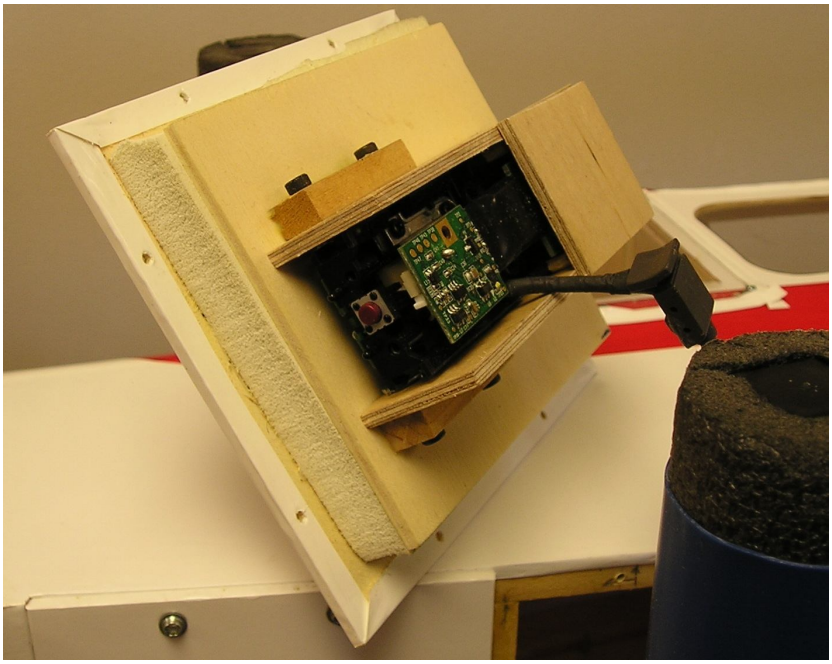


Figure **18**: Webcam Mount Back Side.

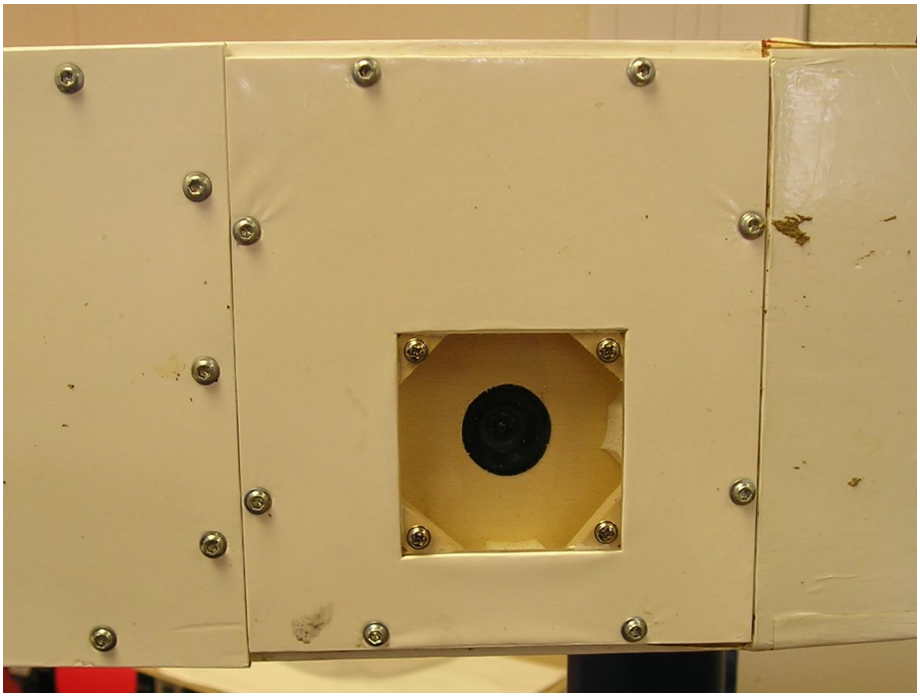


Figure **19**: Underside of UAV with Webcam Installed

One of the new BTE Super Flyin' King UAV will likely have a gimbaled camera system installed. The Sony EVI-D70 Conference Camera (Figure **20**) is a relatively inexpensive camera with quality optics. It's designed to hang upside-down. Table **4** lists the camera specifications.

Table **4**: Sony EVI-D70 Conference Camera Specifications

Sensor	1/4 type 380k pixel EXview HAD CCD
Pan/Tilt Range	Pan: ± 170º (100º/sec) Tilt: -30º to +90º (90º/sec)
Zoom	18x Optical
Video Output	NTSC - 768 (H) x 494 (V) pixels
Field of View (H)	2.7° (tele end) - 48° (wide end)
Dimensions	5-1/4 x 5-3/4 x 5-3/4 inches
Weight	2.125 pounds
Power	12W @ 10.8 to 13.2 VDC

Figure **20**: Sony EVI-D70 Conference Camera [**23**]

The camera pan, tilt, zoom, and other features can be controlled via an RS232 serial connection using the Sony VISCA protocol. An earlier, less expensive Sony camera model, the EVI-D30, was purchased to test this control mechanism on the onboard computer using this protocol. Code was written using an open source VISCA protocol library resulting in successful control of the camera. The camera view will be stabilized by adjusting the pan and tilt angles based on the current attitude estimate from the autopilot telemetry stream. Since this camera has NTSC video output, the analog video can easily be transmitted using an inexpensive wireless TV transmitter. However, for onboard image and video processing, the image data is captured using a component video (NTSC) to USB webcam converter. This makes the video stream function much like the Logitech webcam.

An aluminum scissor lift mechanism has been designed in SolidWorks which will raise and lower the entire camera unit in and out of the nose area of the fuselage. The SFK has a tail-dragger configuration and the electric propulsion motors are mounted on the wings, so the forward nose area of the fuselage is ideally suited to house the camera and lift mechanism. It will use two Futaba high-torque servos to manipulate the lift. The model and scissor lift mechanism is displayed in figure **21**. The system will be controlled much like retractable landing gear on RC airplanes.

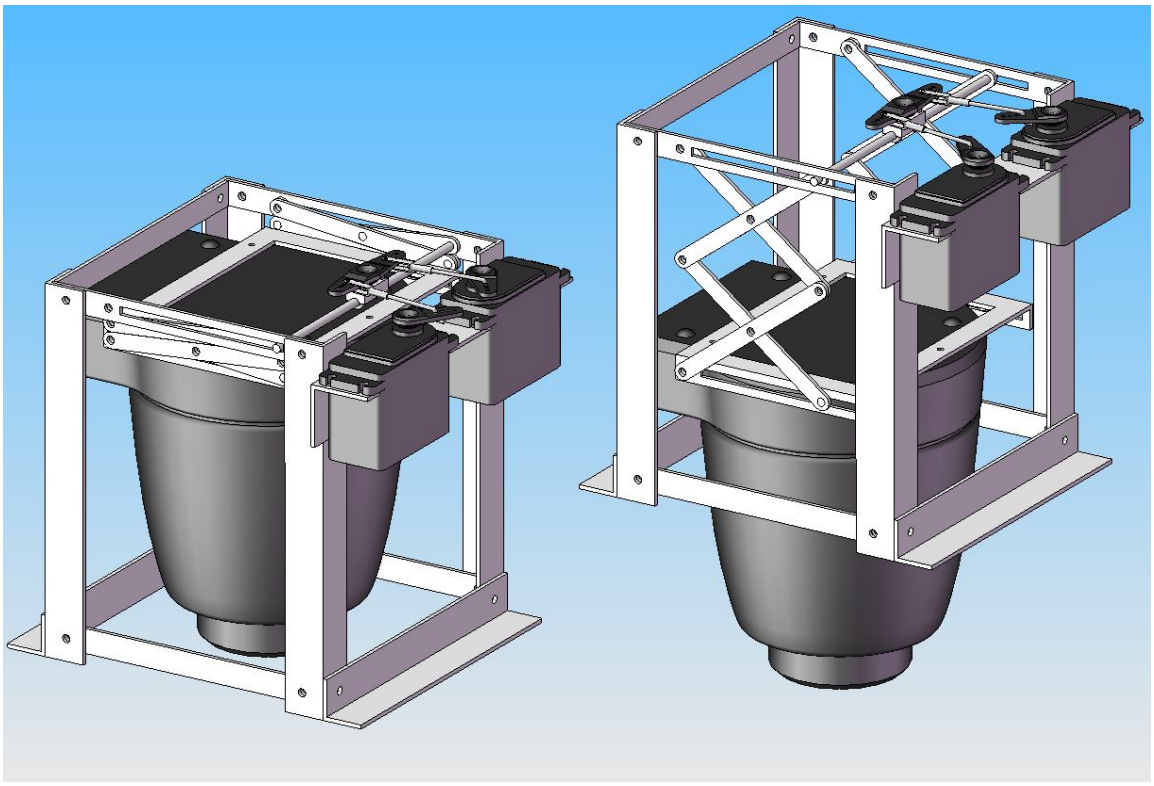


Figure **21**: Custom Camera Scissor Lift Mechanism

3.3 Cloud Cap Technology Piccolo Plus Autopilot

The Piccolo Plus autopilot (Figure **22**) is used for autonomous flight and aircraft stability. The autopilot primarily consists of a 900 MHz transceiver, barometric airspeed and altitude pressure sensors, a GPS module, inertial measurement unit with three-axis accelerometers and gyroscopes, five servo command outputs, two serial data connection, and a 40 MHz embedded processor. This is all mounted inside a carbon fiber box to minimize electromagnetic interference (EMI). The total weight is 212 grams. The GPS antenna, transceiver antenna, and rubber tubing leads from the pitot-static tube are connected to the front face of the autopilot box. The serial connections, servo outputs, power supply inputs, and deadman switch are all built into a 44-pin Standard D-Sub connector. An autopilot hardware diagram is displayed in figure **23** . More information about the autopilot hardware can be found in the CCT 2004 Catalog [**24**].

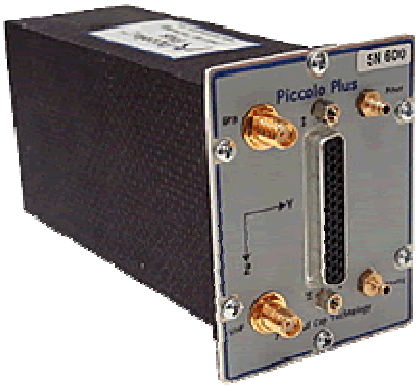


Figure **22**: CCT Piccolo Plus Autopilot [**15**]

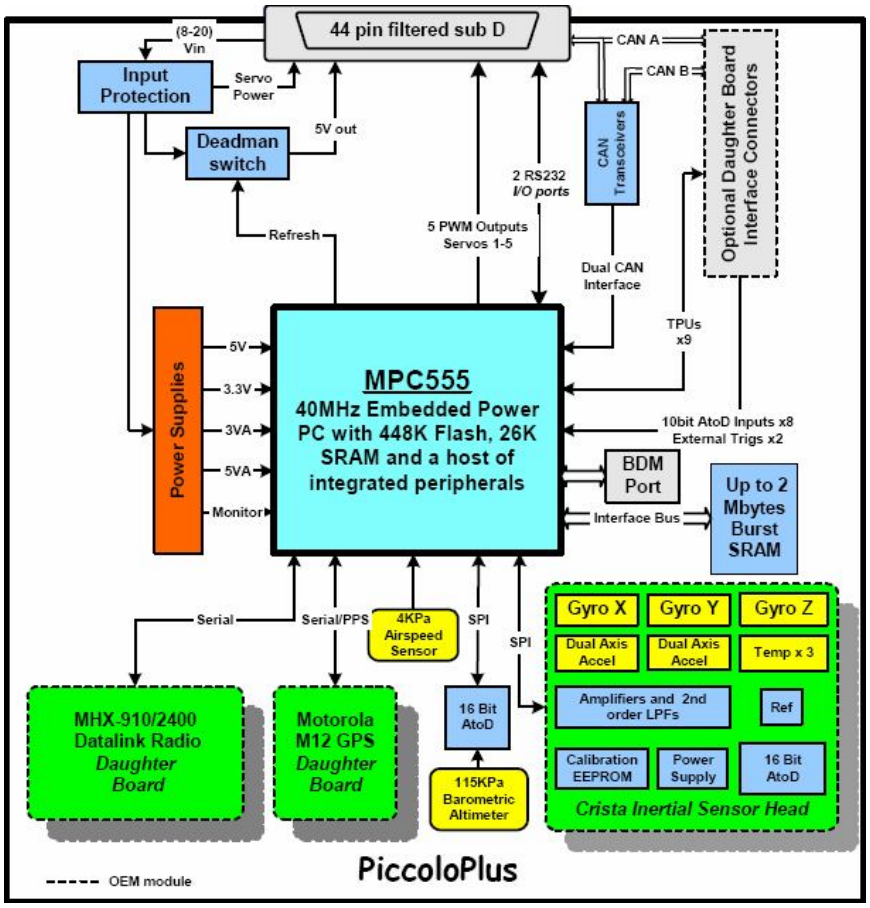


Figure **23**: CCT Piccolo Plus Hardware Diagram [**24**]

The autopilot is located near the center of mass of the aircraft. The elevator and rudder servo leads and the GPS antenna cable run by the onboard computer. This was causing EMI issues resulting in loss of GPS satellite lock and servo jitter. It is speculated that the 1.4 GHz processor is close enough to the GPS L1 frequency (1575.42 MHz) that there was interference; however, the servo jitter issue remains a mystery. To overcome this problem, the cables were wrapped in a carbon fiber tube to shield them from the computer EMI.

3.3.1 GPS

The autopilot and CCT Ground Station both use a Motorola M12 Oncore GPS module. It can typically obtain a satellite fix from a cold start within 60 seconds and 15 seconds for a hot start. It reports NMEA 0183 messages (GGA, GLL, GSA, GSV, RMC, VTG, ZDA) to the autopilot microcontroller at 4800 baud in addition to a couple other proprietary protocols. The positioning accuracy rating is rather poor for a modern GPS receiver at 25 meters SEP, or Spherical Error Probable. The SEP rating is a measurement of 50% probability that the actual position is within that range of the position estimate. Position estimates are calculated at 1 Hz, however, the autopilot IMU uses a Kalman filter and aircraft simulator to report GPS estimates at 20 Hz. The GPS module can track up 12 simultaneous satellites. It can be used with DGPS to improve positioning accuracy. The CCT Ground Station also uses this same module. Both modules use additional external antennas.

3.3.2 Ground Link

The ground link transceiver model, the Microhard MHX-910, operates on the 902-928 MHz industrial, scientific, and medical (ISM) radio band. It uses frequency hopping spread spectrum to maintain an uninterrupted connection and provide some resilience to sources of interference. Both the Piccolo Plus autopilots and the Ground Station have a MHX-910 transceiver and can transfer data at speeds up to 115,200 bps using a TTL or serial interface. This is certainly enough for a telemetry data transfer, but not nearly fast enough for video or images. Error correction is made automatically. Each

module has a radio power output of up to 1 watt to keep within the FCC civilian usage regulations. It's possible to transmit and receive up to 20 miles line-of-sight though flight tests have always stayed within one mile of the ground station.

3.4 Cloud Cap Technology Ground Station

The Desktop Ground Station (Figure **24**) is similar to the Piccolo Plus autopilot in the respect that it has a GPS receiver, a 900 MHz transceiver, and a serial data connection. It does not have servo outputs or an IMU. It does have two power supply connections and an onboard battery backup to ensure continuous communication with the UAVs. This is very important since both the manual controls and Operator Interface connect to the autopilots through the base station. If it fails, the UAVs would return to a loss-of-communication waypoint and orbit it until the fuel ran out and it crashes. The serial data connection is connected to a laptop at the field to run the Operator Interface. This laptop also has a long range wireless LAN card and antenna to communicate with the UAV onboard computer.

3.4.1 Operator Interface

The Operator Interface (OI) (see Figure **24**) is software written by CCT and run on a Windows computer. The OI is used to send commands to the autopilots. These commands are typically waypoints, airspeed, turn rates, altitude settings, mission limits, etc. The OI also serves as a display for the autopilot telemetry data stream. It can display

the current location of each UAV on a single 3D map. It also displays the status of several important things such as communication and GPS links. The OI does not need to be running in order to take manual control of a UAV. However, in order to take manual control with the Pilot Console, the Piccolo autopilot ID number first needs to be specified in the OI if there is more than one UAV flying at a time.

3.4.2 Pilot Console

The Pilot Console (Figure **24**) is setup just like a regular Futaba RC transmitter, except that it is interfaced directly with the Ground Station. It sends joystick and toggle settings directly to the Ground Station which are translated and sent to the autopilots using the 900 MHz transceiver. There is no direct wireless transmission from the transmitter which eliminates the need for a regular RC receiver onboard the UAVs. Manual control is taken by a toggle switch on the Pilot Console. The autopilot control takes over nearly instantly once the toggle is set to automatic.

3.5 Hardware-in-the-Loop Simulation

Hardware-in-the-loop (HIL) simulation is a method of flight testing performed on the ground. It uses all of the hardware that would normally fly in the air. Since the Piccolo Plus autopilots are not physically moving, software is needed to simulate GPS and IMU sensor measurements. A photograph of the setup at the UAV lab appears in figure **24**. The Ground Station is connected, via COM port, to the OI laptop and the Pilot

Console. The simulator laptop is connected to the autopilot via a COM port. The autopilot is powered by a DC power supply and is also connected to the Ground Station using the 900 MHz transceiver.

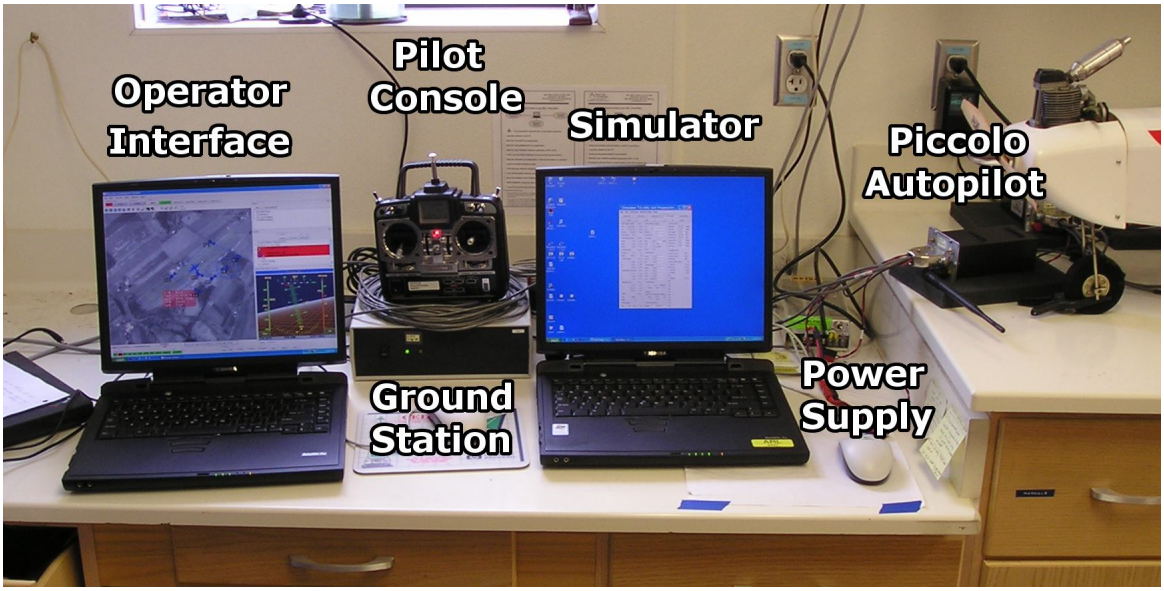


Figure **24**: Hardware-in-the-Loop Simulation

The HIL simulation was used extensively in software development of the computer vision and target localization code explained in the next chapter. Software was tested on the simulator laptop since it already had a serial data connection to stream the telemetry. During flight tests, the software was copied onto the UAV onboard computer. The most difficult thing to simulate is an accurate camera report of ground targets since the webcam is just sitting there, mounted inside the UAV. A piece of paper with red dots was used to simulate the red ball ground targets. The target localization estimate kept moving because of UAV motion so it was only possible to test how well the target localization worked with actual flight images and telemetry.

Chapter 4

Vision Processing and Target Localization

4.1 OpenCV Computer Vision Library

OpenCV is an open source object-oriented C++ library of image processing functions originally developed by Intel [**25**]. It can run on Linux, MacOS X, and Windows. One of the more notable applications of OpenCV was in the vision system of the UGV "Stanley", the winner of the DARPA Grand Challenge. There is a large user base because it is free and relatively easy to use. The OpenCV Yahoo Group and forums have over 31,000 members and gain about 200 more per week [**26**].

The library of code is divided into four major parts. The *CxCore* library comprises all of the data structures, memory management, basic operators, and other low-level functions used for mathematical manipulation of image data that not usually called by user-written code. The *CvReference* library has higher-level image processing functions generally called by user-written code. The *CvAux* library contains experimental and obsolete higher-level functions that aren't usually needed for most user-written code. The *HighGUI* library contains all of the input and output functions needed to create a simple graphical user interface. Loading and saving image and video streams is also controlled by the higher-level functions in this library.

4.2 Image Processing with UAVs

A lot of research work has been done on video stabilization [**27**] and ground photo mosaicking [**28**] for aerial video and photographs. Micro UAVs are particularly vulnerable to annoying shaky video. Stabilized video is much easier for humans to view and understand and it also leads into the challenging computer vision methods of tracking moving objects from a moving camera [**29**]. Tracking objects from video has been researched in detail [**30**] [**31**]. Identifying objects is a more challenging area in computer vision and is usually limited to a set of objects in a well-lit, uncluttered area. Computer vision used to aid autonomous UAV aerial refueling has also been investigated [**32**].

Modern UAVs typically always have a continuous data link to the ground. If there is an onboard video camera, it's usually always transmitting an analog NTSC TV signal to the ground station rather than processing the video onboard. The advantage of processing video imagery on the ground is that more processing power and more complicated vision algorithms can be used. The disadvantage is that there is also bandwidth overhead and communications lag between the ground station and the UAV. Loss of communication and video or data corruption may also be a problem. However, few UAVs do serious autonomous real-time video processing using onboard computer resources.

At least one commercial object tracking, video stabilization, and target localization product is available. Procerus Technologies, makers of the Kestrel autopilot, has developed proprietary software which they call OnPoint Targeting. For more details, refer to section **1.2** of this thesis or the OnPoint Targeting website [**16**].

4.2.1 Common Image Processing Algorithms

There are many image processing algorithms commonly found in many computer vision applications. Noise smoothing is used to deal with noisy image data. Edge detection is used to detect object boundaries. Harris Corner detection is used for image stabilization, projection transformations, and optic flow. The Hough transform is used to detect complex patterns and is often used with binary images from the output of edge detection to find lines and curves. A good reference book for these algorithms and many more is titled "Introductory Techniques for 3-D Computer Vision" by Emanuele Trucco and Alessandro Verri [**33**].

4.2.1.1 Noise Smoothing

Image noise causes problems in computer vision, particularly in edge detection algorithms. There is "salt-and-pepper" noise where a pixel from a photo sensor could be completely saturated or zero value and not representative of the true visual input. This causes spikes in the image data. There is also the much more common Gaussian white noise, which is caused by small variations in elements of the photo sensor. It can also be caused artifacts from image compression. To compensate for image noise, there are several smoothing kernels that may be applied to the image. The simplest is the *mean filter* (Equation **1**). The image, I, is convolved with this filter to yield the product, $I_{avg} = I*K_{avg}$.

$$K_{avg}(m=3)=\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \qquad \mathbf{1}$$

This takes the average of the current pixel and the surrounding pixels in an *m* x *m* box and sets the value in a new array. The edges of the image are treated differently.

Another common smoothing filter is the Gaussian kernel. The formula for smoothing an image with this filter is given in equation **2**. An example kernel appears in equation. **3**.

$$I_G = I * G = \sum_{h=-\frac{m}{2}}^{\frac{m}{2}} e^{-\frac{h^2}{2\sigma^2}} \sum_{k=-\frac{m}{2}}^{\frac{m}{2}} e^{-\frac{k^2}{2\sigma^2}} I(i-h, j-k) \qquad \mathbf{2}$$

$$G(\sigma=1, m=3)=\begin{bmatrix}0.0751 & 0.1238 & 0.0751\\ 0.1238 & 0.2042 & 0.1238\\ 0.0751 & 0.1238 & 0.0751\end{bmatrix} \qquad \mathbf{3}$$

An example of Gaussian smoothing on a noisy image appears below in figure **25** (iterated several times to exaggerate for visual effect).

Figure **25**: Gaussian Smoothing Example

4.2.1.2 Canny Edge Detector

The Canny edge detector is an algorithm that builds off of the Gaussian smoothing. The algorithm consists of three parts:

1. Noise smoothing
2. Edge enhancement
3. Edge localization

The noise smoothing was explained in the previous section. If the image used for edge detection is a three-channel, color image, the colors must be averaged to make a single-channel grayscale image. Then apply the noise smoothing kernel ($I_G = I*G$). The next step consists of finding the gradients in both the x and y dimensions ($J_x = I_G*S_x$, $J_y = I_G*S_y$). This is a simple kernel operator again. It can be a single-dimension discretised kernel (for example, S_x = [-1 0 1] and its transpose for the y-dimension) or some other kernel such as a two-dimensional Sobel masks (equation **4**).

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \; S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad \mathbf{4}$$

Now there are gradient components, J_x and J_y, for each pixel (i, j). Edge strength, E_S, and the edge orientation, E_O, of the gradients are required next and may be calculated with equations **5** and **6**, respectively.

$$E_S(i, j) = \sqrt{J_x^2(i, j) + J_y^2(i, j)} \qquad \mathbf{5}$$

$$E_O(i, j) = \arctan\left(\frac{J_y}{J_x}\right) \qquad \textbf{6}$$

Thus, E_S now has large values where there were harsh edges in the original image. The E_O array has values between 0º and 180º. If one only wanted to find areas where there were edges, the algorithm is complete at this point. However, the Canny edge detector goes on and removes unimportant edges and localized edges to a single pixel value. This is done through thresholding and nonmaximum suppression on E_S.

Nonmaximum suppression considers the best fit of a value at $E_O(i, j)$ with four angles at orientations of 0º, 45º, 90º, and 135º. Using the angle defined in $E_O(i, j)$, one chooses the closest angle and compares the two neighboring pixels in that direction. If $E_S(i, j)$ is smaller than one or both of the neighbors, $I_N(i, j) = 0$; otherwise $I_N(i, j) = E_S(i, j)$. The output, I_N, is a thinned and linked edge image. The edges have been localized to one pixel value. Now the algorithm continues to remove trivial edges using thresholding.

Thresholding I_N consists of first specifying two thresholds, T_{low} and T_{high}, then raster scanning the image, I_N, for the first pixel with a value where $I_N(i,j) > T_{high}$. Following the chains of the linked edges in each direction perpendicular to the edge normal and record each pixel where $I_N(i,j) > T_{low}$. Also, mark each pixel that has been visited. Once the ends of the chains are reached, where $I_N(i,j) < T_{low}$, continue on the next pixel in the raster scan, ignoring pixels already visited. The result of this operation is a "hysteresis threshold". It returns one pixel wide contours of "important" edges. Results will vary depending on the threshold values used. An example of the Canny edge detection process appears in figure **26** for $T_{low} = 1.5$, $T_{high} = 2.6$, $\sigma = 1$ (for the Gaussian smoothing).

Figure **26**: Canny Edge Detection Example

4.2.1.3 Harris Corner Detection

Harris corner detection is very useful in computer vision. Corners of features in a sequence of images are easy to cross-correlate between two frames unlike edges (because of the aperture problem). This is useful for motion detection (optic flow) and mosaicking. Once enough corners have been correlated between two sequential images, an affine transformation can be calculated and one image may be projected into the view of the other. Any features that moved between the two images may easily been seen by comparing one image with the other projected image.

The algorithm begins by computing the *x* and *y* derivatives of the image, J_x and J_y, as explained in the previous section, Canny edge detection. Then, the products of the derivatives are calculated for each pixel.

$$J_{xx}(i,j) = J_x \cdot J_x \quad J_{yy}(i,j) = J_y \cdot J_y \quad J_{xy}(i,j) = J_x \cdot J_y$$

The sums of the products at each pixel are then computed to obtain S_{xx}, S_{yy}, and S_{xy}. This is done by applying an NxN (where N is odd) Gaussian kernel to or even a kernel of ones to each of the products. Then define at each pixel (i,j), the matrix:

$$H(i, j) = \begin{bmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{bmatrix}$$

The response of the detector is calculated at each pixel with the following formula (k is a constant):

$$R(i, j) = Det(H) - k(Trace(H))^2$$

The remainder of the algorithm consists of thresholding R and computing nonmaximum suppression in two-dimensional neighborhoods around maximal pixels. After that, the algorithm has found the locations of many corner points in the image. Applying this to two sequential images in a series of aerial photographs would yield points in both images that could be cross correlated. Using a patch of pixels around each corner point, normalized cross correlation (NCC) can be used to search for the best match in a local area the second image. Once enough corner patches have been cross-correlated between the two images, a statistical method called RANSAC ("RANdom SAmple Consensus") may be used to eliminate the outliers. This method results in the best fit affine transform from one image frame to the next. The image can be projected into the other coordinate frame for mosaicking or to determine if any objects in the image have moved. An example of a moving aerial sequence with moving objects on the ground is presented in figure **27**. It's not easy to tell how the camera moved or if the cars are moving at all from these images, but through this algorithm, it will become apparent.

Figure **27**: Original Sequential Frames

The Harris corner detector is run on both images. Each corner has a patch of pixels around it used for NCC in figure **28**.

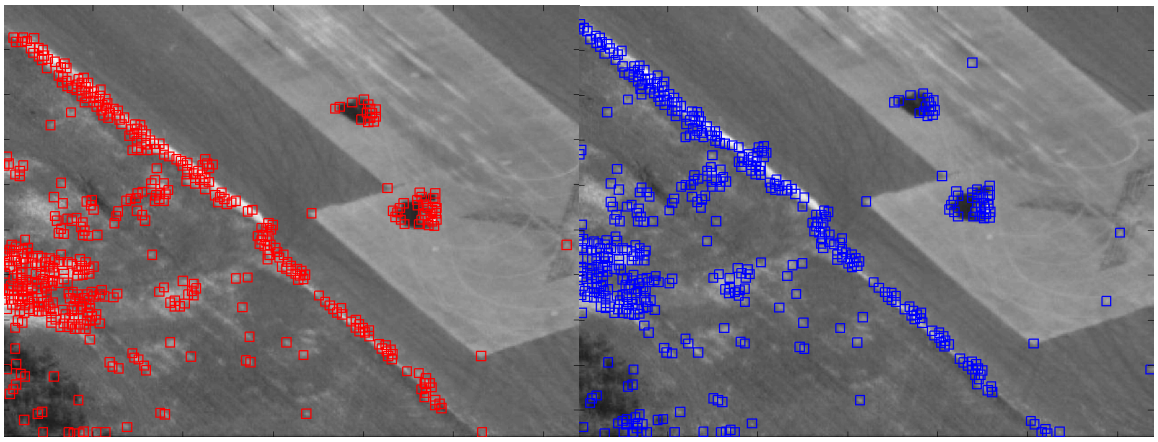


Figure **28**: Best 500 Harris Corners Between Sequential Frames

NCC is run between patches within a specified radius on each image. The best matches, above a specified threshold, are displayed on the left in figure **29**. RANSAC is run by selecting random samples of corner patch matches and calculating an affine transformation for them, then seeing how well the transformation applies to the rest of corner patch matches between the images. The image on the right in figure **29** shows

inliers in green and outliers in red. It is now obvious how the vehicles are moving relative to the rest of the image sequence and the difference may be calculated.

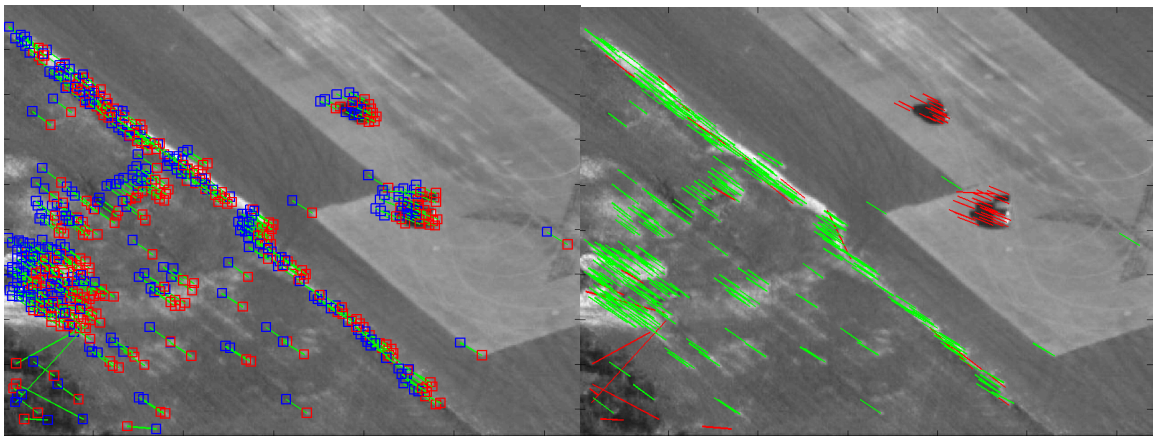


Figure **29**: NCC and RANSAC Results

4.2.1.4 Hough Transform

The Hough transform is useful for line and circle detection. Line detection is useful for finding man-made structures and vanishing points in images. Detailed algorithm outlines can be found in books by Trucco [**33**] or Leavers [**34**]. Hough lines and circles build off of the binary edge array resulting from algorithms like the Canny edge detector. In the case of Hough lines, the algorithm loops through each point in the image and projects a line in all directions and tests how well that line correlates to the binary edge image. The Hough circles algorithm virtually creates circles of varying radii at each pixel in the image and checks how well it matches up to binary edge image. This algorithm was tested using aerial photography for finding balls on the ground, but they're

so small that false positives were being detected in areas of the image with concentration of edges like corn fields.

4.3 Ball Finding and Target Localization Algorithms

The goal of the work found in this thesis was to create software for a UAV that would autonomously identify, in real-time, a target on the ground and calculate an estimate of the geographical position of the target. This is called target localization or sometimes it is referred to as geo-location.

A 75 centimeter red ball was chosen as a ground target because it appears as the same shape from any viewing angle and has a distinct color that doesn't occur often in aerial images of terrain. Early flight tests indicated that the ball was recognizable from about 100 meters altitude. However, the computer vision detection algorithm had a difficult time differentiating between the ball and some low quality views of the 55 gallon red drums that border the airfield. Most of the time the barrels weren't falsely identified as the ball because of their rectangular shape, but it occurred often enough to be a problem. There was a simple solution suggested to overcome this problem. None of the red barrels and other red objects near the airfield appeared very close together so the suggestion was to place two red balls near each other. That way, if a red blob the size of the ball was identified, it would be confirmed as part of the target by the existence of another red ball nearby.

No tracking was used in this code. It searches the entire image frame for targets even when it has one in view. This is not as efficient as tracking an object through a

sequence of images. However, the image processing code is very fast (approximately 20 processed frames per second on the onboard computer) and recording the image data was the slowest part of the code which limited it to about 5 frames per second. Object tracking once the target was located wouldn't have yielded much of a performance boost.

4.3.1 Ball Finding Algorithm

It would be best to read this section with the accompanying image processing subroutine code found in the Appendix section **B.1** as a reference. This subroutine is called by the main function in the program and returns an array of red blobs of appropriate size and shape. More processing is done in the main function to determine the physical size of the blobs on the ground and their geographical location. The red blob finding algorithm is comprised of these parts:

1. Convert webcam RGB color space image to HSV space
2. Filter HSV based on Hue and Saturation to create binary red filtered image
3. Dilate binary filtered image
4. Assign unique IDs to connected blobs in dilated binary image
5. Overlay unique blob IDs on original binary filtered image
6. Remove unimportant blobs based on size and shape
7. Calculate and return blob size information.

The computer vision section of the code begins by converting the webcam-captured image from the Red-Green-Blue (RGB) space to Hue-Saturation-Value (HSV) space. This is easily performed using a built-in function in OpenCV. After conversion,

the algorithm iterates through each pixel, via raster, scan, and tags the pixel location in another 2D array if the hue and saturation values fall within a specified range. This hue range for the red is from 0º to 36º and 320º to 360º. Red loops around in the virtual color wheel so the range of red hue used in the filter is actually from 320º to 36º. OpenCV is a little quirky with this since it can't store values above 255 as an unsigned integer; it divides the hue color space by two. So the code searches for pixels with a hue between 0 to 18 and 160 to 180. Also required to pass through the color filter is a saturation value greater than 30% or 77 as an unsigned integer. These values appear in table **5**. The integer values may be changed in the initialization file while at the field. An example of the filter appears in figure **30**.

Table **5**: Red Color Filter Values

	HSV Color Space Value	OpenCV Unsigned Integer Value
Min Hue	320º	160
Max Hue	36º	18
Min Saturation	30%	77

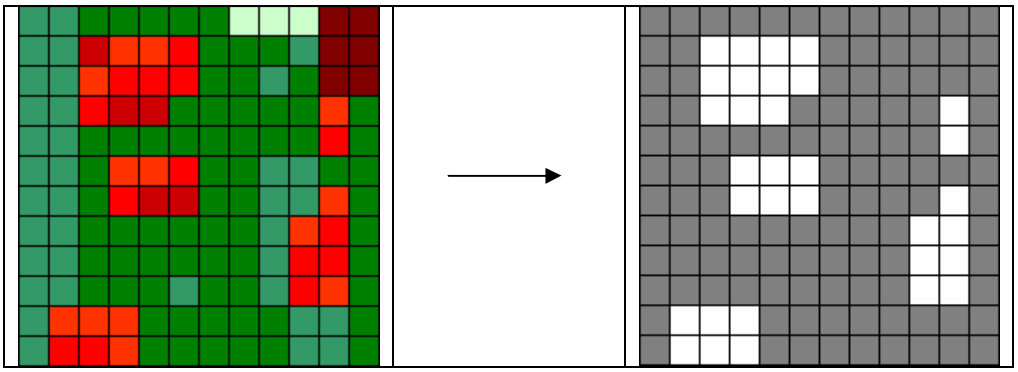


Figure **30**: RGB Color Pixels to HSV Space to Binary Red Filtered Image

Now the algorithm has a binary image of known red pixels. Next, the tagged pixels in the binary image are dilated by a specified number of pixels (one pixel in practice). The dilation serves to later group together clusters of pixels that may have only been separated due to noisy image data. It's not so important to group the red ball pixels together since it's normally a solid blob. Rather, patches of red colored dirt and other noisy red areas need to be ground together. Dilation is performed during the same raster scan as color filtering. A box (with height and width of twice the dilation value plus one), centered around the flagged pixel is flagged in another 2D pixel array. Now there are two pixel arrays; one with the pixels that pass through the color filter and another that has dilated those pixels. This is graphically depicted in figure **31**.

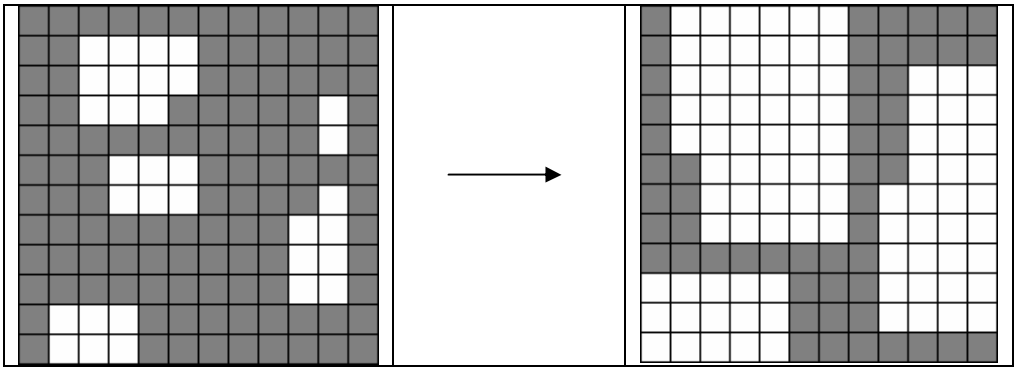


Figure **31**: Binary Red Filtered Image to Dilated Binary Image

The next section of the algorithm finds connected blobs of flagged pixels. It uses the previously generated 2D dilated pixel array for this. A new third pixel array is generated. This contains blobs separated by a unique blob ID value instead of just being a binary array. This code iterates through the dilated binary image by raster scan. If it comes to a flagged value in the dilated binary image, it checks the unique blob ID value of the pixels above and to the left of the current pixel location in the new 2D array of

blob IDs. If a non-zero value exists, the current pixel ID is set to the lower value of the two and an additional subroutine is called to update the blob ID. If the two pixels above and to the left have no ID value, a new unique ID value is incremented and set. If only the above value is non-zero, the current pixel takes on that blob ID. Now the only other remaining possibility, that the above pixel value is zero and the left pixel value is non-zero, is used to set the current pixel blob ID value to the value of the pixel to the left. The result of this process is graphically depicted in figure **32**.

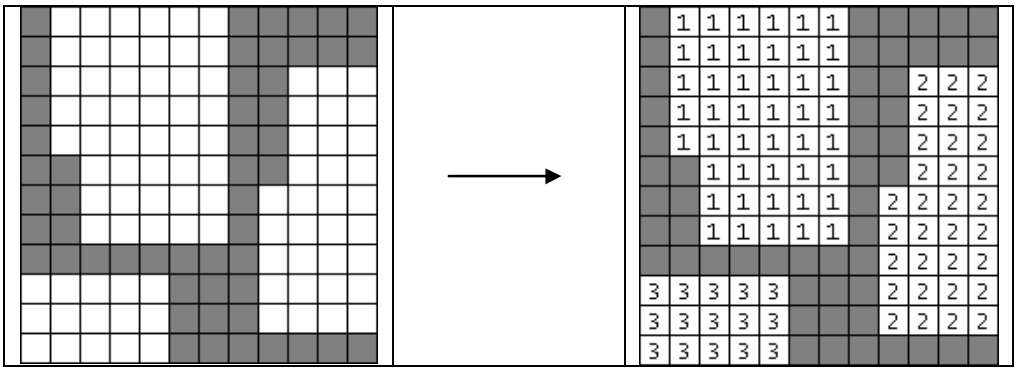


Figure **32**: Dilated Binary Image to Dilated Image with Blob IDs

The subroutine to update blob IDs, mentioned above, simply takes two ID values, one search value and one replace value. A search location is also specified so that it does not go beyond that pixel.

The third pixel array with the unique blob IDs has now been generated. Each pixel in the dilated array now has a blob ID associated with them. The code now virtually overlays the unique blob ID values onto the original filtered pixels and stores the value in a fourth 2D array of pixels. This is accomplished by once again iterating through the filtered binary image, checking to see if it has been flagged as a red pixel, checking the unique blob ID of the corresponding pixel, and setting that value in the new

fourth array. Now there is an array of red filtered pixels that are grouped into blobs that may or may not have an immediate neighboring pixel that is also red, but is still considered part of the blob. The result of this process is graphically depictured in figure **33**.

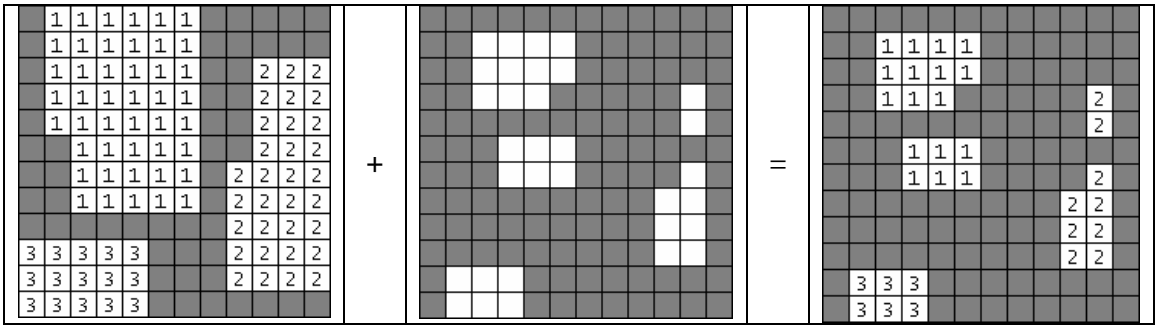


Figure **33**: Grouped Blobs of Red Pixels with Unique Blob IDs

Also during the last raster scan, the upper-left pixel location, the lower right pixel location, the total blob pixel count, and the height-to-width ratio of the bounding box of each blob is calculated. For example, the values of the last image in figure **33** appear below in table **6**.

Table **6**: Red Blob Values

Blob ID	CvPoint Min (x, y)	CvPoint Max (x, y)	Pixel Count	Height-to-Width Ratio	Physical Size (m)	Geographic Position
1	(2, 1)	(6, 7)	17	1.5	Not	Not
2	(9, 3)	(11, 10)	9	3.5	Calculated	Calculated
3	(1, 10)	(3, 12)	6	0.67	Yet	Yet

The next step in the code filters out all blobs that do not have an appropriate height-to-width ratio or an appropriate number of pixels in the bounding box. Memory is then deallocated and the subroutine ends. This is basically how the computer vision section of the code works.

4.3.2 Virtual World Model

One novel feature of this work is the inclusion of a virtual world model which enables the real-time target localization from the camera report. A priori knowledge of the terrain in the flight area enables the computer to estimate the length of a vector from the UAV to the target by calculating the intersection point of the vector and the georeferenced terrain. The computer can then estimate the geographic position, physical size, and shape of a target object. Without this world model, target localization becomes a triangulation problem similar to the work done by DeLullo [**35**]. Triangulation requires two or more frames taken with a significant separation distance, whereas the method presented in this thesis only requires a single frame.

Information about the GPS position of the UAV, its attitude, the camera orientation relative to the airframe, and camera properties are required for both triangulation and terrain intersection methods to work. The UAV position and attitude are estimated by the Piccolo Plus autopilot. The orientation of the camera relative to the UAV airframe remains in a fixed, downward-looking state for the work presented in this thesis. The horizontal and vertical fields of view (HFOV and VFOV, respectively) are also known.

4.3.2.1 Obtaining and Processing Terrain Data

The terrain data is obtained from the United States Geological Survey (USGS) National Map Seamless Server [**36**]. Using the Seamless Data Distribution Viewer, one can select an area of interest using a rectangle selection tool or by specifying the two

latitude and longitude coordinates of the area of interest. Many datasets are available for download in multiple data format types. The useful terrain data is found in the 1 arc second, 1/3 arc second, or 1/9 arc second National Elevation Dataset (NED). The default dataset for download is the 1 arc second NED. The 1/3 arc second and 1/9 arc second data is available by selecting "Modify Data Request" on the Request Summary Page. Selecting these datasets will yield higher resolution terrain data.

In order for the NED data to be useful, it must be in a non-proprietary format. The default data format is ArcGRID, which is proprietary and used primarily by the USGS itself. One can change the data type on the same page as the dataset selection. The GeoTIFF data type is available for download and accessible by many third-party software applications. It's still a bit difficult to work with, however. Fortunately, there exist third-party applications to convert the GeoTIFF data format to a very accessible and easy to use ArcInfo ASCII Grid format (also known as Arc ASCII Grid). Global Mapper is one of those software applications and is available for free (with limitations) download on their website [**37**]. Once installed and running, the GeoTIFF data file of the area of interest can be opened and exported in a number of data formats including the ArcInfo ASCII Grid format.

The ArcInfo ASCII Grid data format is very easy to understand and human-readable. The first five lines contain information about the data succeeding them. The variables are *ncols*, *nrows*, *xllcorner*, *yllcorner*, and *cellsize*. The variables *ncols* and *nrows* refer to the number of columns and rows of grid data, respectively. The variables *xllcorner* and *yllcorner* refer the western latitude line and southern longitude line, respectively. The point at *xllcorner* and *yllcorner* is the lower left corner of the grid of

data. Both values are degrees. The variable labeled *cellsize* refers to the spacing between grid points in degrees. The remainder of the data in the ArcInfo ASCII Grid data file is the elevation data. The first value is the upper-left (northwest) corner and the last value is the lower-right (southeast) corner.

The values found in table **7** are taken from the ArcInfo ASCII Grid data file used to generate the virtual terrain model. The value for *cellsize* corresponds to 1/3 arc second spacing.

Table **7**: ArcInfo ASCII Grid Variables

ncols	379
nrows	325
xllcorner	-77.675093
yllcorner	40.7949074
cellsize	9.2593E-05

This generates approximately 10.3 meter grid spacing north and south along the longitude lines and approximately 7.8 meter grid spacing east and west along the latitude lines. The total area in the virtual model is approximately 9.85 square kilometers, centered on the airfield. An "Atlas Shader" colored elevation map of this terrain data appears in figure **34**. Although the UAVs do not come close to flying outside of this area, much of it can be seen by the onboard camera at altitude, particularly while the UAV is in a banking maneuver. The area displayed in figure **34** has a latitude range of 77.675093º W to 77.640000º W and a longitude range of 40.7949074º N to 40.8250001º N.

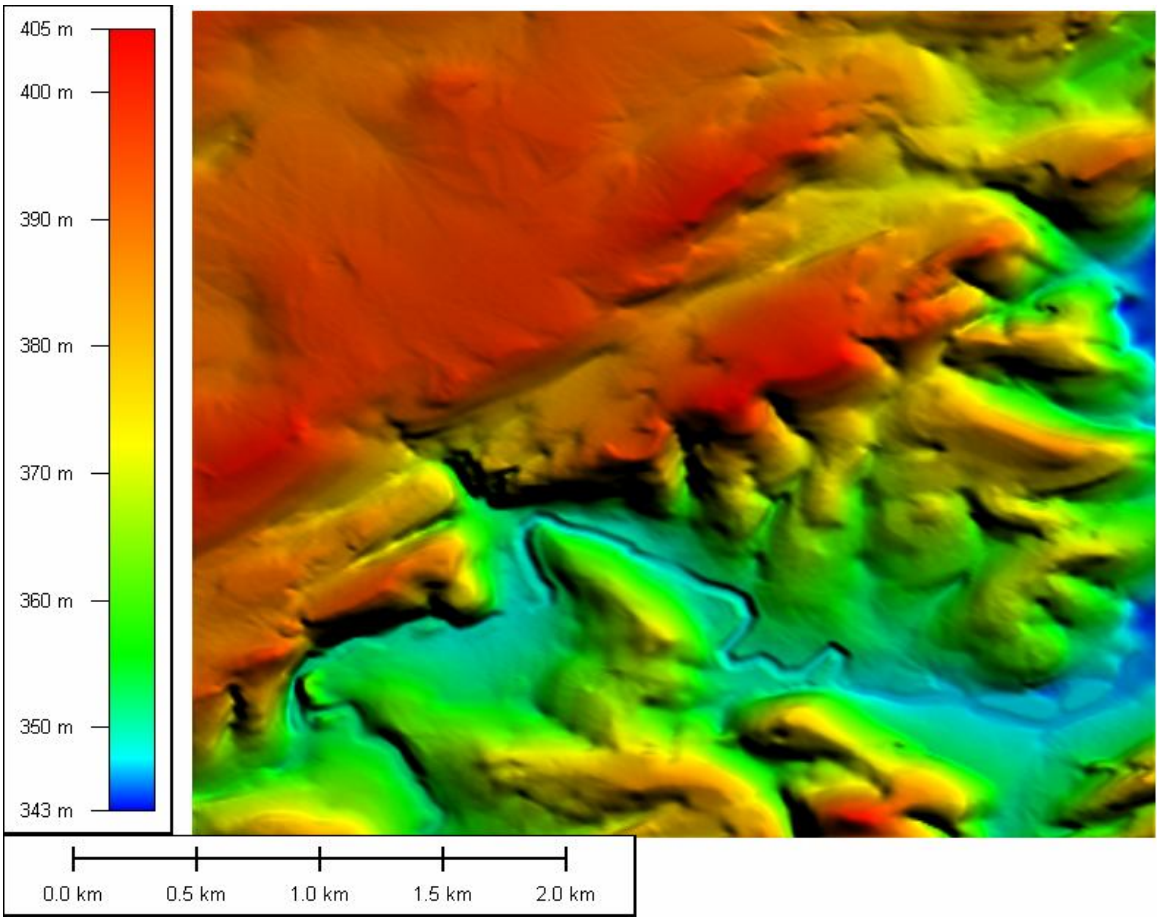


Figure **34**: "Atlas Shader" Colored Elevation Map of Flight Area

4.3.2.2 Target Vector Intersection with Terrain

The previous section explained how to obtain the terrain data needed for the virtual world model. This section will explain how it is parsed and used to geolocate targets from a camera report. It may be helpful to refer to the world model code in the Appendix.

Initialization of the virtual world model begins by calling the World constructor that accepts a file location character string and a geoid height measurement as arguments.

The file location character string refers to the ArcInfo ASCII Grid data file. The WGS-84 geoid does not vary significantly over the flight area to justify compensating for the variation. It's no longer required and would only be used to adjust the terrain in the future if the target position is needed in GPS coordinates. The geoid value used for the flight area is -34.15 meters. After reading in and storing in memory the variables listed in table **7**, the code dynamically allocates a 2D array of grid elements based on the values of *ncols* and *nrows*. A 1D array could have been used, but it is easier to think of it in two dimensions and for display purposes. Each of these grid elements is a structured object (called GridElement) with five double precision values: *xlocal*, *ylocal*, *elevation*, *latitude*, and *longitude*.

A local "flat earth" Cartesian coordinate system is created for the flight area. This can be done for relatively small flight areas without significant error. For small enough flying areas with UAVs at low altitudes and without a large camera view, the error this creates is negligible. The origin was specified as lower right (southeast) corner of the grid data. That corner was selected because the flight area is in the northwestern hemisphere, but it doesn't really matter. The grid spacing in latitude and longitude is orthogonal and based on a separation specified by *cellsize*. However, the grid spacing in the local 3D Cartesian coordinate system is non-orthogonal. As the grid moves away from the equator, the spacing becomes tighter in the east-west direction, whereas the spacing in the north-south direction remains constant in both degrees and the local coordinate system. To calculate the Cartesian coordinate, a function is called that uses equation **7** to solve for the number of meters per degree longitude at a given latitude.

$$\text{meters per degree latitude}(\phi) = \left(111320 + 373 \cdot \sin(\phi)^2\right) \cdot \cos(\phi) \qquad \mathbf{7}$$

The result of Eq. **7** is then multiplied by the number of degrees in longitude the grid point is from the origin to obtain the position in the x-direction (east-west) in the local coordinate system, *xlocal*, for the grid point. The position for each grid point in the y-direction, *ylocal* (north-south), is easier to calculate since the number of meters per degree changed in latitude remains constant at 111,320 meters per degree. The elevation value read from the ArcInfo ASCII Grid data file is then added to the geoid value and stored in *elevation* (the z-dimension) for each grid point. So now each point in the 2D array is specified as a point in a local 3D Cartesian coordinate with their accompanying latitude and longitude coordinates.

Using telemetry data from the autopilot, the UAV state is known. The coordinate system of the aircraft and camera is a bit different than the standard English aircraft conventions. The y-axis is pointing out the nose, the x-axis is pointing out the right wing, and the z-axis is pointing up. This was done to make it easier to work in the camera coordinate system and the local terrain coordinate system. It's simpler than having a third coordinate system going from aircraft coordinates, to camera coordinates, to local terrain coordinates. Since the camera coordinates and aircraft coordinates are fixed relative to each other, it was easier to do it this way.

When doing coordinate rotations from the aircraft/camera coordinate system to the local terrain coordinate system, the rotation is done in the following order: the coordinate system is rotated about the z-axis (psi), then rotated about the y-axis (theta), and lastly rotated about the x-axis (phi). Because of the differences between the aircraft

coordinate system and the camera coordinates mentioned above, the rotations about the y-axis and x-axis are negative. The *x* and *y* position of the aircraft in the local coordinate system is calculated from the *latitude* and *longitude* with the following equations:

$$\begin{aligned} Xlocal(\text{lat}, \text{lon}) &= \text{m\_per\_degree}(\text{lat}) \cdot (\text{lon} - \text{max\_lon}) \\ Ylocal(\text{lat}) &= 111320 \cdot (\text{lat} - \text{min\_lat}) \end{aligned} \quad \textbf{8}$$

The function *m_per_deg(lat)* is from equation **7**. The variables *max_lon* and *min_lat* are the longitude and latitude values at the origin of the local coordinate system. The altitude measurements for the z-axis are the same for both coordinate systems. Similar equations are used to change the local coordinate system back to latitude and longitude.

When a target is identified in the camera view, a unit vector from the UAV position to the target is calculated in the local coordinate system. Now there is a discretized array of terrain points and a vector from the UAV to the target. Five points are required for getting a point of intersection between the terrain and target. Three are points on a the terrain ($\vec{p}_0$, $\vec{p}_1$, and $\vec{p}_2$) and two points are from the target vector ($\vec{l}_a$ and $\vec{l}_b$). These are all three dimensional points in the local coordinate system.

$$\vec{p}_0 = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}, \vec{p}_1 = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}, \vec{p}_2 = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}, \vec{l}_a = \begin{bmatrix} x_a \\ y_a \\ z_a \end{bmatrix}, \vec{l}_b = \begin{bmatrix} x_b \\ y_b \\ z_b \end{bmatrix}$$

These points are also represented graphically in figure **35**.

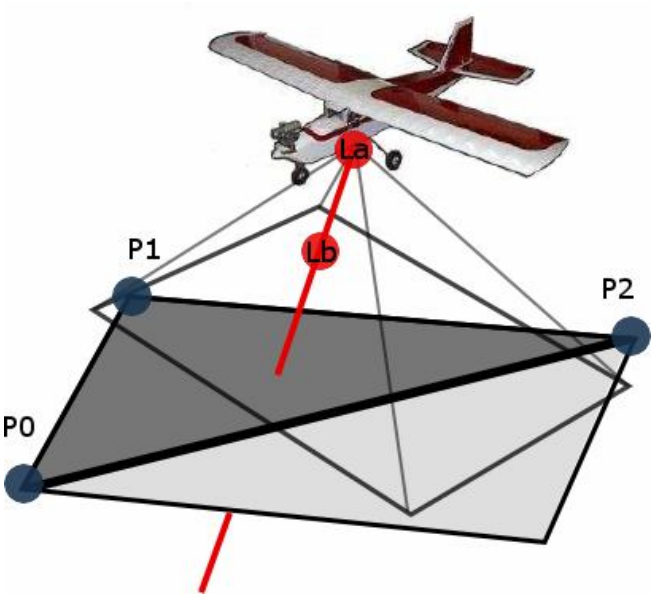


Figure **35**: Five Points Used in Line-Plane Intersection Calculation

The point of intersection is found with the formula $\vec{l_a} + (\vec{l_b} - \vec{l_a}) \cdot t$ and t is found by solving the following:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \begin{bmatrix} x_a - x_b & x_1 - x_0 & x_2 - x_0 \\ y_a - y_b & y_1 - y_0 & y_2 - y_0 \\ z_a - z_b & z_1 - z_0 & z_2 - z_0 \end{bmatrix}^{-1} \begin{bmatrix} x_a - x_0 \\ y_a - y_0 \\ z_a - z_0 \end{bmatrix}$$

The algorithm then iterates on three-point planes in the local grid, using the last calculated point of intersection to update the three grid points in the next plane, to find the exact ground location of the target. The plane directly below the UAV is used for the first guess and convergence usually occurs within 1 or 2 iterations. The local coordinates are then converted into GPS coordinates and recorded.

4.3.3 Bringing It All Together: The Main Loop

Figure **36** is the main software flow chart. Initialization is pretty straight forward and the world model initialization was covered in the last section (**4.3.2**). Variable space is allocated in memory on startup. Values are read in from the initialization file. A camera connection is attempted. If there is no camera connection, the program displays an error and stops. A Piccolo connection is attempted. Likewise, if there is no Piccolo connection, the program displays an error and stops. The world model is initialized and then output data files streams are opened.

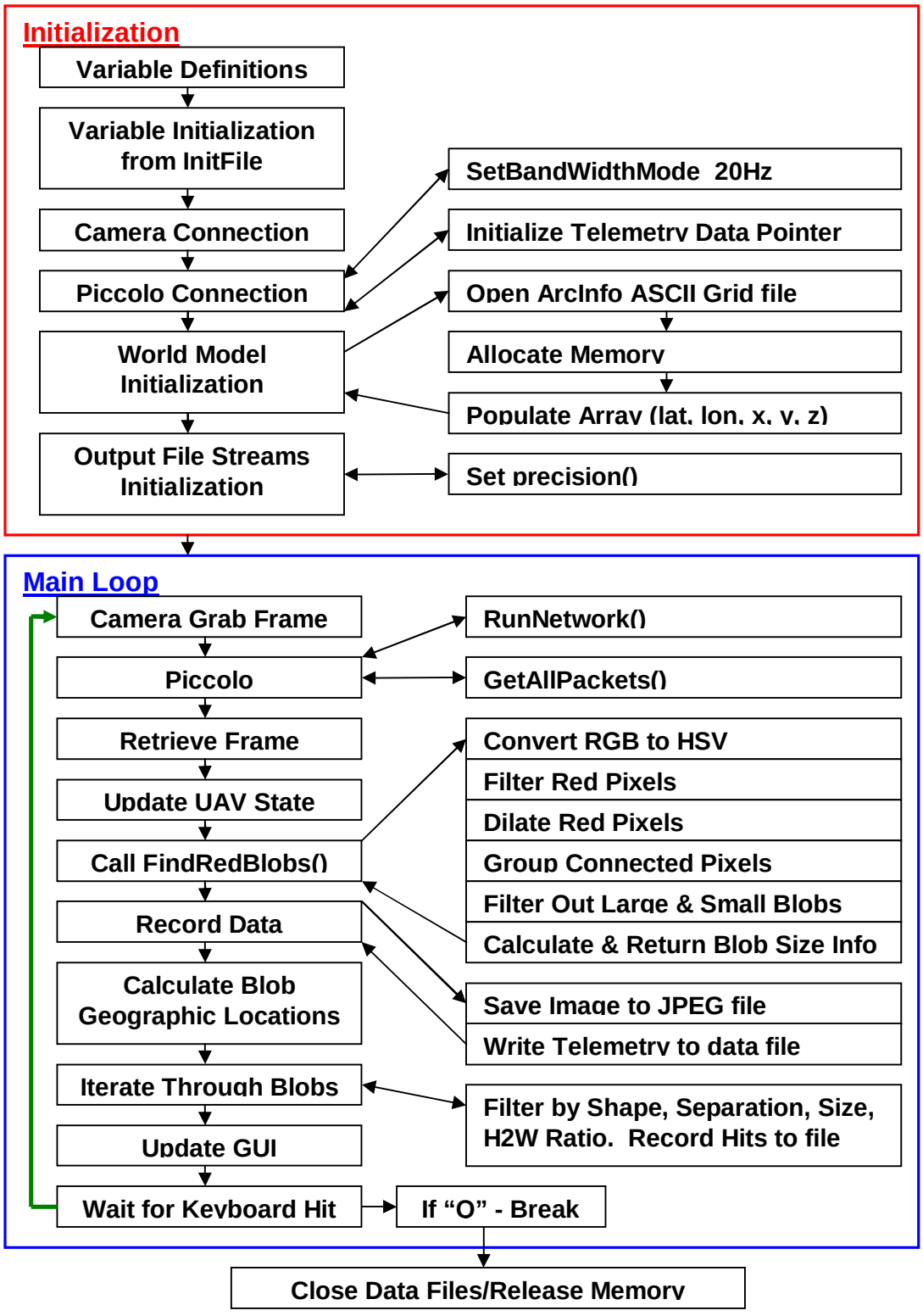


Figure **36**: Main Code Flow Chart

The main loop starts by trying to “grab” a frame from the webcam. This frame is stored internally and not yet loaded into the IPL image structure for use in the code. The purpose of this is to get the image data fast and then immediately move to the next block of code, which is the Piccolo telemetry. The RunNetwork and GetAllPackets functions are called to retrieve the current state of the UAV. Now the webcam image is retrieved from internal storage after the current UAV telemetry is saved. The UAV state object is updated from the telemetry. If the UAV is not over the world model, an error message is displayed and the program ends. The webcam image is saved to the hard drive as a JPEG and some important telemetry values are saved in the telemetry output text file. The FindRedBlobs function is called and executed as detailed in section **4.3.1**.

At this point in the main loop, the webcam image has been saved and processed to find red blobs. The array of blobs returned from the FindRedBlobs function is then analyzed. The location and size of each blob is estimated. The results are then looped through to see if any appropriately sized blobs are of the correct separation distance. If it finds the two blobs that pass through all these filters, the program records a hit. The target location is then the average, or center point, of the two blobs.

There are user interface options available for display and debugging purposes, however using them slows down the program a lot. From within the console window, if you type “W” while the program is executed, this will enable or disable the current webcam image with debugging features overlaid on the image. It will put a border and other debugging data around the red blobs. Figure **37** displays this window

Figure **37**: GUI: Webcam Image with Debugging

Another GUI window option is to display the “back project”, which just the binary image of red pixels. This is used to determine of hue and saturation settings are working correctly. This window can be enabled or disabled by pressing “B” in the command console while the program is running. Figure **38** displays this back project

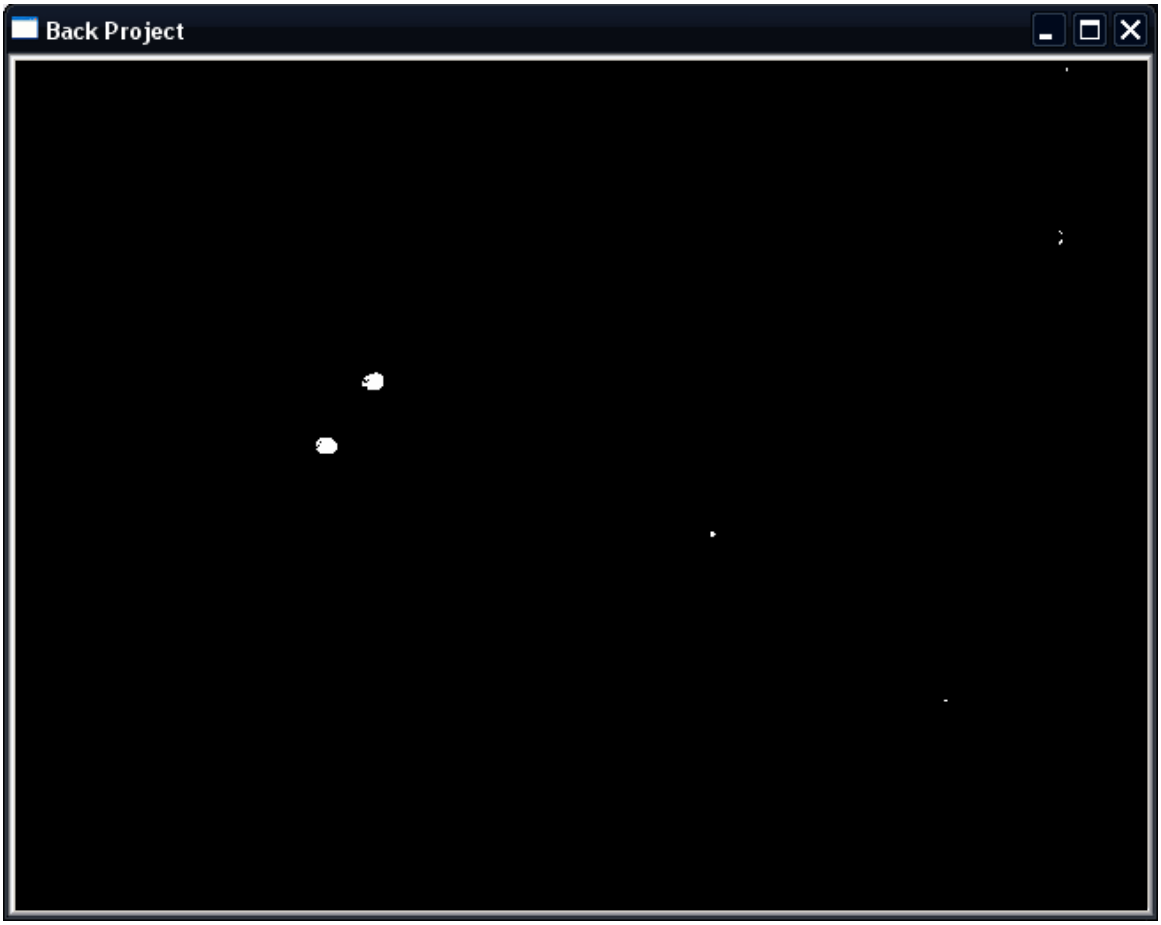


Figure **38**: GUI: Image Processing Back Project

Another useful GUI option is to display the current webcam view projection on the terrain map. The "atlas" colored and shaded terrain map is built directly from the ArcInfo ASCII Grid data as an OpenCV image in the world model code. This can be enabled and disabled by pressing "M" in the command console. Figure **39** shows this window.

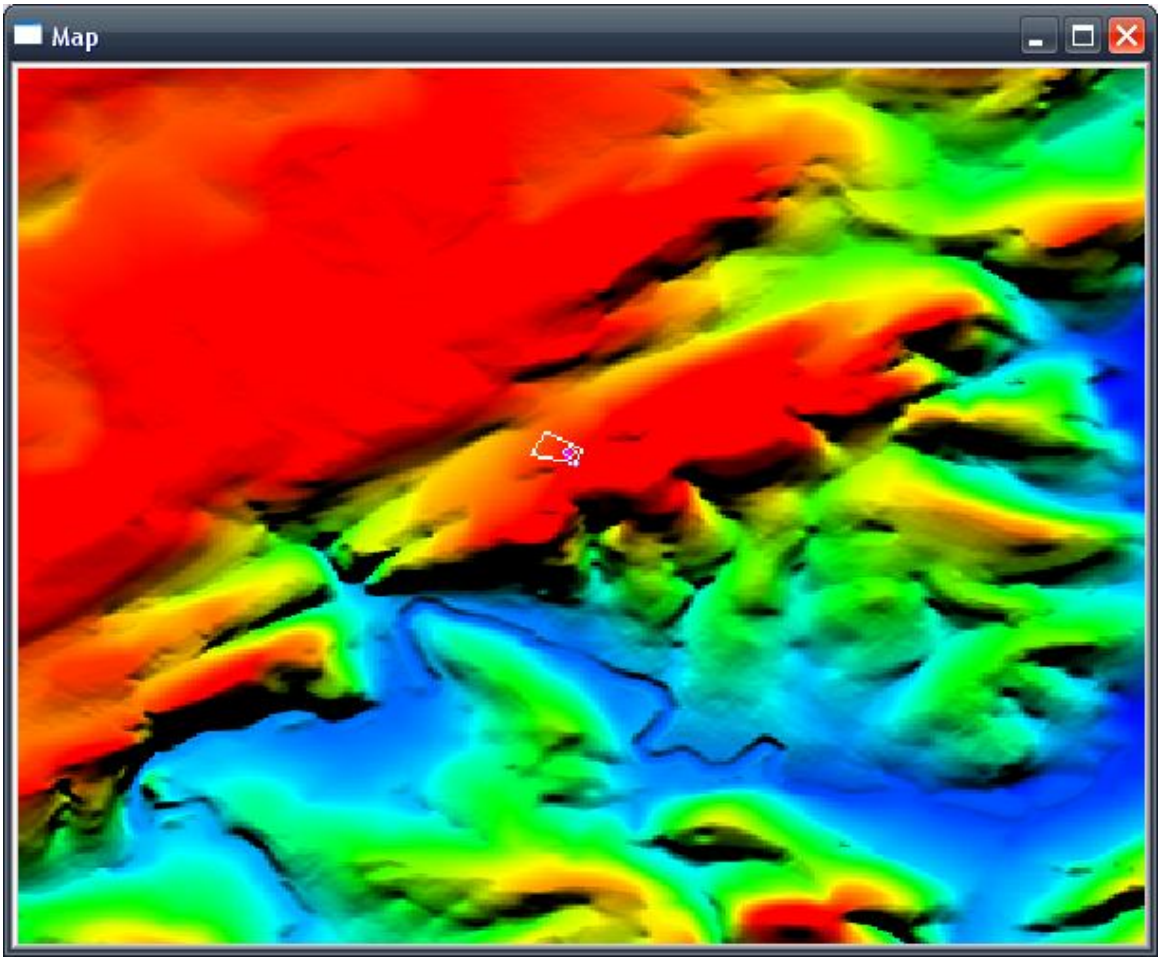


Figure **39**: GUI: Webcam View Projection onto Terrain

Chapter 5

Results

5.1 Flight Test (April 8, 2008)

Flight testing resulted in mediocre target localization capability at first. A geoid-adjusted GPS altitude was used to determine the 3-D position of the aircraft. This turned out to be a mistake because the GPS altitude measurement just isn't very accurate. To fix this, barometric altitude was used in the post-processed calculations to determine the 3-D location of the UAV. This improved the results significantly. A photograph of the setup on the ground appears in figure **40**.

Figure **40**: Photograph of Red Ball Targets/Attempting to Acquire GPS Lock

Several passes were made over the target area. The geographic positions (latitude and longitude) were translated into a local coordinate system measured in meters in figure **41**. Red outlines of the two target balls (75 cm diameter, 5.5 meter separation)

also appear in the figure to depict the size and separation distance. The first pass over the target area was too high for the camera to recognize the red ball targets on the ground. It was lowered to 67 to 75 meters for the remaining passes, which seemed to work out well.

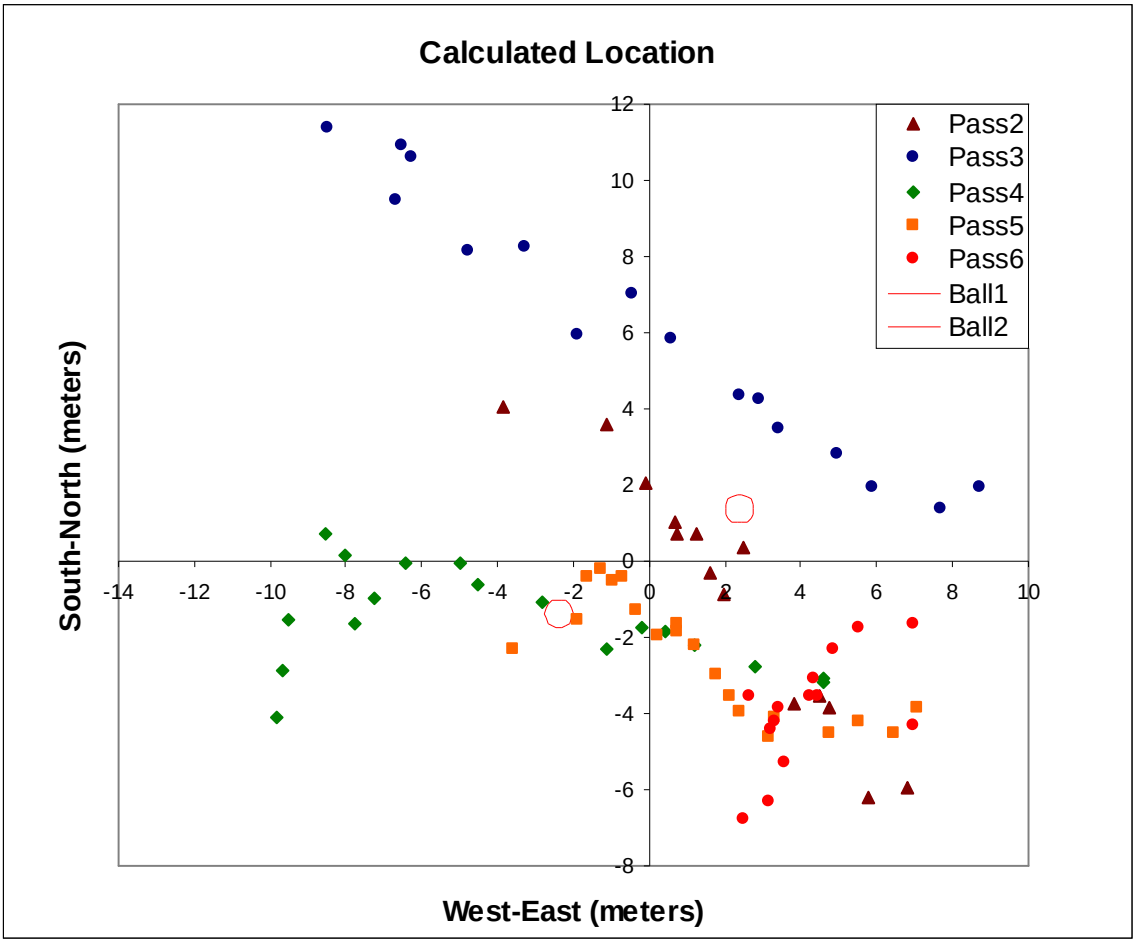


Figure **41**: Calculated Target Locations in Local Coordinate System

5.1.1 Pass #6

Telemetry data from the sixth pass during the flight test is found in table **8**. The UAV passed about 67 meters above the target. The images from this pass appear in the

appendix section **A.1**. In figure **42**, the approximate camera views are projected in the local coordinate system in relation to the center point of the calculated positions for this pass. Note that the corners on the projected view are correct, but the perimeter is not projected in that view correctly. By inspection, it is apparent from both the telemetry data and the images that the aircraft was coming out of a banking maneuver before passing over the target area in a southwesterly direction.

Table **8**: Pass #6 Partial Telemetry

Frame #	Piccolo Time	Latitude (rad)	Longitude (rad)	Altitude (m)	Roll (rad)	Pitch (rad)	Yaw (rad)
139	2328203	0.712302237	-1.355384460	458.33	-0.3388	0.0865	3.7578
140	2328453	0.712301888	-1.355385114	458.33	-0.3155	0.0867	3.7197
141	2328703	0.712301530	-1.355385735	458.3	-0.3054	0.0825	3.6693
142	2328953	0.712301132	-1.355386642	458.1	-0.2626	0.0711	3.6256
143	2329203	0.712300749	-1.355387228	458.05	-0.2402	0.0733	3.5937
144	2329503	0.712300274	-1.355387887	458.17	-0.2057	0.0775	3.5572
145	2329703	0.712299954	-1.355388309	458.2	-0.163	0.0783	3.5414
146	2330003	0.712299387	-1.355389284	458.36	-0.102	0.0829	3.5357
147	2330203	0.712299052	-1.355389710	458.37	-0.0858	0.0898	3.5314
148	2330503	0.712298558	-1.355390331	458.39	-0.0303	0.0888	3.5264
149	2330703	0.712298228	-1.355390743	458.45	-0.0345	0.0832	3.5273
150	2331003	0.712297598	-1.355391611	458.4	0.0726	0.0607	3.5748
151	2331203	0.712297263	-1.355392037	458.4	0.1349	0.0543	3.5914
152	2331453	0.712296846	-1.355392585	458.28	0.1071	0.0412	3.6081

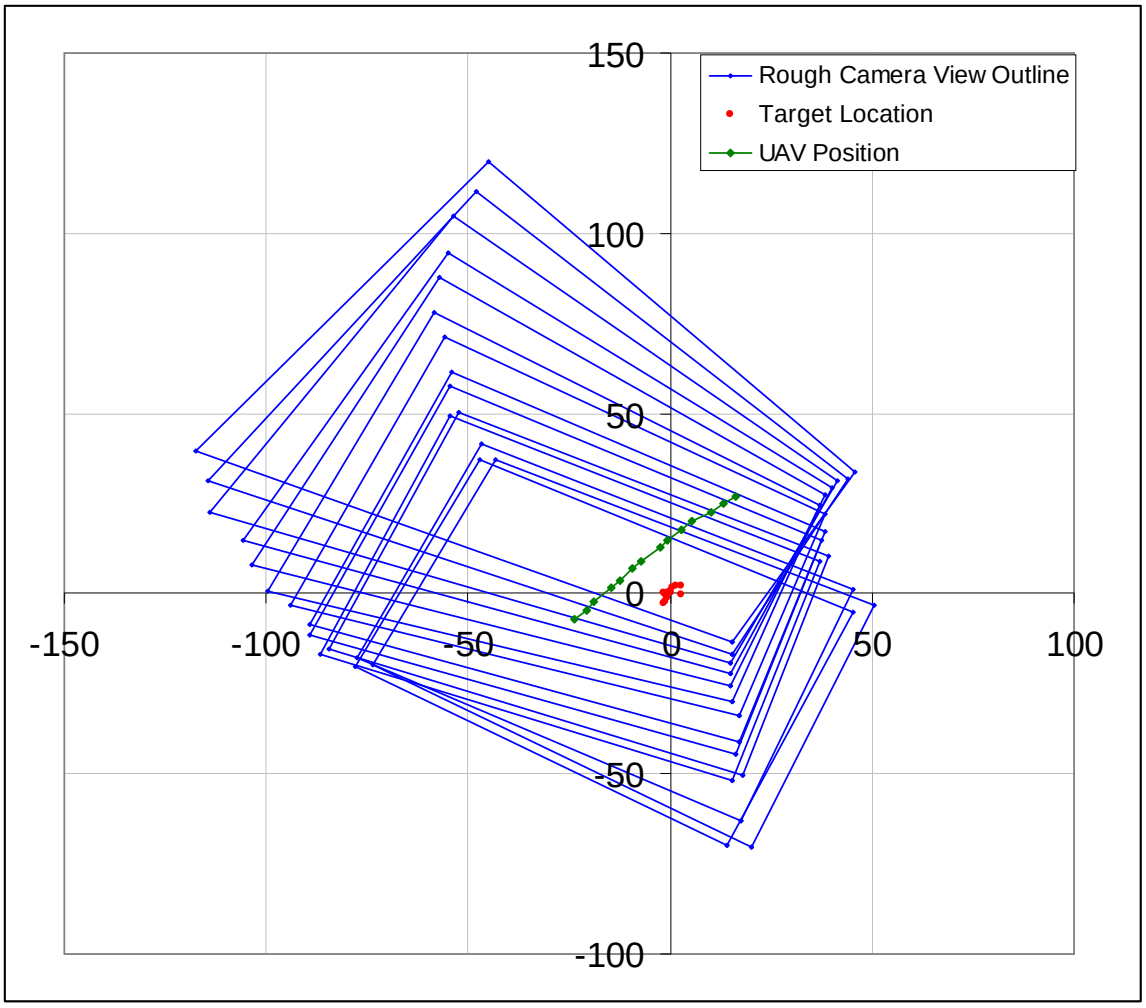


Figure **42**: Pass #6 Camera Views, UAV Position, and Calculated Target Location

5.2 Error Analysis

There are several sources of error. The Euler angles for the aircraft are not directly measured, but rather estimated. Small errors in the estimated attitude of the aircraft change the projection of the camera view onto the ground. Figure **43** depicts the target localization offset due to attitude errors (yaw and pitch in the figure).

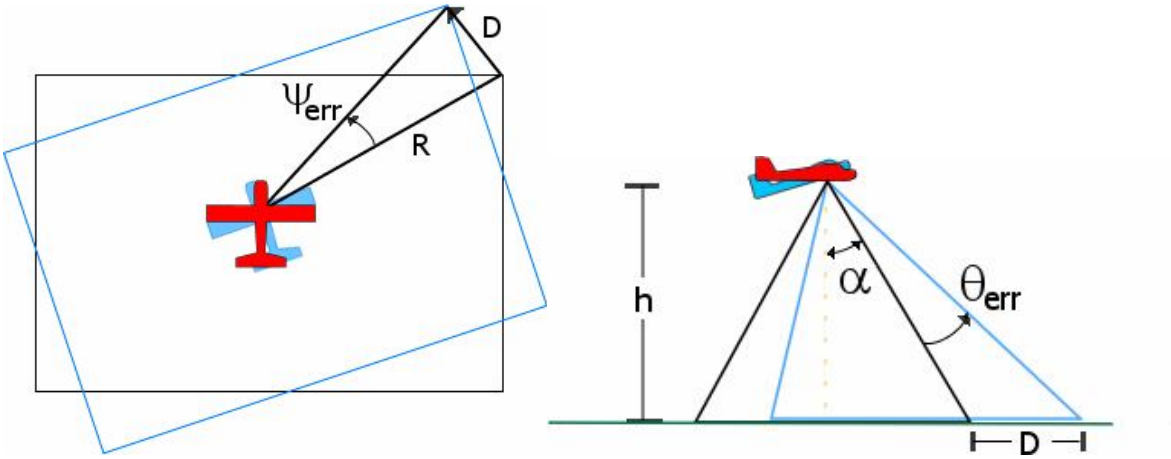


Figure **43**: Diagrams of Target Localization Offset due to Attitude Errors

With the assumption of straight and level flight, the offset error (*D*) due to an error in the attitude measurements (ψ_{err}, θ_{err}, and ϕ_{err}) can be calculated by the geometry. For an error in the yaw measurement (ψ_{err}), the error in calculated position will increase as the object moves away from the center of the image. Using a small-angle approximation, this offset error is $D_\varphi \approx R \cdot \varphi_{err}$, where *R* is the distance on the ground from the point in the center of the image.

The offset error for an error in pitch, θ_{err}, is a function of the height above the ground, *h*, and the angle to the target, α, using the equation: $D_\theta = h \cdot (\tan(\alpha + \theta_{err}) - \tan(\alpha))$. Similarly, the offset error for an error in pitch may be calculated with the equation: $D_\phi = h \cdot (\tan(\beta + \phi_{err}) - \tan(\beta))$.

The image and telemetry data are not perfectly synchronized. There could be a 50 millisecond difference between the two. The position and attitude may change a slight amount during that period. Alignment errors in the camera mount relative to the autopilot would also introduce a fixed angular error into the works. Table **9** displays the

approximate position errors expected due to angular misalignments for the reasons mentioned above.

Table **9**: Position Errors Due to Angular Misalignments

	Position Error (meters/degree misalignment)	
1º Yaw Error	0.87	R = 50 m
1º Pitch Error	1.49	h = 67 m, α = 27º
1º Roll Error	1.88	h = 67 m, β = 37.5º

The estimate of the 3-D position of the aircraft has an error due to noise in the GPS position. Any error in the latitude or longitude estimate of the aircraft would be nearly directly translated to an error in the target position estimate on the ground. This assumes the ground is flat. The altitude measurement is fairly accurate because it is based on barometric pressure, not the GPS altitude. The barometric altimeter was zeroed at the start of the flight shortly before making the passes over the target. However, given an error in height, h_{err}, and an angle to the target from the vertical, χ, (Figure **44**) the offset error can be calculated with the equation: $D_h = h_{err} \cdot \tan(\chi)$.

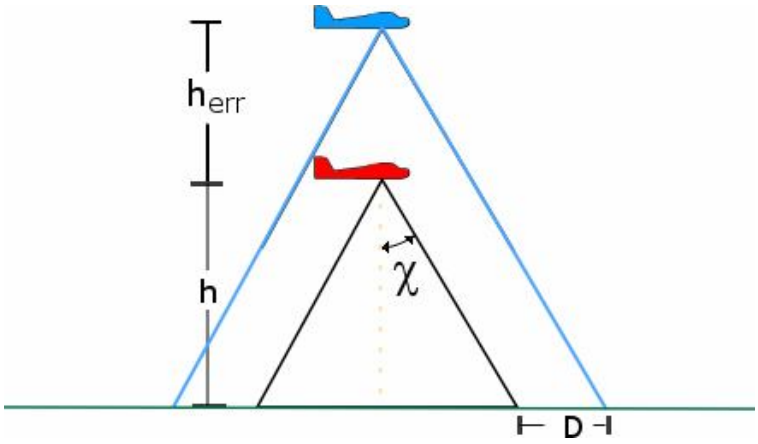


Figure **44**: Diagram of Offset Error Due to Height Error

Table **10** lists the expected target position offset due to 3-D position errors.

Table **10**: Target Position Offsets Due to Position Errors

	Position Error (meters)	
GPS Lat/Lon Error	1 to 3	Typical
1m Height Error	0.58	$\chi = 30^{o}$

At the altitude on the sixth pass, the total target position error due to 1^{o} angular misalignments in the Euler angles and a 1 meter height error could be up to 8 meters on the ground based on the geometry. It's difficult to say how accurate the Euler angle and position estimates are. From the data plotted in figure **41**, one can see there is about 8 meters in variation between the calculated target position points on the sixth pass, so it's possible that the angular misalignment and position errors used in the error analysis are close.

There is an insignificant error in the position estimate of the camera. The camera is not at the center of mass of the aircraft where the autopilot box sits. This is not compensated for in the software since it would be less than one meter. The GPS error in the position is significantly more than this. In a larger aircraft where the camera is mounted further away from the center of mass, this should be accounted for. Also, there is some image distortion due to the piece of Plexiglas in front of the camera, but this is negligible.

Chapter 6

Conclusions

Automatic target identification and target localization software was successfully flight tested on an ARL/PSU SIG Kadet Senior UAV. The results satisfactorily demonstrated the capability of correlating the image and telemetry data to project a camera view onto a virtual terrain model to determine the geographic location of a target in the camera view. The computer vision algorithm was very successful at altitudes of 67 to 75 meters above the ground. It positively identified nearly every instance that the targets were in view in the post-processing with the relaxed filter parameters. There were also no false positives of the target identified during the flight or post-processing. There had been a couple instances of false positives from red bushes with imagery taken in the autumn, but this shouldn't be an issue in the winter, spring, and early summer.

6.1 Future Work

More complex object recognition image processing algorithms could be developed and integrated with the target localization algorithm presented in this thesis and accompanying code. Also, once an object is found, it could be tracked in the video sequence using object tracking algorithms rather than searching every image for the object. Integration into the onboard Intelligent Controller might be useful in collaborative remote sensing between multiple autonomous UAVs. Combining this

target localization work with an onboard path planning algorithm which optimizes the time in which the target is in the camera view is the next step planned for this work.

A method for estimating camera misalignment, relative to the autopilot, based on collected image and telemetry data would be useful. This would minimize some of the error in the target localization.

The logical extension to this work might be using the images and virtual terrain model for conformal mapping to create accurate, high resolution aerial imagery from mosaics of the video frames or still images.

Bibliography

1. http://www.recherche.enac.fr/wiki/index.php/Main_Page

2. United States. Office of the Secretary of Defense. Unmanned Systems Roadmap (2007-2032), 1st ed. 10 December 2007.

3. Pearson, Lee. “Developing the Flying Bomb.” Naval Aviation in World War I. Ed. Adrian O. Van Wyen and the Editors of Naval Aviation News. Washington D.C.: The Chief of Naval Operations, 1969. 70-73.

4. http://www.roynagl.0catch.com/seaplanes9.htm

5. http://www.designation-systems.net/dusrm/app4/sperry-fb.html

6. Newcome, Laurence. Unmanned Aviation: A Brief History of Unmanned Aerial Vehicles. AIAA, 2004.

7. http://www.vectorsite.net/twuav_01.html

8. http://www.historicaircraft.org/British-Aircraft/pages/DeHavilland-QueenBee-1.html

9. Kalman, R. E. "A New Approach to Linear Filtering and Prediction Problems," Transactions of the ASME - Journal of Basic Engineering, ASME Vol. 82 (1960): 35-45.

10. Kalman, R. E., Bucy R. S., "New Results in Linear Filtering and Prediction Theory", Transactions of the ASME - Journal of Basic Engineering, ASME Vol. 83: (1961): 95-107.

11. http://www.vectorsite.net/twuav_04.html

12. Baldor, Lolita. Report: UAV use has doubled over 9 months. 3 January 2008. The Associated Press. <http://www.airforcetimes.com/news/2008/01/ap_uav_080101/>.

13. http://www.diydrones.com

14. Raytheon Company. Multi-Spectral Targeting System AN/AAS-52. U.S.A.: RSAS IMS, 2006.

15. http://www.cloudcaptech.com

16. http://procerusuav.com/cameraControl.php

17. Dufrene, W.R., Jr. "AI Techniques in Uninhabited Aerial Vehicle Flight." Aerospace and Electronic Systems Magazine, IEEE 19.8 (2004): 8-12.

18. Weiss, L., "Intelligent Collaborative Control for UAVs," AIAA Paper No. 2005-6900, AIAA InfoTech@Aerospace Conference, Washington D.C., Sept. 26-29, 2005.

19. Sinsley, G.L., Miller, J.A., Long, L.N., Geiger, B.R., Niessner, A.F., and Horn, J.F., "An Intelligent Controller for Collaborative Unmanned Air Vehicle." IEEE Symposium on Computational Intelligence in Security and Defense Applications, 2007, April 2007, pp. 139-144.

20. Miller, J. A., Minear, P. D., Niessner, A. F., DeLullo, A. M., Geiger, B. R., Long, L. N., and Horn, J. F., "Intelligent Unmanned Air Vehicle Flight Systems." Journal of Aerospace Computing, Information, and Communication, AIAA 4.5 (2007): 816-835.

21. http://www.osengines.com/engines/osmg0890.html

22. http://www.logitech.com/

23. http://www.sony.com/

24. Cloud Cap Technology 2004 Catalog. <http://www.cloudcaptech.com/download/Piccolo/Piccolo%20System%20Software/Version%201.2.4/Docs/Cloud%20Cap%20Catalog.pdf>.

25. http://www.intel.com/technology/computing/opencv/

26. http://tech.groups.yahoo.com/group/OpenCV/

27. Yasuyuki Matsushita, Eyal Ofek, Xiaoou Tang and Heung-Yeung Shum, "Full-Frame Video Stabilization with Motion Inpainting," IEEE International Conference on Computer Vision and Pattern Recognition, Vol. 1, (2005): 50-57.

28. Ready, B.B.; Taylor, C.N.; Beard, R.W., "A Kalman-filter Based Method for Creation of Super-resolved Mosaicks," Proceedings 2006 IEEE International Conference on Robotics and Automation, (2006): 3417-3422.

29. Shoichi Arakil, Takashi Matsuoka, Haruo Takemura, and Naokazu Yokoya, "Real-time Tracking of Multiple Moving Objects in Moving Camera Image Sequences Using Robust Statistics," IEEE Fourteenth International Conference on Pattern Recognition, Vol. 2, (1998): 1433-1435.

30. Sanjeev Arulampalam, M., Maskell, S., Gordon, N., and Clapp, T., “A tutorial on particle Flters for online nonlinear/non-Gaussian Bayesian tracking," IEEE Trans. on Signal Processing, Vol. 50, No. 2, (2002): 174-188.

31. Hue, C., Le Cadre, J.-P., and Perez, P., “Sequential Monte Carlo Method for Multiple Target Tracking and Data Fusion," IEEE Trans. on Signal Processing, Vol. 50, No. 2, (2002): 309-325.

32. Rocco V. Dell’Aquila, Giampiero Campa, Marcello R. Napolitano and Marco Mammarella. “Real-Time Machine-Vision-Based Position Sensing System For UAV Aerial Refueling,” Journal of Real-Time Image Processing, Vol. 1, No. 3, (2007): 213-224.

33. Emanuele Trucco, Alessandro Verri. “Introductory Techniques for 3-D Computer Vision.” Upper Saddle River, New Jersey: Prentice-Hall, Inc., 1998.

34. Leavers, Violet F. “Shape detection in computer vision using the Hough transform.” Springer-Verlag, 1992.

35. DeLullo, Anthony. “Computer Vision-Based Tracking Using Triangulation for Coordinated, Autonomous Unmanned Aerial Vehicles.” ME thesis, The Pennsylvania State University, 2006.

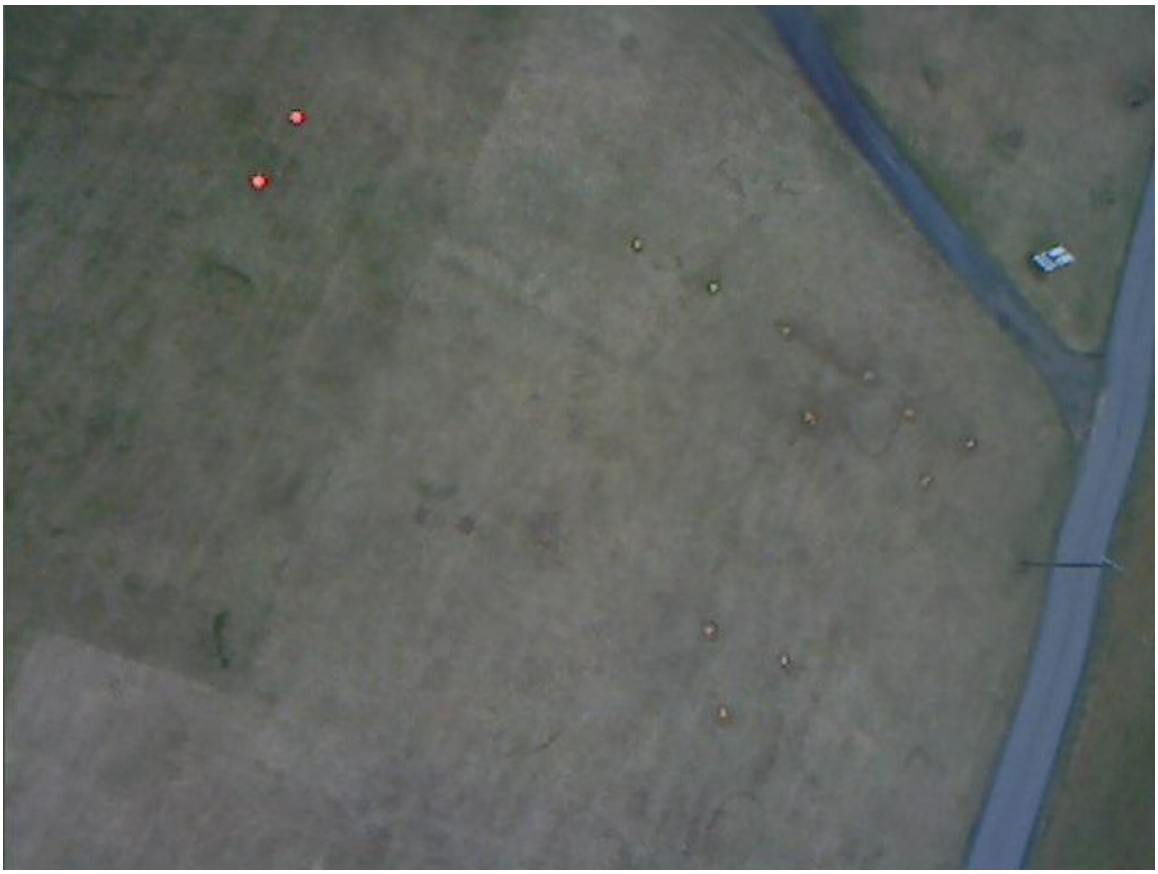
36. http://seamless.usgs.gov/

37. http://www.globalmapper.com/

Appendix A

Test Flight Photographs

A.1 Pass #6 (Image Frames 139.jpg through 152.jpg)



Appendix B

Computer Codes

B.1 ballfinding.h

```
#include "worldmodel.h"
struct Blob
{
  CvPoint min;
  CvPoint max;
  int px_count;
  double hw_ratio; // height to width ratio of blob bounding box
  double size; // in meters
  GridElement location; // GPS location and local coordinates
};

// This function takes an image array, searches for one value and
// replaces it with another value.  x and y are the starting points of
// the search area.
void UpdateBlobID(IplImage* back_project, int find_val,
                  int replace_val, int x, int y) {
  int count = 1;
  int i,j,k = 1;
  int a,b,c;

  CvSize size;
  size.width = back_project->width;
  size.height = back_project->height;

  cvSetReal2D(back_project,y,x,replace_val);

  while(count!=0)
  {
    count = 0;
    if(x-k<0)
      a = 0;
    else
      a = x-k;

    if(x+k>size.width-1)
      b = size.width-1;
    else
      b = x+k;

    if(y-k<0)
      c = 0;
    else
      c = y-k;

    for(i=a;i<=b;i++) {
      if(cvGetReal2D(back_project,c,i) == find_val) {
```

```
        cvSetReal2D(back_project,c,i,replace_val);
        count++;
      }
    }
    for(j=c;j<=y;j++) {
      if(cvGetReal2D(back_project,j,a) == find_val) {
        cvSetReal2D(back_project,j,a,replace_val);
        count++;
      }
    }
    for(j=c;j<=y-1;j++) {
      if(cvGetReal2D(back_project,j,b) == find_val) {
        cvSetReal2D(back_project,j,b,replace_val);
        count++;
      }
    }
    k++;
  }
}

// This is the function for finding all the red blobs in color_img
// and returning back_project1 and red_blobs
void FindRedBlobs(IplImage* color_img, IplImage* back_project1,
          Blob* red_blobs, int blob_px_lower_limit_filter,
          int blob_px_upper_limit_filter, int dilate,
          double hue_upper_limit, double hue_lower_limit,
          double saturation_lower_limit, double height_width_ratio,
          double ratio_pixels) {
  int i,j,k,l,x1,y1,x2,y2,blob_index,count;
  double xDim, yDim;
  CvScalar s,above,left,low,high,blob;
  CvScalar zero = cvScalar(0);
  CvScalar max = cvScalar(0);
  CvScalar min = cvScalar(180);
  IplImage* hsv_img;

  CvSize size = cvSize(color_img->width, color_img->height);
  hsv_img = cvCreateImage(size, IPL_DEPTH_8U, 3);

  // for dialated blobs
  IplImage* back_project2 = cvCreateImage(size, IPL_DEPTH_8U, 1);
  // for Blob IDs
  IplImage* back_project3 = cvCreateImage(size, IPL_DEPTH_8U, 1);
  // for Blob IDs of red pixels
  IplImage* back_project4 = cvCreateImage(size, IPL_DEPTH_8U, 1);

  cvSetZero(back_project1);
  cvSetZero(back_project2);
  cvSetZero(back_project3);
  cvSetZero(back_project4);

  cvCvtColor(color_img, hsv_img, CV_BGR2HSV);

  for(i=0; i<size.height; i++)
  {
    for(j=0; j<size.width; j++)
    {
      s = cvGet2D(hsv_img,i,j);
      if((s.val[0] > hue_upper_limit || s.val[0] < hue_lower_limit) && s.val[1]
> saturation_lower_limit)
      {
```

```
      cvSetReal2D(back_project1,i,j,255);
      // Dilating
      x1 = j - dilate;
      x2 = j + dilate;
      y1 = i - dilate;
      y2 = i + dilate;
      if(x1 < 0)
        x1 = 0;
      if(x2 > size.width-1)
        x2 = size.width-1;
      if(y1 < 0)
        y1 = 0;
      if(y2 > size.height-1)
        y2 = size.height-1;
      for(k=y1; k<=y2; k++)
        for(l=x1; l<=x2; l++)
          cvSetReal2D(back_project2,k,l,255);
    }
  }
}

// Finding connected blobs
blob_index = 1;
count = 1;
for(i=1; i<size.height-1; i++)
{
  for(j=1; j<size.width-1; j++)
  {
    s.val[0] = cvGetReal2D(back_project2,i,j);
    if(s.val[0]==255)
    {
      count ++;
      above.val[0] = cvGetReal2D(back_project3,i-1,j);
      left.val[0] = cvGetReal2D(back_project3,i,j-1);
      if(above.val[0] != 0 && left.val[0] != 0)
      {
        if(above.val[0] != left.val[0])
        {
          if(above.val[0] > left.val[0])
          {
            low = left;
            high = above;
          }
          else
          {
            low = above;
            high = left;
          }

          cvSetReal2D(back_project3,i,j,low.val[0]);
          // Looping through high values and setting to low
          UpdateBlobID(back_project3,(int)high.val[0],(int)low.val[0],j,i);
        }
        else
        {
          cvSetReal2D(back_project3,i,j,above.val[0]);
        }
      }
      else if(above.val[0] == 0 && left.val[0] == 0)
      {
        blob.val[0] = blob_index;
```

```
          cvSetReal2D(back_project3,i,j,blob.val[0]);
          blob_index++;
        }
        else if(above.val[0] != 0)
        {
          cvSetReal2D(back_project3,i,j,above.val[0]);
        }
        else
        {
          cvSetReal2D(back_project3,i,j,left.val[0]);
        }
      }
    }
  }

  //Initializing blob data
  for(k=0; k<256; k++)
  {
    red_blobs[k].px_count = 0;
    red_blobs[k].min = cvPoint(size.width+1, size.height+1);
    red_blobs[k].max = cvPoint(-1, -1);
  }

  for(i=0; i<size.height; i++)
  {
    for(j=0; j<size.width; j++)
    {
      if(cvGetReal2D(back_project1,i,j) == 255)
      {
        k = (int)cvGetReal2D(back_project3,i,j);
        if(k < 256 && k!=0)
        {
          red_blobs[k].px_count++;
          if(red_blobs[k].min.x > j)
            red_blobs[k].min.x = j;
          if(red_blobs[k].max.x < j)
            red_blobs[k].max.x = j;
          if(red_blobs[k].min.y > i)
            red_blobs[k].min.y = i;
          if(red_blobs[k].max.y < i)
            red_blobs[k].max.y = i;
          // Height to Width Ratio
          if (red_blobs[k].max.x - red_blobs[k].min.x != 0)
            red_blobs[k].hw_ratio = ((double)red_blobs[k].max.y -
(double)red_blobs[k].min.y)/((double)red_blobs[k].max.x -
(double)red_blobs[k].min.x);
          else
            red_blobs[k].hw_ratio = 0;
        }
      }
    }
  }

  count = 0;
  for(k=0; k<256; k++)
  {
    xDim = red_blobs[k].max.x-red_blobs[k].min.x;
    yDim = red_blobs[k].max.y-red_blobs[k].min.y;
    if(red_blobs[k].px_count > blob_px_lower_limit_filter &&
red_blobs[k].px_count < blob_px_upper_limit_filter && red_blobs[k].px_count >
```

```
xDim*yDim*ratio_pixels && yDim/xDim < height_width_ratio && xDim/yDim <
height_width_ratio)
    {
      count++;
      if(k!=count)
      {
        red_blobs[count] = red_blobs[k];
        red_blobs[k].px_count = 0;
        red_blobs[k].min = cvPoint(size.width+1, size.height+1);
        red_blobs[k].max = cvPoint(-1, -1);
      }
    }
    else
    {
      red_blobs[k].px_count = 0;
    }
  }

  // Releasing Memory
  cvReleaseImage(&hsv_img);
  cvReleaseImage(&back_project2);
  cvReleaseImage(&back_project3);
  cvReleaseImage(&back_project4);
}
```

B.2 worldmodel.h

```
#include <iostream>
#include <fstream>
#include <math.h>
using namespace std;

struct GridElement
{
  double xlocal; // Local flat earth coord sys X meas (meters)
  double ylocal; // Local flat earth coord sys Y meas (meters)
  double elevation; // Local flat earth coord sys Z meas (meters)
  double latitude; // World latitude coordinate (degrees)
  double longitude; // World longitude coordinate (degrees)
};
struct Euler
{
  double roll; // in radians
  double pitch; // in radians
  double yaw; // in radians
};
struct UAVState
{
  GridElement position;
  Euler orientation;
};
struct CamProp
{
  CvSize image_size;
  double HFOV; // in radians
  double VFOV; // in radians
  Euler orientation;
};

double m_per_deg( double lat_in_degrees ) {
  double PI = 3.141592653589793238;
  double D2R = PI/180;
  double R2D = 180/PI;
  double lat_in_rad = D2R*lat_in_degrees;
  return (111320 + 373*pow(sin(lat_in_rad),2))*cos(lat_in_rad);
}
GridElement ZeroGridElement(void) {
  GridElement foo;
  foo.latitude = 0;
  foo.longitude = 0;
  foo.elevation = 0;
  foo.xlocal = 0;
  foo.ylocal = 0;
  return foo;
}
GridElement AverageGridElements(GridElement A, GridElement B) {
  // Calculates center of two points
  GridElement C;
```

```
  C.elevation = (A.elevation + B.elevation)/2;
  C.latitude = (A.latitude + B.latitude)/2;
  C.longitude = (A.longitude + B.longitude)/2;
  C.xlocal = (A.xlocal + B.xlocal)/2;
  C.ylocal = (A.ylocal + B.ylocal)/2;
  return C;
}
double DistanceBetweenGridElements(GridElement A, GridElement B) {
  // Calculates distance between to points in meters
  return sqrt( pow((A.xlocal - B.xlocal),2) + pow((A.ylocal - B.ylocal),2) );
}
CvPoint InvertPoint(CvPoint point) { // Inverts CV Point
  return cvPoint(point.y, point.x);
}
void print(GridElement point) { // Prints coordinates to console
  cout << point.xlocal << "," << point.ylocal << ",";
  cout << point.elevation << "," << point.latitude;
  cout << "," << point.longitude << endl;
}
void print(Euler orientation) {
  cout << orientation.roll << "," << orientation.pitch << ",";
  cout << orientation.yaw << endl;
}
class World // Main World Class
{
    int i, j;
    string mystr;
    ifstream inFile;
    double geoid, *meters_per_degree, range;
    IplImage *clean_image;
    double lighting;
  public:
    int nrows, ncols;
    double xllcorner, yllcorner, cellsize;
    GridElement **grid;
    GridElement mins;
    GridElement maxs;
    IplImage *image, *gradient;


    World(const char* fileloc, double geoid_height) // Constructor
    {
      // Initializing some values
      geoid = geoid_height;
      mins.ylocal = 0;
      mins.elevation = 100000;
      maxs.xlocal = 0;
      maxs.elevation = 0;

      // Opening File
      inFile.open(fileloc);
      if (!inFile) {
        cerr << "Unable to open file";
        exit(1);   // Terminate
      }

      // Getting values
      for(i=1; i<=5; i++) {
        inFile >> mystr;
        if(mystr == "ncols")
          inFile >> ncols;
```

```
  else if(mystr == "nrows")
    inFile >> nrows;
  else if(mystr == "xllcorner")
    inFile >> xllcorner;
  else if(mystr == "yllcorner")
    inFile >> yllcorner;
  else if(mystr == "cellsize")
    inFile >> cellsize;
  else {
    cerr << "Unable to get AAI meta data" << endl;
    exit(1); // Terminate
  }
}

// Creating image
image = cvCreateImage( cvSize(ncols,nrows), IPL_DEPTH_8U, 3 );
gradient = cvCreateImage( cvSize(ncols,nrows), IPL_DEPTH_32F, 1);

mins.latitude = yllcorner;
mins.longitude = xllcorner;
mins.xlocal = -m_per_deg(mins.latitude)*cellsize*(ncols-1);
maxs.latitude = yllcorner + (nrows-1)*cellsize;
maxs.longitude = xllcorner + (ncols-1)*cellsize;
maxs.ylocal = 111320*cellsize*(nrows-1);

// Allocating
meters_per_degree = new double[nrows];
grid = new GridElement* [nrows];
for(i=0; i<nrows; i++) {
  // Setting each element to NULL in case of error later
  grid[i] = NULL;
}

// Populating
try {
  for(i=0; i<nrows; i++) {
    grid[i] = new GridElement[ncols];
    meters_per_degree[i] = m_per_deg((nrows-(i+1))*cellsize+yllcorner);
    for(j=0; j<ncols; j++) {
      inFile >> grid[i][j].elevation;
      //grid[i][j].elevation += geoid;
      if(grid[i][j].elevation > maxs.elevation)
        maxs.elevation = grid[i][j].elevation;
      if(grid[i][j].elevation < mins.elevation)
        mins.elevation = grid[i][j].elevation;
      grid[i][j].latitude = (nrows-(i+1))*cellsize+yllcorner;
      grid[i][j].longitude = j*cellsize+xllcorner;
      grid[i][j].xlocal = (j-ncols+1)*cellsize*meters_per_degree[i];
      grid[i][j].ylocal = (nrows-(i+1))*cellsize*111320;
    }
  }
  range = maxs.elevation - mins.elevation;
  for(i=0; i<nrows; i++)
  {
    for(j=0; j<ncols; j++) {

      if(i!=0 && i!=nrows-1)
        lighting = (grid[i+1][j].elevation - grid[i-1][j].elevation)/2;
      else if(i==0)
        lighting = grid[i+1][j].elevation - grid[i][j].elevation;
      else
```

```
            lighting = grid[i][j].elevation - grid[i-1][j].elevation;
          cvSet2D(image, i, j, GetGradientColor((grid[i][j].elevation -
mins.elevation)/range, (lighting+1.5)/1.5));
        }
      }
      clean_image = cvCloneImage(image);
    }
    catch(...) {
      for (i=nrows; i>0; i--)
        delete[] grid[i-1];
      delete[] grid;
      cerr << "Threw an exception while allocating memory" << endl;
      throw;    // Re-throw the current exception
    }
  }

  ~World() // Destructor
  {
    // Free memory and clean up
    for(i=nrows; i>0; i--)
      delete[] grid[i-1];
    delete[] grid;
    grid = NULL;
  }
  void CleanImage() {
    cvReleaseImage(&image);
    image = cvCloneImage(clean_image);
  }
  CvScalar GetGradientColor(double percent, double lighting = 1) {
    // percent (0.0-1.0)
    // lighting = (0.0-1.0), no color change when 0/pure black when 1
    if(lighting < 0)
      lighting = 0;
    if(lighting > 1)
      lighting = 1;

    int num_colors = 5;
    int section;
    CvScalar color;
    CvScalar grad_colors[5];

    grad_colors[0] = cvScalar(255,0,0);
    grad_colors[1] = cvScalar(255,255,0);
    grad_colors[2] = cvScalar(0,255,0);
    grad_colors[3] = cvScalar(0,255,255);
    grad_colors[4] = cvScalar(0,0,255);

    section = (int)floor(percent*num_colors);
    if(section>=num_colors-1)
      section = num_colors-2;
    if(percent >= 1) {
      percent = 1;
    }
    for (int i=0; i<3; i++)
    {
      color.val[i] = floor((grad_colors[section].val[i] + (percent*num_colors
- section)*(grad_colors[section+1].val[i]-
grad_colors[section].val[i]))*lighting);
    }
    return color;
  }
```

```
    GridElement LatLon2Local( GridElement latlon ) {
      // Calculations position in local coordinates given lat/lon
      latlon.xlocal = m_per_deg(latlon.latitude)*(latlon.longitude-
maxs.longitude);
      latlon.ylocal = 111320*(latlon.latitude-mins.latitude);
      return latlon;
    }
    GridElement Local2LatLon( GridElement local ) {
      // Calculations position in lat/lon coordinates given local
      local.latitude = yllcorner + local.ylocal/111320;
      local.longitude = maxs.longitude +
local.xlocal/m_per_deg(local.latitude);
      return local;
    }
    CvPoint GetClosestGridPoint( GridElement point ) {
      // Finds closest discrete grid point
      int i, j;
      // Given point.xlocal and point.ylocal
      i = round(nrows-1-point.ylocal/(cellsize*111320));
      if(i>nrows-1)
        i = nrows-1;
      if(i<0)
        i = 0;
      j = round(ncols-1+point.xlocal/(cellsize*meters_per_degree[i]));
      if(j>ncols-1)
        j = ncols-1;
      if(j<0)
        j = 0;
      return cvPoint(i,j);
    }
    GridElement GetTargetLocation(UAVState UAV, CvMat* TargetVector)
    {
      // TargetVector is a Unit Vector from the UAV to the target in the x,y,z
coordinates
      CvMat* R3 = cvCreateMat(3,3,CV_64FC1); // 3x3 rotation matrix of UAV
      CvMat* R3temp1 = cvCreateMat(3,3,CV_64FC1); // 3x3 matrix
      CvMat* R3temp2 = cvCreateMat(3,3,CV_64FC1); // 3x3 matrix
      CvMat* R3temp3 = cvCreateMat(3,3,CV_64FC1); // 3x3 matrix
      CvMat* invR3 = cvCreateMat(3,3,CV_64FC1); // inverse of R3 matrix
      CvPoint xy0,xy1,xy2; // index values for three close terrain grid points
      GridElement pointX;
      double point0[3], point1[3], point2[3], UAVxyz[3];
      int i;
      CvMat p0 = cvMat(3,1, CV_64FC1, point0);
      CvMat p1 = cvMat(3,1, CV_64FC1, point1);
      CvMat p2 = cvMat(3,1, CV_64FC1, point2);
      CvMat la = cvMat(3,1, CV_64FC1, UAVxyz);
      CvMat *lb = cvCreateMat(3,1, CV_64FC1);
      CvMat *px = cvCreateMat(3,1, CV_64FC1);


      RotationMatrix3(UAV.orientation.roll, UAV.orientation.pitch,
UAV.orientation.yaw, R3);

      RotationMatrixByAxes(UAV.orientation.yaw, 3, R3temp1);
      RotationMatrixByAxes(-UAV.orientation.roll, 2, R3temp2);
      cvMatMul(R3temp2, R3temp1, R3temp3); // R3temp2*R3temp1 -> R3temp3
      RotationMatrixByAxes(-UAV.orientation.pitch, 1, R3temp1);
      cvMatMul(R3temp1, R3temp3, R3); // R3temp1*R3temp3 -> R3

      cvInvert(R3, invR3); // inv(R3) -> invR3
```

```
      // Getting index values of closest grid point and two other
      // points for plane intersection calculations
      pointX = UAV.position; // First guess
      i=0;
      do
      {
        xy0 = GetClosestGridPoint(pointX);
        xy1.y = xy0.y;
        xy2.x = xy0.x;
        if(xy0.x == nrows-1)
          xy1.x = nrows-2;
        else
          xy1.x = xy0.x+1;
        if(xy0.y == ncols-1)
          xy2.y = ncols-2;
        else
          xy2.y = xy0.y+1;

        point0[0] = grid[xy0.x][xy0.y].xlocal;
        point0[1] = grid[xy0.x][xy0.y].ylocal;
        point0[2] = grid[xy0.x][xy0.y].elevation;
        point1[0] = grid[xy1.x][xy1.y].xlocal;
        point1[1] = grid[xy1.x][xy1.y].ylocal;
        point1[2] = grid[xy1.x][xy1.y].elevation;
        point2[0] = grid[xy2.x][xy2.y].xlocal;
        point2[1] = grid[xy2.x][xy2.y].ylocal;
        point2[2] = grid[xy2.x][xy2.y].elevation;

        UAVxyz[0] = UAV.position.xlocal;
        UAVxyz[1] = UAV.position.ylocal;
        UAVxyz[2] = UAV.position.elevation;

        cvMatMul(invR3, TargetVector, lb); // invR3*TargetVector -> lb
        cvAdd(&la,lb,lb);

        GetIntersectionPoint(&p0,&p1,&p2,&la,lb,px);

        pointX.xlocal = cvmGet(px,0,0);
        pointX.ylocal = cvmGet(px,1,0);
        pointX.elevation = cvmGet(px,2,0);
        pointX = Local2LatLon(pointX);
        i++;
        xy1 = GetClosestGridPoint(pointX);
      }
      while((xy0.x != xy1.x || xy0.y != xy1.y) && (i<10));

      return pointX;
    }
    void GetIntersectionPoint(CvMat* p0, CvMat* p1, CvMat* p2, CvMat* la,
CvMat* lb, CvMat* px)
    {
      // Formula for getting point of intersection between a plane and a line
      // p0, p1, p2 are three points on a plane, la and lb are points on a line
      // px is point of intersection, 3x1 vector
      // All arguments are 3x1 vectors
      // tuv = inv(A)*b;
      // | t |   | xa-xb  x1-x0  x2-x0 |-1  | xa-x0 |
      // | u | = | ya-yb  y1-y0  y2-y0 |  * | ya-y0 |
      // | v |   | za-zb  z1-z0  z2-z0 |    | za-z0 |
      // Function returns point of intersection, la+(lb-la)*t
```

```
  CvMat* tuv = cvCreateMat(3,1,CV_64FC1); // 3x1 vector
  CvMat A; // 3x3 matrix
  CvMat* Ainv = cvCreateMat(3,3,CV_64FC1); // 3x3 matrix
  CvMat b; // 3x1 vector

  CvMat* foo = cvCreateMat(3,1,CV_64FC1); // helper vector, 3x1
  CvMat* bar = cvCreateMat(3,1,CV_64FC1); // helper vector, 3x1
  double t;
  double Aarray[] = { cvmGet(la,0,0)-cvmGet(lb,0,0),
                      cvmGet(p1,0,0)-cvmGet(p0,0,0),
                      cvmGet(p2,0,0)-cvmGet(p0,0,0),
                      cvmGet(la,1,0)-cvmGet(lb,1,0),
                      cvmGet(p1,1,0)-cvmGet(p0,1,0),
                      cvmGet(p2,1,0)-cvmGet(p0,1,0),
                      cvmGet(la,2,0)-cvmGet(lb,2,0),
                      cvmGet(p1,2,0)-cvmGet(p0,2,0),
                      cvmGet(p2,2,0)-cvmGet(p0,2,0)};
  double barray[] = { cvmGet(la,0,0)-cvmGet(p0,0,0),
                      cvmGet(la,1,0)-cvmGet(p0,1,0),
                      cvmGet(la,2,0)-cvmGet(p0,2,0)};

  cvInitMatHeader(&A, 3, 3, CV_64FC1, Aarray);
  cvInvert(&A,Ainv); // inv(A) -> Ainv
  cvInitMatHeader(&b, 3,1, CV_64FC1, barray);
  cvMatMul(Ainv, &b, tuv); // Ainv*b -> tuv
  t = cvmGet(tuv,0,0);
  cvSub(lb, la, foo); // lb-la -> foo
  for(int i=0; i<3; i++)
    cvmSet(foo,i,0,cvmGet(foo,i,0)*t);
  cvAdd(la,foo,px); // la+bar -> px
}
void RotationMatrix3(double roll, double pitch, double yaw, CvMat* RM3)
{
  cvmSet(RM3,0,0,cos(roll)*cos(yaw));
  cvmSet(RM3,0,1,sin(roll)*sin(pitch)*cos(yaw)-cos(pitch)*sin(yaw));
  cvmSet(RM3,0,2,sin(roll)*cos(pitch)*cos(yaw)+sin(pitch)*sin(yaw));
  cvmSet(RM3,1,0,cos(roll)*sin(yaw));
  cvmSet(RM3,1,1,sin(roll)*sin(pitch)*sin(yaw)+cos(pitch)*cos(yaw));
  cvmSet(RM3,1,2,sin(roll)*cos(pitch)*sin(yaw)-sin(pitch)*cos(yaw));
  cvmSet(RM3,2,0,-sin(roll));
  cvmSet(RM3,2,1,cos(roll)*sin(pitch));
  cvmSet(RM3,2,2,cos(roll)*cos(pitch));
}
void RotationMatrix4(double roll, double pitch, double yaw, CvMat* RM4)
{
  double RM[] = {cos(roll)*cos(yaw),
                 sin(roll)*sin(pitch)*cos(yaw)-cos(pitch)*sin(yaw),
                 sin(roll)*cos(pitch)*cos(yaw)+sin(pitch)*sin(yaw),
                 0,
                 cos(roll)*sin(yaw),
                 sin(roll)*sin(pitch)*sin(yaw)+cos(pitch)*cos(yaw),
                 sin(roll)*cos(pitch)*sin(yaw)-sin(pitch)*cos(yaw),
                 0,
                 -sin(roll),
                 cos(roll)*sin(pitch),
                 cos(roll)*cos(pitch),
                 0,
                 0, 0, 0, 1};
  cvInitMatHeader(RM4, 4, 4, CV_64FC1, RM);
}
void RotationMatrixByAxes(double angle, int axes, CvMat* RM3)
```

```
    {
      // angle is in radians, axes are x,y,z (1,2,3)
      double c = cos(angle);
      double s = sin(angle);

      switch(axes) {
        case 1 :
          cvmSet(RM3,0,0,1);
          cvmSet(RM3,0,1,0);
          cvmSet(RM3,0,2,0);
          cvmSet(RM3,1,0,0);
          cvmSet(RM3,1,1,c);
          cvmSet(RM3,1,2,-s);
          cvmSet(RM3,2,0,0);
          cvmSet(RM3,2,1,s);
          cvmSet(RM3,2,2,c);
          break;
        case 2 :
          cvmSet(RM3,0,0,c);
          cvmSet(RM3,0,1,0);
          cvmSet(RM3,0,2,s);
          cvmSet(RM3,1,0,0);
          cvmSet(RM3,1,1,1);
          cvmSet(RM3,1,2,0);
          cvmSet(RM3,2,0,-s);
          cvmSet(RM3,2,1,0);
          cvmSet(RM3,2,2,c);
          break;
        case 3 :
          cvmSet(RM3,0,0,c);
          cvmSet(RM3,0,1,-s);
          cvmSet(RM3,0,2,0);
          cvmSet(RM3,1,0,s);
          cvmSet(RM3,1,1,c);
          cvmSet(RM3,1,2,0);
          cvmSet(RM3,2,0,0);
          cvmSet(RM3,2,1,0);
          cvmSet(RM3,2,2,1);
          break;
        default :
          cvmSet(RM3,0,0,1);
          cvmSet(RM3,0,1,0);
          cvmSet(RM3,0,2,0);
          cvmSet(RM3,1,0,0);
          cvmSet(RM3,1,1,1);
          cvmSet(RM3,1,2,0);
          cvmSet(RM3,2,0,0);
          cvmSet(RM3,2,1,0);
          cvmSet(RM3,2,2,1);
      }
    }
    void GetCameraUnitVector(UAVState UAV, CvMat* CUV)
    {
      // Doesn't do anything yet since the camera is
      // static relative to UAV coordinate system
      double CV[] = { 0, 0, -1};
      cvInitMatHeader(CUV, 3, 1, CV_64FC1, CV);
    }
    void GetTargetUnitVector(double px_x, double px_y, CamProp camera, CvMat*
TUV)
    {
```

```
      double roll = (px_x-
camera.image_size.width/2)*camera.HFOV/(camera.image_size.width-1);
      double pitch = -(px_y-
camera.image_size.height/2)*camera.VFOV/(camera.image_size.height-1);

      cvmSet(TUV,0,0,sin(roll)*cos(pitch));
      cvmSet(TUV,1,0,sin(pitch)*cos(roll));
      cvmSet(TUV,2,0,-cos(roll)*cos(pitch));

      double norm = sqrt(cvDotProduct(TUV,TUV));

      cvmSet(TUV,0,0,cvmGet(TUV,0,0)/norm);
      cvmSet(TUV,1,0,cvmGet(TUV,1,0)/norm);
      cvmSet(TUV,2,0,cvmGet(TUV,2,0)/norm);
    }
    void TranslationMatrix(double cx, double cy, double cz, CvMat* T)
    {
      double TM[] = { 1, 0, 0, -cx,
              0, 1, 0, -cy,
              0, 0, 1, -cz,
              0, 0, 0, 1};
      cvInitMatHeader(T, 4, 4, CV_64FC1, TM);
    }
    double GetElevation(double lat, double lon) {
      // Calculates elevation of terrain from model given lat/lon
      int i, j;
      if(IsInGrid(lat,lon)) {
        i = round((this->maxs.latitude - lat)/this->cellsize);
        j = this->ncols - 1 - round((this->maxs.longitude - lon)/this-
>cellsize);
        return this->grid[i][j].elevation;// - this->geoid;
      }
      else
        return 0;
    }
    bool IsInGrid(double lat, double lon) {
      // Checks if lat/lon is inside world terrain model
      if(lat >= this->mins.latitude && lat <= this->maxs.latitude && lon >=
this->mins.longitude && lon <= this->maxs.longitude)
        return 1;
      else
        return 0;
    }
    void PrintMatrix3(CvMat* M) { // Output 3x3 matrix to console
      cout << "| " << cvmGet(M,0,0) << "\t" << cvmGet(M,0,1) << "\t";
      cout << cvmGet(M,0,2) << "\t|" << endl;
      cout << "| " << cvmGet(M,1,0) << "\t" << cvmGet(M,1,1) << "\t";
      cout << cvmGet(M,1,2) << "\t|" << endl;
      cout << "| " << cvmGet(M,2,0) << "\t" << cvmGet(M,2,1) << "\t";
      cout << cvmGet(M,2,2) << "\t|" << endl;
    }
    int round(double d) { // Rounds a DOUBLE to INT
      int lo = d>0 ? (int)d : (int)d-1, hi = lo+1;
      return (d-lo)<(hi-d) ? lo : hi;
    }
};
```

B.3 Main Program

```
#include <cv.h>
#include <highgui.h>
#include <math.h>
#include <iostream>
#include <sstream>
#include <fstream>
#include <string>

#include "ballfinding.h"
#include "Exception.h"
#include "InitFile.h"
#include "simTimer.h"
#include "timer.h"
#include <conio.h> // windows only -- _kbhit()
#include "PiccoloOnboardComms.h"

using namespace std;

string itoa(int num) // Convert an integer to a string variable
{
    stringstream converter;
    converter << num;
    return converter.str();
}


int main(int argc, char **argv)
{
  int i,j,k; // Loop vars
  Blob red_blobs[256]; // Array of returned blobs from FindRedBlobs()
  GridElement blob_location[256]; // Blob geo-referenced location
  GridElement target_location; // Positive target identified location
  int blob_count, target_count = 0; // Counts of blobs and targets
  double r; // Radius between two blobs
  GridElement point1, point2, point3, point4; // Geo-ref'd locations
  UAVState UAV; // GPS/Euler state of UAV
  CamProp camera; // Camera properties
  bool showWindow = false; // Showing webcam image window
  bool recordData = false; // Recording image/telemetry data
  bool showMap = false; // Showing Map window
  bool showBackProject = false; // Showing back project window
  bool runLoop = true; // Running main loop
  unsigned long int nCycle = 0; // Number of cycles of main loop
  // Number of recorded frames inside main loop (when recordData=true)
  unsigned long int frameCount = 1;
  // These variables are set in the init file
  int dilate; // The number of pixels that each blob group will dialate
  int blob_px_lower_limit_filter; // Filters out small blobs
  int blob_px_upper_limit_filter; // Filters out large blobs
  double r_max_limit; // Maximum distance between red blobs (in meters)
  // Pixel ratio of rectangle around blob (height/width)
  double height_width_ratio;
```

```
  // Minimum ratio of pixels inside rectangle around blob (value = 0-1)
  double ratio_pixels;
  // Minimum ratio between sizes of blobs.
  // Two balls should be similar in size (0-1)
  double ratio_btw_blob_pixels;
  // Correcting terrain altitude for WGS 84
  double GPS_altitude_correction;
  // Upper (technically lower) limit in range of color to search for
  // (value = 0.0-180.0) (red = 0.89*180)
  double hue_upper_limit;
  // Lower (technically upper) limit in range of color to search for
  // (value = 0.0-180.0) (red = 0.1*180)
  double hue_lower_limit;
  // Limit of saturation (value = 0.0-255.0) (red saturation = 0.3*255)
  double saturation_lower_limit;
  string file_location; // File location of terrain data (AAI format)
  float appUpdateRate; // Contols main loop cycle time (in Hz)
  string file_output; // File output location for telemetry
  string file_output2; // File output location for targets

  string InitFileName("uav.ini"); // Filename for init file
  double targetcameraX; // X location in pixel space
  double targetcameraY; // Y location in pixel space
  CvMat* TV = cvCreateMat(3,1, CV_64FC1); // Target vector
  string filename; // Dynamically generated image filenames

  // Image capture variables
  IplImage* img = NULL; // Pointer to the image is created
  // Setting up capture from USB webcam
  CvCapture* capture = cvCaptureFromCAM( CV_CAP_ANY );

  PiccoloConnectInfo_t OnboardCInfo;

  CPiccoloOnboardComms *pOnboardComm;

  // Command Line Processing -- cascading cases
  switch (argc) {
    case 2:
      InitFileName.assign(argv[1]);
      break;
    default:
      cout << "Invalid command line arguments." << endl;
      cout << "Init filename required" << endl;
      exit(0);
  }

  // read inittializtion file
  InitFile IniFile(InitFileName.c_str(),'#');
  // register variables in init file
  IniFile.Register(&OnboardCInfo.COM_Port, 1, "ONB_COM_Port");
  IniFile.Register(&OnboardCInfo.serverListenPort, 1, "ONB_serverListenPort");
  IniFile.Register(&OnboardCInfo.netAddr, 1, "ONB_netAddr");
  IniFile.Register(&OnboardCInfo.logging, 1, "ONB_logging");
  IniFile.Register(&dilate, 1, "dilate");
  IniFile.Register(&blob_px_lower_limit_filter,  1,
"blob_px_lower_limit_filter");
  IniFile.Register(&blob_px_upper_limit_filter,  1,
"blob_px_upper_limit_filter");
  IniFile.Register(&r_max_limit, 1, "r_max_limit");
  IniFile.Register(&height_width_ratio, 1, "height_width_ratio");
  IniFile.Register(&ratio_pixels, 1, "ratio_pixels");
```

```
IniFile.Register(&ratio_btw_blob_pixels, 1, "ratio_btw_blob_pixels");
IniFile.Register(&GPS_altitude_correction, 1, "GPS_altitude_correction");
IniFile.Register(&camera.image_size.height, 1, "camera_image_size_height");
IniFile.Register(&camera.image_size.width, 1, "camera_image_size_width");
IniFile.Register(&camera.HFOV, 1, "camera_HFOV");
IniFile.Register(&camera.VFOV, 1, "camera_VFOV");
IniFile.Register(&camera.orientation.pitch, 1, "camera_orientation_pitch");
IniFile.Register(&camera.orientation.roll, 1, "camera_orientation_roll");
IniFile.Register(&camera.orientation.yaw, 1, "camera_orientation_yaw");
IniFile.Register(&hue_upper_limit, 1, "hue_upper_limit");
IniFile.Register(&hue_lower_limit, 1, "hue_lower_limit");
IniFile.Register(&saturation_lower_limit, 1, "saturation_lower_limit");
IniFile.Register(&file_location, 1, "file_location");
IniFile.Register(&appUpdateRate, 1, "appUpdateRate");
IniFile.Register(&file_output, 1, "file_output");
IniFile.Register(&file_output2, 1, "file_output2");

// Open Piccolo Communications
try {
  IniFile.ParseInput();

  if (OnboardCInfo.netAddr == "\"\"")
    OnboardCInfo.netAddr.clear();

  // Onboard piccolo
  pOnboardComm = new CPiccoloOnboardComms(OnboardCInfo);
}
catch (Exception &e) {
  cout << "\nError:\t" << e.What() << "\n"
       <<   "File: \t" << e.File() << "\n"
       <<   "Line: \t" << e.Line() << "\n";
  system("PAUSE");
  exit(0);
}

// Initializing output files
ofstream tel(file_output.c_str());
ofstream trgt(file_output2.c_str());

// Initializing back_project image for red pixels
IplImage* back_project = cvCreateImage(camera.image_size, IPL_DEPTH_8U, 1);

// Setting piccolo data stream to 20 hz
pOnboardComm->SetBandwidthMode_20Hz();
// Pointer to store telemetry
const UserData_t* const telem=pOnboardComm->GetPiccoloTelemetryPtr();

// Attempting to initialize world model
cout << "Initializing World Model..." << endl;
World world (file_location.c_str(), GPS_altitude_correction);

// Timer for main loop
simTimer ExecuteTimer(appUpdateRate);
cout << "appUpdateRate (Hz): " << appUpdateRate << endl;

// Setting output precisions
cout.precision(8);
tel.precision(14);
trgt.precision(14);

UInt32 OldSysTime=0;
```

```
// Looping until kbhit break
while (runLoop) {
  ++nCycle;
  //cout << "nCycle: " << nCycle << endl;

  // capture a frame
  if(!cvGrabFrame(capture)) {
    cout << "Could not grab a frame\n\7" << endl;
    system("PAUSE");
    exit(0);
  }


  // do the needed operations to send/receive over the serial link
  pOnboardComm->RunNetwork();

  // get buffered packets, extract data to internal storage variables
  pOnboardComm->GetAllPackets();

  // Check timing every so many seconds
  if (nCycle % (int)(appUpdateRate*5) == 0)
  {
    cout << "[" << telem->SystemTime << "] Time period: ";
    cout << (float)(telem->SystemTime-OldSysTime)/1000.0 <<;
    cout << " sec, #pkts: "<<pOnboardComm->GetNumPacketsRX() << endl;
  }
  OldSysTime = telem->SystemTime;

  // cvGrabFrame(capture) grabs the frame fast.
  // Then the code gets the telemetry data.
  // Then the image is processed from the captured frame
  img = cvRetrieveFrame( capture );

  // Updating UAV state
  UAV.position.latitude = telem->GPS.Latitude*180/3.14159265358979;
  UAV.position.longitude = telem->GPS.Longitude*180/3.14159265358979;
  UAV.position = world.LatLon2Local(UAV.position);
  UAV.position.elevation = telem->Alt;
  UAV.orientation.roll = telem->Euler[0];
  UAV.orientation.pitch = telem->Euler[1];
  UAV.orientation.yaw = telem->Euler[2];

  // Checking to see if UAV is inside world grid area
  if(!world.IsInGrid(UAV.position.latitude,UAV.position.longitude)) {
    cout << "UAV is not over world reference area\n\7" << endl;
    system("PAUSE");
    exit(0);
  }

  if(recordData) {
    // Writing out image
    filename = itoa(frameCount);
    filename += ".jpg";
    cout << "Saving Image as: " << filename << endl;
    cvSaveImage(filename.c_str(), img);
    // writing frame/time/telemetry data to file
    tel << frameCount << "," << telem->SystemTime << ",";
    tel << telem->GPS.Latitude << "," << telem->GPS.Longitude << ",";
    tel << telem->Alt << "," << telem->Euler[0] << ",";
    tel << telem->Euler[1] << "," << telem->Euler[2] << endl;
```

```
      frameCount++;
    }

    // Function returns back_project image and red_blobs locations
    FindRedBlobs(img, back_project, red_blobs,
            blob_px_lower_limit_filter, blob_px_upper_limit_filter,
            dilate, hue_upper_limit, hue_lower_limit,
            saturation_lower_limit, height_width_ratio, ratio_pixels);

    blob_count = 0;
    // Looping through all of the blobs obtained in FindRedBlobs()
    // and calculating geographic location
    for(k=1; k<256; k++) {
      blob_location[k-1] = ZeroGridElement();
      if(red_blobs[k].px_count>0) {
        // drawing rectangle around blob
        cvRectangle(img, red_blobs[k].min, red_blobs[k].max, CV_RGB(255,0,255),
1);
        targetcameraX = (double)red_blobs[k].min.x;
        targetcameraY = (double)red_blobs[k].min.y;
        // Annoying bug fix. Capturing from camera flips the
        // Y coordinate different than loading from image file
        // If loading from image file, remove this line
        targetcameraY = camera.image_size.height - 1 - targetcameraY;
        world.GetTargetUnitVector(targetcameraX, targetcameraY, camera, TV);
        point1 = world.GetTargetLocation(UAV, TV);

        targetcameraX = (double)red_blobs[k].max.x;
        targetcameraY = (double)red_blobs[k].max.y;
        // Annoying bug fix. Capturing from camera flips the
        // Y coordinate different than loading from image file
        // If loading from image file, remove this line
        targetcameraY = camera.image_size.height - 1 - targetcameraY;
        world.GetTargetUnitVector(targetcameraX, targetcameraY, camera, TV);
        point2 = world.GetTargetLocation(UAV, TV);

        red_blobs[k].size = DistanceBetweenGridElements(point1, point2);
        blob_location[k-1] = AverageGridElements(point1, point2);
        red_blobs[k].location = blob_location[k-1];
        //print(blob_location[k-1]);
        if(showMap)
          cvCircle(world.image,
InvertPoint(world.GetClosestGridPoint(blob_location[k-1])), 1,
cvScalar(255,0,255));
        blob_count++;
      }
    }
    // Looping through blob locations to get distance geographic between red
blobs
    target_count = 1;
    for(i=0; i<254; i++) {
      for(j=i+1; j<254; j++) {
        if(red_blobs[i+1].px_count>0 && red_blobs[j+1].px_count>0) {
          r = DistanceBetweenGridElements(blob_location[i], blob_location[j]);
          if(r < r_max_limit && red_blobs[i+1].px_count/red_blobs[j+1].px_count
> ratio_btw_blob_pixels && red_blobs[i+1].px_count/red_blobs[j+1].px_count <
1/ratio_btw_blob_pixels) {
            target_location = AverageGridElements(blob_location[i],
blob_location[j]);
            cout << "Target Location #" << target_count << ": ";
            cout << target_location.latitude << "\t";
```

```
            cout << target_location.longitude << "\t" << endl;
            cout << "Separation Dist.: " << r << " m" << endl;
            cout << "         Ball #1: " << red_blobs[i+1].px_count;
            cout << " px\t" << red_blobs[i+1].size << " m" << endl;
            cout << "         Ball #2: " << red_blobs[j+1].px_count;
            cout << " px\t" << red_blobs[j+1].size << " m" << endl;
            if(recordData) {
              trgt << frameCount-1 << "," << telem->SystemTime << ",";
              trgt << target_location.latitude << ",";
              trgt << target_location.longitude << endl;
            }
            target_count++;
          }
        }
      }
    }

    // Show image
    if(showWindow) {
      cvNamedWindow("Camera", 1 );
      if( img ) cvShowImage( "Camera", img );
      else cout << "No picture taken!!!" << endl;
    }
    if(showBackProject) {
      cvNamedWindow("Back Project", 1 );
      if( img ) cvShowImage( "Back Project", back_project );
      else cout << "No back project!!!" << endl;
    }
    if(showMap) {
      cvNamedWindow("Map", 0 );
      world.GetTargetUnitVector(0, 0, camera, TV);
      point1 = world.GetTargetLocation(UAV, TV);
      world.GetTargetUnitVector(639, 0, camera, TV);
      point2 = world.GetTargetLocation(UAV, TV);
      world.GetTargetUnitVector(639, 479, camera, TV);
      point3 = world.GetTargetLocation(UAV, TV);
      world.GetTargetUnitVector(0, 479, camera, TV);
      point4 = world.GetTargetLocation(UAV, TV);

      cvCircle(world.image,
InvertPoint(world.GetClosestGridPoint(UAV.position)), 2,
cvScalar(255,255,255));
      cvLine( world.image, InvertPoint(world.GetClosestGridPoint(point1)),
InvertPoint(world.GetClosestGridPoint(point2)), cvScalar(255,255,255));
      cvCircle(world.image, InvertPoint(world.GetClosestGridPoint(point1)), 1,
cvScalar(255,128,128));
      cvLine( world.image, InvertPoint(world.GetClosestGridPoint(point2)),
InvertPoint(world.GetClosestGridPoint(point3)), cvScalar(255,255,255));
      cvLine( world.image, InvertPoint(world.GetClosestGridPoint(point3)),
InvertPoint(world.GetClosestGridPoint(point4)), cvScalar(255,255,255));
      cvLine( world.image, InvertPoint(world.GetClosestGridPoint(point4)),
InvertPoint(world.GetClosestGridPoint(point1)), cvScalar(255,255,255));
      cvShowImage( "Map", world.image );
      world.CleanImage();
    }
    // wait for a key
    cvWaitKey(1);

    // Check to see if it is time to exit
    if(_kbhit()) {
      int key = _getch();
```

```
      // Breaks out of loop
      if((key == 'q') || (key == 'Q'))
        runLoop = false;
      // Toggles showing latest image from camera
      if((key == 'w') || (key == 'W')) {
        if(showWindow == true) {
          showWindow = false;
          cout << "Closing CAM window" << endl;
          cvDestroyWindow("Camera");
        }
        else {
          showWindow = true;
          cout << "Showing CAM window" << endl;
        }
      }
      // Toggles showing back project image
      if((key == 'b') || (key == 'B')) {
        if(showBackProject == true) {
          showBackProject = false;
          cout << "Closing Back Project window" << endl;
          cvDestroyWindow("Back Project");
        }
        else {
          showBackProject = true;
          cout << "Showing Back Project window" << endl;
        }
      }
      // Toggles showing map image
      if((key == 'm') || (key == 'M')) {
        if(showMap == true) {
          showMap = false;
          cout << "Closing Map window" << endl;
          cvDestroyWindow("Map");
        }
        else {
          showMap = true;
          cout << "Showing Map window" << endl;
        }
      }
      // Toggles recording telemetry/image data
      if((key == 'r') || (key == 'R')) {
        if(recordData) {
          recordData = false;
          cout << "Pausing Recording Data" << endl;
        }
        else {
          recordData = true;
          cout << "Recording Data" << endl;
        }
      }
    }

    // control the speed at which the loop executes
    ExecuteTimer.Idle();
  }
  tel.close();
  trgt.close();
}
```