7-10-2009

# A GPU Stream Computing Approach to Terrain Database Integrity Monitoring

Sean Patrick McKeon

A GPU STREAM COMPUTING APPROACH TO TERRAIN DATABASE INTEGRITY MONITORING

by

SEAN PATRICK MCKEON

Under the Direction of Dr. Michael Weeks

## ABSTRACT

Synthetic Vision Systems (SVS) provide an aircraft pilot with a virtual 3-D image of surrounding terrain which is generated from a digital elevation model stored in an onboard database. SVS improves the pilot's situational awareness at night and in inclement weather, thus reducing the chance of accidents such as controlled flight into terrain. A terrain database integrity monitor is needed to verify the accuracy of the displayed image due to potential database and navigational system errors. Previous research has used existing aircraft sensors to compare the real terrain position with the predicted position. We propose an improvement to one of these models by leveraging the stream computing capabilities of commercial graphics hardware. "Brook for GPUs", a system for implementing stream computing applications on programmable graphics processors, is used to execute a streaming ray-casting algorithm that correctly simulates the beam characteristics of a radar altimeter during all phases of flight.

INDEX WORDS: Stream computing, Terrain database, Integrity monitor, Brook for GPUs, Synthetic vision, Digital elevation model, Graphics processor, Ray casting

# A GPU STREAM COMPUTING APPROACH TO TERRAIN DATABASE INTEGRITY MONITORING

by

SEAN PATRICK MCKEON

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2009

# A GPU STREAM COMPUTING APPROACH TO TERRAIN DATABASE INTEGRITY MONITORING

by

SEAN PATRICK MCKEON

Committee Chair: Dr. Michael Weeks

Committee: Dr. Scott Owen
Dr. Ying Zhu

Electronic Version Approved:

Office of Graduate Studies
College of Arts and Sciences
Georgia State University
August 2009

This thesis is dedicated to my wife, Alice,

my mother, Sheila,

and to the memory of my father, Donald.

## ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Michael Weeks, for all of his guidance and support.

I would also like to thank Dr. Scott Owen and Dr. Ying Zhu.

Finally, I would like to thank my family and friends for all of their encouragement and patience.

# TABLE OF CONTENTS

**ACKNOWLEDGEMENTS** . . . . . **v**

**LIST OF TABLES** . . . . . **vii**

**LIST OF FIGURES** . . . . . **viii**

**LIST OF ACRONYMS** . . . . . **xi**

**Chapter 1 INTRODUCTION** . . . . . **1**

**Chapter 2 BACKGROUND** . . . . . **4**

2.1 Synthetic Vision Systems . . . . . 4

2.2 GPU Stream Computing . . . . . 45

**Chapter 3 PROPOSAL** . . . . . **53**

**Chapter 4 EXPERIMENTS AND RESULTS** . . . . . **55**

4.1 Experiment 1 . . . . . 70

4.2 Experiment 1: Results . . . . . 70

4.3 Experiment 2 . . . . . 70

4.4 Experiment 2: Results . . . . . 74

4.5 Experiment 3 . . . . . 76

4.6 Experiment 3: Results . . . . . 78

**Chapter 5 CONCLUSIONS** . . . . . **87**

**Chapter 6 FUTURE WORK** . . . . . **89**

**REFERENCES** . . . . . **90**

**APPENDIX: ADDITIONAL SOURCE FILES** . . . . . **93**

# LIST OF TABLES

2.1 DEM Parameters . . . . . 15

2.2 Radar Altimeter Specifications . . . . . 37

## LIST OF FIGURES

1.1 Universal Avionics PFD with Vision-1™ SVS . . . . . 2

2.1 Key Elements of SVS . . . . . 5

2.2 VISTAS III Workstation . . . . . 6

2.3 Simulated Approach, Go Around, and Departure Scenarios at EGE . . . . . 8

2.4 VISTAS III Display Formats . . . . . 9

2.5 Universal Avionics ND with Vision-1™ Exocentric View . . . . . 11

2.6 DEM Circular and Linear Error Probabilities . . . . . 12

2.7 Structure of a UTM Meter Grid . . . . . 14

2.8 Structure of an Arc Second Grid . . . . . 14

2.9 SVS with Integrity Monitor . . . . . 17

2.10 Synthesized Terrain Profile . . . . . 19

2.11 Absolute Disparities Approaching AVL Runway 34 . . . . . 22

2.12 Absolute Disparities Approaching AVL Runway 16 . . . . . 23

2.13 $T$ and $\log Z$ Statistics Approaching AVL Runways 16 and 34 . . . . . 23

2.14 Synthesized Height vs. DTED Approaching AVL Runway 16 . . . . . 24

2.15 Synthesized Height vs. NGS Approaching AVL Runway 16 . . . . . 25

2.16 Radar Altimeter with Leading Edge Tracking . . . . . 26

2.17 Terrain Detection Using "Spot" Algorithm . . . . . 27

2.18 "Spot" Height vs. Original Algorithm Approaching AVL Runway 16 . . . . . 28

2.19 "Spot" Height vs. NGS Approaching AVL Runway 16 . . . . . 28

2.20 $T$ Statistic Comparison for Runway 16 Approaches . . . . . 29

2.21 $T_D$ Threshold Determined Using $H_0$ and $H_1$ Hypotheses . . . . . 31

2.22 SVS Prototype Hardware . . . . . 32

2.23 SVS Prototype Software Flowchart . . . . . 33

2.24 Approach Patterns to UNI Runways 7 and 25 . . . . . 33

2.25 DTED Level 0 vs. Level 1 Approaching UNI Runway 7 . . . . . 34

2.26 DTED Level 0 vs. Level 1 Approaching UNI Runway 25 . . . . . 34

2.27 Approach Patterns to AVL Runways 16 and 34 . . . . . 35

2.28 DTED Level 0 vs. Level 1 Approaching AVL Runway 16 . . . . . 35

2.29 DTED Level 0 vs. Level 1 Approaching AVL Runway 34 . . . . . 36

2.30 Pulse Radar Altimeter Tracking . . . . . 38

2.31 FM-CW Radar Altimeter Block Diagram . . . . . 38

2.32 FM-CW Radar Altimeter Triangle Wave Form . . . . . 39

2.33 RMS Error at AVL . . . . . 42

2.34 RMS Error at UNI . . . . . 43

2.35 Comparison of Approaches to AVL . . . . . 43

2.36 AVL Terrain Disparity Using 'Plumb-bob' Terrain Model . . . . . 44

2.37 AVL Terrain Disparity Using 'Spot' Terrain Model . . . . . 45

2.38 Matrix-Vector Multiplication $y = Ax$ . . . . . 48

2.39 Brook Implementation . . . . . 49

2.40 Brook Test Results . . . . . 50

2.41 Ray Tracing . . . . . 51

2.42 Streaming Ray Tracer . . . . . 52

3.1 Bilinear Interpolation . . . . . 53

4.1 Ray-Triangle Intersection . . . . . 56

4.2 Processing Time: 32 - 128 Triangles . . . . . 71

4.3 Processing Time: 256 - 2048 Triangles . . . . . 72

4.4 Beam Patterns . . . . . 73

4.5 Flight Simulation Environment: View 1 . . . . . 74

4.6 Beam Resolution Comparison . . . . . 75

4.7 Synthesized Terrain Profiles . . . . . 77

4.8 Flight Simulation Environment: View 2 . . . . . 78

4.9 Flight Simulation Environment: View 3 . . . . . 79

4.10 Radar Altitude Comparison: Flight Path 1 . . . . . 80

4.11 Synthesized Terrain Profile Comparison: Flight Path 1 . . . . . 81

4.12 Radar Altitude Comparison: Flight Path 2 . . . . . 82

4.13 Synthesized Terrain Profile Comparison: Flight Path 2 . . . . . 83

4.14 Radar Altitude Comparison: Flight Path 3 . . . . . 85

4.15 Synthesized Terrain Profile Comparison: Flight Path 3 . . . . . 86

## LIST OF ACRONYMS

| | |
|---|---|
| AGATE | Advanced General Aviation Transport Experiment |
| AGL | Above Ground Level |
| ALS | Airborne Laser Scanner |
| ASMD | Airport Safety Modeling Data |
| AvSP | Aviation Safety Program |
| CFIT | Controlled Flight Into Terrain |
| CPU | Central Processing Unit |
| DEM | Digital Elevation Model |
| DGPS | Differential Global Positioning System |
| DME | Distance Measuring Equipment |
| DTED | Digital Terrain Elevation Data |
| EFIS | Electronic Flight Instrument System |
| FAA | Federal Aviation Administration |
| FLIR | Forward-Looking Infrared Radar |
| FOV | Field Of View |
| GA | General Aviation |
| GPS | Global Positioning System |
| GPU | Graphics Processing Unit |
| HDD | Head-Down Display |
| HDG | Heading |
| HUD | Head-Up Display |
| ILS | Instrument Landing System |
| IMC | Instrument Meteorological Conditions |
| JANAIR | Joint Army-Navy Instrumentation Research |
| LAAS | Local Area Augmentation System |
| LCD | Liquid Crystal Display |

| | |
|---|---|
| LiDAR | Light Detection and Ranging |
| MSD | Mean-Square Difference |
| MSL | Mean Sea Level |
| NASA | National Aeronautics and Space Administration |
| ND | Navigation Display |
| NGS | National Geodetic Survey |
| NM | Nautical Mile |
| PFD | Primary Flight Display |
| SA | Situational Awareness |
| SVS | Synthetic Vision System |
| TAWS | Terrain Awareness and Warning System |
| USGS | United States Geological Survey |
| VISTAS | Visual Imaging Simulator for Transport Aircraft Systems |
| VOR | Very High Frequency Omni-directional Range |
| VSD | Vertical Situation Display |

# Chapter 1

# INTRODUCTION

A study of 132 flight accidents between 1984 and 1993 revealed that 41% could be attributed to Controlled Flight Into Terrain (CFIT) [1]. CFIT and Loss of Control accidents can occur when a pilot is unaware of the orientation or proximity of his aircraft with respect to the surrounding terrain, often because of limited visibility due to darkness or Instrument Meteorological Conditions (IMC), i.e. inclement weather. Several government agencies created programs in the late 1990s to improve flight safety and prevent such accidents. The National Aeronautics and Space Administration (NASA) created its Aviation Safety Program (AvSP) in 1998 with the goal of reducing the fatal accident rate by 80% in ten years [2]. That same year, the Federal Aviation Administration (FAA) created its Safer Skies initiative with a similar goal [3]. Both programs identified the development of Synthetic Vision Systems (SVS) as an important strategy in achieving these goals.

Synthetic Vision Systems are designed to provide a pilot with improved Situational Awareness (SA), especially with regard to the position of terrain around the aircraft. A synthetic 3-D terrain image is typically displayed on a Liquid Crystal Display (LCD) screen on the pilot's control panel (Figure 1.1 [4]) and represents what the pilot would see out of the windshield during clear, sunny conditions. The improved SA provided by such systems can help deter CFIT accidents. Although the popularity of these systems has grown tremendously over recent years, the FAA has placed restrictions on their use due to the lack of a quantifiable means of verifying the integrity of the terrain database upon which the images are based [5].

Several models for terrain database integrity monitors have been proposed [1, 5, 6], which use terrain returns from an existing sensor on the aircraft (e.g. radar altimeter, weather radar) to compare against the 3-D terrain database, which is typically in the form of a Digital Elevation Model (DEM). Radar altimeters use radar returns to measure the distance


Figure 1.1. Universal Avionics PFD with Vision-1™ SVS

to terrain directly below the aircraft (i.e. radar altitude), while weather radar systems in terrain mapping mode provide distance to terrain in front of the aircraft (i.e. slant range). Other research introduces integrity-monitoring schemes based on new sensor technology such as Light Detection and Ranging (LiDAR) [7]. All of these models use algorithms to derive a synthetic height or distance to the digital terrain for comparisons with actual sensor returns. We propose an improved algorithm for synthesizing the aircraft sensor returns by leveraging the stream computing capabilities of a commercial Graphics Processing Unit (GPU). A rendering technique called ray casting is used to solve a non-graphical problem, the simulation of a radar altimeter antenna beam. The algorithm is implemented using "Brook for GPUs", a system for developing and executing stream computing applications on programmable graphics hardware.

The remainder of this work is presented as follows. First, background information covering synthetic vision and terrain database integrity monitoring fundamentals and prior research is presented, followed by a similar overview of previous GPU stream computing research. Second, we present our proposal for synthesizing radar altimeter sensor data using our streaming ray-casting algorithm. Third, we describe our experiments for testing the streaming ray-casting algorithm on a commercially available graphics card. We then analyze the results of our experiments and draw conclusions as to the viability of our proposed method for use in a terrain database integrity monitor. Finally, possible areas for future research work are considered.

# Chapter 2

# BACKGROUND

## 2.1 Synthetic Vision Systems

A Synthetic Vision System (SVS) is an electronic means of displaying a computer-generated 3-D image of the external scene topography from the pilot's perspective that is derived from aircraft state data, a precise navigation position, and a geospatial database of terrain, obstacles, and possible relevant cultural features such as runways [8]. The computer-generated image can be presented to the pilot by means of either a Head-Up Display (HUD) or Head-Down Display (HDD), with the head-down image often being incorporated into existing Primary Flight Display (PFD) instrumentation (Figure 1.1 [4]). This technology has the potential to reduce fatal accidents by improving the pilot's Situational Awareness (SA) when flying at night or in low-visibility conditions. SA has been defined as "the pilot's awareness and understanding of all factors that will contribute to the safe flying of their aircraft under normal and non-normal conditions" [9]. Terrain awareness is an essential part of SA. Pilot awareness of the aircraft's spatial position relative to the surrounding terrain is especially important during critical phases of flight such as instrument approaches.

A basic Synthetic Vision System consists of three key elements: an SVS display, a database, and a navigation system (Figure 2.1 [10]). The display provides the pilot with real-time imagery of environmental data and mission critical information. The environmental data is provided by the database and can include terrain, obstacles, airports, and navigation aids. The mission critical information is provided by the navigation system and can include aircraft current position, attitude, and sensor integrity. The aircraft position and attitude are also provided to the database and are used to determine the location and orientation within the digital terrain model from which to render the environmental data. Both the

database and navigation system must be highly accurate or the imagery on the SVS display may not match reality [10].

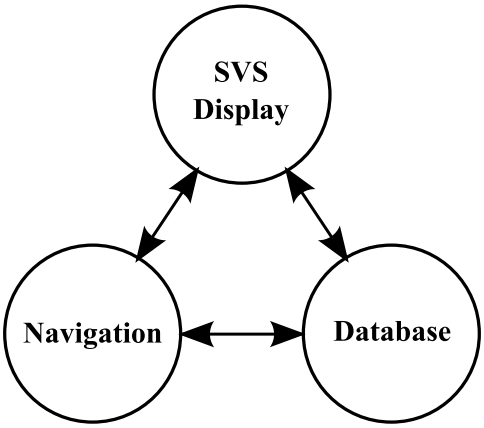



Figure 2.1. Key Elements of SVS

Synthetic Vision Systems are often confused with Enhanced Vision Systems (EVS). EVS uses advanced sensor technology such as millimeter wave radar and Forward-Looking Infrared Radar (FLIR) to improve the visibility of terrain, obstacles, and runways in low-visibility conditions. EVS does not rely on an onboard terrain database for imagery. It depends on sensors that "see" beyond human visual range to penetrate darkness and weather, then typically presents those images in a HUD format. The enhanced imagery generated for the display should always match up with actual terrain that the pilot would see through the HUD screen and windshield during clear, sunny conditions. In contrast, SVS is completely dependent on the onboard terrain database to generate its images. The aircraft sensors used to monitor the SVS terrain database are not directly used to create terrain images, just to verify the database integrity. SVS and EVS imagery can be presented in stand-alone applications, but research has also been done to integrate information from both technologies [2]. Our research is strictly focused on SVS and the digital terrain databases that this technology depends on.

SVS concepts originated in the 1950s with the Joint Army-Navy Instrumentation Research (JANAIR) Program. However, it took several decades for advances in electronic

displays to mature to the point of allowing those early proposals to come to fruition. In 1994, NASA, FAA, and private industry started the Advanced General Aviation Transport Experiment (AGATE) with the goal of developing flight displays that provided virtual clear and sunny weather conditions for pilots on a full-time basis [8].

One of the driving forces behind the development of SVS was the goal of reducing the fatal accident rate due to Controlled Flight Into Terrain (CFIT). These accidents occur when an airworthy aircraft, under pilot control, is flown into terrain due to inadequate awareness by the pilot of the impending disaster [3]. NASA's Aviation Safety Program (AvSP) and the FAA's Safer Skies initiative were both started in the late 1990s to identify new technologies that would improve cockpit safety. SVS was recommended as a technology capable of improving flight safety by reducing pilot workload.

Synthetic Vision has been proven to reduce pilot workload and help prevent CFIT accidents. An experiment performed at the NASA Langley Research Center in 2003 determined that SVS actually improved pilot's situational awareness and helped deter CFIT accidents. Sixteen pilots flew multiple, simulated approaches and departures into Colorado's Eagle County Regional Airport (EGE) using NASA's Visual Imaging Simulator for Transport Aircraft Systems (VISTAS) III (Figure 2.2 [9]). The pilots had an average of 19.6 years flying

Figure 2.2. VISTAS III Workstation

experience with an average of 8200 hours logged. The test group consisted of 4 commercial airline captains, 11 first officers, and one NASA researcher with Air Force transport

aircraft experience [9]. The Eagle County airport was chosen due to the surrounding mountainous terrain, which makes flight operations especially hazardous and thus ideal for CFIT avoidance testing.

Figure 2.3 [9] shows the scenarios flown by the test subjects. The pilots executed a series of approach, go around, and departure maneuvers in simulated low-visibility conditions using instruments only. We added the mountain symbol, color key, and acronym definitions to the original drawing to improve readability. All distances shown are in Nautical Miles (NM) relative to the Snow VOR/DME (SXW) ground station, which provides distance and bearing information to pilots flying in the Eagle County airspace. All aircraft headings (HDG) are relative to north, which is at 0°. The triangles show where changes in the flight segments occur. The scenario started at 5.3 nautical miles (NM) due east of the Snow VOR with the pilots then flying the "down wind" segment at a heading of 220° at a Mean Sea Level (MSL) height of 8100 feet. MSL is altitude relative to an ellipsoid representing the average level of the ocean's surface around the globe, thus providing a fixed height reference for mountainous terrain and other obstacles listed on a pilot's airport approach chart. The pilots then executed a right turn to approach Runway 7 from the southwest. On final approach to the runway at 200 feet Above Ground Level (AGL) the pilots executed a "go around", where the landing was aborted and the aircraft flew over the runway. At this point an engine loss was simulated with both throttles being set to 40% power, which forced the pilots to keep the aircraft closer to the terrain on departure. After passing over the runway, the pilots were given flight guidance for a heading of 50°. This heading eventually takes the aircraft towards a mountain top. During the nominal departure runs, additional flight guidance was provided to change the aircraft HDG to the 59° SXW VOR Radial (R-059°) at a distance of 6.8 NM DME. This turn allows the pilot to avoid the mountain top. During the CFIT data runs, the flight guidance for the turn to 59° was not provided to the pilots. The scenarios ended at 12 NM DME for the nominal runs and after a CFIT event for the CFIT runs.

Each pilot flew the scenario a total of 22 times using 4 different display formats: a baseline Boeing 757 Electronic Flight Instrumentation System (EFIS) HDD, a size A (5" x

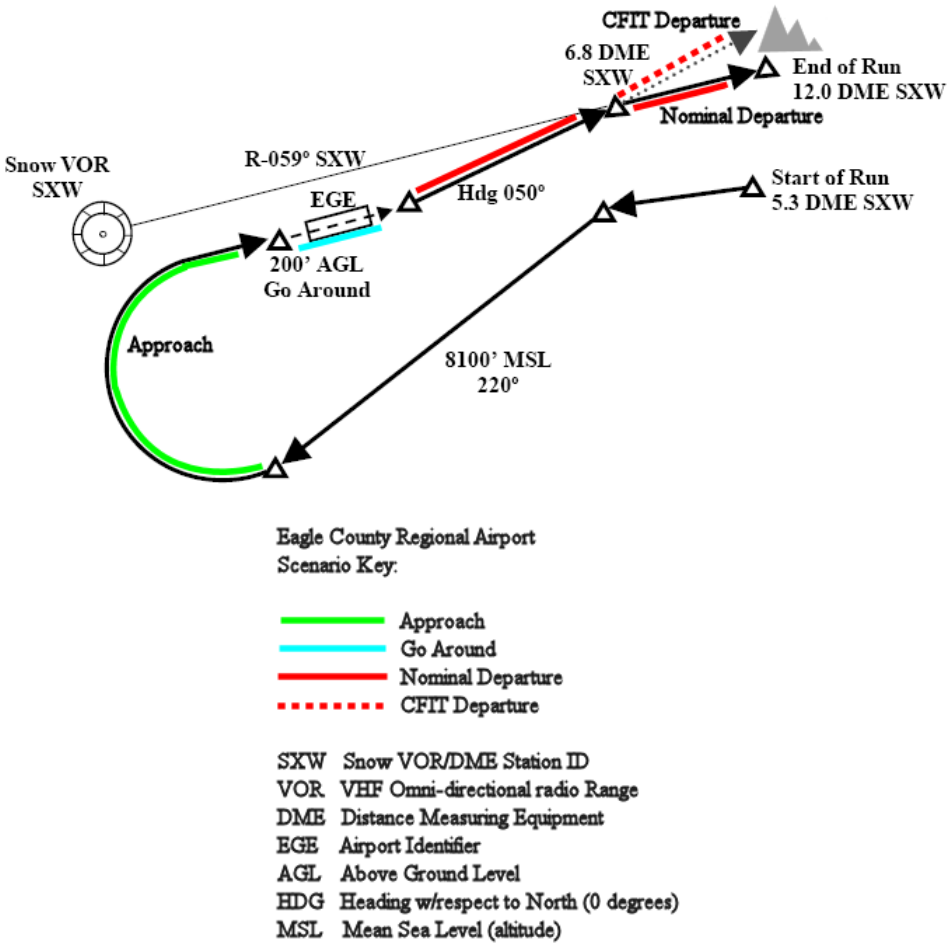



Figure 2.3. Simulated Approach, Go Around, and Departure Scenarios at EGE

5.25") SVS HDD, a size X (8" x 10") SVS HDD, and an SVS HUD (Figure 2.4 [9]). For all

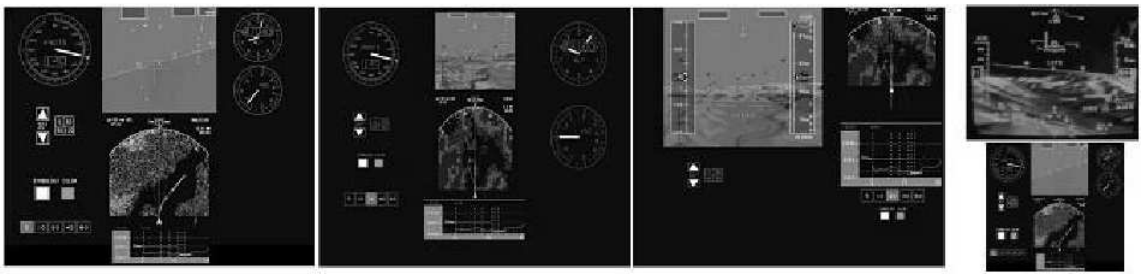


Figure 2.4. VISTAS III Display Formats: Baseline 757 EFIS (left), Size A SVS, Size X SVS, HUD SVS with EFIS HDD

flight tests a Navigation Display (ND) providing a Terrain Awareness and Warning System (TAWS) map view of the terrain was available, along with a Vertical Situation Display (VSD) showing a profile of the aircraft's altitude and position relative to approaching terrain. The pilots were not informed about the nature of the experiment, but were told to take whatever measures necessary to avoid terrain. A "rare event" CFIT scenario was secretly presented to each pilot for the final run, where the flight guidance provided on the departure path directed the pilot into a nearby mountain (see dotted path in Figure 2.3).

The results of the VISTAS III experiment showed a marked improvement in the pilots' terrain awareness when SVS technology was available. During the final run, 12 of the 16 pilots were provided with an SVS display, while the other four only had the baseline EFIS display. All 12 pilots using SVS were able to identify and avoid the CFIT event, while the 4 pilots without Synthetic Vision had a CFIT event. Three of the 4 pilots actually flew into the terrain, while the other had a near miss, flying unknowingly within 58 feet of the mountain peak. In addition, two other CFIT incidents occurred during the final approach leg, prior to the actual data runs. In both cases, the pilots did not have an SVS display available.

Furthermore, statistical analysis of the pilots' lateral path error on approach and departure was shown to be significantly less for all three SVS solutions versus the baseline EFIS display. Subjective analysis using pilot questionnaires revealed that the pilots' SA with the

SVS displays was higher during both approach and departure segments compared with the baseline display. Pilot's also rated their workload as being significantly lower using the SVS displays during both those segments of flight [9]. These results prove that SVS provides a real improvement to cockpit safety, while helping to prevent CFIT accidents.

Although SVS has been proven to enhance the safety of flight operations it is not without certain limitations. Human factors issues such as restricted field of view (FOV), minification or scaling of imagery, low display resolution, and lack of depth perception cues can result in the loss or misinterpretation of terrain data imagery. The technology is also subject to errors caused by inaccuracies in the terrain database or the navigation system(s) providing the current aircraft coordinates. Many of these limitations have been identified and addressed through research and improvements in system design.

Field of view limitations are inherent in the technology due to the need to compress the pilot's out-of-the-windshield view of approaching terrain into a much smaller format, such as the size X (8" x 10") HDD. Display size is often limited due to the restricted space available on existing aircraft cockpit panels. Many commercially available synthetic vision displays have fairly narrow fields of view of approximately 30° to 70°, while the effective human FOV is approximately 200° horizontally and 150° vertically [11]. The reduced FOV on a typical SVS perspective or "egocentric" display format such as that shown in Figure 1.1 [4] means that important terrain features just outside this range will not be displayed.

SVS display formats may also suffer from a lack of depth perception cues, which would provide the pilot with a better sense of distance between the aircraft and approaching terrain. The "exocentric" view (Figure 2.5 [4]) was introduced on SVS displays in part to help compensate for this limitation. The exocentric view provides a"wingman's" perspective of the aircraft from slightly behind and above, that along with additional visual cues showing aircraft heading and altitude can further improve the sense of terrain awareness by the pilot.


Figure 2.5. Universal Avionics ND with Vision-1™ Exocentric View

### 2.1.1 Terrain Databases

The terrain images generated by a synthetic vision system are based on a digital elevation model (DEM). The DEM is a geo-spatial database consisting of a 2-dimensional array of elevation data points which represent the terrain height at each fixed geographic location. Figure 2.6 [12] shows some of the key features that make up a DEM. The top view (a) shows the basic grid layout with the elevation data points, or post positions, indexed by longitude and latitude. The post-spacing represents the distance between any two adjacent elevation samples. The closer the post-spacing, the greater the spatial resolution of the database. The circular error probability (CEP) defines the horizontal accuracy region of each post position based upon the error radius $\rho$. The orthogonal view (b) shows the linear error probability (LEP), which specifies the accuracy of the terrain database in the vertical direction [12]. The smaller the CEP and LEP, the greater the accuracy of the terrain database.

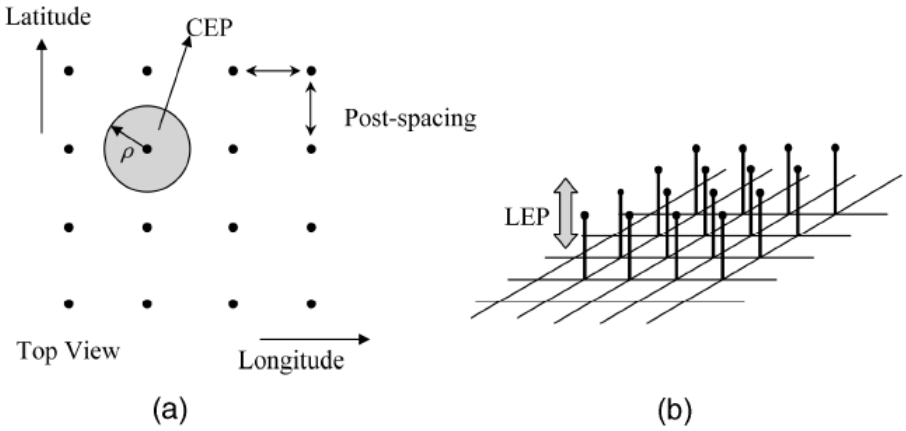



Figure 2.6. DEM Circular and Linear Error Probabilities, (a) CEP, (b) LEP

Most aviation terrain databases use a latitude/longitude/altitude (LLA) format to match the navigation standard for plotting position relative to the Earth's surface. However, DEMs are structured as elevation points above a 2-dimensional grid so there's a greater relative error between data points that are further away from each other due to the projection of the Earth's curved surface onto a plane. Data points that are in closer proximity have less relative error between them. The terrain height at each post position in the grid is measured

relative to a zero-altitude reference, which is normally the mean sea level (MSL) as determined by some vertical datum. However, depending on the accuracy of the method used to establish the MSL the measured terrain height can have variances of several meters [13].

Various means of data collection have been used over the years to create digital elevation models. Some vertical data from older terrain databases was obtained using spirit level surveying techniques [13]. Newer, more accurate technologies such as photogrammetry, hypsographic and hydrographic data digitizing, GPS surveying, and interferometry have mostly supplanted the older methods.

Digital elevation models are available from both the public and private sectors in a variety of formats. Digital Elevation Terrain Data (DTED®) level 0, level 1, and level 2 were created by the National Imagery and Mapping Agency for military use. Higher levels indicate greater resolution (see Table 2.1 [12]). The United States Geological Survey (USGS) has DEMs in several formats covering the contiguous United States, Alaska, and Hawaii. Airport safety monitoring data (ASM100 and ASM12) from the FAA, high-resolution National Geodetic Survey DEMs, and National Geospatial Intelligence Agency (NGA)/NASA elevation models created from the Shuttle Radar Topography Mission (SRTM) are some other examples of terrain data available from the public sector. Jeppesen is a provider of terrain and obstacle databases in the private sector [12].

Figures 2.7 and 2.8 [14] are examples of two different formats of DEM available from the USGS. Figure 2.7 shows the structure of a 7.5-minute DEM cast on a Universal Transverse Mercator (UTM) grid. The post spacing is 30 meters and each data block covers a 7.5- by 7.5-minute area of terrain. The elevation at each data point is decimal or whole units of meters or feet relative to the National Geodetic Vertical Datum of 1929 (NGVD 29). Data are ordered south to north in profiles ordered west to east. Profiles may have different numbers of elevation data points due to a variance in angle between true north and the grid north of the UTM coordinate system [14]. Figure 2.8 [14] shows the structure of a 1° digital elevation model indexed by longitude and latitude. Here the post spacing is 3 arc-seconds, which is approximately 90 meters when measured at the

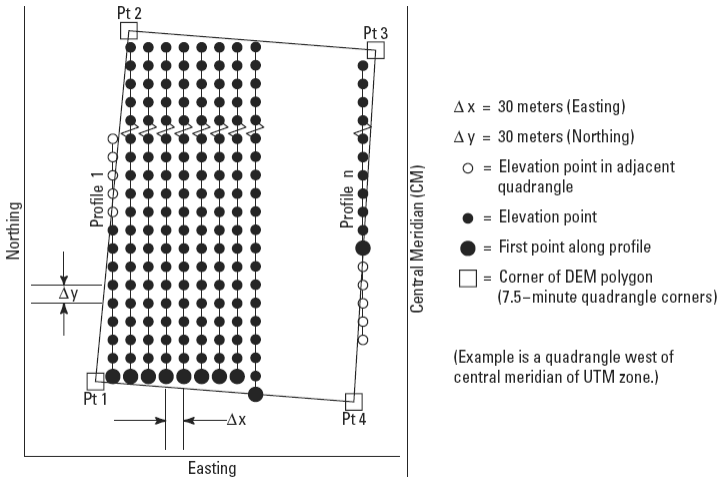



Figure 2.7. Structure of a UTM Meter Grid

equator, and each data block covers a 1- by 1-degree area of terrain. Elevation values are in meters relative to NGVD 29 and the data are ordered similarly to the 7.5-minute DEM [14]. Table 2.1 [12] shows a comparison of parameter values for different DEMs.

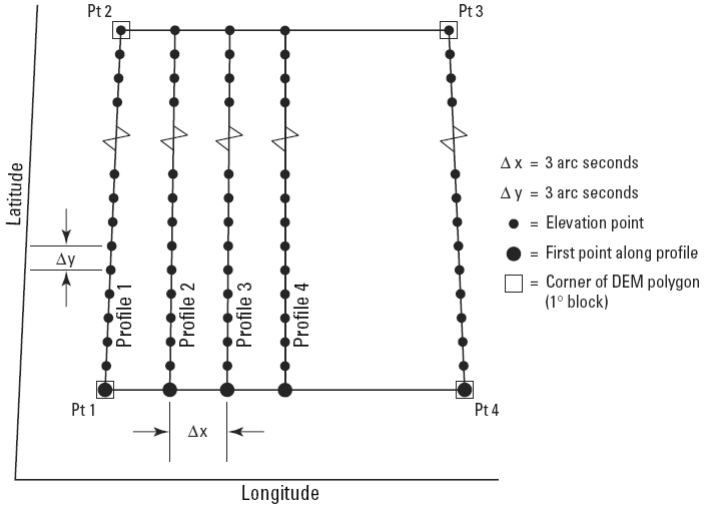



Figure 2.8. Structure of an Arc Second Grid

Spatial resolution is one of the most important characteristics of a DEM. The greater the distance between post locations the more likely that prominent terrain features will be missed

Table 2.1. DEM Parameters

| | Post Spacing | CEP | LEP | Horizontal Datum | Vertical Datum | Segment Size |
|---|---|---|---|---|---|---|
| DTED 0 | 30 arc-sec | < 50m, 90% | < 30m, 90% | WGS84 | EGM[1] 96 (MSL) | 1° × 1° |
| DTED 1 | 3 arc-sec | < 50m, 90% | < 30m, 90% | WGS84 | EGM[1] 96 (MSL) | 1° × 1° |
| DTED 2 | 1 arc-sec | < 50m, 90% | < 30m, 90% | WGS84 | EGM[1] 96 (MSL) | 1° × 1° |
| USGS 1 deg | 3 arc-sec | < 130m, 90% | < 30m, 90% | WGS84 | NGVD[2] 27 | 1° × 1° |
| USGS 7.5 min*** | 1 arc-sec | < 130m, 90% | < 30m, 90% | WGS84 | NAVD[3] 88 | 7.5′ × 7.5′ |
| ASM100 | 15 arc-sec | see DTED 1 | see DTED 1 | WGS84 | EGM[1] 96 (MSL) | 100 nmi × 100 nmi |
| ASM12 | 9 arc-sec | see DTED 1 | see DTED 1 | WGS84 | EGM[1] 96 (MSL) | 12 nmi × 12 nmi |
| NGS | 4*, 5** m | < 2m, 90% | < 4m, 90% | UTM | NAVD[3] 88 | 8.8 nmi × 3 nmi |

*Note:* *at AVL, **at EGE, ***put into usable form by Jepessen. [1]Earth gravitational mode, [2]national geodetic vertical datum, [3]national vertical datum.

between the measured data points. This is especially true for very rugged terrain [12]. NASA synthetic vision experiments have shown that a post spacing of 30 arc-seconds rounds off terrain peaks and fills in valleys, thus making the peaks appear less threatening and the valley terrain more so. The higher-resolution, 1- and 3-arc-second databases were preferred by pilots in the experiments, although the 30-arc-second database still provided better situational awareness than a conventional, non-SVS, instrument panel. Therefore, the FAA determined that the 30-arc-second resolution was the minimum safety standard for SVS applications [8].

Three primary types of error occur in terrain databases. Random errors are the normal operational errors that are specified by the CEP and LEP. These errors occur due to the method used to collect the elevation data, whether GPS surveying, photogrammetry, interferometry, or some other method. Systematic errors include elevation biases, rotational errors, and mismatches of terrain database data points that can occur when a DEM is pieced together from multiple sources. These errors are highly correlated between multiple data points in a terrain database segment and can occur in both the horizontal and vertical directions. Finally, blunders or "human errors" cover non-correlated mistakes at individual data points that occur during the terrain data collection process [12].

In order to verify the integrity of a terrain database there must be a means to compare the accuracy of *registration* or how closely the database elevation corresponds to the real-

world elevation of the terrain. Terrain database integrity monitoring can be performed using existing aircraft sensors to test the agreement between DEM data and the actual terrain.

### 2.1.2 Integrity Monitoring

The basic SVS design in Figure 2.1 can be improved with the addition of a terrain database integrity monitor (Figure 2.9, adapted from [10]). The purpose of the integrity monitor is to warn the pilot when the SVS should be used with caution or not at all [1]. The integrity monitor continuously compares actual terrain measurements with "synthesized" terrain measurements in real time. If the discrepancy between terrain measurements becomes too great the integrity monitor can trigger a warning on the SVS display or simply remove the environmental data from view. This is to prevent the display of hazardously misleading information on the SVS display, such as showing a clear flight path when there is actually a mountain ahead or vice versa. In this way integrity monitoring provides an additional layer of safety by detecting any significant database or navigation system errors.

Measurements of actual terrain near the aircraft can be obtained using a variety of sensor systems. These systems fall into two general classifications: downward-looking sensors and forward-looking sensors. Downward-looking sensors include radar altimeters and airborne laser scanners (ALS), while weather radar is an example of a forward-looking sensor. Radar altimeters and weather radar are sensors currently found on most commercial aircraft. Both systems have primary functions apart from database integrity monitoring, but could also be used for that secondary role. ALS is a recent sensor technology designed specifically for terrain detection and would need to be added to existing aircraft in order to support integrity monitoring.

The primary function of a radar altimeter is to provide terrain clearance or altitude with respect to the ground level directly below the aircraft [15]. Radar altimeters consist of a transmitter/receiver unit, a radar altitude display, and a pair of antennas mounted longitudinally along the belly of the aircraft. The forward antenna transmits a radar signal which bounces off terrain below the aircraft and returns to the aft antenna. By measuring

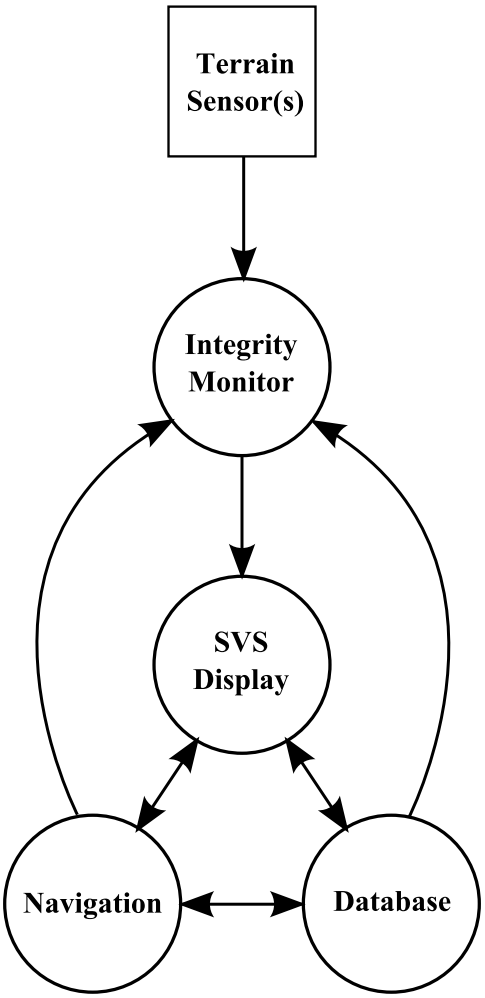



Figure 2.9. SVS with Integrity Monitor

the time it takes the signal to travel and return, the aircraft's altitude above the terrain can be derived then displayed to the pilot.

The primary function of weather radar is to detect the presence and features of weather formations in the aircraft's flight path [15]. This system consists of a transmitter/receiver unit, a weather display, and a radar dish mounted in the nose of the aircraft. Energy is radiated forward from the dish and reflected back from clouds and precipitation. Magnitude and range detection is performed on the returned signal, color-coded based on precipitation rate, and displayed to the pilot on the weather display. Some weather radars provide a terrain-mapping mode, which provides range detection of terrain features ahead of the aircraft.

ALS systems such as Light Detection and Ranging (LiDAR) have become very common in the Geographic Information Systems (GIS) community for use in generating terrain databases [16]. The high-accuracy and high-density characteristics of these systems also make them suitable for use in aircraft navigation and terrain database integrity monitoring. A LiDAR system consists of a downward-looking laser scanner, an Inertial Measurement Unit (IMU), and a Global Positioning System (GPS) receiver. The laser scanner sweeps from side to side, making hundreds of angle-encoded range measurements of the terrain below per sweep. The IMU provides the aircraft's flight attitude (i.e. pitch and roll) and thus the orientation of the scanner with respect to the terrain, while the GPS provides the navigation data, specifically the present position of the aircraft with respect to latitude and longitude. Combining the data from these systems allows for very precise, earth-referenced measurements of each point of terrain illuminated by the laser scanner [7].

Figure 2.10 (adapted from [17]) illustrates how a synthesized terrain profile can be generated for use in an integrity monitoring system. The components in this system include a radar altimeter for terrain sensing, a Differential Global Positioning System (DGPS) for navigation, and a Digital Elevation Model (DEM) for the terrain database. The radar altimeter measures $h_{radalt}$, the height of the aircraft directly above the terrain. The DGPS provides the latitude and longitude of the aircraft as well as $h_{DGPS}$, the height of the aircraft

above MSL. $h_{antenna}$ is the height difference between the antennas used in each system relative to the top and bottom of the aircraft. The height of the terrain below the aircraft can be synthesized by subtracting the radar altitude and antenna height from the DGPS height. This height, $h_{synth}$, can then be compared to the terrain height stored in the terrain database at those coordinates. Equation 2.1 [1, 17] shows how the synthesized height is calculated at any given time $t_i$.

$$h_{synth}(t_i) = h_{DGPS}(t_i) - (h_{radalt}(t_i) + h_{antenna}) \tag{2.1}$$

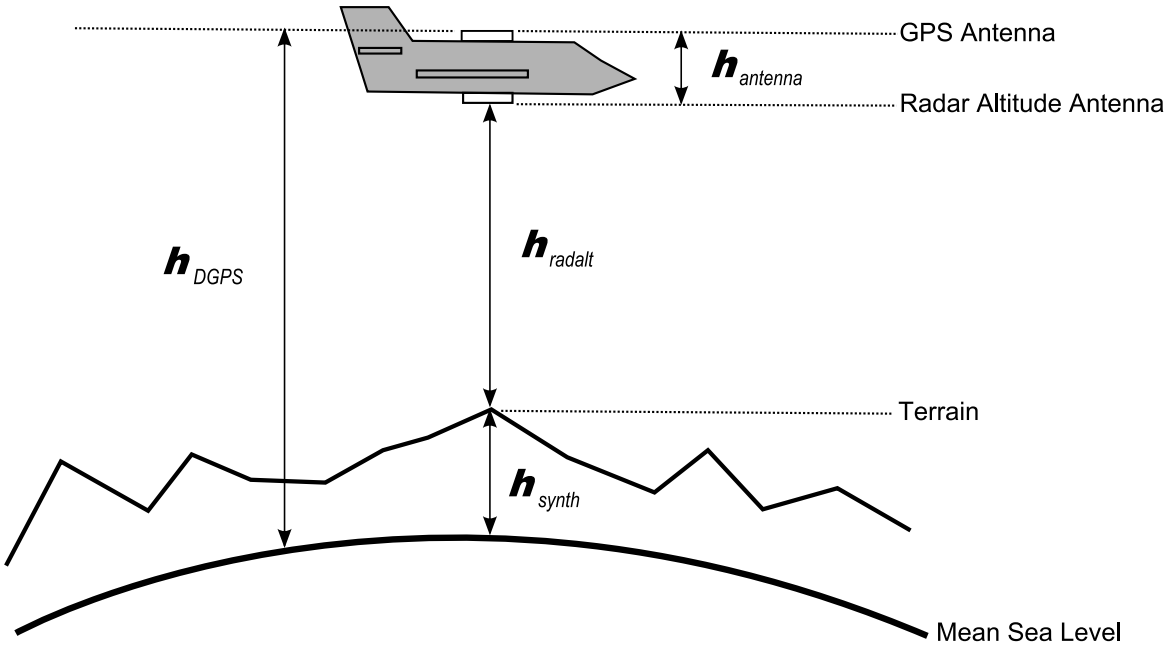



Figure 2.10. Synthesized Terrain Profile

Most of the pioneering work done in SVS integrity monitoring has come from joint research between Ohio University and NASA [1, 5–7, 16, 18–24]. Some of the first flight tests of a terrain database integrity monitor were performed in the fall of 1999 in the vicinity of the Asheville, NC airport (AVL) using an Air Force Convair aircraft specifically designed to be a "flying testbed" for new avionics systems [1]. The integrity monitoring system consisted of a DGPS and radar altimeter, while several terrain databases were used including the Airport Safety Modeling Data (ASM100 and ASM12), the Digital Terrain Elevation Data

(DTED level 1 and DTED level 2), and the United States Geological Survey (USGS) Digital Elevation Model (DEM).

The terrain database integrity monitor developed for the Asheville tests used the following simplified version of the synthesized terrain profile calculation (Equation 2.1), where the difference in antenna height ($h_{antenna}$) was not factored in,

$$h_{synth}(t_i) = h_{DGPS}(t_i) - h_{radalt}(t_i).$$

The integrity monitor also used the absolute and successive disparities between the synthesized height and the digital terrain height from the database as the basic metrics for determining the correctness of the database information. Equation 2.2 gives the absolute disparity between the synthetic height ($h_{synth}$) and the height derived from the Digital Terrain Elevation Data ($h_{DTED}$) at time $t_i$. Equation 2.3 gives the successive disparity between absolute disparity measurements at times $t_i$ and $t_{i-1}$.

$$p(t_i) = h_{synth}(t_i) - h_{DTED}(t_i) \tag{2.2}$$

$$s(t_i) = p(t_i) - p(t_{i-1}) \tag{2.3}$$

Successive disparity measurements were used to remove radar altimeter biases from the integrity monitor. However, this also had the undesirable consequence of causing bias-like errors in the terrain database to be missed [1]. These are the type of systematic errors that were discussed in Section 2.1.1.

Test statistics were derived from the absolute and successive disparities to measure the agreement between synthesized and digital terrain height during nominal or fault-free performance of the system. Equations 2.4 and 2.5 show the Mean-Square Difference (MSD) functions for the absolute and successive disparities respectively. MSD was selected due to its accuracy in terrain correlation applications [17]. The number of samples, $N$, was chosen to be 50 for this study. This value can be interpreted as an integration time of 50 seconds,

i.e. one radar altimeter measurement per second over 50 seconds [1]. $\sigma^2$ represents the variance from the absolute and successive disparity distributions, which were estimated to be $p(t_i) \sim N(0, (18.9)^2)$ and $s(t_i) \sim N(0, (13.0)^2)$ respectively [17].

$$T = \frac{1}{\sigma^2} \sum_{i=1}^{N} p^2(t_i) \tag{2.4}$$

$$Z = \frac{1}{2\sigma^2(N-1)} \sum_{i=2}^{N} s^2(t_i) \tag{2.5}$$

Probability density functions (PDFs) were determined using Equations 2.4 and 2.5. The PDFs were used to compute detection thresholds for the integrity monitor. The detection thresholds identified the points at which the absolute and successive disparities could become great enough to trigger a false alarm for a particular geographic area. The probability of false alarm or "fault free detection" ($P_{FFD}$) for this study was chosen to be $10^{-4}$. Given a $P_{FFD} = 0.0001$ and a sample size of $N = 50$, the absolute disparity threshold, $T_D$, was found to be 96, while the successive disparity threshold, $Z_D$, was found to be 2.2 [1].

The Asheville, NC (AVL) flight tests were performed by 3 evaluation pilots, flying a total of 53 approaches to runways 16 and 34. The pilots used an SVS display for primary tactical guidance cues. Navigation was provided by Ashtech Z-12 GPS receivers and a Honeywell AN/APN-171(V) radar altimeter system provided terrain sensing. The initial approach to runway 34 was characterized by large variations in the terrain profile, which became significantly smaller on final approach. The approach to runway 16 was characterized by significant variations in terrain throughout the landing segment up until the runway threshold [1].

Some results from the AVL flight tests are shown in Figures 2.11 and 2.12 [1]. The absolute disparities (AD) between the synthesized terrain height and the heights from the ASMD, DTED, and USGS terrain databases show some significant deviations from zero, indicating the presence of biases in either the databases, the radar altimeter, or both. There are also noticeable differences between the databases as well. The ASMD was derived from

the DTED, which explains why the disparities follow closely for those two databases. However, the variations in USGS data suggests some significant differences in vertical datum, possibly due to different techniques for deriving terrain elevation [1].

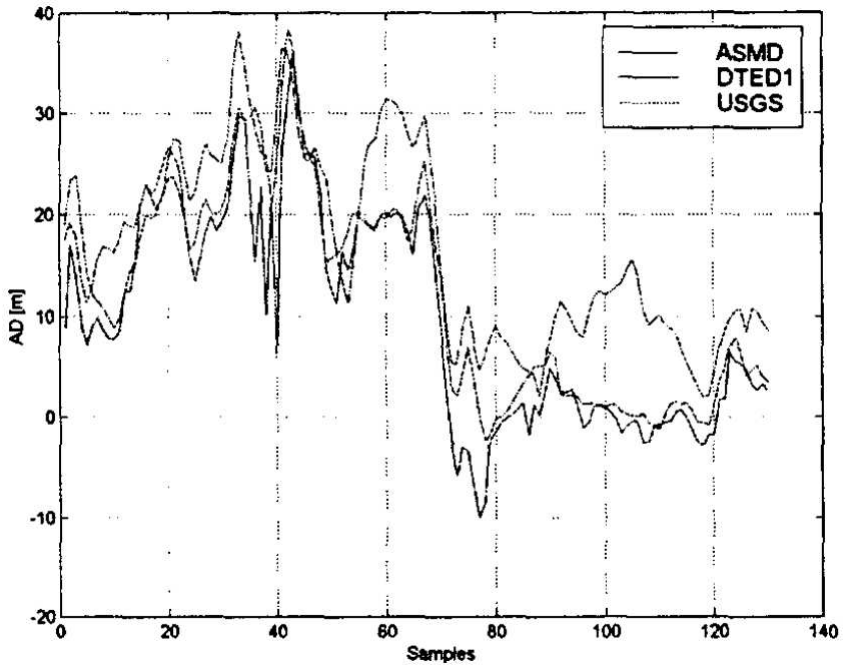



Figure 2.11. Absolute Disparities Approaching AVL Runway 34

Figure 2.13 [1] shows the results from computing the Mean-Square Differences of the absolute and successive disparities ($T$ and $Z$, respectively) for the approach to Runways 16 and 34. The top graph shows the $T$ test statistic for two approaches to Runway 16, one approach to Runway 34, and the holding pattern. Note here that $Th = T_D = 96$, the absolute disparity threshold. The figure shows a violation of the $T_D$ threshold value during the approach to Runway 16. However, the other approaches remain within the threshold value in spite of the biases observed in Figures 2.11 and 2.12 [1]. The bottom graph shows the $Z$ test statistic for the same flight segments. Note that the graph actually uses $\log Z$, so $Th = \log Z_D = \log 2.2 \approx 0.34$. None of the flight segments deviated above the successive disparity threshold, which shows the insensitivity of the $Z$ statistic to biases. This makes the $Z$ test statistic unsuitable for use in a terrain database integrity monitor [1].

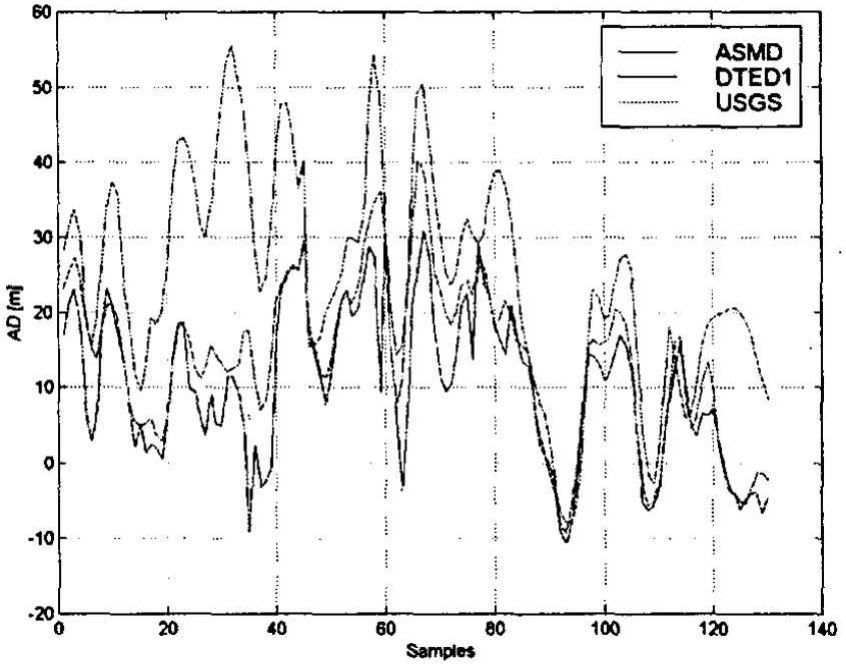



Figure 2.12. Absolute Disparities Approaching AVL Runway 16

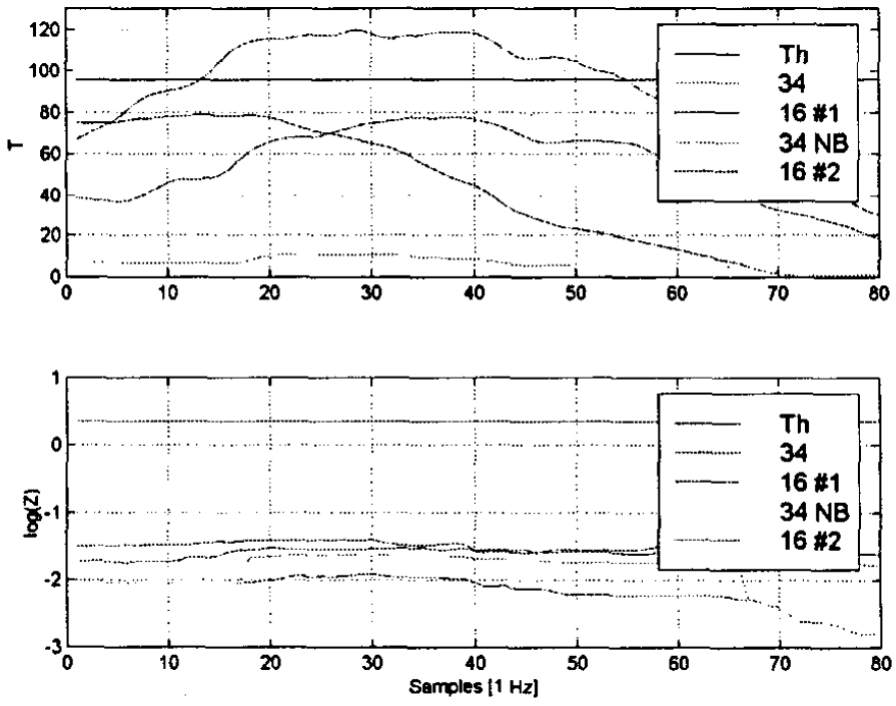



Figure 2.13. $T$ and $\log Z$ Statistics Approaching AVL Runways 16 and 34

Figure 2.14 [1] shows a comparison between the synthesized terrain height and the DTED terrain database height during the approach to Runway 16 that triggered an integrity monitor alert, i.e. where the $T_D$ threshold was exceeded. The visible bias in height is not constant, but still tends to follow changes in terrain elevation.

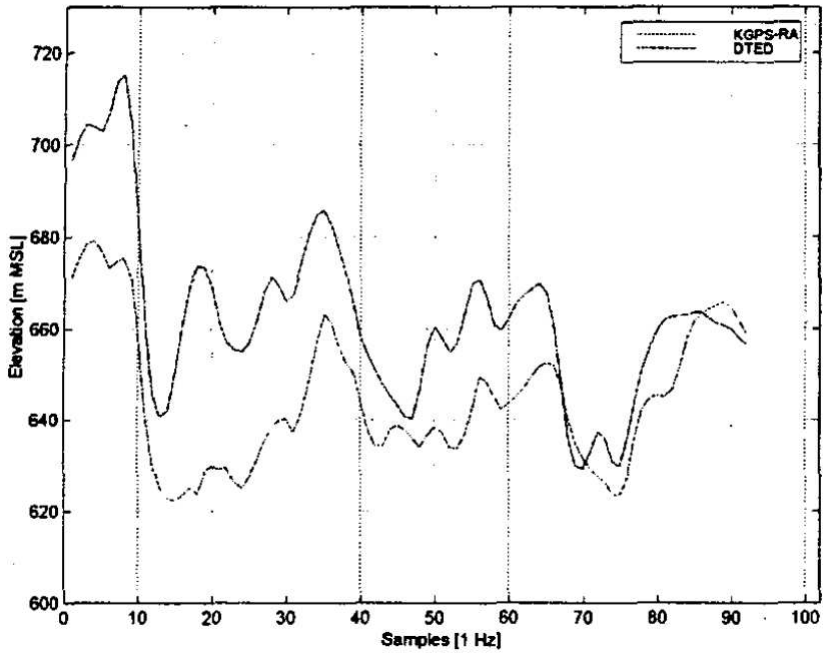



Figure 2.14. Synthesized Height vs. DTED Approaching AVL Runway 16

As a point of comparison, Figure 2.15 [1] shows the terrain height from a more accurate NGS database. This data uses 4 meter post-spacing and 1 meter vertical accuracy of 90%, resulting in better correlation with the same synthesized terrain data. However, a bias is still noticeable. Possible sources for the discrepancy include improper characterization of the radar altimeter performance and the impact of foliage, water, and strong terrain gradients on the radar signals as well as the terrain database vertical datum [1].

The results of the Asheville flight tests led the research team to focus next on the behavior of the Honeywell AN/APN-171(V) radar altimeter that was used. Equation 2.6 shows the altitude accuracy in feet for that system under standard conditions. Note that the error was a function of altitude (range) and altitude rate (range rate), which means that

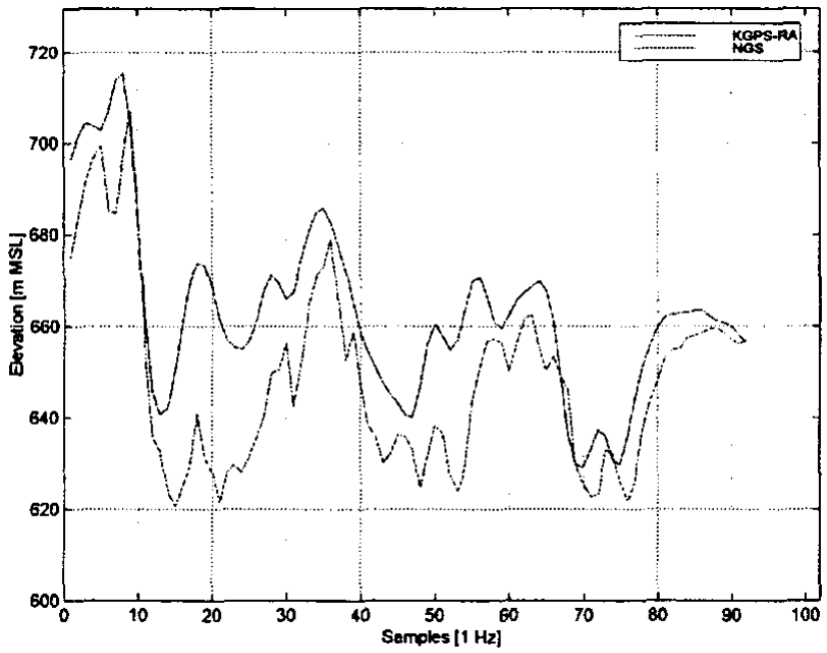



Figure 2.15. Synthesized Height vs. NGS Approaching AVL Runway 16

the system error increased at higher altitudes relative to the terrain, as well as during climb and descent of the aircraft.

$$\begin{aligned} \varepsilon_{radalt} &= f(range, range\ rate) \\ &= 5 + 0.03 \cdot (range) + 0.05 \cdot (range\ rate) \end{aligned} \tag{2.6}$$

This radar altimeter also operated by tracking the leading edge of the returned radar pulse. Figure 2.16 [1] shows examples of how the Honeywell unit would have sensed terrain during level flight and during roll conditions. Point A represents the terrain height that would actually be sensed during level flight (left diagram), while Point B represents the terrain height predicted by the integrity monitor algorithm used in this study. Likewise, Point D represents the terrain height that would have been measured by a leading edge tracking radar altimeter (right diagram) [1]. Although not identified in the results, Point C probably represents the presence of water at the predicted coordinates immediately below the aircraft. Since the test statistics used in this study were based on the terrain height of

the point directly below the aircraft, additional errors may have been introduced into the absolute disparity calculations.

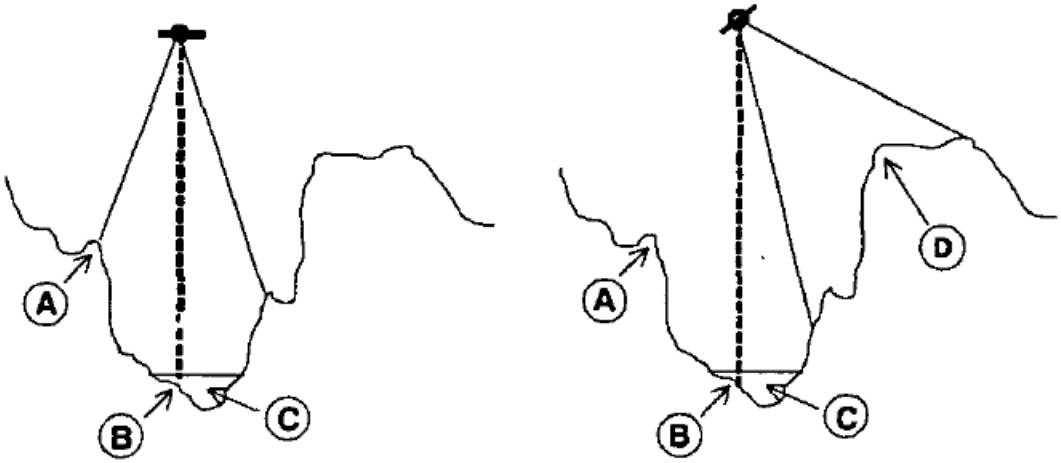



Figure 2.16. Radar Altimeter with Leading Edge Tracking

A second algorithm was designed to better model the behavior of the Honeywell radar altimeter. This algorithm, called the "spot" algorithm, was based on sampling a greater number of terrain points within the radar altimeter beam as shown in Figure 2.17 [1]. Next, the range from the aircraft to each illuminated point was determined and the smallest of those ranges was used for comparison with the radar altimeter measurements [1].

The "spot" algorithm showed only slight improvements when used with the DTED terrain database. However, the improvement was more significant when applied to the NGS data. Figure 2.18 shows a comparison of the synthesized terrain profiles generated using the "spot" algorithm and the original algorithm. The terrain profile generated using the "spot" algorithm correlates better with the NGS terrain data as shown in Figure 2.19. Note that we created Figure 2.19 by merging Figures 2.15 and 2.18 since those results were not presented in a single graph. The darkest line is the synthesized height data derived from the original algorithm. The fainter lines represent the synthesized height data derived using the "spot" algorithm and the NGS terrain data, which is the lower of the two. Although this is

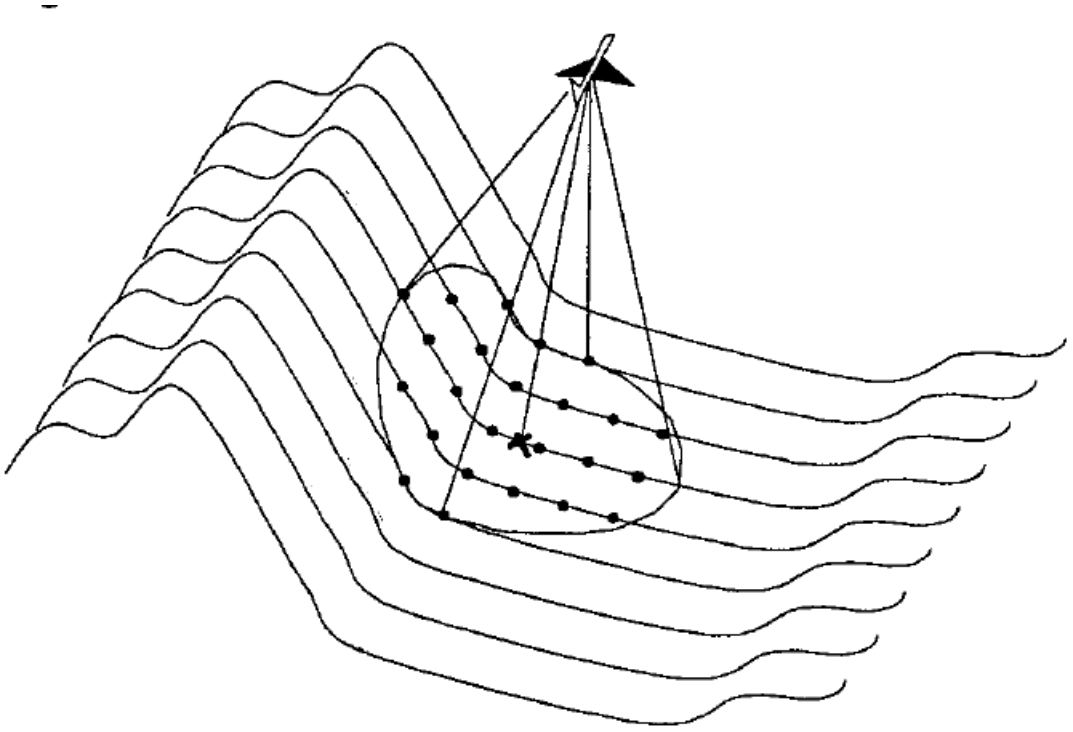


Figure 2.17. Terrain Detection Using "Spot" Algorithm

not a completely accurate depiction, the higher correlation between data from the improved algorithm and the NGS database is quite clear.

Furthermore, Figure 2.20 shows a comparison of the $T$ test statistic (MSD of absolute disparities) for the approaches to Runway 16. The $T_D$ threshold of 96 was still exceeded using the "spot" algorithm (DTED+ T), but shows improvement when compared to the original algorithm used with the DTED data (DTED T). The greatest improvement can be seen when the "spot" algorithm was used with the NGS data (NGS+ T). Although the original algorithm used with the NGS data also remained well within the $T_D$ threshold, the "spot" algorithm provided the smallest disparities (i.e. lower $T$ values) when comparing synthesized heights with terrain database heights.

The Asheville flight test results showed that an aircraft's radar altimeter could be used effectively as part of a terrain database integrity monitor. By tracking disparities between a synthesized terrain profile and the digital profile stored in a terrain database the accuracy of the terrain data could be verified against a statistical threshold. It was also shown that

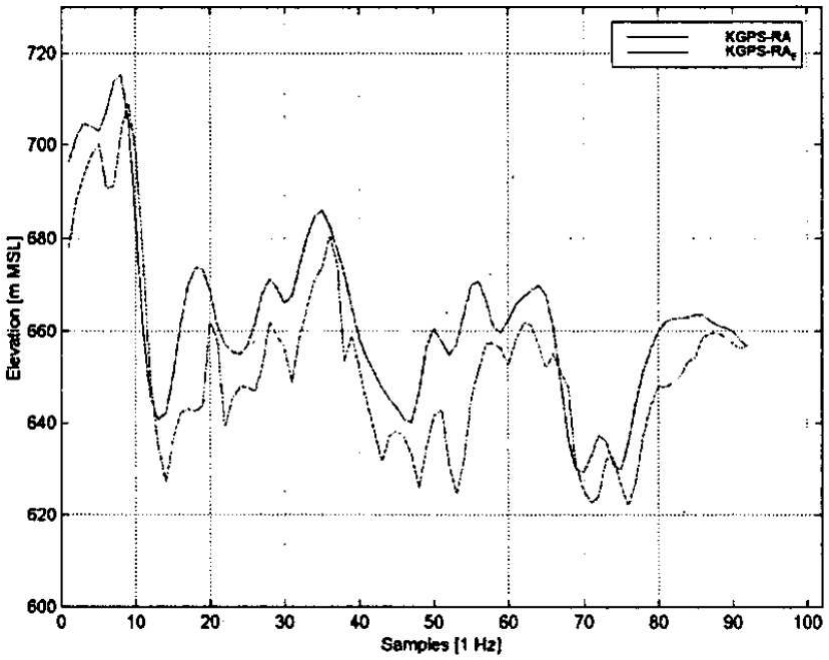



Figure 2.18. "Spot" Height vs. Original Algorithm Approaching AVL Runway 16

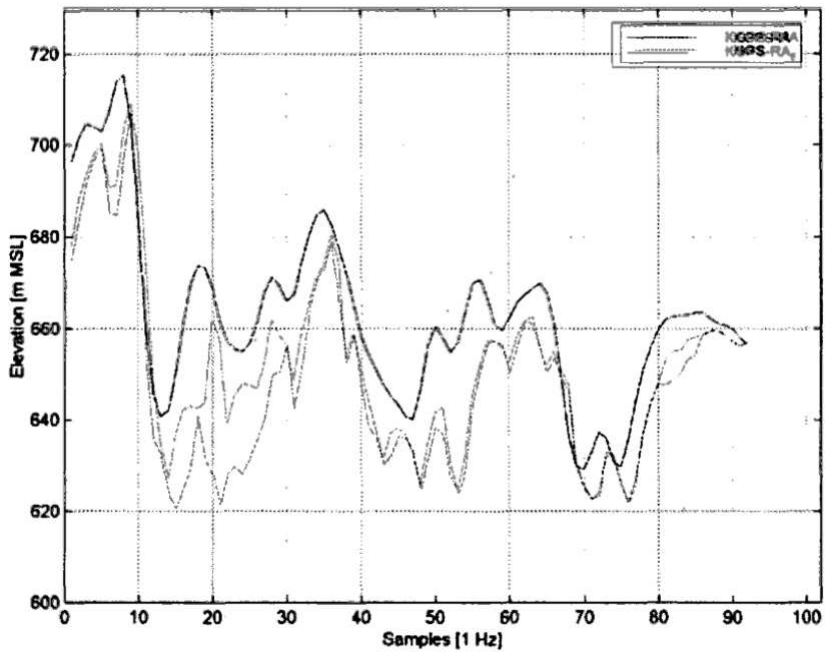



Figure 2.19. "Spot" Height vs. NGS Approaching AVL Runway 16

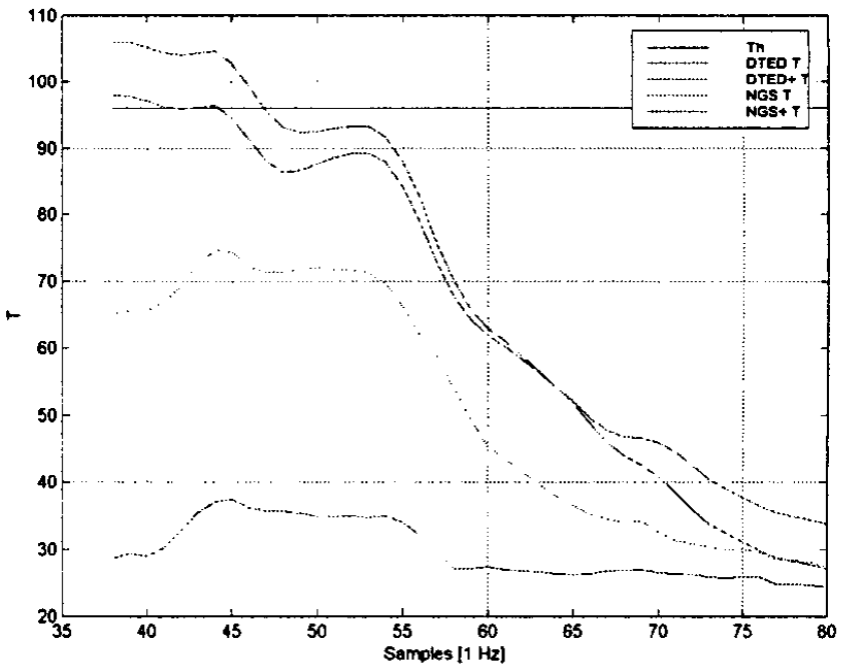



Figure 2.20. $T$ Statistic Comparison for Runway 16 Approaches

biases could exist in both the terrain database and radar altimeter which could skew the results. Finally, it was shown that proper modeling of the radar altimeter's operation was essential to obtaining an accurate profile of the synthesized terrain heights.

The Ohio University research team continued testing their integrity monitor concept in 2000, this time using augmented GPS as the navigation source during test flights at Ohio University airport (UNI) in August and at Asheville, NC (AVL) in September of that year. A prototype of the Local Area Augmentation System (LAAS) was used to support the augmented GPS. DTED level 0 and level 1 digital elevation models (DEMs) were used to compare the effects of database resolution and accuracy on integrity monitor performance [25].

Two hypotheses were identified that define the probability density functions for the absolute disparity under normal and fault conditions. Equation 2.7 defines the $H_0$ hypothesis for fault-free or nominal performance of the integrity monitor [25]. Under fault free conditions nominal errors in the radar altimeter performance and the digital elevation database cause the absolute disparity $p$ to have a normal error distribution with a standard deviation of

$\sigma_p$ [17]. The standard deviation $\sigma_p$ is derived from the standard deviations of the individual sensor errors and the error characteristics of the terrain database being used.

$$H_0 : p \sim N(0, {\sigma^2}_p) \tag{2.7}$$

Equation 2.8 defines an alternative hypothesis $H_1$ where a fault condition exists in the integrity monitor. In this hypothesis the fault is in the form of a bias error in the sensor performance and/or the DEM data. In the fault case the normal distribution has a mean bias $\mu_B$.

$$H_1 : p \sim N(\mu_B, {\sigma^2}_p) \tag{2.8}$$

Again, the integrity monitor test statistic $T$ from Equation 2.4 was used. Under the fault free hypothesis $H_0$ the probability density function (PDF) of $T$ was found to be a $\chi^2$ distribution with $N$ degrees of freedom. Under the $H_1$ hypothesis the PDF was found to be a non-central $\chi^2$ distribution with $N$ degrees of freedom and non-centrality parameter $\lambda$ [17]. Equation 2.9 shows how the non-centrality parameter $\lambda$ relates to the mean bias $\mu_B$ during the fault condition [25].

$$\lambda = \frac{N}{{\sigma^2}_p} {\mu^2}_B \tag{2.9}$$

Figure 2.21 [25] shows how the probability density functions under the $H_0$ and $H_1$ hypotheses were used to determine the test statistic threshold $T_D$. The y-axis represents probability density $p$ and the x-axis represents the test statistic $T$, which is the mean square difference of absolute disparities. The gray area in the graph represents the probability of missed detection $P_{MD}$ under the $H_1$ hypothesis, while the black area represents the probability of fault free detection $P_{FFD}$ or false alarm under the $H_0$ hypothesis. Given an integration time of 50 seconds ($N = 50$) and a probability of false alarm of 0.0001 ($P_{FFD} = 10^{-4}$), the threshold value $T_D = 96$ was calculated from $H_0$. Using the calculated value of $T_D$ and given a probability of missed detection $P_{MD}$, a minimum detectable bias (MDB) can be derived from the $H_1$ hypothesis. For example, given $P_{MD} = 10^{-7}$ a MDB value of 33.96 meters re-

sults [25]. In this case, any bias larger than 33.96 meters will be detected with a probability of $1 - P_{MD} = 1 - 10^{-7}$ or 0.9999999 [12].

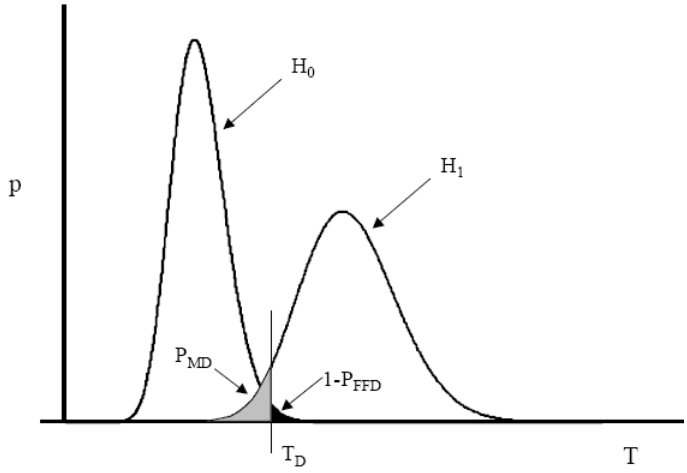



Figure 2.21. $T_D$ Threshold Determined Using $H_0$ and $H_1$ Hypotheses

Figure 2.22 shows the hardware layout of Ohio University's real-time SVS prototype that was used for the UNI flight tests. The LAAS unit provided differential corrections to the GPS position. This information was received from a prototype LAAS ground station. A Honeywell inertial reference unit (IRU) provided attitude, altitude, and airspeed data to the main processor through an interface unit. A Honeywell radar altimeter provided terrain measurements, which were sent to the main processor via a microcontroller in the interface unit. Aircraft state and terrain information was then sent from the main processor to two flat panel liquid crystal displays that were provided by Delft University of Technology [12].

A QNX real-time operating system (RTOS) was used to run the integrity monitor software on the main processor. Figure 2.23 shows the flow of data between the software components. The time handler provided support for synchronization of messages. The parity handler computed the synthesized terrain height, absolute disparities, and test statistic from the measured terrain heights and digital terrain height provided by the radar, LAAS, and DTED handlers respectively. The parity handler compared the test statistic $T$ against the pre-computed integrity threshold $T_D = 96$ and set a flag to the display handler when an integrity alarm was triggered. The display handler provided aircraft state information,

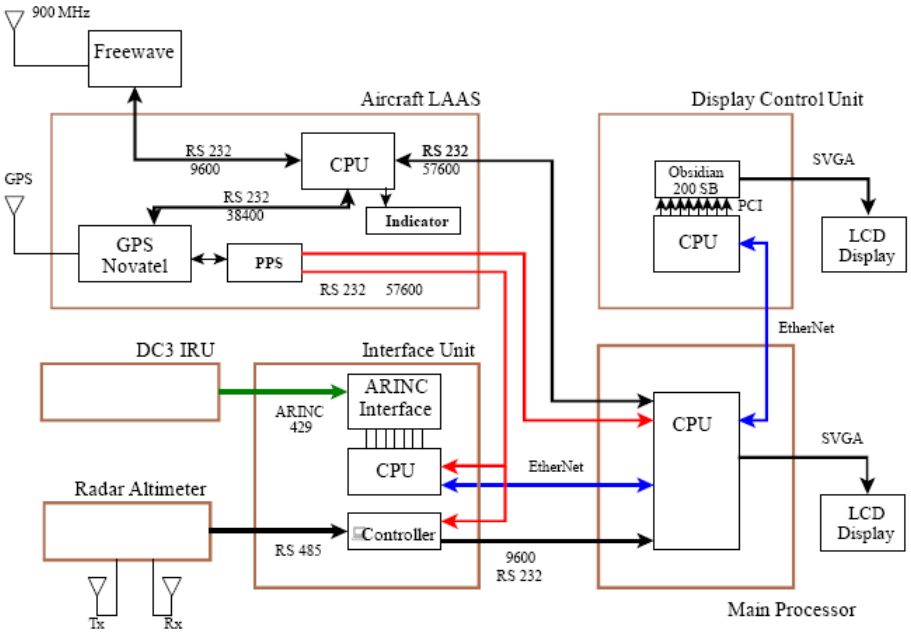



Figure 2.22. SVS Prototype Hardware

e.g. attitude, heading, position, integrity warning, to the SVS display via a TCP/IP connection [12].

Figure 2.24 shows the approach patterns flown to Runways 7 and 25 at Ohio University airport (UNI). A total of 14 approaches were performed using Ohio University's DC-3 aircraft. Two approaches were made to Runway 7 and twelve approaches to Runway 25. The approach to Runway 7 required a turn just prior to final approach, which resulted in curved groundtracks in that graph. The bank angle of the aircraft during those turns caused the radar altimeter to detect terrain not immediately below the aircraft (see Figure 2.16), which introduced additional errors into the test statistic. Runway 25 had a straight approach, so those test flights were not impacted by this type of error.

Flight test results for the approaches to Runways 7 and 25 are shown in Figures 2.25 and 2.26. The graphs directly compare the test statistic $T$ when the DTED level 0 and level 1 terrain data were used. Both databases have similar horizontal and vertical accuracy specifications. However, DTED level 1 has post-spacing of 3 arc seconds, while DTED level 0 has a "lower resolution" post-spacing of 30 arc seconds. The effect of the higher resolution

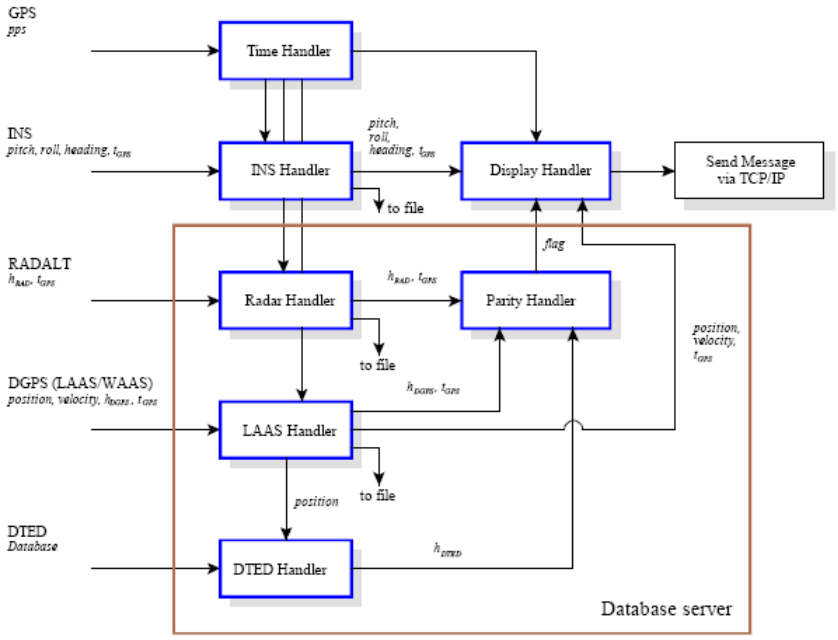



Figure 2.23. SVS Prototype Software Flowchart

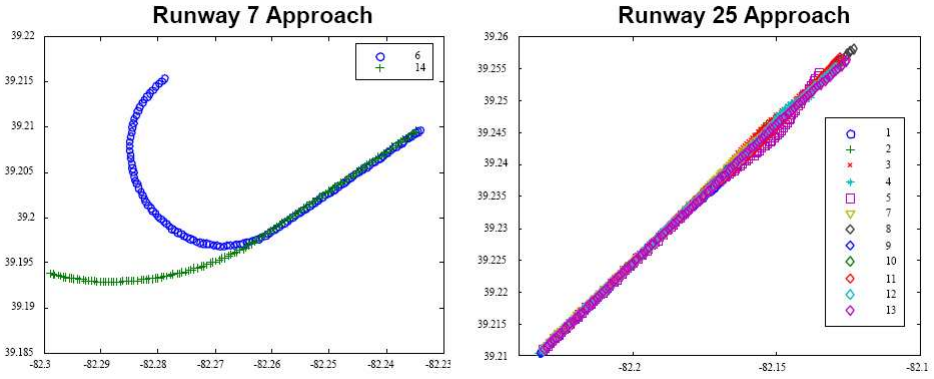



Figure 2.24. Approach Patterns to UNI Runways 7 and 25

post-spacing in the DTED level 1 terrain data can be observed as lower $T$ values during those approaches. The disparity is primarily due to greater errors in heights interpolated from the lower spatial resolution data in DTED level 0 [25]. The $T_D$ integrity threshold violation shown in Figure 2.25 (left) may also be due to the curved approach to Runway 7 and the resulting radar altimeter error.

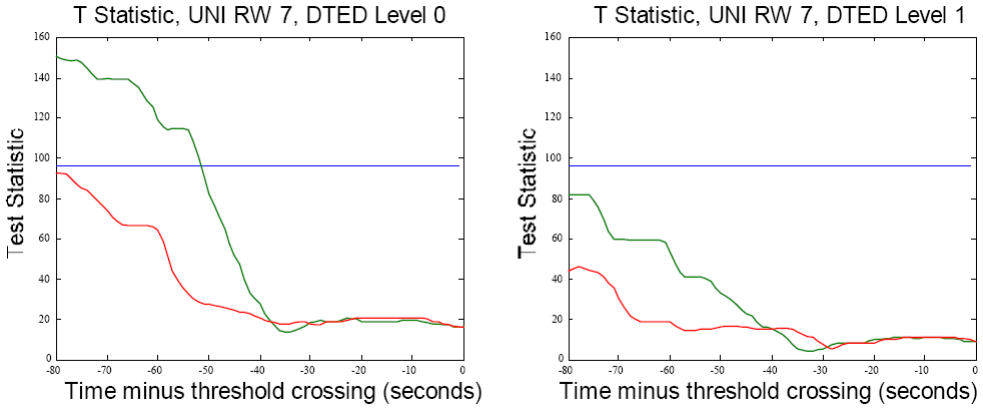



Figure 2.25. DTED Level 0 vs. Level 1 Approaching UNI Runway 7

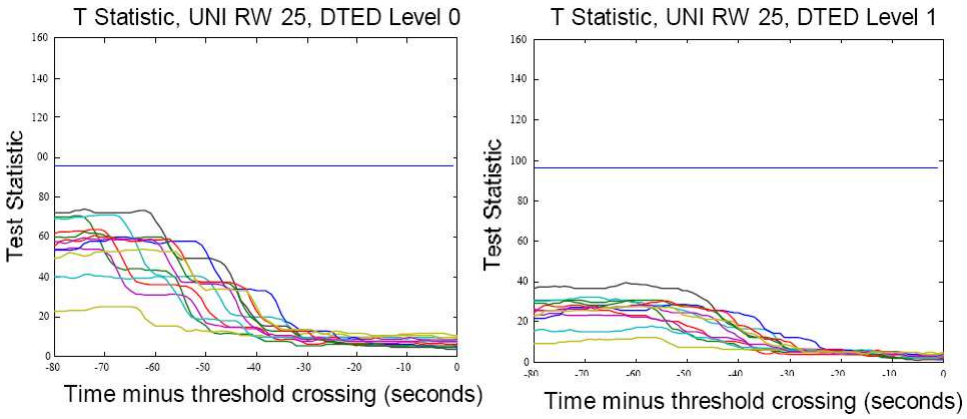



Figure 2.26. DTED Level 0 vs. Level 1 Approaching UNI Runway 25

The second set of flight tests was conducted at Asheville, NC (AVL). Again, the Ohio University DC-3 with the SVS prototype integrity monitor was used. The groundtracks for the approaches to Runway 16 and 34 are shown in Figure 2.27. A total of 14 approaches

was flown, 7 each to Runway 16 and Runway 34. As mentioned previously, the terrain characteristics are different for the two approaches. The terrain profile for the Runway 34 approach is characterized by large variations on initial approach, but much smaller variations on final approach, whereas the Runway 16 approach has significant terrain variations all the way up to the runway [25].

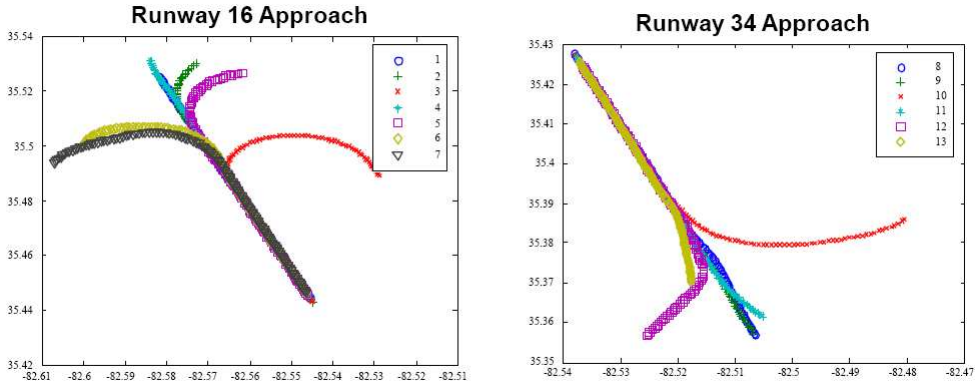



Figure 2.27. Approach Patterns to AVL Runways 16 and 34

Comparisons of the test statistic $T$ for both runways and both levels of DTED are shown in Figures 2.28 and 2.29. As with the UNI flight tests, no integrity violations occurred when DTED level 1 terrain data was used. However, the increased interpolation error from the lower resolution DTED level 0 data caused an integrity violation on both approaches.

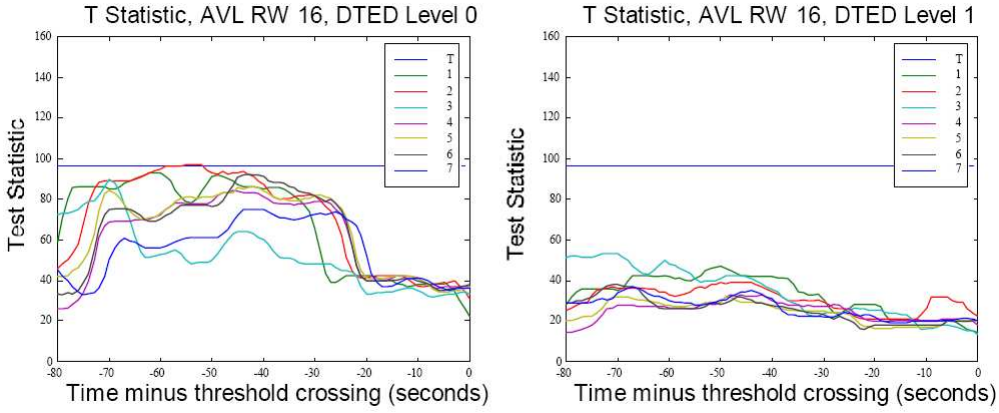



Figure 2.28. DTED Level 0 vs. Level 1 Approaching AVL Runway 16

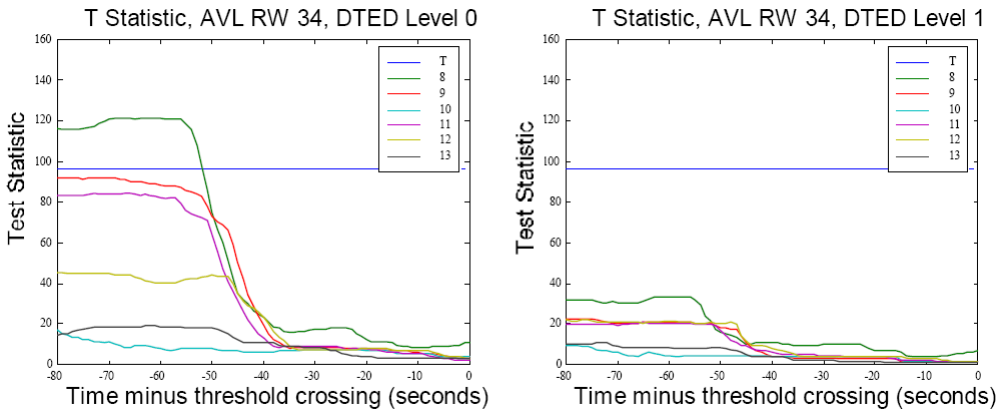



Figure 2.29. DTED Level 0 vs. Level 1 Approaching AVL Runway 34

Overall, the UNI and AVL flight test results showed that the performance of the prototype integrity monitor was significantly improved when higher spatial resolution terrain data was used. It was also shown that lower resolution data, such as that in DTED level 0, could actually cause integrity violations. Although terrain data resolution was a primary factor in integrity monitor performance, the effects of improper modeling of the radar altimeter operation also contributed to the system error.

Another research study [18] was performed at Ohio University to characterize the behavior of different types of radar altimeters. The specifications for two of the radar altimeters used during the previous flight tests are shown in Table 2.2. A NASA 757 radar altimeter was also analyzed for this study, but flight test data was not available for that system. The radar altimeters fall into two general categories of operation, FM continuous wave (FM-CW) modulation and pulse modulation. These systems typically operate in the C radar frequency band with a center frequency of 4.3 GHz. The operating range is from 0 feet to between 2500 and 5000 feet above ground level (AGL). The altitude accuracy is a function of several variables including range, range rate, and noise [18].

Pulse radar altimeters operate by transmitting a pulse down toward the terrain and tracking the time of its reflected return. Equation 2.10 shows how the aircraft height above ground level is determined from $\tau$, the pulse transit time, and $c$, the speed of light. It can be seen that the height is linearly related to the pulse transit time $\tau$. Both the Honeywell

Table 2.2. Radar Altimeter Specifications

| Type | Altitude Accuracy | Modulation | PRF | Beam Width 3 dB |
|---|---|---|---|---|
| Honeywell HG8505DA01 DC-3 Aircraft | ±3 ft, plus 1% | Pulse | 25 kHz | 17 ° |
| Honeywell HG9050D2 TIFS Aircraft | ±5 ft, plus 3% of range, plus 5% of average range rate (ft/s) | Pulse | 10 kHz | 35 ° |
| NASA 757 | 1.0 ft or 2% of range, whichever is greater | FM-CW | N/A | 90 ° |

HG8505DA01 and HG9050D2 pulse radar altimeters used in previous flight tests track the leading edge of the return pulse. This means that the shortest slant range to the terrain is used for determining the aircraft height (refer to Figure 2.16) [18].

$$h_{AGL} = \frac{c\tau}{2} \tag{2.10}$$

Figure 2.30 [18] shows how the pulse transit time $\tau$ is determined. A tracking pulse is generated just prior to the estimated return time of the transmitted pulse. This pulse is then multiplied with the ground return pulse as shown in the figure. The darkened overlapping area relates to the range rate, which is represented by a current $i$. This current is then integrated along with a fixed offset current equal to the normal track current. The result determines the necessary voltage to apply to a range integrator, which then outputs a range value that is used as feedback for adjusting the position of the tracking gate. The time

between the leading edge of the transmit pulse and the tracking gate determines the pulse transit time $\tau$ [18].

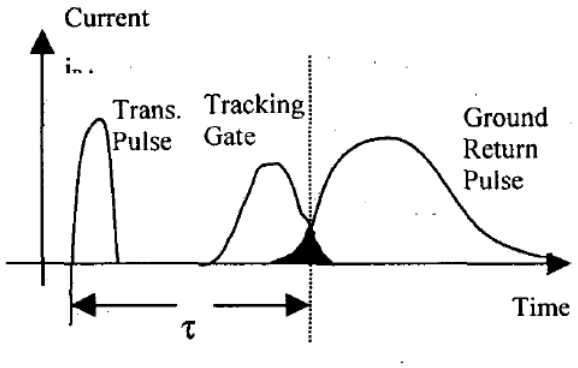



Figure 2.30. Pulse Radar Altimeter Tracking

There is a specified range rate limit of 2000 ft/sec for the radar altimeters that were tested due to the required range and range rate integrations used to position the tracking gate. There is also a range rate error of 5% of the average range rate in ft/sec due to the delay in the feedback loop [18].

FM-CW radar altimeters are much older technology than the pulse modulation systems, but are still commonly used in commercial aircraft. These systems operate by transmitting an FM modulated carrier signal to the ground, mixing the transmitted signal with the reflected signal from the terrain, and measuring the resulting beat-frequency. Figure 2.31 shows a simplified block diagram of an FM-CW radar altimeter [18].

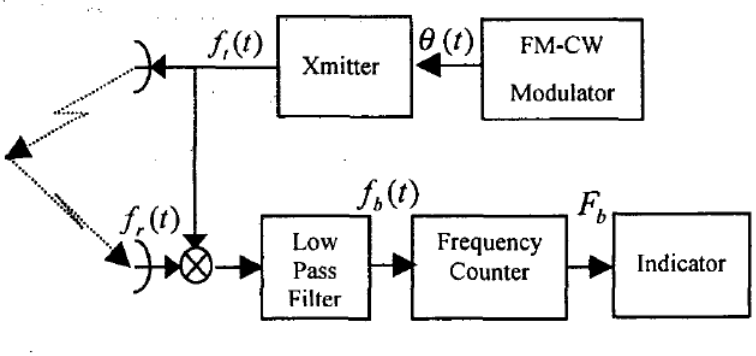



Figure 2.31. FM-CW Radar Altimeter Block Diagram

Equation 2.11 shows that the transmitted frequency $f_t$ is a function of signal amplitude $A$, a carrier frequency $f_0$, and a triangle waveform $\theta$ generated by the FM-CW Modulator. Figure 2.32 shows an example of a triangle waveform $\theta$, where $F_d$ is the magnitude of the frequency deviation and $t_{FM}$ is the period of the waveform [18].

$$f_t(t) = A\cos(2\pi f_0 + 2\pi\theta(t)) \tag{2.11}$$

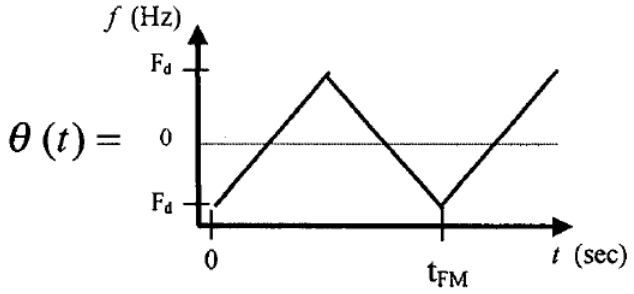



Figure 2.32. FM-CW Radar Altimeter Triangle Wave Form

Equation 2.12 shows the received frequency $f_r$ is also a function of the carrier frequency $f_0$ and the triangle waveform $\theta$. However, it is also dependent on the term $\frac{2h}{c}$, which represents the time required for the transmitted signal to reflect off the ground and return to the radar altimeter. As in Equation 2.10, $h$ represents the aircraft height above ground level and $c$ represents the speed of light [18].

$$f_r(t) = cos(2\pi f_0 + 2\pi\theta(t + \frac{2h}{c})) \tag{2.12}$$

The transmitted frequency $f_t$ and the received frequency $f_r$ are then mixed to generate a beat frequency. Equation 2.13 shows the results of mixing the two signals [18].

$$f_t(t) \otimes f_r(t) = \cos\left[4\pi f_0 + 2\pi(\theta(t) + \theta(t + \frac{2h}{c}))\right] + \cos\left[2\pi(\theta(t) - \theta(t + \frac{2h}{c}))\right] \tag{2.13}$$

The mixed signal $f_t \otimes f_r$ then goes through a low-pass filter to extract the beat frequency. The beat frequency $F_b$ is represented by the terms $\theta(t) - \theta(t + \frac{2h}{c})$ in Equation 2.14. The signal is then fed to a frequency counter to determine the beat frequency. Equation 2.15 shows how the beat frequency $F_b$ can be represented in terms of $F_d$, $t_{FM}$, and $\frac{2h}{c}$, when the slope of the modulated signal is constant [18].

$$f_b(t) = \cos\left[2\pi(\theta(t) - \theta(t + \frac{2h}{c}))\right] \tag{2.14}$$

$$F_b = \frac{2F_d}{0.5t_{FM}}\frac{2h}{c} \tag{2.15}$$

The aircraft height is determined by solving Equation 2.15 in terms of $h$. Equation 2.16 shows the results. It can be seen that the height $h$ is linearly related to the beat frequency $F_b$. The frequency counter in an FM-CW radar altimeter will receive multiple beat frequencies depending on the variance of the terrain and the slant ranges of the returned signals. The heights represented by the various beat frequencies are placed into separate range bins and the radar altitude $h$ is determined by the bin with the most ranges [18].

$$h = \frac{t_{FM}cF_b}{8F_d} \tag{2.16}$$

The triple redundant radar altimeter systems on the NASA 757 ARIES aircraft have a frequency deviation $F_d$ of $\pm$ 50 MHz centered on a 4.3 GHz carrier frequency $f_0$. The three radar altimeters operate at different FM modulation frequencies to avoid interfering with each other [18].

The research study went on to model the behavior of the pulse and FM-CW radar altimeters based on the previously described specifications. Simulations were performed using MATLAB® to model the system performance over uneven terrain. The "spot" method of determining the synthesized terrain profile was used. The radar altimeter beam coverage area, i.e. spot size, was estimated using the height above ground level and the beamwidth

for each unit under test. Equally spaced height samples within the coverage area were taken from the terrain database to represent the synthesized terrain profile. Slant ranges were then calculated from the aircraft position to the terrain height samples. The effects of aircraft roll and pitch on the spot position and size were not simulated for this study [18].

For the pulse radar altimeter simulations, the minimum slant range to the synthesized terrain sample points was used to represent the height determined by the leading edge of the return pulse. Slant ranges were not converted into pulse transmit times $\tau$ due to the linear relationship between height and time identified in Equation 2.10. Effects of propagation loss and attenuation on the slant ranges were not simulated in this study mostly because those factors had less of an impact on the shorter ranges used to determine altitude in a pulse-modulated system [18].

For the FM-CW radar altimeter simulations, the spot size and range determination was performed in a similar manner to the pulse radar simulations. However, the slant ranges were not transformed into the beat frequency domain due to the linear relationship between height and beat frequency identified in Equation 2.16 [18].

The effects of radar altimeter beamwidth on terrain sensing were then simulated using DTED level 1 databases of the terrain for the vicinity of Asheville, NC (AVL) and Ohio University (UNI) airports. These areas were used for previous flight tests of Ohio University's prototype terrain database integrity monitor. The DC-3 and TIFS aircraft that were flown used pulse radar altimeters that operated similarly except for the radar beamwidth (refer to Table 2.2). The impact of beamwidth on terrain sensing is a function of terrain roughness and aircraft altitude. Monte Carlo simulations were run in MATLAB® for one-thousand randomly selected cells (1° latitude by 1° longitude) to determine the effects of different beamwidths on radar altitude [18].

Figure 2.33 shows the root mean squared (RMS) error between the minimum slant range height and the plumb-bob height below the aircraft for the terrain surrounding AVL. The plumb-bob height represents the distance to the terrain directly below the aircraft, while the minimum slant range height represents the altitude tracked with the leading edge of

the return pulse. The terrain roughness $\sigma_T$ was calculated to be 247 meters from the AVL terrain data [18].

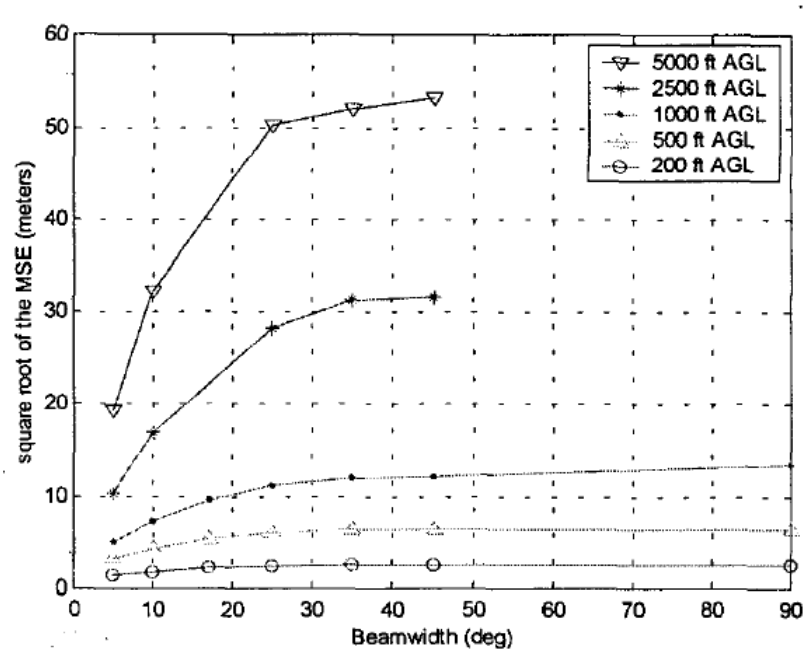



Figure 2.33. RMS Error at AVL

Figure 2.34 shows the effects of beamwidth on the RMS error for the terrain surrounding UNI. The terrain roughness $\sigma_T$ was calculated to be 31 meters in this case. These simulation results show that the radar altimeter beamwidth has a significant impact on the sensed terrain profile error as the beamwidth gets larger, the aircraft altitude increases, and the terrain roughness $\sigma_t$ increases [18].

Next, the research study compared the performance of the two pulse radar altimeters using AVL approach data gathered during previous flight tests. The NASA TIFS aircraft used the Honeywell HG9050D2 radar altimeter with a 3dB beamwidth of 35°, while the Ohio University DC-3 used the Honeywell HG8505DA01 system with a 3db beamwidth of 17°. Figure 2.35 shows a side view of two similar ILS approaches into Asheville airport and the terrain profile from the database. The database used in these evaluations was generated using photogrammetry data, which is much more accurate than the DTED level 1 data used in the original flight tests [18].

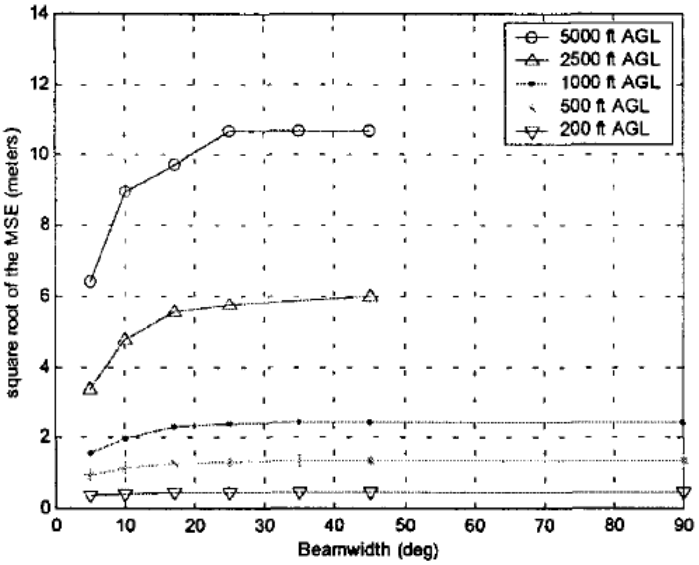



Figure 2.34. RMS Error at UNI

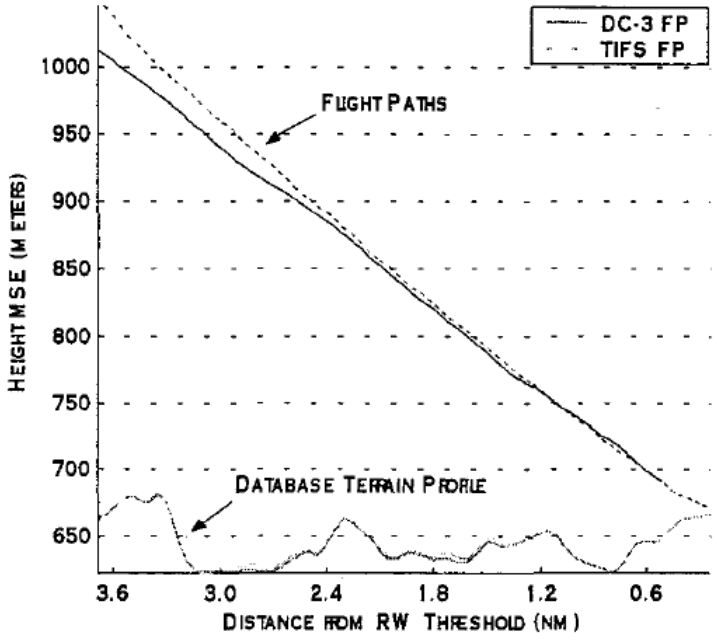



Figure 2.35. Comparison of Approaches to AVL

Figure 2.36 shows a comparison of the synthesized terrain profiles generated by each pulse radar altimeter during the flight tests. The solid lines represent the 'plumb-bob' terrain directly below the aircraft and the dotted lines represent the sensed terrain. It is useful to note the differences in the sensed terrain profiles between the two pulse radar altimeter systems.

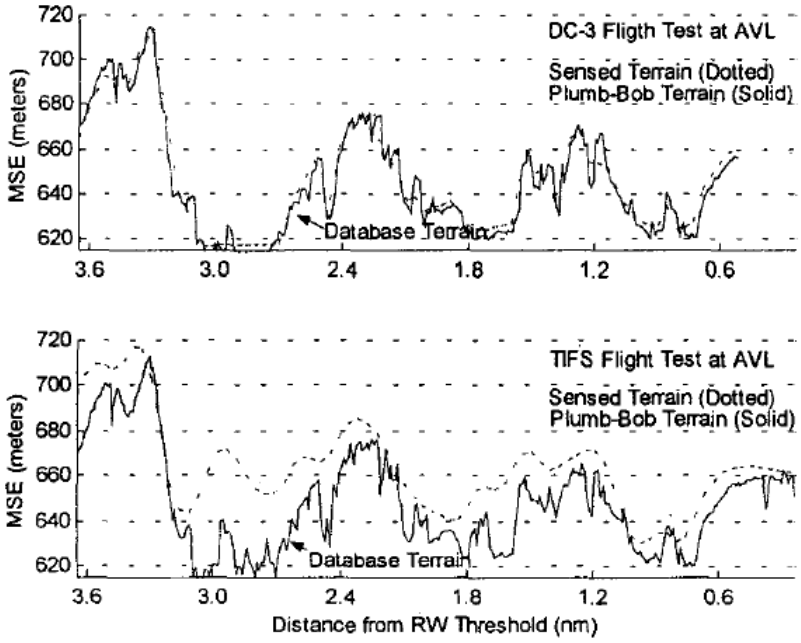



Figure 2.36. AVL Terrain Disparity Using 'Plumb-bob' Terrain Model

Figure 2.37 shows a similar comparison of the synthesized terrain profiles generated by each radar altimeter. However, in this case the solid line represents the modeled terrain below the aircraft, calculated using the "spot" method for each altimeter's beamwidth. The resulting terrain model correlates much closer to the actual profiles sensed by the radar altimeters during the flight tests. The results of these evaluations again show the importance of proper modeling of the radar altimeter behavior. The impact of beamwidth, altitude, and roughness of terrain must be considered when designing a terrain database integrity monitor that uses a radar altimeter for synthesized terrain profile generation [18].

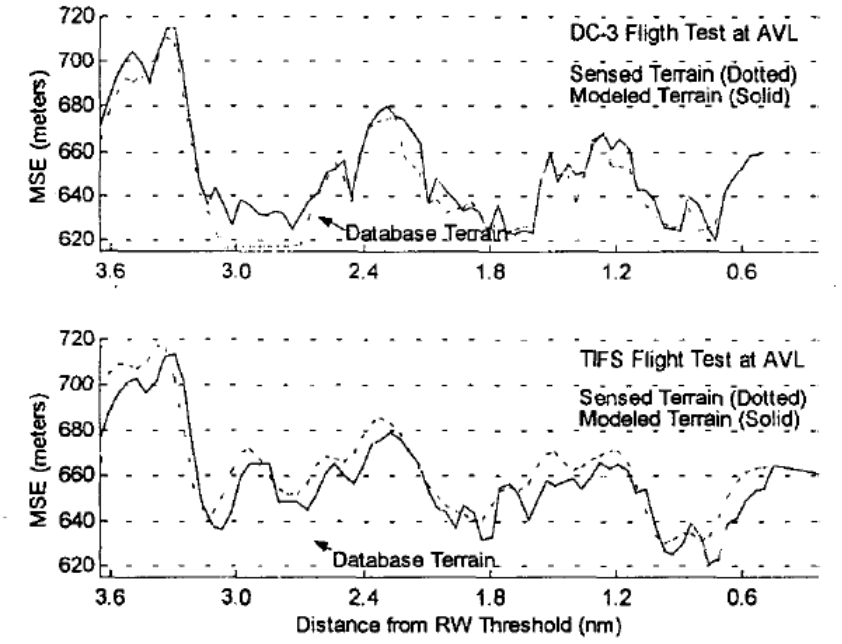



Figure 2.37. AVL Terrain Disparity Using 'Spot' Terrain Model

With the avionics background established, we now turn our attention to how stream computing can be used on a GPU to perform the tasks demanded of a terrain database integrity monitor.

## 2.2 GPU Stream Computing

The computing power of GPUs has increased more rapidly than that of CPUs in recent years [26]. Furthermore, graphics hardware has evolved from a fixed-function pipeline to programmable vertex and fragment processors. Although GPU technology has evolved rapidly, it is still very difficult to program vertex and fragment shaders. The introduction of shader languages such as Cg by NVIDIA and HLSL by Microsoft has helped to alleviate the problem to some extent; however, support for streaming applications on graphics hardware has been relatively weak. In response to this need, a team of researchers at Stanford University has worked to adapt the Brook system for general-purpose stream computation

to commercially available programmable graphics hardware, such as that manufactured by ATI and NVIDIA [27].

### 2.2.1 Brook for GPUs

Brook for GPUs extends C to include simple data-parallel constructs, which allows the GPU to be used as a streaming co-processor. It consists of a compiler and run-time system, which abstracts many of the low-level details of graphics hardware programming, thus making it easier for programmers to take advantage of the GPU's stream computing capabilities. The Brook programming environment provides native support for streams, which exploits data parallelism in applications. Arithmetic intensity is improved by performing computations in kernels, which exploit locality through the use of local registers, thus reducing the bandwidth to memory. Brook has good performance over a wide range of algorithms and it is very portable, mapping to a variety of graphics architectures [27].

The basic features of the Brook programming model include the stream, the kernel, and the reduction. Streams are collections of data, which can be operated on in parallel by the GPU's 4-way SIMD processors. Brook represents streams as floating point textures on graphics hardware and provides support for streams of user-defined structures. Listing 2.1 shows an example of how a stream of rays would be defined for use in a ray-tracing algorithm. Streams are defined using the angle bracket syntax and appear to be similar to arrays, but their access is limited to kernels. The types of streams supported in Brook are Input Streams, Output Streams, and Gather Streams [27].

Listing 2.1. A Stream of Rays

```
typedef struct ray_t {
  float3 o;
  float3 d;
  float tmax;
} Ray;
Ray r<100>;
```

Kernels are the workhorses of the Brook system. They are special functions that perform an implicit loop over the elements of a stream. With stream data stored in textures, Brook executes a kernel function over the stream elements using the GPU's fragment processor. An example of a kernel used to determine the ray-triangle intersections in a streaming ray tracer is shown in Listing 2.2. The kernel is restricted in the type of arguments it can accept in order to allow data-parallel execution without data dependency issues. Kernels can accept the following data types: Input Streams, which contain read-only data, Output Streams, which store the results of kernel computations, Gather Streams, which contain read-only data that can be indexed like an array, and read-only constants.

Listing 2.2. Kernel to Compute Ray-Triangle Intersections

```
kernel void krnIntersectTriangle(Ray ray<>,
                                 Triangle tris[],
                                 RayState oldraystate<>,
                                 GridTrilist trilist[],
                                 out Hit candidatehit<>) {
  float idx, det, inv_det;
  float3 edge1, edge2, pvec, tvec, qvec;
  if(oldraystate.state.y > 0) {
    idx = trilist[oldraystate.state.w].trinum;
    edge1 = tris[idx].v1 - tris[idx].v0;
    edge2 = tris[idx].v2 - tris[idx].v0;
    pvec = cross(ray.d, edge2);
    det = dot(edge1, pvec);
    inv_det = 1.0f/det;
    tvec = ray.o - tris[idx].v0;
    candidatehit.data.y = dot(tvec, pvec) * inv_det;
    qvec = cross(tvec, edge1);
    candidatehit.data.z = dot(ray.d, qvec) * inv_det;
    candidatehit.data.x = dot(edge2, qvec) * inv_det;
    candidatehit.data.w = idx;
    } else {
    candidatehit.data = float4(0,0,0,-1);
    }
}
```

Reductions are operations that provide a data-parallel method for calculating a single value from a set of records. Examples of reductions include arithmetic sum, computing a maximum, and matrix product. Listing 2.3 and Figure 2.38 show an example of a reduction

used to determine the sum in matrix-vector multiplication. The reduction accepts a single input stream and produces either a smaller stream of the same type or a single-element value.

Listing 2.3. Brook Reduction in Matrix Multiplication

```
kernel void mul (float a<>, float b<>, out float c<>) {
  c = a * b;
}

reduce void sum (float a<>, reduce float r<>) {
  r += a;
}

float A<50,50>;
float x<1,50>;
float T<50,50>;
float y<50,1>;
...
mul(A,x,T);
sum(T,y);
```

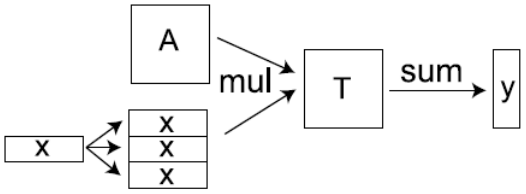



Figure 2.38. Matrix-Vector Multiplication $y = Ax$

The Brook system consists of two components: brcc, which is a source-to-source compiler, and the Brook Runtime (BRT) library. The brcc compiler maps Brook kernels into Cg shaders, which are translated by vendor shader compilers into GPU assembly [27]. This makes it possible to write a stream computing application without specific knowledge of the shader programming language. The Brook Runtime library handles the interface to the OpenGL or DirectX APIs, which also reduces the graphics knowledge burden for the programmer. The Brook Runtime also supports a reference CPU implementation, which is used for performance comparisons. Figure 2.39 shows an overview of the Brook implementation.

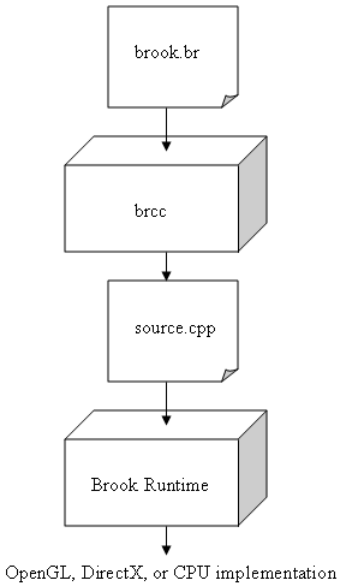
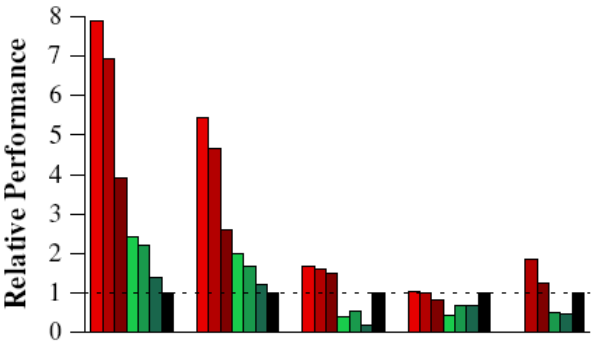



Figure 2.39. Brook Implementation

Figure 2.40 shows the results of Brook performance testing using several types of applications. The applications tested include: SAXPY, a vector scale and sum ($y = ax + y$) algorithm, SGEMV, a matrix-vector product and scaled vector add ($y = \alpha x + \beta y$) algorithm, Segment, a 2-D, nonlinear, diffusion-based, seeded, region-growing algorithm, FFT, a fast Fourier transform algorithm, and Ray, a ray-tracing algorithm. The algorithms were tested using a reference GPU version, a Brook DirectX and OpenGL version, and an optimized CPU version. Results for the ATI hardware are shown in red and NVIDIA in green. The bar graph is normalized by the optimized CPU performance (shown by the dotted line). The table lists the MFLOPS for each application, except for the ray tracer, where the ray-triangle test rate is shown [27].

### 2.2.2 Streaming Ray Tracing

Ray tracing is an effective technique for accurately rendering 3-D imagery, but it is also computationally intensive and therefore difficult to implement in real-time applications. In this technique, a ray of light is traced backwards from the position of the camera or viewer's




| | MFLOPS | | | | RT/sec |
|---|---|---|---|---|---|
| ATI | | | | | |
| Reference | 4923 | 14171 | 2335 | 1278 | - |
| BrookDX | 4324 | 12163 | 2251 | 1212 | 186 |
| BrookGL | 2444 | 6800 | 2086 | 1003 | 125 |
| NVIDIA | | | | | |
| Reference | 1518 | 5200 | 567 | 541 | - |
| BrookDX | 1374 | 4387 | 765 | 814 | 50 |
| BrookGL | 861 | 3152 | 255 | 897 | 45 |
| CPU | 624 | 2616 | 1407 | 1224 | 100 |

Figure 2.40. Brook Test Results

eye in the 3-D coordinate system, through a pixel in the image plane, and into the 3-D scene (Figure 2.41) [28]. If the ray intersects with an object, then additional rays (e.g. shadow ray, reflection ray) can be generated and traced back towards the light source. All of this ray information is then used to determine the color of the pixel in the image plane.

The basic processes of ray tracing include generating rays, traversing the 3-D grid structure, determining ray-triangle intersections, and shading [29]. Shading is the process of generating shadow rays and ultimately determining pixel color. The core of any ray tracer is computing the intersection of rays with the triangles that form the 3-D scene. Each ray can be represented mathematically as $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, where $\mathbf{o}$ is the ray origin, $\mathbf{d}$ is the ray direction, and $t \geq 0$ represents points along the ray [26].

To further improve the performance of ray tracing, a streaming ray tracing model has been proposed [26] to leverage the strength of the graphics processor to perform simple operations on a continuous set of data. Streaming computing differs from traditional computing in that the processor reads the data needed for a computation as a stream of elements [26]. Ray tracing greatly benefits from stream computing techniques due to the highly parallel

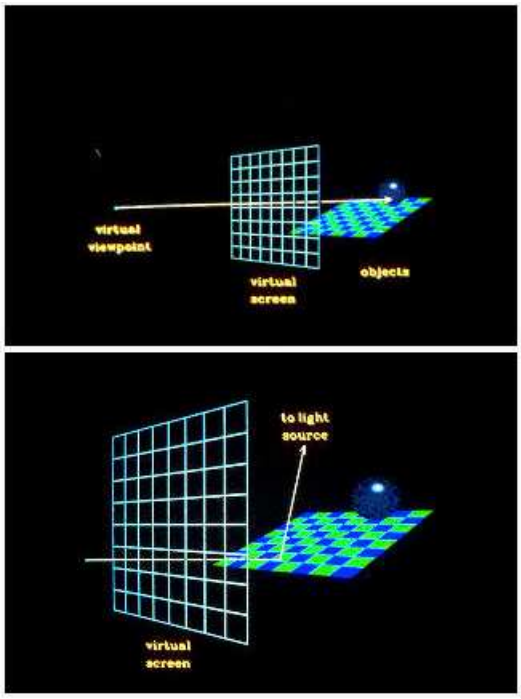



Figure 2.41. Ray Tracing

nature of the problem [29]. The programmable vertex and fragment processors in the GPU each support 4-way SIMD instructions, making it an ideal platform for ray tracing.

Figure 2.42 shows the basic ray tracing processes laid out in the stream-computing model. In this model, the first stage generates a stream of rays; the second stage traverses the grid structure to determine intersections between rays and voxels (i.e. volume elements) and generates a stream of ray-voxel pairs; the third stage then determines the intersections of rays with triangles composing the voxels and generates ray-triangle pairs; finally, the fourth stage determines pixel color or generates a stream of shadow rays to perform additional computations on [26].

Results for a streaming ray tracing model implemented in Brook for GPUs are shown in Figure 2.40. The performance of the ray-tracing algorithm on the ATI hardware is of particular interest. These results clearly show that the Brook stream computing system can outperform the best-optimized CPU solution for ray tracing. The relatively poor perfor-

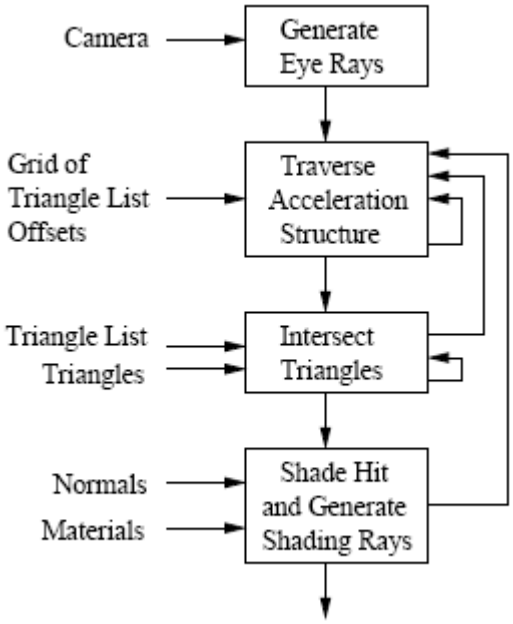



Figure 2.42. Streaming Ray Tracer

mance of the NVIDIA hardware is explained by the reduced floating point texture bandwidth, which is only 1.2 Gfloats/second compared with ATI's 4.5 Gfloats/second [27].

# Chapter 3

# PROPOSAL

We propose to leverage the GPU stream computing capabilities provided by Brook for GPUs to improve upon the "spot" radar altimeter algorithm presented in Ohio University's terrain database integrity monitoring research [1, 6, 12, 25]. That algorithm calculates the terrain height at sample points within the radar altimeter beam's detection area using a bilinear interpolation method. In Figure 3.1, $(\varphi, \lambda)$ is the latitude and longitude of the sample point where the elevation needs to be computed [12]. The surrounding points represent post positions within the terrain database that lie southwest, northwest, northeast, and southeast of the sample point. The interpolation method provides elevation data for sample points that lie between these post datum points. Although accurate, the bilinear interpolation method seems inefficient from the standpoint of the computer processing required to determine the sample point positions in the DEM, gather the surrounding post data values, then perform the interpolation. Since it's unlikely that many sample points lie exactly at a post position, the interpolation must be performed for the majority of terrain height samples.

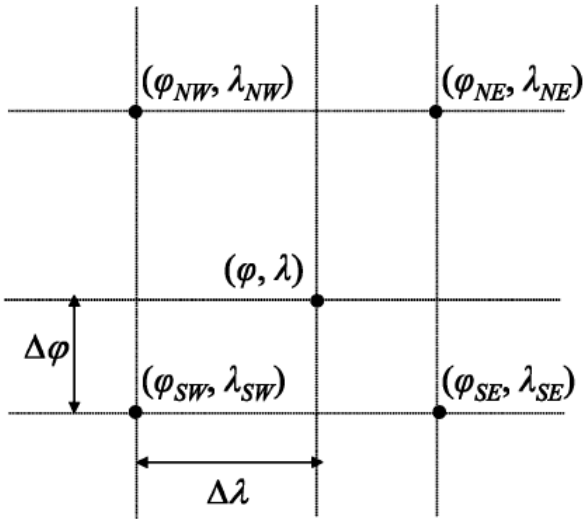



Figure 3.1. Bilinear Interpolation

Another limitation of the "spot" model is that the radar altimeter beam's detection area is assumed to be directly below the aircraft or "the model assumed in which the radar altimeter measures the plumb-bob height AGL" [12] (Figure 2.17 [1]). This restriction creates correlation errors when the aircraft performs any pitch and roll maneuvers during flight, which is especially likely during critical phases of flight such as takeoff and landing. As mentioned in the Ohio University research, "the large bank angles cause a mismatch between the radar altimeter measurements and the interpolated DEM elevation" [25].

We propose a model that improves upon the "spot" method of approximating the radar altimeter beam's characteristics. We have implemented a ray caster using Brook for GPUs that simulates the shape and behavior of the radar altimeter's cone-shaped beam during all phases of flight. Ray casting is a subset of ray tracing in which the shadow and reflection rays are not calculated. It is a technique normally used to render a 3-D scene, but we are using it instead to synthesize terrain distances. A stream of rays originating from a single point, representing the radar altimeter antenna, are projected in a cone shape toward the terrain model and the intersections between the rays and the terrain are calculated. As a prerequisite for this method the digital elevation model must first be converted into a terrain mesh, which can then be processed as a stream of triangles in 3-D space. The distances between the ray origin and the ray-triangle intersection points are calculated, then a Brook reduction is used to derive the minimum distance value. That value represents the synthesized range to terrain that would be expected from a pulse radar altimeter. The synthesized and actual ranges can then be compared as part of the integrity monitor already discussed in the Ohio University research. For simplicity, we shall refer to our proposal as the "Ray Stream" model.

# Chapter 4

# EXPERIMENTS AND RESULTS

Three experiments are performed to measure the speed and accuracy of our "Ray Stream" radar altimeter beam simulation. The first experiment compares the performance of our algorithm when targeted for a GPU versus a CPU. The second compares the accuracy of terrain measurements for different numbers of rays and patterns used to simulate the radar altimeter beam. The final experiment compares the accuracy of the "Ray Stream" model versus the "Spot" model during several flight simulation scenarios.

Our test platform for all experiments is a personal computer with an AMD Athlon 64 processor 3200+ (2 GHz) with 1 GB of memory, running Microsoft Windows XP Professional Version 2002, SP 2. Our commercial graphics processor is an ATI All-In-Wonder X800 Series video card with 256 MB of memory. All code is compiled using Brook for GPUs version 0.4 and the Microsoft Visual C++ Toolkit 2003, version 1.01. MATLAB® is used to chart and graph the experimental data and results. All terrain and flight data created for these experiments is simulated for the sole purpose of testing and does not necessarily represent actual locations or flight conditions.

At the core of these experiments is our "Ray Stream" Radar Altimeter Simulation (`monitor.br`) written in Brook for GPUs (Listing 4.1). The primary components of our simulation are the `rayIntersect` kernel, which we adapted for use with Brook from soft-Surfer's `intersect_RayTriangle` function [30], and the `rayRotate` kernel, developed by concatenating the 3-D rotation matrices described on the ACM SIGGRAPH website [31]. Lines 8-23 of Listing 4.1 are the include statements for the various terrain model and flight path data sets used during the experiments. Lines 25 and 26 define macros used to convert angles from degrees to radians and to convert compass heading, where North is 0°, to a rotation angle in the Cartesian coordinate system. Lines 28-32 define a data structure used to store triangle vertices. Lines 35 and 36 declare the Brook timer functions used during

Experiment 1. The source code for these timer functions (`timer.cpp`) is available in the appendix (Listing 11).

Lines 39-120 contain the `rayIntersect` kernel mentioned previously. The softSurfer algorithm implemented here uses the parametric equation of a plane, **p**, containing a triangle, **T**, similar to the Moller-Trumbore ray-triangle intersection algorithm [32], but computes the parametric coordinates of the intersection point in the plane. To determine the intersection of a ray, **R**, with **T** one must first determine the intersection of **R** and **p**. Given **T** with vertices $V_0, V_1$ and $V_2$, the parametric plane is defined by Equation 4.1.

$$V(s,t) = V_0 + s(V_1 - V_0) + t(V_2 - V_0) = V_0 + s\mathbf{u} + t\mathbf{v} \tag{4.1}$$

where $s$ and $t$ are real numbers, and $\mathbf{u} = (V_1 - V_0)$ and $\mathbf{v} = (V_2 - V_0)$ are edge vectors of **T**. A point $P = V(s,t)$ is in the triangle $T$ when $s \geq 0$, $t \geq 0$, and $s + t \leq 1$. So, given the point of intersection $P_I$, we just need to find its coordinates $(s_I, t_I)$ and then check the inequalities to verify its inclusion in $T$ (Figure 4.1). To solve for $s_I$ and $t_I$ the softSurfer

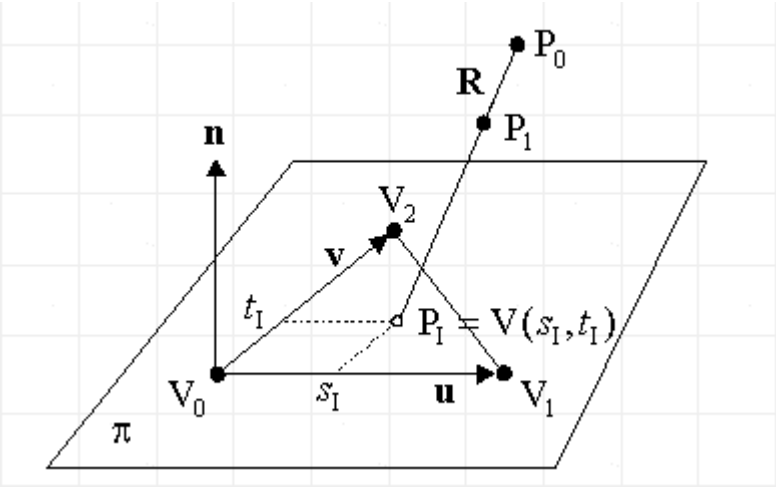



Figure 4.1. Ray-Triangle Intersection

algorithm uses a 3-D generalization of Hill's "perp-dot" product [33]. Given an embedded 2-D plane **p** with a normal vector **n**, and any vector **a** in the plane such that $\mathbf{n} \cdot \mathbf{a} = 0$, the *generalized perp operator* on **p** is defined as $\mathbf{a}^{\wedge} = \mathbf{n} \times \mathbf{a}$. Then, $\mathbf{a}^{\wedge}$ is another vector in plane

$\mathbf{p}$ since $\mathbf{n} \cdot \mathbf{a}^{\wedge} = \mathbf{0}$; it's also perpendicular to $\mathbf{a}$ since $\mathbf{a} \cdot \mathbf{a}^{\wedge} = \mathbf{0}$. Further, the perp operator is linear on vectors in $\mathbf{p}$, so $(A\mathbf{a} + B\mathbf{b})^{\wedge} = A\mathbf{a}^{\wedge} + B\mathbf{b}^{\wedge}$, where $\mathbf{a}$ and $\mathbf{b}$ are vectors in $\mathbf{p}$ and $A$ and $B$ are scalars [30].

The generalized perp operator is used to solve Equation 4.1 for the intersection point $P_I$. Given $\mathbf{w} = (P_I - V_0)$ which is another vector in $\mathbf{p}$, we want to solve the equation $\mathbf{w} = s\mathbf{u} + t\mathbf{v}$ for $s$ and $t$. Taking the dot product of both sides with $\mathbf{v}^{\wedge}$ gives $\mathbf{w} \cdot \mathbf{v}^{\wedge} = s\mathbf{u} \cdot \mathbf{v}^{\wedge} + t\mathbf{v} \cdot \mathbf{v}^{\wedge} = s\mathbf{u} \cdot \mathbf{v}^{\wedge}$. Likewise, taking the dot product with $\mathbf{u}^{\wedge}$ gives $\mathbf{w} \cdot \mathbf{u}^{\wedge} = s\mathbf{u} \cdot \mathbf{u}^{\wedge} + t\mathbf{v} \cdot \mathbf{u}^{\wedge} = t\mathbf{v} \cdot \mathbf{u}^{\wedge}$. We can now solve for $s_I$ and $t_I$ as follows:

$$s_I = \frac{\mathbf{w} \cdot \mathbf{v}^{\wedge}}{\mathbf{u} \cdot \mathbf{v}^{\wedge}} = \frac{\mathbf{w} \cdot (\mathbf{n} \times \mathbf{v})}{\mathbf{u} \cdot (\mathbf{n} \times \mathbf{v})} \qquad t_I = \frac{\mathbf{w} \cdot \mathbf{u}^{\wedge}}{\mathbf{v} \cdot \mathbf{u}^{\wedge}} = \frac{\mathbf{w} \cdot (\mathbf{n} \times \mathbf{u})}{\mathbf{v} \cdot (\mathbf{n} \times \mathbf{u})}$$

The denominators are nonzero whenever $\mathbf{T}$ is nondegenerate, i.e. not a segment or a point. When $\mathbf{T}$ is degenerate the computed normal vector is (0,0,0) [30].

The equations for $s_I$ and $t_I$ are further simplified using the following identity,

$$(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} = (\mathbf{a} \cdot \mathbf{c})\mathbf{b} - (\mathbf{b} \cdot \mathbf{c})\mathbf{a}$$

which holds for left association of the cross product for any three vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$. Applying this identity yields the following:

$$\mathbf{u}^{\wedge} = \mathbf{n} \times \mathbf{u} = (\mathbf{u} \times \mathbf{v}) \times \mathbf{u} = (\mathbf{u} \cdot \mathbf{u})\mathbf{v} - (\mathbf{u} \cdot \mathbf{v})\mathbf{u}$$

$$\mathbf{v}^{\wedge} = \mathbf{n} \times \mathbf{v} = (\mathbf{u} \times \mathbf{v}) \times \mathbf{v} = (\mathbf{u} \cdot \mathbf{v})\mathbf{v} - (\mathbf{v} \cdot \mathbf{v})\mathbf{u}$$

Now, $s_I$ and $t_I$ can be computed using dot products only, as follows:

$$s_I = \frac{(\mathbf{u} \cdot \mathbf{v})(\mathbf{w} \cdot \mathbf{v}) - (\mathbf{v} \cdot \mathbf{v})(\mathbf{w} \cdot \mathbf{u})}{(\mathbf{u} \cdot \mathbf{v})^{\mathbf{2}} - (\mathbf{u} \cdot \mathbf{u})(\mathbf{v} \cdot \mathbf{v})}$$

$$t_I = \frac{(\mathbf{u} \cdot \mathbf{v})(\mathbf{w} \cdot \mathbf{u}) - (\mathbf{u} \cdot \mathbf{u})(\mathbf{w} \cdot \mathbf{v})}{(\mathbf{u} \cdot \mathbf{v})^{\mathbf{2}} - (\mathbf{u} \cdot \mathbf{u})(\mathbf{v} \cdot \mathbf{v})}$$

The terms have been arranged so the denominators are the same and only need to be calculated once [30].

The `rayIntersect` kernel takes a stream of ray vectors with origin **o** and a stream of triangles as inputs, calculates the ray-triangle intersections in parallel using the softSurfer algorithm, and returns an output stream of hit results. The hit result is either +1 for a successful ray-triangle hit or a negative number representing a miss. For each hit result the coordinates of the intersection point are also returned in the stream output. Lines 71-74 calculate the triangle edge vectors and plane normal. Some processing time could be saved by including the plane normals with the vertex data in the triangle database, but that was not done for this research. Lines 76 and 77 check that the triangle is not degenerate and lines 79 - 83 check that the ray vector does not lie within the triangle plane. Lines 86 - 87 check that the ray vector does not go away from the triangle plane. If all the previous checks pass, the ray vector intersects the triangle plane and line 90 calculates the point of intersection. Lines 93 - 99 calculate the parameters used in the simplified softSurfer equations for $s_I$ and $t_I$, then the values of $s_I$ and $t_I$ are calculated and the inequalities checked in lines 102 - 109. If those checks pass, the intersection point lies within the triangle and the coordinates are set in lines 110 - 114.

The `rayRotate` kernel in lines 122 - 146 is another key piece of the simulation. This kernel takes a stream of rays as input, calculates their rotation in 3-D space in parallel, and outputs a stream of rotated rays. The amount of rotation is based on the rotation matrix parameters which are also provided as input. These parameters consist of the cosine, sine, and -sine of the rotation angles for heading, pitch, and roll. The rotation formula in lines 138 - 145 was generated by concatenation of the individual rotation matrices for the $x$, $y$, and $z$ axes as follows:

$$\mathbf{R_T} = \mathbf{R_x R_y R_z}$$

where,

$$\mathbf{R_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & \sin\theta_x & 0 \\ 0 & -\sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R_y} = \begin{bmatrix} \cos\theta_y & 0 & -\sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R_z} = \begin{bmatrix} \cos\theta_z & \sin\theta_z & 0 & 0 \\ -\sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The `dist2pt` kernel in lines 148 - 169 is a small, but important, part of the radar altimeter simulation. This kernel takes the ray origin and the stream of hit results as inputs, calculates the distance between the origin and each successful ray-triangle intersection point, and outputs a stream of distances. For each non-hit result the output distance is set to 9999 meters to represent a value beyond the 762-meter (2500 feet) detection range of a typical radar altimeter. Lines 171 - 184 of Listing 4.1 contain the `minRange` reduction. This Brook reduction takes a stream of ranges as input and returns the minimum value as output. This operation is extremely efficient when executed by the parallel processors on the GPU due to the associative property of calculating a minimum. Line 186 sets the number of rays used in each experiment.

The main function begins at line 188. Stream and array structures for the ray, triangle, intersection, range, and radar altitude data are defined in lines 190 - 199. Lines 201 - 208 define variables used for timing and control loops. The aircraft heading, pitch, roll, latitude, longitude, and altitude variables are defined in lines 213 - 218. Line 219 defines the origin

of the ray vectors based on the longitude, latitude, and altitude of the aircraft. The altitude defined here is not the radar altitude, but the altitude of the aircraft with respect to mean sea level as determined by a navigation computer such as a Differential GPS (DGPS). Lines 223 - 234 contain the constants and variables used to calculate the directions of the ray vectors that comprise the radar altimeter beam. A 50° beam width is typical of most radar altimeter antennas [15], so that is the beam shape used throughout most of these experiments. Lines 236 - 435 set the vector values for all of the rays. The ray vectors are converted from array data into stream data at line 439.

The simulation loop begins at line 441. This is a simulation of an aircraft flying over the terrain defined by the terrain database (Appendix Listing 1). The radar altimeter antenna beam is projected at the terrain based on the location, altitude, heading, and attitude of the aircraft. The aircraft orientation is updated each loop, once for each set of coordinates in 3-D space or "waypoints" in a flight path database. This database simulates the real-time navigation data that would be provided to the terrain database integrity monitor. Fifty waypoints were used for most of the experiments. This was based on a target update rate of 5 Hz for integrity checking, which corresponds to the amount of time it would take for a commercial aircraft traveling at 150 knots to traverse the 720 meters of terrain in our test database. Lines 444 - 448 reset the timer data used in Experiment 1. Lines 450 - 458 update the current aircraft orientation and position from the flight path database. The terrain data is loaded and converted into a stream format in lines 460 - 469. This data could have been initialized before the loop since our terrain database only contains 128 triangles (Appendix Listing 1). However, in an actual integrity monitor the terrain information would need to be continually updated based on the location of the aircraft, so the code was placed here to simulate that process. Lines 471 - 481 calculate the rotation matrix parameters based on the heading, roll, and pitch information provided by the flight path database. Line 483 calculates the stream I/O and set up time for this pass of the loop. The main processing for the simulation loop occurs from lines 485 - 489. The rays are rotated into the correct position based on aircraft orientation using the `rayRotate` kernel. Next, the ray-triangle

intersections are calculated using the `rayIntersect` kernel. Then, the distances between the ray vector origin and the ray-triangle intersections are calculated by the `dist2pt` kernel. Finally, the `minRange` reduction calculates the minimum range from the entire stream of distance values. This minimum range represents the predicted radar altitude based on the leading edge return of an actual pulse radar altimeter. The synthesized radar altitude value is converted from a stream format back into an array at line 495. At this point the value can be used for comparison with the actual radar altitude as part of the integrity monitoring process. The remaining lines of code are used for timing calculations and displaying results.

The source files generated by the Brook BRCC compiler can be found in the appendix (Listings 9 and 10). These files include both the GPU(DX9) and CPU versions of the source file (`monitor.cpp`) created from our `monitor.br` file by the Brook source-to-source compiler. Pixel Shader 2.0 is the profile selected for all GPU builds.

Listing 4.1. 'Ray Stream' Radar Altimeter Simulation (monitor.br)

```
1  //monitor.br
2  //Author: Sean McKeon (2009)
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <math.h>
7
8  // *** Terrain models and flight paths ***
9  //#include "terrain32.h"
10 //#include "terrain64.h"
11 //#include "terrain96.h"
12 #include "terrain128.h"
13 //#include "terrain256.h"
14 //#include "terrain512.h"
15 //#include "terrain1024.h"
16 //#include "terrain2048.h"
17 #include "flightpath1.h"
18 //#include "flightpath2.h"
19 //#include "flightpath3.h"
20 //#include "flightpath1_spot.h"
21 //#include "flightpath2_spot.h"
22 //#include "flightpath3_spot.h"
23 // ************************************
24
25 #define DEG2RAD(x) (((x)*3.1415926535898)/180.0)
26 #define HDG2ANGLE(x) (-x + 90.0)
27
28 typedef struct triangle_t{
29     float3 v0;
30     float3 v1;
31     float3 v2;
32 } Triangle;
33
34 //declare the Brook timer functions
35 void resetTimer(void);
36 float getTimer(void);
```

```
37
38
39  // ********************************************************************
40  // rayIntersect(): intersect a ray with a 3-D triangle
41  // ********************************************************************
42  // Copyright 2001, softSurfer (www.softsurfer.com)
43  // This code may be freely used and modified for any purposes
44  // provided that this copyright notice is included with it.
45  // SoftSurfer makes no warranty for this code, and cannot be held
46  // liable for any real or imagined damage resulting from its use.
47  // Users of this code must verify correctness for their application.
48  // ********************************************************************
49  // Adapted for use with Brook for GPUs by Sean McKeon (2009).
50  // ********************************************************************
51  //    Input:  a stream of ray vectors R with origin o,
52  //            and a stream of triangles T with vertices v0, v1, & v2
53  //    Output: Hit = a stream of intersection points I (when they exist)
54  //                  and intersection test results for each ray-triangle pair
55  //    Return: 1 = intersect in unique point I
56  //           -1 = triangle is degenerate (a segment or point)
57  //           -2 = ray is parallel to triangle plane
58  //           -3 = ray goes away from triangle
59  //           -4 = ray intersects outside of triangle (case 1)
60  //           -5 = ray intersects outside of triangle (case 2)
61  //
62  kernel void rayIntersect( float3 o, float3 R<>, Triangle T<>, out float4 Hit<> )
63  {
64      float3    u, v, n;               // triangle vectors
65      float3    w0, w;                 // ray vectors
66      float     r, a, b;               // params to calc ray-plane intersect
67      float     uu, uv, vv, wu, wv, D; // dot products
68      float     s, t;                  // plane equation params
69      float3    I;                     // intersection point
70
71      // get triangle edge vectors and plane normal
72      u = T.v1 - T.v0;
73      v = T.v2 - T.v0;
74      n = cross(u,v);
75
76      if (n.x == 0.0 && n.y == 0.0 && n.z == 0.0) {
77          Hit.w = -1.0; // triangle is degenerate
78      }else{
79          w0 = o - T.v0;
80          a = -dot(n,w0);
81          b = dot(n,R);
82          if (abs(b) < 0.00001) {
83              Hit.w = -2.0; // ray is parallel to triangle plane
84          }else{
85          // get intersect point of ray with triangle plane
86              r = a / b;
87              if (r < 0.0) {
88                  Hit.w = -3.0; // ray goes away from triangle
89              }else{
90                  I = o + r * R; // intersect point of ray and plane
91
92                  // is I inside T?
93                  uu = dot(u,u);
94                  uv = dot(u,v);
95                  vv = dot(v,v);
96                  w = I - T.v0;
97                  wu = dot(w,u);
98                  wv = dot(w,v);
99                  D = uv * uv - uu * vv;
100
101                 // get and test parametric coords
102                 s = (uv * wv - vv * wu) / D;
103                 if (s < 0.0 || s > 1.0) {
104                     Hit.w = -4.0; // I is outside T
```

```
105                 }else{
106                     t = (uv * wu - uu * wv) / D;
107                     if (t < 0.0 || (s + t) > 1.0){
108                         Hit.w = -5.0; // I is outside T
109                     }
110                     else{
111                         Hit.w = 1.0; // I is in T
112                         Hit.x = I.x;
113                         Hit.y = I.y;
114                         Hit.z = I.z;
115                     }
116                 }
117             }
118         }
119     }
120 }
121 
122 // ********************************************************************
123 // rayRotate(): rotates a ray in 3-D space
124 // ********************************************************************
125 // Author: Sean McKeon (2009).
126 // ********************************************************************
127 //    Input:  a stream of ray vectors R with origin o,
128 //            heading rotation matrix hdgM params x, y, z, (Z-axis rotation)
129 //            pitch rotation matrix pitchM params x, y, z, (Y-axis rotation)
130 //            roll rotation matrix rollM params x, y, z,   (X-axis rotation)
131 //            Note: where x, y, and z are cos, sin, and -sin of the
132 //                  respective angle of rotation.
133 //    Output: rotR = a stream of rotated ray vectors with origin o
134 //    Return: none
135 //
136 kernel void rayRotate (float3 R<>, float3 hdgM, float3 pitchM, float3 rollM, out float3
        rotR<>)
137 {
138     float x_prod1, x_prod2, y_prod1, y_prod2;
139     x_prod1 = rollM.y*pitchM.y*hdgM.x+rollM.x*hdgM.z;
140     x_prod2 = rollM.x*pitchM.y*hdgM.x+rollM.z*hdgM.z;
141     y_prod1 = rollM.y*pitchM.y*hdgM.y+rollM.x*hdgM.x;
142     y_prod2 = rollM.x*pitchM.y*hdgM.y+rollM.z*hdgM.x;
143     rotR = float3(R.x*pitchM.x*hdgM.x+R.y*x_prod1+R.z*x_prod2,
144                   R.x*pitchM.x*hdgM.y+R.y*y_prod1+R.z*y_prod2,
145                   R.x*pitchM.z+R.y*rollM.y*pitchM.x+R.z*rollM.x*pitchM.x);
146 }
147 
148 // ********************************************************************
149 // dist2pt(): calculates distance between two points in 3-D space
150 // ********************************************************************
151 // Author: Sean McKeon (2009).
152 // ********************************************************************
153 //    Input:  an origin point o, with coordinates x, y, z,
154 //            a stream of points hit, with coordinates x, y, z
155 //    Output: dist = a stream of distances between origin o and hit points
156 //    Return: none
157 //
158 kernel void dist2pt (float3 o, float4 hit<>, out float dist<>)
159 {
160     float3 diff, target;
161     if(hit.w != 1.0)
162     {
163         dist = 9999.0; //set a miss > radar altimeter detection range
164     }else{
165         target = float3(hit.x, hit.y, hit.z);
166         diff = o - target;
167         dist = sqrt(dot(diff, diff));
168     }
169 }
170 
171 // ********************************************************************
```

```
172 // minRange(): calculates the minimum distance value
173 // ********************************************************************
174 // Author: Sean McKeon (2009).
175 // ********************************************************************
176 //    Input:  a stream of distances (range)
177 //    Output: min_range = the smallest distance in the stream
178 //    Return: none
179 //
180 reduce void minRange (float range<>, reduce float min_range<>)
181 {
182   if (range < min_range)
183     min_range = range;
184 }
185
186 #define NUM_RAYS 65
187
188 int main () {
189
190     //stream structures for terrain, ray, intersection, range, and altitude data
191     Triangle triangle<1,NUM_TRIANGLES>;
192     Triangle tridat[1][NUM_TRIANGLES];
193     float3 ray<NUM_RAYS,1>;
194     float3 raydat[NUM_RAYS][1];
195     float3 rotated_ray<NUM_RAYS,1>;
196     float4 hit<NUM_RAYS, NUM_TRIANGLES>;
197     float range<NUM_RAYS,NUM_TRIANGLES>;
198     float radalt<1>;
199     float radalt_out[1];
200
201     //timer variables
202     float msec_IO, msec_kernel, msec_total;
203     float msec_IO_sum = 0.0;
204     float msec_kernel_sum = 0.0;
205     float msec_total_sum = 0.0;
206
207     //miscellaneous variables
208     int j,k;
209
210     // ************************************************************************
211     // ***    Aircraft heading, attitude, position, and altitude variables     ***
212     // ************************************************************************
213     double HEADING = 0.0;  //compass heading in degrees (i.e. 0 degrees is North)
214     double PITCH = 0.0;    //positive angle is degrees 'nose up'
215     double ROLL = 0.0;     //positive angle is degrees 'bank right'
216     float LONGITUDE = 0.0; //x-axis world coordinates (meters)
217     float LATITUDE = 0.0;  //y-axis world coordinates (meters)
218     float ALTITUDE = 0.0;  //z-axis world coordinates (meters)
219     float3 origin = float3(LONGITUDE, LATITUDE, ALTITUDE);
220     // ************************************************************************
221
222     //heading, pitch, and roll rotation matrix variables
223     float3 hdg_matrix, pitch_matrix, roll_matrix;
224     float hdg_angle_sin, hdg_angle_cos, pitch_angle_sin, pitch_angle_cos,
            roll_angle_sin, roll_angle_cos;
225
226     //radar altimeter beam shape data
227     double beam_angle_rad;
228     double beam_angle_deg;
229     float beam_angle_sin, beam_angle_cos;
230     const double beam_angle_45 = 45.0;
231     const float beam_angle_offset = (float)sin(DEG2RAD(beam_angle_45));
232     const double beam_angle_22pt5 = 22.5;
233     const float offset_22pt5_cos = (float)cos(DEG2RAD(beam_angle_22pt5));
234     const float offset_22pt5_sin = (float)sin(DEG2RAD(beam_angle_22pt5));
235
236     // ****************************
237     // **** define the ray data ****
238     // ****************************
```

```
239     //center ray
240     //   |
241     // - 0 -
242     //   |
243     float3 CENTER_RAY = float3(0.0, 0.0, -1.0);
244     raydat[0][0] = CENTER_RAY;
245
246     // Five degree primary rays
247     //    2
248     //    |
249     // 3-   -1
250     //    |
251     //    4
252     beam_angle_deg = 5.0;
253     beam_angle_rad = DEG2RAD(beam_angle_deg);
254     beam_angle_sin = (float)sin(beam_angle_rad);
255     beam_angle_cos = (float)cos(beam_angle_rad);
256
257     raydat[1][0] = float3(beam_angle_sin, 0.0, -beam_angle_cos);
258     raydat[2][0] = float3(0.0, beam_angle_sin, -beam_angle_cos);
259     raydat[3][0] = float3(-beam_angle_sin, 0.0, -beam_angle_cos);
260     raydat[4][0] = float3(0.0, -beam_angle_sin, -beam_angle_cos);
261
262     // Ten degree primary rays
263     // 7    6
264     //  \  /
265     //  /  \
266     // 8    5
267     beam_angle_deg = 10.0;
268     beam_angle_rad = DEG2RAD(beam_angle_deg);
269     beam_angle_sin = (float)sin(beam_angle_rad);
270     beam_angle_cos = (float)cos(beam_angle_rad);
271
272     raydat[5][0] = float3(beam_angle_offset*beam_angle_sin,
            -beam_angle_offset*beam_angle_sin, -beam_angle_cos);
273     raydat[6][0] = float3(beam_angle_offset*beam_angle_sin,
            beam_angle_offset*beam_angle_sin, -beam_angle_cos);
274     raydat[7][0] = float3(-beam_angle_offset*beam_angle_sin,
            beam_angle_offset*beam_angle_sin, -beam_angle_cos);
275     raydat[8][0] = float3(-beam_angle_offset*beam_angle_sin,
            -beam_angle_offset*beam_angle_sin, -beam_angle_cos);
276
277     // Twenty-five degree primary rays
278     //        13
279     //  14    |    12
280     //     \     /
281     // 15 -       - 11
282     //     /     \
283     //  16    |    10
284     //        9
285     beam_angle_deg = 25.0;
286     beam_angle_rad = DEG2RAD(beam_angle_deg);
287     beam_angle_sin = (float)sin(beam_angle_rad);
288     beam_angle_cos = (float)cos(beam_angle_rad);
289
290     raydat[9][0]  = float3(0.0, -beam_angle_sin, -beam_angle_cos);
291     raydat[10][0] = float3(beam_angle_offset*beam_angle_sin,
            -beam_angle_offset*beam_angle_sin, -beam_angle_cos);
292     raydat[11][0] = float3(beam_angle_sin, 0.0, -beam_angle_cos);
293     raydat[12][0] = float3(beam_angle_offset*beam_angle_sin,
            beam_angle_offset*beam_angle_sin, -beam_angle_cos);
294     raydat[13][0] = float3(0.0, beam_angle_sin, -beam_angle_cos);
295     raydat[14][0] = float3(-beam_angle_offset*beam_angle_sin,
            beam_angle_offset*beam_angle_sin, -beam_angle_cos);
296     raydat[15][0] = float3(-beam_angle_sin, 0.0, -beam_angle_cos);
297     raydat[16][0] = float3(-beam_angle_offset*beam_angle_sin,
            -beam_angle_offset*beam_angle_sin, -beam_angle_cos);
298
```

```
299     // Twenty degree primary rays
300     //       21
301     //  22    |    20
302     //     \     /
303     // 23 -       - 19
304     //     /     \
305     //  24    |    18
306     //       17
307     beam_angle_deg = 20.0;
308     beam_angle_rad = DEG2RAD(beam_angle_deg);
309     beam_angle_sin = (float)sin(beam_angle_rad);
310     beam_angle_cos = (float)cos(beam_angle_rad);
311
312     raydat[17][0] = float3(0.0, -beam_angle_sin, -beam_angle_cos);
313     raydat[18][0] = float3(beam_angle_offset*beam_angle_sin,
            -beam_angle_offset*beam_angle_sin, -beam_angle_cos);
314     raydat[19][0] = float3(beam_angle_sin, 0.0, -beam_angle_cos);
315     raydat[20][0] = float3(beam_angle_offset*beam_angle_sin,
            beam_angle_offset*beam_angle_sin, -beam_angle_cos);
316     raydat[21][0] = float3(0.0, beam_angle_sin, -beam_angle_cos);
317     raydat[22][0] = float3(-beam_angle_offset*beam_angle_sin,
            beam_angle_offset*beam_angle_sin, -beam_angle_cos);
318     raydat[23][0] = float3(-beam_angle_sin, 0.0, -beam_angle_cos);
319     raydat[24][0] = float3(-beam_angle_offset*beam_angle_sin,
            -beam_angle_offset*beam_angle_sin, -beam_angle_cos);
320
321     // Fifteen degree primary rays
322     //       29
323     //  30    |    28
324     //     \     /
325     // 31 -       - 27
326     //     /     \
327     //  32    |    26
328     //       25
329     beam_angle_deg = 15.0;
330     beam_angle_rad = DEG2RAD(beam_angle_deg);
331     beam_angle_sin = (float)sin(beam_angle_rad);
332     beam_angle_cos = (float)cos(beam_angle_rad);
333
334     raydat[25][0] = float3(0.0, -beam_angle_sin, -beam_angle_cos);
335     raydat[26][0] = float3(beam_angle_offset*beam_angle_sin,
            -beam_angle_offset*beam_angle_sin, -beam_angle_cos);
336     raydat[27][0] = float3(beam_angle_sin, 0.0, -beam_angle_cos);
337     raydat[28][0] = float3(beam_angle_offset*beam_angle_sin,
            beam_angle_offset*beam_angle_sin, -beam_angle_cos);
338     raydat[29][0] = float3(0.0, beam_angle_sin, -beam_angle_cos);
339     raydat[30][0] = float3(-beam_angle_offset*beam_angle_sin,
            beam_angle_offset*beam_angle_sin, -beam_angle_cos);
340     raydat[31][0] = float3(-beam_angle_sin, 0.0, -beam_angle_cos);
341     raydat[32][0] = float3(-beam_angle_offset*beam_angle_sin,
            -beam_angle_offset*beam_angle_sin, -beam_angle_cos);
342
343     // Five degree secondary rays
344     // 35  34
345     //  \  /
346     //  /  \
347     // 36  33
348     beam_angle_deg = 5.0;
349     beam_angle_rad = DEG2RAD(beam_angle_deg);
350     beam_angle_sin = (float)sin(beam_angle_rad);
351     beam_angle_cos = (float)cos(beam_angle_rad);
352
353     raydat[33][0] = float3(beam_angle_offset*beam_angle_sin,
            -beam_angle_offset*beam_angle_sin, -beam_angle_cos);
354     raydat[34][0] = float3(beam_angle_offset*beam_angle_sin,
            beam_angle_offset*beam_angle_sin, -beam_angle_cos);
355     raydat[35][0] = float3(-beam_angle_offset*beam_angle_sin,
            beam_angle_offset*beam_angle_sin, -beam_angle_cos);
```

```
356     raydat[36][0] = float3(-beam_angle_offset*beam_angle_sin,
            -beam_angle_offset*beam_angle_sin, -beam_angle_cos);
357
358     // Ten degree secondary rays
359     //    38
360     //     |
361     // 39-   -37
362     //     |
363     //    40
364     beam_angle_deg = 10.0;
365     beam_angle_rad = DEG2RAD(beam_angle_deg);
366     beam_angle_sin = (float)sin(beam_angle_rad);
367     beam_angle_cos = (float)cos(beam_angle_rad);
368
369     raydat[37][0] = float3(beam_angle_sin, 0.0, -beam_angle_cos);
370     raydat[38][0] = float3(0.0, beam_angle_sin, -beam_angle_cos);
371     raydat[39][0] = float3(-beam_angle_sin, 0.0, -beam_angle_cos);
372     raydat[40][0] = float3(0.0, -beam_angle_sin, -beam_angle_cos);
373
374     // Fifteen degree secondary rays
375     //     43   42
376     //  44  \   / 41
377     //     \     /
378     //     /     \
379     //  45  /   \ 48
380     //     46   47
381     beam_angle_deg = 15.0;
382     beam_angle_rad = DEG2RAD(beam_angle_deg);
383     beam_angle_sin = (float)sin(beam_angle_rad);
384     beam_angle_cos = (float)cos(beam_angle_rad);
385
386     raydat[41][0] = float3(offset_22pt5_cos*beam_angle_sin,
            offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
387     raydat[42][0] = float3(offset_22pt5_sin*beam_angle_sin,
            offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
388     raydat[43][0] = float3(-offset_22pt5_sin*beam_angle_sin,
            offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
389     raydat[44][0] = float3(-offset_22pt5_cos*beam_angle_sin,
            offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
390     raydat[45][0] = float3(-offset_22pt5_cos*beam_angle_sin,
            -offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
391     raydat[46][0] = float3(-offset_22pt5_sin*beam_angle_sin,
            -offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
392     raydat[47][0] = float3(offset_22pt5_sin*beam_angle_sin,
            -offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
393     raydat[48][0] = float3(offset_22pt5_cos*beam_angle_sin,
            -offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
394
395     // Twenty degree secondary rays
396     //     51   50
397     //  52  \   / 49
398     //     \     /
399     //     /     \
400     //  53  /   \ 56
401     //     54   55
402     beam_angle_deg = 20.0;
403     beam_angle_rad = DEG2RAD(beam_angle_deg);
404     beam_angle_sin = (float)sin(beam_angle_rad);
405     beam_angle_cos = (float)cos(beam_angle_rad);
406
407     raydat[49][0] = float3(offset_22pt5_cos*beam_angle_sin,
            offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
408     raydat[50][0] = float3(offset_22pt5_sin*beam_angle_sin,
            offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
409     raydat[51][0] = float3(-offset_22pt5_sin*beam_angle_sin,
            offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
410     raydat[52][0] = float3(-offset_22pt5_cos*beam_angle_sin,
            offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
```

```
411     raydat[53][0] = float3(-offset_22pt5_cos*beam_angle_sin,
            -offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
412     raydat[54][0] = float3(-offset_22pt5_sin*beam_angle_sin,
            -offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
413     raydat[55][0] = float3(offset_22pt5_sin*beam_angle_sin,
            -offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
414     raydat[56][0] = float3(offset_22pt5_cos*beam_angle_sin,
            -offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
415
416     // Twenty-five degree secondary rays
417     //     59   58
418     //  60  \   / 57
419     //     \     /
420     //     /     \
421     //  61  /   \ 64
422     //     62   63
423     beam_angle_deg = 25.0;
424     beam_angle_rad = DEG2RAD(beam_angle_deg);
425     beam_angle_sin = (float)sin(beam_angle_rad);
426     beam_angle_cos = (float)cos(beam_angle_rad);
427
428     raydat[57][0] = float3(offset_22pt5_cos*beam_angle_sin,
            offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
429     raydat[58][0] = float3(offset_22pt5_sin*beam_angle_sin,
            offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
430     raydat[59][0] = float3(-offset_22pt5_sin*beam_angle_sin,
            offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
431     raydat[60][0] = float3(-offset_22pt5_cos*beam_angle_sin,
            offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
432     raydat[61][0] = float3(-offset_22pt5_cos*beam_angle_sin,
            -offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
433     raydat[62][0] = float3(-offset_22pt5_sin*beam_angle_sin,
            -offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
434     raydat[63][0] = float3(offset_22pt5_sin*beam_angle_sin,
            -offset_22pt5_cos*beam_angle_sin, -beam_angle_cos);
435     raydat[64][0] = float3(offset_22pt5_cos*beam_angle_sin,
            -offset_22pt5_sin*beam_angle_sin, -beam_angle_cos);
436
437     //convert ray data to a stream
438     //note: ray data only needs to be initialized once
439     streamRead(ray, raydat);
440
441     for (j = 0; j < NUM_WAYPOINTS; j++)
442     {
443         //reset timer data
444         msec_IO = 0.0;
445         msec_kernel = 0.0;
446         msec_total = 0.0;
447
448         resetTimer();
449
450         //update aircraft heading, attitude, and world coordinates
451         //note: data would be updated from attitude/heading and navigation computers in
                actual system
452         HEADING = flightpath[j].heading;
453         PITCH = flightpath[j].pitch;
454         ROLL = flightpath[j].roll;
455         LONGITUDE = (float)flightpath[j].longitude;
456         LATITUDE = (float)flightpath[j].latitude;
457         ALTITUDE = (float)flightpath[j].altitude;
458         origin = float3(LONGITUDE, LATITUDE, ALTITUDE);
459
460         //load the terrain data
461         //note: data would be updated via a terrain database server in actual system
462         for (k = 0; k < NUM_TRIANGLES; k++)
463         {
464             tridat[0][k].v0 = float3(terrain[k][V0][X], terrain[k][V0][Y],
                    terrain[k][V0][Z]);
```

```
465             tridat[0][k].v1 = float3(terrain[k][V1][X], terrain[k][V1][Y],
                    terrain[k][V1][Z]);
466             tridat[0][k].v2 = float3(terrain[k][V2][X], terrain[k][V2][Y],
                    terrain[k][V2][Z]);
467         }
468         //convert triangle data to a stream
469         streamRead(triangle, tridat);
470
471         //calculate rotation matrix param values
472         hdg_angle_sin = (float)sin(DEG2RAD(HDG2ANGLE(HEADING)));
473         hdg_angle_cos = (float)cos(DEG2RAD(HDG2ANGLE(HEADING)));
474         pitch_angle_sin = (float)sin(DEG2RAD(-PITCH)); //make +pitch 'up' instead of
                'down'
475         pitch_angle_cos = (float)cos(DEG2RAD(-PITCH));
476         roll_angle_sin = (float)sin(DEG2RAD(ROLL));
477         roll_angle_cos = (float)cos(DEG2RAD(ROLL));
478         //store cosine and sine params from rotation matrices
479         hdg_matrix = float3(hdg_angle_cos, hdg_angle_sin, -hdg_angle_sin);
480         pitch_matrix = float3(pitch_angle_cos, pitch_angle_sin, -pitch_angle_sin);
481         roll_matrix = float3(roll_angle_cos, roll_angle_sin, -roll_angle_sin);
482
483         msec_IO = getTimer();
484
485         // *** calculate radar altitude to terrain mesh ***
486         rayRotate(ray, hdg_matrix, pitch_matrix, roll_matrix, rotated_ray);
487         rayIntersect(origin, rotated_ray, triangle, hit);
488         dist2pt(origin, hit, range);
489         minRange(range, radalt);
490         // ************************************************
491
492         msec_kernel = getTimer() - msec_IO;
493
494         //write out stream data
495         streamWrite(radalt, radalt_out);
496
497         msec_total = getTimer();
498         msec_IO = msec_total - msec_kernel;
499
500         //determine timing totals
501         msec_IO_sum += msec_IO;
502         msec_kernel_sum += msec_kernel;
503         msec_total_sum += msec_total;
504
505         //display radar altitude
506         //note: synthesized range would be compared with actual radar altimeter range at
507         //      this point for integrity monitoring
508         printf ("Waypoint: %d, Radar Altitude: %3.2f\n", j, radalt_out[0]);
509         printf ("Position: %3.2f, %3.2f, %3.2f\n", origin.x, origin.y, origin.z);
510         printf ("Heading: %3.2f, Pitch: %3.2f, Roll: %3.2f\n\n", (float)HEADING,
                (float)PITCH, (float)ROLL);
511     }
512
513     //display timing results
514     printf ("Rays: %d, Triangles: %d\n", NUM_RAYS, NUM_TRIANGLES);
515     printf ("Avg kernel time = %3.3f msec\n", msec_kernel_sum/NUM_WAYPOINTS);
516     printf ("Avg IO time     = %3.3f msec\n", msec_IO_sum/NUM_WAYPOINTS);
517     printf ("Avg Total time  = %3.3f msec\n", msec_total_sum/NUM_WAYPOINTS);
518
519     return 0;
520 }
```

## 4.1 Experiment 1

The goal of Experiment 1 is to determine the relative performance improvement when running the "Ray Stream" radar altimeter simulation on a GPU versus a CPU. Brook for GPUs supports multiple targets including DirectX 9, OpenGL, and a CPU benchmark. This experiment tests the processing times for the DX9 and CPU targets. The simulation loop is run for 100 waypoints for each test and the average processing time, i.e. excluding I/O overhead, is calculated. The number of rays used are 1, 5, 9, 17, 33, 49, and 65, while the number of triangles used are 32, 64, 96, 128, 256, 512, 1024, and 2048. The rays are subsets of the ray vector data defined in Listing 4.1. The triangles are subsets or multiple sets of the triangle data in Appendix Listing 1. The 100 waypoints do not represent any flight path data used in any other experiments and are used only to drive the simulation loop the required number of times.

## 4.2 Experiment 1: Results

Figures 4.2 and 4.3 show the timing results for all combinations of rays and triangles tested. The GPU target easily outperforms the CPU target in almost every case. The worst-case processing time for the DX9 version is less than 1.5 msec for 65 rays and 2048 triangles, while the CPU version takes 950 msec. Although not shown in the figures, the I/O time is fairly static for each target. The CPU target does have better I/O speed, posting a time of 0.002 msec. However, the GPU target I/O time is only 0.2 msec, so the tradeoff for the much faster GPU processing speed seems worthwhile.

## 4.3 Experiment 2

The goal of Experiment 2 is to determine the relative accuracy improvement when running the GPU(DX9) version of the "Ray Stream" simulation using different numbers and patterns of rays to represent the radar altimeter beam. Figure 4.4 shows the beam pattern associated with the number of rays used in each test. The patterns vary from a

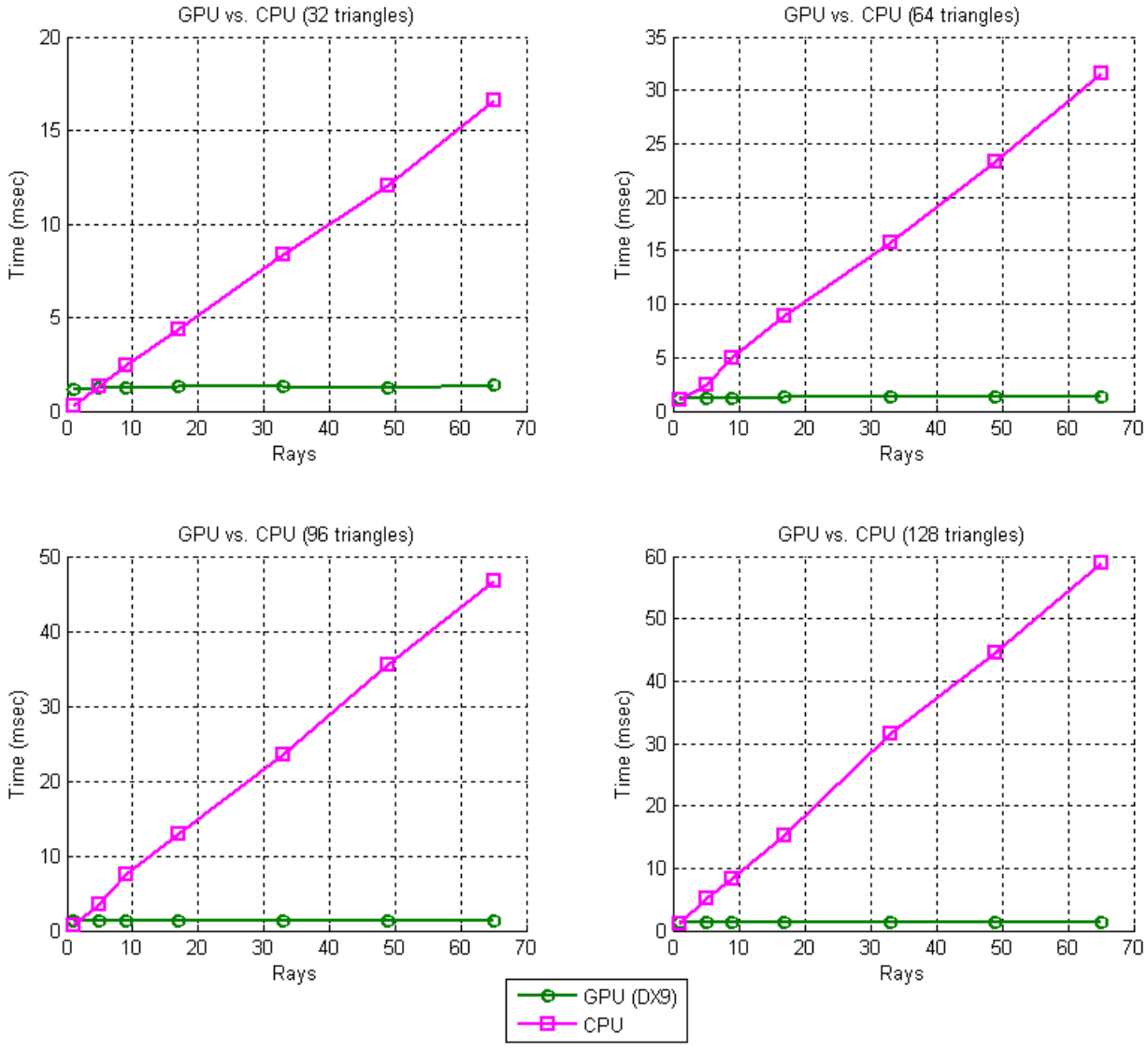



Figure 4.2. Processing Time: 32 - 128 Triangles

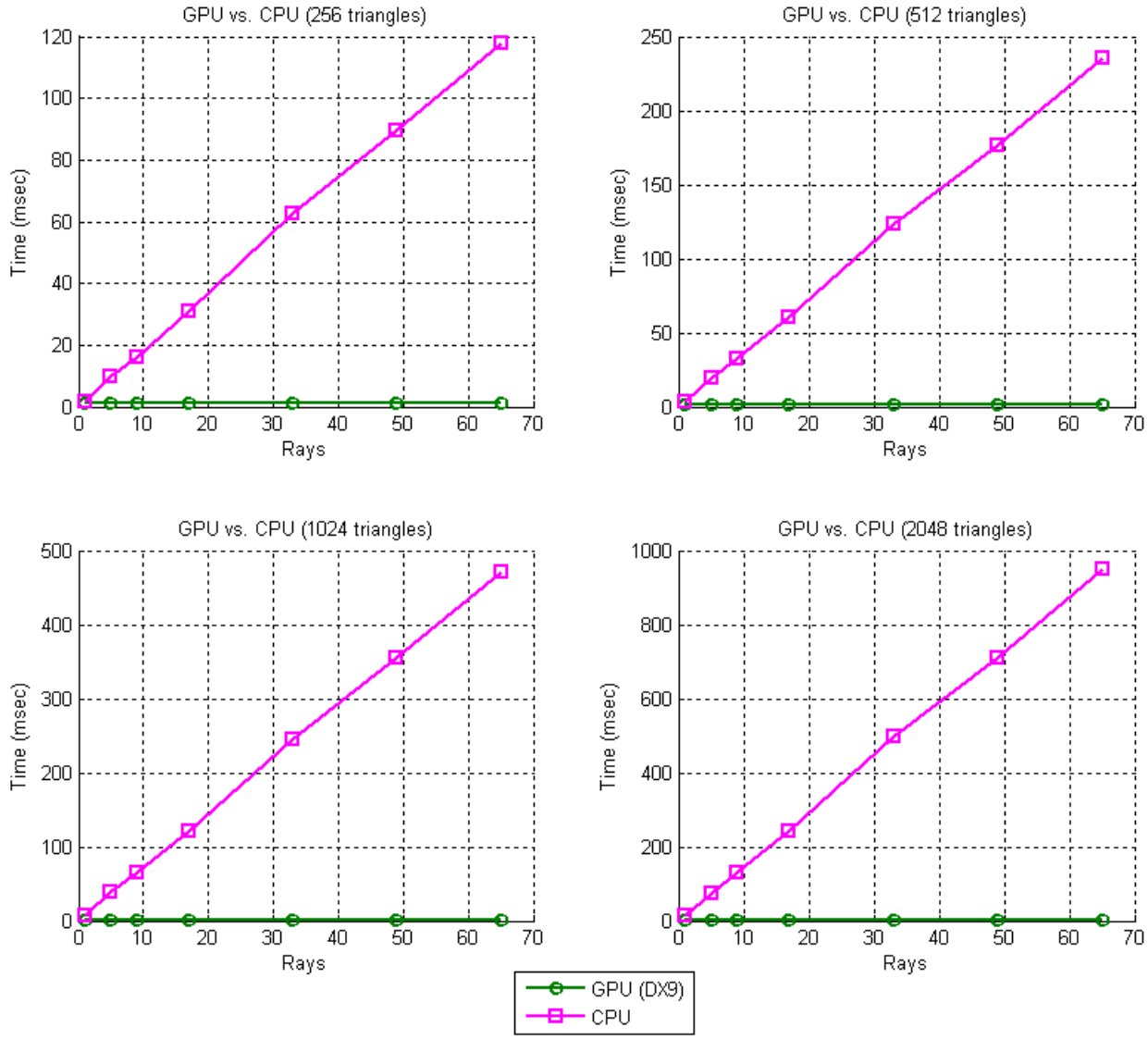



Figure 4.3. Processing Time: 256 - 2048 Triangles

single ray up to 65 rays representing a 50° cone-shaped beam. Appendix Listing 2 shows the aircraft position and orientation data used during this flight simulation. Flight Path 1 is depicted in Figure 4.5 along with other flight paths used in Experiment 3. The terrain shown is a representation of the triangle data in Appendix Listing 1 that is used for all flight simulations.

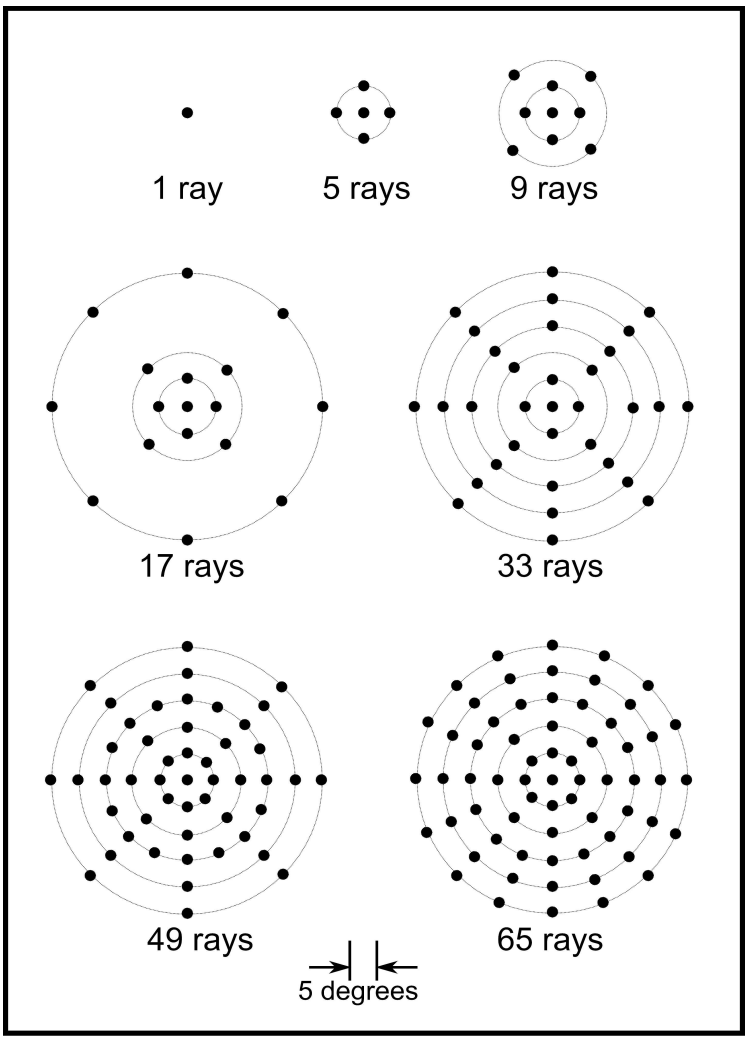



Figure 4.4. Beam Patterns

Flight Path 1 simulates the aircraft flying north along the 270-meter longitude line at an altitude of 300 meters. The aircraft makes a sharp left bank at the 360-meter latitude mark to avoid a collision with the mountainside. The aircraft then proceeds westward until the end of the terrain map is reached. This flight path is characterized by some slightly

rough terrain at the start of the simulation, followed by an abrupt change in terrain due to the mountain, and culminating in low terrain roughness towards the end of the run.

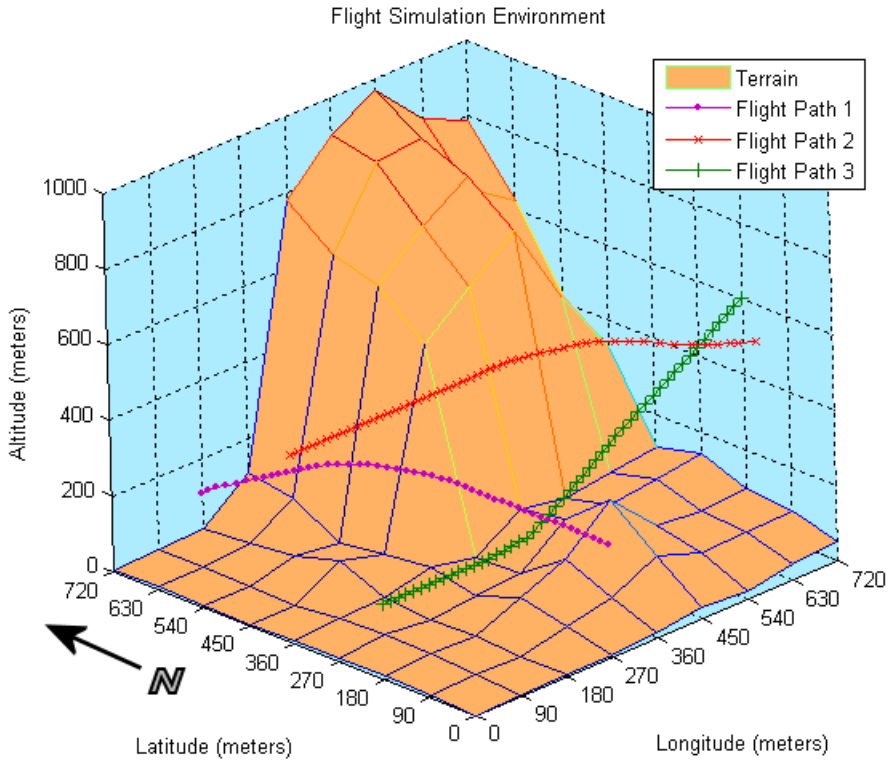



Figure 4.5. Flight Simulation Environment: View 1

## 4.4 Experiment 2: Results

Figure 4.6 shows a comparison of the radar altitude values generated at each waypoint of Flight Path 1. Notice that the graphs tend to converge as the number of rays increases. In fact, once the 17-beam threshold is reached there are only small variances in the range values. This is the point at which the 'outer' ring of 25° rays is added to the beam pattern. However, greater definition can still be observed as the ray count increases to 65. Since 65 rays is the highest number tested in this research, and the GPU target has no performance problems handling that number, the 65-ray beam is selected for use in the remaining experiments.

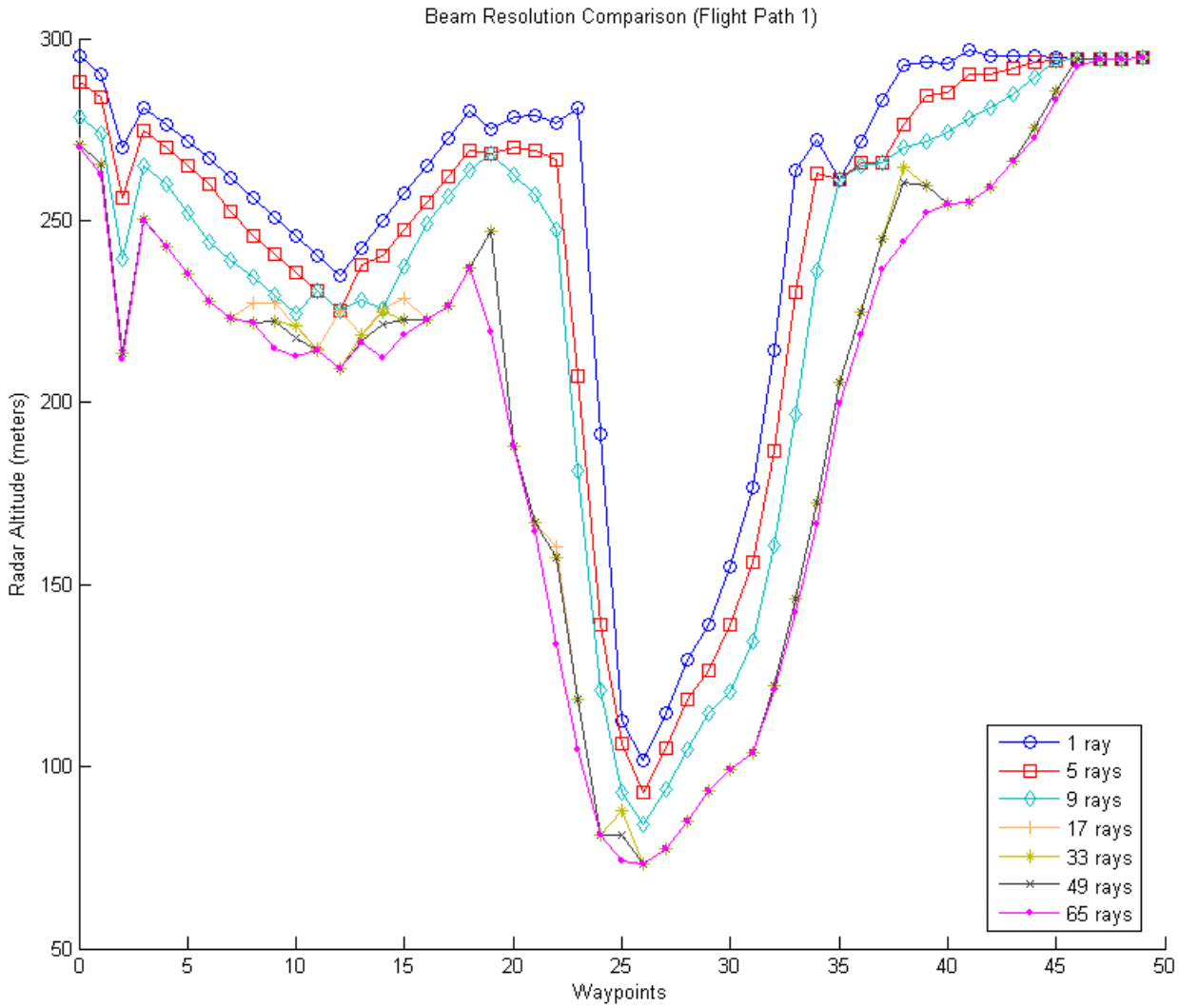



Figure 4.6. Beam Resolution Comparison

Figure 4.7 shows the synthesized terrain profile generated from the radar altitude values shown in Figure 4.6. The terrain profile is simply the aircraft altitude (300 meters) minus the radar altitude at each waypoint. This is just a different way to analyze the same data. It is most useful when the aircraft is in straight-and-level flight, since it approximates the height and contour of the terrain directly below the aircraft. It is slightly less useful when the aircraft is in a more extreme attitude. The most distinguished feature in the terrain profile from Flight Path 1 is the mountainside detected during the hard left bank between Waypoints 20 - 35.

One point of interest in both of the figures from this experiment is the sharp 'peak' in the graphs at Waypoint 2. This is caused by a data entry error when coding Flight Path 1 (Appendix Listing 2). The longitude value is entered as 360 meters instead of 270. That position coincides with a medium-size mountain peak on the terrain map which is correctly detected by the "Ray Stream" model. This exemplifies another benefit of terrain database integrity monitoring - the detection of navigational errors. If the aircraft's navigation system actually provides faulty position data like this the integrity monitor will set off an integrity alert when the predicted radar altitude range deviates from the actual range by such an excessive amount.

## 4.5 Experiment 3

The goal of Experiment 3 is to compare the accuracy of our "Ray Stream" model versus the "Spot" model used in the Ohio University research. Three flight paths are used in our simulation environment to test how the models perform during straight-and-level flight as well as during pitch and roll maneuvers. Since the exact method used to determine terrain sampling points in the "Spot" model is not available, we use our "Ray Stream" model with 0° pitch and 0° roll selected to simulate it. The flight path data used to simulate the "Spot" model is available in the appendix (Listings 6, 7, and 8).

Flight Path 1 is previously described. However, it should be mentioned that the data error at Waypoint 2 is corrected for this experiment (Appendix Listing 3). Flight Paths 2

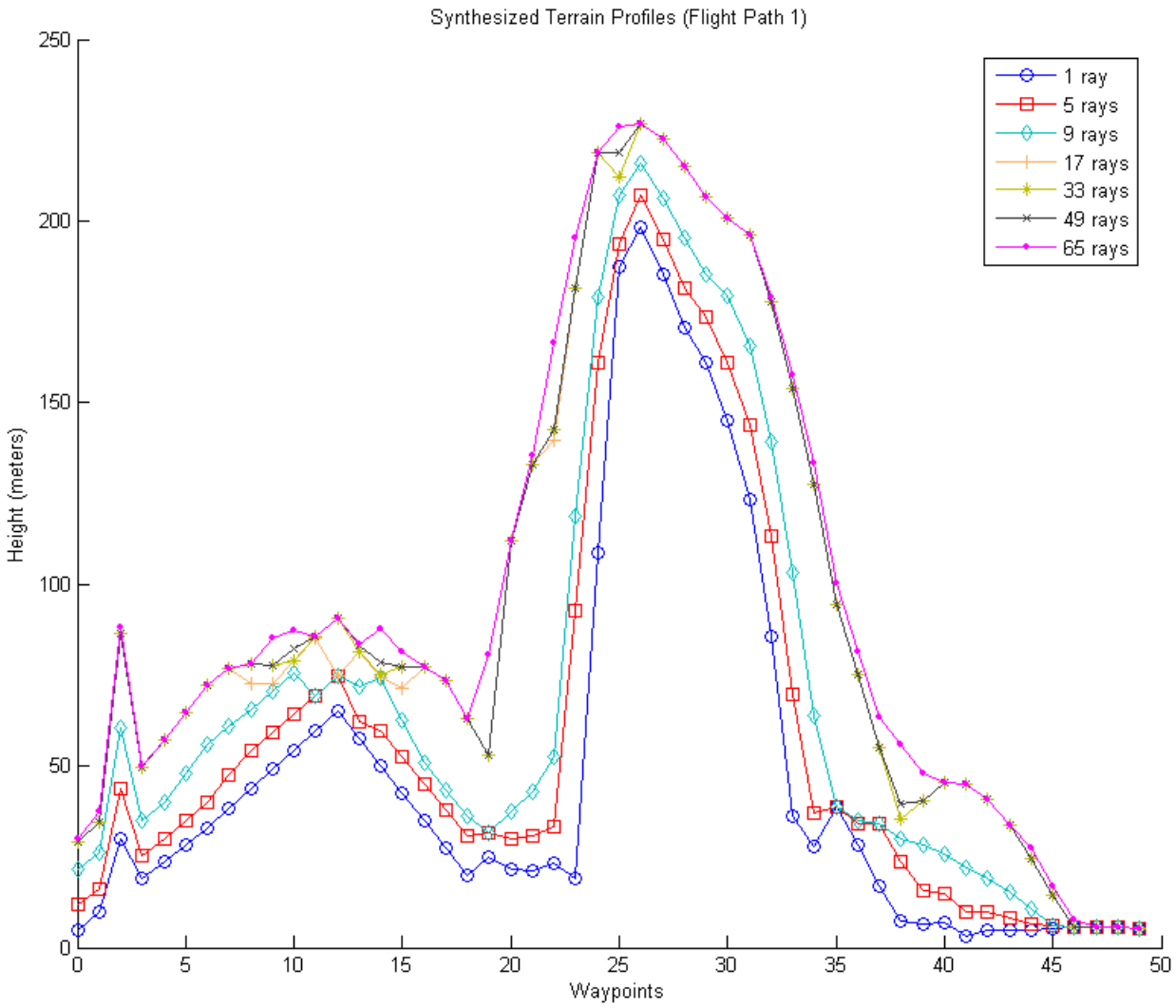



Figure 4.7. Synthesized Terrain Profiles

and 3 are shown in Figures 4.8 and 4.9. The Flight Path 2 data values are shown in Appendix Listing 4. This flight path is similar to the first flight path except that the altitude is 500 meters, the aircraft is heading eastward, and the aircraft makes an abrupt right bank to avoid colliding with the mountainside. Flight Path 3 (Appendix Listing 5) is different than the previous two flight paths in that it simulates an aircraft on final approach to landing. The aircraft starts at an altitude of 600 meters and descends rapidly to 200 meters on a westward course. There is no roll maneuvering during this simulation, but there is a pitch angle of +8° which decreases gradually to 0° at the end of the run. This pitch angle simulates the 'nose up' attitude common to commercial aircraft during approach.

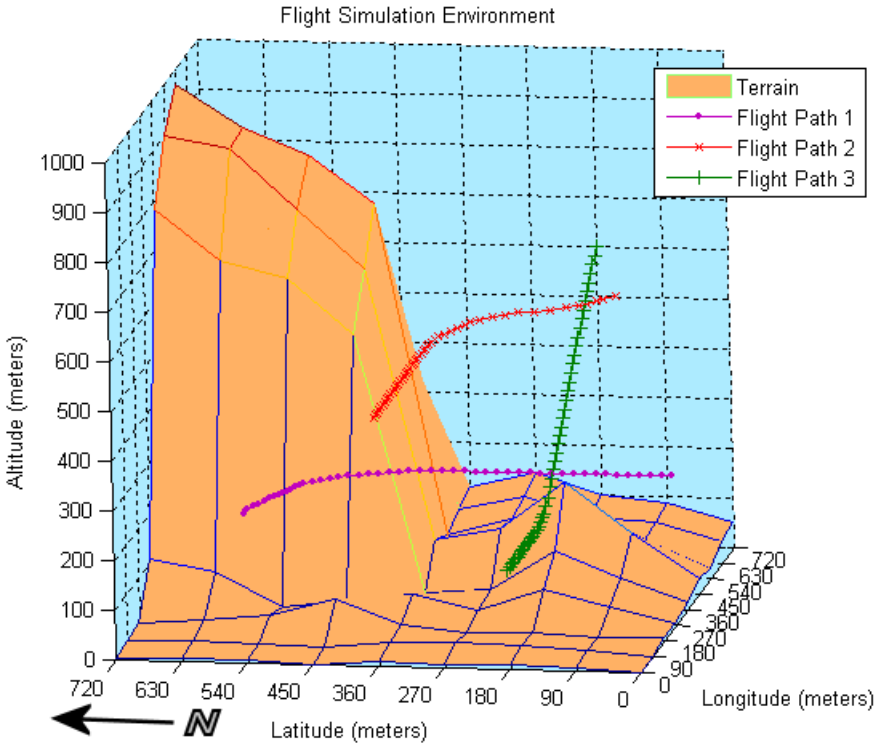



Figure 4.8. Flight Simulation Environment: View 2

## 4.6 Experiment 3: Results

Figure 4.10 shows a comparison of radar altitudes for the two models during the Flight Path 1 simulation. It also shows the aircraft's roll angle at each waypoint. There's a large

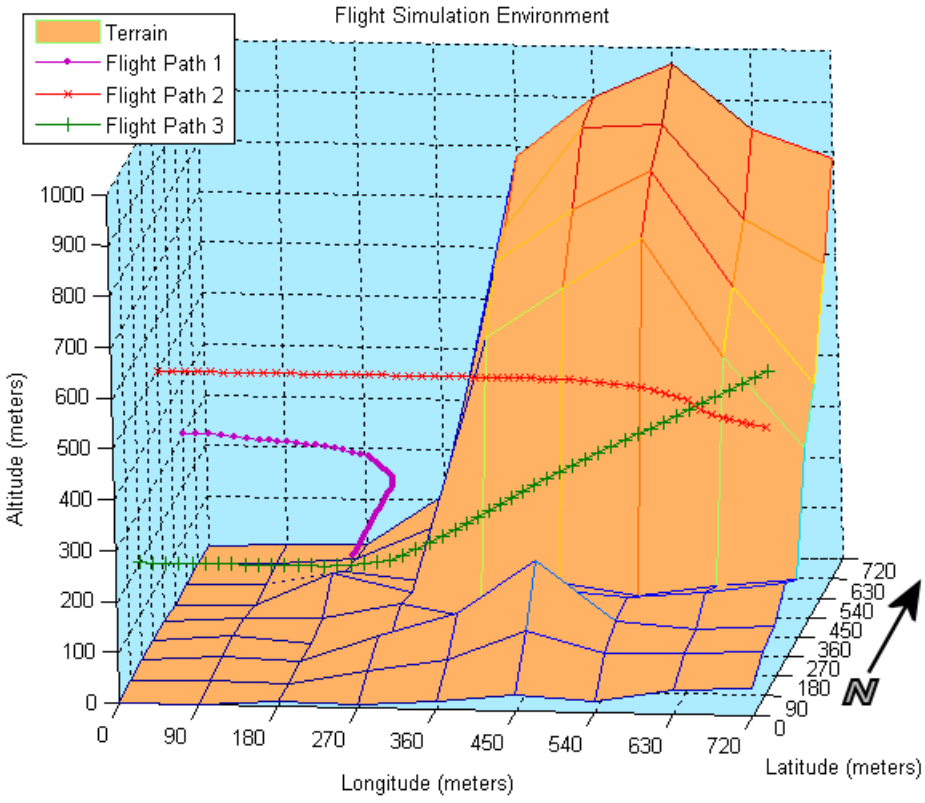



Figure 4.9. Flight Simulation Environment: View 3

deviation between the range values when the aircraft banks to avoid the mountainside. This is due to the fact that the "Ray Stream" model properly simulates the behavior of the radar altimeter beam during roll maneuvers and it correctly measures the distance to the mountainside instead of the plumb-bob distance to the terrain below the aircraft. The accuracy is improved by as much as 34% using our proposed model. Figure 4.11 shows the synthesized terrain profile comparison for the two models. Again, at least some deviation is noticeable during almost all roll maneuvers.

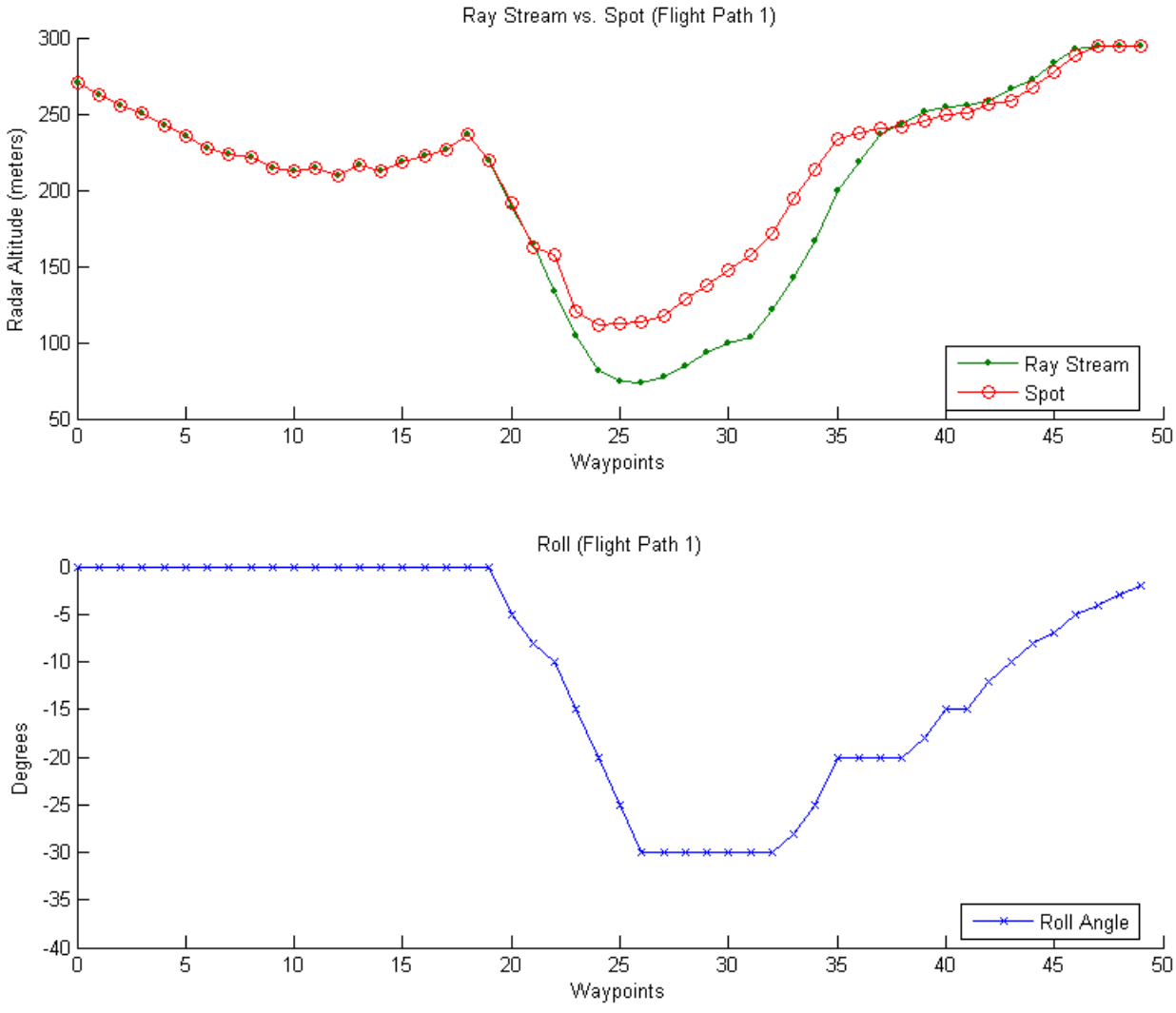



Figure 4.10. Radar Altitude Comparison: Flight Path 1

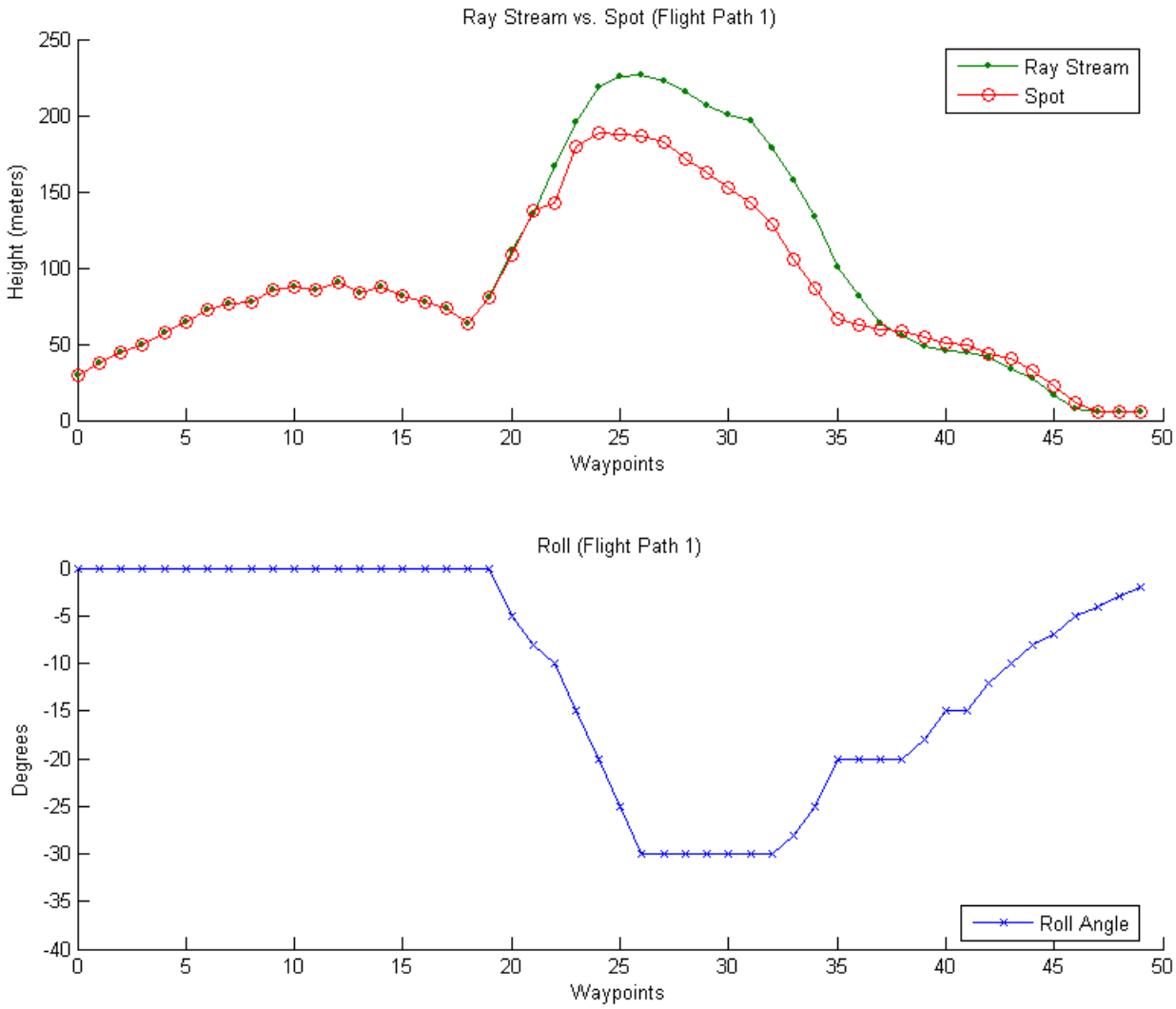



Figure 4.11. Synthesized Terrain Profile Comparison: Flight Path 1

Figure 4.12 shows the radar altitudes detected at each waypoint during the Flight Path 2 simulation. The "Spot" model deviates from our model's range values by as much as 50 meters during the right roll maneuver at Waypoints 30 - 43. This clearly shows the amount of error that can occur by not correctly simulating the characteristics of the radar altimeter. Accuracy is improved by as much as 30% using the "Ray Stream" model in this simulation. Figure 4.13 shows the deviation in the synthesized terrain profiles generated from the radar altitude data. It is quite possible that this much error will cause a false alarm from an integrity monitor if the "Spot" model is used.

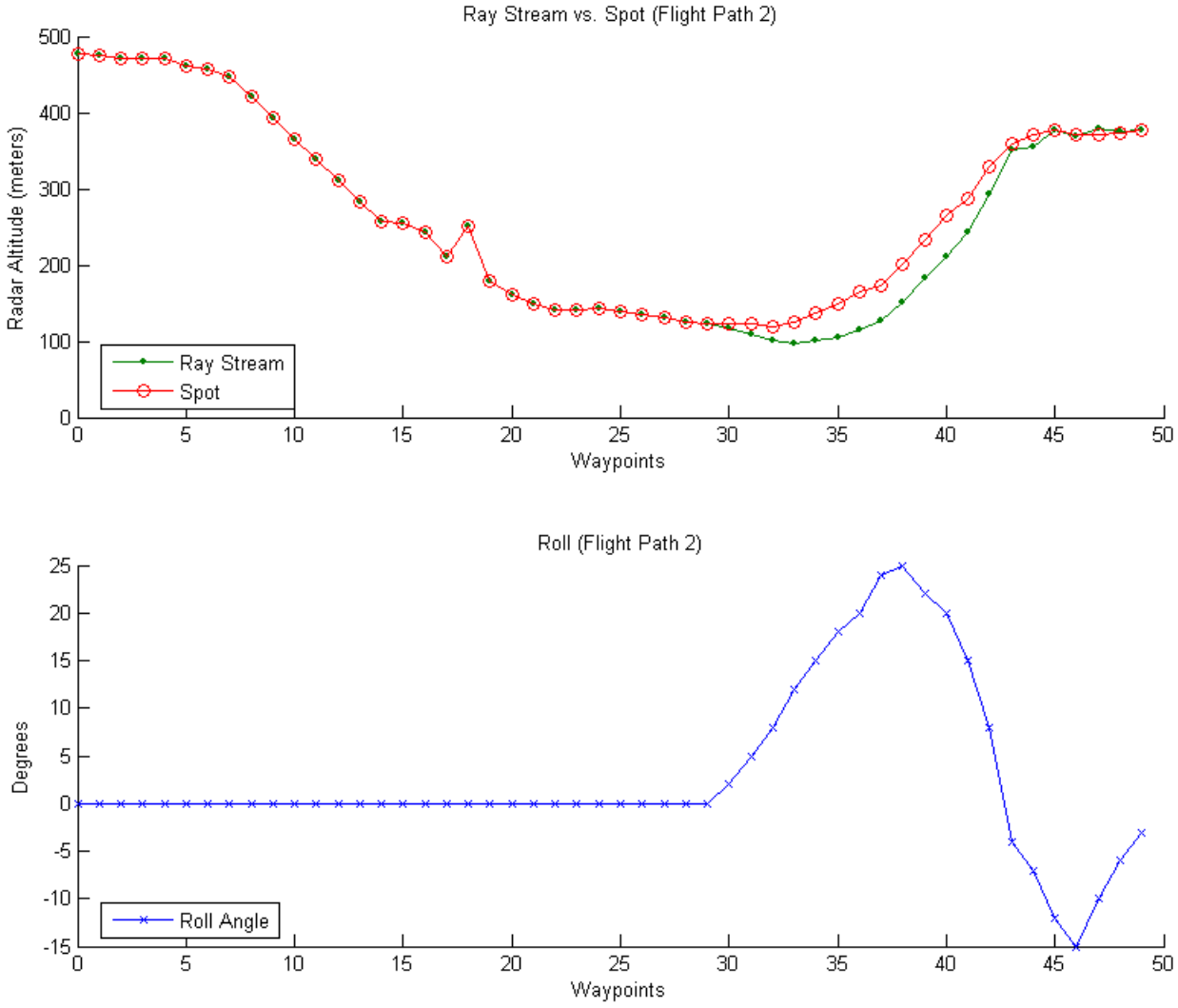



Figure 4.12. Radar Altitude Comparison: Flight Path 2

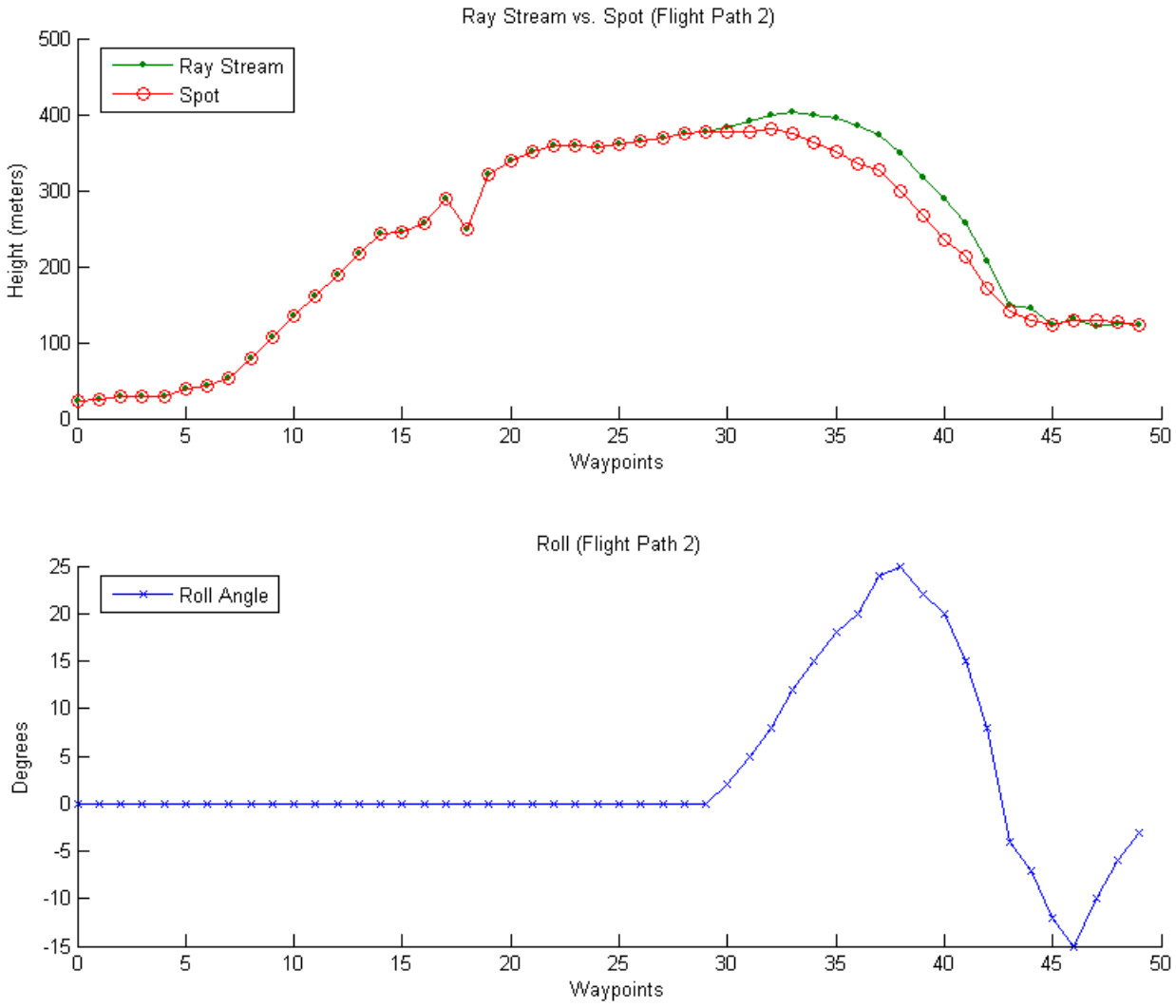



Figure 4.13. Synthesized Terrain Profile Comparison: Flight Path 2

The radar altitude values generated during the Flight Path 3 simulation are shown in Figure 4.14. The pitch angle at each waypoint is also shown. The deviation between the two models is not as noticeable here, but it is present and it coincides with the pitch maneuvers. The relatively small pitch angle can explain the smaller deviation between the range values. An initial deviation is more apparent in the synthesized terrain profile comparison (Figure 4.15). At Waypoint 3 the error is approximately 25 meters. This is probably not enough to trigger a false alarm from the integrity monitor, but it does show the need for correct modeling of the radar altimeter beam. Overall, accuracy is improved by as much as 6% using our model in this simulation.

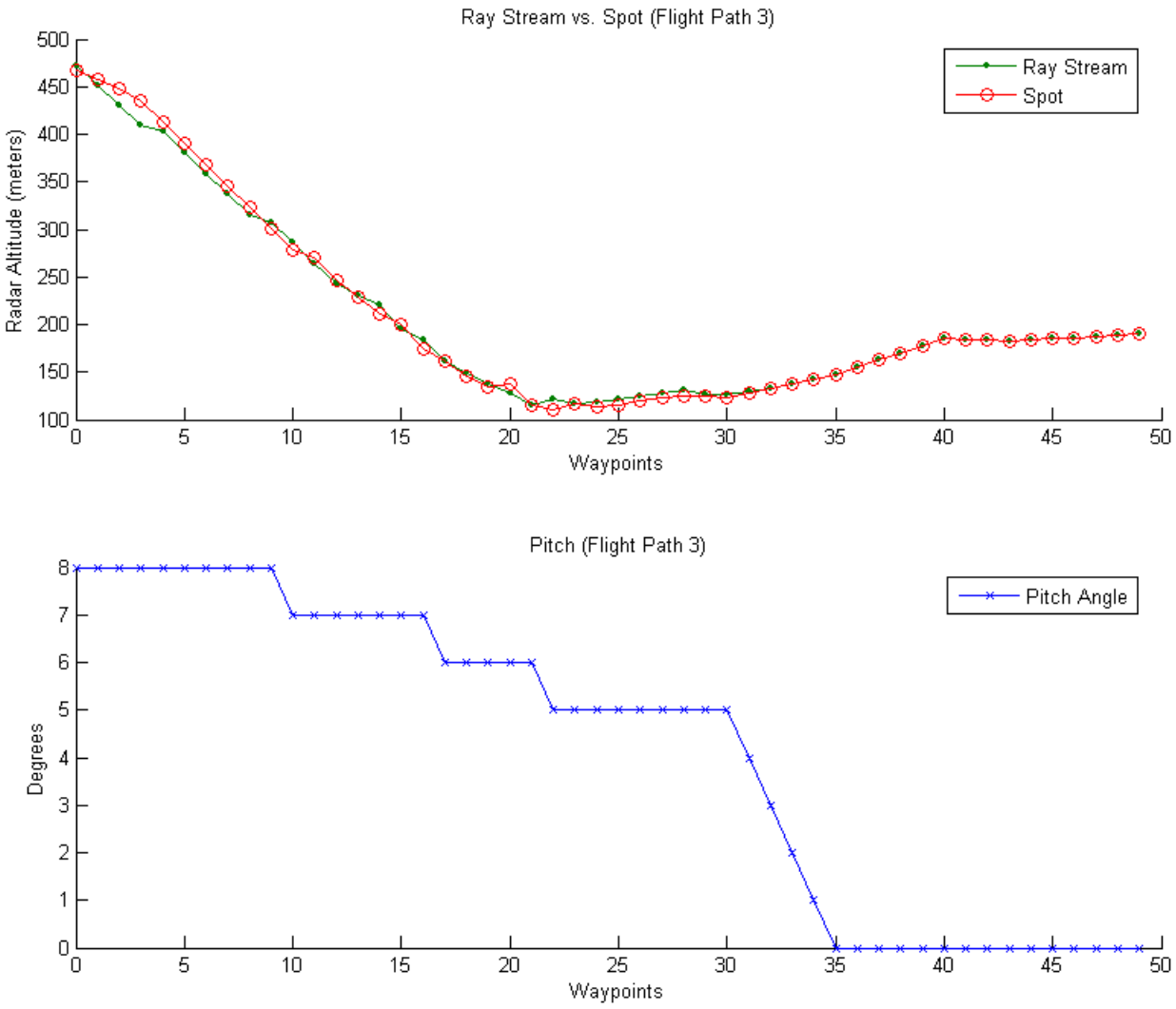



Figure 4.14. Radar Altitude Comparison: Flight Path 3

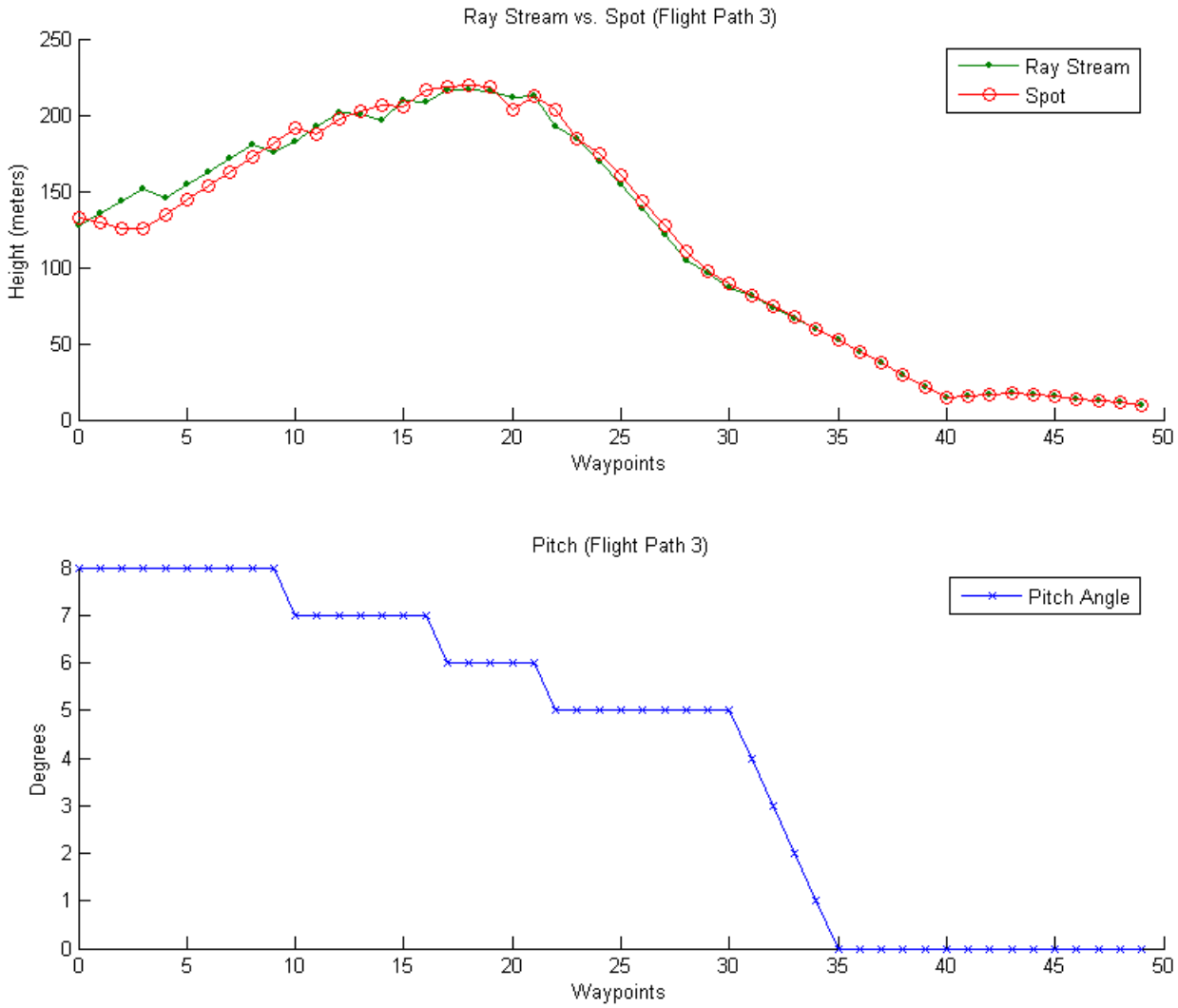



Figure 4.15. Synthesized Terrain Profile Comparison: Flight Path 3

# Chapter 5

# CONCLUSIONS

In this thesis we showed the benefits of Synthetic Vision Systems to flight safety and the need to make those systems more reliable through the use of terrain database integrity monitors. We demonstrated the benefits of stream processing on commercial graphics hardware using Brook for GPUs and presented an improved model for simulating a radar altimeter antenna beam using a streaming ray-casting method. We showed that ray casting could be used to solve scientific problems beyond its traditional use for rendering 3-D computer graphics. Through our experimentation, we were able to show the performance and accuracy improvements of our proposed "Ray Stream" model as compared to the "Spot" model used in previous terrain database integrity monitoring research.

The main advantages of the "Ray Stream" radar altimeter model are the processing speed provided by the GPU implementation and the increased accuracy obtained through proper modeling of the radar altimeter beam behavior during changes in aircraft attitude. Experiment 1 demonstrated that the performance of our streaming ray caster on the GPU was exceptional, with a worst-case execution time of 1.46 milliseconds for 65 rays intersecting 2048 triangles. Experiment 2 showed that the accuracy of our model improved as more rays were added to simulate the antenna beam. Finally, our model was shown to provide up to 34% more accuracy in synthesizing terrain distances than the previous model in Experiment 3.

The disadvantages of our proposal include the need to convert the digital elevation model into a terrain mesh in order to calculate the ray-triangle intersections. As higher-resolution DEMs become available the number of triangles needed to represent these large data sets could become prohibitive. Another disadvantage is the possibility for error when mapping world coordinates onto a Cartesian coordinate system. Since lines of longitude converge at the Earth's poles the distances between horizontal datum in the terrain databases varies, which makes the coordinate mapping task non-trivial.

Our work has the potential to improve flight safety if implemented in a terrain database integrity monitoring system. These systems rely on fast and accurate synthesis of terrain elevation data to compare against actual terrain measurements. Our model has the speed and accuracy to meet those needs. Greater accuracy in terrain height synthesis means fewer false alarms and therefore greater reliability for any Synthetic Vision System that depends on this terrain database integrity monitor. A more reliable SVS means better situational awareness for the pilot and improved flight safety overall.

# Chapter 6

# FUTURE WORK

There are many potential areas for future work with our research. Foremost among these is the requirement to test our "Ray Stream" radar altimeter model as part of a terrain database integrity monitor with actual aircraft systems such as a radar altimeter. Also, with some slight modifications our streaming ray-casting model should work well in simulating newer terrain detection systems such as Airborne Laser Scanners (ALS). There is also the potential for improving the speed and accuracy of our model through the use of improved graphics hardware and the new version of Brook for GPUs (v0.5) which is currently in the beta stage. Other stream-computing systems can also be utilized such as Brook+, which is an enhanced version of Brook for GPUs created by AMD for use with their Compute Abstraction Layer (CAL) architecture, and NVIDIA's Compute Unified Driver Architecture (CUDA) parallel programming model [34, 35]. Finally, the application of a triangulated irregular network (TIN) to store digital elevation data in a more compact format for our integrity monitor model can be explored.

## REFERENCES


[1] M. Uijt de Haag, J. Campbell, R. Gray, "A Terrain Database Integrity Monitor for Synthetic Vision Systems", *19th Digital Avionics Systems Conference Proceedings*, Vol. 1, pp. 2.C.3-1 - 2.C.3-8, October 2000.

[2] S. D. Harrah, W. R. Jones, C. W. Erickson, J. H. White, "The NASA Approach to Realize a Sensor Enhanced-Synthetic Vision System (SE-SVS)", *21st Digital Avionics Systems Conference Proceedings*, Vol. 2, pp. 11.A.4-1 - 11.A.4-11, October 2002.

[3] Federal Aviation Administration, *General Aviation CFIT Joint Safety Implementation Team Final Report*, February 2000.

[4] Universal Avionics Systems Corporation, *Vision-1$^{TM}$Synthetic Vision System Brochure*, October 2006.

[5] S. D. Young, S. D. Harrah, M. Uijt de Haag, "Real-time Integrity Monitoring of Stored Geo-spatial Data Using Forward-looking Remote Sensing Technology", *21st Digital Avionics Systems Conference Proceedings*, Vol. 2, pp. 11.D.1-1 - 11.D.1-10, October 2002.

[6] M. Uijt de Haag, S. Young, J. Sayre, J. Campbell, A. Vadlamani, "DEM Integrity Monitor Experiment (DIME) Flight Test Results", *SPIE 16th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Controls, AeroSense*, April 2002.

[7] J. L. Campbell, M. Uijt de Haag, A. Vadlamani, S. Young, "The Application of LiDAR to Synthetic Vision System Integrity", *22nd Digital Avionics Systems Conference Proceedings*, Vol. 2, pp. 9.C.2-1 - 9.C.2-7, October 2003.

[8] Federal Aviation Administration, *Advisory Circular No. 23-26, Synthetic Vision and Pathway Depictions on the Primary Flight Display*, December 2005.

[9] J. J. Arthur III, L. J. Prinzel III, L. J. Kramer, R. E. Bailey, R. V. Parrish, "CFIT Prevention Using Synthetic Vision", *Enhanced and Synthetic Vision 2003, Proceedings of the SPIE*, Vol. 5081, pp. 146-157, September 2003.

[10] J. Schiefele, L. May, J. Pfister, H. Raabe, C. Schmalz, K.-U. Dörr, W. Kubbat, "Safety Relevant Navigation and Certifiable Databases for 3D Synthetic Vision Systems", *17th Digital Avionics Systems Conference Proceedings*, Vol. 2, pp. J21-1 - J21-8, October 1998.

[11] K. Arthur, "Effects of Field of View on Task Performance with Head-mounted Displays", *Conference on Human Factors in Computing Systems, Conference Companion*, pp. 29-30, 1996.

[12] M. Uijt de Haag, J. Sayre, J. Campbell, S. D. Young, R. A. Gray, "Terrain Database Integrity Monitoring for Synthetic Vision Systems", *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 41, Issue 2, pp. 386-406, April 2005.

[13] A. K. Barrows, "Registration of Synthetic Vision Imagery: Issues and Pitfalls", *21st Digital Avionics Systems Conference Proceedings*, Vol. 2, pp. 11.C.3-1 - 11.C.3-10, October 2002.

[14] U. S. Department of the Interior, U. S. Geological Survey, National Mapping Division, *Standards For Digital Elevation Models*, August 1997.

[15] M. Kayton, W. R. Fried, *Avionics Navigation Systems, 2nd Edition*, John Wiley & Sons, New York, 1997.

[16] D. Venable, J. Campbell, M. Uijt de Haag, "Feature Extraction and Separation in Airborne Laser Scanner Terrain Integrity Monitors", *24th Digital Avionics Systems Conference Proceedings*, Vol. 1, pp. 4.E.3-1 - 4.E.3-11, October 2005.

[17] R. A. Gray, "Inflight Detection of Errors for Enhanced Aircraft Flight Safety and Vertical Accuracy Improvement Using Digital Terrain Elevation Data with an Inertial Navigation System, Global Positioning System and Radar Altimeter", *Dissertation*, The Ohio University, June 1999.

[18] J. L. Campbell, M. Uijt de Haag, "Assessment of Radar Altimeter Performance When Used for Integrity Monitoring in a Synthetic Vision System", *20th Digital Avionics Systems Conference Proceedings*, Vol. 1, pp. 2.C.3-1 - 2.C.3-9, October 2001.

[19] A. Vadlamani, M. Uijt de Haag, "Improving the Detection Capability of Spatial Failure Modes Using Downward-looking Sensors in Terrain Database Integrity Monitors", *22nd Digital Avionics Systems Conference Proceedings*, Vol. 2, pp. 9.C.5-1 - 9.C.5-12, October 2003.

[20] S. Kakarlapudi, M. Uijt de Haag, "The Application of Image Analysis Techniques to Forward Looking Terrain Database Integrity Monitoring", *23rd Digital Avionics Systems Conference Proceedings*, Vol. 1, pp. 4.C.6-1 - 4.C.6-12, October 2004.

[21] A. K. Vadlamani, M. Uijt de Haag, "Improved Downward-looking Terrain Database Integrity Monitor and Terrain Navigation", *2004 IEEE Aerospace Conference Proceedings*, Vol. 3, pp. 1594-1607, March 2004.

[22] A. Vadlamani, M. Uijt de Haag, "A 3-D Spatial Integrity Monitor for Terrain Databases", *23rd Digital Avionics Systems Conference Proceedings*, Vol. 1, pp. 4.C.2-1 - 4.C.2-13, October 2004.

[23] A. Vadlamani, M. Smearcheck, M. Uijt de Haag, "Preliminary Design and Analysis of a LiDAR Based Obstacle Detection System", *24th Digital Avionics Systems Conference Proceedings*, Vol. 1, pp. 6.B.2-1 - 6.B.2-14, October 2005.

[24] M. Uijt de Haag, A. Vadlamani, J. L. Campbell, J. Dickman, "Application of Laser Range Scanner Based Terrain Referenced Navigation Systems for Aircraft Guidance", *3rd IEEE International Workshop on Electronic Design Proceedings*, pp. 269-274, January 2006.

[25] M. Uijt de Haag, J. Sayre, J. Campbell, S. Young, R. Gray, "Flight Test Results of a Synthetic Vision Elevation Database Integrity Monitor", *15th International Symposium on Aerospace/Defense Sensing, Simulation, and Controls, Proceedings of SPIE*, Orlando, FL, April 2001.

[26] T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan, "Ray Tracing on Programmable Graphics Hardware", *29th International Conference on Computer Graphics and Interactive Techniques Proceedings*, pp. 703-712, 2002.

[27] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware", *ACM Transactions on Graphics*, Vol. 23, Issue 3, pp. 777-786, August 2004.

[28] G. S. Owen, "Overview of Ray Tracing", *http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace1.htm*, June 1999. Ray Tracing images produced by Michael Sweeney, University of Waterloo, © 1984.

[29] N. A. Carr, J. D. Hall, J. C. Hart, "The Ray Engine", *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware Proceedings*, pp. 37-46, September 2002.

[30] D. Sunday, "Intersections of Rays, Segments, Planes and Triangles in 3D", *http://www.softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm*, © 2001.

[31] G. S. Owen, "3D Rotation", *http://www.siggraph.org/education/materials/HyperGraph/modeling/mod_tran/3drota.htm*, April 1998.

[32] T. Moller, B. Trumbore, "Fast, Minimum Storage Ray-Triangle Intersection", *Journal of Graphics Tools*, 2(1), pp. 21-28, 1997.

[33] F. S. Hill, "The Pleasures of 'Perp Dot' Products", *Graphics Gems IV*, pp. 138-148, 1994.

[34] M. Houston, "Introduction to the AMD Stream SDK", *Beyond Programmable Shading: Fundamentals, SIGGRAPH 2008 course notes*, August 2008.

[35] D. Luebke, "CUDA Fundamentals", *Beyond Programmable Shading: Fundamentals, SIGGRAPH 2008 course notes*, August 2008.

# APPENDIX: ADDITIONAL SOURCE FILES

Listing 1. Terrain Data

```
//terrain128.h
//Author: Sean McKeon (2009)

#ifndef __TERRAIN128_H__
#define __TERRAIN128_H__

#define NUM_TRIANGLES 128

enum Vertex_e {V0, V1, V2, NUM_VERTICES};
enum Coords_e {X, Y, Z, NUM_COORDS};

// *** Define 4 grids of 32 triangles ***
// Upper Left (UL), Upper Right (UR)
// Lower Left (LL), Lower Right (LR)

static const float terrain[NUM_TRIANGLES][NUM_VERTICES][NUM_COORDS] = {
    {{  0.0,   0.0,   0.0}, { 90.0,  90.0,   8.0}, {  0.0,  90.0,   5.0}}, //Tri 000 LL
    {{  0.0,   0.0,   0.0}, { 90.0,   0.0,   1.0}, { 90.0,  90.0,   8.0}}, //Tri 001
    {{ 90.0,   0.0,   1.0}, {180.0,  90.0,   5.0}, { 90.0,  90.0,   8.0}}, //Tri 002
    {{ 90.0,   0.0,   1.0}, {180.0,   0.0,   8.0}, {180.0,  90.0,   5.0}}, //Tri 003
    {{180.0,   0.0,   8.0}, {270.0,  90.0,  33.0}, {180.0,  90.0,   5.0}}, //Tri 004
    {{180.0,   0.0,   8.0}, {270.0,   0.0,   5.0}, {270.0,  90.0,  33.0}}, //Tri 005
    {{270.0,   0.0,   5.0}, {360.0,  90.0,  60.0}, {270.0,  90.0,  33.0}}, //Tri 006
    {{270.0,   0.0,   5.0}, {360.0,   0.0,  15.0}, {360.0,  90.0,  60.0}}, //Tri 007

    {{  0.0,  90.0,   5.0}, { 90.0, 180.0,  17.0}, {  0.0, 180.0,   9.0}}, //Tri 008
    {{  0.0,  90.0,   5.0}, { 90.0,  90.0,   8.0}, { 90.0, 180.0,  17.0}}, //Tri 009
    {{ 90.0,  90.0,   8.0}, {180.0, 180.0,  12.0}, { 90.0, 180.0,  17.0}}, //Tri 010
    {{ 90.0,  90.0,   8.0}, {180.0,  90.0,   5.0}, {180.0, 180.0,  12.0}}, //Tri 011
    {{180.0,  90.0,   5.0}, {270.0, 180.0,  65.0}, {180.0, 180.0,  12.0}}, //Tri 012
    {{180.0,  90.0,   5.0}, {270.0,  90.0,  33.0}, {270.0, 180.0,  65.0}}, //Tri 013
    {{270.0,  90.0,  33.0}, {360.0, 180.0, 113.0}, {270.0, 180.0,  65.0}}, //Tri 014
    {{270.0,  90.0,  33.0}, {360.0,  90.0,  60.0}, {360.0, 180.0, 113.0}}, //Tri 015

    {{  0.0, 180.0,   9.0}, { 90.0, 270.0,  21.0}, {  0.0, 270.0,   7.0}}, //Tri 016
    {{  0.0, 180.0,   9.0}, { 90.0, 180.0,  17.0}, { 90.0, 270.0,  21.0}}, //Tri 017
    {{ 90.0, 180.0,  17.0}, {180.0, 270.0,   8.0}, { 90.0, 270.0,  21.0}}, //Tri 018
    {{ 90.0, 180.0,  17.0}, {180.0, 180.0,  12.0}, {180.0, 270.0,   8.0}}, //Tri 019
    {{180.0, 180.0,  12.0}, {270.0, 270.0,  20.0}, {180.0, 270.0,   8.0}}, //Tri 020
    {{180.0, 180.0,  12.0}, {270.0, 180.0,  65.0}, {270.0, 270.0,  20.0}}, //Tri 021
    {{270.0, 180.0,  65.0}, {360.0, 270.0,  31.0}, {270.0, 270.0,  20.0}}, //Tri 022
    {{270.0, 180.0,  65.0}, {360.0, 180.0, 113.0}, {360.0, 270.0,  31.0}}, //Tri 023

    {{  0.0, 270.0,   7.0}, { 90.0, 360.0,  16.0}, {  0.0, 360.0,   9.0}}, //Tri 024
    {{  0.0, 270.0,   7.0}, { 90.0, 270.0,  21.0}, { 90.0, 360.0,  16.0}}, //Tri 025
    {{ 90.0, 270.0,  21.0}, {180.0, 360.0,  20.0}, { 90.0, 360.0,  16.0}}, //Tri 026
    {{ 90.0, 270.0,  21.0}, {180.0, 270.0,   8.0}, {180.0, 360.0,  20.0}}, //Tri 027
    {{180.0, 270.0,   8.0}, {270.0, 360.0,  50.0}, {180.0, 360.0,  20.0}}, //Tri 028
    {{180.0, 270.0,   8.0}, {270.0, 270.0,  20.0}, {270.0, 360.0,  50.0}}, //Tri 029
    {{270.0, 270.0,  20.0}, {360.0, 360.0,  23.0}, {270.0, 360.0,  50.0}}, //Tri 030
    {{270.0, 270.0,  20.0}, {360.0, 270.0,  31.0}, {360.0, 360.0,  23.0}}, //Tri 031 LL

    {{360.0,   0.0,  15.0}, {450.0,  90.0, 120.0}, {360.0,  90.0,  60.0}}, //Tri 000 LR
    {{360.0,   0.0,  15.0}, {450.0,   0.0,  32.0}, {450.0,  90.0, 120.0}}, //Tri 001
    {{450.0,   0.0,  32.0}, {540.0,  90.0,  76.0}, {450.0,  90.0, 120.0}}, //Tri 002
    {{450.0,   0.0,  32.0}, {540.0,   0.0,  22.0}, {540.0,  90.0,  76.0}}, //Tri 003
    {{540.0,   0.0,  22.0}, {630.0,  90.0,  80.0}, {540.0,  90.0,  76.0}}, //Tri 004
    {{540.0,   0.0,  22.0}, {630.0,   0.0,  45.0}, {630.0,  90.0,  80.0}}, //Tri 005
    {{630.0,   0.0,  45.0}, {720.0,  90.0,  86.0}, {630.0,  90.0,  80.0}}, //Tri 006
    {{630.0,   0.0,  45.0}, {720.0,   0.0,  52.0}, {720.0,  90.0,  86.0}}, //Tri 007

    {{360.0,  90.0,  60.0}, {450.0, 180.0, 220.0}, {360.0, 180.0, 113.0}}, //Tri 008
```

```
{{360.0,  90.0,  60.0}, {450.0,  90.0, 120.0}, {450.0, 180.0, 220.0}}, //Tri 009
{{450.0,  90.0, 120.0}, {540.0, 180.0, 102.0}, {450.0, 180.0, 220.0}}, //Tri 010
{{450.0,  90.0, 120.0}, {540.0,  90.0,  76.0}, {540.0, 180.0, 102.0}}, //Tri 011
{{540.0,  90.0,  76.0}, {630.0, 180.0,  88.0}, {540.0, 180.0, 102.0}}, //Tri 012
{{540.0,  90.0,  76.0}, {630.0,  90.0,  80.0}, {630.0, 180.0,  88.0}}, //Tri 013
{{630.0,  90.0,  80.0}, {720.0, 180.0,  99.0}, {630.0, 180.0,  88.0}}, //Tri 014
{{630.0,  90.0,  80.0}, {720.0,  90.0,  86.0}, {720.0, 180.0,  99.0}}, //Tri 015

{{360.0, 180.0, 113.0}, {450.0, 270.0, 123.0}, {360.0, 270.0,  31.0}}, //Tri 016
{{360.0, 180.0, 113.0}, {450.0, 180.0, 220.0}, {450.0, 270.0, 123.0}}, //Tri 017
{{450.0, 180.0, 220.0}, {540.0, 270.0, 111.0}, {450.0, 270.0, 123.0}}, //Tri 018
{{450.0, 180.0, 220.0}, {540.0, 180.0, 102.0}, {540.0, 270.0, 111.0}}, //Tri 019
{{540.0, 180.0, 102.0}, {630.0, 270.0, 124.0}, {540.0, 270.0, 111.0}}, //Tri 020
{{540.0, 180.0, 102.0}, {630.0, 180.0,  88.0}, {630.0, 270.0, 124.0}}, //Tri 021
{{630.0, 180.0,  88.0}, {720.0, 270.0, 144.0}, {630.0, 270.0, 124.0}}, //Tri 022
{{630.0, 180.0,  88.0}, {720.0, 180.0,  99.0}, {720.0, 270.0, 144.0}}, //Tri 023

{{360.0, 270.0,  31.0}, {450.0, 360.0,  99.0}, {360.0, 360.0,  23.0}}, //Tri 024
{{360.0, 270.0,  31.0}, {450.0, 270.0, 123.0}, {450.0, 360.0,  99.0}}, //Tri 025
{{450.0, 270.0, 123.0}, {540.0, 360.0,  77.0}, {450.0, 360.0,  99.0}}, //Tri 026
{{450.0, 270.0, 123.0}, {540.0, 270.0, 111.0}, {540.0, 360.0,  77.0}}, //Tri 027
{{540.0, 270.0, 111.0}, {630.0, 360.0,  97.0}, {540.0, 360.0,  77.0}}, //Tri 028
{{540.0, 270.0, 111.0}, {630.0, 270.0, 124.0}, {630.0, 360.0,  97.0}}, //Tri 029
{{630.0, 270.0, 124.0}, {720.0, 360.0, 112.0}, {630.0, 360.0,  97.0}}, //Tri 030
{{630.0, 270.0, 124.0}, {720.0, 270.0, 144.0}, {720.0, 360.0, 112.0}}, //Tri 031 LR

{{  0.0,  360.0,  9.0}, { 90.0, 450.0,   3.0}, {  0.0, 450.0,   1.0}}, //Tri 000 UL
{{  0.0,  360.0,  9.0}, { 90.0, 360.0,  16.0}, { 90.0, 450.0,   3.0}}, //Tri 001
{{ 90.0,  360.0, 16.0}, {180.0, 450.0,  70.0}, { 90.0, 450.0,   3.0}}, //Tri 002
{{ 90.0,  360.0, 16.0}, {180.0, 360.0,  20.0}, {180.0, 450.0,  70.0}}, //Tri 003
{{180.0,  360.0, 20.0}, {270.0, 450.0,  36.0}, {180.0, 450.0,  70.0}}, //Tri 004
{{180.0,  360.0, 20.0}, {270.0, 360.0,  50.0}, {270.0, 450.0,  36.0}}, //Tri 005
{{270.0,  360.0, 50.0}, {360.0, 450.0, 540.0}, {270.0, 450.0,  36.0}}, //Tri 006
{{270.0,  360.0, 50.0}, {360.0, 360.0,  23.0}, {360.0, 450.0, 540.0}}, //Tri 007

{{  0.0, 450.0,   1.0}, { 90.0, 540.0,   7.0}, {  0.0, 540.0,   5.0}}, //Tri 008
{{  0.0, 450.0,   1.0}, { 90.0, 450.0,   3.0}, { 90.0, 540.0,   7.0}}, //Tri 009
{{ 90.0, 450.0,   3.0}, {180.0, 540.0,  35.0}, { 90.0, 540.0,   7.0}}, //Tri 010
{{ 90.0, 450.0,   3.0}, {180.0, 450.0,  70.0}, {180.0, 540.0,  35.0}}, //Tri 011
{{180.0, 450.0,  70.0}, {270.0, 540.0,  20.0}, {180.0, 540.0,  35.0}}, //Tri 012
{{180.0, 450.0,  70.0}, {270.0, 450.0,  36.0}, {270.0, 540.0,  20.0}}, //Tri 013
{{270.0, 450.0,  36.0}, {360.0, 540.0, 650.0}, {270.0, 540.0,  20.0}}, //Tri 014
{{270.0, 450.0,  36.0}, {360.0, 450.0, 540.0}, {360.0, 540.0, 650.0}}, //Tri 015

{{  0.0, 540.0,   5.0}, { 90.0, 630.0,  10.0}, {  0.0, 630.0,   8.0}}, //Tri 016
{{  0.0, 540.0,   5.0}, { 90.0, 540.0,   7.0}, { 90.0, 630.0,  10.0}}, //Tri 017
{{ 90.0, 540.0,   7.0}, {180.0, 630.0,  21.0}, { 90.0, 630.0,  10.0}}, //Tri 018
{{ 90.0, 540.0,   7.0}, {180.0, 540.0,  35.0}, {180.0, 630.0,  21.0}}, //Tri 019
{{180.0, 540.0,  35.0}, {270.0, 630.0,  88.0}, {180.0, 630.0,  21.0}}, //Tri 020
{{180.0, 540.0,  35.0}, {270.0, 540.0,  20.0}, {270.0, 630.0,  88.0}}, //Tri 021
{{270.0, 540.0,  20.0}, {360.0, 630.0, 680.0}, {270.0, 630.0,  88.0}}, //Tri 022
{{270.0, 540.0,  20.0}, {360.0, 540.0, 650.0}, {360.0, 630.0, 680.0}}, //Tri 023

{{  0.0, 630.0,   8.0}, { 90.0, 720.0,   9.0}, {  0.0, 720.0,   3.0}}, //Tri 024
{{  0.0, 630.0,   8.0}, { 90.0, 630.0,  10.0}, { 90.0, 720.0,   9.0}}, //Tri 025
{{ 90.0, 630.0,  10.0}, {180.0, 720.0,  12.0}, { 90.0, 720.0,   9.0}}, //Tri 026
{{ 90.0, 630.0,  10.0}, {180.0, 630.0,  21.0}, {180.0, 720.0,  12.0}}, //Tri 027
{{180.0, 630.0,  21.0}, {270.0, 720.0, 111.0}, {180.0, 720.0,  12.0}}, //Tri 028
{{180.0, 630.0,  21.0}, {270.0, 630.0,  88.0}, {270.0, 720.0, 111.0}}, //Tri 029
{{270.0, 630.0,  88.0}, {360.0, 720.0, 780.0}, {270.0, 720.0, 111.0}}, //Tri 030
{{270.0, 630.0,  88.0}, {360.0, 630.0, 680.0}, {360.0, 720.0, 780.0}}, //Tri 031 UL

{{360.0,  360.0, 23.0}, {450.0, 450.0, 640.0}, {360.0, 450.0, 540.0}}, //Tri 000 UR
{{360.0,  360.0, 23.0}, {450.0, 360.0,  99.0}, {450.0, 450.0, 640.0}}, //Tri 001
{{450.0,  360.0, 99.0}, {540.0, 450.0, 740.0}, {450.0, 450.0, 640.0}}, //Tri 002
{{450.0,  360.0, 99.0}, {540.0, 360.0,  77.0}, {540.0, 450.0, 740.0}}, //Tri 003
{{540.0,  360.0, 77.0}, {630.0, 450.0, 510.0}, {540.0, 450.0, 740.0}}, //Tri 004
{{540.0,  360.0, 77.0}, {630.0, 360.0,  97.0}, {630.0, 450.0, 510.0}}, //Tri 005
```

```
    {{630.0,  360.0, 97.0}, {720.0, 450.0, 333.0}, {630.0, 450.0, 510.0}}, //Tri 006
    {{630.0,  360.0, 97.0}, {720.0, 360.0, 112.0}, {720.0, 450.0, 333.0}}, //Tri 007

    {{360.0, 450.0, 540.0}, {450.0, 540.0, 756.0}, {360.0, 540.0, 650.0}}, //Tri 008
    {{360.0, 450.0, 540.0}, {450.0, 450.0, 640.0}, {450.0, 540.0, 756.0}}, //Tri 009
    {{450.0, 450.0, 640.0}, {540.0, 540.0, 835.0}, {450.0, 540.0, 756.0}}, //Tri 010
    {{450.0, 450.0, 640.0}, {540.0, 450.0, 740.0}, {540.0, 540.0, 835.0}}, //Tri 011
    {{540.0, 450.0, 740.0}, {630.0, 540.0, 611.0}, {540.0, 540.0, 835.0}}, //Tri 012
    {{540.0, 450.0, 740.0}, {630.0, 450.0, 510.0}, {630.0, 540.0, 611.0}}, //Tri 013
    {{630.0, 450.0, 510.0}, {720.0, 540.0, 420.0}, {630.0, 540.0, 611.0}}, //Tri 014
    {{630.0, 450.0, 510.0}, {720.0, 450.0, 333.0}, {720.0, 540.0, 420.0}}, //Tri 015

    {{360.0, 540.0, 650.0}, {450.0, 630.0, 877.0}, {360.0, 630.0, 680.0}}, //Tri 016
    {{360.0, 540.0, 650.0}, {450.0, 540.0, 756.0}, {450.0, 630.0, 877.0}}, //Tri 017
    {{450.0, 540.0, 756.0}, {540.0, 630.0, 888.0}, {450.0, 630.0, 877.0}}, //Tri 018
    {{450.0, 540.0, 756.0}, {540.0, 540.0, 835.0}, {540.0, 630.0, 888.0}}, //Tri 019
    {{540.0, 540.0, 835.0}, {630.0, 630.0, 703.0}, {540.0, 630.0, 888.0}}, //Tri 020
    {{540.0, 540.0, 835.0}, {630.0, 540.0, 611.0}, {630.0, 630.0, 703.0}}, //Tri 021
    {{630.0, 540.0, 611.0}, {720.0, 630.0, 620.0}, {630.0, 630.0, 703.0}}, //Tri 022
    {{630.0, 540.0, 611.0}, {720.0, 540.0, 420.0}, {720.0, 630.0, 620.0}}, //Tri 023

    {{360.0, 630.0, 680.0}, {450.0, 720.0, 900.0}, {360.0, 720.0, 780.0}}, //Tri 024
    {{360.0, 630.0, 680.0}, {450.0, 630.0, 877.0}, {450.0, 720.0, 900.0}}, //Tri 025
    {{450.0, 630.0, 877.0}, {540.0, 720.0, 970.0}, {450.0, 720.0, 900.0}}, //Tri 026
    {{450.0, 630.0, 877.0}, {540.0, 630.0, 888.0}, {540.0, 720.0, 970.0}}, //Tri 027
    {{540.0, 630.0, 888.0}, {630.0, 720.0, 843.0}, {540.0, 720.0, 970.0}}, //Tri 028
    {{540.0, 630.0, 888.0}, {630.0, 630.0, 703.0}, {630.0, 720.0, 843.0}}, //Tri 029
    {{630.0, 630.0, 703.0}, {720.0, 720.0, 788.0}, {630.0, 720.0, 843.0}}, //Tri 030
    {{630.0, 630.0, 703.0}, {720.0, 630.0, 620.0}, {720.0, 720.0, 788.0}}, //Tri 031 UR
};

#endif
```

Listing 2. Flight Path 1 Data (With Error @ Waypoint 2)

```
//flightpath1.h
//Author: Sean McKeon (2009)

#ifndef __FLIGHTPATH1_H__
#define __FLIGHTPATH1_H__

#define NUM_WAYPOINTS 50

typedef struct flightpath_t{
    double heading;   //degrees (North = 0 degrees)
    double pitch;     //degrees (+ is pitch up)
    double roll;      //degrees (+ is bank right)
    double longitude; //meters
    double latitude;  //meters
    double altitude;  //meters
} FlightPath;

static const FlightPath flightpath[NUM_WAYPOINTS] = {
    {   0.0,   0.0,   0.0,  270.000,    0.000,  300.000}, //Wpt 000
    {   0.0,   0.0,   0.0,  270.000,   15.000,  300.000}, //Wpt 001
    {   0.0,   0.0,   0.0,  360.000,   30.000,  300.000}, //Wpt 002
    {   0.0,   0.0,   0.0,  270.000,   45.000,  300.000}, //Wpt 003
    {   0.0,   0.0,   0.0,  270.000,   60.000,  300.000}, //Wpt 004
    {   0.0,   0.0,   0.0,  270.000,   75.000,  300.000}, //Wpt 005
    {   0.0,   0.0,   0.0,  270.000,   90.000,  300.000}, //Wpt 006
    {   0.0,   0.0,   0.0,  270.000,  105.000,  300.000}, //Wpt 007
    {   0.0,   0.0,   0.0,  270.000,  120.000,  300.000}, //Wpt 008
    {   0.0,   0.0,   0.0,  270.000,  135.000,  300.000}, //Wpt 009
    {   0.0,   0.0,   0.0,  270.000,  150.000,  300.000}, //Wpt 010
    {   0.0,   0.0,   0.0,  270.000,  165.000,  300.000}, //Wpt 011
    {   0.0,   0.0,   0.0,  270.000,  180.000,  300.000}, //Wpt 012
```

```
    {   0.0,   0.0,   0.0,  270.000,  195.000,  300.000}, //Wpt 013
    {   0.0,   0.0,   0.0,  270.000,  210.000,  300.000}, //Wpt 014
    {   0.0,   0.0,   0.0,  270.000,  225.000,  300.000}, //Wpt 015
    {   0.0,   0.0,   0.0,  270.000,  240.000,  300.000}, //Wpt 016
    {   0.0,   0.0,   0.0,  270.000,  255.000,  300.000}, //Wpt 017
    {   0.0,   0.0,   0.0,  270.000,  270.000,  300.000}, //Wpt 018
    {   0.0,   0.0,   0.0,  270.000,  285.000,  300.000}, //Wpt 019
    {   0.0,   0.0,  -5.0,  270.000,  300.000,  300.000}, //Wpt 020
    { 358.0,   0.0,  -8.0,  270.000,  315.000,  300.000}, //Wpt 021
    { 355.5,   0.0, -10.0,  268.000,  330.000,  300.000}, //Wpt 022
    { 351.0,   0.0, -15.0,  266.000,  345.000,  300.000}, //Wpt 023
    { 346.5,   0.0, -20.0,  264.375,  360.000,  300.000}, //Wpt 024
    { 342.0,   0.0, -25.0,  258.750,  372.857,  300.000}, //Wpt 025
    { 337.5,   0.0, -30.0,  253.125,  385.714,  300.000}, //Wpt 026
    { 333.0,   0.0, -30.0,  247.500,  398.571,  300.000}, //Wpt 027
    { 328.5,   0.0, -30.0,  241.875,  411.428,  300.000}, //Wpt 028
    { 324.0,   0.0, -30.0,  236.250,  424.285,  300.000}, //Wpt 029
    { 319.5,   0.0, -30.0,  230.625,  437.142,  300.000}, //Wpt 030
    { 315.0,   0.0, -30.0,  225.000,  450.000,  300.000}, //Wpt 031
    { 310.5,   0.0, -30.0,  213.750,  460.000,  300.000}, //Wpt 032
    { 305.0,   0.0, -28.0,  202.500,  470.000,  300.000}, //Wpt 033
    { 300.0,   0.0, -25.0,  191.250,  480.000,  300.000}, //Wpt 034
    { 292.0,   0.0, -20.0,  180.000,  490.000,  300.000}, //Wpt 035
    { 290.5,   0.0, -20.0,  168.750,  493.750,  300.000}, //Wpt 036
    { 289.0,   0.0, -20.0,  157.500,  497.500,  300.000}, //Wpt 037
    { 287.5,   0.0, -20.0,  146.250,  501.250,  300.000}, //Wpt 038
    { 286.0,   0.0, -18.0,  135.000,  505.000,  300.000}, //Wpt 039
    { 284.5,   0.0, -15.0,  123.750,  508.750,  300.000}, //Wpt 040
    { 283.0,   0.0, -15.0,  112.500,  512.500,  300.000}, //Wpt 041
    { 281.5,   0.0, -12.0,  101.250,  516.250,  300.000}, //Wpt 042
    { 280.0,   0.0, -10.0,   90.000,  520.000,  300.000}, //Wpt 043
    { 277.5,   0.0,  -8.0,   75.000,  525.000,  300.000}, //Wpt 044
    { 275.0,   0.0,  -7.0,   60.000,  530.000,  300.000}, //Wpt 045
    { 273.5,   0.0,  -5.0,   45.000,  535.000,  300.000}, //Wpt 046
    { 272.0,   0.0,  -4.0,   30.000,  540.000,  300.000}, //Wpt 047
    { 271.0,   0.0,  -3.0,   15.000,  540.000,  300.000}, //Wpt 048
    { 270.0,   0.0,  -2.0,    0.000,  540.000,  300.000}, //Wpt 049
};

#endif
```

Listing 3. Flight Path 1 Data (Corrected)

```
//flightpath1.h
//Author: Sean McKeon (2009)

#ifndef __FLIGHTPATH1_H__
#define __FLIGHTPATH1_H__

#define NUM_WAYPOINTS 50

typedef struct flightpath_t{
    double heading;   //degrees (North = 0 degrees)
    double pitch;     //degrees (+ is pitch up)
    double roll;      //degrees (+ is bank right)
    double longitude; //meters
    double latitude;  //meters
    double altitude;  //meters
} FlightPath;

static const FlightPath flightpath[NUM_WAYPOINTS] = {
    {   0.0,   0.0,   0.0,  270.000,    0.000,  300.000}, //Wpt 000
    {   0.0,   0.0,   0.0,  270.000,   15.000,  300.000}, //Wpt 001
    {   0.0,   0.0,   0.0,  270.000,   30.000,  300.000}, //Wpt 002
    {   0.0,   0.0,   0.0,  270.000,   45.000,  300.000}, //Wpt 003
    {   0.0,   0.0,   0.0,  270.000,   60.000,  300.000}, //Wpt 004
```

```
    {   0.0,   0.0,   0.0,  270.000,   75.000,  300.000}, //Wpt 005
    {   0.0,   0.0,   0.0,  270.000,   90.000,  300.000}, //Wpt 006
    {   0.0,   0.0,   0.0,  270.000,  105.000,  300.000}, //Wpt 007
    {   0.0,   0.0,   0.0,  270.000,  120.000,  300.000}, //Wpt 008
    {   0.0,   0.0,   0.0,  270.000,  135.000,  300.000}, //Wpt 009
    {   0.0,   0.0,   0.0,  270.000,  150.000,  300.000}, //Wpt 010
    {   0.0,   0.0,   0.0,  270.000,  165.000,  300.000}, //Wpt 011
    {   0.0,   0.0,   0.0,  270.000,  180.000,  300.000}, //Wpt 012
    {   0.0,   0.0,   0.0,  270.000,  195.000,  300.000}, //Wpt 013
    {   0.0,   0.0,   0.0,  270.000,  210.000,  300.000}, //Wpt 014
    {   0.0,   0.0,   0.0,  270.000,  225.000,  300.000}, //Wpt 015
    {   0.0,   0.0,   0.0,  270.000,  240.000,  300.000}, //Wpt 016
    {   0.0,   0.0,   0.0,  270.000,  255.000,  300.000}, //Wpt 017
    {   0.0,   0.0,   0.0,  270.000,  270.000,  300.000}, //Wpt 018
    {   0.0,   0.0,   0.0,  270.000,  285.000,  300.000}, //Wpt 019
    {   0.0,   0.0,  -5.0,  270.000,  300.000,  300.000}, //Wpt 020
    { 358.0,   0.0,  -8.0,  270.000,  315.000,  300.000}, //Wpt 021
    { 355.5,   0.0, -10.0,  268.000,  330.000,  300.000}, //Wpt 022
    { 351.0,   0.0, -15.0,  266.000,  345.000,  300.000}, //Wpt 023
    { 346.5,   0.0, -20.0,  264.375,  360.000,  300.000}, //Wpt 024
    { 342.0,   0.0, -25.0,  258.750,  372.857,  300.000}, //Wpt 025
    { 337.5,   0.0, -30.0,  253.125,  385.714,  300.000}, //Wpt 026
    { 333.0,   0.0, -30.0,  247.500,  398.571,  300.000}, //Wpt 027
    { 328.5,   0.0, -30.0,  241.875,  411.428,  300.000}, //Wpt 028
    { 324.0,   0.0, -30.0,  236.250,  424.285,  300.000}, //Wpt 029
    { 319.5,   0.0, -30.0,  230.625,  437.142,  300.000}, //Wpt 030
    { 315.0,   0.0, -30.0,  225.000,  450.000,  300.000}, //Wpt 031
    { 310.5,   0.0, -30.0,  213.750,  460.000,  300.000}, //Wpt 032
    { 305.0,   0.0, -28.0,  202.500,  470.000,  300.000}, //Wpt 033
    { 300.0,   0.0, -25.0,  191.250,  480.000,  300.000}, //Wpt 034
    { 292.0,   0.0, -20.0,  180.000,  490.000,  300.000}, //Wpt 035
    { 290.5,   0.0, -20.0,  168.750,  493.750,  300.000}, //Wpt 036
    { 289.0,   0.0, -20.0,  157.500,  497.500,  300.000}, //Wpt 037
    { 287.5,   0.0, -20.0,  146.250,  501.250,  300.000}, //Wpt 038
    { 286.0,   0.0, -18.0,  135.000,  505.000,  300.000}, //Wpt 039
    { 284.5,   0.0, -15.0,  123.750,  508.750,  300.000}, //Wpt 040
    { 283.0,   0.0, -15.0,  112.500,  512.500,  300.000}, //Wpt 041
    { 281.5,   0.0, -12.0,  101.250,  516.250,  300.000}, //Wpt 042
    { 280.0,   0.0, -10.0,   90.000,  520.000,  300.000}, //Wpt 043
    { 277.5,   0.0,  -8.0,   75.000,  525.000,  300.000}, //Wpt 044
    { 275.0,   0.0,  -7.0,   60.000,  530.000,  300.000}, //Wpt 045
    { 273.5,   0.0,  -5.0,   45.000,  535.000,  300.000}, //Wpt 046
    { 272.0,   0.0,  -4.0,   30.000,  540.000,  300.000}, //Wpt 047
    { 271.0,   0.0,  -3.0,   15.000,  540.000,  300.000}, //Wpt 048
    { 270.0,   0.0,  -2.0,    0.000,  540.000,  300.000}, //Wpt 049
};

#endif
```

Listing 4. Flight Path 2 Data

```
//flightpath2.h
//Author: Sean McKeon (2009)

#ifndef __FLIGHTPATH2_H__
#define __FLIGHTPATH2_H__

#define NUM_WAYPOINTS 50

typedef struct flightpath_t{
    double heading;   //degrees (North = 0 degrees)
    double pitch;     //degrees (+ is pitch up)
    double roll;      //degrees (+ is bank right)
    double longitude; //meters
    double latitude;  //meters
    double altitude;  //meters
```

```
} FlightPath;

static const FlightPath flightpath[NUM_WAYPOINTS] = {
    {  90.0,   0.0,   0.0,    0.000,  360.000,  500.000}, //Wpt 000
    {  90.0,   0.0,   0.0,   15.000,  360.000,  500.000}, //Wpt 001
    {  90.0,   0.0,   0.0,   30.000,  360.000,  500.000}, //Wpt 002
    {  90.0,   0.0,   0.0,   45.000,  360.000,  500.000}, //Wpt 003
    {  90.0,   0.0,   0.0,   60.000,  360.000,  500.000}, //Wpt 004
    {  90.0,   0.0,   0.0,   75.000,  360.000,  500.000}, //Wpt 005
    {  90.0,   0.0,   0.0,   90.000,  360.000,  500.000}, //Wpt 006
    {  90.0,   0.0,   0.0,  105.000,  360.000,  500.000}, //Wpt 007
    {  90.0,   0.0,   0.0,  120.000,  360.000,  500.000}, //Wpt 008
    {  90.0,   0.0,   0.0,  135.000,  360.000,  500.000}, //Wpt 009
    {  90.0,   0.0,   0.0,  150.000,  360.000,  500.000}, //Wpt 010
    {  90.0,   0.0,   0.0,  165.000,  360.000,  500.000}, //Wpt 011
    {  90.0,   0.0,   0.0,  180.000,  360.000,  500.000}, //Wpt 012
    {  90.0,   0.0,   0.0,  195.000,  360.000,  500.000}, //Wpt 013
    {  90.0,   0.0,   0.0,  210.000,  360.000,  500.000}, //Wpt 014
    {  90.0,   0.0,   0.0,  225.000,  360.000,  500.000}, //Wpt 015
    {  90.0,   0.0,   0.0,  240.000,  360.000,  500.000}, //Wpt 016
    {  90.0,   0.0,   0.0,  255.000,  360.000,  500.000}, //Wpt 017
    {  90.0,   0.0,   0.0,  270.000,  360.000,  500.000}, //Wpt 018
    {  90.0,   0.0,   0.0,  285.000,  360.000,  500.000}, //Wpt 019
    {  90.0,   0.0,   0.0,  300.000,  360.000,  500.000}, //Wpt 020
    {  90.0,   0.0,   0.0,  315.000,  360.000,  500.000}, //Wpt 021
    {  90.0,   0.0,   0.0,  330.000,  360.000,  500.000}, //Wpt 022
    {  90.0,   0.0,   0.0,  345.000,  360.000,  500.000}, //Wpt 023
    {  90.0,   0.0,   0.0,  360.000,  360.000,  500.000}, //Wpt 024
    {  90.0,   0.0,   0.0,  372.857,  360.000,  500.000}, //Wpt 025
    {  90.0,   0.0,   0.0,  385.714,  360.000,  500.000}, //Wpt 026
    {  90.0,   0.0,   0.0,  398.571,  360.000,  500.000}, //Wpt 027
    {  90.0,   0.0,   0.0,  411.428,  360.000,  500.000}, //Wpt 028
    {  90.0,   0.0,   0.0,  424.285,  360.000,  500.000}, //Wpt 029
    {  90.0,   0.0,   2.0,  437.142,  358.000,  500.000}, //Wpt 030
    {  90.0,   0.0,   5.0,  450.000,  356.000,  500.000}, //Wpt 031
    {  95.0,   0.0,   8.0,  468.000,  354.000,  500.000}, //Wpt 032
    { 100.0,   0.0,  12.0,  486.000,  348.000,  500.000}, //Wpt 033
    { 105.0,   0.0,  15.0,  504.000,  342.000,  500.000}, //Wpt 034
    { 111.0,   0.0,  18.0,  522.000,  336.000,  500.000}, //Wpt 035
    { 117.0,   0.0,  20.0,  540.000,  330.000,  500.000}, //Wpt 036
    { 123.0,   0.0,  24.0,  555.000,  325.000,  500.000}, //Wpt 037
    { 129.0,   0.0,  25.0,  570.000,  315.000,  500.000}, //Wpt 038
    { 135.0,   0.0,  22.0,  585.000,  300.000,  500.000}, //Wpt 039
    { 137.0,   0.0,  20.0,  600.000,  285.000,  500.000}, //Wpt 040
    { 139.0,   0.0,  15.0,  615.000,  270.000,  500.000}, //Wpt 041
    { 142.0,   0.0,   8.0,  625.000,  250.000,  500.000}, //Wpt 042
    { 145.0,   0.0,  -4.0,  635.000,  230.000,  500.000}, //Wpt 043
    { 141.0,   0.0,  -7.0,  650.000,  210.000,  500.000}, //Wpt 044
    { 138.0,   0.0, -12.0,  665.000,  195.000,  500.000}, //Wpt 045
    { 135.0,   0.0, -15.0,  675.000,  185.000,  500.000}, //Wpt 046
    { 128.0,   0.0, -10.0,  690.000,  175.000,  500.000}, //Wpt 047
    { 120.0,   0.0,  -6.0,  700.000,  168.000,  500.000}, //Wpt 048
    { 112.0,   0.0,  -3.0,  720.000,  155.000,  500.000}, //Wpt 049
};

#endif
```

Listing 5. Flight Path 3 Data

```
//flightpath3.h
//Author: Sean McKeon (2009)

#ifndef __FLIGHTPATH3_H__
#define __FLIGHTPATH3_H__

#define NUM_WAYPOINTS 50

typedef struct flightpath_t{
    double heading;   //degrees (North = 0 degrees)
    double pitch;     //degrees (+ is pitch up)
    double roll;      //degrees (+ is bank right)
    double longitude; //meters
    double latitude;  //meters
    double altitude;  //meters
} FlightPath;

static const FlightPath flightpath[NUM_WAYPOINTS] = {
    { 270.0,   8.0,   0.0,  720.000,  180.000,  600.000}, //Wpt 000
    { 270.0,   8.0,   0.0,  705.000,  180.000,  587.000}, //Wpt 001
    { 270.0,   8.0,   0.0,  690.000,  180.000,  574.000}, //Wpt 002
    { 270.0,   8.0,   0.0,  675.000,  180.000,  561.000}, //Wpt 003
    { 270.0,   8.0,   0.0,  660.000,  180.000,  548.000}, //Wpt 004
    { 270.0,   8.0,   0.0,  645.000,  180.000,  535.000}, //Wpt 005
    { 270.0,   8.0,   0.0,  630.000,  180.000,  522.000}, //Wpt 006
    { 270.0,   8.0,   0.0,  615.000,  180.000,  509.000}, //Wpt 007
    { 270.0,   8.0,   0.0,  600.000,  180.000,  496.000}, //Wpt 008
    { 270.0,   8.0,   0.0,  585.000,  180.000,  483.000}, //Wpt 009
    { 270.0,   7.0,   0.0,  570.000,  180.000,  470.000}, //Wpt 010
    { 270.0,   7.0,   0.0,  555.000,  180.000,  457.000}, //Wpt 011
    { 270.0,   7.0,   0.0,  540.000,  180.000,  444.000}, //Wpt 012
    { 270.0,   7.0,   0.0,  525.000,  180.000,  431.000}, //Wpt 013
    { 270.0,   7.0,   0.0,  510.000,  180.000,  418.000}, //Wpt 014
    { 270.0,   7.0,   0.0,  495.000,  180.000,  405.000}, //Wpt 015
    { 270.0,   7.0,   0.0,  480.000,  180.000,  392.000}, //Wpt 016
    { 270.0,   6.0,   0.0,  465.000,  180.000,  379.000}, //Wpt 017
    { 270.0,   6.0,   0.0,  450.000,  180.000,  366.000}, //Wpt 018
    { 270.0,   6.0,   0.0,  437.150,  180.000,  353.000}, //Wpt 019
    { 270.0,   6.0,   0.0,  424.300,  180.000,  340.000}, //Wpt 020
    { 270.0,   6.0,   0.0,  411.450,  180.000,  327.000}, //Wpt 021
    { 270.0,   5.0,   0.0,  398.600,  180.000,  314.000}, //Wpt 022
    { 270.0,   5.0,   0.0,  385.750,  180.000,  301.000}, //Wpt 023
    { 270.0,   5.0,   0.0,  372.900,  180.000,  288.375}, //Wpt 024
    { 270.0,   5.0,   0.0,  360.000,  180.000,  275.750}, //Wpt 025
    { 270.0,   5.0,   0.0,  345.000,  180.000,  263.125}, //Wpt 026
    { 270.0,   5.0,   0.0,  330.000,  180.000,  250.500}, //Wpt 027
    { 270.0,   5.0,   0.0,  315.000,  180.000,  235.875}, //Wpt 028
    { 270.0,   5.0,   0.0,  300.000,  180.000,  222.250}, //Wpt 029
    { 270.0,   5.0,   0.0,  285.000,  180.000,  212.625}, //Wpt 030
    { 270.0,   4.0,   0.0,  270.000,  180.000,  210.000}, //Wpt 031
    { 270.0,   3.0,   0.0,  255.000,  180.000,  207.000}, //Wpt 032
    { 270.0,   2.0,   0.0,  240.000,  180.000,  204.000}, //Wpt 033
    { 270.0,   1.0,   0.0,  225.000,  180.000,  202.000}, //Wpt 034
    { 270.0,   0.0,   0.0,  210.000,  180.000,  200.000}, //Wpt 035
    { 270.0,   0.0,   0.0,  195.000,  180.000,  200.000}, //Wpt 036
    { 270.0,   0.0,   0.0,  180.000,  180.000,  200.000}, //Wpt 037
    { 270.0,   0.0,   0.0,  165.000,  180.000,  200.000}, //Wpt 038
    { 270.0,   0.0,   0.0,  150.000,  180.000,  200.000}, //Wpt 039
    { 270.0,   0.0,   0.0,  135.000,  180.000,  200.000}, //Wpt 040
    { 270.0,   0.0,   0.0,  120.000,  180.000,  200.000}, //Wpt 041
    { 270.0,   0.0,   0.0,  105.000,  180.000,  200.000}, //Wpt 042
    { 270.0,   0.0,   0.0,   90.000,  180.000,  200.000}, //Wpt 043
    { 270.0,   0.0,   0.0,   75.000,  180.000,  200.000}, //Wpt 044
    { 270.0,   0.0,   0.0,   60.000,  180.000,  200.000}, //Wpt 045
    { 270.0,   0.0,   0.0,   45.000,  180.000,  200.000}, //Wpt 046
    { 270.0,   0.0,   0.0,   30.000,  180.000,  200.000}, //Wpt 047
    { 270.0,   0.0,   0.0,   15.000,  180.000,  200.000}, //Wpt 048
```

```
    { 270.0,   0.0,   0.0,    0.000,  180.000,  200.000}, //Wpt 049
};

#endif
```

Listing 6. Flight Path 1 Data: 'Spot' Simulation

```
//flightpath1_spot.h
//Author: Sean McKeon (2009)

#ifndef __FLIGHTPATH1_SPOT_H__
#define __FLIGHTPATH1_SPOT_H__

#define NUM_WAYPOINTS 50

typedef struct flightpath_t{
    double heading;   //degrees (North = 0 degrees)
    double pitch;     //degrees (+ is pitch up)
    double roll;      //degrees (+ is bank right)
    double longitude; //meters
    double latitude;  //meters
    double altitude;  //meters
} FlightPath;

static const FlightPath flightpath[NUM_WAYPOINTS] = {
    {   0.0,   0.0,   0.0,  270.000,    0.000,  300.000}, //Wpt 000
    {   0.0,   0.0,   0.0,  270.000,   15.000,  300.000}, //Wpt 001
    {   0.0,   0.0,   0.0,  270.000,   30.000,  300.000}, //Wpt 002
    {   0.0,   0.0,   0.0,  270.000,   45.000,  300.000}, //Wpt 003
    {   0.0,   0.0,   0.0,  270.000,   60.000,  300.000}, //Wpt 004
    {   0.0,   0.0,   0.0,  270.000,   75.000,  300.000}, //Wpt 005
    {   0.0,   0.0,   0.0,  270.000,   90.000,  300.000}, //Wpt 006
    {   0.0,   0.0,   0.0,  270.000,  105.000,  300.000}, //Wpt 007
    {   0.0,   0.0,   0.0,  270.000,  120.000,  300.000}, //Wpt 008
    {   0.0,   0.0,   0.0,  270.000,  135.000,  300.000}, //Wpt 009
    {   0.0,   0.0,   0.0,  270.000,  150.000,  300.000}, //Wpt 010
    {   0.0,   0.0,   0.0,  270.000,  165.000,  300.000}, //Wpt 011
    {   0.0,   0.0,   0.0,  270.000,  180.000,  300.000}, //Wpt 012
    {   0.0,   0.0,   0.0,  270.000,  195.000,  300.000}, //Wpt 013
    {   0.0,   0.0,   0.0,  270.000,  210.000,  300.000}, //Wpt 014
    {   0.0,   0.0,   0.0,  270.000,  225.000,  300.000}, //Wpt 015
    {   0.0,   0.0,   0.0,  270.000,  240.000,  300.000}, //Wpt 016
    {   0.0,   0.0,   0.0,  270.000,  255.000,  300.000}, //Wpt 017
    {   0.0,   0.0,   0.0,  270.000,  270.000,  300.000}, //Wpt 018
    {   0.0,   0.0,   0.0,  270.000,  285.000,  300.000}, //Wpt 019
    {   0.0,   0.0,   0.0,  270.000,  300.000,  300.000}, //Wpt 020
    { 358.0,   0.0,   0.0,  270.000,  315.000,  300.000}, //Wpt 021
    { 355.5,   0.0,   0.0,  268.000,  330.000,  300.000}, //Wpt 022
    { 351.0,   0.0,   0.0,  266.000,  345.000,  300.000}, //Wpt 023
    { 346.5,   0.0,   0.0,  264.375,  360.000,  300.000}, //Wpt 024
    { 342.0,   0.0,   0.0,  258.750,  372.857,  300.000}, //Wpt 025
    { 337.5,   0.0,   0.0,  253.125,  385.714,  300.000}, //Wpt 026
    { 333.0,   0.0,   0.0,  247.500,  398.571,  300.000}, //Wpt 027
    { 328.5,   0.0,   0.0,  241.875,  411.428,  300.000}, //Wpt 028
    { 324.0,   0.0,   0.0,  236.250,  424.285,  300.000}, //Wpt 029
    { 319.5,   0.0,   0.0,  230.625,  437.142,  300.000}, //Wpt 030
    { 315.0,   0.0,   0.0,  225.000,  450.000,  300.000}, //Wpt 031
    { 310.5,   0.0,   0.0,  213.750,  460.000,  300.000}, //Wpt 032
    { 305.0,   0.0,   0.0,  202.500,  470.000,  300.000}, //Wpt 033
    { 300.0,   0.0,   0.0,  191.250,  480.000,  300.000}, //Wpt 034
    { 292.0,   0.0,   0.0,  180.000,  490.000,  300.000}, //Wpt 035
    { 290.5,   0.0,   0.0,  168.750,  493.750,  300.000}, //Wpt 036
    { 289.0,   0.0,   0.0,  157.500,  497.500,  300.000}, //Wpt 037
    { 287.5,   0.0,   0.0,  146.250,  501.250,  300.000}, //Wpt 038
    { 286.0,   0.0,   0.0,  135.000,  505.000,  300.000}, //Wpt 039
    { 284.5,   0.0,   0.0,  123.750,  508.750,  300.000}, //Wpt 040
```

```
    { 283.0,   0.0,   0.0,  112.500,  512.500,  300.000}, //Wpt 041
    { 281.5,   0.0,   0.0,  101.250,  516.250,  300.000}, //Wpt 042
    { 280.0,   0.0,   0.0,   90.000,  520.000,  300.000}, //Wpt 043
    { 277.5,   0.0,   0.0,   75.000,  525.000,  300.000}, //Wpt 044
    { 275.0,   0.0,   0.0,   60.000,  530.000,  300.000}, //Wpt 045
    { 273.5,   0.0,   0.0,   45.000,  535.000,  300.000}, //Wpt 046
    { 272.0,   0.0,   0.0,   30.000,  540.000,  300.000}, //Wpt 047
    { 271.0,   0.0,   0.0,   15.000,  540.000,  300.000}, //Wpt 048
    { 270.0,   0.0,   0.0,    0.000,  540.000,  300.000}, //Wpt 049
};

#endif
```

Listing 7. Flight Path 2 Data: 'Spot' Simulation

```
//flightpath2_spot.h
//Author: Sean McKeon (2009)

#ifndef __FLIGHTPATH2_SPOT_H__
#define __FLIGHTPATH2_SPOT_H__

#define NUM_WAYPOINTS 50

typedef struct flightpath_t{
    double heading;   //degrees (North = 0 degrees)
    double pitch;     //degrees (+ is pitch up)
    double roll;      //degrees (+ is bank right)
    double longitude; //meters
    double latitude;  //meters
    double altitude;  //meters
} FlightPath;

static const FlightPath flightpath[NUM_WAYPOINTS] = {
    {  90.0,   0.0,   0.0,    0.000,  360.000,  500.000}, //Wpt 000
    {  90.0,   0.0,   0.0,   15.000,  360.000,  500.000}, //Wpt 001
    {  90.0,   0.0,   0.0,   30.000,  360.000,  500.000}, //Wpt 002
    {  90.0,   0.0,   0.0,   45.000,  360.000,  500.000}, //Wpt 003
    {  90.0,   0.0,   0.0,   60.000,  360.000,  500.000}, //Wpt 004
    {  90.0,   0.0,   0.0,   75.000,  360.000,  500.000}, //Wpt 005
    {  90.0,   0.0,   0.0,   90.000,  360.000,  500.000}, //Wpt 006
    {  90.0,   0.0,   0.0,  105.000,  360.000,  500.000}, //Wpt 007
    {  90.0,   0.0,   0.0,  120.000,  360.000,  500.000}, //Wpt 008
    {  90.0,   0.0,   0.0,  135.000,  360.000,  500.000}, //Wpt 009
    {  90.0,   0.0,   0.0,  150.000,  360.000,  500.000}, //Wpt 010
    {  90.0,   0.0,   0.0,  165.000,  360.000,  500.000}, //Wpt 011
    {  90.0,   0.0,   0.0,  180.000,  360.000,  500.000}, //Wpt 012
    {  90.0,   0.0,   0.0,  195.000,  360.000,  500.000}, //Wpt 013
    {  90.0,   0.0,   0.0,  210.000,  360.000,  500.000}, //Wpt 014
    {  90.0,   0.0,   0.0,  225.000,  360.000,  500.000}, //Wpt 015
    {  90.0,   0.0,   0.0,  240.000,  360.000,  500.000}, //Wpt 016
    {  90.0,   0.0,   0.0,  255.000,  360.000,  500.000}, //Wpt 017
    {  90.0,   0.0,   0.0,  270.000,  360.000,  500.000}, //Wpt 018
    {  90.0,   0.0,   0.0,  285.000,  360.000,  500.000}, //Wpt 019
    {  90.0,   0.0,   0.0,  300.000,  360.000,  500.000}, //Wpt 020
    {  90.0,   0.0,   0.0,  315.000,  360.000,  500.000}, //Wpt 021
    {  90.0,   0.0,   0.0,  330.000,  360.000,  500.000}, //Wpt 022
    {  90.0,   0.0,   0.0,  345.000,  360.000,  500.000}, //Wpt 023
    {  90.0,   0.0,   0.0,  360.000,  360.000,  500.000}, //Wpt 024
    {  90.0,   0.0,   0.0,  372.857,  360.000,  500.000}, //Wpt 025
    {  90.0,   0.0,   0.0,  385.714,  360.000,  500.000}, //Wpt 026
    {  90.0,   0.0,   0.0,  398.571,  360.000,  500.000}, //Wpt 027
    {  90.0,   0.0,   0.0,  411.428,  360.000,  500.000}, //Wpt 028
    {  90.0,   0.0,   0.0,  424.285,  360.000,  500.000}, //Wpt 029
    {  90.0,   0.0,   0.0,  437.142,  358.000,  500.000}, //Wpt 030
    {  90.0,   0.0,   0.0,  450.000,  356.000,  500.000}, //Wpt 031
    {  95.0,   0.0,   0.0,  468.000,  354.000,  500.000}, //Wpt 032
```

```
    { 100.0,   0.0,   0.0,  486.000,  348.000,  500.000}, //Wpt 033
    { 105.0,   0.0,   0.0,  504.000,  342.000,  500.000}, //Wpt 034
    { 111.0,   0.0,   0.0,  522.000,  336.000,  500.000}, //Wpt 035
    { 117.0,   0.0,   0.0,  540.000,  330.000,  500.000}, //Wpt 036
    { 123.0,   0.0,   0.0,  555.000,  325.000,  500.000}, //Wpt 037
    { 129.0,   0.0,   0.0,  570.000,  315.000,  500.000}, //Wpt 038
    { 135.0,   0.0,   0.0,  585.000,  300.000,  500.000}, //Wpt 039
    { 137.0,   0.0,   0.0,  600.000,  285.000,  500.000}, //Wpt 040
    { 139.0,   0.0,   0.0,  615.000,  270.000,  500.000}, //Wpt 041
    { 142.0,   0.0,   0.0,  625.000,  250.000,  500.000}, //Wpt 042
    { 145.0,   0.0,   0.0,  635.000,  230.000,  500.000}, //Wpt 043
    { 141.0,   0.0,   0.0,  650.000,  210.000,  500.000}, //Wpt 044
    { 138.0,   0.0,   0.0,  665.000,  195.000,  500.000}, //Wpt 045
    { 135.0,   0.0,   0.0,  675.000,  185.000,  500.000}, //Wpt 046
    { 128.0,   0.0,   0.0,  690.000,  175.000,  500.000}, //Wpt 047
    { 120.0,   0.0,   0.0,  700.000,  168.000,  500.000}, //Wpt 048
    { 112.0,   0.0,   0.0,  720.000,  155.000,  500.000}, //Wpt 049
};

#endif
```

Listing 8. Flight Path 3 Data: 'Spot' Simulation

```
//flightpath3_spot.h
//Author: Sean McKeon (2009)

#ifndef __FLIGHTPATH3_SPOT_H__
#define __FLIGHTPATH3_SPOT_H__

#define NUM_WAYPOINTS 50

typedef struct flightpath_t{
    double heading;   //degrees (North = 0 degrees)
    double pitch;     //degrees (+ is pitch up)
    double roll;      //degrees (+ is bank right)
    double longitude; //meters
    double latitude;  //meters
    double altitude;  //meters
} FlightPath;

static const FlightPath flightpath[NUM_WAYPOINTS] = {
    { 270.0,   0.0,   0.0,  720.000,  180.000,  600.000}, //Wpt 000
    { 270.0,   0.0,   0.0,  705.000,  180.000,  587.000}, //Wpt 001
    { 270.0,   0.0,   0.0,  690.000,  180.000,  574.000}, //Wpt 002
    { 270.0,   0.0,   0.0,  675.000,  180.000,  561.000}, //Wpt 003
    { 270.0,   0.0,   0.0,  660.000,  180.000,  548.000}, //Wpt 004
    { 270.0,   0.0,   0.0,  645.000,  180.000,  535.000}, //Wpt 005
    { 270.0,   0.0,   0.0,  630.000,  180.000,  522.000}, //Wpt 006
    { 270.0,   0.0,   0.0,  615.000,  180.000,  509.000}, //Wpt 007
    { 270.0,   0.0,   0.0,  600.000,  180.000,  496.000}, //Wpt 008
    { 270.0,   0.0,   0.0,  585.000,  180.000,  483.000}, //Wpt 009
    { 270.0,   0.0,   0.0,  570.000,  180.000,  470.000}, //Wpt 010
    { 270.0,   0.0,   0.0,  555.000,  180.000,  457.000}, //Wpt 011
    { 270.0,   0.0,   0.0,  540.000,  180.000,  444.000}, //Wpt 012
    { 270.0,   0.0,   0.0,  525.000,  180.000,  431.000}, //Wpt 013
    { 270.0,   0.0,   0.0,  510.000,  180.000,  418.000}, //Wpt 014
    { 270.0,   0.0,   0.0,  495.000,  180.000,  405.000}, //Wpt 015
    { 270.0,   0.0,   0.0,  480.000,  180.000,  392.000}, //Wpt 016
    { 270.0,   0.0,   0.0,  465.000,  180.000,  379.000}, //Wpt 017
    { 270.0,   0.0,   0.0,  450.000,  180.000,  366.000}, //Wpt 018
    { 270.0,   0.0,   0.0,  437.150,  180.000,  353.000}, //Wpt 019
    { 270.0,   0.0,   0.0,  424.300,  180.000,  340.000}, //Wpt 020
    { 270.0,   0.0,   0.0,  411.450,  180.000,  327.000}, //Wpt 021
    { 270.0,   0.0,   0.0,  398.600,  180.000,  314.000}, //Wpt 022
    { 270.0,   0.0,   0.0,  385.750,  180.000,  301.000}, //Wpt 023
    { 270.0,   0.0,   0.0,  372.900,  180.000,  288.375}, //Wpt 024
```

```
    { 270.0,   0.0,   0.0,  360.000,  180.000,  275.750}, //Wpt 025
    { 270.0,   0.0,   0.0,  345.000,  180.000,  263.125}, //Wpt 026
    { 270.0,   0.0,   0.0,  330.000,  180.000,  250.500}, //Wpt 027
    { 270.0,   0.0,   0.0,  315.000,  180.000,  235.875}, //Wpt 028
    { 270.0,   0.0,   0.0,  300.000,  180.000,  222.250}, //Wpt 029
    { 270.0,   0.0,   0.0,  285.000,  180.000,  212.625}, //Wpt 030
    { 270.0,   0.0,   0.0,  270.000,  180.000,  210.000}, //Wpt 031
    { 270.0,   0.0,   0.0,  255.000,  180.000,  207.000}, //Wpt 032
    { 270.0,   0.0,   0.0,  240.000,  180.000,  204.000}, //Wpt 033
    { 270.0,   0.0,   0.0,  225.000,  180.000,  202.000}, //Wpt 034
    { 270.0,   0.0,   0.0,  210.000,  180.000,  200.000}, //Wpt 035
    { 270.0,   0.0,   0.0,  195.000,  180.000,  200.000}, //Wpt 036
    { 270.0,   0.0,   0.0,  180.000,  180.000,  200.000}, //Wpt 037
    { 270.0,   0.0,   0.0,  165.000,  180.000,  200.000}, //Wpt 038
    { 270.0,   0.0,   0.0,  150.000,  180.000,  200.000}, //Wpt 039
    { 270.0,   0.0,   0.0,  135.000,  180.000,  200.000}, //Wpt 040
    { 270.0,   0.0,   0.0,  120.000,  180.000,  200.000}, //Wpt 041
    { 270.0,   0.0,   0.0,  105.000,  180.000,  200.000}, //Wpt 042
    { 270.0,   0.0,   0.0,   90.000,  180.000,  200.000}, //Wpt 043
    { 270.0,   0.0,   0.0,   75.000,  180.000,  200.000}, //Wpt 044
    { 270.0,   0.0,   0.0,   60.000,  180.000,  200.000}, //Wpt 045
    { 270.0,   0.0,   0.0,   45.000,  180.000,  200.000}, //Wpt 046
    { 270.0,   0.0,   0.0,   30.000,  180.000,  200.000}, //Wpt 047
    { 270.0,   0.0,   0.0,   15.000,  180.000,  200.000}, //Wpt 048
    { 270.0,   0.0,   0.0,    0.000,  180.000,  200.000}, //Wpt 049
};

#endif
```

Listing 9. BRCC GPU(DX9) Source Code (monitor.cpp)

```
//////////////////////////////////////////
// Generated by BRCC v0.1
// BRCC Compiled on: Feb 21 2005 22:01:25
//////////////////////////////////////////

#include <brook/brook.hpp>
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include "terrain32.h"

#include "flightpath.h"

#define DEG2RAD(x) (((x)*3.1415926535898)/180.0)

#define HDG2ANGLE(x) (-x + 90.0)

typedef struct triangle_t {
  float3  v0;
  float3  v1;
  float3  v2;
} Triangle;typedef struct __cpustruct_triangle_t {
__BrtFloat3 v0;
__BrtFloat3 v1;
__BrtFloat3 v2;
}
__cpustruct_Triangle;

namespace brook {
        template<> const StreamType* getStreamType(Triangle*) {
                static const StreamType result[] = {__BRTFLOAT3, __BRTFLOAT3, __BRTFLOAT3,
                    __BRTNONE};
```

```
                return result;
        }
}

void  resetTimer(void );
float  getTimer(void );

namespace {
        using namespace ::brook::desc;
        static const gpu_kernel_desc __rayIntersect_ps20_desc = gpu_kernel_desc()
                .technique( gpu_technique_desc()
                        .pass( gpu_pass_desc(
                                "    ps_2_0\n"
                                "    def c1, -4, -3, -2, -1\n"
                                "    def c2, 1, 0, -9.99999975e-006, -5\n"
                                "    dcl t0.xy\n"
                                "    dcl t1.xy\n"
                                "    dcl_2d s0\n"
                                "    dcl_2d s1\n"
                                "    dcl_2d s2\n"
                                "    dcl_2d s3\n"
                                "    texld r0, t0, s0\n"
                                "    texld r1, t1, s1\n"
                                "    texld r3, t1, s2\n"
                                "    texld r2, t1, s3\n"
                                "    add r5.xyz, -r1, c0\n"
                                "    add r3.xyz, -r1, r3\n"
                                "    add r4.xyz, -r1, r2\n"
                                "    mul r2.xyz, r3.zxyw, r4.yzxw\n"
                                "    mad r2.xyz, r3.yzxw, r4.zxyw, -r2\n"
                                "    dp3 r6.x, r2, r0\n"
                                "    dp3 r5.x, r2, r5\n"
                                "    rcp r0.w, r6.x\n"
                                "    mul r1.w, -r5.x, r0.w\n"
                                "    mad r0.xyz, r1.w, r0, c0\n"
                                "    add r1.xyz, -r1, r0\n"
                                "    dp3 r5.x, r1, r4\n"
                                "    abs r0.w, r6.x\n"
                                "    dp3 r8.x, r3, r3\n"
                                "    dp3 r7.x, r4, r4\n"
                                "    mul r2.w, r5.x, r8.x\n"
                                "    dp3 r6.x, r1, r3\n"
                                "    mul r3.w, r8.x, r7.x\n"
                                "    dp3 r1.x, r3, r4\n"
                                "    mul r4.w, r7.x, r6.x\n"
                                "    mad r2.w, r1.x, r6.x, -r2.w\n"
                                "    mad r3.w, r1.x, r1.x, -r3.w\n"
                                "    mad r4.w, r1.x, r5.x, -r4.w\n"
                                "    rcp r5.w, r3.w\n"
                                "    mul r2.w, r2.w, r5.w\n"
                                "    mad r3.w, r4.w, r5.w, r2.w\n"
                                "    cmp r2.w, r2.w, c2.y, c2.x\n"
                                "    add r3.w, -r3.w, c2.x\n"
                                "    cmp r3.w, r3.w, c2.y, c2.x\n"
                                "    add r2.w, r2.w, r3.w\n"
                                "    cmp r3.w, -r2.w, c2.x, c2.w\n"
                                "    mul r2.w, r4.w, r5.w\n"
                                "    mad r4.w, r4.w, -r5.w, c2.x\n"
                                "    cmp r2.w, r2.w, c2.y, c2.x\n"
                                "    cmp r4.w, r4.w, c2.y, c2.x\n"
                                "    add r2.w, r2.w, r4.w\n"
                                "    cmp r2.w, -r2.w, r3.w, c1.x\n"
                                "    cmp r1.w, r1.w, r2.w, c1.y\n"
                                "    add r0.w, r0.w, c2.z\n"
                                "    cmp r1.w, r0.w, r1.w, c1.z\n"
                                "    mul r0.w, r2.x, r2.x\n"
                                "    cmp r0.w, -r0.w, c2.x, c2.y\n"
                                "    mul r3.w, r2.y, r2.y\n"
```

```
                                "    mul r2.w, r2.z, r2.z\n"
                                "    cmp r3.w, -r3.w, c2.x, c2.y\n"
                                "    mul r0.w, r0.w, r3.w\n"
                                "    cmp r2.w, -r2.w, c2.x, c2.y\n"
                                "    mul r0.w, r0.w, r2.w\n"
                                "    cmp r0.w, -r0.w, r1.w, c1.w\n"
                                "    mov oC0, r0\n"
                                "\n"
                                " \n"
                                "//!!BRCC\n"
                                "//narg:4\n"
                                "//c:3:o\n"
                                "//s:3:R\n"
                                "//s:0:T\n"
                                "//o:4:Hit\n"
                                "//workspace:1024\n"
                                "//!!multipleOutputInfo:0:1:\n"
                                "//!!fullAddressTrans:0:\n"
                                "//!!reductionFactor:0:\n"
                                "")
                                .constant(1, 0)
                                .sampler(2, 0)
                                .sampler(3, 0)
                                .sampler(3, 1)
                                .sampler(3, 2)
                                .interpolant(2, kStreamInterpolant_Position)
                                .interpolant(3, kStreamInterpolant_Position)
                                .output(4, 0)
                        )
                );
        static const void* __rayIntersect_ps20 = &__rayIntersect_ps20_desc;
}

static const char *__rayIntersect_ps2b= NULL;
static const char *__rayIntersect_ps2a= NULL;
static const char *__rayIntersect_ps30= NULL;
static const char *__rayIntersect_fp30= NULL;
static const char *__rayIntersect_fp40= NULL;
static const char *__rayIntersect_arb= NULL;
static const char *__rayIntersect_cpu= NULL;
void  rayIntersect (const float3  o,
                ::brook::stream R,
                ::brook::stream T,
                ::brook::stream Hit) {
  static const void *__rayIntersect_fp[] = {
     "fp30", __rayIntersect_fp30,
     "fp40", __rayIntersect_fp40,
     "arb", __rayIntersect_arb,
     "ps20", __rayIntersect_ps20,
     "ps2b", __rayIntersect_ps2b,
     "ps2a", __rayIntersect_ps2a,
     "ps30", __rayIntersect_ps30,
     "cpu", (void *) __rayIntersect_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__rayIntersect_fp);

  __k->PushConstant(o);
  __k->PushStream(R);
  __k->PushStream(T);
  __k->PushOutput(Hit);
  __k->Map();

}

namespace {
        using namespace ::brook::desc;
        static const gpu_kernel_desc __rayRotate_ps20_desc = gpu_kernel_desc()
                .technique( gpu_technique_desc()
```

```
                        .pass( gpu_pass_desc(
                                "    ps_2_0\n"
                                "    def c3, 0, 0, 0, 0\n"
                                "    dcl t0.xy\n"
                                "    dcl_2d s0\n"
                                "    texld r1, t0, s0\n"
                                "    mov r0.xyz, c2\n"
                                "    mul r1.w, r0.z, c0.z\n"
                                "    mul r0.w, r0.x, c1.y\n"
                                "    mad r3.w, r0.w, c0.x, r1.w\n"
                                "    mul r1.w, r0.x, c0.z\n"
                                "    mul r2.w, r0.y, c1.y\n"
                                "    mad r1.w, r2.w, c0.x, r1.w\n"
                                "    mul r4.w, r1.y, r1.w\n"
                                "    mul r1.w, r1.x, c1.x\n"
                                "    mad r4.w, r1.w, c0.x, r4.w\n"
                                "    mul r5.w, r0.z, c0.x\n"
                                "    mad r0.w, r0.w, c0.y, r5.w\n"
                                "    mul r5.w, r0.x, c0.x\n"
                                "    mad r2.w, r2.w, c0.y, r5.w\n"
                                "    mul r2.w, r1.y, r2.w\n"
                                "    mad r1.w, r1.w, c0.y, r2.w\n"
                                "    mul r2.w, r1.y, c2.y\n"
                                "    mul r2.w, r2.w, c1.x\n"
                                "    mad r0.x, r1.z, r3.w, r4.w\n"
                                "    mad r2.w, r1.x, c1.z, r2.w\n"
                                "    mad r0.y, r1.z, r0.w, r1.w\n"
                                "    mul r1.w, r1.z, c2.x\n"
                                "    mov r0.w, c3.x\n"
                                "    mad r0.z, r1.w, c1.x, r2.w\n"
                                "    mov oC0, r0\n"
                                "\n"
                                " \n"
                                "//!!BRCC\n"
                                "//narg:5\n"
                                "//s:3:R\n"
                                "//c:3:hdgM\n"
                                "//c:3:pitchM\n"
                                "//c:3:rollM\n"
                                "//o:3:rotR\n"
                                "//workspace:1024\n"
                                "//!!multipleOutputInfo:0:1:\n"
                                "//!!fullAddressTrans:0:\n"
                                "//!!reductionFactor:0:\n"
                                "")
                                .constant(2, 0)
                                .constant(3, 0)
                                .constant(4, 0)
                                .sampler(1, 0)
                                .interpolant(1, kStreamInterpolant_Position)
                                .output(5, 0)
                        )
                );
        static const void* __rayRotate_ps20 = &__rayRotate_ps20_desc;
}

static const char *__rayRotate_ps2b= NULL;
static const char *__rayRotate_ps2a= NULL;
static const char *__rayRotate_ps30= NULL;
static const char *__rayRotate_fp30= NULL;
static const char *__rayRotate_fp40= NULL;
static const char *__rayRotate_arb= NULL;
static const char *__rayRotate_cpu= NULL;
void  rayRotate (::brook::stream R,
                const float3  hdgM,
                const float3  pitchM,
                const float3  rollM,
                ::brook::stream rotR) {
```

```
  static const void *__rayRotate_fp[] = {
     "fp30", __rayRotate_fp30,
     "fp40", __rayRotate_fp40,
     "arb", __rayRotate_arb,
     "ps20", __rayRotate_ps20,
     "ps2b", __rayRotate_ps2b,
     "ps2a", __rayRotate_ps2a,
     "ps30", __rayRotate_ps30,
     "cpu", (void *) __rayRotate_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__rayRotate_fp);

  __k->PushStream(R);
  __k->PushConstant(hdgM);
  __k->PushConstant(pitchM);
  __k->PushConstant(rollM);
  __k->PushOutput(rotR);
  __k->Map();

}

namespace {
        using namespace ::brook::desc;
        static const gpu_kernel_desc __dist2pt_ps20_desc = gpu_kernel_desc()
                .technique( gpu_technique_desc()
                        .pass( gpu_pass_desc(
                                "    ps_2_0\n"
                                "    def c1, -1, 9999, 0, 0\n"
                                "    dcl t0.xy\n"
                                "    dcl_2d s0\n"
                                "    texld r0, t0, s0\n"
                                "    add r0.xyz, -r0, c0\n"
                                "    add r0.w, r0.w, c1.x\n"
                                "    dp3 r0.x, r0, r0\n"
                                "    rsq r1.w, r0.x\n"
                                "    rcp r1.w, r1.w\n"
                                "    mul r0.w, r0.w, r0.w\n"
                                "    cmp r0.x, -r0.w, r1.w, c1.y\n"
                                "    mov r0.yzw, c1.z\n"
                                "    mov oC0, r0\n"
                                "\n"
                                " \n"
                                "//!!BRCC\n"
                                "//narg:3\n"
                                "//c:3:o\n"
                                "//s:4:hit\n"
                                "//o:1:dist\n"
                                "//workspace:1024\n"
                                "//!!multipleOutputInfo:0:1:\n"
                                "//!!fullAddressTrans:0:\n"
                                "//!!reductionFactor:0:\n"
                                "")
                                .constant(1, 0)
                                .sampler(2, 0)
                                .interpolant(2, kStreamInterpolant_Position)
                                .output(3, 0)
                        )
                );
        static const void* __dist2pt_ps20 = &__dist2pt_ps20_desc;
}

static const char *__dist2pt_ps2b= NULL;
static const char *__dist2pt_ps2a= NULL;
static const char *__dist2pt_ps30= NULL;
static const char *__dist2pt_fp30= NULL;
static const char *__dist2pt_fp40= NULL;
static const char *__dist2pt_arb= NULL;
static const char *__dist2pt_cpu= NULL;
```

```
void  dist2pt (const float3  o,
               ::brook::stream hit,
               ::brook::stream dist) {
  static const void *__dist2pt_fp[] = {
     "fp30", __dist2pt_fp30,
     "fp40", __dist2pt_fp40,
     "arb", __dist2pt_arb,
     "ps20", __dist2pt_ps20,
     "ps2b", __dist2pt_ps2b,
     "ps2a", __dist2pt_ps2a,
     "ps30", __dist2pt_ps30,
     "cpu", (void *) __dist2pt_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__dist2pt_fp);

  __k->PushConstant(o);
  __k->PushStream(hit);
  __k->PushOutput(dist);
  __k->Map();

}

namespace {
        using namespace ::brook::desc;
        static const gpu_kernel_desc __minRange_ps20_desc = gpu_kernel_desc()
                .technique( gpu_technique_desc()
                        .reduction_factor(2)
                        .pass( gpu_pass_desc(
                                "    ps_2_0\n"
                                "    def c0, 0, 0, 0, 0\n"
                                "    dcl t0.xy\n"
                                "    dcl t1.xy\n"
                                "    dcl_2d s0\n"
                                "    dcl_2d s1\n"
                                "    texld r2, t0, s0\n"
                                "    texld r1, t1, s1\n"
                                "    min r0.x, r1.x, r2.x\n"
                                "    mov r0.yzw, c0.x\n"
                                "    mov oC0, r0\n"
                                "\n"
                                " \n"
                                "//!!BRCC\n"
                                "//narg:2\n"
                                "//s:1:range\n"
                                "//s:1:min_range\n"
                                "//workspace:1024\n"
                                "//!!multipleOutputInfo:0:1:\n"
                                "//!!fullAddressTrans:0:\n"
                                "//!!reductionFactor:2:\n"
                                "")
                                .sampler(0, 0)
                                .sampler(0, 1)
                                .interpolant(0, 0)
                                .interpolant(0, 1)
                                .output(0, 0)
                        )
                )
                .technique( gpu_technique_desc()
                        .reduction_factor(3)
                        .pass( gpu_pass_desc(
                                "    ps_2_0\n"
                                "    def c0, 0, 0, 0, 0\n"
                                "    dcl t0.xy\n"
                                "    dcl t1.xy\n"
                                "    dcl t2.xy\n"
                                "    dcl_2d s0\n"
                                "    dcl_2d s1\n"
                                "    texld r2, t0, s0\n"
```

```
                        "    texld r0, t1, s0\n"
                        "    texld r1, t2, s1\n"
                        "    min r1.w, r0.x, r2.x\n"
                        "    min r0.x, r1.x, r1.w\n"
                        "    mov r0.yzw, c0.x\n"
                        "    mov oC0, r0\n"
                        "\n"
                        " \n"
                        "//!!BRCC\n"
                        "//narg:2\n"
                        "//s:1:range\n"
                        "//s:1:min_range\n"
                        "//workspace:1024\n"
                        "//!!multipleOutputInfo:0:1:\n"
                        "//!!fullAddressTrans:0:\n"
                        "//!!reductionFactor:3:\n"
                        "")
                        .sampler(0, 0)
                        .sampler(0, 1)
                        .interpolant(0, 0)
                        .interpolant(0, 1)
                        .interpolant(0, 2)
                        .output(0, 0)
                )
        )
        .technique( gpu_technique_desc()
                .reduction_factor(4)
                .pass( gpu_pass_desc(
                        "    ps_2_0\n"
                        "    def c0, 0, 0, 0, 0\n"
                        "    dcl t0.xy\n"
                        "    dcl t1.xy\n"
                        "    dcl t2.xy\n"
                        "    dcl t3.xy\n"
                        "    dcl_2d s0\n"
                        "    dcl_2d s1\n"
                        "    texld r3, t0, s0\n"
                        "    texld r2, t1, s0\n"
                        "    texld r0, t2, s0\n"
                        "    texld r1, t3, s1\n"
                        "    min r0.w, r2.x, r3.x\n"
                        "    min r1.w, r0.x, r0.w\n"
                        "    min r0.x, r1.x, r1.w\n"
                        "    mov r0.yzw, c0.x\n"
                        "    mov oC0, r0\n"
                        "\n"
                        " \n"
                        "//!!BRCC\n"
                        "//narg:2\n"
                        "//s:1:range\n"
                        "//s:1:min_range\n"
                        "//workspace:1024\n"
                        "//!!multipleOutputInfo:0:1:\n"
                        "//!!fullAddressTrans:0:\n"
                        "//!!reductionFactor:4:\n"
                        "")
                        .sampler(0, 0)
                        .sampler(0, 1)
                        .interpolant(0, 0)
                        .interpolant(0, 1)
                        .interpolant(0, 2)
                        .interpolant(0, 3)
                        .output(0, 0)
                )
        )
        .technique( gpu_technique_desc()
                .reduction_factor(5)
                .pass( gpu_pass_desc(
```

```
                        "    ps_2_0\n"
                        "    def c0, 0, 0, 0, 0\n"
                        "    dcl t0.xy\n"
                        "    dcl t1.xy\n"
                        "    dcl t2.xy\n"
                        "    dcl t3.xy\n"
                        "    dcl t4.xy\n"
                        "    dcl_2d s0\n"
                        "    dcl_2d s1\n"
                        "    texld r4, t0, s0\n"
                        "    texld r3, t1, s0\n"
                        "    texld r2, t2, s0\n"
                        "    texld r0, t3, s0\n"
                        "    texld r1, t4, s1\n"
                        "    min r1.w, r3.x, r4.x\n"
                        "    min r0.w, r2.x, r1.w\n"
                        "    min r1.w, r0.x, r0.w\n"
                        "    min r0.x, r1.x, r1.w\n"
                        "    mov r0.yzw, c0.x\n"
                        "    mov oC0, r0\n"
                        "\n"
                        " \n"
                        "//!!BRCC\n"
                        "//narg:2\n"
                        "//s:1:range\n"
                        "//s:1:min_range\n"
                        "//workspace:1024\n"
                        "//!!multipleOutputInfo:0:1:\n"
                        "//!!fullAddressTrans:0:\n"
                        "//!!reductionFactor:5:\n"
                        "")
                        .sampler(0, 0)
                        .sampler(0, 1)
                        .interpolant(0, 0)
                        .interpolant(0, 1)
                        .interpolant(0, 2)
                        .interpolant(0, 3)
                        .interpolant(0, 4)
                        .output(0, 0)
                )
        )
        .technique( gpu_technique_desc()
                .reduction_factor(6)
                .pass( gpu_pass_desc(
                        "    ps_2_0\n"
                        "    def c0, 0, 0, 0, 0\n"
                        "    dcl t0.xy\n"
                        "    dcl t1.xy\n"
                        "    dcl t2.xy\n"
                        "    dcl t3.xy\n"
                        "    dcl t4.xy\n"
                        "    dcl t5.xy\n"
                        "    dcl_2d s0\n"
                        "    dcl_2d s1\n"
                        "    texld r5, t0, s0\n"
                        "    texld r4, t1, s0\n"
                        "    texld r3, t2, s0\n"
                        "    texld r2, t3, s0\n"
                        "    texld r0, t4, s0\n"
                        "    texld r1, t5, s1\n"
                        "    min r0.w, r4.x, r5.x\n"
                        "    min r1.w, r3.x, r0.w\n"
                        "    min r0.w, r2.x, r1.w\n"
                        "    min r1.w, r0.x, r0.w\n"
                        "    min r0.x, r1.x, r1.w\n"
                        "    mov r0.yzw, c0.x\n"
                        "    mov oC0, r0\n"
                        "\n"
```

```
                        " \n"
                        "//!!BRCC\n"
                        "//narg:2\n"
                        "//s:1:range\n"
                        "//s:1:min_range\n"
                        "//workspace:1024\n"
                        "//!!multipleOutputInfo:0:1:\n"
                        "//!!fullAddressTrans:0:\n"
                        "//!!reductionFactor:6:\n"
                        "")
                        .sampler(0, 0)
                        .sampler(0, 1)
                        .interpolant(0, 0)
                        .interpolant(0, 1)
                        .interpolant(0, 2)
                        .interpolant(0, 3)
                        .interpolant(0, 4)
                        .interpolant(0, 5)
                        .output(0, 0)
                )
        )
        .technique( gpu_technique_desc()
                .reduction_factor(7)
                .pass( gpu_pass_desc(
                        "    ps_2_0\n"
                        "    def c0, 0, 0, 0, 0\n"
                        "    dcl t0.xy\n"
                        "    dcl t1.xy\n"
                        "    dcl t2.xy\n"
                        "    dcl t3.xy\n"
                        "    dcl t4.xy\n"
                        "    dcl t5.xy\n"
                        "    dcl t6.xy\n"
                        "    dcl_2d s0\n"
                        "    dcl_2d s1\n"
                        "    texld r6, t0, s0\n"
                        "    texld r5, t1, s0\n"
                        "    texld r4, t2, s0\n"
                        "    texld r3, t3, s0\n"
                        "    texld r2, t4, s0\n"
                        "    texld r0, t5, s0\n"
                        "    texld r1, t6, s1\n"
                        "    min r1.w, r5.x, r6.x\n"
                        "    min r0.w, r4.x, r1.w\n"
                        "    min r1.w, r3.x, r0.w\n"
                        "    min r0.w, r2.x, r1.w\n"
                        "    min r1.w, r0.x, r0.w\n"
                        "    min r0.x, r1.x, r1.w\n"
                        "    mov r0.yzw, c0.x\n"
                        "    mov oC0, r0\n"
                        "\n"
                        " \n"
                        "//!!BRCC\n"
                        "//narg:2\n"
                        "//s:1:range\n"
                        "//s:1:min_range\n"
                        "//workspace:1024\n"
                        "//!!multipleOutputInfo:0:1:\n"
                        "//!!fullAddressTrans:0:\n"
                        "//!!reductionFactor:7:\n"
                        "")
                        .sampler(0, 0)
                        .sampler(0, 1)
                        .interpolant(0, 0)
                        .interpolant(0, 1)
                        .interpolant(0, 2)
                        .interpolant(0, 3)
                        .interpolant(0, 4)
```

```
                                .interpolant(0, 5)
                                .interpolant(0, 6)
                                .output(0, 0)
                        )
                )
                .technique( gpu_technique_desc()
                        .reduction_factor(8)
                        .pass( gpu_pass_desc(
                                "    ps_2_0\n"
                                "    def c0, 0, 0, 0, 0\n"
                                "    dcl t0.xy\n"
                                "    dcl t1.xy\n"
                                "    dcl t2.xy\n"
                                "    dcl t3.xy\n"
                                "    dcl t4.xy\n"
                                "    dcl t5.xy\n"
                                "    dcl t6.xy\n"
                                "    dcl t7.xy\n"
                                "    dcl_2d s0\n"
                                "    dcl_2d s1\n"
                                "    texld r7, t0, s0\n"
                                "    texld r6, t1, s0\n"
                                "    texld r5, t2, s0\n"
                                "    texld r4, t3, s0\n"
                                "    texld r3, t4, s0\n"
                                "    texld r2, t5, s0\n"
                                "    texld r0, t6, s0\n"
                                "    texld r1, t7, s1\n"
                                "    min r0.w, r6.x, r7.x\n"
                                "    min r1.w, r5.x, r0.w\n"
                                "    min r0.w, r4.x, r1.w\n"
                                "    min r1.w, r3.x, r0.w\n"
                                "    min r0.w, r2.x, r1.w\n"
                                "    min r1.w, r0.x, r0.w\n"
                                "    min r0.x, r1.x, r1.w\n"
                                "    mov r0.yzw, c0.x\n"
                                "    mov oC0, r0\n"
                                "\n"
                                " \n"
                                "//!!BRCC\n"
                                "//narg:2\n"
                                "//s:1:range\n"
                                "//s:1:min_range\n"
                                "//workspace:1024\n"
                                "//!!multipleOutputInfo:0:1:\n"
                                "//!!fullAddressTrans:0:\n"
                                "//!!reductionFactor:8:\n"
                                "")
                                .sampler(0, 0)
                                .sampler(0, 1)
                                .interpolant(0, 0)
                                .interpolant(0, 1)
                                .interpolant(0, 2)
                                .interpolant(0, 3)
                                .interpolant(0, 4)
                                .interpolant(0, 5)
                                .interpolant(0, 6)
                                .interpolant(0, 7)
                                .output(0, 0)
                        )
                );
        static const void* __minRange_ps20 = &__minRange_ps20_desc;
}

static const char *__minRange_ps2b= NULL;
static const char *__minRange_ps2a= NULL;
static const char *__minRange_ps30= NULL;
static const char *__minRange_fp30= NULL;
```

```
static const char *__minRange_fp40= NULL;
static const char *__minRange_arb= NULL;
static const char *__minRange_cpu= NULL;
void  minRange (::brook::stream range,
                ::brook::stream min_range) {
  static const void *__minRange_fp[] = {
     "fp30", __minRange_fp30,
     "fp40", __minRange_fp40,
     "arb", __minRange_arb,
     "ps20", __minRange_ps20,
     "ps2b", __minRange_ps2b,
     "ps2a", __minRange_ps2a,
     "ps30", __minRange_ps30,
     "cpu", (void *) __minRange_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__minRange_fp);

  __k->PushStream(range);
  __k->PushReduce(&min_range, __BRTReductionType(&min_range));
  __k->Reduce();

}

void  minRange (::brook::stream range,
                float  & min_range) {
  static const void *__minRange_fp[] = {
     "fp30", __minRange_fp30,
     "fp40", __minRange_fp40,
     "arb", __minRange_arb,
     "ps20", __minRange_ps20,
     "ps2b", __minRange_ps2b,
     "ps2a", __minRange_ps2a,
     "ps30", __minRange_ps30,
     "cpu", (void *) __minRange_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__minRange_fp);

  __k->PushStream(range);
  __k->PushReduce(&min_range, __BRTReductionType(&min_range));
  __k->Reduce();

}


#define NUM_RAYS 65

int  main()
{
  ::brook::stream triangle(::brook::getStreamType(( Triangle  *)0), 1 , NUM_TRIANGLES,-1);
  Triangle  tridat[1][NUM_TRIANGLES];
  ::brook::stream ray(::brook::getStreamType(( float3  *)0), NUM_RAYS , 1,-1);
  float3  raydat[NUM_RAYS][1];
  ::brook::stream rotated_ray(::brook::getStreamType(( float3  *)0), NUM_RAYS , 1,-1);
  ::brook::stream hit(::brook::getStreamType(( float4  *)0), NUM_RAYS , NUM_TRIANGLES,-1);
  ::brook::stream range(::brook::getStreamType(( float  *)0), NUM_RAYS , NUM_TRIANGLES,-1);
  ::brook::stream radalt(::brook::getStreamType(( float  *)0), 1,-1);
  float  radalt_out[1];
  float  msec_IO;
  float  msec_kernel;
  float  msec_total;
  float  msec_IO_sum = 0.000000f;
  float  msec_kernel_sum = 0.000000f;
  float  msec_total_sum = 0.000000f;
  int  j;
  int  k;
  double  HEADING = 0.000000f;
  double  PITCH = 0.000000f;
  double  ROLL = 0.000000f;
```

```
float  LONGITUDE = 0.000000f;
float  LATITUDE = 0.000000f;
float  ALTITUDE = 0.000000f;
float3  origin = float3 (LONGITUDE,LATITUDE,ALTITUDE);
float3  hdg_matrix;
float3  pitch_matrix;
float3  roll_matrix;
float  hdg_angle_sin;
float  hdg_angle_cos;
float  pitch_angle_sin;
float  pitch_angle_cos;
float  roll_angle_sin;
float  roll_angle_cos;
double  beam_angle_rad;
double  beam_angle_deg;
float  beam_angle_sin;
float  beam_angle_cos;
const double  beam_angle_45 = 45.000000f;
const float  beam_angle_offset = (float ) (sin(DEG2RAD(beam_angle_45)));
const double  beam_angle_22pt5 = 22.500000f;
const float  offset_22pt5_cos = (float ) (cos(DEG2RAD(beam_angle_22pt5)));
const float  offset_22pt5_sin = (float ) (sin(DEG2RAD(beam_angle_22pt5)));
float3  CENTER_RAY = float3 (0.000000f,0.000000f,-1.000000f);

raydat[0][0] = CENTER_RAY;
beam_angle_deg = 5.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[1][0] = float3 (beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[2][0] = float3 (0.000000f,beam_angle_sin,-beam_angle_cos);
raydat[3][0] = float3 (-beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[4][0] = float3 (0.000000f,-beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 10.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[5][0] = float3 (beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[6][0] = float3 (beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[7][0] = float3 (-beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[8][0] = float3 (-beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 25.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[9][0] = float3 (0.000000f,-beam_angle_sin,-beam_angle_cos);
raydat[10][0] = float3 (beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[11][0] = float3 (beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[12][0] = float3 (beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[13][0] = float3 (0.000000f,beam_angle_sin,-beam_angle_cos);
raydat[14][0] = float3 (-beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[15][0] = float3 (-beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[16][0] = float3 (-beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 20.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[17][0] = float3 (0.000000f,-beam_angle_sin,-beam_angle_cos);
raydat[18][0] = float3 (beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
```

```
raydat[19][0] = float3 (beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[20][0] = float3 (beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[21][0] = float3 (0.000000f,beam_angle_sin,-beam_angle_cos);
raydat[22][0] = float3 (-beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[23][0] = float3 (-beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[24][0] = float3 (-beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 15.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[25][0] = float3 (0.000000f,-beam_angle_sin,-beam_angle_cos);
raydat[26][0] = float3 (beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[27][0] = float3 (beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[28][0] = float3 (beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[29][0] = float3 (0.000000f,beam_angle_sin,-beam_angle_cos);
raydat[30][0] = float3 (-beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[31][0] = float3 (-beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[32][0] = float3 (-beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 5.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[33][0] = float3 (beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[34][0] = float3 (beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[35][0] = float3 (-beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[36][0] = float3 (-beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 10.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[37][0] = float3 (beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[38][0] = float3 (0.000000f,beam_angle_sin,-beam_angle_cos);
raydat[39][0] = float3 (-beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[40][0] = float3 (0.000000f,-beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 15.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[41][0] = float3 (offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[42][0] = float3 (offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[43][0] = float3 (-offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[44][0] = float3 (-offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[45][0] = float3 (-offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[46][0] = float3 (-offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[47][0] = float3 (offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[48][0] = float3 (offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 20.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
```

```
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[49][0] = float3 (offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[50][0] = float3 (offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[51][0] = float3 (-offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[52][0] = float3 (-offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[53][0] = float3 (-offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[54][0] = float3 (-offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[55][0] = float3 (offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[56][0] = float3 (offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 25.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[57][0] = float3 (offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[58][0] = float3 (offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[59][0] = float3 (-offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[60][0] = float3 (-offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[61][0] = float3 (-offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[62][0] = float3 (-offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[63][0] = float3 (offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[64][0] = float3 (offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
for (k = 0; k < NUM_TRIANGLES; k++)
{
  tridat[0][k].v0 = float3 (terrain[k][V0][X],terrain[k][V0][Y],terrain[k][V0][Z]);
  tridat[0][k].v1 = float3 (terrain[k][V1][X],terrain[k][V1][Y],terrain[k][V1][Z]);
  tridat[0][k].v2 = float3 (terrain[k][V2][X],terrain[k][V2][Y],terrain[k][V2][Z]);
}

streamRead(ray,raydat);
streamRead(triangle,tridat);
for (j = 0; j < NUM_WAYPOINTS; j++)
{
  msec_IO = 0.000000f;
  msec_kernel = 0.000000f;
  msec_total = 0.000000f;
  resetTimer();
  HEADING = flightpath[j].heading;
  PITCH = flightpath[j].pitch;
  ROLL = flightpath[j].roll;
  LONGITUDE = flightpath[j].longitude;
  LATITUDE = flightpath[j].latitude;
  ALTITUDE = flightpath[j].altitude;
  origin = float3 (LONGITUDE,LATITUDE,ALTITUDE);
  hdg_angle_sin = (float ) (sin(DEG2RAD(HDG2ANGLE(HEADING))));
  hdg_angle_cos = (float ) (cos(DEG2RAD(HDG2ANGLE(HEADING))));
  pitch_angle_sin = (float ) (sin(DEG2RAD(-PITCH)));
  pitch_angle_cos = (float ) (cos(DEG2RAD(-PITCH)));
  roll_angle_sin = (float ) (sin(DEG2RAD(ROLL)));
  roll_angle_cos = (float ) (cos(DEG2RAD(ROLL)));
  hdg_matrix = float3 (hdg_angle_cos,hdg_angle_sin,-hdg_angle_sin);
  pitch_matrix = float3 (pitch_angle_cos,pitch_angle_sin,-pitch_angle_sin);
  roll_matrix = float3 (roll_angle_cos,roll_angle_sin,-roll_angle_sin);
```

```
    msec_IO = getTimer();
    rayRotate(ray,hdg_matrix,pitch_matrix,roll_matrix,rotated_ray);
    rayIntersect(origin,rotated_ray,triangle,hit);
    dist2pt(origin,hit,range);
    minRange(range,radalt);
    msec_kernel = getTimer() - msec_IO;
    streamWrite(radalt,radalt_out);
    msec_total = getTimer();
    msec_IO = msec_total - msec_kernel;
    msec_IO_sum += msec_IO;
    msec_kernel_sum += msec_kernel;
    msec_total_sum += msec_total;
  }

  printf("Rays: %d, Triangles: %d\n",NUM_RAYS,NUM_TRIANGLES);
  printf("Avg kernel time = %3.3f msec\n",msec_kernel_sum / NUM_WAYPOINTS);
  printf("Avg IO time     = %3.3f msec\n",msec_IO_sum / NUM_WAYPOINTS);
  printf("Avg Total time  = %3.3f msec\n",msec_total_sum / NUM_WAYPOINTS);
  return 0;
}
```

Listing 10. BRCC CPU Source Code (monitor.cpp)

```
//////////////////////////////////////////
// Generated by BRCC v0.1
// BRCC Compiled on: Feb 21 2005 22:01:25
//////////////////////////////////////////

#include <brook/brook.hpp>
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include "terrain32.h"

#include "flightpath.h"

#define DEG2RAD(x) (((x)*3.1415926535898)/180.0)

#define HDG2ANGLE(x) (-x + 90.0)

typedef struct triangle_t {
  float3  v0;
  float3  v1;
  float3  v2;
} Triangle;typedef struct __cpustruct_triangle_t {
__BrtFloat3 v0;
__BrtFloat3 v1;
__BrtFloat3 v2;
}
__cpustruct_Triangle;

namespace brook {
        template<> const StreamType* getStreamType(Triangle*) {
                static const StreamType result[] = {__BRTFLOAT3, __BRTFLOAT3, __BRTFLOAT3,
                    __BRTNONE};
                return result;
        }
}

void  resetTimer(void );
float  getTimer(void );
static const char *__rayIntersect_ps20= NULL;
static const char *__rayIntersect_ps2b= NULL;
```

```
static const char *__rayIntersect_ps2a= NULL;
static const char *__rayIntersect_ps30= NULL;
static const char *__rayIntersect_fp30= NULL;
static const char *__rayIntersect_fp40= NULL;
static const char *__rayIntersect_arb= NULL;
void  __rayIntersect_cpu_inner(const __BrtFloat3  &o,
                               const __BrtFloat3  &R,
                               const __cpustruct_Triangle &T,
                               __BrtFloat4  &Hit)
{
  __BrtFloat3  u;
  __BrtFloat3  v;
  __BrtFloat3  n;
  __BrtFloat3  w0;
  __BrtFloat3  w;
  __BrtFloat1  r;
  __BrtFloat1  a;
  __BrtFloat1  b;
  __BrtFloat1  uu;
  __BrtFloat1  uv;
  __BrtFloat1  vv;
  __BrtFloat1  wu;
  __BrtFloat1  wv;
  __BrtFloat1  D;
  __BrtFloat1  s;
  __BrtFloat1  t;
  __BrtFloat3  I;

  u = T.v1 - T.v0;
  v = T.v2 - T.v0;
  n = __cross_cpu_inner(u,v);
  if (n.swizzle1(maskX) == __BrtFloat1(0.000000f) && n.swizzle1(maskY) ==
      __BrtFloat1(0.000000f)
      && n.swizzle1(maskZ) == __BrtFloat1(0.000000f))
  {
    Hit.mask1(-__BrtFloat1(1.000000f),maskW);
  }

  else
  {
    w0 = o - T.v0;
    a = -__dot_cpu_inner(n,w0);
    b = __dot_cpu_inner(n,R);
    if (__abs_cpu_inner(b) < __BrtFloat1(0.000010f))
    {
      Hit.mask1(-__BrtFloat1(2.000000f),maskW);
    }

    else
    {
      r = a / b;
      if (r < __BrtFloat1(0.000000f))
      {
        Hit.mask1(-__BrtFloat1(3.000000f),maskW);
      }

      else
      {
        I = o + r * R;
        uu = __dot_cpu_inner(u,u);
        uv = __dot_cpu_inner(u,v);
        vv = __dot_cpu_inner(v,v);
        w = I - T.v0;
        wu = __dot_cpu_inner(w,u);
        wv = __dot_cpu_inner(w,v);
        D = uv * uv - uu * vv;
        s = (uv * wv - vv * wu) / D;
        if (s < __BrtFloat1(0.000000f) || s > __BrtFloat1(1.000000f))
```

```
        {
          Hit.mask1(-__BrtFloat1(4.000000f),maskW);
        }

        else
        {
          t = (uv * wu - uu * wv) / D;
          if (t < __BrtFloat1(0.000000f) || s + t > __BrtFloat1(1.000000f))
          {
            Hit.mask1(-__BrtFloat1(5.000000f),maskW);
          }

          else
          {
            Hit.mask1(__BrtFloat1(1.000000f),maskW);
            Hit.mask1(I.swizzle1(maskX),maskX);
            Hit.mask1(I.swizzle1(maskY),maskY);
            Hit.mask1(I.swizzle1(maskZ),maskZ);
          }

        }

      }

    }

  }

}
void  __rayIntersect_cpu(::brook::Kernel *__k, const std::vector<void *>&args)
{
  __BrtFloat3 *arg_o = (__BrtFloat3 *) args[0];
  ::brook::StreamInterface *arg_R = (::brook::StreamInterface *) args[1];
  ::brook::StreamInterface *arg_T = (::brook::StreamInterface *) args[2];
  ::brook::StreamInterface *arg_Hit = (::brook::StreamInterface *) args[3];

  do {
    __rayIntersect_cpu_inner (*arg_o,
                              *(__BrtFloat3 *) __k->FetchElem(arg_R),
                              *(__cpustruct_Triangle*) __k->FetchElem(arg_T),
                              *(__BrtFloat4 *) __k->FetchElem(arg_Hit));
  } while (__k->Continue());
}

void  rayIntersect (const float3  o,
                ::brook::stream R,
                ::brook::stream T,
                ::brook::stream Hit) {
  static const void *__rayIntersect_fp[] = {
     "fp30", __rayIntersect_fp30,
     "fp40", __rayIntersect_fp40,
     "arb", __rayIntersect_arb,
     "ps20", __rayIntersect_ps20,
     "ps2b", __rayIntersect_ps2b,
     "ps2a", __rayIntersect_ps2a,
     "ps30", __rayIntersect_ps30,
     "cpu", (void *) __rayIntersect_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__rayIntersect_fp);

  __k->PushConstant(o);
  __k->PushStream(R);
  __k->PushStream(T);
  __k->PushOutput(Hit);
  __k->Map();

}
```

```
static const char *__rayRotate_ps20= NULL;
static const char *__rayRotate_ps2b= NULL;
static const char *__rayRotate_ps2a= NULL;
static const char *__rayRotate_ps30= NULL;
static const char *__rayRotate_fp30= NULL;
static const char *__rayRotate_fp40= NULL;
static const char *__rayRotate_arb= NULL;
void  __rayRotate_cpu_inner(const __BrtFloat3  &R,
                            const __BrtFloat3  &hdgM,
                            const __BrtFloat3  &pitchM,
                            const __BrtFloat3  &rollM,
                            __BrtFloat3  &rotR)
{
  __BrtFloat1  x_prod1;
  __BrtFloat1  x_prod2;
  __BrtFloat1  y_prod1;
  __BrtFloat1  y_prod2;

  x_prod1 = rollM.swizzle1(maskY) * pitchM.swizzle1(maskY) * hdgM.swizzle1(maskX)
            + rollM.swizzle1(maskX) * hdgM.swizzle1(maskZ);
  x_prod2 = rollM.swizzle1(maskX) * pitchM.swizzle1(maskY) * hdgM.swizzle1(maskX)
            + rollM.swizzle1(maskZ) * hdgM.swizzle1(maskZ);
  y_prod1 = rollM.swizzle1(maskY) * pitchM.swizzle1(maskY) * hdgM.swizzle1(maskY)
            + rollM.swizzle1(maskX) * hdgM.swizzle1(maskX);
  y_prod2 = rollM.swizzle1(maskX) * pitchM.swizzle1(maskY) * hdgM.swizzle1(maskY)
            + rollM.swizzle1(maskZ) * hdgM.swizzle1(maskX);
  rotR = __BrtFloat3 (R.swizzle1(maskX) * pitchM.swizzle1(maskX) * hdgM.swizzle1(maskX)
         + R.swizzle1(maskY) * x_prod1 + R.swizzle1(maskZ) * x_prod2,R.swizzle1(maskX)
         * pitchM.swizzle1(maskX) * hdgM.swizzle1(maskY) + R.swizzle1(maskY) * y_prod1
         + R.swizzle1(maskZ) * y_prod2,R.swizzle1(maskX) * pitchM.swizzle1(maskZ)
         + R.swizzle1(maskY) * rollM.swizzle1(maskY) * pitchM.swizzle1(maskX)
         + R.swizzle1(maskZ) * rollM.swizzle1(maskX) * pitchM.swizzle1(maskX));
}
void  __rayRotate_cpu(::brook::Kernel *__k, const std::vector<void *>&args)
{
  ::brook::StreamInterface *arg_R = (::brook::StreamInterface *) args[0];
  __BrtFloat3 *arg_hdgM = (__BrtFloat3 *) args[1];
  __BrtFloat3 *arg_pitchM = (__BrtFloat3 *) args[2];
  __BrtFloat3 *arg_rollM = (__BrtFloat3 *) args[3];
  ::brook::StreamInterface *arg_rotR = (::brook::StreamInterface *) args[4];

  do {
    __rayRotate_cpu_inner (*(__BrtFloat3 *) __k->FetchElem(arg_R),
                           *arg_hdgM,
                           *arg_pitchM,
                           *arg_rollM,
                           *(__BrtFloat3 *) __k->FetchElem(arg_rotR));
  } while (__k->Continue());
}

void  rayRotate (::brook::stream R,
                 const float3  hdgM,
                 const float3  pitchM,
                 const float3  rollM,
                 ::brook::stream rotR) {
  static const void *__rayRotate_fp[] = {
     "fp30", __rayRotate_fp30,
     "fp40", __rayRotate_fp40,
     "arb", __rayRotate_arb,
     "ps20", __rayRotate_ps20,
     "ps2b", __rayRotate_ps2b,
     "ps2a", __rayRotate_ps2a,
     "ps30", __rayRotate_ps30,
     "cpu", (void *) __rayRotate_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__rayRotate_fp);
```

```
  __k->PushStream(R);
  __k->PushConstant(hdgM);
  __k->PushConstant(pitchM);
  __k->PushConstant(rollM);
  __k->PushOutput(rotR);
  __k->Map();

}


static const char *__dist2pt_ps20= NULL;
static const char *__dist2pt_ps2b= NULL;
static const char *__dist2pt_ps2a= NULL;
static const char *__dist2pt_ps30= NULL;
static const char *__dist2pt_fp30= NULL;
static const char *__dist2pt_fp40= NULL;
static const char *__dist2pt_arb= NULL;
void  __dist2pt_cpu_inner(const __BrtFloat3  &o,
                          const __BrtFloat4  &hit,
                          __BrtFloat1  &dist)
{
  __BrtFloat3  diff;
  __BrtFloat3  target;

  if (hit.swizzle1(maskW) != __BrtFloat1(1.000000f))
  {
    dist = __BrtFloat1(9999.000000f);
  }

  else
  {
    target = __BrtFloat3 (hit.swizzle1(maskX),hit.swizzle1(maskY),hit.swizzle1(maskZ));
    diff = o - target;
    dist = __sqrt_cpu_inner(__dot_cpu_inner(diff,diff));
  }

}
void  __dist2pt_cpu(::brook::Kernel *__k, const std::vector<void *>&args)
{
  __BrtFloat3 *arg_o = (__BrtFloat3 *) args[0];
  ::brook::StreamInterface *arg_hit = (::brook::StreamInterface *) args[1];
  ::brook::StreamInterface *arg_dist = (::brook::StreamInterface *) args[2];

  do {
    __dist2pt_cpu_inner (*arg_o,
                         *(__BrtFloat4 *) __k->FetchElem(arg_hit),
                         *(__BrtFloat1 *) __k->FetchElem(arg_dist));
  } while (__k->Continue());
}

void  dist2pt (const float3  o,
               ::brook::stream hit,
               ::brook::stream dist) {
  static const void *__dist2pt_fp[] = {
     "fp30", __dist2pt_fp30,
     "fp40", __dist2pt_fp40,
     "arb", __dist2pt_arb,
     "ps20", __dist2pt_ps20,
     "ps2b", __dist2pt_ps2b,
     "ps2a", __dist2pt_ps2a,
     "ps30", __dist2pt_ps30,
     "cpu", (void *) __dist2pt_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__dist2pt_fp);

  __k->PushConstant(o);
  __k->PushStream(hit);
  __k->PushOutput(dist);
```

```
  __k->Map();

}


static const char *__minRange_ps20= NULL;
static const char *__minRange_ps2b= NULL;
static const char *__minRange_ps2a= NULL;
static const char *__minRange_ps30= NULL;
static const char *__minRange_fp30= NULL;
static const char *__minRange_fp40= NULL;
static const char *__minRange_arb= NULL;
void  __minRange_cpu_inner(const __BrtFloat1  &range,
                           __BrtFloat1  &min_range)
{
  if (range < min_range)
    min_range = range;
}
void  __minRange_cpu(::brook::Kernel *__k, const std::vector<void *>&args)
{
  ::brook::StreamInterface *arg_range = (::brook::StreamInterface *) args[0];
  ::brook::StreamInterface *arg_min_range = (::brook::StreamInterface *) args[1];

  do {
    __minRange_cpu_inner (*(__BrtFloat1 *) __k->FetchElem(arg_range),
                          *(__BrtFloat1 *) __k->FetchElem(arg_min_range));
  } while (__k->Continue());
}

void  minRange (::brook::stream range,
                ::brook::stream min_range) {
  static const void *__minRange_fp[] = {
     "fp30", __minRange_fp30,
     "fp40", __minRange_fp40,
     "arb", __minRange_arb,
     "ps20", __minRange_ps20,
     "ps2b", __minRange_ps2b,
     "ps2a", __minRange_ps2a,
     "ps30", __minRange_ps30,
     "cpu", (void *) __minRange_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__minRange_fp);

  __k->PushStream(range);
  __k->PushReduce(&min_range, __BRTReductionType(&min_range));
  __k->Reduce();

}

void  minRange (::brook::stream range,
                float  & min_range) {
  static const void *__minRange_fp[] = {
     "fp30", __minRange_fp30,
     "fp40", __minRange_fp40,
     "arb", __minRange_arb,
     "ps20", __minRange_ps20,
     "ps2b", __minRange_ps2b,
     "ps2a", __minRange_ps2a,
     "ps30", __minRange_ps30,
     "cpu", (void *) __minRange_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__minRange_fp);

  __k->PushStream(range);
  __k->PushReduce(&min_range, __BRTReductionType(&min_range));
  __k->Reduce();

}
```

```
#define NUM_RAYS 65

int  main()
{
  ::brook::stream triangle(::brook::getStreamType(( Triangle  *)0), 1 , NUM_TRIANGLES,-1);
  Triangle  tridat[1][NUM_TRIANGLES];
  ::brook::stream ray(::brook::getStreamType(( float3  *)0), NUM_RAYS , 1,-1);
  float3  raydat[NUM_RAYS][1];
  ::brook::stream rotated_ray(::brook::getStreamType(( float3  *)0), NUM_RAYS , 1,-1);
  ::brook::stream hit(::brook::getStreamType(( float4  *)0), NUM_RAYS , NUM_TRIANGLES,-1);
  ::brook::stream range(::brook::getStreamType(( float  *)0), NUM_RAYS , NUM_TRIANGLES,-1);
  ::brook::stream min_range(::brook::getStreamType(( float  *)0), NUM_RAYS ,
      NUM_TRIANGLES,-1);
  float  msec_IO;
  float  msec_kernel;
  float  msec_total;
  float  msec_IO_sum = 0.000000f;
  float  msec_kernel_sum = 0.000000f;
  float  msec_total_sum = 0.000000f;
  int  j;
  int  k;
  double  HEADING = 0.000000f;
  double  PITCH = 0.000000f;
  double  ROLL = 0.000000f;
  float  LONGITUDE = 0.000000f;
  float  LATITUDE = 0.000000f;
  float  ALTITUDE = 0.000000f;
  float3  origin = float3 (LONGITUDE,LATITUDE,ALTITUDE);
  float3  hdg_matrix;
  float3  pitch_matrix;
  float3  roll_matrix;
  float  hdg_angle_sin;
  float  hdg_angle_cos;
  float  pitch_angle_sin;
  float  pitch_angle_cos;
  float  roll_angle_sin;
  float  roll_angle_cos;
  double  beam_angle_rad;
  double  beam_angle_deg;
  float  beam_angle_sin;
  float  beam_angle_cos;
  const double  beam_angle_45 = 45.000000f;
  const float  beam_angle_offset = (float ) (sin(DEG2RAD(beam_angle_45)));
  const double  beam_angle_22pt5 = 22.500000f;
  const float  offset_22pt5_cos = (float ) (cos(DEG2RAD(beam_angle_22pt5)));
  const float  offset_22pt5_sin = (float ) (sin(DEG2RAD(beam_angle_22pt5)));
  float3  CENTER_RAY = float3 (0.000000f,0.000000f,-1.000000f);

  raydat[0][0] = CENTER_RAY;
  beam_angle_deg = 5.000000f;
  beam_angle_rad = DEG2RAD(beam_angle_deg);
  beam_angle_sin = (float ) (sin(beam_angle_rad));
  beam_angle_cos = (float ) (cos(beam_angle_rad));
  raydat[1][0] = float3 (beam_angle_sin,0.000000f,-beam_angle_cos);
  raydat[2][0] = float3 (0.000000f,beam_angle_sin,-beam_angle_cos);
  raydat[3][0] = float3 (-beam_angle_sin,0.000000f,-beam_angle_cos);
  raydat[4][0] = float3 (0.000000f,-beam_angle_sin,-beam_angle_cos);
  beam_angle_deg = 10.000000f;
  beam_angle_rad = DEG2RAD(beam_angle_deg);
  beam_angle_sin = (float ) (sin(beam_angle_rad));
  beam_angle_cos = (float ) (cos(beam_angle_rad));
  raydat[5][0] = float3 (beam_angle_offset * beam_angle_sin,-beam_angle_offset *
      beam_angle_sin,-beam_angle_cos);
  raydat[6][0] = float3 (beam_angle_offset * beam_angle_sin,beam_angle_offset *
      beam_angle_sin,-beam_angle_cos);
```

```
raydat[7][0] = float3 (-beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[8][0] = float3 (-beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 25.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[9][0] = float3 (0.000000f,-beam_angle_sin,-beam_angle_cos);
raydat[10][0] = float3 (beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[11][0] = float3 (beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[12][0] = float3 (beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[13][0] = float3 (0.000000f,beam_angle_sin,-beam_angle_cos);
raydat[14][0] = float3 (-beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[15][0] = float3 (-beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[16][0] = float3 (-beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 20.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[17][0] = float3 (0.000000f,-beam_angle_sin,-beam_angle_cos);
raydat[18][0] = float3 (beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[19][0] = float3 (beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[20][0] = float3 (beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[21][0] = float3 (0.000000f,beam_angle_sin,-beam_angle_cos);
raydat[22][0] = float3 (-beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[23][0] = float3 (-beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[24][0] = float3 (-beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 15.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[25][0] = float3 (0.000000f,-beam_angle_sin,-beam_angle_cos);
raydat[26][0] = float3 (beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[27][0] = float3 (beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[28][0] = float3 (beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[29][0] = float3 (0.000000f,beam_angle_sin,-beam_angle_cos);
raydat[30][0] = float3 (-beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[31][0] = float3 (-beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[32][0] = float3 (-beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 5.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[33][0] = float3 (beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[34][0] = float3 (beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[35][0] = float3 (-beam_angle_offset * beam_angle_sin,beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
raydat[36][0] = float3 (-beam_angle_offset * beam_angle_sin,-beam_angle_offset *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 10.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
```

```
raydat[37][0] = float3 (beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[38][0] = float3 (0.000000f,beam_angle_sin,-beam_angle_cos);
raydat[39][0] = float3 (-beam_angle_sin,0.000000f,-beam_angle_cos);
raydat[40][0] = float3 (0.000000f,-beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 15.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[41][0] = float3 (offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[42][0] = float3 (offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[43][0] = float3 (-offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[44][0] = float3 (-offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[45][0] = float3 (-offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[46][0] = float3 (-offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[47][0] = float3 (offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[48][0] = float3 (offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 20.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[49][0] = float3 (offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[50][0] = float3 (offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[51][0] = float3 (-offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[52][0] = float3 (-offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[53][0] = float3 (-offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[54][0] = float3 (-offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[55][0] = float3 (offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[56][0] = float3 (offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
beam_angle_deg = 25.000000f;
beam_angle_rad = DEG2RAD(beam_angle_deg);
beam_angle_sin = (float ) (sin(beam_angle_rad));
beam_angle_cos = (float ) (cos(beam_angle_rad));
raydat[57][0] = float3 (offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[58][0] = float3 (offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[59][0] = float3 (-offset_22pt5_sin * beam_angle_sin,offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[60][0] = float3 (-offset_22pt5_cos * beam_angle_sin,offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[61][0] = float3 (-offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
raydat[62][0] = float3 (-offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[63][0] = float3 (offset_22pt5_sin * beam_angle_sin,-offset_22pt5_cos *
    beam_angle_sin,-beam_angle_cos);
raydat[64][0] = float3 (offset_22pt5_cos * beam_angle_sin,-offset_22pt5_sin *
    beam_angle_sin,-beam_angle_cos);
for (k = 0; k < NUM_TRIANGLES; k++)
{
  tridat[0][k].v0 = float3 (terrain[k][V0][X],terrain[k][V0][Y],terrain[k][V0][Z]);
  tridat[0][k].v1 = float3 (terrain[k][V1][X],terrain[k][V1][Y],terrain[k][V1][Z]);
```

```
    tridat[0][k].v2 = float3 (terrain[k][V2][X],terrain[k][V2][Y],terrain[k][V2][Z]);
  }

  streamRead(ray,raydat);
  streamRead(triangle,tridat);
  for (j = 0; j < NUM_WAYPOINTS; j++)
  {
    msec_IO = 0.000000f;
    msec_kernel = 0.000000f;
    msec_total = 0.000000f;
    resetTimer();
    HEADING = flightpath[j].heading;
    PITCH = flightpath[j].pitch;
    ROLL = flightpath[j].roll;
    LONGITUDE = flightpath[j].longitude;
    LATITUDE = flightpath[j].latitude;
    ALTITUDE = flightpath[j].altitude;
    origin = float3 (LONGITUDE,LATITUDE,ALTITUDE);
    hdg_angle_sin = (float ) (sin(DEG2RAD(HDG2ANGLE(HEADING))));
    hdg_angle_cos = (float ) (cos(DEG2RAD(HDG2ANGLE(HEADING))));
    pitch_angle_sin = (float ) (sin(DEG2RAD(-PITCH)));
    pitch_angle_cos = (float ) (cos(DEG2RAD(-PITCH)));
    roll_angle_sin = (float ) (sin(DEG2RAD(ROLL)));
    roll_angle_cos = (float ) (cos(DEG2RAD(ROLL)));
    hdg_matrix = float3 (hdg_angle_cos,hdg_angle_sin,-hdg_angle_sin);
    pitch_matrix = float3 (pitch_angle_cos,pitch_angle_sin,-pitch_angle_sin);
    roll_matrix = float3 (roll_angle_cos,roll_angle_sin,-roll_angle_sin);
    msec_IO = getTimer();
    rayRotate(ray,hdg_matrix,pitch_matrix,roll_matrix,rotated_ray);
    rayIntersect(origin,rotated_ray,triangle,hit);
    dist2pt(origin,hit,range);
    minRange(range,min_range);
    msec_kernel = getTimer() - msec_IO;
    msec_total = getTimer();
    msec_IO = msec_total - msec_kernel;
    msec_IO_sum += msec_IO;
    msec_kernel_sum += msec_kernel;
    msec_total_sum += msec_total;
  }

  printf("Rays: %d, Triangles: %d\n",NUM_RAYS,NUM_TRIANGLES);
  printf("Avg kernel time = %3.3f msec\n",msec_kernel_sum / NUM_WAYPOINTS);
  printf("Avg IO time     = %3.3f msec\n",msec_IO_sum / NUM_WAYPOINTS);
  printf("Avg Total time  = %3.3f msec\n",msec_total_sum / NUM_WAYPOINTS);
  return 0;
}
```

Listing 11. Brook Timer Functions (timer.cpp)

```
// Copyright (c) 2003, Stanford University
// All rights reserved.

#include <assert.h>

#ifdef _WIN32

#include <windows.h>

static LARGE_INTEGER t;
static float         f;
static int           freq_init = 0;

void resetTimer(void) {
  if (!freq_init) {
    LARGE_INTEGER freq;
    assert (QueryPerformanceFrequency(&freq));
```

```
    f = (float) freq.QuadPart;
    freq_init = 1;
  }
  assert (QueryPerformanceCounter(&t));
}

float getTimer(void) {
  LARGE_INTEGER s;
  float d;
  assert (QueryPerformanceCounter(&s));

  d = ((float)(s.QuadPart - t.QuadPart)) / f;

  return (d*1000.0f);
}

#else

#include <sys/time.h>

static struct timeval t;

void resetTimer(void) {
  gettimeofday(&t);
}

float getTimer(void) {
  static struct timeval s;
  gettimeofday(&s);

  return (int) (s.tv_sec - t.tv_sec)*1000.0f +
    (s.tv_usec - t.tv_sec)/1000.0f;
}


#endif
```