

ECE532 Design Project Final Report

Latchezar Dimitrov, Jonathon Riley, Steven Doo

1 Overview

1.1 Goals

The goal of this project was to implement a “laser-pointer” screen interface system and use that in the design of a “shooter-like” video game. The user would be able to use a laser pointer to point at and destroy targets moving across the screen.

The game is to run completely in software, executed by the Microblaze processor. The coordinates of the laser would come from a hardware module which processes frames coming from the video camera. The hardware would locate the coordinates of the point, and pass them along to the software. The software would be doing the appropriate scaling and the coordinates could then be used as the cursor for the game.

1.2 Background and Motivation

The idea behind this game originally came from the “Fruit Ninja” game on the iPhone. In this game, fruits would be “tossed” around the screen and the user would “slice” these fruits by using their finger to swipe the screen. We thought that this would be an interesting game to implement on the Xilinx board along with laser pointers and video cameras. Also, the laser-pointer screen interface system implemented in this project could be used for other things besides a game as well (i.e. – laser pointer white board, etc).

1.3 System Overview

Below is the system level overview diagram.

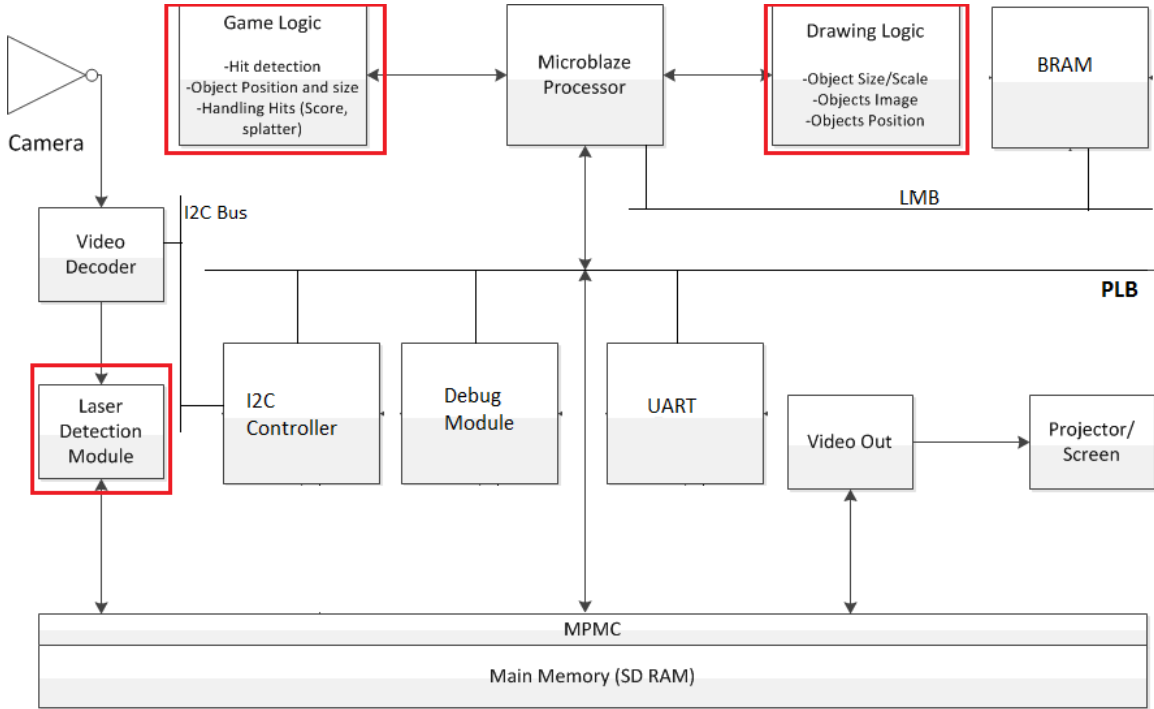


Figure 1 – System Level Overview

1.4 IP and Hardware Descriptions

IP	Function	Author
laser_detector (called video_to_ram in actual project)	<ul style="list-style-type: none"> - Modified version of video_to_ram module from the example Video to RAM system - Detects the position of the laser and writes it to a specific location in the RAM. 	Group (modified video_to_ram)
Microblaze	Processor core for setting up video, and setting up/executing the game	Xilinx; program implemented by group; used some code from previous projects/examples

dlmb/dlmb_ctrl	Data memory controller interfaced through Local Memory Bus (LMB)	Xilinx
llmb/ilmb_ctrl	Instruction memory controller interfaced through LMB	Xilinx
PLB (plb_v46)	Processor Local Bus used to interface to various IP cores including the microblaze, laser_detector, and video_out	Xilinx
IIC (xps_iic)		Xilinx
Debug module (mdm)	Debug Module to enable XMD	Xilinx
DDR_SDRAM (mpmc)	Memory to hold laser coordinates data, and video data	Xilinx
Video Daughter Card	Interface with video camera	Analog Devices
Video camera	Captures the projected game and laser dots on the screen	
UART	Debug info	Xilinx
VGA projector	Display game	
clock_generator		Xilinx
led_debug_mux	LED debug info	

2 Project Outcome

The project went as planned and the team achieved what we wanted to. We were able to implement the laser-to-screen interface and be able to play the implemented game with it. The final game allows two users to be playing it at the same time. Two laser dots are detected by the hardware, and in the game, a line forms between the two dots. Whatever objects hit this line will be “destroyed.” There are also special objects that the user must avoid or try to hit. More details regarding this can be found later in the report. The game also features a score which is displayed at the bottom of the screen.

The laser detection works accurately. We were able to configure it so that only the laser dot would be picked up by the detection hardware. However, we did encounter some problems as the screen itself acted as a light source under some lighting situations. When the room is dark, the projected game appears brighter to the camera. As a result, this causes the detected laser point to be unstable as the detector may pick up the light from the screen. To solve this problem, we considered adding code to stabilize the point in software. However, this would cause delay, so the solution was to simply ensure that the room is well lit. Overall, we are pleased with the final outcome.

3 Description of Modules

3.1 Laser Detector Module

The laser detector hardware is entirely housed inside the (modified) video_to_ram.v source file. The existing video_to_ram system (originally developed by Jeffrey Goeders) was used as a starting point for the laser detector implementation since it already accomplished the first task that we require in our application - namely the conversion of the incoming video feed from YCbCr to RGB and the population of the two RGB line buffers (Figure 2).

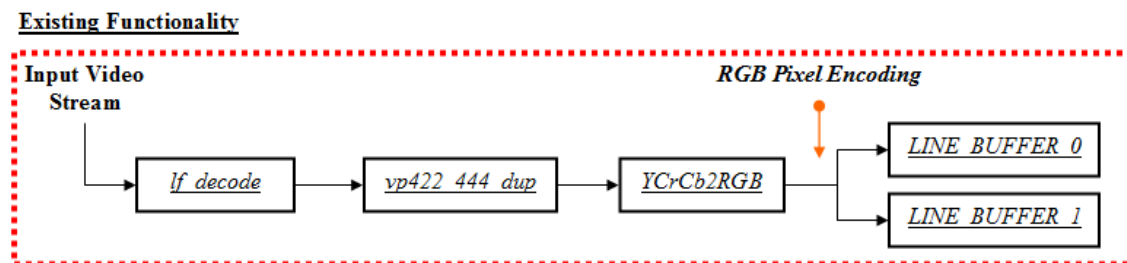


Figure 2

Further discussion on the implementation of the components in the diagram above will not be provided here as they were effectively used "as is" in our application. The laser detector logic is implemented on top of the structure shown in **Figure 2**.

The detector hardware reads every pixel out of the line buffers (alternating between the buffers such that the one currently being read from is always the one that is NOT being written to by the RGB decoder). As every pixel is being read out of the buffer, its location in the overall frame is kept track of. The pixel's color encoding is then checked to see whether or not it falls within the ranges specified by the **current color limits register**. If it does, then the pixel's coordinates in the overall frame will be written to one of two location registers - **location register 1** if this is the first time that a pixel of the appropriate color has been detected in the current frame or **location register 2** otherwise (**Figure 3**).

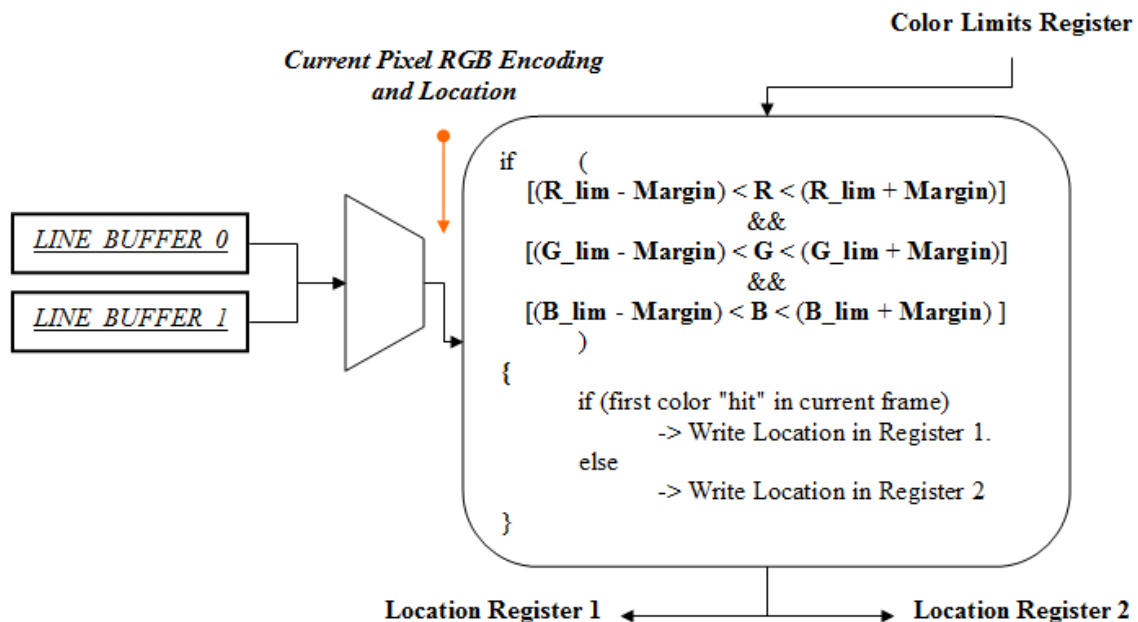


Figure 3

The encoding used in the three main (32-bit) registers (i.e. **current color limits register**, **location register 1** and **location register 2**) is described below (**Figure 4**):

Location Registers 1 & 2 Encoding

X_coordinate [31:21]	Y_coordinate [20:10]	Frame_number [9:0]
--------------------------------	--------------------------------	------------------------------

Color Limits Register Encoding

R_lim [31:26]	G_lim [25:20]	B_lim [19:14]	Margin [13:8]	Refresh_rate [7:0]
-------------------------	-------------------------	-------------------------	-------------------------	------------------------------

Figure 4

The frame number in the location registers is only used for testing / debugging purposes as it simply holds the value of a running counter that gets incremented at the completion of each video frame. The refresh rate number in the **current color limits register** will be explained further below.

Intuitively, the detection algorithm will set the top-left-most pixel that meets the color criteria in **location register 1** and the bottom-right-most pixel that meets the color criteria in **location register 2 (Figure 5)**.

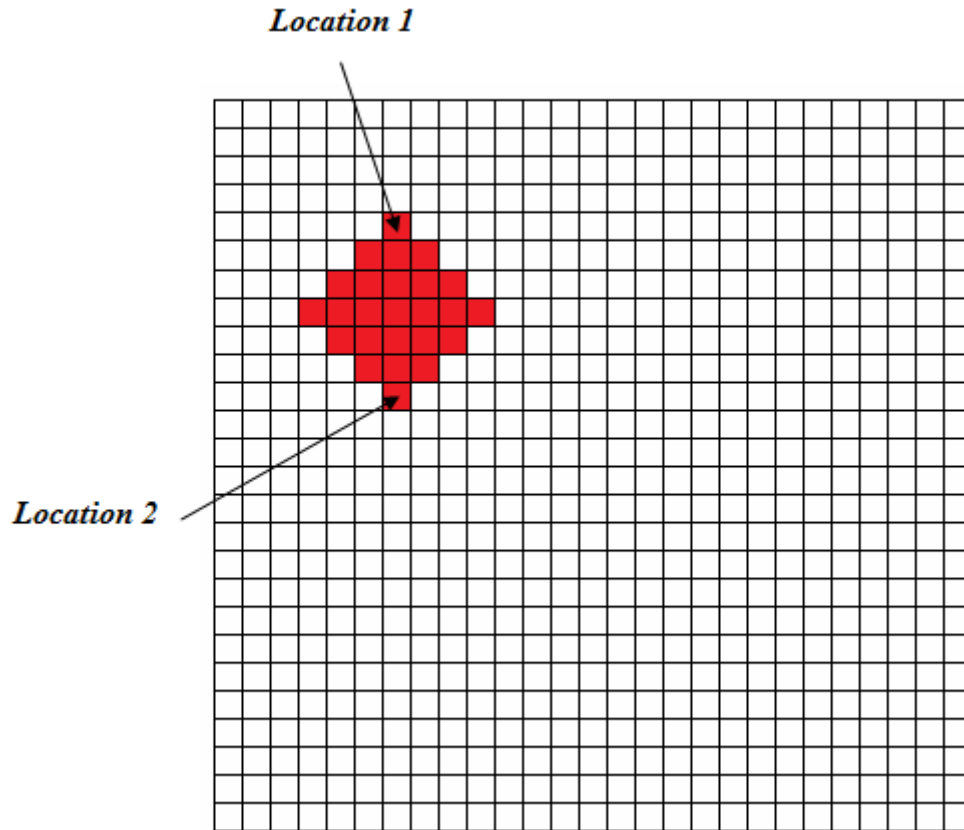


Figure 5

If there are two laser dots visible on the screen, **location register 1** would get the top-left-most pixel associated with the first dot and **location register 2** would get the bottom-right-most pixel associated with the second dot (**Figure 6**). It is these pixel locations that will henceforth be regarded as the coordinates of the first and second lasers respectively.

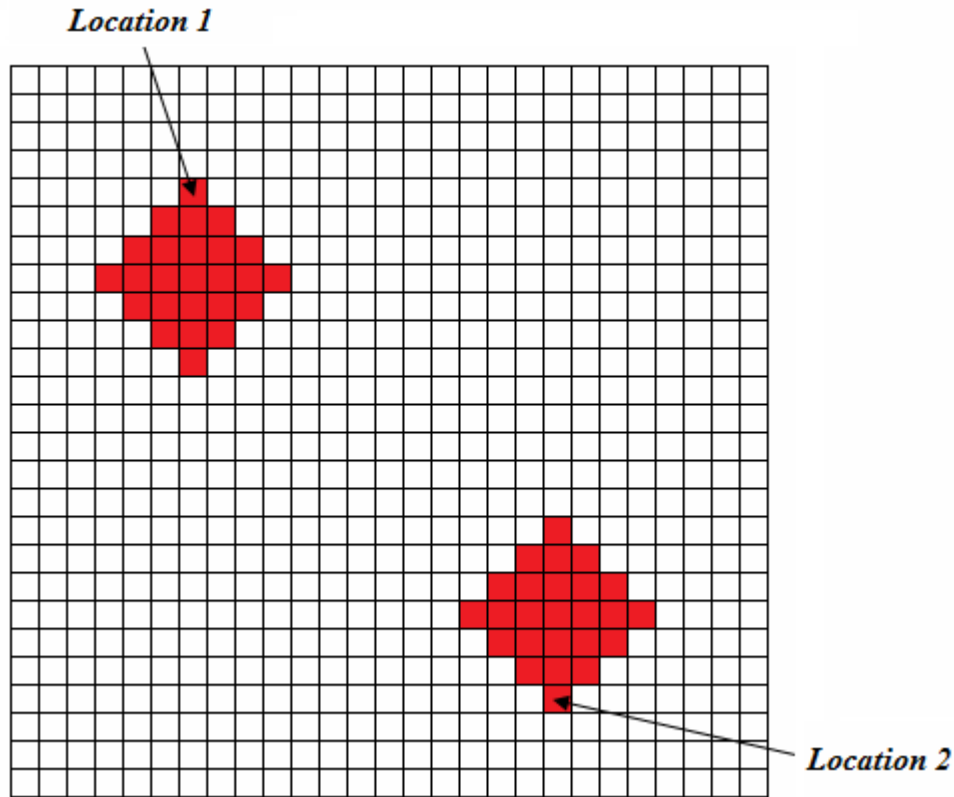


Figure 6

In the event that no pixels meet the color criteria, then both location registers would retain the coordinates (0, 0) since that is what they are reset to at the beginning of each new frame.

An overview of the laser detector FSM is shown below (**Figure 7**). Essentially, an entire frame is processed while the FSM spins in *S_LINE_HOLDER*. During this time, the location registers are updated accordingly with the coordinates of the two laser dots. Once the whole frame is finished, the FSM either moves on to another frame or it proceeds to perform memory updates depending on whether or not a particular refresh count has been reached (under normal operation, the refresh rate is set to 0, which means that memory updates will occur after each frame). If memory updates are to be done, the FSM first reads a 32-bit word from a predefined location in memory (0x45000000) and updates the **current color limits register** with the provided data. It

then writes the values currently stored in the two location registers to memory addresses 0x45000004 and 0x45000008 which are associated with laser 1 and laser 2 respectively.

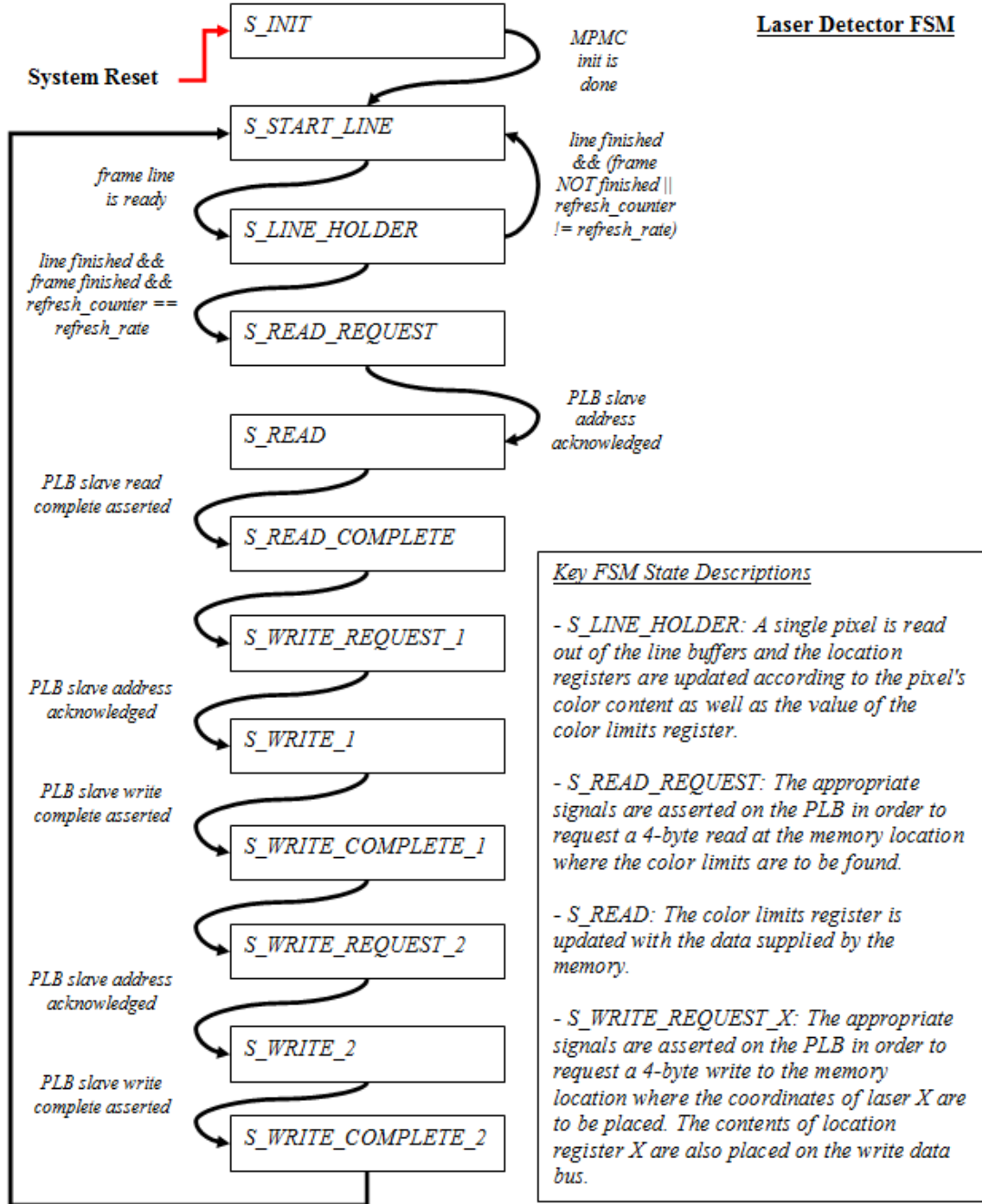


Figure 7

3.2 Software Modules

3.2.1 VGA Modules (VGA.c/h)

This module consisted of code used to write pixels to the screen. It was taken from <http://www.eecg.toronto.edu/~pc/courses/432/2009/projects/tft-edk.txt>. A new function named “cal_screen” was added to be able to print the calibration screen sequence. Depending on the parameters provided by the calibration module (section 3.2.3) the cal_screen module will draw an X at the appropriate spot on the screen, directing the user to point the laser at that spot for calibration. The cal_screen() function also effectively acts as the replacement of the “clear_screen()” function as it clears the screen and draws a 590x430 box in the screen to represent the playing area.

Other unmodified functions in this module allow other the other modules to write a pixel given the x,y location. It also does double buffering by swapping between two vga screens in memory.

3.2.2 Laser Detector Software Interface (Laser_track.c / Laser_track.h / video_setup.c)

This is the software abstraction that directly communicates with the laser detection hardware (through memory addressing). In turn, this abstraction (i.e. methods and data structures) (**Figure 8**) is used by the actual game software whenever laser coordinates are needed.

```
struct LaserTrackerModule {
    int colorLimits;
    long long currentPositions;
};
struct LocalTrackerDataStruct {
    long long in_buf_L;
    int in_buf_I, out_buf_I;
    int X_pos_1, Y_pos_1, F_num_1, X_pos_2, Y_pos_2, F_num_2;
};
void getRawLaserCoordinates(struct LaserTrackerModule * modulePtr, struct LocalTrackerDataStruct * dataStructPtr);
```

Figure 8

The interface is established in main() [video_setup.c] through the declaration of two pointers and a local data struct (**Figure 9**).

```
//==== Tracker Variable Instantiations =====  
struct LaserTrackerModule * LM_ptr;  
struct LocalTrackerDataStruct LocalTrackerData;  
struct LocalTrackerDataStruct * LD_ptr;  
//=====
```

Figure 9

LM_ptr is made to point to the color limits word in memory (0x40000000) while *LD_ptr* is made to point to *LocalTrackerData*. *LocalTrackerData* is where the decoded coordinates are to be stored. All of the needed initializations are shown below (**Figure 10**). Note how as part of the initialization, the color limits which the hardware will use, are set to represent a particular color (actual value is defined in Laser_track.h).

```
//==== Initializing the Tracker =====  
LM_ptr = (struct LaserTrackerModule *) LASER_TRACK_BASE_ADDR;  
LD_ptr = &LocalTrackerData;  
LD_ptr->out_buf_I = (int) RED_LASER_COLOR;  
LM_ptr->colorLimits = LD_ptr->out_buf_I;  
LD_ptr->in_buf_I = LM_ptr->colorLimits;  
//=====
```

Figure 10

Once the initializations are performed, the rest of the game software can obtain the laser coordinates by first calling the `getRawLaserCoordinates()` function [Laser_track.c] with the LM and LD pointers as parameters. This function will perform the needed memory access to extract the encoded laser positions and will then decode them (based on the structure shown in **Figure 4**) before saving all of the coordinates in the *LocalTrackerData* struct. After calling `getRawLaserCoordinates()`, the caller can directly access the coordinates from the *LocalTrackerData* struct (to which *LD_ptr* points). The

example below shows how the laser coordinates can be obtained in the rest of the game software (**Figure 11**).

```
getRawLaserCoordinates(LM_ptr, LD_ptr); //Getting the Coordinates
if (LD_ptr->X_pos_1 !=0 && LD_ptr->Y_pos_1 !=0 && LD_ptr->X_pos_1 < 150
{
    .
    .
    .
}
```

X coordinate of Laser 1 (integer)

Figure 11

3.2.3 Setup and Calibration Module (video_setup.c)

Below is the flow-diagram for this software module:

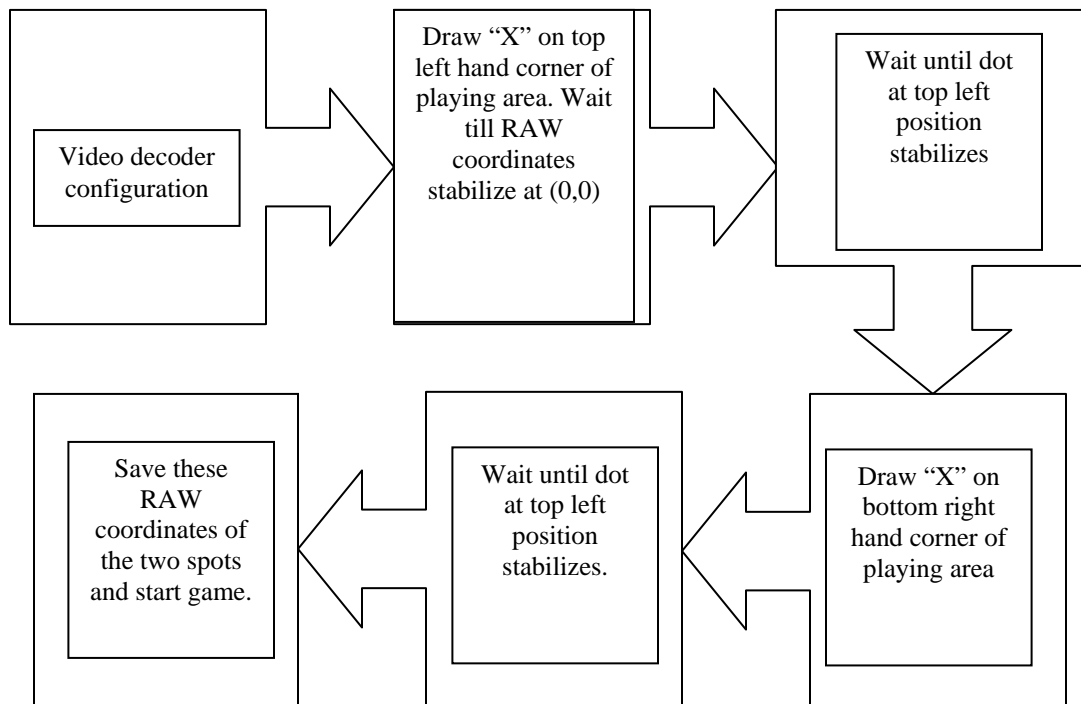


Figure 12

This module first sets up the video decoder module. This was done with existing code taken from the 2010 virtual pong project. Configuration constants are read and sent to the video decoder one by one through the IIC bus.

The goal of this algorithm is to obtain the coordinates of the top left hand corner and the bottom left of corner of the screen as seen by the camera. With these values, we are able to convert the coordinates of the laser dot from the camera's domain to the domain of the game with the following formula:

Let

- X_{vid_cam}/Y_{vid_cam} = raw coordinates of the laser as the video camera sees it
- X_{ul}/Y_{ul} = coordinates of the upper left hand corner of the playing area found during the calibration stage
- X_{lr}/Y_{lr} = coordinates of the lower right hand corner of the playing area found during the calibration stage
- X/Y_{Game} = coordinates relative to the game.

$$X_{Game} = (X_{vid_cam} - X_{lr}) / (X_{ul} - X_{lr}) * 640$$

$$Y_{Game} = (Y_{vid_cam} - Y_{lr}) / (Y_{ul} - Y_{lr}) * 480$$

3.2.4 Object Modules (Objects.h, Objects.c, star.h, evilbomb.h, smiley.h)

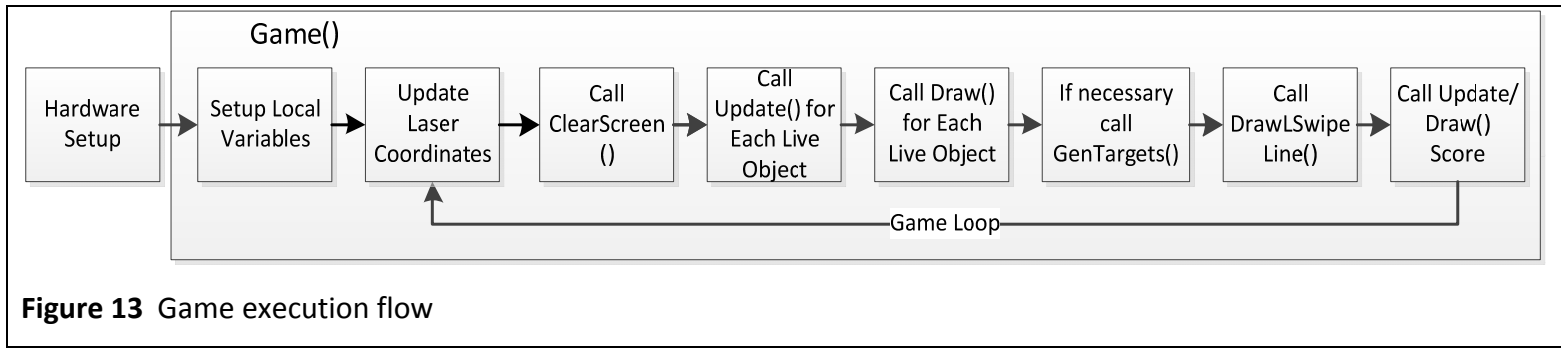
The object module contains the definitions for various structs that do not belong in other area of the program. Amongst them is the struct definition for the Target object (for a full struct breakdown see Appendix A.1). Any object that is to be drawn to the screen is an instance of the Target struct. Additionally, Objects.h/.c contains other struct definitions such as Vector2, IntVector2, and IntVector4 (see Appendix A.2) as well as

functions for operating on the Vector structs. Vector structs are used throughout the program to group together related ints/floats, primarily to indicate points or movement direction.

Also contained as part of this module are a series of static arrays that store the RGB value of various bitmap images. These static arrays are used for drawing the various objects to the screen using the images represented by these arrays as their representation (See Appendix A.3 for an example of a BMP RGB array). The individual objects are stored in separate header files along with a set of constant static ints the rest of the program uses to determine the object's size and other properties. This allows changes to objects to be made quickly and easily with only the header file and possibly the object generator function needing to be modified.

3.2.5 Game Module (Game() in Game.c)

The primary function in the project, the Game function takes over after the initial hardware setup has been completed. The Game function has a set of local variables used for keeping track of the game's state/game events. When first called, Game() will set these state/event variables to their initial state. Afterwards, the game enters what is essentially an infinite loop that fetches the user input (laser coordinates), updates the objects and then draws the objects before starting again. It is modeled after the standard operation of a game loop (see Figure 13). The various functions called by Game are capable of modifying the game state, so Game() is primarily responsible for enforcing any high level events/state properties in addition to executing the primary loop.



3.2.6 Update Laser Coordinates Module (Game.c)

This function is responsible for sampling the laser detection hardware module to get the coordinates of the laser pointers. The laser detection module is set to store the laser coordinates at specific locations, which can be accessed to determine the where the laser pointers currently are.

Within Game.c is a variable `IntVector4 resCoords`. When `UpdateLaserCoords` is called, the function reads in the raw laser coordinates detected by the camera. Making use of the calibration settings, the function scales the raw laser coordinates into in game coordinates and then stores these values in `resCoords`

3.2.7 Draw Target Module (Game.c)

Function Definition:

```
void DrawTarget(struct Screen* targetScreen, struct Target* target);
```

As the name indicates, `DrawTarget` is responsible for taking an in game object and drawing it to the screen. To that end, `DrawTarget` accepts as parameters, the screen to draw to as well as the target to draw. `Game()` iterates through each target, calling `DrawTarget` once per target per loop iteration.

The function will, using the target's size as indexes, iterate through the RGB array attached to the target, drawing each pixel sequentially. The drawing routine supports the Swap property of the object and if the object is set as Swapped, the function iterates through the target's SwappedData RGB array. Otherwise it iterates through the target's ImageData RGB array. After finishing drawing the object, the function returns back into the calling function.

3.2.8 Update Target Module (Game.c)

Function Definition:

void UpdateTarget(struct Target* target)

The UpdateTarget function is responsible for updating the game targets in response to the laser coordinates input as well as the passage of time/loop iterations. For each target in the game that is Alive, Game() will call Update once per loop iteration.

Because it exists inside Game.c, this function has access to the resCoords and LoopCount variables which will give it access to the laser coordinates and the Loop iteration count respectively.

The first action performed by the function is to determine if the target has been killed by the laser input coordinates. For information on how hits are determined, see **3.2.10 HitDetection Module**.

If the object is killed by the hit detection modules, the UpdateTarget function will return to the calling function. Otherwise, the UpdateTarget function continues execution.

After the hit check, the function checks the targets lifetime. If the target's Timed property is set and the current Loop Iteration is greater than the target's ExpiryTime property, the object is killed and the game is set to generate a new target. Since this form of kill was not triggered by the player, they are awarded no points and additional effects are not activated.

Next the function checks if the target's Swap property is set. If Swap is set and the current loop iteration is greater than the target's LastSwap + SwapSpeed, the object's Swapped property is inverted. The target's Swapped value is used to determine which of its RGB arrays are used to draw the target.

Finally, the UpdateTarget function updates the targets position based on the target's speed. If the new Position of the target/speed of the target will put it out of the playing area, Update reflects the target's speed to force it back into the playing area.

3.2.9 Generate Targets Module (Game.c)

Whenever a target is killed, the killing function is responsible for setting the Game variable readyForGen to true. Inside the main game loop, the game checks if this property is set to true. If it is true, the game calls the GenTargets function;

Function Definition:

void GenTargets(struct Target* TargetArray, int GenAmount);

The function takes as inputs an array of targets, and the maximum amount of targets to generate. The functions will iterate through the array and for each target in the array that is not marked as Alive, it will regenerate that target into a new target. A random number generator is used to determine what the target should be regenerated into. As part of the generation module, a separate function is used for each of the possible things the target can be regenerated into. The current Generative functions are:

void MakeSmiley(struct Target* target);

Makes the basic target type worth 25 points on kill.



void MakeEvilBomb(struct Target* target, int CurrentIteration);

Makes a bad target that when killed reduces score by 500. The target expires after several iterations



void MakeStar(struct Target* target, int CurrentIteration);

Makes a special target type that is worth 200 points when killed. Also activates temporary invincibility from bombs upon being killed. Expires after a few iterations.

After generating the new target, it is given a random position on the playing field as well as a random, but upper and lower bounded, speed. Finally, the number of targets generated is increased by one. As long as the end of the array hasn't been hit and the number of generated

targets is less than GenAmount, the function will continue to iterate and generate new targets from dead targets.

An example of the generative functionality is shown in Figure 14. The Generator determines that Targets[N-2] is Dead, so after a random number generation, the function determines it should turn it into a star object using the void MakeStar() function. After passing through the function, the target has been regenerated into a star target and will be included into the game on the next iteration. Each function is responsible for setting the necessary data of the target. This follows a factory design

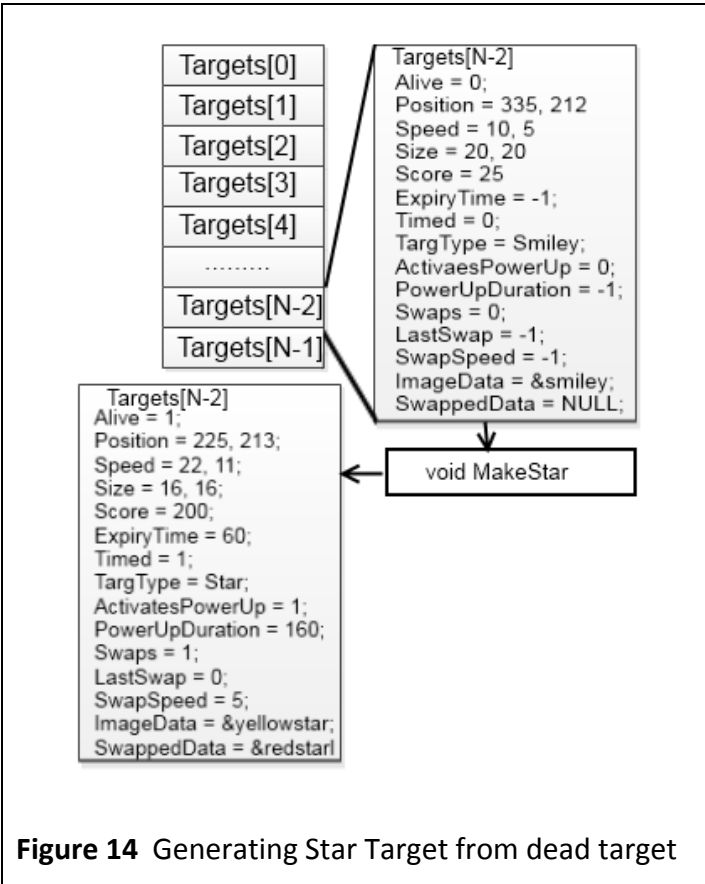


Figure 14 Generating Star Target from dead target

pattern, where each function is responsible for assembling the target into the desired target type.

3.2.10 Hit Detection Module (Game.c)

Hit detection is performed inside the Update stage of the Game. Every time Update is called for a target, HitDetection is also called. There are two built in hit detection schemes based on the game's current mode;

1. Point Based Detection (for GameMode = Points)

For each set of laser Coordinates, checks if the coordinates are with a certain distance of the target's position. The distance is defined inside Game.c using the LASER_EFFECTIVE_RADIUS variable. If the laser coordinates are within that distance, the target is killed and the game is set to generate a new target. The player's score is incremented by a multiple of the target's Score property. The multiple is based on the speed and size of the target that was destroyed as faster, smaller targets are more difficult to hit with the laser pointers. Finally, if the target activates an additional in game effect, it is triggered by a call to ActivatePowerUps(). See 3.2.12 for information on Powerups.

2. Line Base Detection (for GameMode = Line)

For the set of laser coordinates, a line is assumed to exist between them. That line is treated as a line segment and if the target is passed through by that line segment, the target is killed and the game is set to generate a new target. The player's score is incremented by the target's score property.

Determining if the line segment passes through a target is determined by the function LineCutsObject. Similarly, after target destruction, if the target activates an additional in game effect, it is triggered by a call to ActivatePowerUps().

The LineCutsObject Function is used exclusively as a Hit Detection function so it is considered part of the Hit Detection Module.

It has the following Definition:

```
int LineCutsObject(struct Target* target, struct IntVector2* LineStart, struct  
IntVector2* LineEnd);
```

When called, LineStart and LineEnd will form the “kill line” that, if it passes through the target, will kill the target. In order to perform the hit test, the target’s four sides are broken up into four line segments. The segments are tested for intersection against the kill line using vector math (see **Appendix A.4** for the intersection test code). If at least two line segments are intersected by the kill line (one segment for entry into the target, another segment for exiting out of the target) the target is killed. Otherwise, the function takes no action and returns to the calling function.

3.2.11 Update Score Module (Game.c)

This function takes in the score in integer form and draws the digits onto the bottom of the screen. The algorithm simply extracts each digit of the score and draws them onto the bottom of the screen in the correct order. Each digit was a BMP image created with paint, and then converted to an RGB array using the tool described in section 3.3.

3.2.12 Power Ups

Power ups are special game states that alter the way the game is played. Currently one power up is available although at least one more will be attempted for the final product. PowerUps are activated during the Hit Detection module. If a target is destroyed by the lasers and the ActivatesPowerUp property is set, the Hit Detection module will call ActivatePowerUps, passing the destroyed target as an argument.

Function Definition:

```
void ActivatePowerUps(struct Target* target, int CurrentIteration);
```

The function uses a switch statement on target->targType to determine the correct variables to set for the given target.

The function UpdatePowerUps() is also called within the main game loop.

Function Definition

```
void UpdatePowerUps(int CurrentIteration);
```

The function compares the current iteration against the expiry iteration for any timed powerups. If the current iteration is greater than the expiry time of the power, the power up is deactivated and any variable set to enforce the power up are reverted to their initial values.


Power Up Name	TargType	Symbol	Effect
Star Power	Star		While the powerup is in play, the played can destroy any target without losing points (namely bombs)
Nuke	Nuke		Adds a nuke to the player inventory. On use, Destroys everything on screen and adds their kill scores to the players scores. Ignores targets with negative scores
Bonus Play	Bonus		Doubles the score received when killing positive targets.

Figure 15

3.2.13 Update Score Module (Game.c)

This function takes in the score in integer form and draws the digits onto the bottom of the screen. The algorithm simply extracts each digit of the score and draws them onto the bottom of the screen in the correct order. Each digit was a BMP image created with paint, and then converted to an RGB array using the tool described in section 3.3.

3.2.14 Draw "Swipe" Line Module (Laser_track.c)

This is the function that draws out a line between two points on the screen. It is used to draw out the "swipe" line between the two lasers. The function is passed the X and Y coordinates of both points (in addition to a desired line color) as parameters.

The drawing algorithm is based on a single loop that increments a counter from 0 to DeltaX or Delta Y (**Figure 16**) (depending on which one is larger).

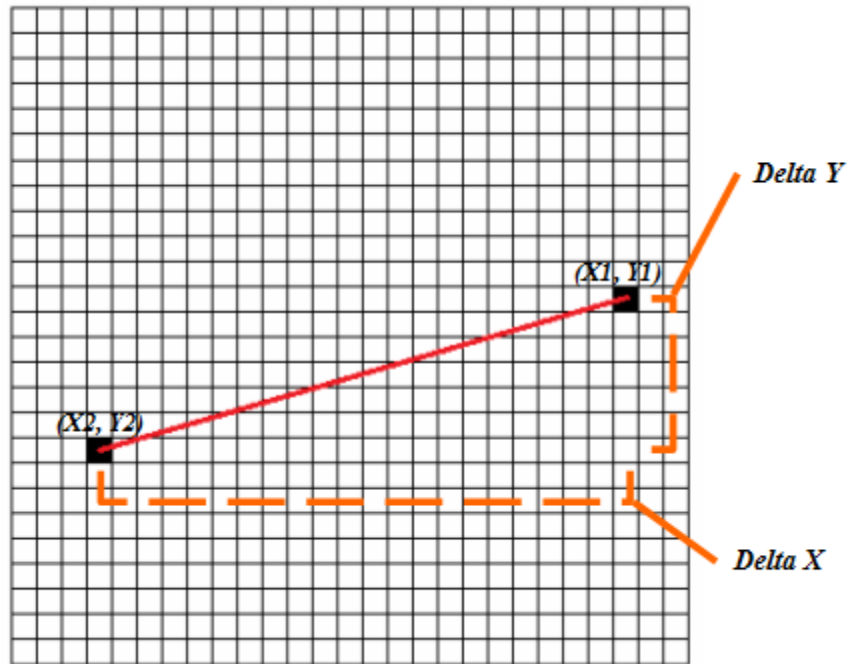


Figure 16

In the above example, Delta X has a larger magnitude so the counter will count up to Delta X. In each iteration of this loop, a single pixel is drawn at:

$$(X2 + \text{count}, Y2 + \text{count} * \text{DeltaY} / \text{DeltaX})$$

Essentially, the X coordinate will always be incremented while the Y will only be incremented at specific intervals (since the offset from Y2 is rounded off to an integer amount). However, both the X and Y coordinates of a successive pixel can only, at most, differ by 1 from those of the previous pixel. The end result is a continuous line (i.e. one without breaks) (**Figure 17**).

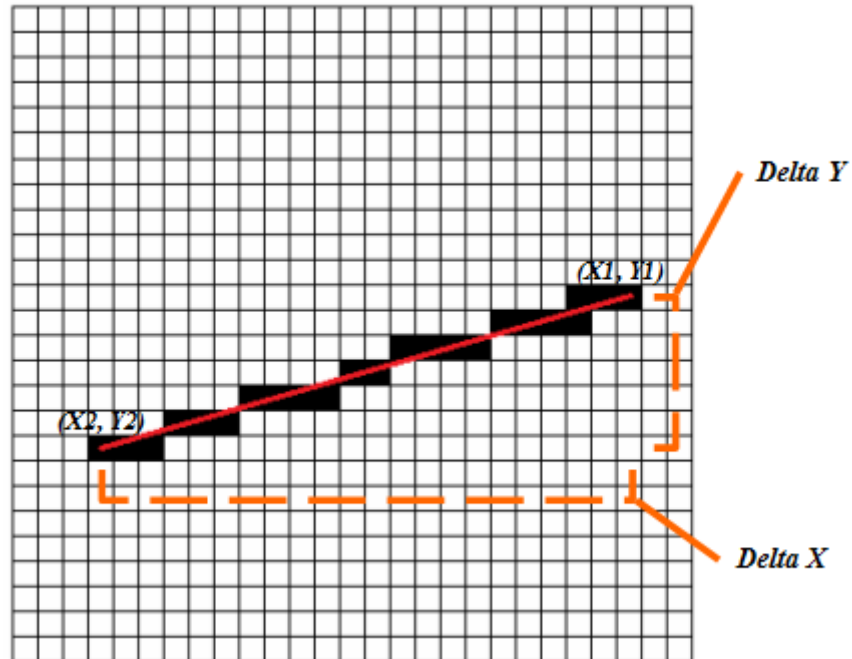


Figure 17

This algorithm is slightly expanded upon in the `drawSwipeLine()` function in order to account for all possible corner cases (i.e. negative deltas) and ultimately, it is able to produce a line between any two points on screen.

Once the line is drawn, the function also draws markers at each end (which in our application, represent the laser locations). These markers are in the form of triangles centered at $(X1, Y1)$ and $(X2, Y2)$ respectively (**Figure 18**).

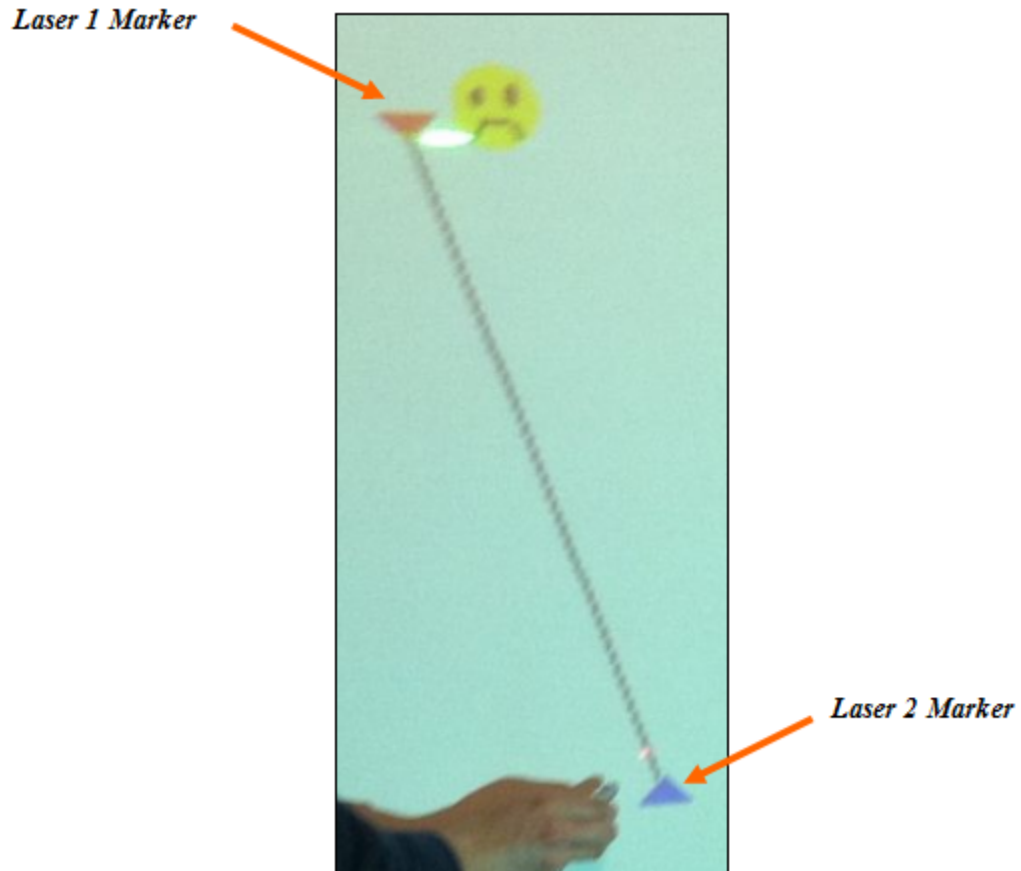


Figure 18

3.3 Extra Tools

We created a tool to convert 24-bit color BMP images into RGB arrays. This tool was created by modifying an existing bmp file parser found here:

<http://paulbourke.net/dataformats/bmp/>.

This tool takes in the bmp file as input and outputs the RGB array of the image which can be declared in the code and used to draw the image onto the screen.

4 Description of Design Tree

This section provides an overview of the directory structure.

- /__XPS/ – Xilinx generated directory*
- /blkdiagram/ – Xilinx generated directory*
- /data/ – Xilinx generated directory*
- /doc/ – Contains group report*
 - /report.pdf – Group report*
- /etc/ – Xilinx generated directory*
- /lib/ – Xilinx generated directory*
- /pcores/ – Cores directory*
 - /video_to_ram_v1_00_a/ - Laser Detector Core*
(other cores in this directory not used)
- /report/ – Xilinx generated directory*
- /sw/ – Software directory*
 - bomb.h – bomb image array*
 - evilbomb.h – evilbomb image array*
 - Game.c – code for game*
 - Game.h*
 - Laser_Track.h – code for reading laser coordinates*
 - Laser_Track.c*
 - Objects.h – contains objects modules*
 - Objects.c*
 - Smiley.h – smiley image array*
 - Sniper.h – sniper image array*
 - Star.h – star image array*
 - VGA.h – contains VGA sw modules*
 - VGA.c*
 - Video_setup.c – contains video initialize and initial setup code*
- /README – information regarding project directory*

Appendix A – Software Constructs

A.1 Target Struct

```
1. struct Target
2. {
3.     //Indicates whether the object should still be drawn to the screen/Can be hit
4.     int Alive;
5.     //X, Y coordinates indicating the objects current position
6.     struct IntVector2 Position;
7.     //X,Y value indicating the current velocity of the object
8.     struct IntVector2 Speed;
9.     //X, Y values indicating the length/height of an object
10.    struct IntVector2 Size;
11.    //The colour to draw the object as. Only used if ImageData is not set
12.    int Colour;
13.    //The change in score caused by killing this object
14.    int Score;
15.    //If the object is timed, how many iterations should it be kept alive for
16.    int ExpiryTime;
17.    //Indicates if the object will be destroyed after a number of iterations
18.    int Timed;
19.    //Specifies the target type. Used to generate object specific behaviour
20.    enum TargetType targType;
21.    //indicates if destroying this object activates an in game bonus
22.    int ActivatesPowerup;
23.    //Number of iterations the powerup lasts for
24.    int PowerUpDuration;
25.    //Indicates if the object swaps between ImageData and Swapped Data
26.    int Swaps;
27.    //Indicates whether the object should be drawn with ImageData or SwappedData
28.    int Swapped;
29.    //Iteration the most recent swap occurred on
30.    int LastSwap;
31.    //Indicates how many iterations between swaps
32.    int SwapSpeed;
33.    //Points to the primary RGB array for the object
34.    unsigned int* ImageData;
35.    //Points to the secondary RGB array to use if the object is Swapped
36.    unsigned int* SwappedData;
37. };
```

The full struct definition for the Target object.

A.2 – Vector Structures

```
1. struct Vector2
2. {
3.     float x;
4.     float y;
5. };
6.
7. struct IntVector2
8. {
9.     int x;
10.    int y;
11.};
```

The struct definitions for the Vector objects used in the game

```
1. struct IntVector4
2. {
3.     int x1;
4.     int y1;
5.     int x2;
6.     int y2;
7.};
```

A.3 – Image Arrays

```
1. static unsigned int smiley[400] = {0x00ffffff,
```

```

2.  0x00ffffff,
3.  0x00ffffff,
....
400.  0x00ffffff
401. };

```

Array of RGB values for a dumped BMP file

A.4 Intersection Test

```

1.  int Intersects(struct IntVector2 L1_Start, struct IntVector2 L1_End, struct IntVector2 L2_Start,
2.  struct IntVector2 L2_End)
3.  {
4.      struct IntVector2 E, F, G, H, Q;
5.      float LimitTop, LimitBottom;
6.      E.x = L1_End.x - L1_Start.x;
7.      E.y = L1_End.y - L1_Start.y;
8.
9.      F.x = L2_End.x - L2_Start.x;
10.     F.y = L2_End.y - L2_Start.y;
11.
12.     G.x = -E.y;
13.     G.y = E.x;
14.     H.x = L1_Start.x - L2_Start.x;
15.     H.y = L1_Start.y - L2_Start.y;
16.     LimitBottom = -F.x * E.y + E.x * F.y;
17.     if (LimitBottom != 0)
18.     {
19.         LimitTop = -E.y * H.x + E.x * H.y;
20.         LimitTop /= LimitBottom;
21.         if (LimitTop >= 0 && LimitTop <= 1)
22.         {
23.             Q.x = L1_Start.y - L2_Start.y;
24.             Q.y = L1_Start.x - L2_Start.x;
25.             LimitTop = F.x * Q.x - F.y * Q.y;
26.             LimitTop /= LimitBottom;
27.             return LimitTop >= 0 && LimitTop <= 1;
28.         }
29.     }
30.     return 0;
31. }

```

Limit Intersection Test. The Function computes the scale by which a line segment would need to be multiplied by to intersect another line segment. If the Multiplier is ≥ 0 or ≤ 1 , the lines already intersect and no extension would be necessary. Returns 1 (true) if the segments intersect and 0 if they do not

Appendix B – PLB Master, Burst Write Operation



5.1.16 Fixed-Length Burst Write Transfer

Figure 5-15 shows the operation of a fixed-length burst write from a slave device on the PLB. During the request phase of the transfer, the master has continuously provided the length of the burst on the Mn_BE signals and is requesting to write 4 words. The slave uses this length value to count the number of transfers and assert the SI_wrBTerm signal in the cycle before the last assertion of the SI_wrDAck signal.

Figure 5-15. Fixed-Length Burst Write Transfer

