Camera-Projector Based Action-Shooting Arcade with Still and Moving Targets

By: Mwaffaq HajAli

A graduation project submitted in partial fulfillment of the requirements for the degree of Bachelor's of Science in the department of Computer Engineering, An-Najah National University.

December 4, 2007

Supervised by: Dr. Raed Alqadi

Index

1. Abstract ……………………………………………… 3

   1.1 The OCV Thread Functionality …………………..3

   1.2 OCV and The Camera's Frame Rate ……………..6

   1.3 The OGL Thread Functionality ………………….6

2. The Games ……………………………………...7

   2.1 The Bull's Eye Game …………………………….7

   2.2 The Birt Hunt Game ……………………………..7

3. OGL's Button Mode …………………………………3

4. LASER Gun Construction …………………………..10

5. The Implementation …………………………………12

   5.1 Why OpenCV ……………………….……...12

   5.2 Why OpenGL …………………………………....12

6. References ……………………………………...13

7. Appendix ……………………………………………15

1. Abstract

Typical games use peripherals such as the mouse and keyboard to control the game play course. Even though the high quality of the game's graphics give some feel of reality, moving the mouse and clicking a button to hit a target is far different from holding a playing gun, aiming at the target and pulling the trigger!

The purpose of this project is to introduce an arcade game model with increased degree of interactivity using a camera-projector based system.

This project is mainly oriented towards games requiring minimum amount of controls. Examples of such games would be some kinds of gun-games and board games. However, games requiring more complex controls can be supported by adding the suitable peripherals to support such games.

This project implements two examples of the games the model can support:

1. The Bull's-Eye Game (a still target)

2. Bird-Hunt (moving targets)

The application consists mainly of two threads:

1. The OCV Thread:

Responsible for capturing the video frames from the camera and processing them using OpenCV to detect and locate a LASER blob, then reporting it to other modules to check if the shot has hit any object and take action accordingly.

2. The OGL Thread:

Responsible for graphics generation using OpenGL.

1.1 The OCV Thread functionality:

Figure1 shows the OCV thread's flow chart. The functional blocks are explained below.

#2. Splitting the red channel: the captured image consists of three 8-bit channels (RGB). Since the gun uses a 630-680nm Class II-A **Red** LASER, the channel of interest would be the R channel. It's split from the captured RGB channel into a new 8-bit grayscale (monochrome) channel.

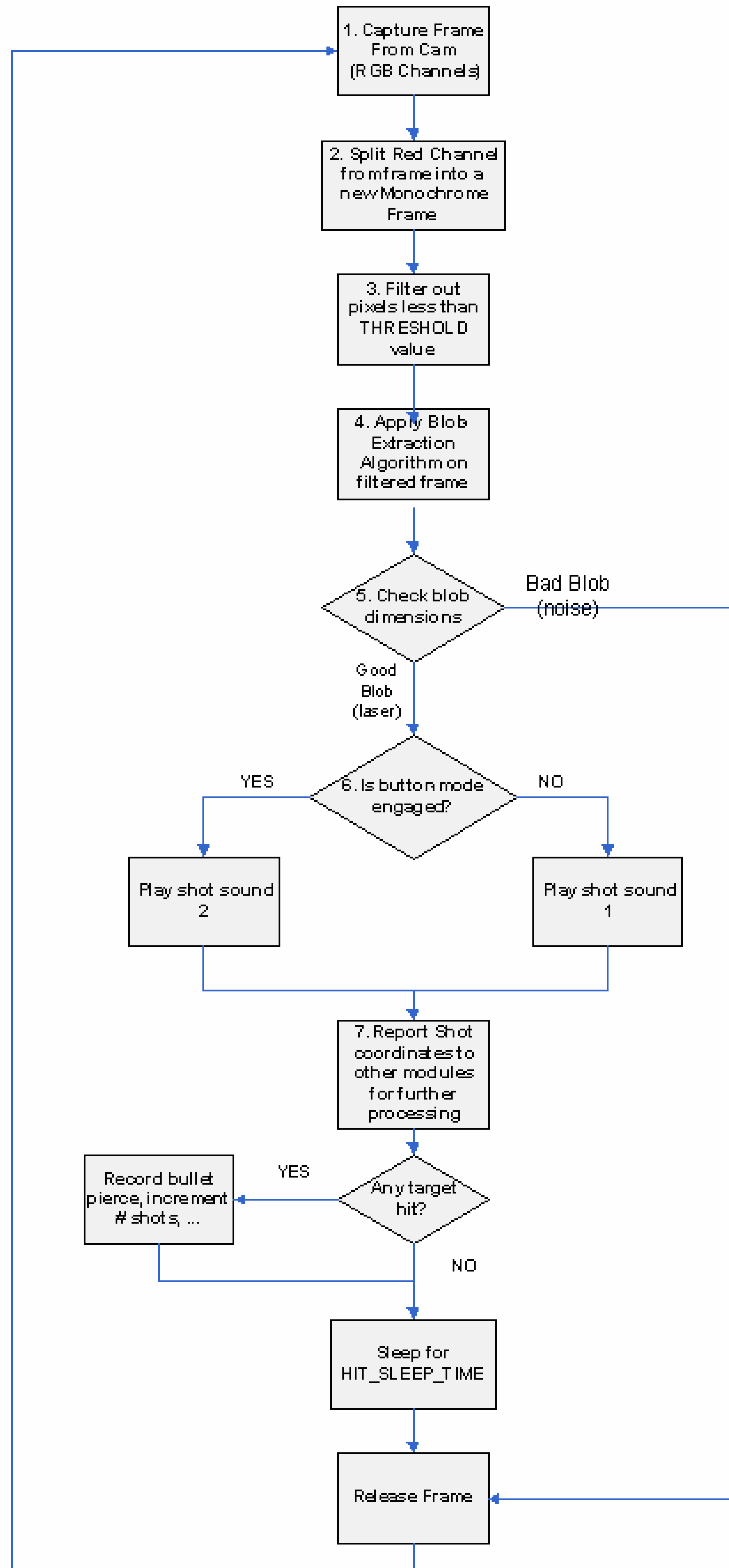
Figure1: The OCV Thread Flow Chart
1. Capture Frame From Cam (RGB Channels)
2. Split Red Channel from frame into a new Monochrome Frame
3. Filter out pixels less than THRESHOLD value
4. Apply Blob Extraction Algorithm on filtered frame
5. Check blob dimensions
Bad Blob (noise)
Good Blob (laser)
6. Is button mode engaged?
YES
NO
Play shot sound 2
Play shot sound 1
7. Report Shot coordinates to other modules for further processing
Any target hit?
YES
Record bullet pierce, increment # shots, ...
NO
Sleep for HIT_SLEEP_TIME
Release Frame

#3. Threshold Filtering: The LASER blob is the brightest light spot in the red channel. Pixel values less than a predefined THRESHOLD value are filtered out. The threshold value is adjustable, and is set to 254 at program initialization. Any pixel values less than or equal to this value will be nullified, while only the value 255 passes the filter. The LASER spot will be the brightest in the red channel, therefore it will have pixel values of 255 and pass the filter.

The disadvantage of this method is that if there are any other bright light sources or illuminant objects that reflect light perfectly in the range of the camera view, they cause noise since bright light has color components of all three channels.

Despite the disadvantage, this is the method with least computational overhead I could use to filter-out pixels not belonging to a LASER blob from a frame.

#4. Blob-Extraction Algorithm: This algorithm is implemented in the Blob Extraction Library (cvBlobsLib), which is part of Intel's OpenCV. It's used here to find the blob caused by the LASER gun in the image. For more on cvBlobsLib, refer to the Appendix.

#5. Check blob dimensions: If the blobs dimensions are above or below those that would usually be produces by a LASER spot, ignore this blob. It could be caused by a LASER shot wile the gun was being moved swiftly so that it appeared to the camera as a curve instead of a spot, or it could be caused by another source of light. In any such abnormal case, the resulting connected shape should not be considered a gun-shot.

Perfectly spot-like blobs, however, can hardly be produced since the gun usually tends to move slightly as the player pulls the trigger. Therefore, deformed blob shapes are accepted within a predefined error range.

#6. Button Mode: This mode is engaged when the player accesses the game's menu. The menu system is explained in more detail in section "OGL's Button Mode".

#7. Shot Reporting: the shot is reported to the OGL thread through global variables for further processing and drawing. This is explained in more detail in the OGL section. After a shot is reported, OCV sleeps for an amount of time predefined by (HIT_SLEEP_TIME). This method was used as a temporary solution to prevent a continuous emission of LASER causing a series of shots. This problem was solved by hardware by implanting circuitry inside the LASER gun to allow only one LASER burst per shot. The construction and operation of the LASER gun is explained in detail in section "LASER Gun Construction and Operation".

1.2 OCV and the Camera's Frame Rate Limitation:

OCV runs in an infinite loop to detect the blob as soon as it occurs. Since the processor speed is much higher than the camera's frame rate, the same frame used to be tested several time until a new frame arrives from the camera in the earlier implementation. This used to degrade the overall performance, since OCV in that case captures the CPU for more time than is actually needed. This problem was solved by having OCV sleep for a period of time equal to about one frame time, i.e.
(1/Frame Rate). This way, OCV won't capture the CPU until a new frame arrives from the camera.

The Camera:

The camera used in this project is a 640X480 pixel 30 FPS CMOS-Sensor Webcam. Even though it gave only about 15 FPS during tests, the response time (time between pulling the trigger and observing the reaction) was hardly noticeable by human and thus acceptable.
Using a higher Frame Rate camera would even give faster response.

Problems with the Camera:

The major problem with using this type of webcams is that they give a curved image at the edges. A straight object near an edge would look curved. This drifts the locations of blobs seen near the edges.
Using a higher-quality digital camera solves this problem.

Another problem is that the used webcam, of course, does not support optical zooming, not even digital! This causes the problem described in the next section.

Camera-Projector Calibration:

This has been a challenging task during the testing of the project. The Projector is to be positioned far enough from the screen to give a large enough image.
The camera has to be positioned such that the captured image exactly bounds the projector's output image. This task is not easy, having to place the camera very close to the screen while the projector is away from it, both connected to the same machine.

A digital camera with optical zoom can solve this problem.

1.3 The OGL Thread functionality:

This thread uses OpenGL to generate the game's graphics. While the OCV thread works the same for both the two games implemented in this project, OGL works differently as each game has its own graphics. OGL's functionality is explained though the two games operation in the next section.

2. The games:

2.1. The Bull's Eye Game:

This game is an exact implementation of the real bull's eye game. The OGL thread generates a bull's eye that is projected on the wall through the projector. The camera is calibrated such that it captures a full view of the projected scene. The player is to aim at the bull's eye and make the allowed number of shots. OCV detects the shots -- as explained in section 1.1 "The OCV Thread functionality"--, passes their coordinates to OGL through global variables and signals OGL.

As soon as OGL gets the signal, it does the following:

1. Draws a bullet pierce wherever a shot was located in the bull's eye.

2. Calculates a score between 0 and 10 according to how close the shot is to the center of the bull's eye.

3. Displays the last bullet's score, total number of shots made so far, and the average score.

Note: The image processing and graphics generation are **performed in real time**.

Refer to the Appendix for information about OpenCV's real-time operation.

2.2. The Bird Hunt Game:

This game demonstrates the ability of the model to handle more interactive games involving moving-targets.

OCV functions the same as described in the Bull's Eye game, where the OGL operates differently.
It defines and initializes the target objects (birds in this game), loads their graphics into main memory for fast operation and determines the number of objects and overall object motion speed depending on the selected game's difficulty level.

Once object initialization and graphics loading is done, the game starts and OGL handles moving the objects around. OGL performs the following actions as it receives a shot signal from OCV:

1. Checks if the shot has hit any of the target objects by testing for intersection between the blob coordinates and each object's coordinates.

2. If a target has been hit, OGL changes the hit object's course from a horizontal chaotic motion (the simulated motion of flying bird) to a vertical fall-down motion.

Once the falling object reaches the ground, it's marked 'dead' and seizes to be displayed.

When all objects have been hunted down, OGL moves to the next level of the game by doing the following:

1. OGL reincarnates dead objects. Dead objects are reincarnated by being remarked as 'alive' and by resetting some of their parameters such as position and speed. This step saves the time that would be wasted if the objects were to be recreated and reinitialized from the beginning.

2. Adds more objects to increase difficulty.

3. Increases the motion speed of all objects to increase difficulty.

The player has limited time to hunt all targets. There is a variety of ways this time limitation can be implemented. Some of which are:

1. Time limitation remains the same through all levels.

2. Time limitation is increased slightly in conjunction with the game level number.

3. Time limitation is a function of the score achieved in the previous level.

3. OGL's Button Mode:

Entering the *button mode* is equivalent to using the menu in a traditional game. The player can access the game's menu while in the game by *shooting* the menu ***"Shoot-Button"*** in the lower right corner of the screen.

This displays a large cube and two large arrows beneath it. The cube displays the current menu selection. When an arrow is shot, the cube rotates left or right displaying the next menu item. Selecting the current menu item is done by shooting the cube itself.

The available menu items are:

1. Restart Game
2. Player's Name Entry
3. Level Selection
4. Exit Game
5. Back

Entering the player's name is done the same way a menu item is selected. The cube face displays the current letter, selecting a different letter is done by shooting the left and right arrows, and printing a letter is done by shooting the cube while displaying the desired letter. When done entering the name, the player can choose to go back to the previous menu.

4. LASER Gun Construction and Operation:

The laser gun is responsible for emitting a surge of LASER light that will be viewed as a red blob by the camera whenever the trigger is pulled.

Figure2 shows a diagram of the LASER gun.

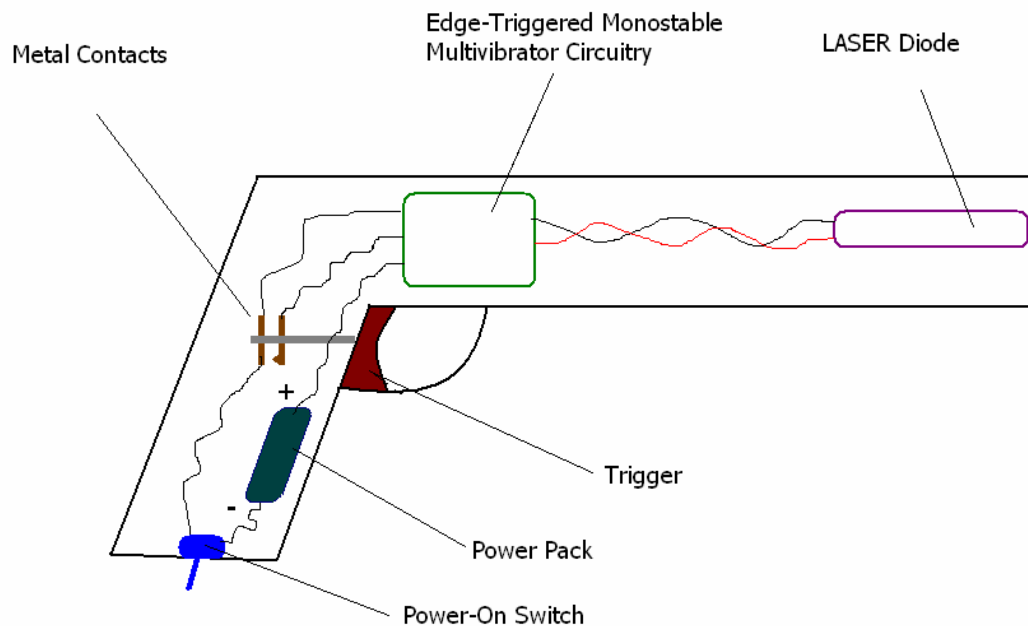

Figure 2: The LASER Gun Construction

When the trigger is pulled, the metal contacts conduct causing the trigger pin on the Edge-Triggered Monostable Multivibrator to go low. This causes the output go high for a period of time T set by the RC network composed of R1 and C1, then go off even if the trigger remains pulled.

Figure3 shows the circuit diagram for the Edge-Triggered Monostable Multivibrator.

The ON time is define by the following equation:
T = 1.1 X R1 X C1

R1 is a variable resistor that is used to adjust the ON time T.

T is adjusted so that it equals at least **one Frame Time**. Since used camera's Frame Rate is about 15 FPS, T must equal at least 1/15 seconds in order for the camera to be able to capture the laser blob.

The values selected for R1 and C1 are 10K and 47 uF respectively, so the maximum value for T is
T = 1.1 X 10K X 47 uF = 0.517 seconds which equals 7.755 Frame Times.

The Power Pack consists of 6 LR44 1.5V Button Cell Batteries stacked back to head forming a 9V power source.

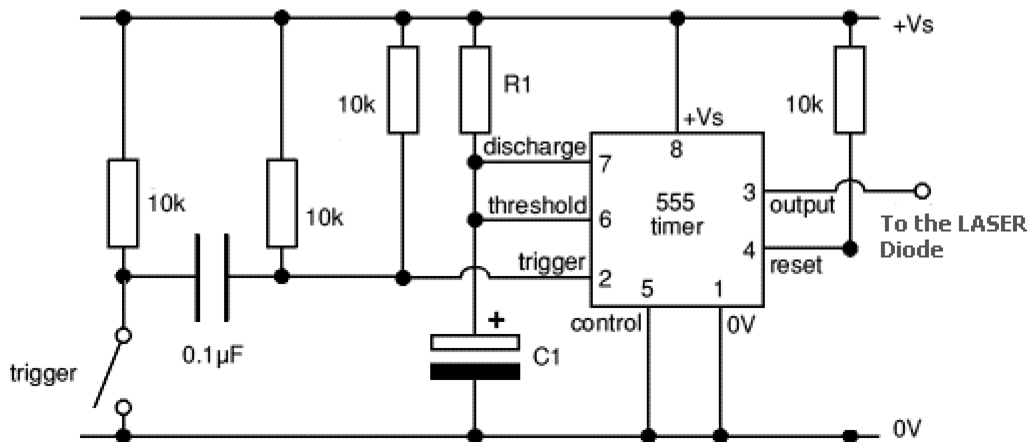

Figure 3: The Edge-Triggered Monostrable Multivibrator Schematic

5. The Implementation:

This project was developed using Visual C++ using the following libraries:

- The Microsoft Foundation Classes (MFC).
- Open Source Computer Vision Library (OpcnCV).
- Open Source Graphics Library (OpenGL).
- Developer's Image Library (DevIL).

Explanation of the project's code is beyond the scope of this document.
The code for the project is extensively commented and easy to understand.

Why OpenCV:

Intel's OpenCV implements more than 300 Image Processing and Computer Vision functions and is specially designed for the Intel Microprocessor families. It is optimized using the microprocessor's MMX and SSE instructions to perform Image Processing in real time.
The reasons for using OpenCV in this project are:

1. The necessity for real time performance.

2. All of the needed image processing functions are implemented in OpenCV.

3. The Computer System used to develop and test this project is an Intel Microprocessor based system.

Why OpenGL:

Perhaps OpenGL is not the best graphics library for game development, but it's sufficient for the purpose of this project. In addition, it was the graphics library I had been familiar with upon starting this project.

6. References

- Open Source Computer Vision Library- Reference Manual, Copyright © 1999-2001, Intel Corporation.

- **OpenCV Wiki**, URL: http://opencvlibrary.sourceforge.net/

- Developer's Image Library Manual, By Denton Woods, Abysmal Software, March 2002.

- Automatic Optimization Using Integrated Performance Primitives, Intel Corporation, URL: http://www.intel.com/technology/itj/2005/volume09issue02/art03_learning_vision/vol09_art03.pdf

- Shooting Sports, URL: http://en.wikipedia.org/wiki/Category:Shooting_sports

- Camera-Projector System, URL: http://vision.eng.shu.ac.uk/mediawiki/index.php/Interactive_Camera-Projector_System#Camera-Projector_System

7. Appendix

7.1 OpenCV

OpenCV means Intel® Open Source Computer Vision Library. It is a collection of C functions and a few C++ classes that implement many popular Image Processing and Computer Vision algorithms.

OpenCV also uses Intel's IIP Library to achieve higher performance.

7.2 IPP

Intel Integrated Performance Primitives (IPP) library is a large collection of low-level computational kernels highly optimized for Intel architectures, including the latest Pentium®, Itanium®, and XScale® processors. It consists of multiple domains that reside in separate dynamic libraries: signal and image processing, matrix processing, audio and video codecs, computer vision, speech recognition, cryptography, data compression, text processing, etc.

7.3 Blob extraction library

A library to perform binary images connected component labelling. It also provides functions to manipulate, filter and extract results from the extracted blobs.

Algorithm

The algorithm is based on Dave Grossman's code with some few additions. Dave Grossman explained his code in this post:

I first saw this algorithm about 30 years ago in a presentation by Gerry Agin at SRI International (then called Stanford Research Institute). I have been unable to find it in any journal articles, so I implemented it by memory. The basic idea is to raster scan, numbering any new regions that are encountered, but also merging old regions when they prove to be connected on a lower row. The computations for area, moments, and bounding box are very straightforward. But the computation of perimeter is very complicated. The way I implemented it was in two passes. The first pass converts to run code. The second pass processes the run codes for two consecutive rows. It looks at the starting and ending columns of a region in each row. It also looks at their colors, whether or not they were previously encountered, and whether a region is new or old, or a bridge between two old regions. There are lots of possible states, but I deal explicitly with the only states that are important. There are 8 of these states:

I think this might be the paper, W. Synder and A. Cowart, “An iterative approach to region growing,” IEEE Transactions on Pattern Analysis and Machine Intelligence, 1983.

W. Synder is a professor at NCSU and explains the algorithm and how to implement it in one pass in his book "Machine Vision".

At each stage, the algorithm is scanning through two consecutive rows, termed LastRow and ThisRow. During this scan, it sees a region in each row. In Cases 1-4, the region in LastRow starts 2 or more columns before the region in ThisRow. In Case 1, the region in LastRow ends one or more columns before the region in ThisRow starts. Therefore, these regions are NOT connected. In Case 2, the region in LastRow starts before the region in ThisRow, but the LastRow region continues at least to the column just before the region in ThisRow starts. Or it continues further, but it ends before the region in ThisRow ends. Therefore, if these regions have the same color, then they are connected.

RickyPetit changed my code to provide 4 or 8 connectivity. I had used 6/2 connectivity, which meant that at a corner like this 01 10 the 0's are connected but the 1s are not connected. In other words, the connectivity of a pixel is given by this mask: 110 1X1 011 i.e., pixel X is connected to the 6 pixels with a 1 but is not connected to the 2 pixels with a 0. This introduces a slight bias in favor of regions that slope to the left, but I consider it preferable to using either 4-connectivity or 8-connectivity.

7.4 Developer's Image Library (DevIL)

Developer’s Image Library was previously called OpenIL, but due to trademark issues, OpenIL is now known as DevIL. DevIL is an open source programming library for programmers to incorporate in to their own programs. DevIL loads and saves a large variety of images for use in a software developer’s program. This library is capable of manipulating images in various ways and passing image information to display APIs, such as OpenGL and Direct3D.