University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

ECE Fourth Year Design Project Final Report

Project: René Desdartes - I Dart, Therefore I Am

Group 39

Group 039 members:

Project Member	Email	Signature
Ching-Pei Lin	c7lin@engmail.uwaterloo.ca	
Ken Woo	k2woo@engmail.uwaterloo.ca	
John Crisostomo	jpcrisos@engmail.uwaterloo.ca	
Bryan Bo Yan	b2yan@engmail.uwaterloo.ca	

This report is submitted as the final report deliverable for the ECE.492B course. It has been written solely by us and has not been submitted for academic credit before at this or any other academic institution

Consultant:
Dr. Mohamed-Yahia Dabbagh

Date Submitted:
Feb 7th 2011

Abstract

Darts is a game in which players receive varying amounts of points for placing darts in different regions of the dartboard. Points may be doubled or tripled if the dart lands in special areas. Points from multiple shots are summed and added to a player's score. Consequently, scoring is arithmetically complex and difficult without a pen, paper, and calculator. Currently, electronic dartboards exist that can automatically score a game. However, these systems have limitations. To accommodate the electronic sensors and components in the board, they are built differently than ordinary dartboards. Thus, electronic dartboards do not have the same touch and feel as ordinary dartboards, which makes them less appealing to users. Therefore, we propose a vision--based automatic scoring system which can be used with an ordinary dartboard. The system has a camera which inspects the board. The visual data from the camera is analyzed using computer vision techniques. The placement of the dart is identified, and the corresponding score is calculated, recorded, and displayed. The system simplifies the bookkeeping aspect of the game, without sacrificing game quality. An optional advanced feature is the ability for the system to statistically analyze the placements of a player's shots and provide feedback.

Acknowledgements

We wish to thank our project advisor Dr. Mohamed-Yahia Dabbagh for his guidance and support throughout the course of the project. We would also like to thank the ECE FYDP inventory for providing us with the cameras used in our project.

Glossary

- **Background estimate**
  - An estimate of the background in an image. The background estimate is usually obtained from the **moving average** of an image
- **Background subtraction**
  - A technique used to obtain the foreground objects (i.e. objects of interest) in an image frame by subtracting the **background estimate** from the frame. Typically, the subtraction result is then **thresholded** to filter out noise
- **Game facilitation**
  - A feature so that while playing a game, the system use the rules of the game and the score to determine what should happen next (i.e. player wins, player should throw another dart, player switches, etc.)
- **Haar object detection**
  - An object detection technique which looks for certain edge, line, and center-surround features characteristic of the object we're looking for [1]; we need to train the object detector to look for the correct features
- **Hough transform**
  - A feature extraction technique used to find instances of lines or shapes (such as circles or ellipses) in an image [2]
- **K-means clustering**
  - A technique to partition an image into *k* clusters, with each cluster sharing some similar property; often used in image **segmentation** [2]
- **Motion history image (mhi)**
  - A data structure which stores the motion history of an image; it is defined as:

$$\texttt{mhi}(x,y) = \begin{cases} \texttt{timestamp} & \text{if } \texttt{silhouette}(x,y) \neq 0 \\ 0 & \text{if } \texttt{silhouette}(x,y) = 0 \text{ and } \texttt{mhi} < (\texttt{timestamp} - \texttt{duration}) \\ \texttt{mhi}(x,y) & \text{otherwise} \end{cases}$$

where silhouette is a 2-dimensional mask that has non-zero pixels where the motion occurs. Basically, MHI pixels where motion occurs are set to the current timestamp, while the pixels where motion happened far ago are cleared [3]

- **Moving average**
    - An average of the pixel values from the past *N* frames. Each of the *N* frames may be weighted differently in the calculation of the average. The moving average is typically used to create a **background estimate** [2]
- **OpenCV**
    - An open source library of programming functions for real time computer vision [4]
- **Perspective transform**
    - A method of mapping three-dimensional points from a certain perspective onto a two-dimensional plane; performed as a matrix multiplication operation [2][5]
- **Segmentation**
    - A general term which refers to the process of dividing an image into different segments; for our purposes, the we usually refer to the partitioning of an image of the dart board into the different scoring regions [2]
- **Thresholding**
    - The process of obtaining a binary image from a grayscale image by applying a function to the original image such that if the original image pixel value is above/below some specified value, then the output image pixel will have value 1; otherwise the output image pixel has value 0. Used for filtering out pixels with too small or too large values [6]
- **wxPython**
    - A GUI toolkit for the Python programming language [7]

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Project Specifications

1.1.1 Background

When playing with darts, there are three types of dartboards that are used. The first type of dartboard is made of sisal fibre. These are durable and are the most common type of dartboards when playing in competitions. The second type of dartboard is the electronic dartboard. The dartboard itself is made up of holes which allow a special type of soft-tip dart to enter. The scoring is based on which hole the tip enters. Lastly, there are also magnetic dartboards, which attract magnetic darts. These are usually cheaper and are less accurate, since the dart tips have a rather large surface area and the board is a simple mat.

Currently, electronic dartboards are expensive and require a different type of dart to use. The board itself is also very different, since the darts have to land in one of the holes of the board and it can create an artificial feeling, since the holes guide the dart when it reaches the board. The goal of our project is to allow a regular bristle dartboard to be scored electronically via the use of a camera system. This system will be easy to use and would not require any special hardware so that it can appeal to all kinds of users. By using this system, the user will be able to have the benefits of an electronic system, such as being able to calculate the scores of a game, play against an imaginary opponent, and save statistics, while not having to buy any new darts or a new dartboard.

1.1.2 Users

There are many different types of users that will be using the system. Table 1 below lists out the various types of users considered for our system, what the background of each user has and their needs in terms of using the system.

Table 1: Type of users using the product

Type of User	Background	Needs
Professional Dart Players	Have years of experience playing darts and competing in tournaments.	Needs to have accurate automatic scoring since many of the darts will be cluttered together. May want to have a high level AI opponent and practice routines to allow them to work on their skills. Having statistics would allow them to see what weaknesses they have.
Casual Players	Have little experience playing darts and would like to improve their game.	Needs to have automatic scoring so that it saves the user time when they play. May need an AI opponent that is around their level so that they can continue to improve. Practice routines and statistics could be used to improve their throwing.
Sports Bar Customers	Have various experience playing darts and playing mostly for fun.	Needs to have automatic scoring and game facilitation.
Darts Tournament Hosts	May or may not have experience playing darts.	Needs to have automatic scoring and game facilitation. Statistics could be used to allow players to review the games.
System Installer	May or may not have experience playing darts but has a good understanding of the system	Needs to be able to calibrate the system to ensure the accuracy of the system.

As noted in the Table 1, there are several types of users for the system. The needs for each type of player changes depending on the level the users can perform. For instance, a professional darts player will be able to group all three of their darts in a very tight space, so they would need to have a more accurate scoring as opposed to an average player who will often miss. Certain types of users will also not require as many features. For example, if you are playing in a sports bar, it is assumed that you would have someone to play with and hence, won't require an AI opponent. The tournament

host and system installer will not actually be playing with the system, but more to set up the system.

1.1.3 Functional Requirements

The main function of our system is to locate darts thrown on the dartboard and calculate the score. There are other functions that the system could have, which would strengthen the overall system. This includes features like statistics tracking, an automatically generated opponent, and a way to play through a dart game without having to record anything outside of the system. The functional requirements for our system are listed below in Table 2. The table includes the requirement, how it will be measured, the description of the requirement and the priority. The priority is either a Must or a Should. Any requirement with a priority of Must needs to be implemented on the system, while the Should requirements are optional and serve as a “nice-to-have” feature.

Table 2: Functional requirements

Requirement Name	Measure	Description	Priority
Video / Image Processing	Throw N darts and determine how many times, out of N, the system can detect the darts	Process the image of the dartboard and determine if there are any darts on the board.	Must
Dartboard Calibration	Time it takes to calibrate the system	The user must be able to calibrate the system to the dartboard in a efficient manner when setting up the board.	Must
Dart Location Detection	Throw N darts and determine how many times, out of N, the system can correctly locate	If there are darts detected, locate the dart on the dartboard and determine where the tip is in order to score the dart.	Must

	the darts		
Error Detection	By design	The user must be able to correct any mislocated darts in a easy manner.	Should
Score Calculation	Compare the calculated score to the score calculated by hand	After determining the tip of the dart, calculate the score and automatically add up the score of the other throws.	Must
Exceptional Case Handling	By design	The system should be able to handle exception cases, such as when the dart hits the board and bounces back, or when the dart does not hit the board.	Should
Game Facilitation	By design	If playing a game, use the rules of the game and the score of the darts to determine the next step.	Should
Statistic Tracking	By design	The system should be able to record the darts thrown and calculate where it lands in comparison to a target.	Should
Artificial Opponent	By design	The system should be able to use the statistics taken and generate an opponent that is roughly equally as skilled as the player. The difficulty should be flexible, allowing the user to change it manually.	Should

The *Must* requirements mostly relate to the actual functionality of the system. Many of these requirements are necessary in order for the system to properly locate the dart on the board and determine what the score is. Most of the extra features, such as an AI opponent, game facilitation and statistics for throws are treated as not necessarily needed in order to implement the system properly. These requirements enhance the overall system, but are only considered secondary compared to the main function of the

system.

1.1.4 Non-Functional Requirements

In addition to the main functional requirements, there are also non-functional requirements. There are certain factors like portability which cannot be measured, but is an important aspect to our system. Table 3 lists the various non-functional requirements and their priority.

Table 3: Non-functional requirements

Requirement Name	Description	Priority
Universality	The system must be able to be implemented on a regular dart board, without any changes to it.	Must
Minimal Interference	The system should not be in the way of the player or the board. There should be no difference to the player whether the system is in place or not	Should
Portability	The system must be easy to move around and can be switched from dartboard to dartboard	Should
Flexibility	The system must be able to adjust to different cameras and computer systems.	Should

There are several non-functional requirements for the darts system. The most important is that our system needs to work with any regular dartboard without having to change anything on it. This is one of the design goals of our system, so it is a *Must* for the project. The aim is to be able to use the system with any existing dartboard, with minimal equipment as well. The other non-functional requirements serve to guide the design of our system so that it allows for a better overall system.

1.1.5 Cost

This section will describe the cost of our system in terms of building a prototype and mass production. Table 4 outlines all the cost of producing a prototype and mass production.

Table 4: Cost of the system

	Prototype	Massed Produced
Quantity	1	~10000
Estimated Cost Per Unit	~$200	~$100

The prototype will include the cost of a dartboard, since it will be required for showing off the system. The main cost of the system comes from the webcam, as the system is mostly software which analyzes input from a camera. When mass produced, the cost is without the dartboard, since our system is designed to be used on any existing dartboard. The majority of the cost is also due to the production cost of a good quality video camera.

1.1.6 Block Diagram

A block diagram of our system is provided in Figure 1 below. It provides an overall system architecture view, showing how the hardware are linked to our intended software design. Also, it breaks our design algorithms into several independent components that we can implement in parallel.

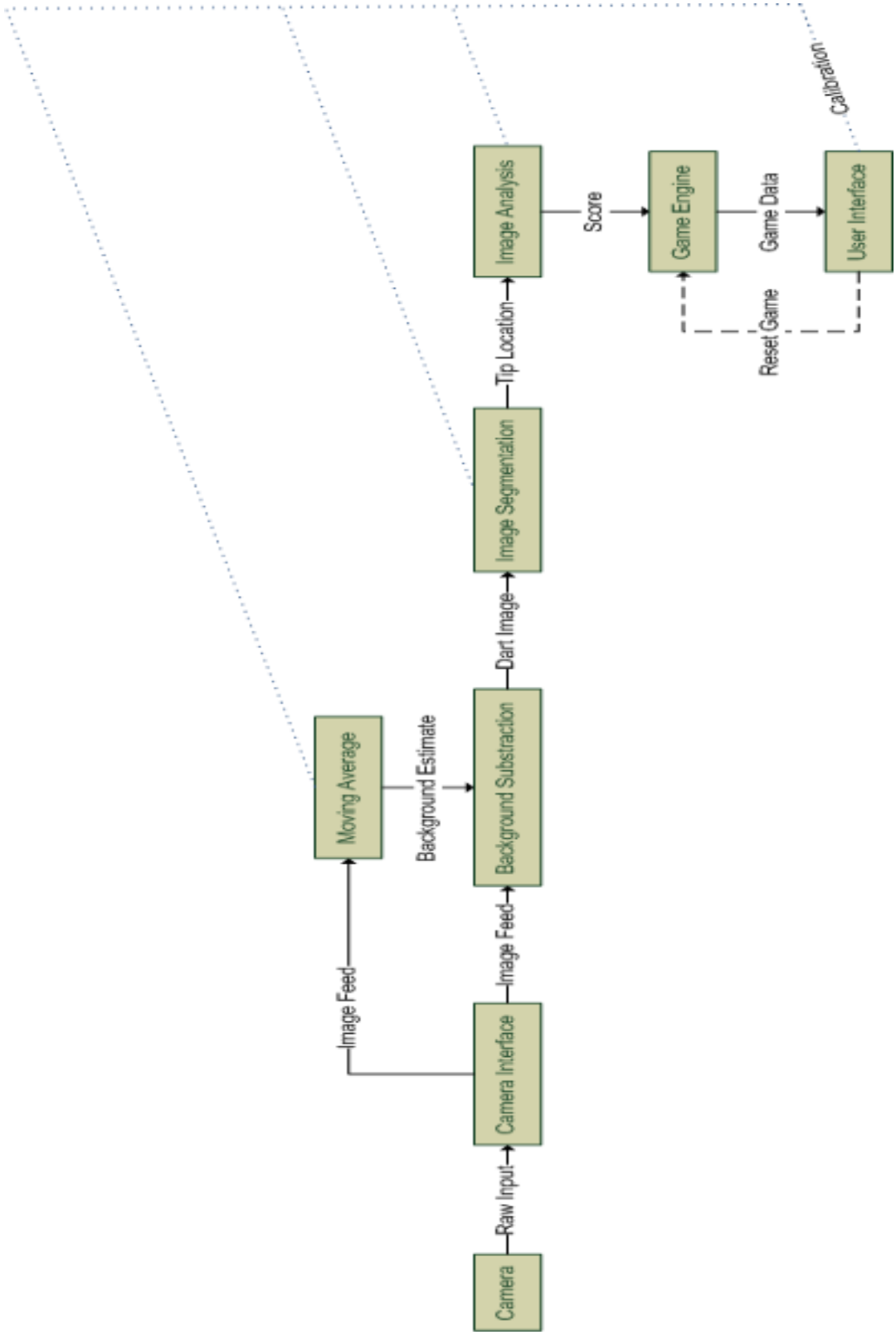

Figure 1: Block diagram of the system

The block diagram shows that the system takes a raw input from a camera and through several image processing techniques, produces some sort of image analysis as a product. After the image analysis, the system is able to start calculating the score. The game engine then takes the score, and based on control information, various actions are performed and then produce an output to the user via a user interface. All of the image analysis and processing techniques will require some sort of calibration to adjust for the differences in lighting and certain subtleties of different dartboards.

1.1.7 Risks

With designing and creating our system, there will be various risks involved. Table 5 lists out the various risks and the response our teamerill ta e when we encounter these problems. The level describes how serious the risk is to the team and the overall project.

Table 5: Risks for the project

Description	Response	Level
The team cannot find an appropriate algorithm for detecting the darts.	Consult a professional with prior experience in image / video processing such as the consultant.	High
The team does not have enough time to realize a suitable algorithm for image / video processing.	Read books recommended by the project consultant or other professionals, and also discuss ideas with them.	Medium
There is a large amount of error in locating the dart on the board.	Consult a professional in order to improve the location algorithm and possibly adjust the way the board is calibrated.	Medium
There is not enough time to implement the system.	Minimize the amount of extra features that our system has.	Low

The risks for this project are mostly related to the main functions of the system. This includes errors in the designing the algorithm, or if we encounter a lot of errors after implementing the algorithm. Most of the time, we will need to learn more from a professional who has more experience than the group.

1.1.8 Plan

The schedule of our project is shown in a Gantt chart format in Figure 2.

	May				June					July				Hours				
Tasks	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**JP**	**CL**	**KW**	**BY**	
Get Materials	+	...	...	...	...	...	...	...	...	...	...	...	...			3		3
Capture Image from Camera	+	+	...	...	...	...	...	...	...	...	...	...	...	5	5			10
Verification Analysis	...	d	...	...	...	...	...	...	...	...	...	...	...	5	5	5	5	20
Dartboard Recognition	...	...	+	+	...	...	...	...	...	...	...	...	...		10		10	20
Dartboard Sectional Division	...	...	...	+	+	...	...	...	...	...	...	...	...	20	20	20	20	80
Detailed Design	...	...	...	...	d	...	...	...	...	...	...	...	...	5	5	5	5	20
Dart Detection	...	...	...	...	...	+	+	...	...	...	...	...	...	10	10	10		30
Dart Scoring	...	...	...	...	...	+	+	...	...	...	...	...	...				10	10
Fine Tuning Detection	...	...	...	...	...	...	...	+	+	...	...	...	...	15		15		30
Designing UI	...	...	...	...	...	...	...	+	+	...	...	...	...		5		5	10
Coding UI	...	...	...	...	...	...	...	+	+	...	...	...	...		5		5	10
Implementing Game Logic	...	...	...	...	...	...	...	...	+	...	...	...	...		10		10	20
Statistic Recording	...	...	...	...	...	...	...	...	+	+	...	...	...	15				15
AI Opponent Coding	...	...	...	...	...	...	...	...	+	+	...	...	...			12		12
Exception Handling	...	...	...	...	...	...	...	...	...	+	...	...	...			5	5	10
Test and Debug Prototype	...	...	...	...	...	...	...	...	...	+	+	...	...	10	10	10	10	40
Prototype Testing Checklist	...	...	...	...	...	...	...	...	...	...	d	...	...	1	1	1	1	4
Debug Prototype	...	...	...	...	...	...	...	...	...	...	...	+	...	8	8	8	8	32
Prototype Demonstration	...	...	...	...	...	...	...	...	...	...	...	d	...	2	2	2	2	8
Experience Report	...	...	...	...	...	...	...	...	...	...	...	...	d	2	2	2	2	8
Journals & Logs	...	...	...	...	...	...	...	...	...	...	...	...	d	1	1	1	1	4
Write draft final report	...	...	...	...	...	...	...	...	...	...	...	...	+	4	4	4	4	16
	Legend:	...	Wait Time											103	103	103	103	412
		+	Active Time															
		d	Deliverable															

Figure 2: Team schedule

The team will aim to even out the amount of work each member does, while at the same time making the group as effective as possible. Most of the work will be done in the summer term from May to July. The schedule will be a rough outline of what will happen, as members might require extra help from other members. There will also be unexpected meetings when an unexpected event happens.

1.1.9 Budget

With our system, the group will need various types of materials to build a prototype. Table 6 summarizes the various equipment required for the prototype, and the source of these materials.

Table 6: Materials needed for the system

Item	Description	Source
Prototype Material		
Dartboard	A regular sisal-fibre dartboard	Hobby shop or a sponsor
Darts	A set of brass or tungsten darts	Hobby shop or a sponsor
Camera	A webcam with OpenCV	Computer store or the ECE project inventory
Computer	A computer to run the software	One of the students
Other Material		
Open source software	OpenCV, wxPython	Internet
Services		
Poster	Symposium Poster	UW Graphics

As the prototype did not require too much equipment, the group was able to obtain

everything very quickly. The dartboard and darts were purchased from a hobby store. One of the main components required was a webcam that is compatible with OpenCV, which was obtained from the ECE inventory (Logitech QuickCam Pro 4000). Since each group member has a laptop as well, the group decided to just use one of the laptops, instead of purchasing some sort of system. We also obtained the open source libraries we needed from the Internet.

There is a cost associated with each item, and Table 7 lists out the price of the items the team will need to purchase as well as where the cost will come from.

Table 7: Budget of the system

Item	Price	Source
Dartboard	$100	Team
Darts (x3)	$50	Team
Computer	$500	Team
Poster	$200	Team
Team Total	$850	
Camera	$200	ECE Project Inventory
ECE Project Inventory Total	$200	

Some of the cost was actually not required, as the team already had the computer. The items that were purchased were the dartboard and darts. The camera was borrowed from the ECE Project Inventory as well.

Chapter 2. High-Level Analysis

In this section we describe how we decompose our system into different blocks (subsystems), and then we list, for each subsystem, the constraints that each subsystem must satisfy. For each constraint, we describe what the constraint entails, its priority level {MUST/SHOULD}, and how we verified the feasibility of the constraint. Many higher-level constraint feasibilities are expressed as a conjunction of lower-level constraint feasibilities.

2.1 Overall System

Constraints

- overall_system.1 [MUST]
    - The overall system must receive visual input from a game of darts, inputs from users, and compute the scores of the players involved
        - **VERIFICATION**: If camera.1, image_processing.1, game_engine.1, game_engine.5, and user_interface.1 are satisfied, then this constraint will also be satisfied
- overall_system.2 [MUST]
    - The system must be able to work with most regular dartboards/darts, without changes to the dartboard/darts themselves
        - **VERIFICATION**: If image_processing.2 is satisfied, then this constraint will also be satisfied
- overall_system.3 [MUST]
    - The system must work with most webcams and most computer systems (i.e. PC, laptop, netbooks. etc.) without needing additional hardware
        - **VERIFICATION**: If camera.2 and image_processing.3 are satisfied, then this constraint will also be satisfied
- overall_system.4 [SHOULD]
    - The system should produce the correct score

        - **VERIFICATION**: If image_processing.4 is satisfied, then this constraint will also be satisfied
- overall_system.5 [MUST]
    - The user must be able to correct the score if the system miscalculates it
        - **VERIFICATION**: If game_engine.7 and user_interface.2 are satisfied, then this constraint will also be satisfied
- overall_system.6 [SHOULD]
    - The system should guide the user through the game by knowing the rules of the game and ensuring that they have been followed (i.e. do not allow players to throw out of order)
        - **VERIFICATION:** game_engine.3, user_interface.3 are satisfied, then this constraint will be satisfied
- overall_system.7 [SHOULD]
    - The system should be able to record the darts thrown, calculate where it lands in comparison to a target, then perform statistical performance on the data
        - **VERIFICATION:** if game_engine.6 and user_interface.4 are satisfied, then this constraint will be satisfied
- overall_system.8 [SHOULD]
    - The system should be able to use the statistics taken and generate an opponent is that is roughly equally as skilled as the player. The difficulty should be flexible, allowing the user to change it manually
        - **VERIFICATION**: If game_engine.8 and user_interface.5 are satisfied, then this constraint will also be satisfied
- overall_system.9 [SHOULD]
    - The system should be robust in varying lighting conditions
        - **VERIFICATION**: If image_processing.5 is satisfied, then this constraint will also be satisfied
- overall_system.10 [SHOULD]
    - The system should be able to be calibrated in an efficient manner when setting up the board

        - **VERIFICATION**: If calibration_unit.1 and user_interface.6 are satisfied, then this constraint will be satisfied
- overall_system.11 [SHOULD]
    - The system should perform the score computations in real-time
        - **VERIFICATION**: if image_processing.7 is satisfied, then this constraint will be satisfied

Block Diagram

The block diagram of the overall system is given in Figure 3.

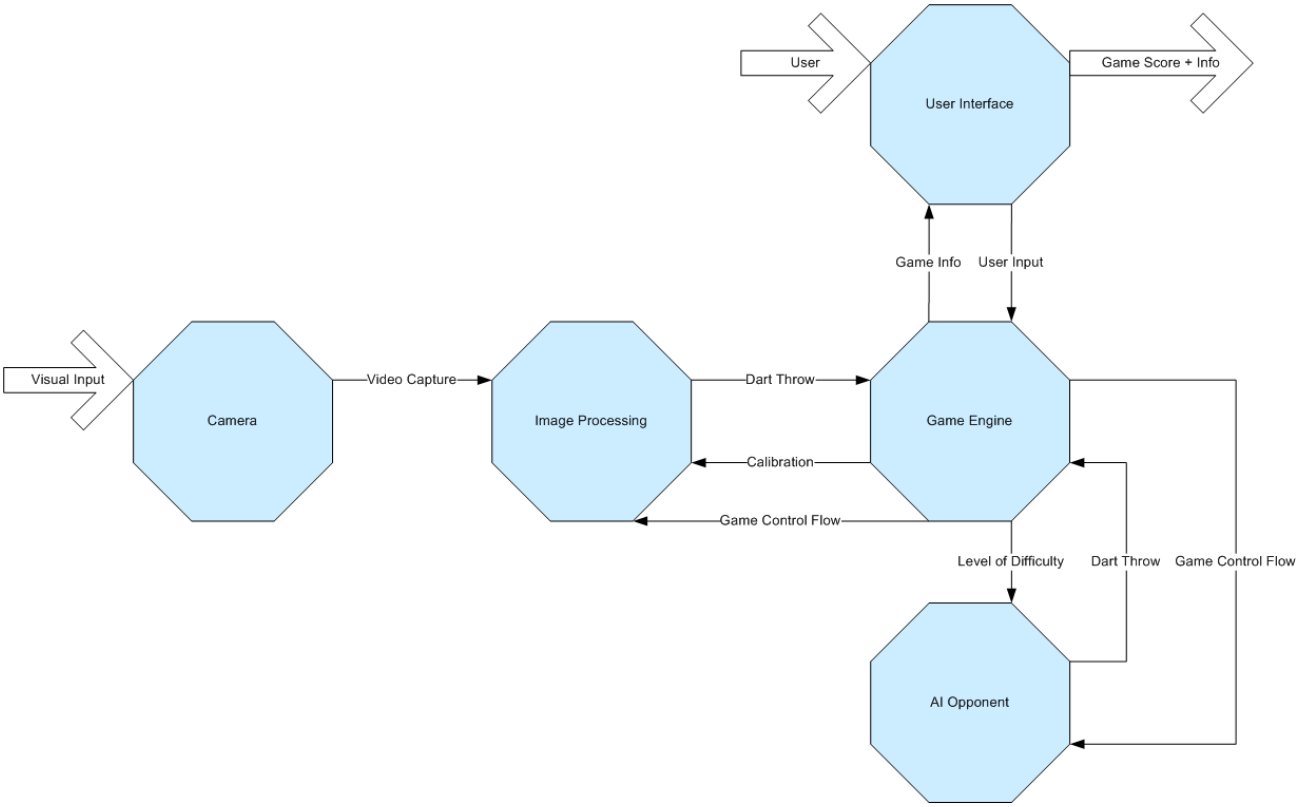

Figure 3: High-level block diagram

2.2 Camera

The **Camera** component takes the visual input from the dart game, and produces a video output for the **Image Processing** component.

Constraints

- camera.1 [MUST]
    - The system must receive visual input from a game of darts, and produce a video output for the **Image Processing** component
        - **VERIFICATION**: We are using OpenCV, an open source library for computer vision. We've tried and successfully, captured video from several cameras (2 built-in cameras on laptops, and a Logitech QuickCam Pro 4000) using OpenCV.
- camera.2 [MUST]
    - The system must work with most webcams
        - **VERIFICATION**: We're using OpenCV, an open source library for computer vision. We've tried and successfully, captured video from several cameras (2 built-in cameras on laptops, and a Logitech QuickCam Pro 4000) using OpenCV.

2.3 Image Processing

The **Image Processing** component takes as inputs video output from the **Camera** component, and game control flow information from the **Game Engine** component. From these inputs, the **Image Processing** component will produce dart throw output information (where the dart landed) for the **Game Engine**. In addition, the **Image Processing** component can calibrate itself with input from the **Game Engine**.

Constraints

- image_processing.1 [MUST]
    - The system must receive video output from the **Camera** component, and game control flow information from the **Game Engine** component. From these inputs, the system must produce dart throw output information (where the dart landed) for the **Game Engine**
        - **VERIFICATION:** if hough_ellipse_detection.1, moving_average.1, subtract_threshold.1, feature_selection.1, segmentation.1, and compute_dart_throw_result.1 are satisfied, then this constraint will

also be satisfied

- image_processing.2 [MUST]
    - The system must be able to work on most regular dartboards/darts, without changes to the dartboard/darts
        - **VERIFICATION**: if hough_ellipse_detection.2, moving_average.2, subtract_threshold.2, feature_selection.2, and segmentation.2 are satisfied, then this constraint will also be satisfied
- image_processing.3 [MUST]
    - The system must work with most computer systems (i.e. PC, laptop, netbooks. etc.)
        - **VERIFICATION**: We are using OpenCV, an open source library for computer vision, which works on both Windows and Linux.
- image_processing.4 [SHOULD]
    - The image processing performed by the system should be correct
        - **VERIFICATION**: if hough_ellipse_detection.3, moving_average.3, subtract_threshold.3, feature_selection.3, segmentation.3, and compute_dart_throw_result.2 are satisfied, then this constraint will also be satisfied
- image_processing.5 [SHOULD]
    - The system should be robust in varying lighting conditions
        - **VERIFICATION:** if image_processing.6 is met, then the system can be quickly calibrated in varying lighting conditions
- image_processing.6 [SHOULD]
    - The system should be able to be calibrated in an efficient manner when setting up the board
        - **VERIFICATION**: OpenCV has built-in features for camera calibration
- image_processing.7 [SHOULD]
    - The system should perform computations in real-time
        - **VERIFICATION**: if hough_ellipse_detection.4, moving_average.4, subtract_threshold.4, feature_selection.4, segmentation.4, and

compute_dart_throw_result.3 are satisfied, then this constraint will also be satisfied

Block Diagram

The block diagram for the image processing subsystem is given in Figure 4.

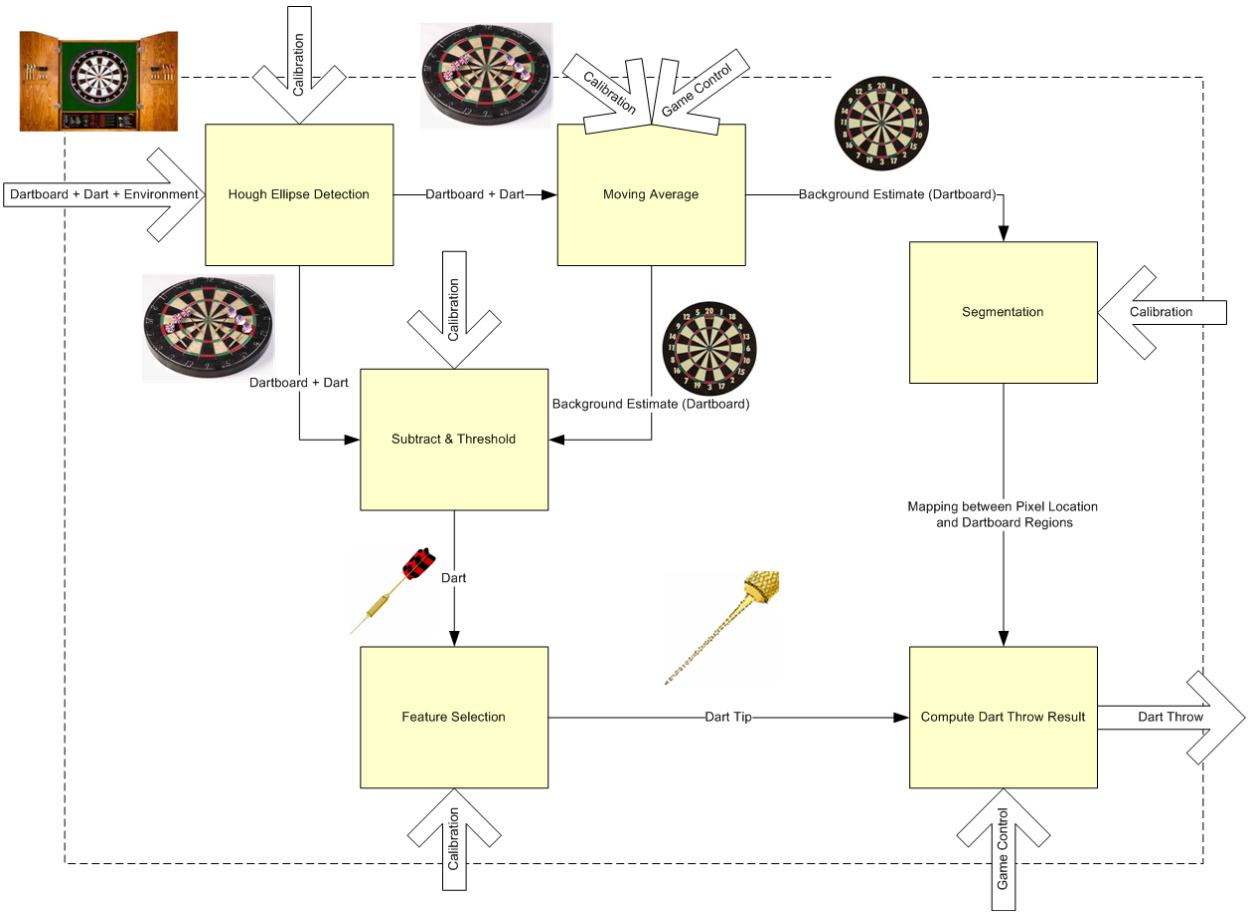

Figure 4: Image processing block diagram

2.3.1 Hough Ellipse Detection

The **Hough Ellipse Detection** component uses the technique of the Hough Transform to fit an ellipse (including circles). It takes as input: a video feed (from the **Camera**) which contains the dartboard, the darts, and other surrounding objects. It produces a truncated video output which contains only the dartboard and the darts. This output goes to the **Moving Average** and the **Substract & Threshold** components.

Constraints

- hough_ellipse_detection.1 [MUST]
    - The system must receive video output from the **Camera** component(containing the dartboard, the darts, and surrounding objects) and produce a truncated video output which contains only the dartboard and the darts for the **Moving Average** and the **Substract & Threshold** components
        - **VERIFICATION:** an example application in OpenCV can detect ellipses in an image
- hough_ellipse_detection.2 [MUST]
    - The system must be able to work on most regular dartboards/darts, without changes to the dartboard/darts
        - **VERIFICATION:** We tried the example ellipse detection application in OpenCV on printouts of dartboards, and the application was able to successfully detect the outline of the board
- hough_ellipse_detection.3 [SHOULD]
    - The image processing performed by the system should be correct
        - **VERIFICATION:** We tried the example ellipse detection application in OpenCV, and the application was able to successfully detect the outline of a dartboard
- hough_ellipse_detection.4 [SHOULD]
    - The system should perform computations in real-time
        - **VERIFICATION:** We tried the example ellipse detection application in OpenCV, and the performance was indistinguishable from real-time

2.3.2 Moving Average

The **Moving Average** component estimates the value of a particular background pixel as a weighted average of previous values. Typically, pixels in the distant past should be weighted at zero, and the weights increase smoothly from the distant past to current time. **Moving Average** takes as inputs: the truncated video which contains only the

dartboard and the darts from the **Hough Ellipse Detection** component, and produces a background estimate (which should be only the dartboard) for the **Substract & Threshold** and **Segmentation** components. **Moving average** takes the game control flow as input from the Game Engine and adjusts its weightings accordingly based on the current game situation. Moving Average can also calibrate itself based on input from the **Game Engine**.

Constraints

- moving_average.1 [MUST]
    - The system must receive as input: the truncated video from the **Hough Ellipse Detection** component(containing only the dartboard, and the darts) and produce a background estimate (which should be only the dartboard) for the **Substract & Threshold** and **Segmentation** components
        - **VERIFICATION:** Harpia, a software which allows one to build image processing systems (and is based on OpenCV), allows one to sum and average images.
- moving_average.2 [MUST]
    - The system must be able to work on most regular dartboards/darts, without changes to the dartboard/darts
- moving_average.3 [SHOULD]
    - The image processing performed by the system should be correct
- moving_average.4 [SHOULD]
    - The system should perform computations in real-time

2.3.3 Subtract & Threshold

The **Subtract & Threshold** component takes the image of the dartboard and darts from the **Hough Ellipse Detection** component, and substracts from it the background estimate (which should be only the dartboard) from the **Moving Average** component, yielding only the image of the dart, which is then passed on to the **Feature Selection** component.

Constraints

- subtract_threshold.1 [MUST]
    - The system must take the image of the dartboard and darts from the **Hough Ellipse Detection** component(input), and substract from it the background estimate (which should be only the dartboard) from the **Moving Average** component(input), yielding only the image of the dart (output), which is then passed on to the **Feature Selection** component.
        - **VERIFICATION:** Harpia, a software which allows one to build image processing systems (and is based on OpenCV), allows one to subtract images
- subtract_threshold.2 [MUST]
    - The system must be able to work on most regular dartboards/darts, without changes to the dartboard/darts
- subtract_threshold.3 [SHOULD]
    - The image processing performed by the system should be correct
- subtract_threshold.4 [SHOULD]
    - The system should perform computations in real-time

2.3.4 Feature Selection

The **Feature Selection** component takes as input the image of the dart (from the **Subtract & Threshold** component), and selects only the tip. **Feature Selection** produces the pixel coordinate of the very end of the tip as output to **Compute Dart Throw Result**.

Constraints

- feature_selection.1 [MUST]
    - The system must take as input the image of the dart (from the **Subtract & Threshold** component), and selects only the tip. The system must produce the pixel coordinate of the very end of the tip as output to **Compute Dart Throw Result**.

        - **VERIFICATION:** A feature selection example application is found in OpenCV and does something similar to what we need
- feature_selection.2 [MUST]
    - The system must be able to work on most regular dartboards/darts, without changes to the dartboard/darts
        - **VERIFICATION:** A feature selection example application is found in OpenCV and seems to work with different objects
- feature_selection.3 [SHOULD]
    - The image processing performed by the system should be correct
        - **VERIFICATION:** A feature selection example application is found in OpenCV and seems to work reasonably well
- feature_selection.4 [SHOULD]
    - The system should perform computations in real-time
        - **VERIFICATION:** A feature selection example application is found in OpenCV and gives near real-time performance

2.3.5 Segmentation

The **Segmentation** component takes as input: the image of the board from **Moving Average** and, using some segmentation technique (i.e. k-means clustering), segments the image of the board into the appropriate regions, and produce as an output (for **Compute Dart Throw Result**) a mapping between pixel coordinates and dartboard regions.

Constraints

- segmentation.1 [MUST]
    - The system must take as input the image of the board from **Moving Average** and, using some segmentation technique (i.e. k-means clustering), segment the image of the board into the appropriate regions, and produce as an output (for **Compute Dart Throw Result**) a mapping between pixel coordinates and dartboard regions.
        - **VERIFICATION:** Several segmentation example applications,

including one using k-means clustering, is found in OpenCV and work well

- segmentation.2 [MUST]
    - The system must be able to work on most regular dartboards/darts, without changes to the dartboard/darts
        - **VERIFICATION:** Several segmentation example applications are found in OpenCV and work well with images of various dartboards
- segmentation.3 [SHOULD]
    - The image processing performed by the system should be correct
        - **VERIFICATION:** Several segmentation example applications are found in OpenCV and produce the correct result
- segmentation.4 [SHOULD]
    - The system should perform computations in real-time
        - **VERIFICATION:** Several segmentation example applications are found in OpenCV and give real-time performance

2.3.6 Compute Dart Throw Result

The **Compute Dart Throw Result** component takes as input the location of the dart tip (from **Feature Selection**), along with the mapping between pixel coordinates and dartboard regions from **Segmentation**. It also takes as input the game flow information from the **Game Engine**. If the game is expecting a throw from the user, then **Compute Dart Throw Result** simply looks up what region the end tip pixel is in, and returns the dart throw information to **Game Engine**. However, if the game is not expecting a throw from the user, then **Compute Dart Throw Result** returns null.

Constraints

- compute_dart_throw_result.1 [MUST]
    - The system must take as input the location of the dart tip (from **Feature Selection**), along with the mapping between pixel coordinates and dartboard regions (from **Segmentation)**. It also takes as input the game flow information from the **Game Engine**. If the game is expecting a throw

from the user, then **Compute Dart Throw Result** simply looks up what region the end tip pixel is in, and returns the dart throw information to **Game Engine**. However, if the game is not expecting a throw from the user, then **Compute Dart Throw Result** returns null.

        - **VERIFICATION:** By design

- compute_dart_throw_result.2 [SHOULD]
    - The image processing performed by the system should be correct
        - **VERIFICATION:** If segmentation.3 and feature_selection.3 are satisfied, then this constraint will also be satisfied
- compute_dart_throw_result.3 [SHOULD]
    - The system should perform computations in real-time

2.4 AI Opponent

The **AI Opponent** component is responsible for generating dart throws in order to allow a single user to play competitively. The dart throws are generated based on the difficulty set by the user through the **User Interface** and passed from the **Game Engine**. First, a target is selected depending on the type of game being played as well as the current situation. Then, a dart throw is randomly generated with various accuracy and precision depending on the level of difficulty. The generated dart throw is passed through to the **Game Engine** and is treated as a regular dart throw.

Constraints

- AI_opponent.1 [MUST]
    - The AI Opponent must generate a dart throw which differs in accuracy and precision depending on the difficulty set
        - **VERIFICATION**: If throw_simulation.1 is satisfied, then this constraint will also be satisfied
- AI_opponent.2 [MUST]
    - The AI Opponent should select a proper target depending on the type of game being played and the progress of the game

        - **VERIFICATION:** If target_selection.1 is satisfied, then this constraint will also be satisfied

Block Diagram

The block diagram for the AI subsystem is given in Figure 5.

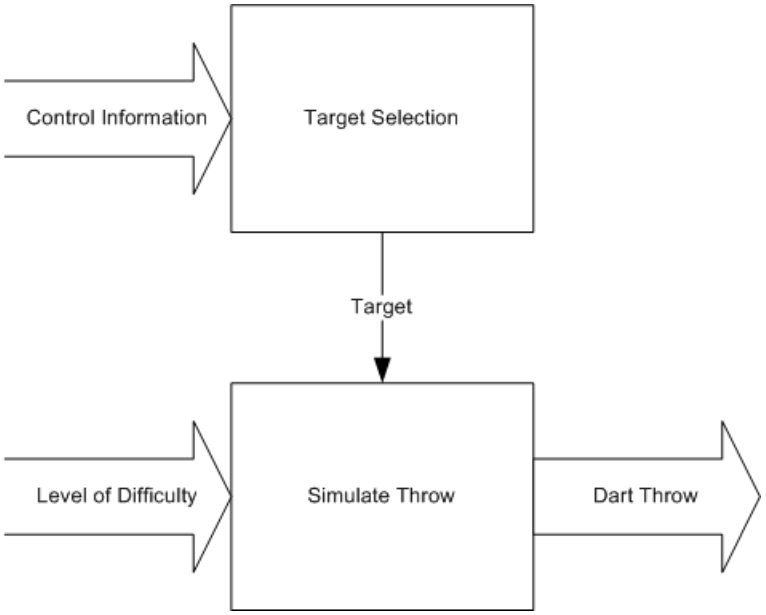

Figure 5: AI block diagram

2.4.1 Target Selection

The **Target Selection** component takes *game control flow* information about the dart game from the **Game Engine** and selects an appropriate target. The information includes the type of game being played and the current scores of the players. The **Target Selection** uses the goals of the game in order to determine the correct target to aim at.

Constraints

- target_selection.1 [MUST]
    - A target is selected for the AI Opponent to use
        - **VERIFICATION**: The AI logic can be programmed to use the control information to determine a target for the next dart

- target_selection.2 [SHOULD]
    - The target selected should be the best target to be used

2.4.2 Throw Simulation

The **Throw Simulation** component uses the target generated by the **Target Selection** component and the *Level of Difficulty* generated by the **Game Engine** in order to simulate a dart toss. On an easier difficulty, the throws will be less accurate and less precise, while on a higher difficulty, the throws will be more accurate and more precise.

Constraints

- throw_simulation.1 [MUST]
    - A Dart Throw is generated using the target and level of difficulty
        - **VERIFICATION**: The AI logic can be programmed to randomly generate a dart throw with varying accuracy and precision
- throw_simulation.2 [SHOULD]
    - The level of accuracy and precision for each dart throw is tuned properly for various difficulty levels

2.5 Game Engine

The game engine controls the overall system. The engine control handles the user input and the output to the **User Interface**. Depending on the user input, it also generates a level of difficulty signal that is passed to the **AI Opponent** component. The engine control also sends information about the game to the game logic, which handles the dart throws and produces control information for the AI Component and the **Image Processing** block. After processing the dart throws depending on the type of game being played, Dart Data is passed back to the engine, which outputs it to the user. If the user is in a practice mode, each dart data is also stored in a database and is used to keep statistics for the user. The engine will retrieve information about the user from the database. When necessary, the Game Engine also starts the calibration process for the system.

Constraints

- game_engine.1 [MUST]
    - The game engine must handle the user inputs properly and send the correct information to the user interface
        - **VERIFICATION**: If engine_control.1 and engine_control.2 are satisfied, then this constraint will also be satisfied
- game_engine.2 [MUST]
    - The game engine must be able to generate various levels of difficulty for the AI Opponent
        - **VERIFICATION**: If engine_control.5 is satisfied, then this constraint will also be satisfied
- game_engine.3 [MUST]
    - The game engine should be able to control different types of dart games and guide users through each type of game
        - **VERIFICATION** If game_logic.1 and engine_control.3 are satisfied, then this constraint will also be satisfied
- game_engine.5 [MUST]
    - The game engine must be able to calibrate the dart board
        - **VERIFICATION**: If engine_control.6 and calibration_unit.1 are satisfied, then this constraint will also be satisfied
- game_engine.5 [MUST]
    - The game engine must handle dart throws properly from either the **Image Processing** or **AI Opponent**
        - **VERIFICATION:** If game_logic.2 is satisfied, then this constraint will also be satisfied
- game_engine.6 [SHOULD]
    - The game engine should be able to store the dart throws of various users and use them as statistics
        - **VERIFICATION:** If dart_throw_database.1, dart_throw_database.2, and dart_throw_database.3 are satisfied, then this constraint will be

satisfied

- game_engine.7 [MUST]
    - The game engine must allow the user to manually override the score results in case of an **Image Processing** error
        - **VERIFICATION:** if engine_control.7 and dart_throw_database.4 are satisfied, then this constraint will be satisfied
- game_engine.8 [SHOULD]
    - The game engine should be able to use statistics in order to generate an AI opponent that is roughly equal to the user's level
        - **VERIFICATION**: By design

Block Diagram

The block diagram for the game engine subsystem is given in Figure 6.

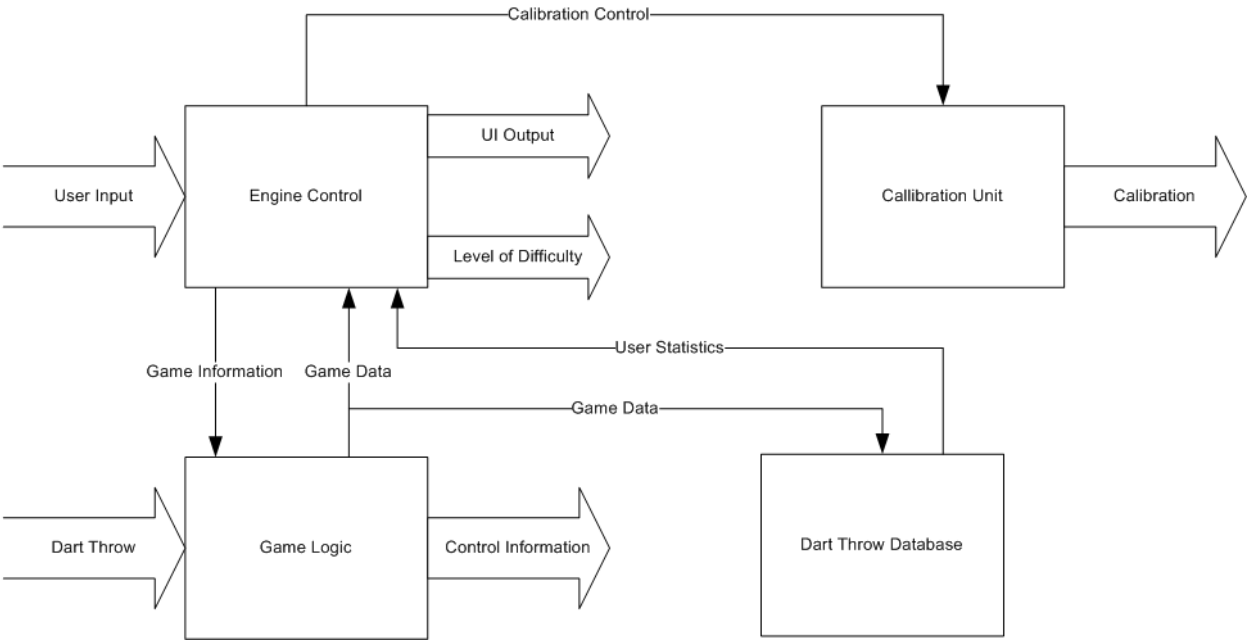

Figure 6: Game engine block diagram

2.5.1 Engine Control

The Engine Control is responsible for controlling the overall functionality of the engine and handles the Game Logic component. This component is mainly responsible for capturing input from and sending information to the User Interface. The engine is also

responsible for setting up the game, and sends the user input to the game logic as *game information*. It also takes *game data* from the current game and *statistics* from the user and outputs it to the UI. The Engine Control will change the difficulty of the AI Opponent depending on the input from the user. This block is also used to start the calibration process by sending a *calibration* signal to the Calibration Unit.

Constraints

- engine_control.1 [MUST]
    - The engine must be able to handle inputs from the user
        - **VERIFICATION**: By design
- engine_control.2 [MUST]
    - The engine must be able to control the output to the user interface
        - **VERIFICATION**: By design
- engine_control.3[MUST]
    - The engine control must send information to the game logic to start a game
        - **VERIFICATION**: The engine control can be programmed to communicate with the game logic
- engine_control.4 [MUST]
    - The engine control must recieve *game data* and *statistics* and generate an appropriate output to the UI.
        - **VERIFICATION**: The engine can be programmed to handle various inputs and generate an output accordingly
- engine_control.5 [MUST]
    - The engine control must be able to set the level of difficulty of the AI opponent
        - **VERIFICATION**: The engine can take the input related to AI difficulty and set the level correctly
- engine_control.6 [MUST]
    - The engine control must be able to start and control the calibration process

        - **VERIFICATION**: By design
- engine_control.7 [MUST]
    - The engine control must allow the user to manually override the score results in case of an **Image Processing** error
        - **VERIFICATION**: By design

2.5.2 Game Logic

The Game Logic component handles the various game types and determines how the system will process each dart throw. The Game Logic will process each *dart throw* and progress the game being played. It will also output a *game control flow* signal to facilitate and control the game being played. It will generate *game data* to be used in recording statistics as well as for the engine to output the updated scores.

Constraints

- game_logic.1 [MUST]
    - A game must be able to run from start to finish
        - **VERIFICATION**: The game logic can be programmed for various game types with differing control signals
- game_logic.2 [MUST]
    - The game logic must use *dart throw* information to progress the game
        - **VERIFICATION**: The game logic can be programmed to use the dart throw to facilitate a game
- game_logic.3 [MUST]
    - The game logic must produce *game control flow* information to control other components
        - **VERIFICATION**: By design
- game_logic.4 [SHOULD]
    - The game logic should have logic to handle different types of games
        - **VERIFICATION**: By design

2.5.3 Calibration Unit

The Calibartion Unit is responsible for starting and controlling the calibration process. It will recieve a start signal from the Engine Control and proceed to calibrate the dart board through the Image Processing component.

Constraints

- calibration_unit.1 [MUST]
    - A calibration must be able to calibrate the dartboard properly
        - **VERIFICATION**: If image_processing.6 is satisfied, then this constraint will also be satisfied

2.5.4 Dart Throw Database

The Dart Throw Database will store *dart throw* data for different users and produce different statistics based on the data. Data is only stored in the database when practice mode is being used, since it will be the only game mode that asks the user for a target to aim at.

Constraints

- dart_throw_database.1 [MUST]
    - Dart throws must be stored for each user
        - **VERIFICATION**: A relational database can be used to store various information about the dart throws for various users
- dart_throw_database.2 [MUST]
    - The user must be able to retrieve data stored in the database
        - **VERIFICATION:** A relational database can be used to retrieve information about the dart throws
- dart_throw_database.3 [SHOULD]
    - Statistics should be generated using the information in the database
- dart_throw_database.4 [MUST]
    - The database must be able to be manually overridden in case of erroneous dart throws

2.6 User interface

The User Interface component takes inputs (menu selections, etc.) from the user and forwards the information to the Game Engine. User Interface also takes as input the game information from the Game Engine and displays that into the the user as output.

Constraints

- user_interface.1 [MUST]
    - The **User Interface** must take inputs (menu selections, etc.) from the user and forwards the information to the **Game Engine**. **User Interface** also must take as input the game information from the **Game Engine** and displays that into the the user as output
        - **VERIFICATION**: By design
- user_interface.2 [MUST]
    - The **User Interface** must allow the user to correct the score if the system miscalculates it
        - **VERIFICATION**: By design
- user_interface.3 [SHOULD]
    - The **User Interface** should guide the user through the game by following the rules of the game and ensuring that they have been followed (i.e. do not allow players to throw out of order)
        - **VERIFICATION**: if game_engine.3 is satisfied, then this constraint will be satisfied
- user_interface.4 [SHOULD]
    - The **User Interface** should allow the user to access the statistics analysis features
        - **VERIFICATION**: By design
- user_interface.5 [SHOULD]
    - The **User Interface** should allow the user to change the difficulty of the AI opponent

        - **VERIFICATION**: By design
- user_interface.6 [SHOULD]
    - The **User Interface** should allow the user to calibrate the system

VERIFICATION: By design

Chapter 3. Detailed Design

In this chapter, we summarize the detailed design of the system.

3.1 Overall Design

The overall system has four distinct components: dartboard calibration, game engine, user interface, and dart detection. The components of the system interact through events so that each component may run on different threads. Figure 7 illustrates the block diagram of the system components. One advantage of this design is that the game engine, and image processing can take place simultaneously as the user interacts with the user interface.

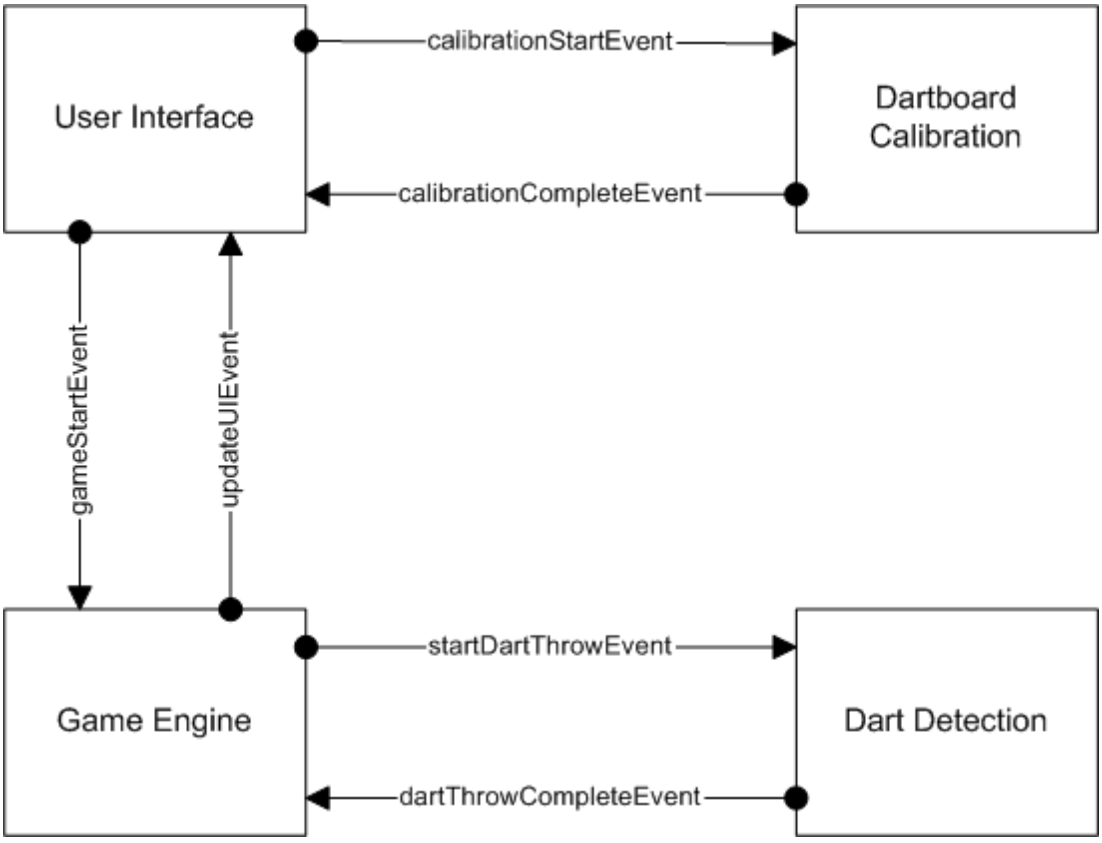

Figure 7: System components block diagram

3.2 Dartboard Calibration

In order for the automatic dart scoring system to know where the dartboard is and how it looks like, a calibration process take place. In this system, calibration means storing the rings and point dividers, so that the dartboard can be sliced into scoring regions when a dart arrives (see Figure 8). All the data are stored in polar coordinates relative to the center of the dartboard, and the rings of the dartboard are stored as an array of dictionaries where the angles are the keys and the magnitudes are the values, and the angles of each scoring region divider are stored as an array.

Figure 8: Regions of the dartboard

The advantage of using polar coordinates is that finding which scoring region the dart landed in is relatively trivial: compare the dart location angle with angle of the scoring region dividers and compare the dart location magnitude with the magnitude of the rings of the dartboard. However, getting the rings and the scoring region dividers is not an

easy task because the web cam is set up at an angle from the left side of the dartboard, and the distance between the web cam and the dartboard is not constant. Therefore, manual calibration is required for accurate dart scoring.

3.2.1 Calibration Function

The calibration function is called by the game engine through the calibrationStartEvent. When the calibration is complete, a calibrationCompleteEvent fires. The calibration definition and the manual calibration steps are listed below.

Pseudocode:

```
def CalibrateDartboard()
while ( systemOn )
     # wait until calibrationStartEvent is triggered
     wait ( calibrationStartEvent )
     # clear the calibrationStartEvent
     clear( calibrationStartEvent )
     ...
```

1. When the calibration function is initiated, the web cam immediately takes an image of the dartboard.

Pseudocode:

```
capture = cvCaptureFromCAM(0)
image = 0
cvGrabFrame(capture)
image = cvRetrieveFrame(capture)
```

2. A screen from OpenCV appears and prompts the user to select the middle of the dartboard (the center of the bull's eye). This is the reference point for our dartboard; all objects on the dartboard use polar coordinates with respect to this reference point. See Figure 9 below.

Figure 9: Reference point of the dartboard

Pseudocode:

```
def refPoint( x, y ):
     # Get the coordinate of the user's click, and return the x
and y
     # coordinates
     getUserClick( userClickX, userClickY )
     x = userClickX
     y = userClickY
```

3. Next, the UI asks the user to pick 4 points for each of the rings of the dartboard using mouse clicks, starting from the outer ring and finishes at the bull's eye, and an array containing the ellipses are created. The ellipses are represented by an array of dictionaries where the angles are the keys and the magnitudes are the values. There are 7 rings on the dartboard, including the dartboard border. Figure 10 shows the 4 calibration points for the dartboard border (red X's), outermost ring (yellow X's), and the single bull's eye (turquoise X's).

Figure 10: Calibration points required

Pseudocode:

```
# UI pops up with an image of the dartboard

# Create an empty array and dictionary of ellipses, The array of
# ellipses are represented in Cartesian coordinates, and the
# dictionary of ellipses are represented in polar coordinates.
ellipsesCart = []
ellipsesPolar = []

# Create an empty array of Cartesian points (points = []), used
# for storing the points the user picked
points = []

# Loop 7 times, and build the ellipses representing the rings of
# the dartboard. The first ellipse is the border of the
dartboard,
# and each subsequent ellipse are the rings of the dartboard,
# ordering from the largest ring to the smallest ring (the
bull's
# eye).
for index in range ( 0, 7 ):
```

```
# Starting from the dartboard border, prompts the user to
# pick 4 points that best represent the ring of the
dartboard,
# and those points are adjusted by the reference point and
# appended to the points array.
refPoint( refPointX, refPointY )
getUserClick( userClickX, userClickY )
adjustX = userClickX - refPointX
adjustY = userClickY - refPointY
points.append( {adjustX, adjustY} )

# Use FitEllipse2(points) on the points array, and an
# ellipse will be returned. Append this ellipse to the
array of
# ellipses.
ellipsesCart.append(FitEllipse2(points))

# Convert each of the Cartesian points of the ellipse in to
# polar coordinates and append it to the ellipses
dictionary
# data structure using the angle as the key and the
magnitude as
# the value.
tempAngleArray = []
tempMagArray = []
CartToPolar( ellipsesCart[index][0], ellipsesCart[index]
[1], tempAngleArray, tempMagArray, 1 )
for i in tempAngleArray:
     # dictionaries take strings as keys, so convert the
     angle
     # integer to string
     ellipsesPolar[index][str(tempAngleArray[i])] =
     tempMagArray[i]
```

4. Next, the UI asks the user to pick the lines that divide the scoring regions of the dartboard (select the furthest tip of each line, see Figure 11). Starting from the line that separates the 20 point and the 1 point region, and moving clockwise. There are 20 major scoring regions and 20 lines that divide the scoring regions. Figure 11 illustrates the 20 score region dividers that need to be calibrated. The X's are the points the user needs to click on, and the turquoise X is the first point of calibration. The user moves clockwise and click on the yellow X's in order. The angles of the region dividers are required and stored.

Figure 11: Points required to slice the dartboard into regions; the teal x is the only point required for calibration

Pseudocode:

```
dividerCart = []
dividersAngle = []
for x in range ( 0, 20 ):
     regionDivider = []
     getUserClick( userClickX, userClickY )
     adjustX = userClickX - refPointX
     adjustY = userClickY - refPointY
     dividerCart.append( {adjustX, adjustY} )
CartToPolar ( dividerCart[0], dividerCart[1], dividersAngle,
NULL, 1)
```

5. Once calibration is done the data remain in the variables defined, and a calibrationCompleteEvent is sent to the game engine.

Pseudocode:

```
calibrationCompleteEvent.result = success/failure #need to
```

```
determine # whether calibration was successful
set ( calibrationCompleteEvent )
```

3.3 Finding the Dart's Scoring Region

To find which region the dart landed on the dartboard, the dart's angle (about the middle of the board) is used to determine which base point region it landed in, then its magnitude from the origin is used to determine which multiplier region it landed in. Figure 12 below shows the selected ring and base region if the dart were to land in the double 20 space. Further, there are 3 special regions to consider in determining the dart's scoring region: the bull's eye, the double bull's eye, and missing the dartboard. In total, there are 83 scoring regions (20 different points regions * 4 different rings per point region + bull's eye + double bull's eye + missing the dartboard).

Figure 12: Example translation of dart coordinates into dartboard scoring region

3.3.1 Finding Dart's Scoring Region Function

The dart detection module sends the raw Cartesian coordinates of the tip of the dart to this find dart region function, which returns the scoring region where the dart has

landed. The algorithm for the dart scoring region function is described below.

Pseudocode:

```
def FindDartRegion(x,y)
```

1) First the dart's raw Cartesian coordinates are converted into polar coordinates relative to the center of the board

Pseudocode:

```
adjustedDartX = x - refPointX
adjustedDartY = y - refPointY
dartAngle = 0
dartMag = 0
# Angles are in degrees for easy hashing, 0.1 degree accuracy
CartToPolar ( adjustedDartX, adjustedDartY, dartAngle, dartMag,
1 )
```

2) Next, use the angle of the dart to check which point region the dart belongs to. This is done through a loop that compares the angle of the dart with the angles of the dartboard region dividers. The corresponding point region (1 to 20) is returned depending on which region dividers the dart lies between.

Pseudocode:

```
def DartBasePoint ( angle, angleArray ):
     # Work through algorithm to find the the point region (1 to
20)

     # Have a loop from index = 0 to index = 19 to find the
index
     # that angleArray[index] < angle < angleArray[index+1]

     # The trick is that angleArray[0] may not always be the
smallest
     # angle, so the loop either start from the smallest angle,
or
     # wrap around the array when traversing it.

     # Based on the index, return the point region
```

3) Next, use the magnitude of the dart to check which ring the dart belongs to. This is done through a loop that compares the magnitude of the dart with the magnitude of the dartboard ring magnitudes. The ring magnitude is retrieved by using the dart angle as the key in the ring ellipses dictionary data structure. The corresponding ring (0 to 6) is returned depending on which ring dividers the dart lies in between.

Pseudocode:

```
def DartRing( angle, magnitude, rings ):
     # Work through algorithm to find the ring the dart has
landed in

     # Have a loop from index = 0 to index = 6 to find the index
that
     # rings[index][angle] < magnitude < rings[index+1][angle]

     # Based on the index, return a ring value
```

4) Finally, using the point region value and ring returned, a nested if statement can figure out the scoring region set value. For example,

Pseudocode:

```
if basePoint == 20:
     ...
     # double ring has value of 5
     if ring == 5:
          scoringRegion = 'double 20'
     ...
...
```

5) When the scoring region is found, return the region value.

Pseudocode:

```
return scoringRegion
```

3.4 Game Engine

The game engine will control the flow of a dart game by collecting information from the user and the dart throws. The game engine starts when the system starts and will stay

on until the user shuts down the system. The game engine will wait for a game start event and then gather all of the information regarding game settings sent from the UI. Using this information, the game engine will create a new game to facilitate a game of darts. The game engine will also act as a control between the game and the camera. The game engine will also be the link between the UI and the calibration process.

3.4.1 Main Engine Functions:

1. **Engine Control:** The engine itself will control various aspects of the system and will startup when the system turns on. The engine will control the system and wait for events to start various functions. This will include creating a new game, and starting the calibration process.

```
def mainEngine() :
     # initialize the system state
     systemOn = true
     clear ( calibrationStartEvent )
     clear ( calibrationCompleteEvent )
     clear ( gameStartEvent )
     clear ( updateUIEvent )
     clear ( dartThrowStartEvent )
     clear ( dartThrowCompleteEvent )
     # setup the DartboardCalibration thread
     SetupDartboardCalibration()
     # setup the UI thread
     setupUI()
     # setup the Dart Detection thread
     setupDartDetection()

     while ( systemOn ):
          # the system will basically wait for the UI to
               # signal the gameStartEvent
          wait ( gameStartEvent )
          # reset the event for next time
          clear ( gameStartEvent)
          # when the signal is sent, start a new game of darts
               playGame(gameStartEvent.UISettings )
     # "Clean up"
     cleanUp()
```

2. **Game Control:** The main purpose of the game engine will be to facilitate a game of darts The play game function will take in a settings objected created by the UI

which contains various settings for a game of darts. The engine will initialize a game of darts by creating a *dartGame* object using the settings parameter. The game loop will basically wait for a player to throw, update the score using the result and then switch the thrower until an end game condition is completed. The psuedocode of the above is as follows:

```
def playGame( settings ) :
     dartGame = initializeGame( settings )
     while ( dartGame.stillPlaying ):
          throwResult = playerThrow( dartGame )
          if isSet( correctScoreEvent ):
               if correctScoreEvent.player == playerOne:
                    dartGame.playerOne.score =
                                   correctScoreEvent.score
               else:
                    dartGame.playerTwo.score =
                                   correctScoreEvent.score
               clear ( correctScoreEvent )

          else:
               if dartGame.gameType == practiceMode:
                    dartGame.updateScorePractice( throwResult )
               else:
                    dartGame.updateScoreGame( throwResult )
     endGame()
```

3. **Create Game:** In order for a game to run, the game has to be initialized using the *initializeGame* function. The function will take a settings object as a parameter and create a dart game object. The settings object will include various parameters like the game type, the player information, the number of players and the difficulty of the AI (if applicable). This function may just end up being part of the constructor for the dartGameClass, but is seperate for now to allow for flexibility.

```
def initializeGame( settings ) :
     dartGame = dartGameClass()
     dartGame.gameType = settings.gameType
     if dartGame.gameType == practiceMode:
          dartGame.target = generateTarget()
          # a function that randomly generates a target
     dartGame.numberOfPlayers = settings.numberOfPlayers
     dartGame.playerInformation = settings.playerInformation
     dartGame.difficulty = settings.difficulty
```

```
    dartGame.stillPlaying = true
    dartGame.currentPlayer = playerOne
    return dartGame
```

4. **Facilitate Player Throw:** The engine will facilitate player throws by getting dart information from the other components. Depending on whether or not the current player is the computer player, a dart throw will be generated. If the current player is a human player, it will send a signal to the camera to start the image capture process and will wait for the dart throw complete signal to be returned. The result of the throw will be logged into a database. If the current player is a computer player, the function will generate a computer dart throw depending on the difficulty level.

```
def playerThrow( dartGame ) :
    if dartGame.currentPlayer != computerPlayer:
        set ( startDartThrowEvent )
        wait ( dartThrowCompleteEvent )
        # clear the dartThrowCompleteEvent for next time
        clear ( dartThrowCompleteEvent )
        #gets result from Dart Detection
        dartThrow.region = dartThrowCompleteEvent.region
        dartThrow.thrower = dartGame.currentPlayer
        logDartThrow( dartGame.currentPlayer, dartThrow.region)
        return dartThrow
    else
        dartThrow = generateComputerThrow( dartGame.difficulty
    )
        return dartThrow
```

5. **Update Score:** The engine will update the score and send information to the UI based on the results of the throw. The update score function will also contain the game logic in order to control the game. There will be two function to update the scores depending on the game type.

```
def updateScoreGame( dartThrow ) :
    currentPlayer.score =- score(dartThrow.region)
    # check the score and update results
    # if the score is 0, it means the current thrower wins
    if currentPlayer.score = 0
        winner = currentPlayer
        stillPlaying = false
    # if the score is negative, the thrower busts and the score
        # resets
```

```
     if currentPlayer.score < 0
          currentPlayer.score =+ dartThrow.score
     # refresh the UI with new scores
     updateUIEvent.score = currentPlayer.score
     updateUIEvent.stillPlaying = stillPlaying
     updateUIEvent.dartThrow = dartThrow

     set ( updateUIEvent )
     if switch == true
          switchPlayer()      # switches the current thrower

def updateScorePractice( dartThrow ) :
     angleDiff = target.angle - dartThrow.angle
     magnitudeDiff = target.magnitude - dartThrow.magnitude
     logDifference( currentPlayer, angleDiff, magnitudeDiff )
     updateUIEvent.angleDiff = angleDiff
     updateUIEvent.magnitudeDiff = magnitudeDiff
     set ( updateUIEvent )
     target = generateTarget()
```

3.5 User Interface

The user interface exchanges events with the Game Engine in order to facilitate communication with the end user. The main menu allows the user to select whether he or she would like to play a practice game, a game against another person, a game against a computer player, or to calibrate the system with the board. The main game screen shows the user whose turn it is.

3.5.1 Main UI Functions

1. The setupUI() function is called by the game engine and initiates the user interface with the user. The main menu selection screen is the first window that appears as shown below in Figure 13 and it is initiated via a call to the showMainMenu() function that draws the splash screen and displays it to the user. The return value of the showMainMenu() function is the option that the user selects. The pseudocode for the setupUI() function is given below. After the UI has been setup, an appropriate event is sent back to the Game Engine.

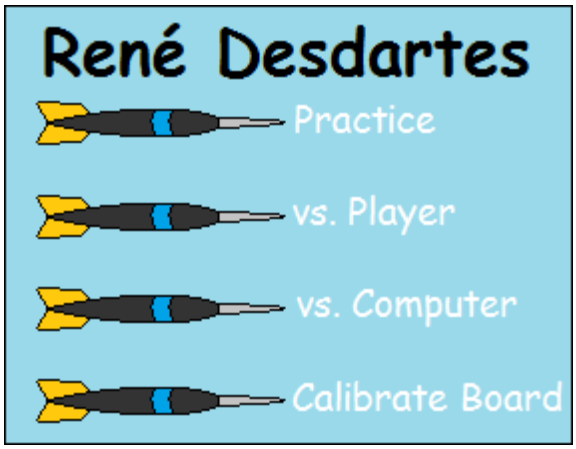

Figure 13: UI splash screen

```
def setupUI():
     while( systemOn ):
     #show the main menu
          selection = showMainMenu()

     # calibrate
     if selection == calibrate:
          set ( calibrationStartEvent )
          hideMainMenu()
          wait ( calibrationCompleteEvent )
     # quit game
     elif selection == quit:
          systemOn = false
     else
          if selection == practice:
               settings.gameType = practice
               settings.numberofPlayers = 1

          elif selection == versusPlayer:
               settings.gameType = versus
               settings.numberofPlayers = 2

          elif selection == versusComputer:
               settings.gameType = versus
               settings.numberofPlayers = 2

          #Show the game screen and hide the main menu.  Upon
          #opening the game screen, the user will be prompted to
          #input any extra settings such as user information
          startGameScreen()
          hideMainMenu()

          #set   gameStartEvent.UISettings   to   the   settings
```

```
previously
#established through user input
gameStartEvent.UISettings = settings
set( gameStartEvent )

stillPlaying = true

while(stillPlaying):
     wait( updateUIEvent )
     clear ( updateUIEvent )
     updateGameScreen( updateUIEvent )
     stillPlaying = updateUIEvent.stillPlaying
```

2. The startGameScreen() function displays the user interface of the main game. A sample of this user interface is given in Figure 14 where the user is facing an AI opponent and the AI opponent is throwing darts.

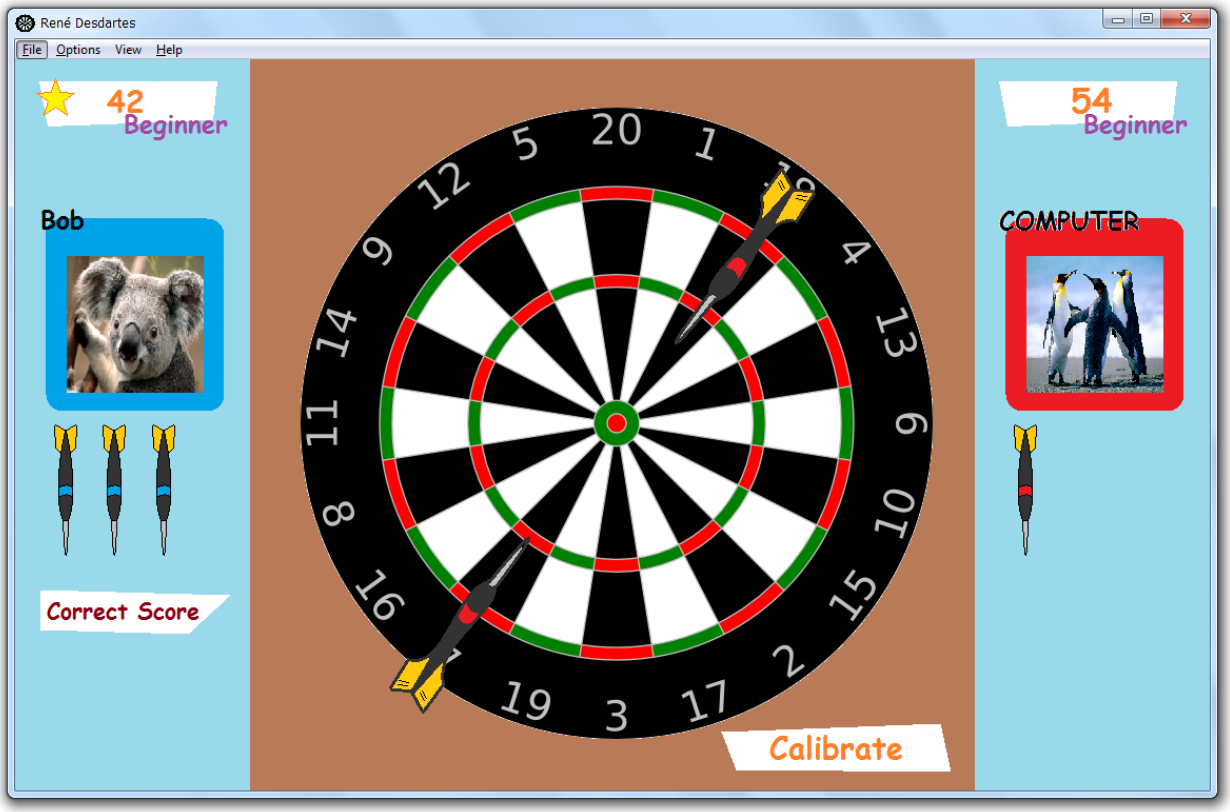

Figure 14: UI for playing with AI

```
def startGameScreen():
```

```
    #create the main game window and load any images
    gameScreen = CreateWindow("Réne Desdartes")
    gameScreen.loadImages()

    #prompt the user with extra settings input
    userInput = promptUser()
    settings.playerInformation = userInput.playerInformation
    settings.difficulty = userInput.difficulty

    return gameScreen
```

3. The updateGameScreen(event) function is called whenever an UpdateUIEvent event is detected and is passed to this function as a parameter. This function updates the score and current player, and draws a dart in an appropriate region of the dart board.

```
def updateGameScreen( event ):
    if currentplayer != event.dartThrow.thrower:
        clearDarts()
        displayCurrentPlayer()

    # Draw the dart on the board
    drawDart(event.DartThrow.region)

    # Update the score
    drawScore (event.score)
```

4. The correctScore() function is invoked when a user clicks the "Correct Score" button on the game screen as shown above. Here, the user will be prompted to input an integer value to correct the score in the case where the system is incorrect in assigning the score and needs to be corrected. The "Correct Score" button is only available for human players.

```
def correctScore( ):
    #Clear the event first in the case where the user attempts
to
    #fix the score more than once
    clear( correctScoreEvent )
    correctScoreEvent.player = currentPlayer
    correctScoreEvent.score = newScore
    set( correctScoreEvent )
```

3.6 AI Opponent

The responsibility of the AI opponent is to generate a fake dart throw in order to allow a single player to play against an opponent. The difficulty will change how accurate the throw would be.

First, a target will be selected by the AI opponent depending on the score and game type. Once a target has been selected, a Gaussian random variable will be used to get both magnitude and angle of a fake dart throw. The mean for these variables will be the target and the variance for these will change depending on the difficulty. The higher the difficulty, the lower the variance. Using this distribution, the AI Opponent will generate a random magnitude and angle, and the result will be a randomly generated dart throw.

3.7 Dart Detection

We need to detect when a dart has been thrown and extract the location of the tip. We then need to determine which region the dart has landed in. We use a simple background subtraction algorithm to detect the presence of the dart. We use a moving average as the background estimate. The algorithm is as follows:

Let *acc*(x,y) be the accumulator storing the moving average, and *image*(x,y) the latest frame. Then calculate the weighted sum of the input image *image* and the accumulator *acc* so that *acc* becomes a running average of frame sequence:

$$\texttt{acc}(x,y) \leftarrow (1-\alpha) \cdot \texttt{acc}(x,y) + \alpha \cdot \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

where α regulates the update speed (how fast the accumulator forgets about previous frames).

For example, if we have the initial *acc* as shown in Figure 15*:*

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Figure 15: Initial image

And this *image* as shown in Figure 16:

1	1	1	1	1	1	1	1	1	1
1	1	1	3	1	1	1	1	1	5
1	1	1	1	1	1	1	5	5	5
1	1	2	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Figure 16: Image with dart and noise

Assuming α = 0.6, then we have this new *acc* as shown in Figure 17:

1	1	1	1	1	1	1	1	1	1
1	1	1	2.2	1	1	1	1	1	3.4
1	1	1	1	1	1	1	3.4	3.4	3.4
1	1	1.6	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Figure 17: Image with dart and noise and alpha equals 0.6

Then from the new *acc*, we calculate a difference image *difference* such that:

`difference(x,y) = image(x,y) - acc(x,y)`

In our example, this would be (blank entries denote '0'), as shown in Figure 18:

			0.8						1.6
							1.6	1.6	1.6
		0.4							

Figure 18: Image of difference between *image* and *acc*

Finally, calculate a silhouette image *silhouette* by thresholding the difference image:

```
silhouette(x,y) = '1' if difference(x,y) > threshold
                =  '0' otherwise
```

Assuming threshold = 1.0, then we have the following silhouette in our example as shown in Figure 19:

									1
							1	1	1

Figure 19: Image of dart and all noise removed after applying threshold

(From this silhouette image we conclude the that the pixels in cyan have experienced motion)

The OpenCV library has the RunningAvg() function for maintaining a moving average, the Sub() function for performing pixel-wise subtraction between two frames, and a Threshold() function for thresholding.

Once the silhouette image is obtained, it can then be used to update the motion history image *mhi* as follows:

$$\texttt{mhi}(x,y) = \begin{cases} \texttt{timestamp} & \text{if } \texttt{silhouette}(x,y) \neq 0 \\ 0 & \text{if } \texttt{silhouette}(x,y) = 0 \text{ and } \texttt{mhi} < (\texttt{timestamp} - \texttt{duration}) \\ \texttt{mhi}(x,y) & \text{otherwise} \end{cases}$$

(*timestamp* is the current time in milliseconds or other units; *duration* is the maximal duration of time for which we track motion for a pixel, and has the same units as *timestamp)*

That is, MHI pixels where motion occurs are set to the current timestamp, while the pixels where motion happened far ago are cleared.

If we imagine having a 2x3 “object” moving across from-right-to-left at a speed of (1pixel/time unit), and assuming a very large duration parameter (so that MHI entries never get zeroed out), then we’ll have the following MHI as shown in Figure 20 after 6 time units:

		6	6	6	5	4	3	2	1
		6	6	6	5	4	3	2	1

Figure 20: Motion history index with dart example

OpenCV provides the the UpdateMotionHistory() function for updating the *motion history image*.

From the motion history image, the motion can be segmented to obtain a segment mask *seg_mask*. Basically, we have:

$$seg\_mask(x,y) = 1 \text{ when } mhi(x,y) = time_1$$
$$= 2 \text{ when } mhi(x,y) = time_2$$
$$\ldots$$
$$= n \text{ when } mhi(x,y) = time_n$$
$$= 0 \text{ when } mhi(x,y) = 0$$

where all the seg_mask elements with the same non-zero value ‘k’ represents “motion segment k” - a connected block of pixels which marks the location of the dart at $time_k$.

Because of the way we chose our time stamps, the segment mask in our example would be the same as the motion history image, and each colour would represent a motion segment.

A OpenCV provides the SegmentMotion() function to convert the motion history image into motion segments.

If we set up the camera such that it's looking at the dartboard from the left , then the thrown dart will enter from the right of the image and will proceed leftwards. Thus, given the motion segments of the dart, we can assume that the leftmost segment is the "tip" of the dart. Thus, we can assume that the dart has "stopped" when we no longer make leftward progress. I.e. if we have the segment mask at $time_{n+1}$,seg_mask_{n+1}, and the segment mask at $time_n$, seg_mask_n, then

```
if (leftmost motion segment of seg_mask_{n+1})= (leftmost motion
segment of seg_mask_n) then
     dart has stopped
else
     dart is still moving
end if
```

Once the dart has stopped, then we're ready to exact the location of the tip. We need to identify candidates for the "tip pixel". The obvious candidates are the leftmost non-zero pixels in the segment mask. However, a potential problem with this simple selection criterion is that a simple glitch (i.e. a light shown on the very left of the board) could lead us to identify those pixels as the tip. To minimize such false-positives, we require that for any candidate pixel to be chosen as the tip, we must be able to trace consecutive motion segments from that pixel to the very right of the image; in order words, if we detected that a leftmost non-zero pixel in the segment mask "arrived from the right", then it's a valid tip candidate. Otherwise, choose the next left-most motion segment (pixel). The x-coordinate of the tip is the x-coordinate of the tip candidates (which should all be the same, since they are all "left-most").

Once we've determined the x-coordinate of the tip, then, to find the y-coordinate, take the average y value of all the pixels identified as tip candidates.

For example, assume we have the following segment mask as shown in Figure 21 (blank entries denote '0'):

4	4								
4									
		7	7	6	5	4	3	2	1
	7	7	7	6	5	4	3	2	1

Figure 21: Detecting the tip of the dart

The pixel in cyan will be identified as the tip; the pixels in red in the top-left corner ignored, because they cannot be traced to the very right of the image.

Once the tip location has been located, then this can be mapped to a region of the dartboard, using the results of the board calibration. This result is returned to the game engine.

There remains some synchronization issues between the Dart Detection block and the Game Engine block. They will both be running on separate threads. We will use two threading.Event object to synchronize between the two. One Event object is for the Game Engine to signal to Dart Detection when to start expecting a dart. At this signal, the Dart Detection will need to clear structures like the MHI and the segment mask. The other Event object is for Dart Detection to signal to Game Engine when a dart throw has completed. When the Game Engine expects a dart throw, it will clear this Event object and wait on it.

3.8 Team Member Responsibilities

The reliabilities of each group members are listed in Table 8. The overall design has been discussed by the whole group and design decision are made through consensus. In addition, each group member has invested time and effort towards one specific components of the the system, so the responsibilities of building this system has been distributed this way. There will be many group interactions when the components are put together.

Table 8: Team member responsibilities

Components	**Team Member Responsible**
User Interface	John Patrick Crisistomo
Dartboard Calibration (including finding dart region)	Bryan Yan
Game Engine	Ken Woo
Dart Detection	Ching-Pei Lin

Chapter 4. Experimental Results

4.1 Prototype Testing Checklist

Group 39
René Desdartes – I Dart, Therefore I Am
Prototype Testing Checklist

Ching-Pei Lin, Ken Woo, John Crisostomo, Bryan Bo Yan
Consultant: Dr. Mohamed-Yahia Dabbagh

DESCRIPTION	TEST METHOD	PASS/FAIL	COMMENT
System must be able to detect the presence of darts on the board	Start the system and throw a dart to see if the system detects its presence	pass	
System should be able to detect the presence of a dart after another has been thrown	Throw two darts in a row, and verify to see if the system detects the presence of both darts	pass	
Calibration should take less than 5 minutes	Measure the amount of time it takes to calibrate the system	pass	done in < 1 min.
The system should be able to be re-calibrated if the environment changes	Move the board and recalibrate the system	pass	
Under controlled conditions, the system should be able to correctly locate the dart 70% of the time	Throw 10 darts, and measure how many darts were correctly located by the system	almost pass	detection 60-70% need to improve accuracy.
User must be able to correct the score in case of a system error	Change the score manually	pass	
The system should be able to correctly calculate the game scores if the darts are located correctly	Play a game and see if the game calculates the score correctly	pass	
The game should use the rules of darts to guide players through a game (i.e. letting players know when to throw, determining a winner, etc.)	Play a game and see if the system correctly identifies which player's turn it is, and if the system correctly identifies a winner	NA	not implemented yet

The system should be able to log a player's past throws	Throw a few darts under practice mode and see if the system logs the user's past throws	pass	one previous throw.
The player should be able to play against an AI opponent with flexible difficulty	Play in the single player mode and check to see if the AI component generates a dart throw. Try out the different difficulty levels	pass	done in the code. need to be included in the user interface
The system should interfere as little as possible with the dart game	Verify to see that the system setup does not physically obstruct the player's motions	pass	
The system should be able to locate a dart no longer than 7 seconds after the dart has come to a complete stand-still	Throw a dart and measure the time it takes from the time the dart is completely still until a location is determined	pass	
The system should take less than 15 minutes to setup (minus the calibration)	Measure the time it takes to setup the system (minus calibration)	pass	less than 2 min.
The system should be inactive if the user is not in one of the active game modes	Throw a dart while the system is in the main menu screen and verify that no score is computed	pass	

Grade: A

Date: July 23, 10

Overall Comments:

X

Dr. Mohamed-Yahia Dabbagh
Consultant

4.2 Experience Report

4.2.1 Organization challenges

It was a challenge to effectively coordinate a four-man software team; most of the

projects in our previous courses were in teams of two. We wanted to divide the work in a way such that each group member had a clear domain/area which he “owned”. While working on the block diagram, we had identified four main components which could be worked on concurrently, and we assigned those components to team members as follows

- Image Processing (Ben)
- Calibration (Bryan)
- Game Engine (Ken)
- User Interface (JP)

For the large part we stuck with this division of labour. We ran into some minor integration issues when we tried to put everything together. For example, Bryan used angles in the Calibration component, while JP used radians in the User Interface component. We had to jury-rig some angles-to-radians conversion code to make the two components work together.

To avoid integration issues such as the one above, it’s probably wise for one member (i.e. the head architect) to write stub implementations of all the modules / classes / functions first. Also, we need to clearly outline how the different components are supposed to talk to each other. However, due to time constraint, we didn’t have time to do this before different group members started working on his own component.

4.2.2 Function and Non-functional Requirements

All functional requirements are met as originally planned, except there were changes in the implementation methods for dartboard calibration procedure (segmentation method), dart location detection. In addition, there is future work planned to make the dart detection more accurate. All non-functional requirements are met. However, there are future works planned to make the system more robust in various environments (i.e. low/high lighting), to make the system setup out of the user’s way, and to make the UI more user friendly.

4.2.3 Changes from original design

For the most part the prototype followed the original design, yet it deviated in the two

following areas:

4.2.3.1 Segmentation of the board

In the high-level analysis performed prior to the implementation of the system, we said that we would segment the board (i.e. find all the different regions of the board) using a k-means clustering algorithm. However, in the detailed design, and on the actual prototype, we opted for a different approach. We realized that a dart board had very well defined geometric features – it's essentially six concentric circles divided into twenty slices of equal area (see Figure 22). Therefore, in order to completely segment the board, we only need to specify:

- An origin (1 click)
- The radius of the six concentric circles (6 clicks)
- An angle of reference for dividing the board into slices – once we have an angle of reference, we can divide the circle into twenty equal slices, each slice occupying 360 degrees/20 = 18 degrees (1 click)

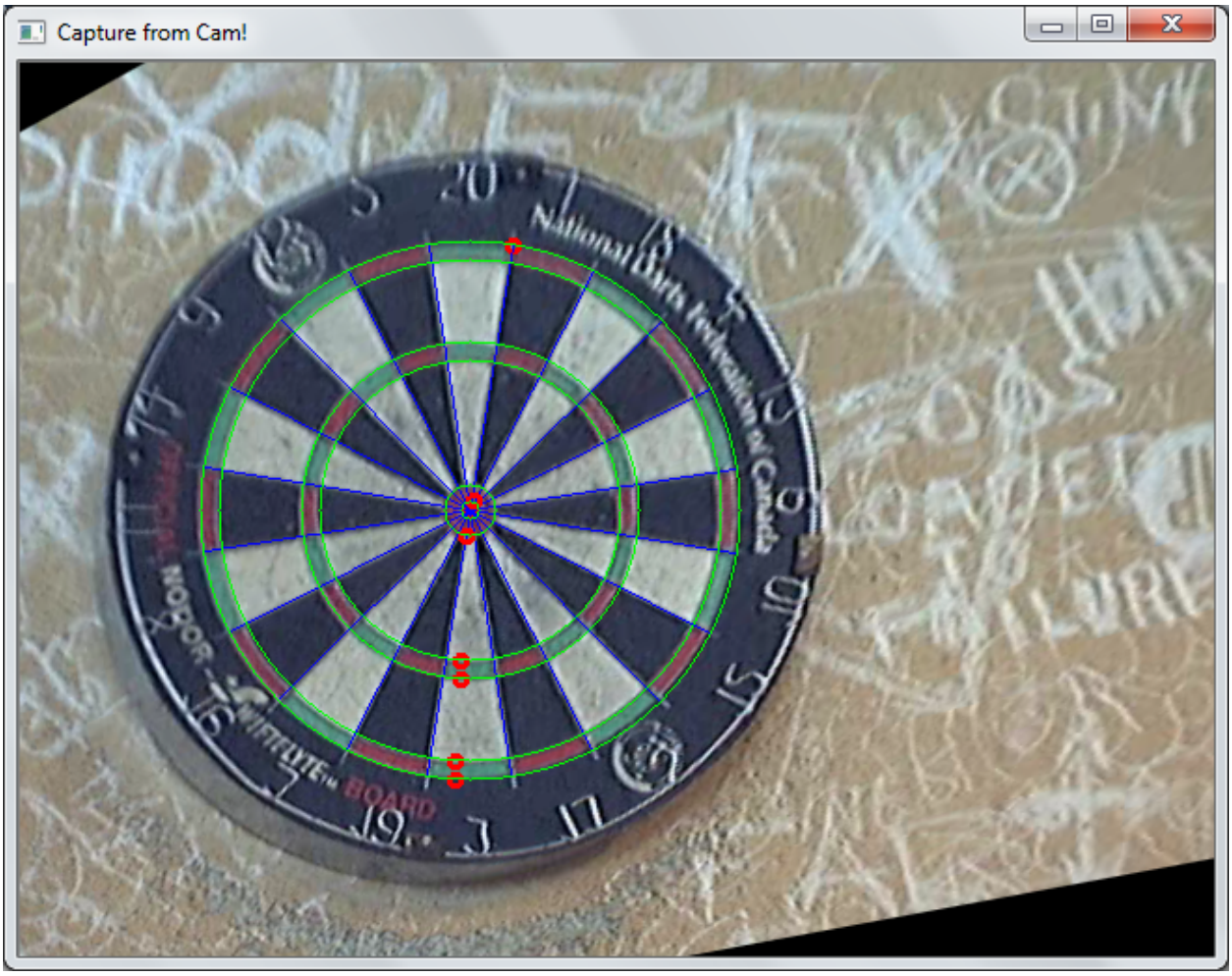

Figure 22 A dartboard, consisting of six concentric circles divided into twenty equal "slices"

Therefore, to obtain a complete segmentation of a dartboard, we only require eight clicks from the user. We felt this was a much simpler way to segment the board than a k-means clustering algorithm. The disadvantage is that we need to collect those eight clicks from the user every time set up the system. However, during the testing (and during the prototype demo) we've showed that this step can be completed in slightly less than 30 seconds.

One complication is that, since our camera is placed at an angle with respect to the board, the image of the board we receive is in fact not a perfect circle (or even an ellipse); therefore, prior to the segmentation step, we first need to apply a "perspective transform" to obtain the image of the board as a circle. Thankfully, OpenCV has already

implemented this perspective transform functionality (see Figure 23 and Figure 24)

Figure 23: Raw image of dart board - notice that it is not a perfect circle, and the '20' is not above the '3'

Figure 24: Corrected image of the dart board - it is now a perfect circle such that the '20' is above the '3'

4.2.3.2 Detecting the dart

In the design, we used a background subtraction technique to identity the pixels which have “moved”. We then reasoned that the “moved” pixels must belong to the dart, since we assumed a perfectly still board and a static environment. However, in practice we found that:

- The whole board shakes when the dart makes impact, which means that we’ll have non-dart pixels that move when the dart hits
- In low lighting conditions, the camera becomes very sensitive and often detects false pixel movements

And, finally, we realized that:

- When the player goes to retrieve the darts (or, if there’s someone/something other than the dart moving in the video), then we’ll have lots of pixels that move

Due to the problems mentioned above, we realized we needed a more refined

technique for isolating “dart pixels that moved” from “non-dart pixels that moved”. We opted to use a Haar feature-based object detection technique, since it was already implemented in OpenCV. Basically, the technique finds objects by looking for particular edge, line, and center-surround features. However, in order to find the right features to look for, we first have to train the system. We do this by taking pictures of the object we’re trying to find, annotating them (tracing out by hand where the object is in the picture), then feeding the data into the training program. We collected and annotated approximately one thousand pictures of the dart hitting the board (see Figure 25), and fed it into the Haar trainin program.

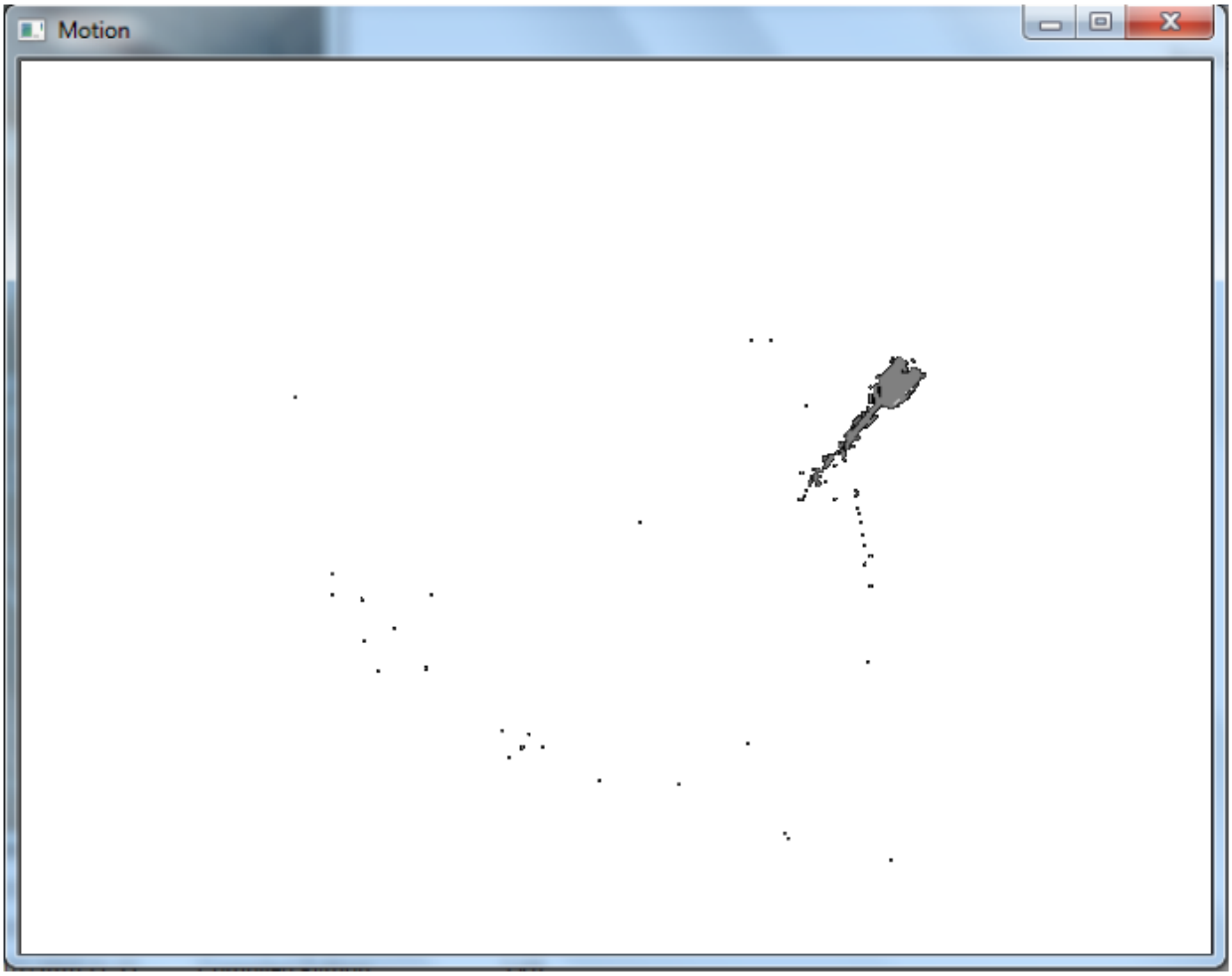

Figure 25: A dart hitting the board

The Haar training program creates a “cascade” file, which contains the features that

one should look for when trying to detect the object of interest. Using this technique, we were greatly able to reduce the number of false-positives (instances where the system detects a dart when there isn't one).

We also applied a simple threshold on the total number of pixels that "moved" in a frame. From the sample pictures we took of the dart hitting the board, we determined that 90% of the time, when the dart hits the board, between 200 - 2400 pixels "move" during the frame (out of a total of 640 x 480 = 307200 pixels, since our resolution was 640 x 480). When there are less than 200 pixels that move, then we assume that the pixel movements are caused by lighting glitches rather than a dart throw, and we ignore them; when there are more than 2400 pixels that change, we assume that the pixel changes are caused by someone moving in front of the camera (i.e. to retrieve the darts), so we also ignore them. Therefore, we only detect a "dart" when:

- Total number of pixels that moved during the frame is between 200 and 2400 pixels
- The Haar object detection returns a hit

Combining the two techniques, we were basically able to completely eliminate false-positives.

4.2.3.3 Detecting the tip of the dart

In our original design, we said that once we've found the dart, we'd use some feature detection technique to identify the tip of the dart. However, in the detailed design and prototype, we opted for a much simpler technique. We placed the camera at an angle to the board (in our demo, to the left of the board), such that when the dart comes in, it always arrives from-right-to-left. Therefore, to detect the tip, we simply needed to find the "left-most" pixel that moved.

In practice, due the complications mentioned above, it's not as easy as "finding the left-most pixel that moved". However, with the Haar object detection method, we're able to find the rectangle which encloses the dart (see Figure 26). Therefore, we just need to find the "tip" pixel within this rectangle.

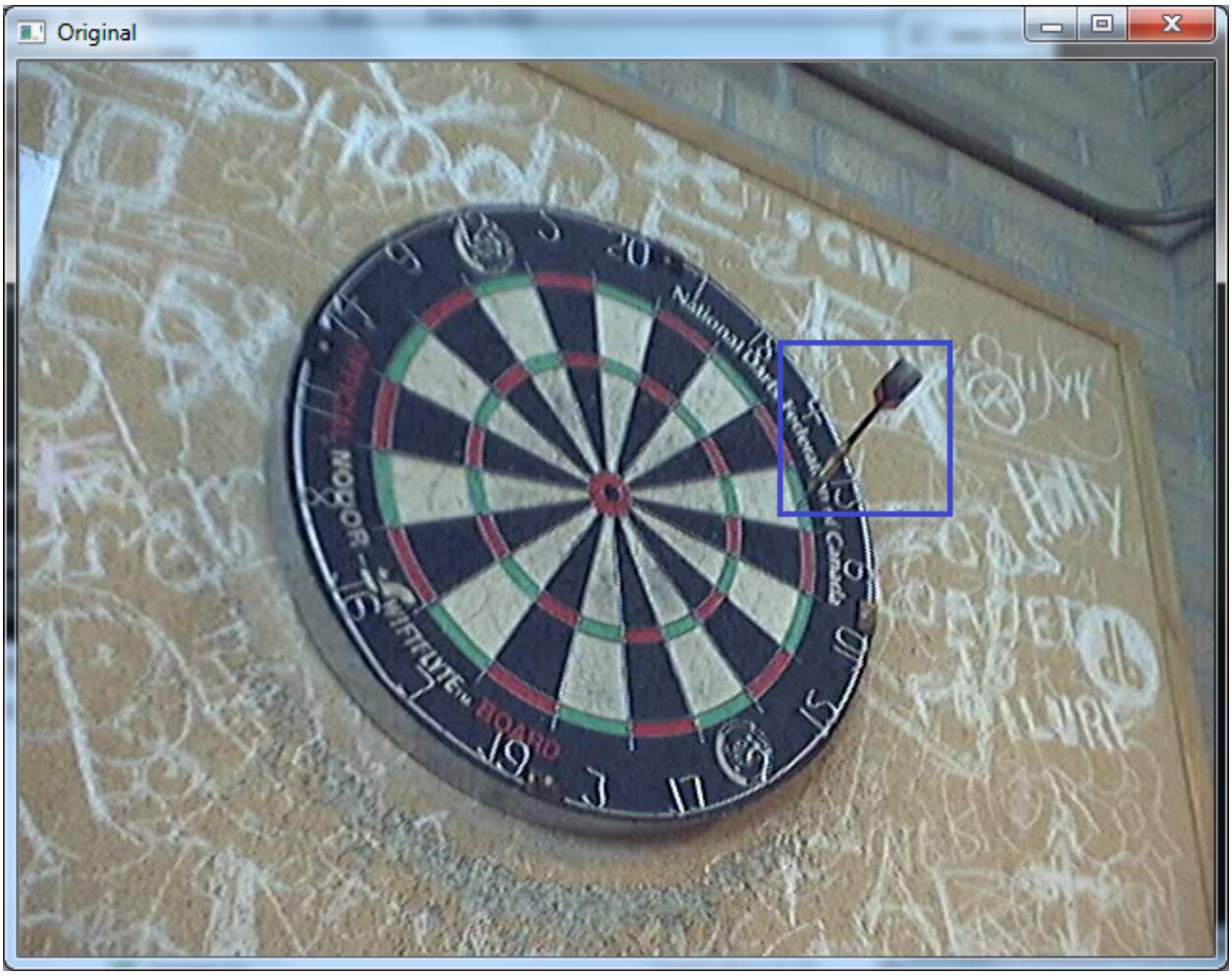

Figure 26: Haar object detection finds the rectangle which encloses the dart

The way we do this is by searching pixel-by-pixel through the detected rectangle, going left-to-right and bottom-to-up (since, with this particular setup, the dart tip is always on the left and often near the bottom of the rectangle, like in Figure 26). We're looking for a pixel that moved. However, due to the complications identified earlier (poor lighting, board shaking on impact, etc.), we see in Figure 25 that there are several "speckles" of movements that are not dart pixels (the speckles roughly form a ring where the outside edge of the board is). Therefore, to isolate these speckles from the actual dart tip, we require that, in order for a moved pixel to quality as the dart tip, there must be at least three other moved pixels in the immediate 3 x 3 region surrounding the pixel in question (see Figure 27); otherwise, we ignore the pixel. Using this technique, we've successfully detected the tip pixel while filtering out noise pixels.

Figure 27: A valid tip pixel - notice that there are three other moved pixels in the immediate 3 x 3 region

4.2.4 Successes

4.2.4.1 Dart detection

After the dart's location is detected from image processing, the location can be accurately converted to a score based on the regions calibrated by the system by first performing a perspective transform on the raw dart location, then compare the magnitude and the angle of the dart (reference is the origin of the dartboard) with each scoring region and rings of the dartboard.

4.2.4.2 Board segmentation

The calibration could very accurately segment the dartboard if users take good care to follow through the calibration procedure. If the user accurately picks the origin of the dartboard and the line divider between the 20 points and 1 point, then each scoring regions can be accurately calculated: there are 20 scoring regions on the dartboard, and each scoring region is 18 degrees large (360 degrees divided by 20); the blue lines in the Figure 22 illustrates the division of the dartboard scoring regions. If the user accurately selects the rings of the dartboard, then each multiplier region of the dartboard can be accurately fitted by a perfect circle (the radius of each circle is calculated by the magnitude of each user selection); the thin green lines Figure 22 illustrates the dartboard's multiplier regions and bull's eye. The red dots in Figure 22 illustrate the user clicks during the calibration procedure.

4.2.4.3 Game facilitation

In order to simulate a real dart game, the rules of the popular "501" was used in our system. In this game, the user aims to reach 0 points starting from 501, with the only restriction being that the last dart thrown has to land on the double ring or the double

bull's eye. The system correctly facilitates the rules of this game as the system correctly calculates the scores and switches users when appropriate. The system also correctly identifies a winner and ends a player's turn whenever they "bust" (making it impossible for them to win). In addition, the UI also correctly links up with the game, and displays where the darts were thrown. The UI also allows a user to correct their past throw, which was great help when testing the system and making sure it followed the rules of the game.

4.2.4.4 AI

The AI opponent generates a dart throw based on the current score of the game. The AI has been programmed so that it selects the most optimal target to aim at in order to win the game. In order to mimic a real dart throw, after a target is selected, a Gaussian distribution is applied to the target magnitude and angle. This creates a chance for the AI to "miss" its intended target and hit a neighbouring region. Gaussian distribution is ideal for simulating a dart throw, since the variance can be adjusted to set the difficulty. For a difficult opponent, a small variance is used, so that the AI hits the target much of the time. The variance can be increased in order for the AI to be easier.

The implementation of the AI in the system was a success, since a dart throw is randomized around a selected target. Different difficulty settings were tested in order to generate a fair AI. The AI was successfully linked to the system without much difficulty and the AI makes its throws during its turn when a user finishes their turn. Due to the variance we used, the AI ended up being fairly tough, winning most of the time against the player. It is also apparent that the AI selects the correct target to aim at based on the proximity of the 3 darts thrown or when the AI needs a certain amount of points to win.

4.2.4.5 UI

With the UI, we were successfully able to draw darts on the UI rendered board using dart data provided through image processing and passed through the game engine to the UI. The dart data from the real board is passed to the UI and redrawn, and this is accurate in terms of the data provided. Also, in the case that the dart data is incorrect,

the user can simply drag and drop the darts pictures on the UI to the correct location. This will automatically update the score to the new corrected dart configuration. Currently, the darts can be modified for the current set and the previous set for each player.

The drawn darts simulate whose turn it is and whose darts are currently on the board. Every time the players switch turns, the darts are redrawn on the respective user's panels to signify that the darts are off the board. In the Figure 28, it is player 1's turn and player has thrown 2 darts.

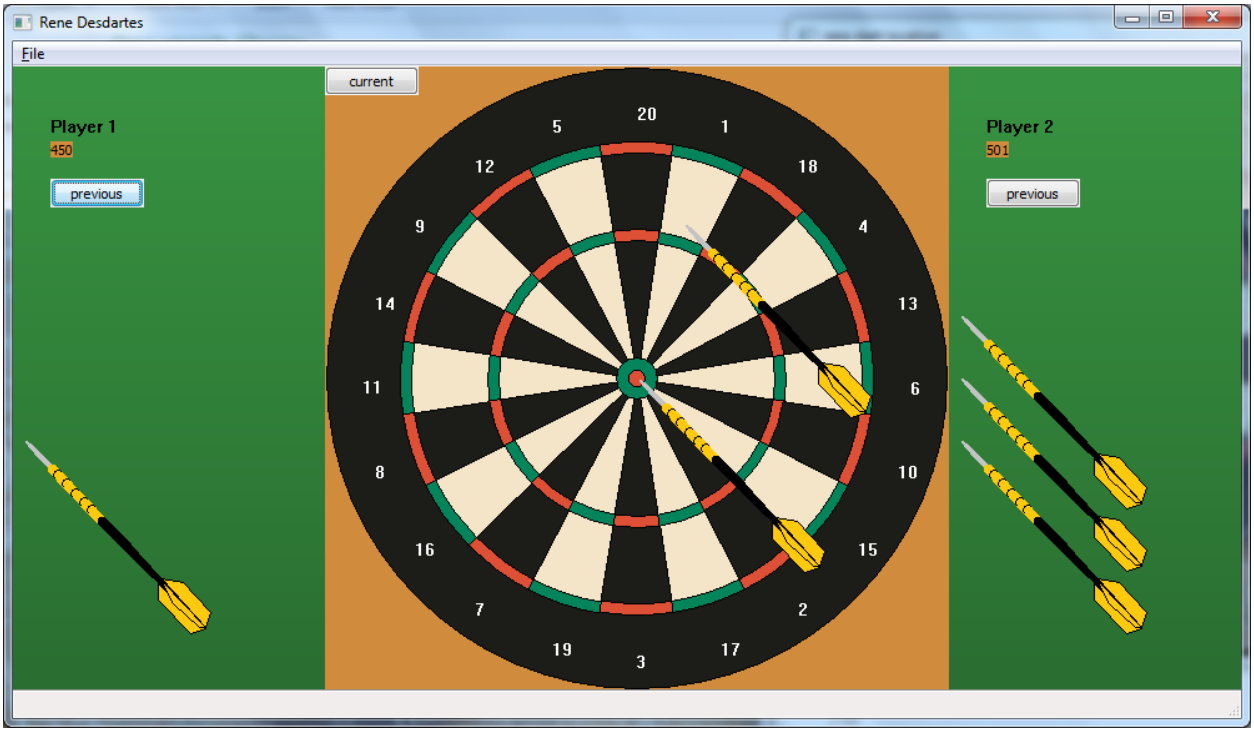

Figure 28: The user interface

Also, with the ScoreKeeper object, both the UI and the game engine can edit the score and darts by accessing this object instead of trying to manipulate the score through each other. The UI updates the respective set of darts when correcting the score.

4.2.4.6 Game logging

Whenever the user is in a practice mode, each throw that is recorded is stored in a file located in a save folder. The file is unique to each player, and has a history of their past throws. The format of the file is created using Python's pickle function, which is very useful when retrieving information from the file. Basically, the system stores the dart throw as an object, containing a magnitude and an angle. This is then appended to a list of their previous throws. By storing the throws as objects, it allows a lot of flexibility when retrieving the stored information, because when the file is read, the list of dart throws is created, and no parsing is needed.

Chapter 5. Discussion and Conclusions

In this section we summarize the limitations of our system, discuss the novelty of the design, and suggest possible future work on the system.

5.1 Limitations

Table 9 summarizes the limitations of our system:

Table 9: Limitations of the system

Limitation	Reason for limitation	Comparison to existing products
The camera has to be set up either to the left or to the right of the board - it will not function if the camera was mounted directly below or above the board	We currently find the "tip" of the dart by looking for the leftmost/rightmost pixel that changed recently, depending on whether or not the camera is mounted left/right of the board	Electronic dartboards have no such constraints
The object detection routine can only detect throws it's been trained for	Different throws have different angles of entry into the board, making object detection difficult	Electronic dartboards do not need to be trained
System can falsely identify shadows as darts	The system uses a simple frame subtraction technique to detect motion; unfortunately, this technique cannot distinguish between shadows and actual objects	Electronic dartboards unaffected by lighting
System needs to be manually calibrated by the user	The system requires the user to manually identify the different sections of the board	Electronic dartboards do not require manual calibration

5.2 Novelty

All the of the computer vision techniques employed in our system (frame subtraction, Haar object detection, etc.) were well known to the computer vision community. However, we believe we were one of the first groups to combine these techniques for the detection of darts. There is another group at the University of Sheffield who have also been working on a vision-based scoring system for dart games; this group completed their work in May 2009 [8]. They, like us, used frame subtraction to detect the presence of a dart; they also used the OpenCV library. The two groups worked independently to achieve these similar results.

5.3 Future work

5.3.1 Making the system work under different conditions

5.3.1.1 Dart orientations

As mentioned in the section “Detecting the dart”, we had to train the Haar object detection method to find features to look for object detection. For the prototype demo, we only trained with dart throws from one member (Ben). However, when we tried the system with throws from another member (Ken), we found that the system didn’t work very well. We realized that this was due to the d difference in throwing s style between the two members. Ken throws the dart a lot harder, which means that the dart enters the board nearly parallel to the ground; by contrast, Ben throws a lot softer, so since the front of the dart is heavier than the back, Ben’s throws tend to enter the board pointing down. Since we trained the Haar object detection with Ben’s throws, the detection technique has learned to track an edge pointing downwards; however, the object detection sometimes failed to detect Ken’s throws (which results in edges that are flat). The way to address this issue is to training the Haar object detection with pictures of darts with varying angles of entry. We’re planning to showcase our system at the next “Warrior Weekends” event at the University of Waterloo; we will invite participants

to try out our system and, in the process, collect throw samples which we will use to train the object detection algorithm.

5.3.1.2 Camera setup

Currently, the system setup only allows the camera to be mounted on the lower-left side of the dartboard, because the darts are trained to be viewed from a lower-left angle, and the dart detection module can only successfully detect the darts from this setting. In the future, the training of the dart detection module could be done from all angles, or another solution has to be explored.

Also, the current camera setup may be in the way of the user since it is installed on the same level as users. In the future, we may be able to mount the camera high above the height of human beings, so the camera will be for sure out of the user's and the bystanders' way.

5.3.1.3 Lighting

Currently, if there is a strong external light source shining onto the dartboard, the image processing picks up the shadow of the dart as a "phantom" dart, and that "phantom" dart may skew the calculation of the dart location and affect the dart scoring. To solve this problem in the future, the system could employ a strong lighting source emitting from the camera to remove all the shadows on the dartboard.

5.3.2 Connecting the UI to the rest of the system

Currently, our UI is connected only to the Game Engine. One of the nuances is that the Game Engine calls the board calibration routine and brings up the corresponding screen without any action on the UI side. In the future, we would like for this interaction and all other game flow interactions to be controllable from the UI. For example, a user interaction with the UI could trigger an even t that will either bring up the calibration screen by itself or through event handling procedures in the game engine. This way, the overall interface becomes more user friendly.

Another aspect to be improved upon with the calibration routine in terms of interface is that it uses two windows: the first being a picture of the dart board to be calibrated,

and another for user instructions in a console window. Since this setup is rather cryptic from the user interface point of view, an improvement to the UI is that the dart board picture and user instructions be merged into one window for easier usage by the user. Doing this will require more connection between the calibration component and the UI component of our current implementation. Also, we would like to hide the game window during the calibration process so as to avoid having too many shown windows at a point in time.

With respect to the console window, we would like to eventually suppress this window and have only our implemented UI on the screen. All information (i.e. calibration instructions, dart throw info) that is currently fed to the console window will be made available on UI. Doing this will improve user friendliness by not having too many windows open, especially with the cryptic console window.

With the ScoreKeeper object, the score and darts can be modified from the UI as previously mentioned, however there are issues when a player wins/loses through incorrect data but loses/wins through corrected data. To solve this, more communication is needed between the UI and the game engine for correcting the score events. Also with the correcting the score, we currently have no way to fix the score of a dart that failed to be registered completely or a dart that failed to stick on the board.

Overall, we would like to have the UI more user friendly and have the look and feel of a typical computer dart game.

5.3.3 Game Improvements

Currently, the only game that our system can facilitate is “501”. Even though that game is one of the most played dart games, there are a couple of variations that the group can potentially implement. One obvious one is “301”, which is basically the same game, except the user starts with 301 points in the beginning. The user should also be able to tweak various minor rules that people may play with, such as starting out with a double as well, or not having to hit a double on the last dart. These are minor tweaks to the game facilitation, but would allow users to have more control as to what kind of game they want to play.

In addition to the “-01” games, a dart game known as Cricket is also very popular.

Instead of calculating scores for each throw, a player aims to hit a region multiple times before the other player. This is a competitive game and would also fit in nicely with our system. Many of the aspects of the current game facilitation can be used in Cricket, such as players having 3 darts each and so on. In addition, the AI would also be easy to implement for Cricket. The main difference is that instead of a score, the engine would have to keep track of how many times a region has been hit. This may require a bit of rework in the engine, but adding a different type of game would be a great addition to our system.

A bit of research can also allow us to find out some other dart games that people play, so that we can implement this in our system as well. The more options that can be presented to the user, the better the system will be.

As an improvement to the currently practice mode, a target generation system could be implemented, where the system either automatically generates a target for the user to aim at, or the user can choose a region to aim at. This would add more depth to the practice mode, as it only logs throws right now and does not allow a user to keep track of whether or not they hit a certain target. Having a target to aim at would allow the user to practice with more of a goal, rather than just throwing towards the board. Having a target would also allow us to keep track of how accurate the user is at throwing darts and may allow us to adjust the AI so that the play is competitive.

In mentioning the AI, the system should also allow a user to easily adjust the difficulty of the AI. This would allow players to switch the difficulty if the player finds the computer too difficult or easy. Maybe a style of personality would increase the fun factor of playing against a computer. The style could adjust how they aim, or may even produce “taunts” or comments during the course of a game. This would allow the AI to seem more personable, rather than an item that just generates throws.

5.3.4 Installer

In the future, our group hopes to create an installer that allows the user to install all the tools necessary to run the system. Ideally, it would bundle the installer for Python, OpenCV, WxPython, and the camera driver. This would allow the user to just run one installer, rather than 4 different installers in order to start up our system. We hope to add

flexibility to the installer so that the user can adjust many of the settings of each installer directly, rather than being prompted by each installer individually. Another feature that would be useful is that the installer detects either already installed, or older versions of the programs we want to install. This would prevent any duplicate programs being installed, and would allow a user to update an old copy of a program directly from our installer. The aim of the installer is to aid users that might not know how to find and install the programs that the system needs and to save the user time when starting our system for the first time.

5.3.5 Better calibration scheme

We need a new way to calibrate the system. Right now, the calibration requires the user to select the centre of the board, then select each of the rings on the board, then select the edge between the 20 score section section and the 1 score section. The system is extremely sensitive to the correctness of the calibration. Thus, we currently require the user to make very accurate calibration choices. In the future, we wish to either automate the calibration process, or at least aid the user in the process. For example, we could, as we did in our original design, use Hough transforms to try and detect candidate eclipses and circles to fit to the rings of the board. We could also use edge detection to try and detect the different scoring regions.

5.3.6 Open source project

We've hosted the project on Google Code under the MIT licence. It's our hope that, after the symposium, we can invite other developers to join the project.

References

[1] opencv v2.1 documentation. (2010). *Haar Feature-based Cascade Classifier for Object Detection* [online]. Available: http://opencv.willowgarage.com/documentation/python/objdetect_cascade_classification.html

[2] D. Forsyth and J. Ponce, Computer vision: a modern approach. Upper Saddle River, NJ: Prentice Hall, 2003.

[3] opencv v2.1 documentation. (2010). *Motion Analysis and Object Tracking* [online]. Available: http://opencv.willowgarage.com/documentation/python/video_motion_analysis_and_object_tracking.html

[4] OpenCV. (2010). *OpenCV* [online]. Available: http://opencv.willowgarage.com/wiki/

[5] opencv v2.1 documentation. (2010). *Geometric Image Transformations* [online]. Available: http://opencv.willowgarage.com/documentation/python/imgproc_geometric_image_transformations.html

[6] opencv v2.1 documentation. (2010). *Miscellaneous Image Transformations* [online]. Available: http://opencv.willowgarage.com/documentation/python/imgproc_miscellaneous_image_transformations.html

[7] wxPython. (2010). *wxPython* [online]. Available: http://www.wxpython.org/

[8] R. B. White, “Improving a Video-Based Darts Match Analyser,” COM 2021 report, Department of Computer Science, The University of Sheffield, Sheffield, United Kingdom, 2009.

Appendix A

The final project budget is listed in Table A-1.

Table A-1: Budget of the system

Item	Price	Source
Dartboard	$100	Team
Darts (x3)	$50	Team
Computer	$500	Team
Poster	$200	Team
Team Total	$850	
Camera	$200	ECE Project Inventory
ECE Project Inventory Total	$200	

Some of the cost was actually not required, as the team already had the computer. The items that were purchased were the dartboard and darts. The camera was borrowed from the ECE Project Inventory as well.