Line Detection for Moth Flight Simulation

Brennan Rusnell
Mark Eramian

August 18, 2008

Contents

List of Tables

List of Figures

Abstract

During a Hawkmoth's (*Manduca sexta*) flight path, much attention is directed towards the orientation and location of vertical edges. Previous studies have collected moth flight data where a moth controls its own virtual three-dimensional visual environment through an interface with the video game Descent 3. Descent 3 is also responsible for the rendering of the three-dimensional virutal environment. Movies of the three-dimensional scenes visible to the moths during its simulated flight are recorded. This research aims to extract the vertical edges from individual video frames by reconstructing the three-dimensional geometry associated with each frame.

Scenes were reconstructed using Descent 3 to render the three-dimensional geometry stored within recorded "demo files". The geometric information sent to the graphics processing unit from the Open Graphics Library was recorded and analyzed to extract the desired vertical edges.

Overall, the proposed vertical edge detection methodology had a specificity of 0.983913 and a sensitivity of 1.000000. Most errors resulting in edges incorrectly marked as vertical can be attributed to a single type of failure case.

Chapter 1

Introduction

This research analyzed data that represents the basic principles of moth flight. Figure 1.1 is a frame from a two-dimensional (2D) movie of the flight path of a moth in a "virtual world". This figure serves as an example of the analyzed data. In the example scene, rectangular objects emanate from a flat plane. Furthermore, the background is black while the ground plane and rectangular objects share a black and white checkerboard texture. The objective of this research was to find the following data in image-space (screen coordinates):

1. α: the horizon orientation;
2. β: the angle of each rectangular block's edge (edge refers to one of the four lines that collectively define a face of a rectangular block) that is visible and orthogonal to the horizon;
3. length: the length of these edges;
4. location: the location of the midpoint of these edges.

The information of interest can be seen in Figure 1.2.

During a Hawkmoth's (*Manduca sexta*) flight path, much attention is directed towards the orientation and location of vertical edges [1, 3, 9, 10, 12]. This data is collected to provide metrics of a visual stimulus that can be related to simultaneously recorded physiological data from the moth's nervous system. The information extracted from these images is beneficial because, combined with the physiological data and knowledge of behavior, it will provide an understanding of what the moth's nervous system responds to and how this relates to flight steering [1, 3, 9, 10, 12].

The initial methodology for collecting horizon and edge data was to continue the work of Madsen [6] and use edge detection algorithms from the domain of image processing on individual frames. However, edge detection failed due to the texture present in the scenes (or "virtual world"). As depicted in Figure 1.1, the boundary between white and black coloured texture regions creates an edge. Thus, edge detection finds edges within the faces of the rectangular blocks. Furthermore, because the background is black in colour, it becomes difficult to find corners that contain a black texture region. Edge detection algorithms would need to be complemented with line reconstruction algorithms in order to fill in the missing edge information. Due to the large number of edges that existed after edge detection, most of which were from texture boundaries (not geometry), line reconstruction became a tedious, inaccurate process. Thus, other options were explored.

The option considered next was edge detection on the Open Graphics Library (OpenGL) depth buffer. GLIntercept (described in Section 2) is a piece of software which allows one to save the depth buffer on a per frame basis. GLIntercept captures an image of the depth buffer by assigning a different intensity of gray to each depth value such that more distant objects have a greater intensity and closer objects have a lower intensity. However, Figure 1.3 shows that the resulting images were low in contrast. This made it difficult to perform edge detection. In an attempt to aid in edge detection, histogram equalization was performed on the images in Matlab, but this process exaggerated the quantization of depth values, as shown in Figure 1.4. As a result, edges were detected along the visible depth discontinuities in the histogram equalized depth buffer image. Furthermore, GLIntercept created multiple depth buffer images per frame, making it difficult to determine which image contained the final scene data. Therefore, this method was found to be ineffective.

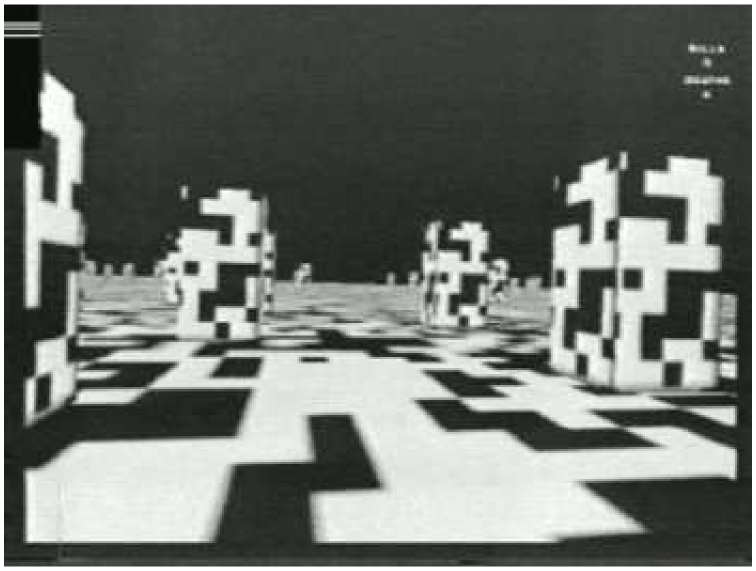

Figure 1.1: Example scene capture. This is a frame from a 2D movie of the flight path of a moth and serves as an example of the analyzed data.

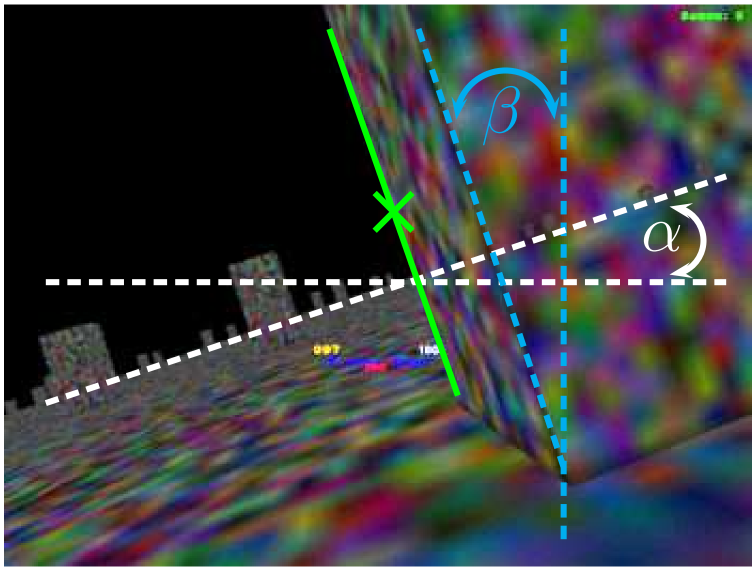

Figure 1.2: Sample scene illustrating the line parameters to be extracted. The solid line denotes an edge of interest which is visible and orthogonal to the horizon. α is the horizon orientation, β is the angle that the edge of interest makes with the vertical, and and the “X” is the midpoint of the highlighted edge.

Figure 1.3: Image of the depth buffer before image processing. The image contrast is low.

Another option was to determine if the three-dimensional (3D) geometry of the scene could be reconstructed on a per frame basis. This is traditionally a difficult problem but was a viable option due to the setup of the experiments which generated the scenes. In the experiments, which were conducted during a previous study [3], a live moth was fixed in place in a simulator with its wings and abdomen free to move. Descent 3 acted as both the simulator and scene generator. Sensors that monitored the position of the freely moving body parts provided feedback that allowed the moth to control its own visual environment.

Descent 3 is a 3D "first-person shooter" (FPS) video game noted for popularizing the use of true 3D rendering technology and providing the player with six full degrees of freedom to move and look around. Two important file types within Descent 3 were used in this research: a demo file and a level file. The demo file records the events that take place: in this case, the movement of the live moth. The level file is used to store the geometry of the game environment in a 3D coordinate system. Given that this information is attainable from the demo and level files, solving the problem by replaying the demos using Descent 3 in order to reconstruct the 3D scene geometry was considered.

The following approach to edge and horizon detection was used: Record the geometric information being sent to the graphics processing unit (GPU) from OpenGL. Because Descent 3 uses OpenGL for rendering, it is possible to extract the location of the geometry in image space for each scene. This was achieved through use of the programs GLIntercept and OGLE: The OpenGLExtractor (Described in Chapter 2).

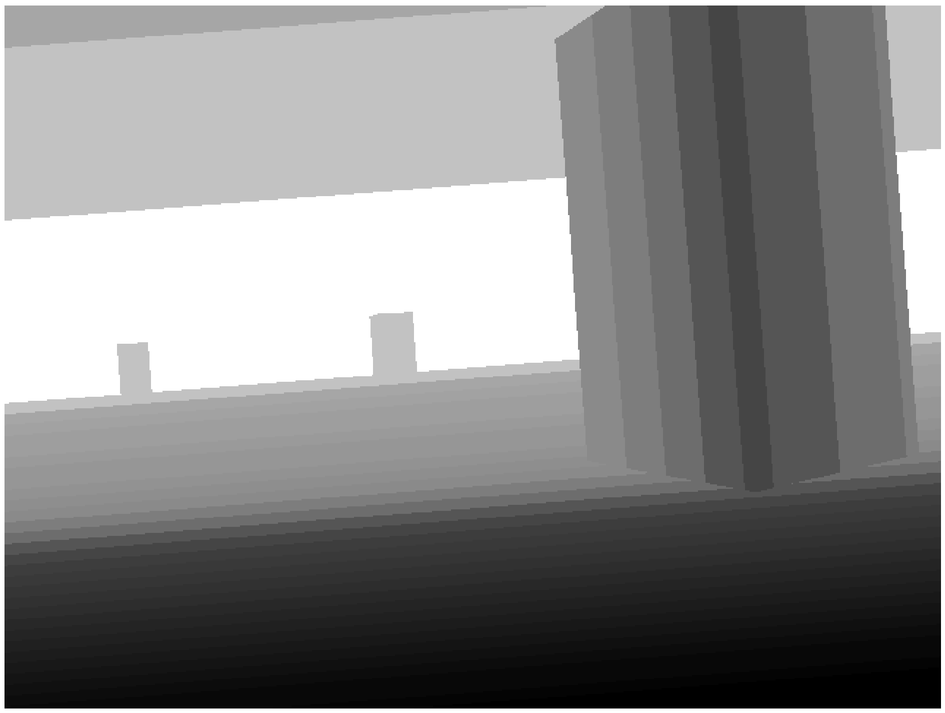

Figure 1.4: Image of the depth buffer after histogram equalization. Histogram equalization exaggerates the quantization of depth values.

Chapter 2

Theory

The successful completion of the horizon and edge detection was possible by extracting geometrical information from OpenGL. The programs GLIntercept and OGLE were used because of their integration with OpenGL.

GLIntercept is an OpenGL function call interceptor [8]. The following functionality of GLIntercept was used to aid in this research (GLIntercept has additional functionality as well):

- Saving all OpenGL function calls to text or Extensible Markup Language (XML) format with the option to log individual frames.
- Saving of the OpenGL frame buffer (color/depth/stencil) pre and post render calls. The ability to save the difference of pre and post images is also available.

OGLE is a software package created by Eyebeam R&D [2] that allows for the capture and re-use of 3D geometry data from 3D graphics applications running on Microsoft Windows. It works by observing the data flowing between 3D applications and the system's OpenGL library, and recording that data in a standard 3D file format. In other words, it is a "screen grab" or "view source" operation for 3D data.

GLIntercept and OGLE take advantage of the fact that one can write listeners (code segments that are executed when a given action occurs) for OpenGL calls and store the information OpenGL is sending to the GPU. This is possible since OpenGL is an open-source graphics application programming interface (API). OGLE is a plug-in to GLIntercept that extends the functionality of GLIntercept by writing out the geometric information being sent to the GPU. OGLE writes out an .obj file (Wavefront Object File). An example of the file format is given below. Without modification, OGLE is meant to capture a single frame when the user presses a predetermined combination of keys. The most common usage of OGLE is for 3D-printing. A minor modification was made to OGLE to enable it to write out an .obj file for each frame rendered during the playback of a demo. This creates the scene data of interest.

The file format of a .obj file is as follows for each polygon in the scene:

- `#POLYGON [N]`: This line indicates that there are `N` vertices for this polygon.
- `g M`: This line indicates that this polygon belongs to group `M`.
- The next `N` lines have:
  - `v x y z`: This line indicates that `(x,y,z)` is a geometric vertex.
- `f 1 2 ... (N-1) N`: The ordering in which the vertices are connected (not necessarily numerically ordered).

This format is repeated for every polygon in the scene, which makes parsing trivial. With proper parsing and processing of this file, it is possible to find the edges orthogonal to the horizon. This methodology is described in Section 3.

Chapter 3

Experimental Method

Once it was possible to obtain the scene geometry on a per frame basis, an .obj file parser was written to read in the vertex data, as described in Chapter 2. Since scene geometry contained in the .obj files was already in screen coordinates, the goal of extracting screen coordinates and orientation of vertical edges was simplified. The information was not stored in its initial 3D coordinate system because Descent 3 was written with portability in mind. The developers of Descent 3 were able to render level geometry with the Direct3D, 3Dfx Glide, and OpenGL graphics libraries. To achieve this compatibility, the game was written such that all geometrical transformations were performed on the central processing unit (CPU), making it independent of any graphics API. The only information being sent to the GPU was the final screen coordinates of each visible scene feature.

It was also noted that the moth was only permitted to control horizontal movement in its virtual environment. This intentional restriction was made because during odour-guided flight male moths tend to maintain a fixed altitude [5, 11]. Also, the feedback system needed for the moth to control the scene in this way would have been too complex. If the degree of freedom along the y-axis is added into the experiment, the current methodology for extracting scene edge information will likely fail.

All code was written in C++ using Microsoft Visual Studio 6.0. The proposed implementation relies on the FreeImage library, the OpenGL Utility Toolkit (GLUT), and OpenGL. There is minimal reliance on the Windows API and the aforementioned libraries/toolkits are available for Windows, Linux, and Mac OS X. Thus, with minimal modification, this program can run on any platform. However, since OGLE and GLIntercept are Windows only, this program has been developed for and tested on Windows. More specifically, Windows XP Professional was used as the primary operating system (OS).

The .obj file parser was used to input the 3D geometry of each frame into the proposed system. The following steps were taken:

1. removed user interface (UI) elements;
2. removed edges that terminated in a corner of the screen;
3. removed edges shorter than 30 pixels in length;
4. calculated the angle of the horizon;
5. removed edges that were occluded by objects closer to the screen;
6. created a histogram of the edge angles;
7. obtained the peaks (2) from the histogram;
8. obtained the vertical edges (orthogonal to the horizon);
9. wrote out vertical edge information to file (.csv[1] (comma-separated values) and .jpg).

Each of these steps is discussed in detail in the remainder of the chapter.

[1].csv files are text files that separate data fields by a comma. Spreadsheet applications can read this format and import the data for further analysis.

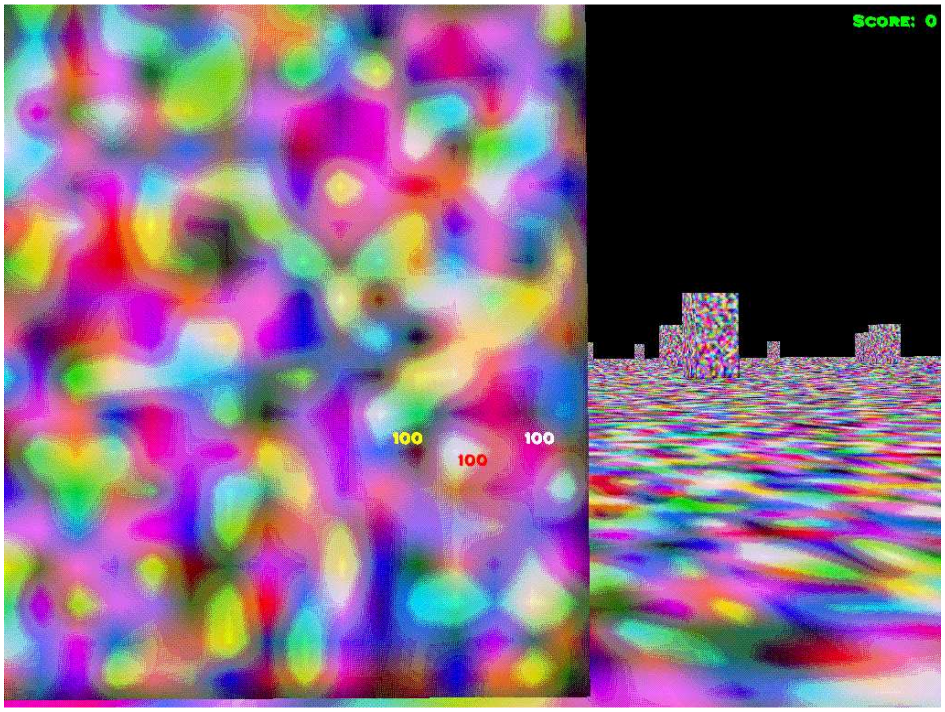

Figure 3.1: Example of demo user interface (autocontrasted for viewing purposes).

3.1 UI Removal

In the analyzed demos, it was noted that three numbers consistently appeared in the center of the screen and that in the top right corner of the screen, the current score was consistently displayed. This can be seen in Figure 3.1 [7]. It was determined that these 3D elements had a z-coordinate of exactly 0.000 (this was determined by manually inspecting the .obj files). Thus, any geometrical face containing a vertex with a z-coordinate of 0.000 was disregarded.

3.2 Edge Removal

There were two initial criteria for edge removal. The first was if an edge terminated in a corner of the screen. It was found, from empirical observation, that some of the scenes were outlined by edges, as if the scene contained a border (see Figure 3.2). The border occurred irregularly, making it difficult to determine if it was actual geometry or "garbage" data. A simple detection for edges terminating in a corner of the screen removed these edges. It was necessary to remove these edges in order to accurately perform the occlusion tests, as discussed in Section 3.4.

The second criterion was to remove edges less than 30 pixels in length. Based on prior knowledge of the spatial resolution of the moth's eye, any line less than 30 pixels in length can be ignored because it is believed that the moth cannot see it. This is because at the light levels used, the spatial resolution of the moth's eye was about ten degrees of the field of view [4]. An Euclidean distance calculation for each edge (using the edge's endpoints) was used to remove edges shorter than 30 pixels.

3.3 Horizon Calculation

Horizon detection was a crucial step in the proposed methodology. Without knowledge of the horizon orientation, it would be difficult to determine which edges are orthogonal to the horizon. Several methods of horizon detection were attempted. Initially, it was noticed that the longest edge in the scene was the

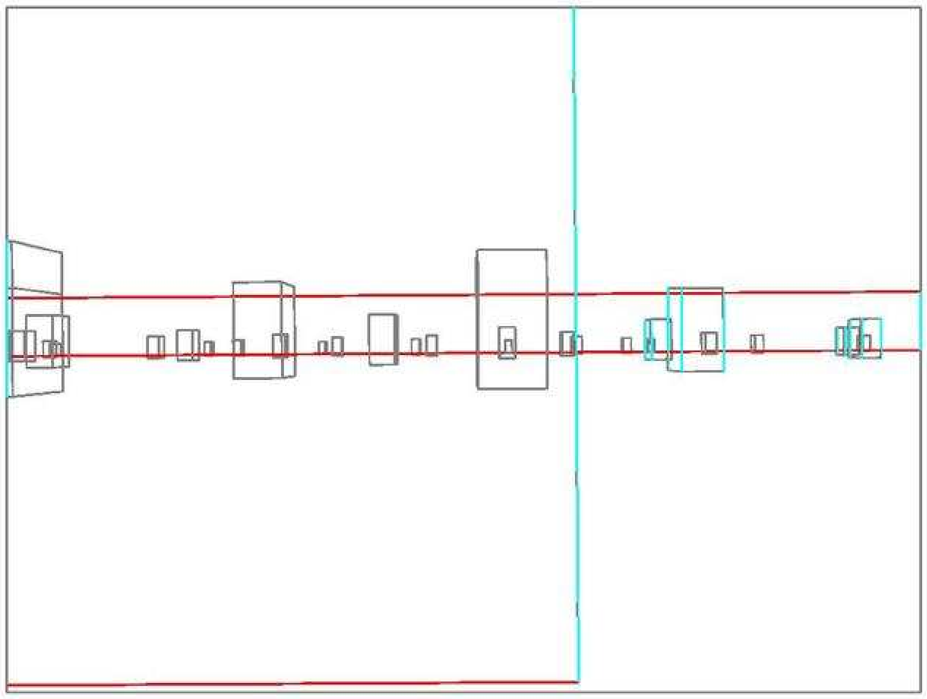

Figure 3.2: Example scene demonstrating a border surrounding the image.

horizon line itself. However, this was not always the case, so this method was abandoned. This lead to a key assumption: Each face in the .obj file is rendered from the top-right vertex in a counter-clockwise fashion. This was evident from inspecting the vertices and the ordering in the .obj file itself, as discussed in Section 2. Under this assumption, the orientation of the first and third edge of each face are approximately parallel to that of the horizon. However, their orientation is not necessarily equal to the horizon's orientation due to perspective distortion. Thus, the average orientation of the first and third edges over all edges remaining after edge removal was taken as an approximation to the horizon orientation. This approximation produced robust results (it found the horizon regardless of the horizon's orientation) even when the longest edge was not present. This was not the most accurate method, but due to the fact that the scene consisted of either edges parallel or orthogonal to the horizon, a large tolerance about the orthogonal orientation (within tolerances of the application) was permissible for accurately finding the vertical edges of interest. An analysis of the accuracy of the detected horizon is presented in Section 5.1.

3.4 Occlusion Detection

The next step was to remove occluded edges. The geometry stored in the .obj files does not include occlusion information. As a result, all geometry is included in the .obj file even if it is occluded. In OpenGL, occluded geometry is removed at a later stage in the rendering pipeline.

Two additional assumptions were needed for the occlusion detection step. It was clear from empirical observation that the rectangular blocks emanating from the ground had the same dimensions (height, width, and depth) in world-space (**not** image-space). Thus, because of perspective projection, all blocks further from the camera than a given block b will never have screen coordinates with a greater y-coordinate than that of b. The second assumption was that the y-coordinate of the camera is low enough for this to be true and that it is constant. Under these assumptions, a vertical edge will never be partially occluded. These assumptions, which are empirically true, eliminated the need to test for partially occluded vertical edges.

A simple occlusion test based on the aforementioned assumptions was used. If both vertices of an edge were contained within a face of a rectangular block, then that edge was completely occluded and could be discarded. If not, it was either visible or partially visible. If it were visible, it could potentially be a vertical

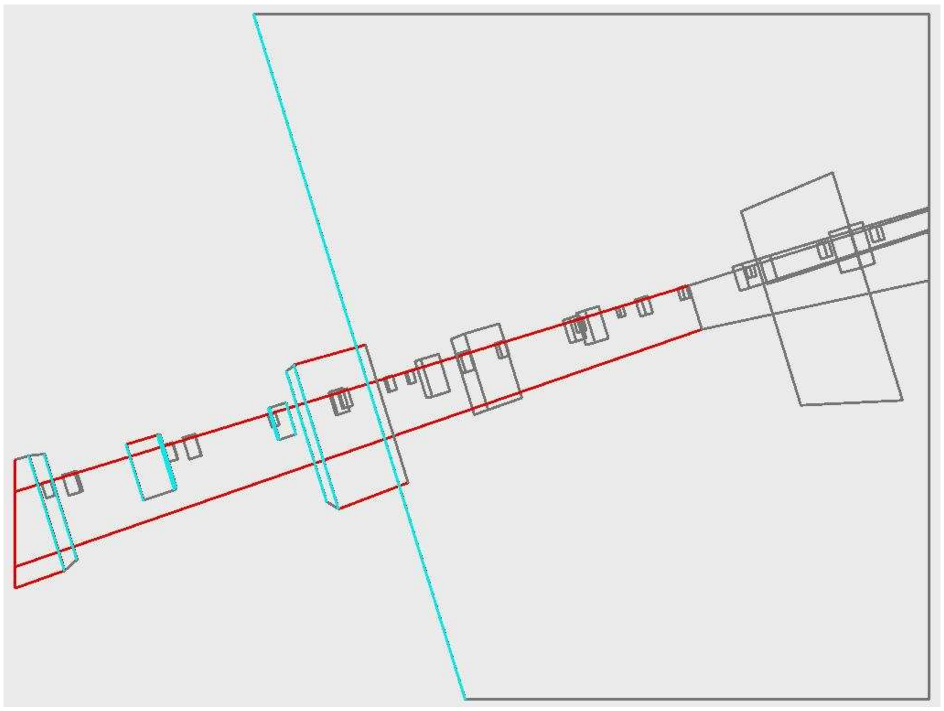

Figure 3.3: Output image.

edge in the final set of vertical edges. If it was partially occluded (one vertex is contained within a face and the other is not), then it was not vertical. Thus, it must have been horizontal (relative to the horizon) and, since horizontal edges were not of interest, it was discarded.

3.5 Vertical Edge Determination

After completion of steps 1-5, a set of visible edges remains. For each visible edge, if it was orthogonal in image-space (within a tolerance of five degrees) to the calculated horizon orientation, it was stored as a vertical edge and its statistics were output to a .csv file. It was also ensured that the same edge was not reported multiple times, which arises when two faces share an edge.

The statistical data is also depicted visually in a .jpg image. There is one image per frame which is created in the proposed software (see Section 4.3). Figure 3.3 shows an example image. The cyan edges are the non-occluded edges orthogonal to the horizon that are longer than 30 pixels, the red edges are either visible or partially visible but not orthogonal to the horizon (these edges are of no further importance after the horizon orientation calculation, so they are ignored), and the gray edges are not visible (occluded or shorter than 30 pixels).

Chapter 4

Setup

There are three main steps in the proposed methodology. The first step is to generate the .obj files for parsing. The second step is to parse the .obj files and create the analysis files (which contain the orthogonal edge information) and image files. The last step is to create a video from the resulting images to verify the accuracy of the proposed methodology.

In order to carry out these steps, one needs to install the following programs:

4.1 Installation

1. Install Descent 3 from CD
2. Update Descent 3 to v1.4 (http://www.descent3.com/patch/D3_US_1.4_Patch.exe)
3. Install the latest ATI/nVidia drivers for latest version of OpenGL
   (a) Our system used a nVidia GeForce 7600 GS with driver version 9.1.3.1
4. Install GLIntercept 0.5 (http://glintercept.nutty.org/Ver0_5/GLIntercept0_5.exe)
5. Install OGLE version 0.3b (http://ogle.eyebeamresearch.org/dist/ogle-0.3b.bin.zip)
6. Install GLUT (http://www.xmission.com/∼nate/glut/glut-3.7.6-bin.zip)
7. Install Cygwin with default packages (http://www.cygwin.com/setup.exe)
8. Install MPlayer/Mencoder (http://lyseo.edu.ouka.fi/∼pexu/h/mplayer_windows_howto/)
   (a) The installation information can also be found in the included file install_mplayer.txt. This file contains a few modifications suited for the purposes of this research.

Once the programs from Section 4.1 are installed, the system needs to be configured:

4.2 Configuration

1. Copy C:\WINDOWS\system32\opengl32.dll to C:\Games\Descent3\opengl32.orig.dll
2. Copy gliConfig.ini to C:\Games\Descent3
3. Copy OGLE.dll to C:\Program Files\GLIntercept0_5\Plugins\OGLE
4. Launch Descent 3. In the "Descent 3 Launcher" window, choose Setup and select the Video tab. Press "Redetect Available Rendering Devices" and select OpenGL from the drop-down menu.

5. While still in Setup, select the Misc. tab and for “Optional D3 Command-line Switches” enter: -nointro -timetest experiments\demofile.dem -framecap 999, where “demofile.dem” is the name of the demo file (assuming the experimental demo files are located in the experiments directory, which is a subdirectory in the demo directory).

6. Set resolution to 640x480 in Descent 3 (once application is open)

Upon successful configuration (Section 4.2) and playback of a demo file, one needs to perform the actions in Section 4.3 in order to analyze the .obj files and verify the correctness of the proposed methodology.

4.3 Execution

1. Each demo creates a folder in the format YEAR-MONTH-DAY_HR-MIN-SEC

2. Copy Lines.exe, convert.sh, and FreeImage.dll to this folder

3. In Cygwin, change the current directory to this folder and type: `./convert.sh`

4. Upon successful completion, there will be three new directories (results, images, and objects), an .avi file titled ogle.avi, and a results file titled analysis.csv.

   (a) The .avi file should be run with MPlayer. The following command was used to playback the .avi files (assuming one is in the directory that contains the .avi): `mplayer -fps N output.avi`, where `N` is the desired frames per second.

   (b) The analysis file analysis.csv contains data formatted as follows: frame_num (frame number), edge_num (edge number), orientation (in degrees), location (in screen-coordinates), α (in degrees), β (in degrees). This repeats for every edge of every frame. It is important to note that edge N in frame M is not necessarily edge N in other frames. The proposed program does not track edges across multiple frames. It analyzes data on a per frame basis.

Chapter 5

Analysis

To quantitatively analyze the results, 32 of the resulting .jpg images from a demo run were manually inspected. These images were selected at random over the entire length of one demo. The demo started on frame 1 and ended on frame 7752.

5.1 Horizon Orientation

To analyze the accuracy of the proposed horizon orientation methodology from Section 3.3, 32 images from Section 5.2 were manually inspected. The horizon angle in each image was manully computed by locating the endpoints of what was believed to be the horizon line and by taking the arctan of this data. The generated data can be found in Figure 5.1 (angles in degrees).

Table 5.1 shows that the horizon calculation from Section 3.3 is a good approximation to the actual horizon orientation. The root mean square error (RMSE) was computed using Equation 5.1 ($p_1 ... p_n$ are the orientations calculated by the proposed methodology, $m_1 ... m_n$ are the manually calculated orientations, and n is the number of test frames). The RMSE measures the accuracy of the previously mentioned approximation to the horizon orientation. Intuitively, it is desirable to to minimize the error. The RMSE was found to be 0.190520, meaning that on average the error was around 0.190520 degrees.

$$RMSE = \sqrt{\frac{(p_1 - m_1)^2 + (p_2 - m_2)^2 + ... + (p_n - m_n)^2}{n}} \tag{5.1}$$

5.2 Orthogonal Edges

The information found in Table 5.2 was of interest: the number of true positives (TP) (the number of edge correctly identified as being of interest), the number of true negatives (TN) (the number of edges correctly identified as being not of interest), the number of false positives (FP) (the number of edges identified as being of interest that should not have been), the number of false negatives (FN) (the number of edges that were not identified as being of interest that should have been), sensitivity (the fraction of edges that are of interest which were correctly detected, also known as the true positive fraction (TPF)), the false positive fraction (FPF) (the fraction of edges that are not of interest which are incorrectly classified as not of interest), and the specificity ((1 - FPF)).

Sensitivity and specificity are of utmost importance. The closer to 1 these values are, the more accurate and correct the methodology is. The values for TP, FP, and FN were calculated by manually counting the edges in the image. The value for TN was provided by the program. The remaining values were calculated by the formulas given in Table 5.2. The results are shown in Table 5.3.

As Table 5.4 shows, the overall results from the proposed methodlogy are promising. Over the 32 analyzed images, the findings show that on average, sensitivity is 1.000 and specificity is 0.984.

However, there is one case that the proposed methodology fails to capture. In some scenes, the moth flies alongside the face of a rectangular object and the face occupies the entire screen. In this case, an in-game screenshot of the scene reveals no geometry and the geometrical data contains the occluded geometry and

Frame	Program Orientation	Manual Orientation
1	-0.273055	-0.369050
256	-0.273055	-0.369645
509	-3.112016	-3.372115
805	-26.106232	-26.233011
1004	-0.205696	-0.318307
1267	-0.348081	-0.313660
1513	21.830700	21.801409
1763	-0.052990	0.000000
2013	0.040491	0.000000
2268	-2.108806	-2.216794
2516	0.203742	-0.092562
2795	-2.554685	-2.615755
3004	-0.405929	-0.830315
3264	-0.030439	0.000000
3511	10.110803	9.730127
3782	-30.005651	-29.536463
4007	-8.633488	-8.759983
4284	0.063965	-0.092115
4495	0.048302	-0.095493
4785	-4.028216	-4.151285
5015	-0.344200	-0.558965
5274	-7.564497	-7.678964
5507	1.302736	1.295645
5774	-0.045837	-0.184824
6008	0.081636	-0.092264
6271	12.297679	12.440856
6509	32.821873	32.780685
6804	-23.935978	-23.971022
7010	-0.039531	-0.184527
7262	-1.414261	-1.704165
7512	-1.414261	-1.612204
7752	-1.414261	-1.609540

Table 5.1: To quantitatively analyze the horizon orientation detection algorithm, 32 of the resulting .jpg images from a demo run were inspected. These images were selected at random over the entire length of one demo. The demo started on frame 1 and ended on frame 7752. The data shows how the approximation to the horizon orientation (Program Orientation) compares to the actual horizon (Manual Orientation), which was manually computed (The horizon angle in each image was manually computed by tracking the endpoints of what was believed to be the horizon line and by taking the arctan of this data). All angles are in degrees.

Term	Definition
True Positive (TP)	The number of edges correctly identified as being of interest.
True Negative (TN)	The number of edges correctly identified as being not of interest.
False Positive (FP)	The number of edges identified as being of interest that should not have been.
False Negative (FN)	The number of edges that were not identified as being of interest that should have been.
Sensitivity	$[TP/(TP+FN)]$ The fraction of edges that are of interest which were correctly detected. Also known as the **True Positive Fraction (TPF)**.
False Positive Fraction (FPF)	$[FP/(TN+FP)]$ The fraction of edges that are **not** of interest which are incorrectly classified as being of interest.
Specificity	$[1-FPF]$.

Table 5.2: Terms/calculations used in analysis. The numbers generated by these calculations facilitate quantitative analysis of the success of the proposed methodology. Sensitivity and specificity are of utmost importance. The closer to to 1 these values are, the more accurate and correct the methodology is.

draws a border around the scene. Since all detected borders are eliminated, as discussed in Section 3.2, the proposed methodology erroneously analyzes the geometry concealed behind the face of the rectangular object. In this case, the specificity is slightly lower on average because the edges that are included are found erroneously. However, this situation does not frequently occur. As Table 5.3 shows, none of the 32 test frames contained this known failure case.

Frame	TP	TN	FP	FN	Sensitivity	Specificity
1	11	253	4	0	1.000	0.984
256	11	253	4	0	1.000	0.984
509	18	143	3	0	1.000	0.979
805	25	163	0	0	1.000	1.000
1004	15	160	4	0	1.000	0.976
1267	24	185	3	0	1.000	0.984
1513	20	189	3	0	1.000	0.984
1763	24	181	3	0	1.000	0.984
2013	19	247	6	0	1.000	0.976
2268	21	156	3	0	1.000	0.981
2516	22	189	5	0	1.000	0.974
2795	15	169	4	0	1.000	0.977
3004	17	179	4	0	1.000	0.978
3264	22	190	4	0	1.000	0.979
3511	18	217	1	0	1.000	0.995
3782	35	317	0	0	1.000	1.000
4007	26	243	3	0	1.000	0.988
4284	21	184	3	0	1.000	0.984
4495	21	169	2	0	1.000	0.988
4785	28	263	5	0	1.000	0.981
5015	15	194	3	0	1.000	0.985
5274	22	193	1	0	1.000	0.995
5507	19	242	7	0	1.000	0.972
5774	20	134	2	0	1.000	0.985
6008	19	157	4	0	1.000	0.975
6271	24	220	0	0	1.000	1.000
6509	24	267	1	0	1.000	0.996
6804	28	245	3	0	1.000	0.988
7010	22	182	4	0	1.000	0.978
7262	24	159	5	0	1.000	0.970
7512	24	159	5	0	1.000	0.970
7752	24	159	5	0	1.000	0.970

Table 5.3: To quantitatively analyze the quality of the proposed vertical line detection algorithm, 32 of the resulting .jpg images from a demo run were manually inspected. These images were selected at random over the entire length of one demo. The demo started on frame 1 and ended on frame 7752. The data in this table was generated by using the calculations from Table 5.2 and from manually inspecting the images. The data shows the correctness and completeness of the proposed methodology for orthogonal edge detection.

Sensitivity	Specificity
1.000000	0.983913

Table 5.4: Overall data from Table 5.3. The values from Table 5.3 were summed and used in the equations from Table 5.2 in order to create an overall value for each field.

Chapter 6

Conclusion

Though the previous work in this area [6] was promising, a better solution was still needed in order to generate useful data. The proposed methodology is robust and accurate enough to generate such data. The work done by Madsen [6] was incomplete, did not provide results for all stages, and could still be enhanced by improvements to accuracy and efficiency. The proposed solution is complete, provides analysis for every frame in a demo, and though it has some limitations (Section 5), the level of accuracy and efficiency achieved reduces the work required to extract the data needed for further analysis of moth behavior.

In order to improve upon this methodology, the number of False Positives could be reduced. When a specificity less than 1 is returned, it is because it over-detected the number of orthogonal edges. Moreover, some of the steps from Chapter 3 could be combined to optimize the proposed methodology. For example, Section 3.3 and Section 3.5 could be combined. The second and fourth edges of each polygon could be selected and then occlusion detection could be performed. Doing this in one pass would improve upon the current implementation. However, in order to aid in project maintenance, the code was kept readable and the steps independent. Lastly, even though the information being sent to the GPU is in screen coordinates, the vertices do contain some depth information. The depth information is in homogeneous coordinates, which means depth values are scaled to the range [-1.0, 1.0]. From empirical observation, the z-coordinates of the vertices were found to cluster around -0.9. However, there is enough depth information to make use of the depth buffer, as shown in Figure 6.1. To aid in solving the problem when the moth flies alongside a face, the depth data could be used.

The proposed methodology is consistent and robust. Analysis shows promising results with minimal erroneous conditions. The quality of the horizon detection algorithm is of special importance. A RMSE of 0.190520 means that on average the error is around 0.190520 degrees. Given that the horizon orientation serves as the basis for orthogonal edge detection, having such a low RMSE greatly increases the accuracy of the proposed orthogonal edge detection methodology.

As discussed above, work can be done to make the methodology more correct and robust, but in its current state, a strong underlying methodology has been developed that can be the basis for any future work relating to this project.

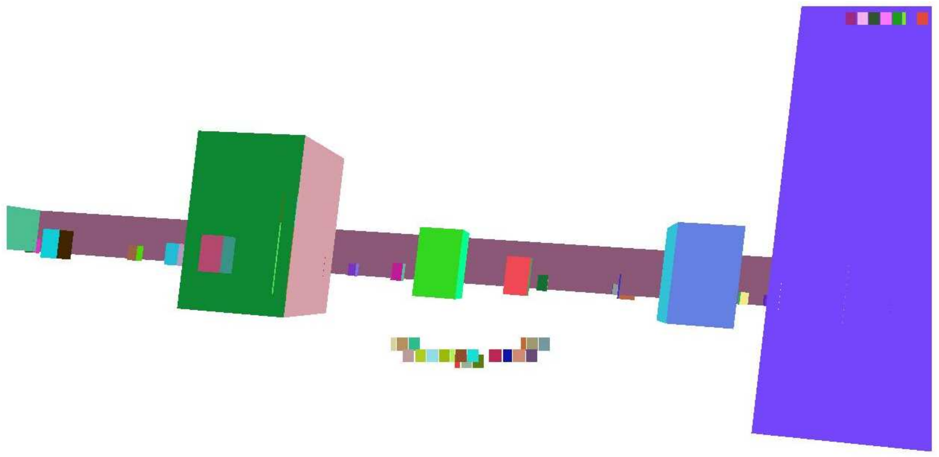

Figure 6.1: Rendering the .obj data using the depth buffer (the smaller square polygons are from the UI elements). This rendering demonstrates how the limited depth data in the .obj files can be used to perform occlusion testing. However, as this scene also highlights, some of the data information is incorrect (notice how a some of the smaller blocks are completely encompassed by a larger block and appear to be closer to the camera).

Chapter 7

Acknowledgments

Special thanks to Jack Gray for providing the data sets and background information relating to moth physiology. Thanks to Michael Frumin, R&D Tehnical Director at Eyebeam, for making OGLE compatible with Descent 3, Tony Arkles for lending his technical expertise/advise to the project, and most importantly, a very special thanks to Mark Eramian, who made this research possible. This research was funded in part by the Natural Sciences and Engineering Research Council (NSERC).

Bibliography

[1] J. H. Belanger and E. A. Arbas. Behavioral strategies underlying pheromone-modulated flight in moths: lessons from simulation studies. *J.Comp.Physiol.[A]*, 183:345–360, 1998.

[2] Michael Frumin. OGLE (OpenGLExtractor). http://ogle.eyebeamresearch.org/, 2006.

[3] J. R. Gray, V. Pawlowski, and M. A. Willis. A method for recording behavior and multineuronal CNS activity from tethered insects flying in virtual space. *J.Neurosci.Methods*, 120:211–223, 2002.

[4] Jack Gray. Personal Communication, July 2006.

[5] R. Kanzaki, E. A. Arbas, and J. G. Hildebrand. Physiology and morphology of descending neurons in pheromone-processing olfactory pathways in the male moth Manduca sexta. *J.Comp.Physiol.[A]*, 169:1–14, 1991.

[6] Dana Madsen. CMPT 859 Final Report. April 2005.

[7] Divakar Roy. Auto contrast. http://www.mathworks.com, March 2006.

[8] Damian Trebilco. GLIntercept. http://glintercept.nutty.org/, 2006.

[9] M. Wicklein and T. J. Sejnowski. Perception of change in depth in the hummingbird hawkmoth Manduca sexta (Sphingidae, Lepidoptera). *Neurocomputing*, 38-40:1595–1602, 2001.

[10] M. Wicklein and N. J. Strausfeld. Organization and significance of neurons that detect change of visual depth in the hawk moth Manduca sexta. *J.Comp.Neurol.*, 424:356–376, 2000.

[11] M. A. Willis and E. A. Arbas. Odor-modulated upwind flight of the sphinx moth, Manduca sexta. *J.Comp.Physiol.[A]*, 169:427–440, 1991.

[12] M. A. Willis and E. A. Arbas. *Neurons, networks and motor behavior.* Cambridge, MA: MIT Press., 1997.