Ace Lander: an Exploration in Software Engineering

An Honors Capstone Project

Presented to

The Department of Mathematics and Computer Science

Abilene Christian University

In Partial Fulfillment

of the Requirements for

University Honors

by

Jace Ryan Miller

April 2006

This Capstone Project, directed and approved by the candidate's committee, has been accepted by the Honors Program of Abilene Christian University in partial fulfillment of the requirements for the distinction

UNIVERSITY HONORS

_____
Director of the Honors Program

_____
Date

Advisory Committee

_____
Committee Chair

_____
Committee Member

_____
Committee Member

_____
Department Head

ABSTRACT

The typical undergraduate curriculum provides few opportunities for developing large and complete software projects. Working on Ace Lander, a real-time game application, provided me with some experience on such a project. Any real-time application contains several technical and organizational challenges. Addressing these issues requires use of proper algorithms, data structures, and consistent programming over several files. Confronting these problems in the context of a game supplies a little extra motivation that is not inherent in developing more mundane applications. Creating Ace Lander brought me a step closer to experiencing the type of programming done as part of the software development process in the real world as opposed to academia.

During the development of Ace Lander, I learned about what it takes to bring a software project from the beginning concept to the final product. I want to describe some of the challenges presented during the process of creating Ace Lander and the methods I used to overcome them. Development tools, major system components, and some algorithms will be included in this reflection on the software development process.

Developing a game, or any large piece of software for that matter, requires relying on libraries and using tools. I used Microsoft Visual C++ 2005 Express Edition as the development environment to help with source code management and to assist with the build process. OpenGL and the OpenGL Utility Toolkit were used for their primitive rendering and window handling capabilities, respectively. I chose the FMOD library to handle the basic loading and playing of sound files. The standard template library was used in several places, mainly for its reliable and flexible data structures. Install Creator was used to create a stand-alone installation program that was easy to distribute.

The scope of the Ace Lander project prompted the use of the software engineering tool of abstraction to manage levels of detail. To make it easier to perform common tasks, I began coding for this project by constructing a general framework that could be used to create generic OpenGL-based games. The general modules that make up the framework define mathematical types like vectors and matrices, control timing and game states, and load audio assets – functionality that is common to several types of games.

I chose C++ for the programming language because it is commonly used

in game development and it is the language I am most familiar with. However, when I began programming the framework, I had not yet had any formal training in C++. Throughout development, I would recognize areas of the code that could benefit from utilizing a feature provided by the C++ language. At one point, I realized there was much overlap in the code between different game states. Multiple instances of Ship and Level objects in the title screen, introduction, menu, and main game states were causing much confusion. Since each object was being created in much the same way, I used the singleton pattern to simplify the use of these objects by restricting them to one instance. The phrase “glorified global pointers” is a good description of the way I viewed using the singletons.

Another area where the choice of C++ was crucial to program design was in creating the game state machine, which demonstrates a practical application of function pointers. Game states are used to help separate the different functionality of a game. For instance, input should be handled differently in the contexts of choosing a menu item and piloting the spaceship. A simple implementation of game states can be achieved with a switch statement, but function pointers allow for a more elegant solution. Each game state object is derived from a GameState base object with pure virtual methods like HandleInput(), Update(), and Render(). The game state objects must implement these methods to be used by the framework's state machine. Depending on the current state of the game, the framework calls the correct state's overridden methods. Dividing the game logic into states allows similar code to be grouped

and helps with the overall organization of the code.

The audio manager handles all aspects of playing sounds, including loading, keeping track of, and unloading multiple sound files. The audio manager is a main component of the framework and it demonstrates good use of the standard template library. The FMOD library is good at handling various types of sound files, but using the raw library function calls was difficult. The functions required storing information that a gameplay programmer should not have to deal with, such as integers representing the channels a particular sound is mapped to before playing. Also, different types of data structures were used for different sound files, further complicating the direct use of FMOD. To make using FMOD easier, I created the audio manager to encapsulate all the functionality that I needed without the cumbersome interface. To load a song using the audio manager, only the path of the file to be loaded and a string identifier, like a title, is required. The audio manager takes care of allocating space for new file handles and keeps track of details specific to different audio types – all without demanding the programmer's attention.

Ace Lander employs a real-time simulation. This is a different model of interaction than the event driven model used by many Windows programs. One of the challenges of real-time simulation on a computer is timing related issues. Maintaining a constant rate of gameplay as perceived by the user across different CPU speeds requires that time sensitive events be handled appropriately. One cannot simply tell the computer to execute instructions as fast as possible using a simple loop. For each step of the simulation, time-

sensitive calculations must be adjusted to account for the amount of time that has passed since the last step. To help with timing issues, I created a Timer object. I used the Windows system call QueryPerformanceCounter() inside the Timer object to obtain the time between frames. This call provides timing resolution of less than one microsecond. The introductory cinematic sequence was the source of the most complex combination of timing and camera transitioning in the game. To help with tracking the different time-critical signals, I created an object called Event that could be set like an alarm clock and periodically checked for expiration.

A simple Newtonian physics model was simulated to provide a realistic interaction between the ship and the game environment. Simulating Newtonian physics is less complicated than it sounds. The velocity of the ship is updated each frame based on forces like gravity and the thrust due to the ship's engines firing. The position of the ship is then updated based on the velocity. The simulation is only slightly complicated when compensating for a variable frame rate. Changing frame rates is really a timing issue, and adjusting the simulation step proportionally to the time elapsed per frame achieves the desired result.

Originally, the exhaust of the ship was represented by a triangle. Later, a particle system was implemented to make the exhaust look more realistic. The particle system creates and destroys objects representing individual particles. Each particle has a position, velocity, color, and lifetime. The particles are drawn as single pixels shooting from a point attached to the ship's position. They are given a random color in the range of red to yellow to produce a color mimicking

the typical rocket exhaust plume.

Most of the interactive gameplay occurs in the piloting state of Ace Lander. This state gives the player control over the ship and tests their piloting skills with a series of increasingly difficult levels. To achieve a successful landing, the ship must touch down on top of the platform with a relatively slow velocity. Also, the base of the ship must be oriented near horizontal and positioned so that the center of the ship does not hang over the edge of the platform.

Levels are loaded from text files that describe starting positions for the ship and platform as well as a list of polygons that make up the terrain. Each polygon is specified by a list of vertices. During initial level testing, collisions between the ship and invisible line segments would occur. This problem was caused by line segments created from the last vertex of one polygon and the first vertex of the next polygon in the file. These segments were being tested for collision with the ship without being rendered. The issue was resolved in code by ignoring collisions of line segments formed at boundaries of polygon definitions.

Most levels begin with the ship positioned far from the landing platform. As the ship approaches the platform, the camera zooms in for a closer view. The width to height ratio of the screen is kept at a constant 4 : 3. The distance between the ship and the platform determines the size of the viewport. At first, I chose to keep the center of the camera always focused on the ship. This led to the platform leaving the viewport at times. Feedback from a friend convinced me to change the camera focus so the platform was always visible. The final version centers the camera at a point one third of the way toward the platform along an

imaginary line connecting the center of the ship with the center of the platform.

Collision detection is a major area of current interest in the game development community. A technical article included with the Microsoft DirectX 9.0 SDK Documentation describes the top ten issues with current Windows titles. Number one on the list is CPU limiting problems, in which collision detection plays a part. The following quote from the article states the importance of implementing efficient collision detection algorithms.

> "Because most titles are CPU-limited, the biggest performance wins come from optimizations made to CPU-intensive game systems. Typically, the AI or physics systems and the associated collision detection logic are the primary consumer of CPU cycles in well-behaving Microsoft Direct3D applications. Any work to improve these systems can improve the overall performance for the game."

Although line segment collision detection is simpler than the types of collision detection plaguing current games, it is integral to the gameplay of Ace Lander, and not a trivial problem. Collision detection between the ship and the terrain is performed each step of the simulation to figure out if the ship should crash or land safely. To implement collision detection, I used an algorithm based off of one described in the computational geometry chapter of *Introduction to Algorithms*. An overview of this algorithm follows. First, the coordinates of the two line segments that are being tested for intersection must be known. The relative orientation, either clockwise or counterclockwise, of two consecutive line segments determined by three points can be computed using the cross product

of two vectors whose endpoints are the endpoints of the line segment and whose origins are the point shared by the line segments. A line segment straddles a line if one point of the segment lies on one side of the line and the other point of the segment lies on the other side of the line. Two line segments intersect when each line segment straddles the other.

Rendering concave polygons with holes was one of the more difficult technical issues I had to deal with. Figure 1 shows an initial attempt at rendering the title screen, which is constructed as a polygon with holes. Obviously, the

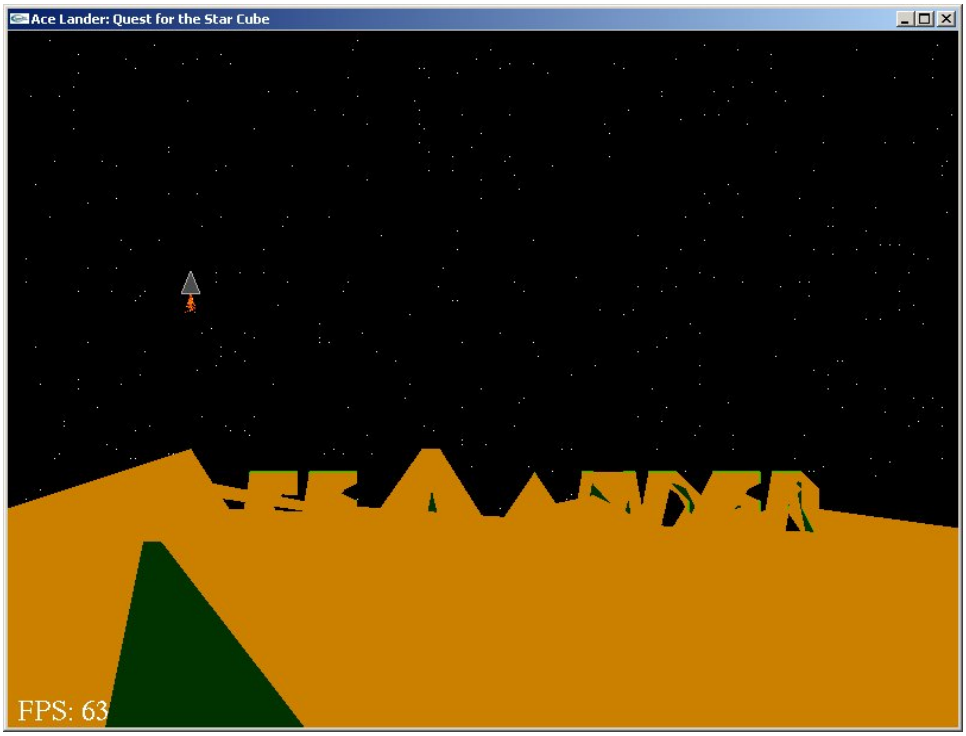

Figure 1: Naive Rendering of Concave Polygons with Holes

jagged orange shape was not the intended result. The problem lay in that OpenGL does not natively render concave polygons, as it does convex polygons.

I did some research and found two methods of rendering concave polygons. I chose to use an algorithm that utilizes the stencil buffer – a surface which can be rendered to, but is used for intermediate calculation and is not seen. Figure 2

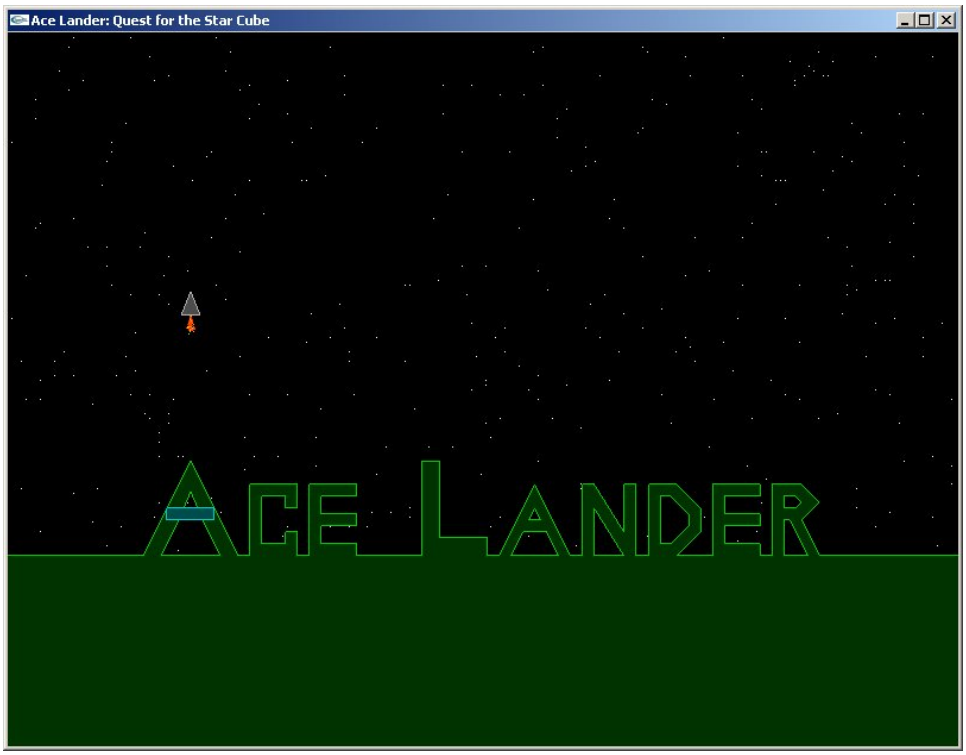

Figure 2: Rendering Concave Polygons with Holes using Stencil Buffer

shows the result of implementing this algorithm. Drawing a polygon contained inside another produces a hole. Stars in the background of the title screen are visible through the holes in certain letters. The filled, concave polygon rendering algorithm requires a list of points ordered along the edge of the polygon. The algorithm considers all the points of a polygon in sets of three points, one of which remains fixed while the other two cycle around the edge of the polygon to be drawn. For every set of three points, a triangle is drawn to the stencil buffer.

When drawing the final image, only pixels that correspond to those that were marked an odd number of times on the stencil buffer are drawn to the color buffer. Figure 3 helps illustrate how the algorithm works. Imagine starting with a

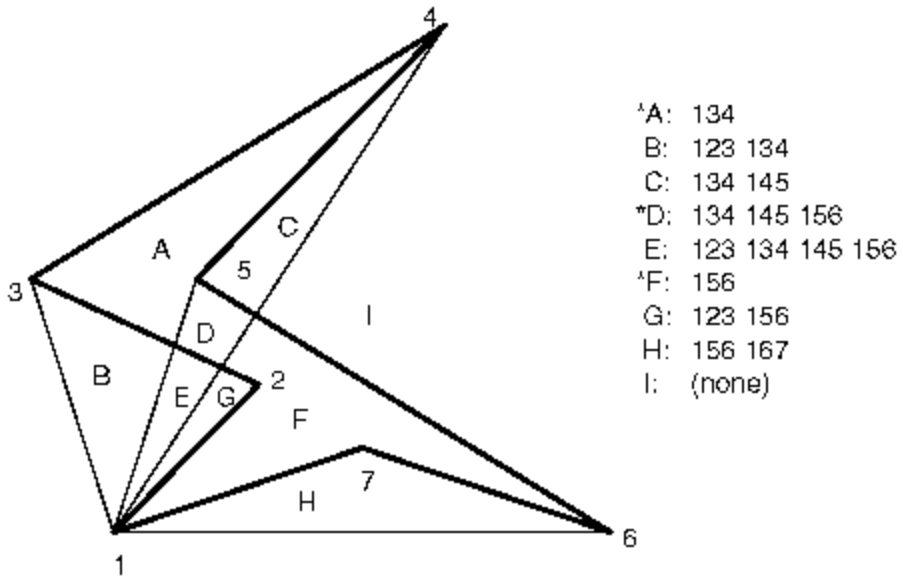

Figure 3: Illustration of Concave Polygon Filling Algorithm

stencil buffer where each value is cleared to zero. Notice that by flipping bits in the stencil buffer corresponding to the area covered by triangles drawn for each set of three vertices, the polygons labeled A, D, and F are covered an odd number of times and all other areas are covered an even number of times. When the final polygon is drawn, only those three polygons will be filled.

In real software development, the goal is to sell a product complete with packaging and a user friendly installation interface. These types of considerations regarding the end user's experience are not taken into account in typical school assignments. As part of the publishing process, I created a

distributable CD that automatically runs when inserted into a CD drive. An autorun initialization file automatically loads a webpage in the default browser. The webpage uses cascading style sheets to display a background layout that adjusts to fit the window when it is resized. The webpage contains a link that will run the installer for Ace Lander. Since I intend to distribute the CD to potential employers as an example of my work, there is also a link to view the included source code.

Working on Ace Lander has taught me that there is a difference between understanding the ideas behind algorithms and implementing them both efficiently and in a short time span. I gained practical experience in taking abstract concepts and integrating them into a real, finished product. Completing Ace Lander has made me better prepared for a position in the software development industry.

Sources

Angel, Edward. *OpenGL: A Primer*, 2nd edition. Addison Wesley, Boston, 2004.

Cormen, Thomas H., et al. *Introduction to Algorithms*, 2nd edition. The MIT Press, Boston, 2001.

"Drawing Filled, Concave Polygons Using the Stencil Buffer." OpenGL Programming Guide; 2006 January. http://glprogramming.com/red/chapter14.html#name13.

Foster, Tony. "Managing Game States in C++." Tony and Paige; 2005 August. http://tonyandpaige.com/tutorials/game1.html.