Assisted Puzzle Assembly

Second Semester Report
Spring Semester 2014

- Full Report -

by
Casey Anderson
Tess Bloom
Sam Felton
Laura Imbler

Prepared to partially fulfill the requirements for
ECE 402

Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, Colorado 80523

Project advisors: Dr. Anura Jayasumana, Dr. Sudeep Pasricha

Approved by: Anura Jayasumana and Sudeep Pasricha

Abstract

The goal of the Assisted Puzzle Assembly project is to provide children with developmental delays a tool to practice puzzles and other games with interactive assistance, while reducing the need for constant adult supervision and aid. Children with such delays require tremendous repetition of puzzles and games to achieve basic developmental milestones, and may have difficulty completing the games on their own. Most physical toys and puzzles (non-electronic) on the market are very limited in the options and customizability they can provide and tend to require constant adult assistance. Most electronic educational systems, while more flexible, use a touch-screen or button interface that does not give children with developmental delays the type of physical practice that they need. Our project aims to include the benefits of both physical and electronic games.

For this project, we have developed an electronic game system that provides visual hints to help guide the child through the completion of the puzzle or game, and encouraging audio feedback to keep the child interested and motivated. It uses physical game pieces, just like a manual puzzle, to give the full tactile experience. It is also designed to monitor the basics of the child's performance, such as the time taken to complete the puzzle, to help track progress and identify trouble areas. To allow for these physical game pieces and a potential variety of different games, tracking and monitoring of game play is done via a camera suspended above the game board coupled with color tracking software. Each piece is tracked by its color; this information is used to identify which piece the child is placing at any point in time, generate hints and feedback, and determine whether the piece has been placed correctly.

The game system has been designed to function as a platform upon which new games and functionalities can be developed. As of now, the basic tracking and hint algorithms, a mechanical structure, and variations of the knob puzzle game have been developed. Future development may include a greater variety of puzzles and puzzle types, 3D games, greater customizability to the user and more complex hints and feedback.

Table of Contents

Abstract 2
List of Figures 5
List of Tables 5
1 INTRODUCTION 6
1.1 Project Overview and Requirements 6
1.2 Design Overview 6
2 SOFTWARE DESIGN 8
2.1 Visual Tracking Library 8
2.2 Language and IDE 9
2.3 Object Tracking Algorithm 9
2.3.1 Object Tracking Challenges 13
2.4 Graphics and Hints/Feedback 14
2.4.1 Graphics 14
2.4.2 Hints 15
2.5 Gameplay Structure 17
2.6 User Interface 17
2.7 Calibration 18
3 MECHANICAL DESIGN 20
3.1 Objectives and Constraints 20
3.2 Design Summary 21
3.3 Design Decisions 22
3.3.1 Concept Selection 22
3.3.2 Feasibility Analysis 25
3.3.3 First prototype 26
3.4 Failure Modes and Effects Analysis (FMEA) 27
4 PROJECT MANAGEMENT AND ETHICS 29
4.1 Project Management 29
4.2 Ethics 29
5 BUDGET, MANUFACTURABILITY, AND MARKETABILITY 30
5.1 Budget 30
5.2 Manufacturing 30
5.3 Marketability 31

6 FUTURE WORK 31
6.1 Performance 31
6.2 New Games and Features 32
7 CONCLUSION 32
References 33
Acknowledgments 33
Appendix A: Abbreviations 34
Appendix B: Budget 34
Appendix C: User Manual 36
C.1 Hardware 36
C.1.1 Hardware Requirements 36
C.1.2 Hardware Setup 38
C.1.3 Changing the Camera 39
C.2 Software 39
C.2.1 Installing OpenCV 39
C.2.2 Installing Assisted Puzzle Assembly Software 40
C.2.3 Obtaining Assisted Puzzle Assembly Source Code for Development 40
C.2.4 Installation Folder 40
C.3 Using the Assisted Puzzle Assembly System 40
C.3.1 Camera Settings 40
C.3.2 Launching the Program 41
C.3.3 Calibration 41
C.3.4 Calibration and Color Tracking Tips 43
C.3.5 Running a Game 43
C.3.6 Displaying Results 44
C.3.7 Changing Sound Effects 44
C.4 Software Development 44
C.4.1 Source Code 44
C.4.2 Organization 44
C.4.3 Major Components 45
C.4.4 Adding New KnobPuzzles 46
C.4.4.1 Steps: 47
C.4.5 Adding new game types: 48

C.4.6 Known Bugs 49
Appendix D: Timelines 50
Appendix E: Source Code 53

List of Figures

Figure 1: Mechanical Assembly 7
Figure 2: Knob-puzzle boards 7
Figure 3: Object tracking algorithm 10
Figure 4: Basic shapes puzzle board 15
Figure 5: Algorithm to generate visual hints 16
Figure 6: Main GUI Window 18
Figure 7: Screenshot of calibration process 19
Figure 8: CAD models of prototype 21
Figure 9: Stress results of applied load on acrylic base 25
Figure 10: Initial prototype of game board 26
Figure 11: Initial prototype of camera stand 26
Figure 12: Moment diagram of camera mount 27
Figure 13: Small parts cylinder 30
Figure 14: Camera Stand 37
Figure 15: Puzzle board and pieces 37
Figure 16: Camera stand position 38
Figure 17: What the camera view should look like 39
Figure 18: Icons 41
Figure 19: Color calibration - the red piece is correctly calibrated 42
Figure 20: Sample knobpuzzle input file 47

List of Tables

Table 1: Descriptions of object tracking algorithm steps 11
Table 2: Challenges and solutions to object tracking 13
Table 3: Objectives 20
Table 4: Constraints 21
Table 5: Board Design Decision 23
Table 6: Board Attachment to Monitor Decision 23
Table 7: Board Attachment to Monitor Decision (Round 2) 24
Table 8: Vertical Camera Mount Design 24
Table 9: Failure Modes and Effects Analysis (FMEA) 28
Table 10: Expenditures to Date 34

1 INTRODUCTION

1.1 Project Overview and Requirements

This project was requested by a local medical campus. The system will be used mainly in a clinical setting, but the system may be adapted for use in a household setting in the future.

Based on these goals and the requested functionality, the primary objectives of the system were determined as followed:

- Use of physical objects for game play
- Provide hints and signals to help the child solve the puzzle or complete the game
- Provide encouraging feedback to the child
- Monitor and track child's performance in the game and over time
- Keep the child interested and motivated - keep his/her attention
- Be suitable for a variety of ages - from young children (around 3 years old) to young adults

The design team consisted of two computer engineers, one electrical engineer, and one mechanical engineer. The design team imposed the following additional objectives, with an emphasis on extensibility:

- Allow for incremental development and new features
- Allow for addition of new game types and varying levels of difficulty
- Ability to customize gameplay to user

The end-of-year prototype has satisfied these primary goals and requirements.

1.2 Design Overview

The design team was given great latitude in deciding how best to achieve the objectives and in adding new functionality as desired. The decision to allow for multiple game types made a flexible, extensible system infrastructure a necessity. As well, the use of physical game objects made most current educational game systems based on touchscreens and buttons inapplicable.

It was decided to use a computer/TV monitor as a game board, topped by a clear plastic overlay with cutouts to fit plastic game pieces. A standard desktop setup (desktop PC, upright monitor, keyboard, and mouse) is used to run the system. A webcam is suspended above the game board to monitor gameplay and accommodate visual tracking. The assembly can be seen in Figure 1 (desktop computer setup not shown):

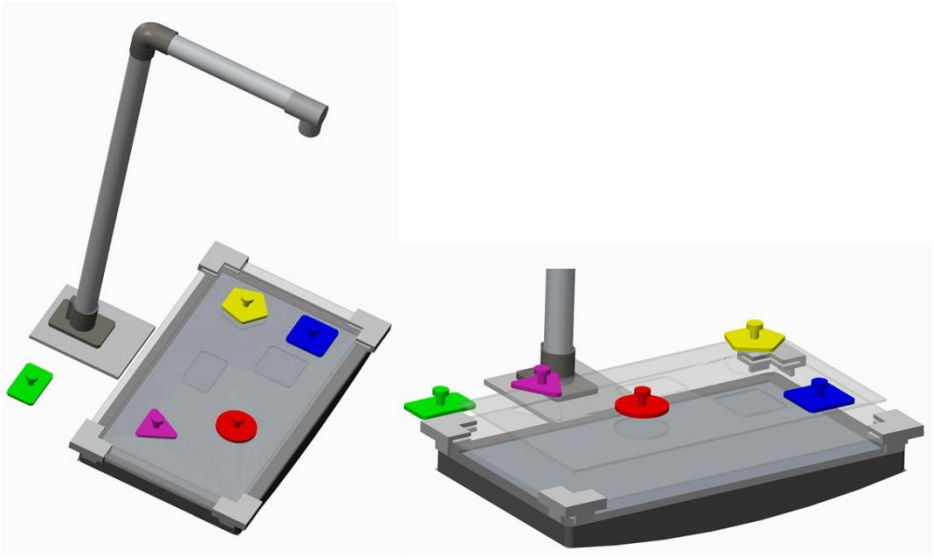


Figure 1: Mechanical Assembly

The camera is mounted to the game board monitor via custom support structure. Individual plastic game board overlays and corresponding pieces have been custom made for each game or puzzle. Game board overlays and pieces can easily be switched out by lifting the overlay from the monitor, putting on the new overlay, and then loading the game file and calibrating to set up the new game. Two knob-puzzles were developed this year shown in Figure 2.

Figure 2: Knob-puzzle boards

The camera is the primary driver behind the assistive gameplay; visual tracking software is used to track where each game piece is and recognize which piece the child is currently holding, and when that piece is placed correctly. Pieces are tracked by color and movement is determined by tracking the location of each piece over time. The information gathered by the tracking software controls when hints and feedback should be given. The times taken for the child to place each individual piece and finish the puzzle are recorded and stored. The current tracking algorithms are tailored for the Knob Puzzle game - in the future, various tracking algorithms could be developed to monitor and interpret different types of games and puzzles.

For information on how to set up and operate the system, refer to the User Manual in Appendix C.

2 SOFTWARE DESIGN

The software for this project can be found on the public GitHub repository at this link:

https://github.com/asimo42/PuzzleAssembly

The source code can also be seen in Appendix D.

2.1 Visual Tracking Library

Visual recognition and tracking is a complicated software task that was outside the scope of this project. Fortunately, there are a variety of open-source libraries available that provide basic visual recognition and tracking capabilities. The library OpenCV was ultimately chosen. Some of the other libraries that were considered are as follows:

- OpenTLD
- SwisTrack
- Skilligent
- ARToolKit

All of these libraries have different strengths and weaknesses and their own method of recognition and tracking. For instance, some look for patterns of light and dark, some compare images to large image databases, and others track color. For this project, it was decided that color tracking would be the most effective method. This is because the game pieces will be partially occluded as they are picked up, changing their visual 'shape.' However, as long as some part of the piece is visible, the color can still be tracked. Also, the puzzle/game nature of the project lends itself to brightly colored, highly differentiated pieces.

Of the color tracking libraries researched, OpenCV was chosen because it appeared to be the most extensive, fully developed, and well documented.

2.2 Language and IDE

The choice of programming language and Integrated Development Environment (IDE. Note: all abbreviations are available in Appendix A) was based on compatibility with OpenCV and the experience of the team members. Ultimately, the use of C++/CLI in Microsoft Visual Studio 2012 was chosen.

OpenCV is written in C++. When development began, it was uncertain whether it would be necessary to work directly with the OpenCV source code to modify and customize functionality. It was decided it would be easiest, in the case that customization was necessary, to code in C++ for direct compatibility. Wrappers are available to use OpenCV in other languages, but this seemed an unnecessary complication and restriction.

However, it is difficult and time consuming to create a Graphical User Interface (GUI) and other on-screen visualizations in standard C++; C#, especially in a Visual Studio environment, is the best choice for GUIs because of the built in functionality of the .NET framework. C# is also much easier and faster to code. Fortunately, Microsoft created a 'bridge' language known as C++/CLI. This language can compile directly alongside C++, but contains the added functionality of the .NET framework and can be used in a manner that is syntactically very similar to C#. Thus, C++/CLI was chosen for development.

2.3 Object Tracking Algorithm

Consistently and accurately tracking each puzzle piece is crucial for the smooth operation of all other features of the project. It is the base upon which all of the user feedback features (i.e. visual and audio hints, congratulations for correctly placed pieces) depend. Therefore, significant effort was put into tuning the tracking algorithm to be as consistent and robust to noise as possible.

The pieces' locations are tracked continuously as the user plays the game. This is done by filtering out the specific color of each piece from each camera frame and calculating the center of the color blob. Each piece is a different enough color that it can be tracked independently of the other pieces. This location data is then processed to determine which piece is moving and if a piece has been placed in its correct location on the board.

The object tracking algorithm uses several different OpenCV functions applied sequentially to each camera frame. An overview of the algorithm with a picture of the output of each image transformation is shown in Figure 3. Table 1 provides more detail on each step.

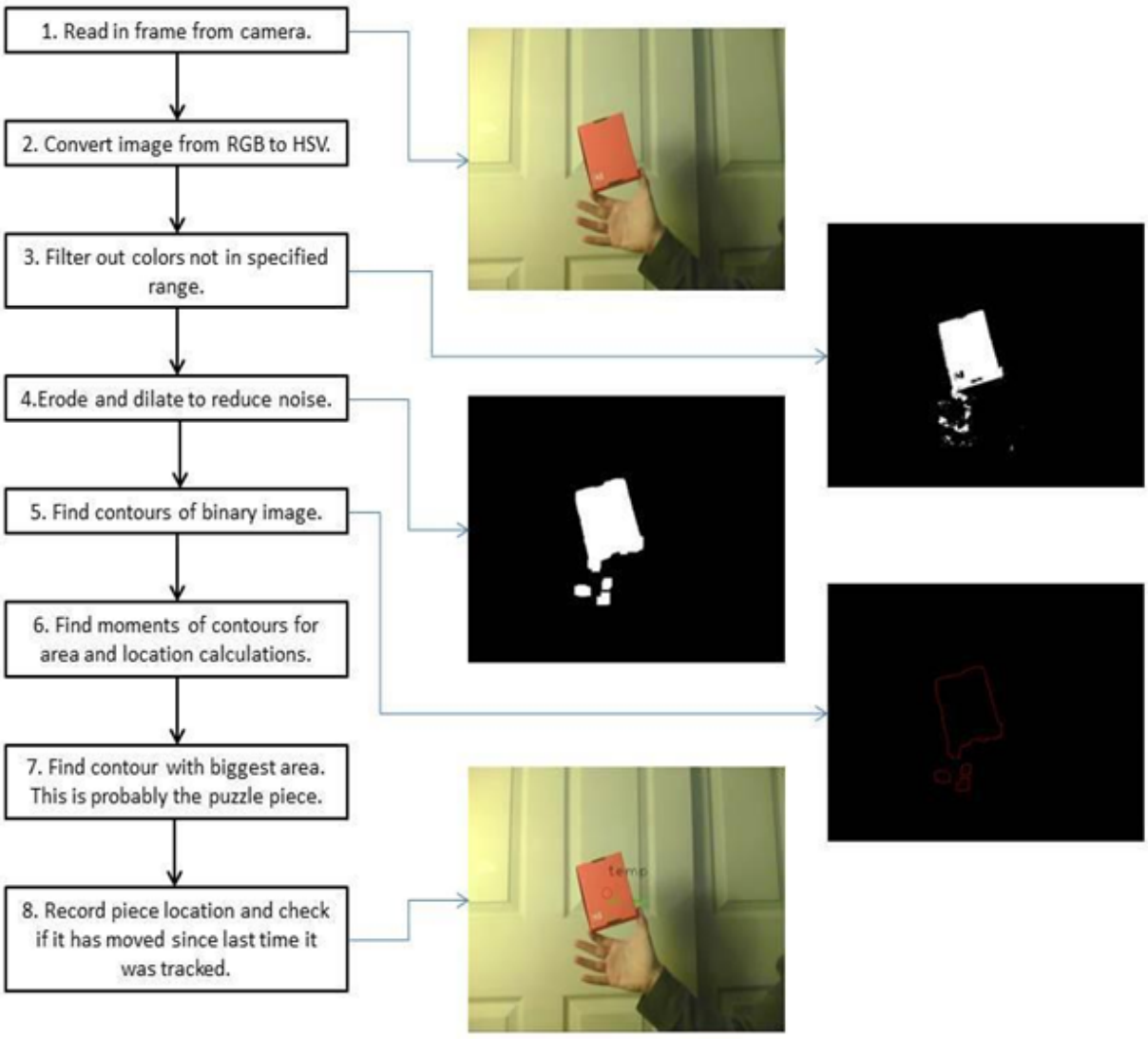


Figure 3: Object tracking algorithm

Table 1: Descriptions of object tracking algorithm steps

Algorithm Step	Description
1. Read in frame from camera.	Reads in a frame from the USB camera at 640x480 pixels in BGR (Blue, Green, Red) format.
2. Convert image from RGB to HSV.	Converts image to HSV (Hue, Saturation, Value) format. Hue describes the "color" (e.g. yellow, orange, blue). Saturation is similar to the "grayness" (e.g. pure/bright colors have a high saturation while pale/grey tinted colors have low saturation). Value is similar to "brightness" (e.g. black has zero brightness). This is a much more natural color format for humans to associate to colors and makes finding ranges for specific colors much easier than BGR format.
3. Filter out colors not in specified range.	Converts HSV image into a binary (black and white) image where white pixels are ones that fall within minimum and maximum values of hue, saturation, and value.
4. Erode and dilate to reduce noise.	First erodes the binary image which "eats" away at the edges of white blobs. Small blobs which were probably noise will erode away entirely. Then the white blobs are dilated which grows back the edges of white blobs. This returns the blobs to their original size but coarser than before.
5. Find contours of binary image.	This finds contours or outlines of the white blobs in the binary image. This technique is used because it allows the area and location of each contour to easily be calculated later by finding the moments each contour. It also

	allows multiple objects of the same color to be tracked since each object will show up as its own contour.
6. Find moments of contours for area and location calculations.	This applies Green's Theorem to each contour to calculate moments up to the third order. The area and center location of each contour is extracted from the moments.
7. Find contour with biggest area. This is probably the puzzle piece.	This step compares the area of each contour and picks the largest area as the one corresponding to the piece being tracked. The other contours are ignored and assumed to be noise. This technique assumes only one object of each color needs to be tracked, but allows noise to be present while making a best guess of which blob is the piece. It also allows the piece to be tracked while partially covered by the user's hand as long as there is still some amount of color showing.
8. Record piece location and check if it has moved since last time it was tracked.	The location of the piece is extracted from the moments and it is compared to the previous location of the piece the last time it was successfully tracked. If the location is different enough based on a threshold value, it is noted that the piece has moved and this information is saved to be processed in order to give hints to the user. (It is assumed that the piece that is moving consistently for a few frames is the one the user is currently holding and moving around.)

For each camera frame, the sequence in Figure 3 is executed for each piece being tracked. This means the algorithm runs through these steps five times on each frame for the five piece knob puzzle - once for each piece. With each frame being 640x460 pixels, this becomes a significant amount of computation and

limits the number of unique colors that can be tracked. The computer system running this algorithm must have high enough performance to reach an acceptable frame rate. Initial testing suggests that most standard desktop PCs have sufficient processing power, but older laptops may experience difficulty.

2.3.1 Object Tracking Challenges

There are many challenges with consistently and accurately tracking different colored puzzle pieces. Most issues have to do with changing HSV ranges for pieces based on different conditions and dealing with noise in the filtered image. The solutions involve a balance between robustness to noise, flexibility of HSV ranges, and complexity/execution time of the algorithm. Table 2 shows challenges encountered and ways they have been alleviated. There will likely be more issues uncovered as more testing is done.

Table 2: Challenges and solutions to object tracking

Challenge/Issue	**Solution**
Changing lighting conditions change the HSV ranges needed to filter each color.	The "value" (the V in HSV) range is similar to the brightness of the object. By relaxing the "value" range, the piece can be tracked more consistently in different lighting conditions.
When the user picks up a piece, their hand conceals part of the piece from the camera.	The algorithm selects only the largest blob of color found, so as long as part of the piece is still showing and it is still the biggest blob detected, it will still be tracked. If the piece is concealed entirely, there is not much that can be done. The piece will be tracked again once it becomes visible.
Shiny/reflective objects can create a reflective glare on the surface at certain angles that drastically changes the HSV range of the object's color.	A rough surface or matte finish is easier to track since it is not as shiny and reflective. The 3D printed pieces have a rough finish that does not easily reflect light and helps alleviate this problem.
There is almost always some amount of noise after HSV filtering even in ideal	The erode and dilate step in the algorithm helps remove small bits of noise. Also, since the algorithm only selects the blob with the

conditions.	largest area, noise with smaller areas are ignored.
The user's clothing or item worn on the wrist may be picked up and confused for a piece depending on the color of the clothing or item.	Careful calibration may reduce this issue. The HSV range for the pieces may need to be tightened so similar colors on clothing will be filtered out as long as they are not the exact same color as a piece. Otherwise, troublesome objects may need to be removed from the game area.
A piece may disappear from the camera's view because it was moved outside of the camera's viewing angle or because the user's arm/body is covering it up.	With the current tracking techniques, there is not a way to track something that the camera cannot see. If a piece is not found, its location data will not be updated. It will be tracked again as soon as it becomes visible again.

2.4 Graphics and Hints/Feedback

2.4.1 Graphics

In order to utilize the monitor being used to display game boards and visual hints, a simple way to display basic graphics had to be developed. Fortunately, OpenCV provides ways to draw simple shapes, such as rectangles and circles, as well as a way to draw polygons as long as the locations of all the vertices are known. Using the functions provided within OpenCV, a class was developed to draw all of the shapes used in the puzzle. This class contained separate functions for each shape that made it easy to draw them at any size or color. Each function simply takes in a starting point, edge length, and line thickness as arguments and then calculates all the necessary information required to make a call to an OpenCV function that will draw the shape. Some shapes require more arguments, such as a rectangle which takes two edge lengths as arguments. Figure 4 shows the basic shape puzzle board drawn on the monitor using this class.

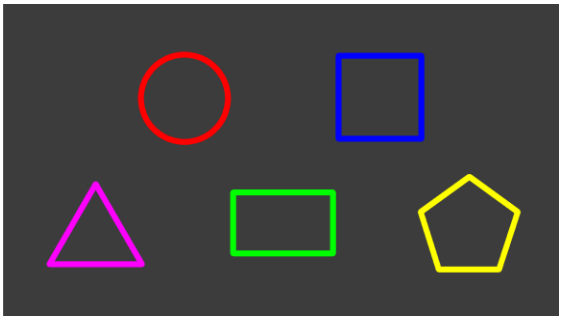


Figure 4: Basic shapes puzzle board

2.4.2 Hints

One of the main aspects of this project is providing the user with visual hints when they are playing the game. Being able to control what is displayed on the monitor below the game board gives the ability to supply a variety of visual hints. After getting feedback from Anschutz Medical Campus, it was decided to provide three different difficulties of the puzzle, each giving a different visual hint. The hardest difficulty simply flashes on the monitor whatever shape was being moved by the user, the medium difficulty flashes the shape being moved and dims all the other shapes shown on the monitor, and the easiest difficulty flashes the shape being moved and turns off all the other shapes on the monitor.

The difficult part about implementing these hints was not manipulating the shapes on the monitor but getting the hints to occur fluidly and have a natural timing during game play. The final algorithm for giving hints can be seen in Figure 5. This algorithm acts as a low-pass filter on the movement of pieces.

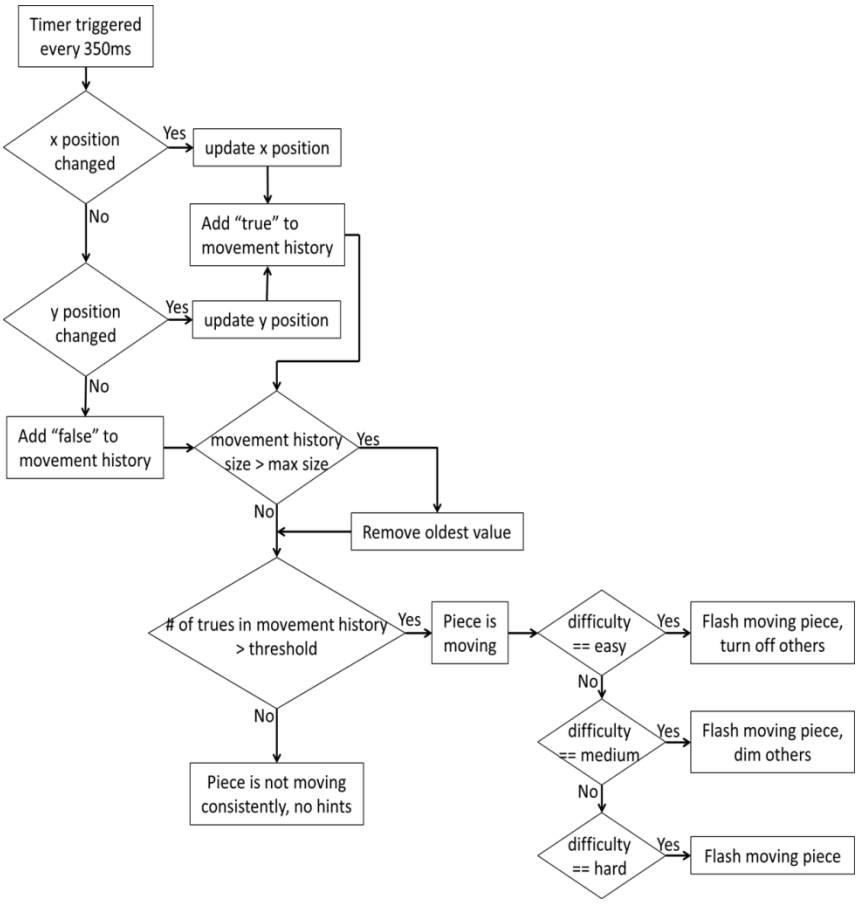


Figure 5: Algorithm to generate visual hints

Within the code there is a timer that gets triggered every 350ms. When this timer is triggered a series of checks is run on each piece to determine if the visual hint should be triggered for that piece. This means that the flow chart above is run through multiple times, once for each piece, every 350ms. The first condition that is checked is whether or not the X and/or Y coordinate of that piece has changed. If either has, it will update the current position of piece and push a "true" onto the movement history for that piece. If the piece has not moved, a "false" is pushed to the movement history. The movement history is simply a queue of Boolean values that records if the piece has been moving recently or not. When a value is pushed onto the movement history, a check is run to see if the history has exceeded its max size. If it has, the oldest value is pushed out of the queue. Next the number of "trues" in the movement history is compared to a predefined threshold. This threshold determines how soon a hint will be triggered after the user starts moving the piece. The greater the threshold, the longer it will take to trigger a visual hint. If the number of "true" values is less than the threshold, no hint is triggered. If it greater than the threshold, the visual hint based on the difficulty chosen will be triggered. Using the movement history and the threshold helps make the hints appear at a time when you would expect them appear. A user must continuously move a piece for a short amount of time before a hint will be triggered. This prevents a hint being displayed because a piece was bumped by the users arm or hand. This also means a hint will be

displayed for a short amount of time after the piece stops moving, allowing the user to take short breaks in movement without continually turning the visual hint on and off.

2.5 Gameplay Structure

The overall game play structure was designed to be as open-ended and extensible as possible, with potential for adding new games, new game types, and new functionality with minimal changes to the base code. The structure is primarily object oriented, with individual classes handling all variables and records, and running the tracking code. Some main structural elements are:

- GUI for loading a new game, starting the game, stopping the game, and reading stats.
- Calibration system - leads user through calibration process and records calibration data.
- System for importing game board data. Each individual game board will come with a standardized file that describes the key information about that game (e.g. type, number of pieces).
- System for score keeping, to record times stats, and progress. This system monitors the tracking process, stores data, runs necessary calculations, and then shows the stats when requested.
- System for running and managing the visual tracking and hint algorithms

Adding new knob puzzle games is easy. Each knob puzzle game comes with a text file that specifies each piece, the HSV range and destination of each piece, and the shape-drawing values required to draw it on the game board. This file is modified each time the camera is calibrated. This means that new knob puzzle games, if they contain only already-recognized shapes, can be added by manually creating a new game file and placing it in the game directory. If the puzzle contains new shapes, those shapes must first be coded into the program.

New game types must also be coded into the program, but can be implemented with relatively little impact on the overall architecture, especially on the user interface. Additional or modified tracking algorithms may be necessary to handle different game types.

2.6 User Interface

The user interface was designed to be simple and easy to use. It is self-contained, so the user rarely, if ever, has to do anything with the computer outside of launching the GUI using the executable file. The GUI is for use by an adult supervisor, not by the patient. All GUIs were programmed using C++/CLI and the Windows Forms Application template in Visual Studio. Appendix C provides a user manual for setting up and running the program.

The main GUI shown in Figure 6 allows the user to:

- Select a patient name (if a record exists) or create a new one
- Select a game to play from the available games and a level of difficulty
- Start a game or stop a running game

- Launch calibration
- Go to performance data

Help documents are provided for the main program and for the calibration process.

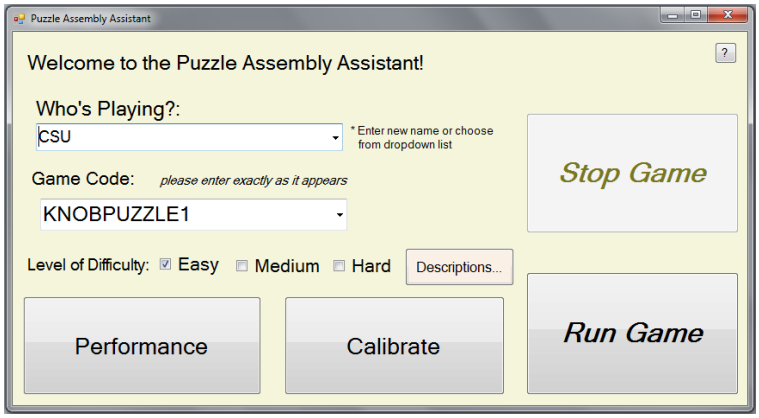


Figure 6: Main GUI Window

2.7 Calibration

The calibration menus step the user through the calibration process, and save all calibration data at the end. Figure 7 shows a screenshot of the calibration process. The calibration process is:

1. User places all pieces on board
2. OpenCV color tracker is launched. HSV sliders are provided for adjusting the color filter, and are preset to the most recent calibration settings.
3. When prompted, user adjusts HSV range using sliders for each puzzle piece, one at a time.
4. Once color calibration is complete, the user is prompted to place all pieces in their assigned locations on the board.
5. Program automatically searches for each piece and records its location - this data determines the 'final destination' of each piece, in order to recognize when a piece has been placed correctly.
6. User is asked if they would like to save calibration data for future sessions.

Depending on lighting conditions, which may affect the ease of calibrating, this process normally takes under 5 minutes. A quick calibration check is recommended before starting each new session.

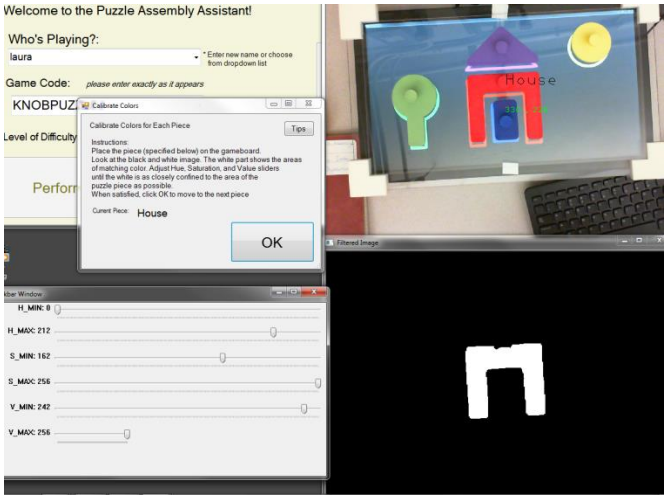


Figure 7: Screenshot of calibration process

3 MECHANICAL DESIGN

3.1 Objectives and Constraints

In order to ensure that this product satisfies the needs of the user as well as complies with the limits of this project, objectives and constraints were decided upon. The objectives shown in Table 3 are a summary of the quantified goals of this project. The priority level of each objective is noted by a number on a scale from 1 to 5, 1 being the least important and 5 being the most important. The constraints shown in Table 4 are the identified limits of this project that can be quantified.

Table 3: Objectives

Objective Name	Priority Rating	Method of Measurement	Objective Direction	Target
Tolerances	4	Dimension (in)	Minimize	≤ ±0.05
Dimensions of Puzzle Board	5	Dimensions (in x in x in)	Equal	$= 17\frac{1}{8} \times 9\frac{3}{8} \times \frac{1}{2}$ (To fit dimensions of 20" monitor)
Cost of Materials	3	Dollars ($)	Minimize	< $250
Weight of board	4	Weight (lb)	Minimize	≤ 2
Weight of entire device	4	Weight measurement of board, base, arm extension, and camera as one unit (lb)	Minimize	≤ 6

Highest Priority = 5, Lowest Priority = 1

Table 4: Constraints

Constraint Name	Method of Measurement	Limits
Weight of puzzle board	Weight (lb)	3
Distance of camera from board (to allow for best visual tracking results)	Dimension (ft)	2
Weight of camera	Weight (lb)	0.3 - 0.4
Safety	Size of puzzle pieces (in x in) Number of sharp edges present	ASTM F 963-11 None
Budget	Dollars ($)	$1400

3.2 Design Summary

The final design for this project is illustrated in Figure 8 below.

Figure 8: CAD models of prototype

For the puzzle board, it was decided that the best concept considering functionality and manufacturability was that shown in Figure 8 above. The design consists of two components: a solid base sheet of acrylic and a sheet of acrylic with shapes cut all the way through. Both sheets of acrylic are clear so that hints, etc. needed to correctly solve the puzzle can be seen on the monitor screen. The base sheet of acrylic has four corner pieces that serve two functions: to act as placement guides and to hold the top acrylic sheet in place. The protruding material on the bottom side of the corner pieces snugly fit around the edges of the monitor to prevent the board from sliding. The top sheet of acrylic fits within the inner dimensions of the corner pieces. This allots for the absence of a fastener. The absence of a fastener makes for ease of interchangeability of puzzle boards if the user would like to play a different game.

Considering manufacturability, this design is ideal because it is difficult to cut holes to a blind depth. The puzzle board is manufactured by first creating a model in Creo, a 3D simulation software, and then programming a CNC machine to cut precise shapes. The four corner pieces and puzzle pieces will be modeled in Creo and 3D printed. The material used in the 3D printing process is ABS plastic. This is a good material choice for the corner pieces because it is lightweight but still strong. ABS plastic is also an ideal choice for the puzzle pieces because it is safe and non-toxic for a child to be handling.

For the camera mount, the final design chosen is shown above. It consists of a vertical, free-standing rod and a horizontal rod that are attached by an aluminum rail fitting as the joint. In addition to the hollow tubing being relatively lightweight, it allows for the USB plug and cord on the back of the camera to run through the center of the tubing. This prevents the cord from obstructing game play. Both of these rods are made out of hollow 1” aluminum tubing as this material is strong but still lightweight. At the end of the horizontal rod, the camera is attached by a hollow plastic piece.

The vertical rod is attached to a free-standing steel base by a floor-mount flanged rail fitting. The steel base acts as a weight to resist the moment caused by the added weight of the camera and horizontal rod.

Hardware specifications can be found in Appendix C.

3.3 Design Decisions

3.3.1 Concept Selection

Rounds of Pugh matrices used to evaluate the design concepts are shown in Tables 5-8. How well each design satisfies a list of criteria that is crucial to this product was ranked on a scale of 1 to 10. A ranking of 1 means that the idea satisfies the criteria very poorly and 10 means that the idea satisfies the criteria very well. The rankings that each idea received were added together for a total sum. The idea that had the highest total sum was chosen as the best concept.

Table 5: Board Design Decision

		Idea	
#	Criteria	Single piece of clear acrylic with groves cut into it where puzzle pieces fit	Piece of clear acrylic with shapes cut all the way through that sits on an acrylic base
1	Board fits monitor	9	9
2	Is lightweight	8	8
3	Clear so can see through to monitor (For visual clues)	10	10
4	Ease of Manufacturing	5	8
5	Cost	8	8
6	Board can be interchangeable	6	9
7	Can be easily attached to monitor	6	8
	Sum	**52**	**60**

Best Idea: Piece of clear acrylic with shapes cut all the way through that sits on an acrylic base

*Base design does not exist so table above is based on number ranking

Table 6: Board Attachment to Monitor Decision

		Idea		
#	Criteria	2 clamps on each side of monitor	4 clamps: on the front, back, and sides	"Table" structure with side supports
1	Lines board with monitor screen accurately	3	6	7
2	Is stable	7	7	7
3	Not likely to break after repeated use	6	6	8
4	Ease of manufacturing	5	5	9
5	Reasonable cost	7	7	7
	Sum	**28**	**31**	**38**

Best Design: "Table" structure with side supports

"Table" structure will consist of a free-standing structure that sits on a surface and does not directly attach to monitor.

Table 7: Board Attachment to Monitor Decision (Round 2)

		Idea		
#	Criteria		"Table" structure with notches that fit monitor	Free-standing structure that stands directly on monitor
1	Lines board with monitor screen accurately	**Base Design ("Table" Structure)**	+	+
2	Is stable		0	0
3	Not likely to break after repeated use		0	0
4	Ease of manufacturing		-	+
5	Reasonable cost		0	+
	Sum		**0**	**3**

+ = better than base design
0 = equivalent to base design
- = worse than base design
Scoring:
+ = 1
0 = 0
- = -1

Best Design: Free-standing structure that stands directly on monitor

Table 8: Vertical Camera Mount Design

#	Criteria	Drawer runner	Two vertical supports on either side of board that are adjustable by pulling spring-loaded knobs	Vertical support with rod attached at joint that extends forward over board	Linkages similar to that of an adjustable desk lamp
1	Does not have rotational or horizontal motion for ease of programming	10	10	10	2
2	Stable	5	10	8	8
3	Ease of manufacturing	6	8	9	6
4	Does not get in the way of someone trying to solve puzzle	8	6	8	8
5	Cost	8	8	9	7
6	Weight	5	7	7	7
	Sum	**42**	**49**	**51**	**38**

Best Design: Vertical support with rod attached at joint that extends forward over board

Best design choices were implemented in the final design, previously shown in Figure 8.

3.3.2 Feasibility Analysis

In order to analyze whether acrylic is a reasonable material to select for the puzzle board, the properties of acrylic including the flexural strength and ultimate stress must be taken into consideration. Because this product will be used by children, it is possible that a child will lean on or put pressure on the acrylic board which will cause the board to flex and/or break. It must be verified that the acrylic will be able to withstand any potential load applied. According to Professional Plastics [1], the flexural strength of acrylic is within the range of 12,000 - 17,000 psi. The tensile strength is within the range of 8,000 - 11,000 psi.

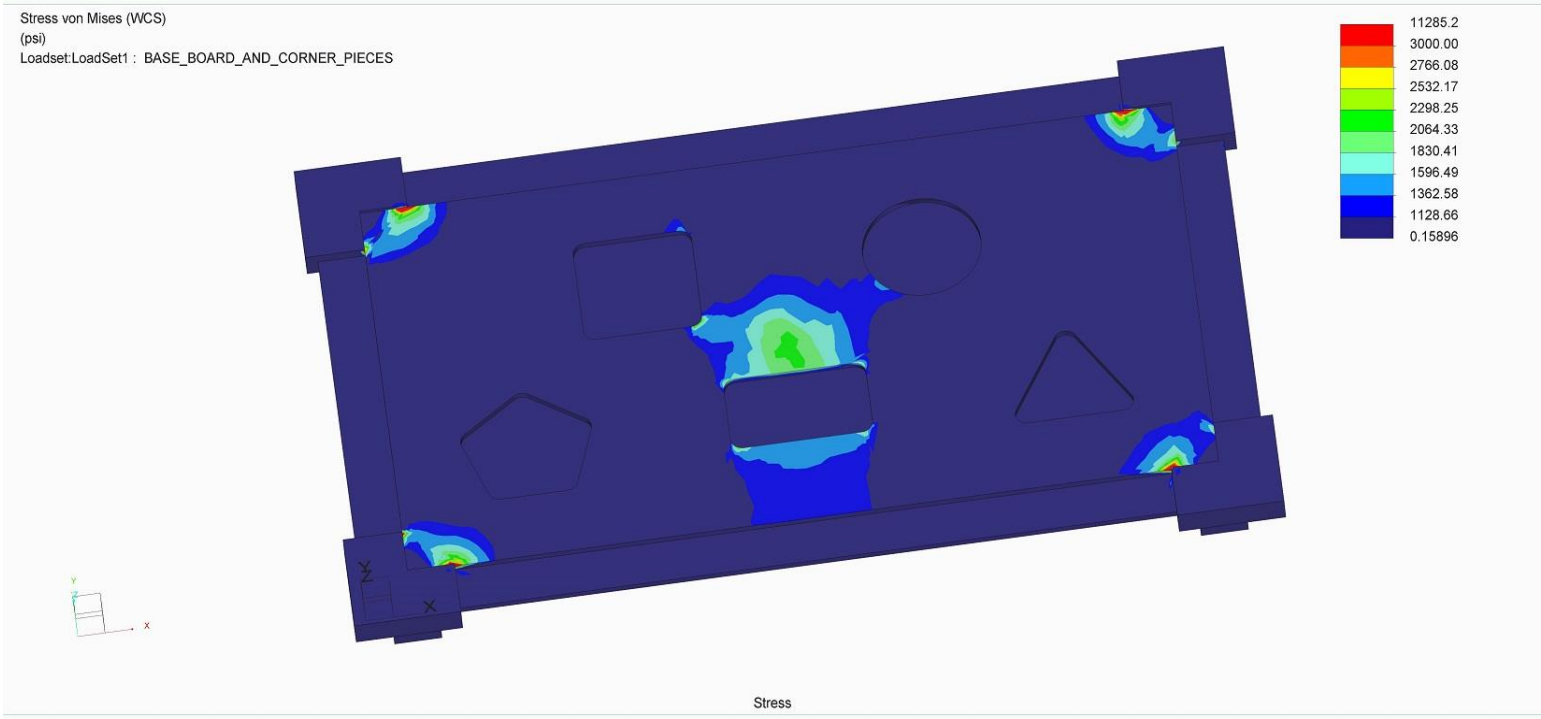


Figure 9: Stress results of applied load on acrylic base

As seen in Figure 9, a simulation was done in which a 60 lb point load was applied to the center of the base acrylic sheet and each of the four corners (where the plastic pieces are in contact with the board) were constrained. As can be expected, the simulation showed a higher stress concentration towards the center of the acrylic where the load was applied and near the corner pieces. As applying a center point load to the acrylic will cause the acrylic to bend, the flexural strength of the acrylic is the property that needs to be taken into consideration. The majority of the resulting stress from the applied load is well under the flexural strength of 12,000 - 17,000 pounds. There are very small areas of high stress near the plastic corner pieces that approach the

flexural strength so all users of this product should be notified that more than 60 lb of force should not be applied to the puzzle board.

3.3.3 First prototype

The chosen design for this project needed meet the design requirements shown in the Design Constraints table above. It could be verified that the weight and size requirements were satisfied by evaluating an initial prototype. The initial prototype is shown in Figures 10 and 11.

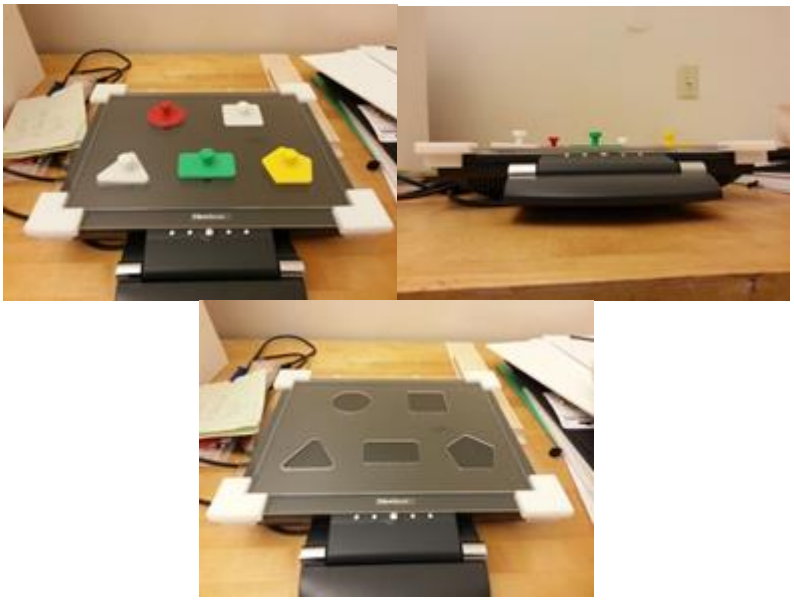


Figure 10: Initial prototype of game board

Figure 11: Initial prototype of camera stand

In the original design, the camera mount was to be attached to the acrylic base. When evaluating this as a physical prototype, however, the camera mount had a tendency to lean. This is because the weight applied by the camera mount caused a moment about the base.

A moment diagram analyzing the moment applied by the camera mount is shown in Figure 12.

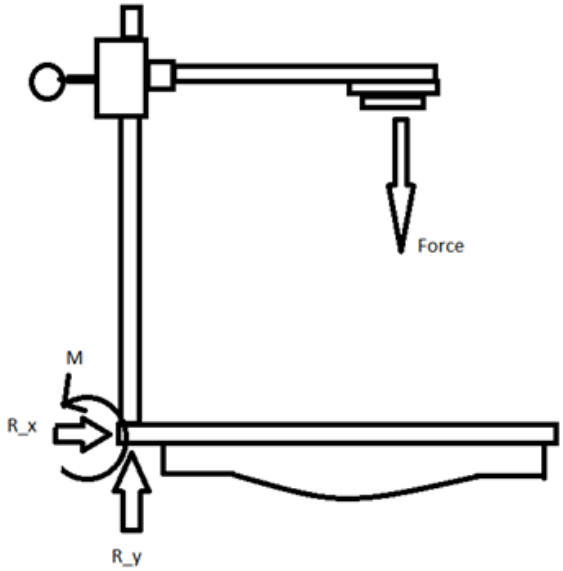


Figure 12: Moment diagram of camera mount

Because an equal and opposite force must be applied at the base of the camera mount to resist the moment that caused the tilt mentioned above, it was decided that the best design was a free-standing camera mount with a weight attached at the base.

3.4 Failure Modes and Effects Analysis (FMEA)

Table 9 shows an FMEA on what we expect to be our most severe consequence failure modes. Numbers for severity, occurrence, detection, and the RPN number showing the overall significance of the failure is shown. In the far right column is the identified action that should be taken if the function or item listed was to fail.

Table 9: Failure Modes and Effects Analysis (FMEA)

Function or Item	Failure Type	Impact	SEV	Potential Causes	OCC	Current Detection Mode	DET	RPN	Action
Puzzle pieces tracking	Pieces not tracked consistently	System no longer can register correct location of pieces	10	Different lighting conditions or pieces change color over time; noise from user's clothing or reflections off the plastic	5	None	7	350	Add a way for the user to re-calibrate the HSV color values for the pieces.
Code exception	Unhandled exception in the code that is very rare or that occurs in a piece of code we have not tested thoroughly	Would cause the whole program to crash	10	It is very rare or occurs in a piece of code we have not tested thoroughly	3	None	8	240	Lots of testing. Try to get high test coverage.
Camera mount	Camera mount breaks and fall over	Could hurt someone	10	Weight not equalized, inaccurate stress analysis	4	Stress and prototype analysis	2	80	Same as detection
Puzzle pieces	Puzzle pieces break after repeated use	Puzzle would be incomplete	4	Repeated use	4	Test breaking point of pieces	3	48	Have spare pieces
Software Installation	Incompatibility, failure to install	Would not be able to install system	10	Changes to OS, incompatibility, corrupted system	3	Failure to install	8	240	Reinstall or recompile source code

The most significant failure would be if the puzzle pieces were not tracked consistently. If this were to occur, the system would no longer be able to register the correct location of the pieces. Correct audio and visual hints would then be unable to be implemented meaning that the system would lose all functionality. This item could be divided into two because there is more than one potential cause for this failure. Some of the causes of this failure could include a person wearing the same colored shirt as the puzzle piece being manipulated, different lighting conditions, or a piece fading in color over time.

4 PROJECT MANAGEMENT AND ETHICS

4.1 Project Management

For the development of the project, all team members were assigned parallel tasks. The electrical and computer engineers separated the code into the user interface, visual tracking, and graphics/hints, and worked on their assigned sections independently. The code was designed such that progress in each section depended little on development of other sections. This gave each team member a high level of autonomy during development, and reduced delays or conflicts due to interdependent code. Version control was managed by GitHub, which also greatly aided development and eliminated repeat work. The mechanical engineering team member worked mainly independently, but design decisions were made as a group and meetings were held frequently to monitor progress and troubleshoot problems.

4.2 Ethics

Because the Assisted Puzzle Assembly project is centered on helping children in a medical environment, it was particularly important to keep ethics in mind during design. This meant keeping our design safe, listening to the feedback we got from Michael Melonis and the Anschutz Medical Campus, and being honest about the intention of our puzzle.

One of the first and foremost goals of our project was to keep the design simple and safe for children to interact with. This especially pertains to the mechanical components of the puzzle, and the puzzle pieces in particular. In order to keep what we design safe we followed the standards outlined in Standard ASTM F 963-11 (Standard Consumer Safety Specification for Toy Safety) [2]. For example, toys cannot have sharp edges or contain toxic components, so we carefully designed our pieces to have rounded edges, built them from a non-toxic and relatively soft plastic, and painted them with certified non-toxic paint. In addition, it is specified in ASTM F 963-11 that all toys intended for children under 36 months of age are subjected to the requirements of 16 CFR 1501. 16 CFR 1501 (Size Requirements and Test Procedure) specifies that no children's toy article intended for use by children is to be able to fit within the "Small Parts Cylinder" shown in Figure 13. This requirement was taken into consideration for the design of the puzzle pieces.

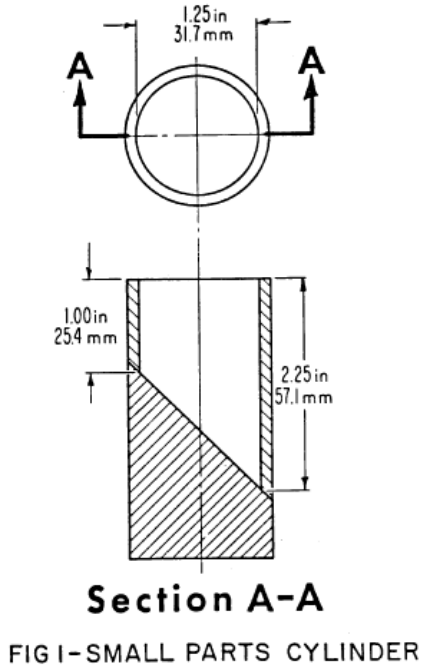


Figure 13: Small parts cylinder

Because our product is to be used in a medical facility, we made sure it was reviewed by the medical personnel that will be using it. As our project developed, we stayed in contact with Michael Melonis and the Anschutz Medical Campus to get feedback on whether the project was on the right track, and whether changes needed to be made.

Lastly, it is important that we remain honest with ourselves and with any users as to what the purpose of our puzzle is. For instance, our final goal is not to cure any diseases or disabilities, but to provide a tool to physicians and children with developmental delays that can assist them with repetitive practice of puzzles. We will make no claims that our system serves any purpose other than that, or that it contains any functionality that it currently does not.

5 BUDGET, MANUFACTURABILITY, AND MARKETABILITY

5.1 Budget

The project was sponsored by Agilent Technologies, the Electrical and Computer Engineering department at CSU, and the CSU Honors program. This funding was fully sufficient, and the project came out under budget. Budget details can be found in Appendix B.

5.2 Manufacturing

This product has been developed with manufacturability in mind, and mass manufacturing would be expected to dramatically reduce the overall cost of the product.

All electronic components of the system are Commercial Off-The-Shelf (COTS) components. While a full computer setup was purchased to aid in the development of the prototype, in the future the software needed to run the system could simply be installed on any personal desktop or laptop setup, provided it runs a compatible version of windows and has sufficient computing power. In this system, only the camera and the extra monitor need be purchased. A simple USB camera can be used, and because the monitor will only be showing simple shapes, a very cheap monitor may be used; in a large scale manufacturing situation, a very simplified graphical display could be used instead of a full commercial computer monitor.

Most of the mechanical components must be custom made, but they are very simple and could be easily and cheaply made at a large scale using plastic molds and standard plumbing piping.

5.3 Marketability

The project is currently designed to be used by clinicians working with children with developmental delays. If it proves effective, it may be expanded to use by people of various ages and levels of developmental delay, or with people with brain damage due to stroke, injury, or similar. If kept to clinical use, the market for the product will remain limited. However, there is reason to believe that this product, once developed further and rigorously tested, could be used in a home environment by people who are not trained professionals - for instance, by the parents of children with developmental delays. This would greatly increase the marketability of the product, as well as provide even more opportunities for users to benefit from the product in a comfortable setting. Further development of the project, including the development of a greater variety of games and functionalities, would also increase its marketability.

6 FUTURE WORK

Future work can involve both improvement of performance and addition of new games and features.

6.1 Performance

Some improvement of tracking performance will likely be necessary, especially if new games are to be added. While the implemented color tracking algorithm is effective for its purpose, it is limited in its ability to respond to human behavior that is outside of its expected scope. For instance, if a puzzle piece is mostly or completely occluded by a hand, it will not be tracked. This is an inherent flaw of camera tracking, but there may be algorithms that can reduce the impact that this has on game play. A more important issue is recognition of orientation and shape. One of the benefits of color tracking is that it is immune to changes to the object shape, but the downside of strict color tracking is that the tracker cannot tell whether or not the piece has been oriented properly when placed. Currently, if the averaged center of mass of the colored shape is in the correct location, then the piece will be recorded as successfully placed, whether or not it is actually oriented correctly and fit into its grooved slot. Some additional algorithms for recognizing shapes, or even an auxiliary tracking method may need to run alongside the color tracking in order to identify the correct placement of shapes.

Improvement to the scorekeeping methods would also be of great benefit. The system for recording and displaying scores is currently rudimentary, involving mainly writing text data to files and reading it

straight back in. The system could be improved by parsing text file data to create plots of performance over time, or other visual displays.

Finally, the current system for displaying graphics on the game board, while convenient and sufficient for basic shapes, has proved relatively limited for more complex shapes and requires more hardcoding than is desirable. A new method of displaying graphics would allow for greater extensibility.

6.2 New Games and Features

Variety in games and game types will help make the system more capable of working different skills and keeping its users entertained.

Some additional game possibilities are: block puzzles, snake, stacking objects, Simon Says, sequences (e.g. place pieces in a particular order), mazes, and jigsaw puzzles. However, the options are very open; any game that can be imagined to be compatible with webcam tracking and a non-touchscreen monitor game board can be designed and implemented. Each game may include multiple levels of difficulty.

For each game type, additional variety could help customize game play to the user. For instance, knob puzzles could be made with truck/car shapes, or animal shapes depending on the user preference, and the system could even be adaptable to allow the addition of new pieces - for instance, a child could use their favorite toy as a game piece.

A greater variety of visual and audio hints and feedback would also be a great addition. Audio hints that are specific to game play (such as referring to the piece being placed) would be particularly useful. There could also be the addition of data management features, such as user profiles, to help keep track of additional information from all the games, performance, levels of difficulty, trouble areas, etc. This would allow greater customization of game play for the individual user.

7 CONCLUSION

Our goal in this project was to develop a system to aid and encourage children with developmental delays to complete basic puzzles, and reduce the need for constant adult supervision. This was achieved by using color tracking to monitor the child’s actions and trigger visual hints and auditory feedback.

Because this was a new project, we had a great deal of freedom in the conceptual design of the system. The final design was chosen after extensive research on current educational technologies and available resources, and weighing of the requirements. It was decided to go with the most extensible system possible, at the risk of increased cost and development time, in order to keep the project open for further development and expansion. The prototype was completed successfully, and will be loaned to Anschutz medical center for the duration of the summer for further testing.

References

[1] *Typical Properties of Cast Acrylic* [Online]. Available: http://www.professionalplastics.com/professionalplastics/content/castacrylic.pdf

[2] *Standard Consumer Safety Specification for Toy Safety*, ASTM F963-11, 2011

Acknowledgments

Thanks to *Assistive Technology Partners*, a program of the Department of Rehabilitation Medicine at the University of Colorado Denver, Anschutz Medical Campus, for the project request and continued assistance.

Funding from Agilent Technologies is gratefully acknowledged.

Appendix A: Abbreviations

BGR - Blue, Green, Red
CAD - Computer-Aided Design
CNC - Computer Numerical Control
COTS - Commercial Off-The-Shelf
CSU - Colorado State University
FEA – Finite Element Analysis
FMEA - Failure Mode and Effects Analysis
GUI - Graphical User Interface : A type of user interface that allows the computer user to visually interact with software (e.g. through a display window) rather than solely through text (e.g. command line)
HSV - Hue, Saturation, Value
IDE - Integrated Development Environment : A program that aids in writing code (the programming equivalent of using Microsoft Word for editing text)
RGB - Red, Green, Blue

Appendix B: Budget

Agilent Technologies, a premier technical measurement company that provides measurement devices for electronics, chemical analysis, and life sciences, has accepted the team's application for sponsorship and provided $1,000 for the development of the project through the first year.

Another $400 has been received from the Electrical and Computer Engineering department of CSU through the Senior Design program, for a **total budget of $1,400.**

All project expenditures are shown in Table 10.

Table 10: Expenditures to Date

Item	Cost Per Item	Quantity	Total Cost
Dell Desktop PC	586.98	1	586.98
HDMI to DVI cable	7.49	1	7.49
speakers	12.99	1	12.99
Lifecam webcam	41.55	1	41.55
Monitor	128.81	1	128.81
Desk mat	36.9	1	36.9
Acrylic	6.3		6.3
Acrylic	16.13		16.13

Acrylic	20.39		20.39
Fasteners	5.42		5.42
Fasteners	4.82		4.82
Fasteners and Hex key	11.52		11.52
Aluminum Rail Fittings	9.53 and 9.19	1 of each	25.27
Corrosion-resistant pull ring	14.07	1	14.07
Sainsmart ABS Filament	38.97	1	38.97
Metal	15.11		15.11
Paint, brushes, and foam	17.51	7 paints, 1 set of brushes	17.51
		Total:	990.23

All software used was open-source (OpenCV) and/or provided free of charge by the CSU engineering department (Visual Studio 2012). Final expenditures were below budget, with a **surplus of $409.77**.

Appendix C: User Manual

C.1 Hardware

C.1.1 Hardware Requirements

The hardware components required to run the Assisted Puzzle Assembly System are:

- Desktop Computer
    - Must have Windows OS. Windows 7 recommended. Other Windows versions have not been tested, but Windows 8 should work as well.
    - Should have similar performance to an Intel Core-i3 platform to achieve a high enough frame rate for smooth gameplay.
    - Current system uses: DELL i3847-5077BK Desktop PC Intel Core i5 4440 (3.10GHz) 8GB DDR3 1TB HDD Capacity Windows 7 Home Premium (64Bit)

- Monitor for gameboard
    - Must fit plastic gameboard dimensions and be able to lay flat.
    - Current system uses Dell - E2014H 19.5" LED Monitor
    - Should be 1600x900 resolution or else drawing of gameboard will not match plastic board.

- Second monitor to run GUI

- Microsoft LifeCam Cinema Webcam
    - Part #: H5D-00013, H5D-00001
    - Other webcams may work if they fit into the camera stand and have a wide viewing angle (~73 degrees). However, the Microsoft LifeCam Cinema has automatic contrast/brightness/white balance features that produce consistent and bright colors which makes it easier to track the puzzle pieces in software.

- Camera Stand:
    Camera stand (Figure 14) is custom made to fit the current webcam model described above.

Figure 14: Camera Stand

- Puzzle boards and puzzle pieces:

Puzzle boards and pieces (Figure 15) must be custom made and match the dimensions of the monitor

Figure 15: Puzzle board and pieces

- Speakers for audio

- Mouse and keyboard

- Black desk mat

C.1.2 Hardware Setup

Plug in the monitors, webcam, speakers, mouse and keyboard to desktop PC. Place the gameboard monitor flat on the black desk mat. (The black desk mat is not required, but it ensures that the table color will not interfere with the color tracking. It also provides a consistent brightness background which gives a more consistent auto-brightness setting.)

**IMPORTANT: The Dell 19.5" LED monitor must be placed upside down. i.e. the silver Dell logo is closest to the camera stand. If left right side up, the sharp viewing angle from looking at the horizontal screen will cause the monitor image to disappear.

Make sure the display is flipped on the upside down gameboard monitor by going to the Windows "Screen Resolution" Control Panel utility and selecting *Orientation: Landscape (flipped).*

Place the clear puzzle boards on the monitor. **Important note: at no time should there be more than 60 pounds on top of the puzzle boards.**

Assemble the camera stand if needed and place it looking down over the puzzle board, as shown in Figure 16.

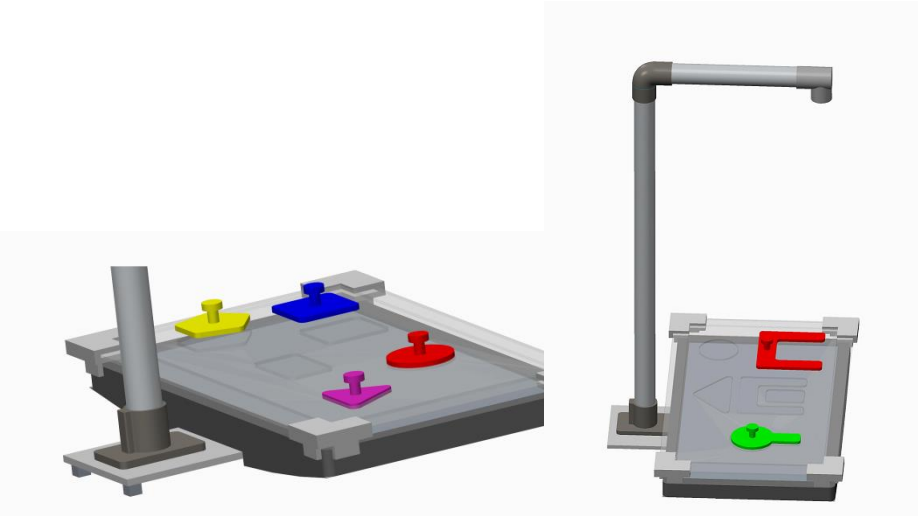


Figure 16: Camera stand position

Part of the stand should fit under the monitor so more area in front of the puzzle is visible in the camera view than behind it. The camera view should look something like Figure 17:

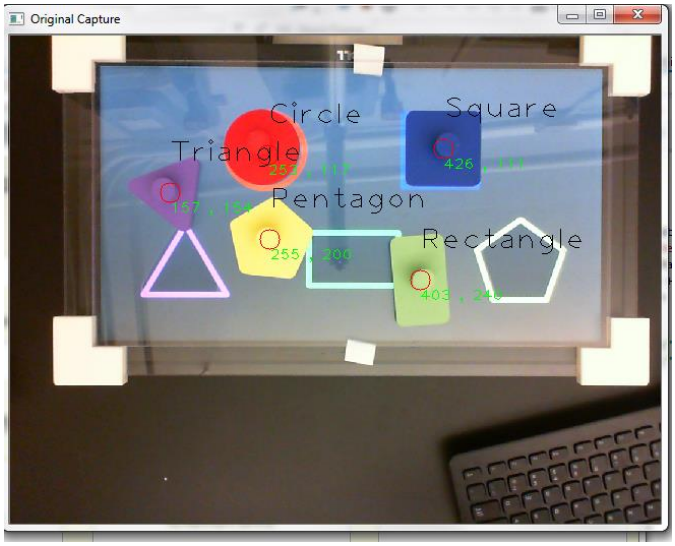


Figure 17: What the camera view should look like

C.1.3 Changing the Camera

Any standard USB webcam is compatible with the code. If the Lifecam is not used, you do not have to run CameraPrefs or adjust the white balance (discussed later) before beginning a new session. A new camera may require modification to the plastic camera holder on the camera stand.

You may have to adjust the settings of the camera for accurate tracking. For instance, a camera with Autofocus turned on may have trouble focusing on the board. The color profiles may also have to be adjusted for accurate colors. These adjustments will vary greatly depending on the camera purchased, so it is recommended to spend some time testing a new camera and finding the best settings before using it to play.

C.2 Software

C.2.1 Installing OpenCV

OpenCV is a free, open source software library with many powerful image processing tools. It is very important that OpenCV is properly installed in order to use the Assistem Puzzle Assembly system.

There a good instructions on the OpenCV website on how to install the pre-built libraries that are provided. Go to opencv.org/quickstart.html and click on the "Installation in Windows" link. Follow the instructions found in the "Installation by Using the Pre-built Libraries" to install OpenCV.

If you plan on developing the Assisted Puzzle Assembly software further, you will need to set up Visual Studio to compile with the OpenCV libraries. To find the instructions on how to do this, go back to opencv.org/quickstart.html and click on the "Using OpenCV with Microsoft Visual Studio" link. After following the instructions found on that page, you should be able to compile the software's source code.

C.2.2 Installing Assisted Puzzle Assembly Software

If you wish to use the Assisted Puzzle Assembly software as it is (without seeing or developing the source code), you must first install OpenCV using the instructions in the section above. Once OpenCV is properly installed, the executable to run the software can be found at https://github.com/asimo42/PuzzleAssembly. The executable is called ConsoleApplication4.exe. Any additional files needed to play the game such as game files or CameraPrefs.exe can be found on github as well.

C.2.3 Obtaining Assisted Puzzle Assembly Source Code for Development

The source code for this project is open available to see and download. It is hosted at https://github.com/asimo42/PuzzleAssembly. This includes the source code and all the file required by Visual Studio to open it as a Visual Studio project. After downloading all the files, open Visual Studio and select Open Project. Open "ConsoleApplication4.vcxproj" to open the Visual Studio Project and begin developing!

*Note: You may have to adjust your project properties (Project->Properties) if your OpenCV paths differ from what was suggested.

C.2.4 Installation Folder

The code will require access to support text files, including the input game files, help files, and patient performance. The paths to these text files are hardcoded into the code, and are all relative to the location of the executable file. For instance, the currently folders used are:

'C:/PuzzleAssembly/Executable' -- for the executable files
ExecutablePath + '/../' -- for the input game files and help files
ExecutablePath + '/..'PatientPerformanceData' - for the user performance data
ExecutablePath + '/..'Sounds' -- for the audio sounds to be played
ExecutablePath + '/../CameraPrefs' -- all camera prefs files

If desired, these paths can be changed at any time by going to Functions.h and changing the 'hardcoded file paths' in the Constants class. If you create a visual studio project or Github directory to work on the code, which will normally involve moving files around, make sure you update these paths.

C.3 Using the Assisted Puzzle Assembly System

* Make sure that all hardware components (webcam, monitor) are properly connected before beginning.

C.3.1 Camera Settings

Run CameraPrefs.exe by double clicking on the desktop shortcut (desktop shortcut icons can be seen in Figure 18). This will turn auto-focus off and modify some other camera settings.

The camera settings set by CameraPrefs.exe can be modified by opening CamerPrefs.xml and editing the fields. There is a bug with running CameraPrefs.exe where auto white balance is always turned off even if it is set to "true." It is recommended that auto white balance be turned back on. If the colors appear saturated or too white, then auto white balance is probably off. To turn it back on and to manually adjust all other camera settings, the Microsoft LifeCam software can be used. Open the software and click on the little arrow on the mid-right of the screen. Then click on the gear icon. Then click camera settings to modify the camera settings.

Figure 18: Icons

C.3.2 Launching the Program

The program can then be launched by double clicking on the desktop shortcut, or by clicking the .exe file in the install directory. Launch will take you directly to the main GUI.

C.3.3 Calibration

A calibration system is provided to specify the colors that are to be tracked, and the destination location of those pieces. Calibration is necessary when the system is first set up, any time the lighting conditions change, or the camera stand is moved in relation to the game board. Every game board must be calibrated individually. Calibration settings are specific to individual game boards and can be saved for future use.

Clicking on the 'Calibration' button on the main GUI will begin calibration. The user will be guided through calibrating the color of each individual piece using its HSV range. A filtered black and white image will show the filtered colors (the color being tracked will show up as white) based on the selected HSV range. Figure 19 shows an example of a piece that has been correctly calibrated.

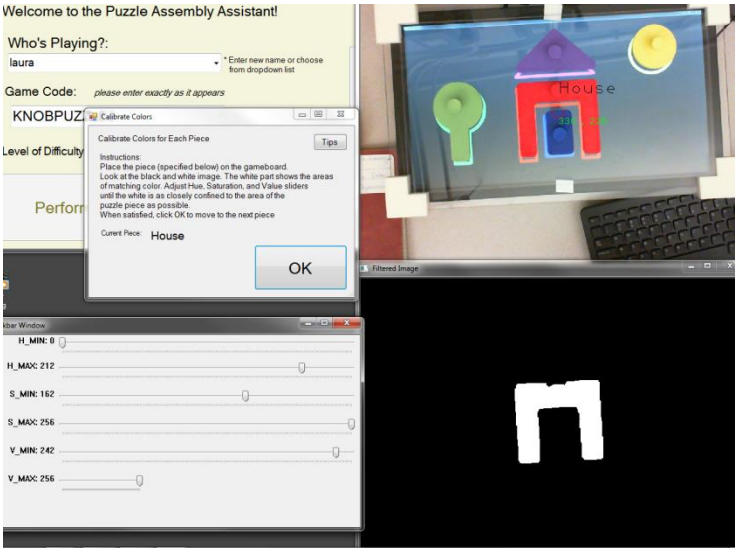


Figure 19: Color calibration - the red piece is correctly calibrated

When all colors have been calibrated, the user will place all puzzle pieces in their assigned locations, and the program will automatically search for and memorize those locations.

Suggested process for calibrating colors:

1. Put all pieces at their dedicated spots on the game board.
2. Click ‘Calibration’ on the main GUI and follow the prompts until the color tracking windows pop up. The popup menu will specify a puzzle piece - that is the piece you are calibrating now.
3. Start with max and min values far apart. Make sure the piece shows up as white on the filtered image - this means the camera can see it. There may also be a lot of white everywhere else at this point.
4. HUE: adjust this first. Think of the hue values (0-256) as being like a rainbow that you are selecting a small portion of. Hue doesn't change much in lighting, so you can get this interval pretty tight.
5. SATURATION: Adjust this next. The pieces will generally have pretty high saturation (intensity of color) but not always. Keep this interval wider than hue to accommodate inconsistent lighting.
6. VALUE: Value can change significantly depending on whether a piece is in shadow or light. As much as possible, try to keep this range large (the full range is best, but a half range is pretty good too) and filter mainly through hue and saturation.
7. Iteratively tighten or widen the HSV ranges until only the selected piece is white on the filtered image (a little bit of noise is okay) and the live camera feed shows crosshairs on the piece.
8. Click ‘next’ to repeat with the rest of the pieces
9. Follow the remaining prompts to let the program automatically find the destination locations of the pieces.

When you are done, you will be asked if you want to save your calibration settings. If you say yes, then those settings will be saved for future sessions. If you say no, those settings will still be in effect as long as the program is open, but will revert to old settings when the program is closed.

C.3.4 Calibration and Color Tracking Tips

It is very important that all of the puzzle pieces are able to be tracked consistently. All other features of the game such as giving visual hints, detecting when a piece has been placed correctly, and recording game statistics rely on the pieces being tracked consistently. If you are having issues with game play/tracking the pieces, here are some tips to get more reliable tracking:

- If game play seems off, or it seems like the camera is not tracking properly, try recalibrating. Because calibration settings are saved, you can tweak the calibration settings without having to redo everything each time.

- If the game no longer detects when a piece is placed correctly or when the game is completed, try recalibrating. Either one or more pieces was no longer being tracked, or the camera stand or game board were moved and the positions of each piece need to be resaved during calibration.

- If there is an orange or yellow piece on the board, and the user has light skin, put the user's hand on the game board while calibrating those pieces to make sure it gets filtered out. In some lighting conditions the skin can be very orange or yellow.

- The color red may be tricky to calibrate because the hue rainbow starts and ends on red (so to get all the red, you have to have the sliders cover the whole range). Sometimes you may have to use the full hue range, and filter out the other colors using saturation and value. You may also have to calibrate this piece more often, as subtle lighting changes can cause it to switch from one side of the split spectrum to the other.

- The color tracking works best indoors where there is consistent artificial lighting. Avoid areas with direct sunlight as this can cause pieces to appear too saturated and as the sunlight changes, the tracking may have to be recalibrated.

C.3.5 Running a Game

Running a game is done from the main GUI. The steps are as follows:

1. Select a user. A dropdown menu will automatically be populated by past users. You may create a new user simply by typing in a new name. All game result data will be tied to the selected user.
2. Type in or select the name of the puzzle you would like to play. A dropdown menu will automatically be populated by available games. If no games show up, check your software install settings (Section CC.2: Software) to make sure you have your puzzle files in the correct location.
3. Select a level of difficulty.
4. Calibrate the camera if necessary (Section C3.3: Calibration). This will likely be necessary if the game has been moved to a new location or the lighting situation has changed.

5. Hit "Run Game." This will initialize the camera tracking and gameplay.
6. Play the game. Hints will be given based on the user's perceived actions. The game will end when every piece is placed, or when the "Stop Game" button is hit.
7. When prompted, choose whether or not to save game results.
8. Performance information can be found by hitting "Performance". This will show data for every game played in that session.

C.3.6 Displaying Results

The results for each game will be shown after the game is completed.

For more, hit the "Performance" button. All results from the current session will be shown.

To see old data, go to the Performance section, then hit 'look at old data'. Select the user and the game you want to see results for, and the program will find what dates are available. Select one or more dates to see the results from those dates.

C.3.7 Changing Sound Effects

The current sound effects are guitar noises aimed to be exciting and encouraging when a piece is placed correctly. If desired, the sound effects can be changed without modifying the code. In the PuzzleAssembly project directory, navigate to the Sounds folder. This holds the sounds used in the game. By switching out a file for a new sound file with the same name, the sound will be changed. For example, to change the sound played when the game starts up, change guitar_start.mp3 to a different sound file also named guitar_start.mp3. guitar_end.mp3 is played when the game is completed successfully. guitar1.mp3 - guitar7.mp3 are short sounds played randomly when a piece is placed correctly. Any other modifications to how sound is played will require modifying source code.

C.4 Software Development

C.4.1 Source Code

The source code for the project can be found in the public GitHub repository here:

https://github.com/asimo42/PuzzleAssembly

C.4.2 Organization

The code is primarily composed of Windows Applications forms (which run the GUIs), data classes, and various functions/algorithms. All action within the program is triggered by user interaction with the GUI through a collection of callbacks. The main data classes hold all puzzle piece, game board, and

scorekeeping information. Input files are required to load data for game boards, and output files with performance information are created; all these files will be in the program folder.

The best way to get introduced to the code would be to start with MainGUIForm.h. All user interactions with the main GUI have a corresponding callback function there. Look at the ‘runGameButtonClicked()’ callback - this function is called when the user tries to start a new game. Follow this function step-by-step through its various function calls. It will take you through most of the main components of the code, including loading a game board, running the tracking and hint algorithms, ending a game and more. When you are familiar with how the game is run, you may wish to repeat the process with ‘calibrateButtonClicked()’ and ‘performanceButtonClicked()’.

Code has been commented in detail, and function names were made as descriptive as possible.

C.4.3 Major Components

The GUI is constructed of the following Windows Application Forms. Most functions and processes are initiated from within these forms:

- **MainGUIForm.h :** This form is the main GUI by which the user starts and stops games. This is also the 'entry-point' of the program. All callbacks for users selecting items on the main GUI are handled here. The 'global' class variables here hold information that is consistent through different 'phases' of the program, e.g. a knobpuzzle loaded here can be transferred to calibration, or to a running game, etc, and all game results are stored in these variables. This is because this is the only form/class that never closes or ends throughout a session.
- **displayResultsForm.h :** This form displays results either from the current session (default) or from loaded data. Loaded data is pulled in via a selectOldResultsForm.
- **selectOldResultsForm.h :** This form allows the user to select old data for display. It finds what dates are available for data based on the entered username and game, and the expected location of the data. The form will return the selected player, game, and dates (not the data). It is called from the DisplayResultsForm class when the user clicks the ‘look at old data’ button
- **CalibrationMainPrompt.h :** This form guides the user through the general calibration process. It is created when the user clicks the ‘Calibrate’ button on the main GUI. It launches the ColorCalibrationForm for the user to calibrate colors. It then uses the newly calibrated colors and launches OpenCV (via CalibrationTracking) to record the destination locations of each piece.
- **ColorCalibrationForm.h :** This form guides the user through the color calibration process. It launches OpenCV tracking, and then steps through each piece in the puzzle as the user calibrates.

The following classes hold the gameplay, puzzle board, and scorekeeping data:

- **GameBoard.h/cpp:** ‘Gameboard’ and ‘Knobpuzzle’ classes are defined here. These classes hold the data for each individual gameboard. There is a GameBase class which has all the basic information in it,and classes for individual game types can derive from it. Only the KnobPuzzle has been developed this year. An instance of the KnobPuzzle class contains all information as to

the name, shape, location, placement, etc. of each puzzle piece for a given board. This instance will be passed all around through the program, to be used by tracking, scorekeeping, and the GUI

- **PuzzlePiece.h/cpp :** This class hold the data for an individual puzzle piece. This class is currently tailored for the KnobPuzzle, but extensions could be made. Basic information included is the shape of the piece, its color, its destination coordinates, and the time at which it was placed.
- **TrackedPiece.h/cpp**: The tracked piece class contains data and methods for a piece being tracked. An instance of the class is instantiated for each piece being tracked which is then added to a vector that holds all the pieces of a game. This class is virtually identical to the PuzzlePiece class, except it uses unmanaged c++ code (compatible with the color tracker), where PuzzlePiece is a managed class (compatable with the GUI and everything else). Functions exist to translate between tracked and puzzle pieces.
- **ScoreKeeping.h/cpp :** Contains GamePlayed, GamePlayedData, and ScoreKeeping classes. These classes record and compile the performance data for each session (i.e. users, games, times pieces were placed). They also control file IO for saving performance data.

The following classes and files contain the various color tracking, hint, and shape drawing algorithms and processes, and any other required functions.:

- **RunTracking.h/cpp :** This class controls the operation of the color tracking and hint triggering algorithms. It starts OpenCV running, monitors/controls tracking, gathers time data, and shuts OpenCV down once the game is completed or stopped. An instance of this class is created in "Functions.cpp - initializeTracking()" when the user hits the Run button on the GUI, and it creates a GamePlayed^ instance that holds the performance information from the game.
- **Tracking.cpp:** This contains functions of the RunTracking class involved in the actual color tracking algorithm. trackTrackedPiece() contains the color tracking algorithm that is run on each piece being tracked each frame. This file also include the timers used to periodically check for piece movement and to flash the shapes on the screen. startTrack() contains a while(1) loop that continuously runs through the color tracking algorithm until the game is stopped.
- **CalibrationTracking.h/cpp:** This class controls the operation of OpenCV for the calibration process (both color and location). Conceptually very similar to Tracking.cpp
- **Shape.h/cpp:** The Shape class contains the functions for drawing puzzle piece shapes on the monitor. Each shape has its own dedicated function, there is also a function that will take in a TrackedPiece and draw the corresponding shape.
- **SoundEffectPlayer.h/cpp:** The SoundEffectPlayer class uses the Microsoft Directshow API to play audio files. An instance of this class is used to play the sounds effects during the game.
- **Functions.h/cpp :** Any general or miscellaneous functions that are used in the program (esp. functions that are used in multiple places) are put here. 'Global' constants also go here, in a class called Constants. These variables can be accessed as Constants::VariableName.

C.4.4 Adding New KnobPuzzles

* It is assumed that the puzzle board and pieces have already been constructed.

Adding a new knobpuzzle requires adding the new graphic shapes to the code, then creating a text file input that is used to load the game.

Adding new shape graphics requires adjustment of the code. Currently, the color of the shape drawn is hardcoded. This should be adjusted in future iterations of the system to be pulled in through the text file with the rest of the shape data.

Shape drawing is primarily done through series of 'if' statements in the code to handle the different shapes. These 'if' statements occur at a few points in the code; in the Gameboard.cpp at KnobPuzzle::ParseShapeInformation(), in Functions.cpp in puzzlePieceToTrackedPiece() and trackedPieceToPuzzlePiece(). As well, the PuzzlePiece class may need to have variables added to it (currently it has radius, width, length, etc) and function to set those variables. Follow the pattern and format of the other pieces, and add in the new piece. The shape will correspond to the piece name.

A function must also be created in the Shapes class to draw the new shape. Shapes are drawn using functions provided by OpenCV. It provides functions to draw simple shapes such as a circle or rectangle and another function to draw any polygon as long as the vertices of that polygon are known. Examples of how to use each of these functions can be found throughout Shape.cpp. The Shape class is used to make dedicated functions for each shape that needs to be drawn. These functions should take very few input arguments and then calculate all the information that is necessary to make a call to a draw function from OpenCV.

Create the text file as follows. Please reference the sample template in Figure 20.

KNOBPUZZLE1
Difficulty 1
LOC 433 108 COLOR 107 151 0 123 256 256 Square 958 128 238
LOC 342 217 COLOR 35 54 0 51 198 256 Rectangle 650 510 287 175
LOC 257 112 COLOR 167 156 0 186 256 256 Circle 510 244 125
LOC 170 228 COLOR 123 23 0 146 188 256 Triangle 255 490 266
LOC 513 224 COLOR 16 75 135 28 256 256 Pentagon 1329 465 173

** note: LOC xloc yloc COLOR Hmin Smin Vmin Hmax Smax Vmax name (shapedrawing specifics)
*** Shapedrawing data will vary as follows:
Circle: middle_x, middle_y, radius
Rectangle: corner_x, corner_y, width, height
Square: corner_x, corner_y, width
Triangle: top_x, top_y, length
Pentagon: top_x, top_y, length

Figure 20: Sample knobpuzzle input file

C.4.4.1 Steps:

1. Copy an old knob puzzle input file to use as a template.
2. Replace title and level of difficulty (the difficulty of the game board is not currently used in the code). Replace all game piece names with the new game piece names add or remove lines as

necessary. * Piece names are used to identify shape in code; arbitrary piece names cannot be used unless they have been coded for a shape

3. Place shape drawing variables in the order specified, or have been newly programmed.
4. If new shape types were added, please add the type and order of variables to the reference at bottom. Everything below the dashed lines is comments.
5. Name the file as GAMENAME.txt, then make a copy named GAMENAME_Default.txt. If something happens to the main file during calibration, the default file serves as backup. Place both files in the folder holding all game input texts.

**The values for the location data and HSV for the pieces do not matter (though there must be the correct number of numbers there) - they will be reset the first time the new puzzle is calibrated

Now select the new puzzle on the main GUI (it should appear in the dropdown) and calibrate. The game is now ready to go.

C.4.5 Adding new game types:

Please be aware that all development focus was on the KnobPuzzle. New games must be added directly to the source code. Some cases of 'hardcoding' do exist, by which the knobpuzzle is assumed to be the only game available. When adding new games, you will need to step through the game flow to make sure that the knobpuzzle class is not being used by default. Add switch statements where necessary to change declarations/code paths.

Suggested process for adding new games:

To start, go to GameBoard.h and create a new class derived from the GameBoard class. Put all required information and functions in there, including puzzle file IO. Look at the KnobPuzzle to get an idea of what needs to be there.

Because the tracking style will probably be different for the new game, a new tracking class will probably have to be created. That is, create different versions of RunTracking and Tracking.cpp to handle the new algorithms. A new shape drawing class may also have to be created, and so on.

Check the current scorekeeping classes to see if they are compatible with the new game type. Add variables as needed, or create new class types tailored to the new game (will be necessary if game is not based on the placement of pieces).

The GameBoard class from which game classes inherit includes a gameType variable. From the GUIs, this variable can be used to determine which type of game is being played, and take the appropriate course of action. For instance, before initializing a game, add a switch statement to check the game type and then funnel it into the proper tracking algorithm. If switch statements are wisely placed, then the GUI forms should require minimal changes for new games.

* Note: in MainGUI.h, the 'currentpuzzle' variable is initialized as a knobpuzzle. 'CurrentPuzzle' is used as a 'global variable' so that the various callbacks can access the current game. This structure must be modified to allow for different types of games.

Long term, a better, smoother structure for adding new game types is recommended.

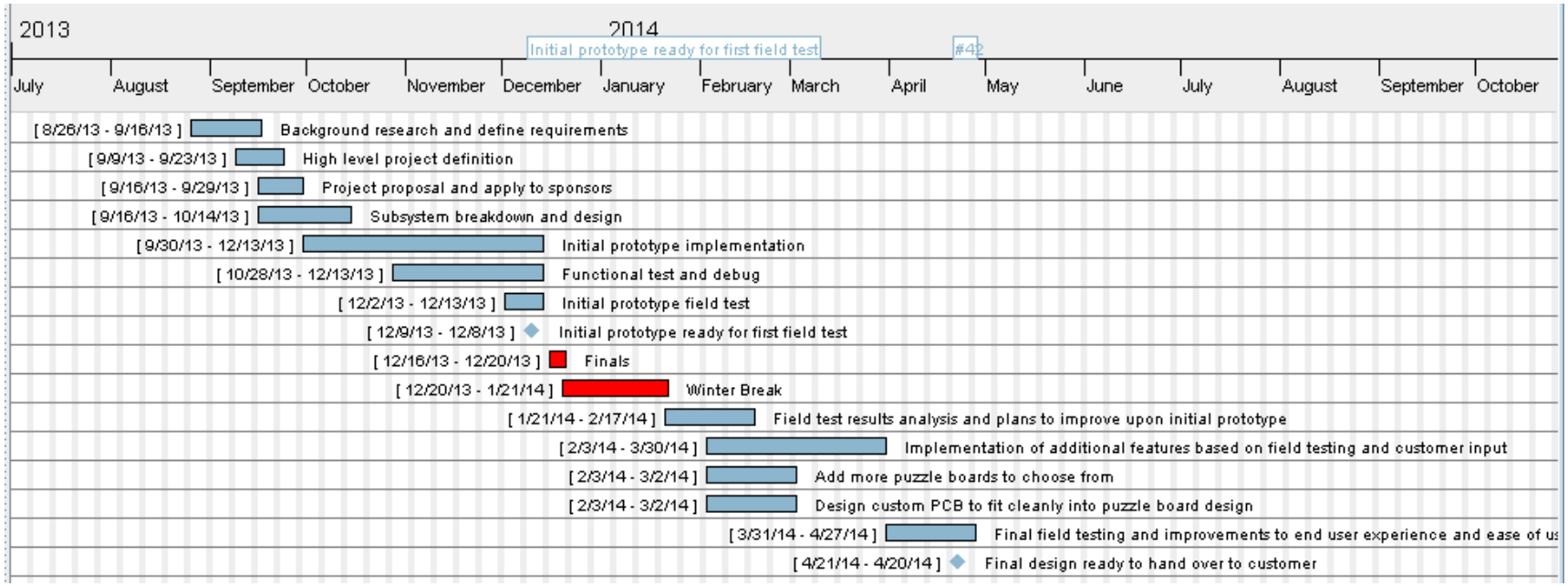
C.4.6 Known Bugs

- The CameraPrefs.exe utility always turns auto white balance off even when it is set to “true” in the xml file. Auto white balance must be turned back on manually using the Microsoft LifeCam software.

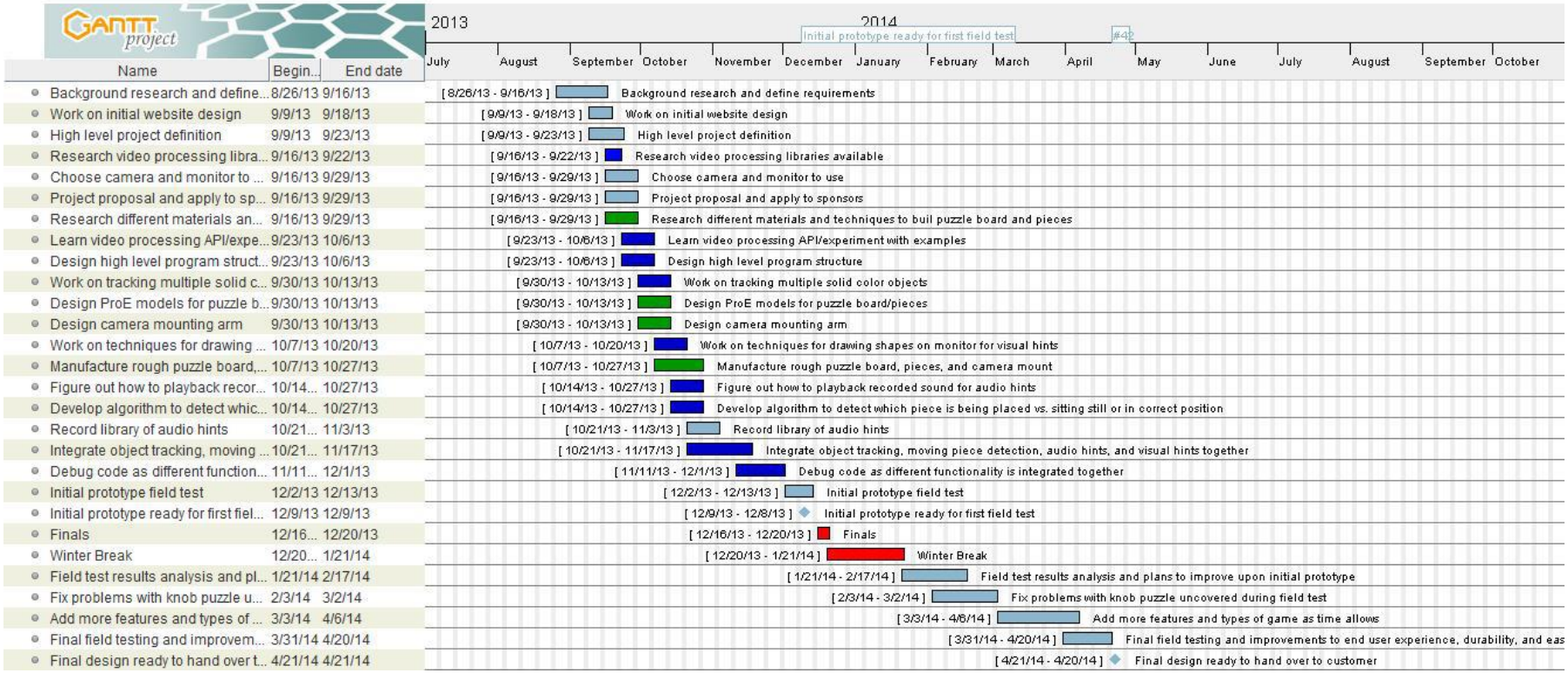
- There was an occasional unhandled exception thrown from the openCV code somewhere in color.cpp. The cause of this was never determined. This has not been seen in a while and it is unclear if the issue was solved or not. If an unhandled exception is thrown when starting a game, try restarting the whole program and it should go away.

Appendix D: Timelines

Version 1:

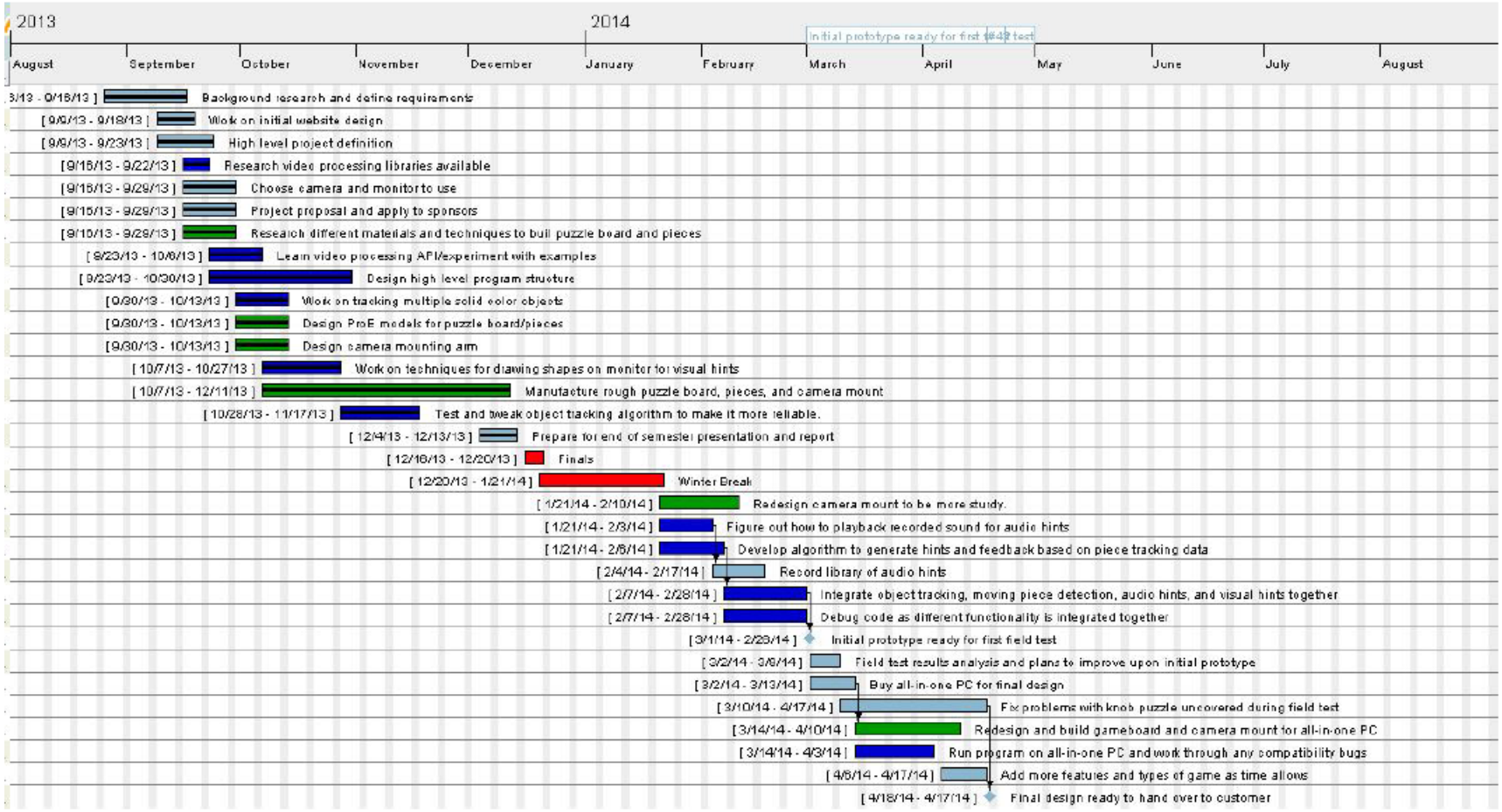


Version 2:



Software development
Mechanical
Other
Time off

Version 3:



Software development
Mechanical
Other
Time off

Appendix E: Source Code

This appendix contains all source code written for the project. Files automatically generated by Visual Studio for compilation are not included.

MAINGUI.H

```
/*
	This form is the main GUI by which the user starts and stops games. This is also the 'entry-point' of the program.
	All callbacks for users selecting items on the main GUI are handled here.
	The 'global' class variables here hold information that is consistent through different 'phases' of the program;
	e.g. a knobpuzzle loaded here can be transfered to calibration, or to a running game, etc, and all game results are stored
	in these variables. This is because this is the only form/class that never closes or ends throughout a session.
*/
#include <Windows.h>
#include "stdafx.h"
#include <WinBase.h>
#include <WinUser.h>
#using <System.dll>
#include <stdlib.h>
#include  <stdio.h>
#include <vcclr.h>
#include "Functions.h"
#include "displayResultsForm.h"
#include "CalibrationMainPrompt.h"
#include "RunTracking.h"

#pragma once

namespace PuzzleAssembly {

	using namespace System;
	using namespace System::ComponentModel;
	using namespace System::Collections;
	using namespace System::Windows::Forms;
	using namespace System::Data;
	using namespace System::Drawing;

	/// <summary>
	/// Summary for MainGUIForm
	/// </summary>
	public ref class MainGUIForm : public System::Windows::Forms::Form
	{
	public:
		MainGUIForm(void)
		{
			InitializeComponent();

			// initialize all the status variables and the knobpuzzle class
			turnAllButtonsOnExceptStop();
			this->gameRunning = false;
			this->calibrating = false;
			this->sessionDataSaved = false;
			this->puzzleComboBox->Text = "KNOBPUZZLE1";                    // REMOVE FOR FINAL VERSION
			this->currentPuzzle = gcnew KnobPuzzle();
			this->ScoreKeeper = gcnew ScoreKeeping();

			//see if the results directory for the patient results data exists yet. If not, create it.
			if (!System::IO::Directory::Exists(Constants::RESULTS_DIRECTORY)) {
				// * I don't know how to error check this yet.
				System::IO::Directory::CreateDirectory(Constants::RESULTS_DIRECTORY);
```

```
                                        Console::WriteLine("MainGuiForm.h::Initialize(): Created results directory " +
Constants::RESULTS_DIRECTORY);
                                }
                                else {
                                        Console::WriteLine("MainGuiForm.h::Initialize(): Results Directory found: " +
Constants::RESULTS_DIRECTORY);
                                }
                                //******

                                // For me, CameraPrefs folder is located 2 folders above the consolepplication4.exe file
//                              System::String^ cameraExecutablePath = System::Windows::Forms::Application::StartupPath +
"/../../CameraPrefs/CameraPrefs.exe";
                                //System::String^ cameraExecutablePath = "C:\\CameraPrefs\\CameraPrefs.exe";
                                //MessageBox::Show("Attempting to run : " + cameraExecutablePath);
                                //if (System::IO::File::Exists(cameraExecutablePath)) {
                                //      MessageBox::Show("Executable file found");
                                //      System::Diagnostics::Process^ process = System::Diagnostics::Process::Start(cameraExecutablePath);
                                //}
                                //else {
                                //      MessageBox::Show("Can't find CameraPrefs.exe. Please change my file path in MainGUIForm.h ::
MainGUIform(void)");
                                //}


                }

        protected:
                /// <summary>
                /// Clean up any resources being used.
                /// </summary>
                ~MainGUIForm()
                {
                        if (components)
                        {
                                delete components;
                        }
                }

        // My Variables
        public: bool gameRunning;
        public: bool calibrating;
        public: bool sessionDataSaved;

        private: System::Windows::Forms::Label^  label1;
        private: System::ComponentModel::IContainer^  components;
        private: KnobPuzzle^ currentPuzzle;
        private: ScoreKeeping^ ScoreKeeper;

        // Visual Studio's GUI stuff
        private: System::Windows::Forms::Button^  runGameButton;
        private: System::Windows::Forms::Button^  scoresButton;
        private: System::Windows::Forms::Button^  calibrateButton;

        private: System::Windows::Forms::Button^  stopGameButton;
        private: System::Windows::Forms::Label^  label2;
        private: System::Windows::Forms::Label^  label3;
        private: System::Windows::Forms::HelpProvider^  helpProvider1;
        private: System::Windows::Forms::CheckBox^  level1CheckBox;
        private: System::Windows::Forms::CheckBox^  level2CheckBox;
        private: System::Windows::Forms::CheckBox^  level3CheckBox;
        private: System::Windows::Forms::Button^  levelDescriptionsButton;
```

```
        private: System::Windows::Forms::Label^  label5;
        private: System::Windows::Forms::ComboBox^  playerNameComboBox;

        private: System::Windows::Forms::Label^  label6;
        private: System::Windows::Forms::ComboBox^  puzzleComboBox;
        private: System::Windows::Forms::Button^  helpButton;
        private: System::Windows::Forms::Label^  label4;

        protected:

        private:
                /// <summary>
                /// Required designer variable.
                /// </summary>


#pragma region Windows Form Designer generated code
                /// <summary>
                /// Required method for Designer support - do not modify
                /// the contents of this method with the code editor.
                /// </summary>
                void InitializeComponent(void)
                {
                        this->runGameButton = (gcnew System::Windows::Forms::Button());
                        this->label1 = (gcnew System::Windows::Forms::Label());
                        this->scoresButton = (gcnew System::Windows::Forms::Button());
                        this->calibrateButton = (gcnew System::Windows::Forms::Button());
                        this->stopGameButton = (gcnew System::Windows::Forms::Button());
                        this->label2 = (gcnew System::Windows::Forms::Label());
                        this->label3 = (gcnew System::Windows::Forms::Label());
                        this->helpProvider1 = (gcnew System::Windows::Forms::HelpProvider());
                        this->level1CheckBox = (gcnew System::Windows::Forms::CheckBox());
                        this->level2CheckBox = (gcnew System::Windows::Forms::CheckBox());
                        this->level3CheckBox = (gcnew System::Windows::Forms::CheckBox());
                        this->levelDescriptionsButton = (gcnew System::Windows::Forms::Button());
                        this->label4 = (gcnew System::Windows::Forms::Label());
                        this->label5 = (gcnew System::Windows::Forms::Label());
                        this->playerNameComboBox = (gcnew System::Windows::Forms::ComboBox());
                        this->label6 = (gcnew System::Windows::Forms::Label());
                        this->puzzleComboBox = (gcnew System::Windows::Forms::ComboBox());
                        this->helpButton = (gcnew System::Windows::Forms::Button());
                        this->SuspendLayout();
                        //
                        // runGameButton
                        //
                        this->runGameButton->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windows::Forms::AnchorStyles::Bottom |
System::Windows::Forms::AnchorStyles::Right));
                        this->runGameButton->AutoSizeMode = System::Windows::Forms::AutoSizeMode::GrowAndShrink;
                        this->runGameButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 24,
static_cast<System::Drawing::FontStyle>((System::Drawing::FontStyle::Bold | System::Drawing::FontStyle::Italic)),
                                        System::Drawing::GraphicsUnit::Point, static_cast<System::Byte>(0)));
                        this->runGameButton->Location = System::Drawing::Point(597, 278);
                        this->runGameButton->Name = L"runGameButton";
                        this->runGameButton->Size = System::Drawing::Size(247, 142);
                        this->runGameButton->TabIndex = 0;
                        this->runGameButton->Text = L"Run Game";
                        this->runGameButton->UseVisualStyleBackColor = true;
                        this->runGameButton->Click += gcnew System::EventHandler(this, &MainGUIForm::runGameButton_Click);
                        //
                        // label1
                        //
```

```
            this->label1->AutoSize = true;
            this->label1->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 15.75F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                static_cast<System::Byte>(0)));
            this->label1->Location = System::Drawing::Point(17, 157);
            this->label1->Name = L"label1";
            this->label1->Size = System::Drawing::Size(132, 25);
            this->label1->TabIndex = 7;
            this->label1->Text = L"Game Code:";
            //
            // scoresButton
            //
            this->scoresButton->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windows::Forms::AnchorStyles::Bottom |
System::Windows::Forms::AnchorStyles::Left));
            this->scoresButton->AutoSizeMode = System::Windows::Forms::AutoSizeMode::GrowAndShrink;
            this->scoresButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 20.25F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                static_cast<System::Byte>(0)));
            this->scoresButton->Location = System::Drawing::Point(12, 306);
            this->scoresButton->Name = L"scoresButton";
            this->scoresButton->Size = System::Drawing::Size(277, 114);
            this->scoresButton->TabIndex = 8;
            this->scoresButton->Text = L"Performance";
            this->scoresButton->UseVisualStyleBackColor = true;
            this->scoresButton->Click += gcnew System::EventHandler(this, &MainGUIForm::scoresButton_Click);
            //
            // calibrateButton
            //
            this->calibrateButton->Anchor = System::Windows::Forms::AnchorStyles::Bottom;
            this->calibrateButton->AutoSizeMode = System::Windows::Forms::AutoSizeMode::GrowAndShrink;
            this->calibrateButton->Enabled = false;
            this->calibrateButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 20.25F,
System::Drawing::FontStyle::Regular,
                System::Drawing::GraphicsUnit::Point, static_cast<System::Byte>(0)));
            this->calibrateButton->Location = System::Drawing::Point(316, 306);
            this->calibrateButton->Name = L"calibrateButton";
            this->calibrateButton->Size = System::Drawing::Size(256, 114);
            this->calibrateButton->TabIndex = 9;
            this->calibrateButton->Text = L"Calibrate";
            this->calibrateButton->UseVisualStyleBackColor = true;
            this->calibrateButton->Click += gcnew System::EventHandler(this, &MainGUIForm::calibrateButton_Click);
            //
            // stopGameButton
            //
            this->stopGameButton->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windows::Forms::AnchorStyles::Bottom |
System::Windows::Forms::AnchorStyles::Right));
            this->stopGameButton->AutoSizeMode = System::Windows::Forms::AutoSizeMode::GrowAndShrink;
            this->stopGameButton->Enabled = false;
            this->stopGameButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 24,
static_cast<System::Drawing::FontStyle>((System::Drawing::FontStyle::Bold | System::Drawing::FontStyle::Italic)),
                System::Drawing::GraphicsUnit::Point, static_cast<System::Byte>(0)));
            this->stopGameButton->Location = System::Drawing::Point(597, 94);
            this->stopGameButton->Name = L"stopGameButton";
            this->stopGameButton->Size = System::Drawing::Size(247, 139);
            this->stopGameButton->TabIndex = 11;
            this->stopGameButton->Text = L"Stop Game";
            this->stopGameButton->UseVisualStyleBackColor = true;
            this->stopGameButton->Click += gcnew System::EventHandler(this, &MainGUIForm::stopGameButton_Click);
            //
            // label2
```

```
                //
                this->label2->AutoSize = true;
                this->label2->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 18,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                        static_cast<System::Byte>(0)));
                this->label2->Location = System::Drawing::Point(12, 21);
                this->label2->Name = L"label2";
                this->label2->Size = System::Drawing::Size(474, 29);
                this->label2->TabIndex = 12;
                this->label2->Text = L"Welcome to the Puzzle Assembly Assistant!";
                //
                // label3
                //
                this->label3->AutoSize = true;
                this->label3->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 11.25F,
System::Drawing::FontStyle::Italic, System::Drawing::GraphicsUnit::Point,
                        static_cast<System::Byte>(0)));
                this->label3->Location = System::Drawing::Point(168, 162);
                this->label3->Name = L"label3";
                this->label3->Size = System::Drawing::Size(225, 18);
                this->label3->TabIndex = 13;
                this->label3->Text = L"please enter exactly as it appears";
                //
                // level1CheckBox
                //
                this->level1CheckBox->AutoSize = true;
                this->level1CheckBox->Checked = true;
                this->level1CheckBox->CheckState = System::Windows::Forms::CheckState::Checked;
                this->level1CheckBox->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 15.75F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                        static_cast<System::Byte>(0)));
                this->level1CheckBox->Location = System::Drawing::Point(171, 255);
                this->level1CheckBox->Name = L"level1CheckBox";
                this->level1CheckBox->Size = System::Drawing::Size(79, 29);
                this->level1CheckBox->TabIndex = 14;
                this->level1CheckBox->Text = L"Easy";
                this->level1CheckBox->UseVisualStyleBackColor = true;
                this->level1CheckBox->CheckedChanged += gcnew System::EventHandler(this,
&MainGUIForm::level1CheckBox_CheckedChanged);
                //
                // level2CheckBox
                //
                this->level2CheckBox->AutoSize = true;
                this->level2CheckBox->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 15.75F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                        static_cast<System::Byte>(0)));
                this->level2CheckBox->Location = System::Drawing::Point(260, 257);
                this->level2CheckBox->Name = L"level2CheckBox";
                this->level2CheckBox->Size = System::Drawing::Size(107, 29);
                this->level2CheckBox->TabIndex = 15;
                this->level2CheckBox->Text = L"Medium";
                this->level2CheckBox->UseVisualStyleBackColor = true;
                this->level2CheckBox->CheckedChanged += gcnew System::EventHandler(this,
&MainGUIForm::level2CheckBox_CheckedChanged);
                //
                // level3CheckBox
                //
                this->level3CheckBox->AutoSize = true;
                this->level3CheckBox->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 15.75F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                        static_cast<System::Byte>(0)));
                this->level3CheckBox->Location = System::Drawing::Point(373, 257);
```

```
            this->level3CheckBox->Name = L"level3CheckBox";
            this->level3CheckBox->Size = System::Drawing::Size(77, 29);
            this->level3CheckBox->TabIndex = 16;
            this->level3CheckBox->Text = L"Hard";
            this->level3CheckBox->UseVisualStyleBackColor = true;
            this->level3CheckBox->CheckedChanged += gcnew System::EventHandler(this,
&MainGUIForm::level3CheckBox_CheckedChanged);
            //
            // levelDescriptionsButton
            //
            this->levelDescriptionsButton->BackColor = System::Drawing::Color::Linen;
            this->levelDescriptionsButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 12,
System::Drawing::FontStyle::Regular,
                  System::Drawing::GraphicsUnit::Point, static_cast<System::Byte>(0)));
            this->levelDescriptionsButton->Location = System::Drawing::Point(456, 250);
            this->levelDescriptionsButton->Name = L"levelDescriptionsButton";
            this->levelDescriptionsButton->Size = System::Drawing::Size(127, 44);
            this->levelDescriptionsButton->TabIndex = 17;
            this->levelDescriptionsButton->Text = L"Descriptions...";
            this->levelDescriptionsButton->UseVisualStyleBackColor = false;
            this->levelDescriptionsButton->Click += gcnew System::EventHandler(this,
&MainGUIForm::levelDescriptionsButton_Click);
            //
            // label4
            //
            this->label4->AutoSize = true;
            this->label4->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 14.25F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                  static_cast<System::Byte>(0)));
            this->label4->Location = System::Drawing::Point(13, 257);
            this->label4->Name = L"label4";
            this->label4->Size = System::Drawing::Size(152, 24);
            this->label4->TabIndex = 18;
            this->label4->Text = L"Level of Difficulty:";
            //
            // label5
            //
            this->label5->AutoSize = true;
            this->label5->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 18,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                  static_cast<System::Byte>(0)));
            this->label5->Location = System::Drawing::Point(22, 74);
            this->label5->Name = L"label5";
            this->label5->Size = System::Drawing::Size(181, 29);
            this->label5->TabIndex = 20;
            this->label5->Text = L"Who\'s Playing\?:";
            //
            // playerNameComboBox
            //
            this->playerNameComboBox->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 14.25F,
System::Drawing::FontStyle::Regular,
                  System::Drawing::GraphicsUnit::Point, static_cast<System::Byte>(0)));
            this->playerNameComboBox->FormattingEnabled = true;
            this->playerNameComboBox->Location = System::Drawing::Point(27, 106);
            this->playerNameComboBox->Name = L"playerNameComboBox";
            this->playerNameComboBox->Size = System::Drawing::Size(357, 32);
            this->playerNameComboBox->TabIndex = 21;
            this->playerNameComboBox->Text = L"<enter name>";
            this->playerNameComboBox->Click += gcnew System::EventHandler(this,
&MainGUIForm::playerNameComboBox_Click);
            //
            // label6
```

```
                                //
                                this->label6->AutoSize = true;
                                this->label6->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 9.75F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                                        static_cast<System::Byte>(0)));
                                this->label6->Location = System::Drawing::Point(390, 106);
                                this->label6->Name = L"label6";
                                this->label6->Size = System::Drawing::Size(174, 32);
                                this->label6->TabIndex = 22;
                                this->label6->Text = L"* Enter new name or choose\r\n   from dropdown list";
                                //
                                // puzzleComboBox
                                //
                                this->puzzleComboBox->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 18,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                                        static_cast<System::Byte>(0)));
                                this->puzzleComboBox->FormattingEnabled = true;
                                this->puzzleComboBox->Location = System::Drawing::Point(32, 193);
                                this->puzzleComboBox->Name = L"puzzleComboBox";
                                this->puzzleComboBox->Size = System::Drawing::Size(357, 37);
                                this->puzzleComboBox->TabIndex = 23;
                                this->puzzleComboBox->Text = L"<enter game>";
                                this->puzzleComboBox->Click += gcnew System::EventHandler(this, &MainGUIForm::puzzleComboBox_Click);
                                //
                                // helpButton
                                //
                                this->helpButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 9.75F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                                        static_cast<System::Byte>(0)));
                                this->helpButton->Location = System::Drawing::Point(816, 12);
                                this->helpButton->Name = L"helpButton";
                                this->helpButton->Size = System::Drawing::Size(26, 25);
                                this->helpButton->TabIndex = 24;
                                this->helpButton->Text = L"\?";
                                this->helpButton->UseVisualStyleBackColor = true;
                                this->helpButton->Click += gcnew System::EventHandler(this, &MainGUIForm::helpButton_Click);
                                //
                                // MainGUIForm
                                //
                                this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
                                this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
                                this->BackColor = System::Drawing::Color::Beige;
                                this->ClientSize = System::Drawing::Size(854, 432);
                                this->Controls->Add(this->helpButton);
                                this->Controls->Add(this->puzzleComboBox);
                                this->Controls->Add(this->label6);
                                this->Controls->Add(this->playerNameComboBox);
                                this->Controls->Add(this->label5);
                                this->Controls->Add(this->label4);
                                this->Controls->Add(this->levelDescriptionsButton);
                                this->Controls->Add(this->level3CheckBox);
                                this->Controls->Add(this->level2CheckBox);
                                this->Controls->Add(this->level1CheckBox);
                                this->Controls->Add(this->label3);
                                this->Controls->Add(this->label2);
                                this->Controls->Add(this->stopGameButton);
                                this->Controls->Add(this->calibrateButton);
                                this->Controls->Add(this->scoresButton);
                                this->Controls->Add(this->label1);
                                this->Controls->Add(this->runGameButton);
                                this->Name = L"MainGUIForm";
                                this->Text = L"Puzzle Assembly Assistant";
```

```
                this->FormClosing += gcnew System::Windows::Forms::FormClosingEventHandler(this,
&MainGUIForm::MainGUIForm_FormClosing);
                this->ResumeLayout(false);
                this->PerformLayout();

            }
#pragma endregion

// User hits 'Run Game'
private: System::Void runGameButton_Click(System::Object^  sender, System::EventArgs^  e) {

                // Lock down thread while loading puzzle, so that only one thread is accessing it (I'm not actually sure this is doing
anything)
                HANDLE myMutex = CreateMutex(NULL, FALSE, (LPCWSTR) "runGameButton_Click : loading game");
                WaitForSingleObject(myMutex, INFINITE);

                System::String^ userName = playerNameComboBox->Text->ToLower();
                // if player is not recognized, ask if want to save new player, then do so.
                if (!playerNameComboBox->Items->Contains(userName) && !userName->Equals("") && !userName-
>Equals("<enter name>"))
                {
                        System::String^ messageString = "Do you want to save new user " + userName + "?";
                        System::Windows::Forms::DialogResult result = MessageBox::Show(messageString, "Warning",
MessageBoxButtons::YesNo, MessageBoxIcon::Warning);
                        // if user says yes, save the settings to the hardcoded location (user doesn't select)
                        if(result == System::Windows::Forms::DialogResult::Yes)
                        {
                                // create the results directory
                                System::String^ fileStr = Constants::RESULTS_DIRECTORY + userName;
                                System::IO::Directory::CreateDirectory(fileStr);
                                Console::WriteLine("MainGuiForm.h::runGameButton_Click(): Created results directory " +
fileStr);
                                // refill the player drop down to include the new name
                                playerNameComboBox->Items->Clear();
                                array<System::String^>^ patientNames = findPatientNames();
                                playerNameComboBox->Items->AddRange(patientNames);
                        }
                        // if user says no, then return.
                        else if (result == System::Windows::Forms::DialogResult::No) {
                                MessageBox::Show("Please select a valid username and try running again.");
                                return;
                        }
                }
                // if no player has been entered, return
                else if (userName->Equals("") || userName->Equals("<enter name>"))  {
                        MessageBox::Show("Please enter a username and try running again!");
                        return;
                }

                // load up puzzle if not already loaded (make sure it's the same puzzle that the user has entered in the text box too).
                if (!this->currentPuzzle->checkIsInitialized(this->getCodeStringFromGUI())) {
                        Console::WriteLine("MainGUIForm.h : runGameButton_Click() : Loading Puzzle");

                        // load the puzzle from the given code
                        int success = this->loadPuzzleFromCode();

                        // if loading was unsuccessful, alert user and cancel running game
                        if (success == -1) {
                                System::Windows::Forms::MessageBox::Show("Error loading puzzle. \nPlease check code
string");
                                Console::WriteLine("MainGUIForm.h : runGameButton_Click() : Error loading puzzle. Please
check code string");
```

```
                                        ReleaseMutex(myMutex);
                                        return;
                                }
                        }

                        // reload the level of difficulty in case it's changed
                        this->currentPuzzle->setLevelOfDifficulty(this->getLevelOfDifficulty());
                        if (this->currentPuzzle->getLevelOfDifficulty() == -1) {
                                        Console::WriteLine("MainGUIForm.h : runGameButton_Click() : Error loading puzzle. Please
check code string");
                                        ReleaseMutex(myMutex);
                                        return;
                        }
                        // Set global difficulty level
                        Globals::difficultylevel = this->getLevelOfDifficulty();

                        // release lock
                        ReleaseMutex(myMutex);

                        // MAY WANT A COMPREHENSIVE ERROR CHECK within KnobPuzzle, for a one line check

                        // Turn on 'Stop' button and turn off the other buttons for while game is running
                        this->gameRunning = true;
                        turnAllButtonsOff();
                        this->stopGameButton->Enabled = true;

                        // now start the game by initializing the tracking. Pass in the puzzle. It will return the game stats for that game
                        GamePlayedData^ gameResults = initializeTracking( this->currentPuzzle, userName);

                        //add new game results to the ScoreKeeper
                        this->ScoreKeeper->AddNewGame(gameResults);

                        // reset the 'endgame' variable in KnobPuzzle (in case it was set by StopButtonClick)
                        this->currentPuzzle->resetEndGame();

                        // Turn off 'Stop' button, and enable all other buttons
                        turnAllButtonsOnExceptStop();

                        this->gameRunning = false;
                }
//-------------------------------------------------------------------------------------------------
// User stops game before it ends naturally
private: System::Void stopGameButton_Click(System::Object^  sender, System::EventArgs^  e) {
                        // if game isn't running, then return ( this shouldn't be able to happen)
                        if (!gameRunning) {
                                return;
                        }

                        // tell KnobPuzzle to end; RunTracking will see the change in this variable and end.
                        this->currentPuzzle->setEndGame();
                        this->gameRunning = false; // set gameRunning to false (for main gui)
                }

//-------------------------------------------------------------------------------------------------
// User clicks the "Performance" button
private: System::Void scoresButton_Click(System::Object^  sender, System::EventArgs^  e) {

                        // set up the form to display the results, and load it with the necessary data
                        ConsoleApplication4::displayResultsForm^ displayResults = gcnew ConsoleApplication4::displayResultsForm();
                        displayResults->currentPlayer = this->playerNameComboBox->Text->ToLower();
                        displayResults->currentGame = this->puzzleComboBox->Text;
                        displayResults->recordKeeper = this->ScoreKeeper;
```

```
                        // show it as a dialog, so that it pulls focus and ends when the user clicks ok or cancel.
                        System::Windows::Forms::DialogResult dialogResult = displayResults->ShowDialog();

                }

//-------------------------------------------------------------------------------------------------------
// mini function to disable all buttons on the main GUI
private: System::Void turnAllButtonsOff() {
                        this->runGameButton->Enabled = false;
                        this->calibrateButton->Enabled = false;
                        this->scoresButton->Enabled = false;
                        this->stopGameButton->Enabled = false;
                }

// mini function to enable all buttons except the stop button (this is used whenever a game is running)
private: System::Void turnAllButtonsOnExceptStop() {
                        this->runGameButton->Enabled = true;
                        this->calibrateButton->Enabled = true;
                        this->scoresButton->Enabled = true;
                        this->stopGameButton->Enabled = false;
                }
//-------------------------------------------------------------------------------------------------------
// Handle a user clicking the "Calibrate" button
private: System::Void calibrateButton_Click(System::Object^  sender, System::EventArgs^  e) {

                        // Lock down thread for entire calibration process to minimize conflicts. I don't know if this does anything
                        HANDLE myMutex = CreateMutex(NULL, FALSE, (LPCWSTR) "calibrateButton_Click : loading and calibrating");
                        WaitForSingleObject(myMutex, INFINITE);

                        // load up puzzle if not already loaded (compare current KnobPuzzle to the combobox input)
                        if (!this->currentPuzzle->checkIsInitialized(this->getCodeStringFromGUI())) {
                                //MessageBox::Show("Loading Puzzle");
                                System::Console::WriteLine("MainGUIForm.h : calibrateButton_Click() : Loading Puzzle");
                                int success = this->loadPuzzleFromCode();
                                if (success == -1) {
                                        System::Console::WriteLine("MainGUIForm.h : calibrateButton_Click() : Error loading puzzle.
\nPlease check code string");
                                        ReleaseMutex(myMutex);
                                        return;
                                }
                        }

                        // all buttons off while calibrating
                        turnAllButtonsOff();
                        this->calibrating = true;

                        // create new calibration main form and pass it the puzzle. User will now enter the calibration process
                        ConsoleApplication4::CalibrationMainPrompt^ calibForm = gcnew ConsoleApplication4::CalibrationMainPrompt();
                        calibForm->puzzle = this->currentPuzzle;

                        // show the form and wait until the calibration form has exited.
                        System::Windows::Forms::DialogResult dialogResult = calibForm->ShowDialog();

                        this->calibrating = false;
                        ReleaseMutex(myMutex);

                        // if DialogResult is OK, then calibration has been completed successfully (or should)
                        if (dialogResult == System::Windows::Forms::DialogResult::OK) {
                                MessageBox::Show("You're done with calibration!");
                                delete calibForm;
                        }
```

```
                                // if DialogResult was Cancel (user exited prematurely, or there was an error)
                                // then cancel the calibration and reload all of the old calibration data
                                else if (dialogResult == System::Windows::Forms::DialogResult::Cancel) {
                                        delete calibForm;
                                       // reload old data into current puzzle
                                        MessageBox::Show("Re-Loading old puzzle data");
                                        System::Console::WriteLine("MainGUIForm.h : calibrateButton_Click() : ReLoading old Puzzle data");
                                        int success = this->loadPuzzleFromCode();
                                        if (success == -1) {
                                                System::Console::WriteLine("MainGUIForm.h : calibrateButton_Click() : Error reLoading puzzle.
\nPlease check code string");
                                                turnAllButtonsOnExceptStop();
                                                return;
                                        }
                                        // no point in continuing to next stage (saving settings) so return
                                       turnAllButtonsOnExceptStop();
                                       return;
                                }

                                // color and location info should be embedded now in this->currentPuzzle, which should be passed to tracking
initializer

                                // ask user if they want to save settings
                                System::Windows::Forms::DialogResult result = MessageBox::Show("Do you want to save calibration settings for
future sessions?", "Warning", MessageBoxButtons::YesNoCancel, MessageBoxIcon::Warning);

                                // if user says yes, save the settings to the hardcoded location (user doesn't select)
                                if(result == System::Windows::Forms::DialogResult::Yes)
                                {
                                        Console::WriteLine("Saving Settings");
                                int success = this->currentPuzzle->SaveCalibrationSettings();
                                        if (success != 0) {
                                                MessageBox::Show("Error: Failed to save settings. Calibrated values will be used for this session
only.");
                                                Console::WriteLine("Mainguiform::CalibrateButton_Click() : Failed to save settings. Calibrated
values not saved for future use.");
                                        }
                                }

                                // Otherwise, cancel
                                else if(result == System::Windows::Forms::DialogResult::No || result ==
System::Windows::Forms::DialogResult::Cancel)
                                {
                                  Console::WriteLine("Mainguiform::CalibrateButton_Click() : Not saving settings.");
                                }

                                // turn buttons back on
                                turnAllButtonsOnExceptStop();
                                return;
                        }


//------------------------------------------------------------------------------------------------
private: System::Void textBox1_TextChanged(System::Object^  sender, System::EventArgs^  e) {

                                // if puzzle code box is blank, de-enable all buttons (because there is obviously no game to play)
                                if (this->puzzleComboBox->Text->Length == 0) {
                                        this->calibrateButton->Enabled = false;
                                        this->runGameButton->Enabled = false;
                                }
```

```
                        // otherwise enable load and run button (assumes a game has been entered)
                        else {
                                this->calibrateButton->Enabled = true;
                                this->runGameButton->Enabled = true;
                        }
                }


//-----------------------------------------------------------------------------------------------
// take code from input textbox and load puzzle from it (Should I have this take a string argument?)
private: int loadPuzzleFromCode() {

                        int success = 0;

                        // make sure the user has properly selected a level of difficulty
                        int level = this->getLevelOfDifficulty();
                        if ( level == -1) {
                                MessageBox::Show("Please select a level of difficulty");
                                return -1;
                        }
                        // load level of difficulty to puzzle
                        else { this->currentPuzzle->setLevelOfDifficulty(level); }

                        // pull puzzle name from GUI
                        System::String^ CodeString = this->getCodeStringFromGUI();
                        System::String^ puzzleType = searchPuzzleType(CodeString);
                        //KNOB PUZZLE IS STILL HARDCODED HERE- WILL NEED TO GO THROUGH ALL CODE IF YOU WANT
TO ADD NEW GAME TYPES

                        // load up puzzle class. If unsuccessful, will return -1
                        if (puzzleType->Equals("KnobPuzzle")) {
                                // reset the knob puzzle just to be sure everything is cleared correctly
                                this->currentPuzzle = gcnew KnobPuzzle();
                                success = this->currentPuzzle->setGame(CodeString); // this function will load up all puzzle data
                        }

                        // check if loading was successful
                        if (success != 0) {
                                //MessageBox::Show("MainGUIForm.h : loadPuzzleFromCode(): error loading knob puzzle from code");
                                System::Console::WriteLine("MainGUIForm.h : loadPuzzleFromCode(): error loading knob puzzle from
code");
                                return success;
                        }

                        return success;
}


//-----------------------------------------------------------------------------------------------
// Handle the form closing via X button
private: System::Void MainGUIForm_FormClosing(System::Object^  sender, System::Windows::Forms::FormClosingEventArgs^  e) {

                        // if game is running, need to end the game before we can quit
                        if (this->gameRunning) {
                                this->currentPuzzle->setEndGame();
                        }
                        // if currently calibrating, just don't let the form close. User must close out of calibration first.
                        if (this->calibrating) {
                                Console::WriteLine("MainGUIForm.h: MainGUIForm_FormClosing(): attempted to exit main gui during
calibration. Cancelled exit.");
                                e->Cancel = true;
                        }
```

```
}

//-------------------------------------------------------------------------------------------------------
// if user selects a level of difficulty box, set that box to check and uncheck the other difficulty boxes.
private: System::Void level2CheckBox_CheckedChanged(System::Object^  sender, System::EventArgs^  e) {
                                    if (level2CheckBox->Checked == true) {
                                                level1CheckBox->Checked = false;
                                                level3CheckBox->Checked = false;
                                    }

                        }
private: System::Void level3CheckBox_CheckedChanged(System::Object^  sender, System::EventArgs^  e) {
                                    if (level3CheckBox->Checked == true) {
                                                level1CheckBox->Checked = false;
                                                level2CheckBox->Checked = false;
                                    }

                        }
private: System::Void level1CheckBox_CheckedChanged(System::Object^  sender, System::EventArgs^  e) {
                                    if (level1CheckBox->Checked == true) {
                                                level2CheckBox->Checked = false;
                                                level3CheckBox->Checked = false;
                                    }

}

// display a little messagebox describing the difference between the levels of difficulty
private: System::Void levelDescriptionsButton_Click(System::Object^  sender, System::EventArgs^  e) {
                                    System::String^ tmp = "Levels of Difficulty: \n\nEasy: Flash piece being moved, dim all other pieces, then turn off all
other pieces\
                                                                                                                \nMedium: Flash piece being moved, dim all other pieces \n\nHard:
Only flashes piece being moved";
                                    MessageBox::Show(tmp);
                        }

//-------------------------------------------------------------------------------------------------------
// mini function that pulls the game code from the GUI (isolated in case text input method changes
private: System::String^ getCodeStringFromGUI() {
                                    System::String^ resultString = this->puzzleComboBox->Text;
                                    return resultString;
                        }

//-------------------------------------------------------------------------------------------------------
// mini function that pulls the level of difficulty from the GUI
private: int getLevelOfDifficulty() {
                                   int level = -1; // boxes aren't properly checked, will return error (-1)
                                   if (level1CheckBox->Checked == true) { return 1; }
                                   else if (level2CheckBox->Checked == true) { return 2; }
                                   else if (level3CheckBox->Checked == true) { return 3; }
                                   return level;
                        }

//-------------------------------------------------------------------------------------------------------
// handle user clicking on the username text box. Generates drop down menu options
private: System::Void playerNameComboBox_Click(System::Object^  sender, System::EventArgs^  e) {

                                    // This will be called the first time the box is clicked (when the initial prompt <enter name> is still displayed)
                                    if (playerNameComboBox->Text->Equals("<enter name>")) {

                                                // get rid of <enter name> prompt
                                                playerNameComboBox->Text = "";
                                    }
```

```
                                playerNameComboBox->Items->Clear();
                                // find add list of kids that currently have records to drop down list. Each kid should have their own folder in the
patient results mother-folder
                                array<System::String^>^ patientNames = findPatientNames();
                                playerNameComboBox->Items->AddRange(patientNames);
                        }

//=----------------------------------------------------------------------------------------------------
private: array<System::String^>^ findPatientNames() {
                                // find add list of kids that currently have records to drop down list. Each kid should have their own folder in the
patient results mother-folder
                                array<System::String^>^ patientNames = System::IO::Directory::GetDirectories(
Constants::RESULTS_DIRECTORY );
                                for (int i = 0; i < patientNames->Length; i++) {
                                        patientNames[i] = System::IO::Path::GetFileNameWithoutExtension(patientNames[i]);
                                }
                                return patientNames;
                        }
//----------------------------------------------------------------------------------------------------
// handle user clicking on the game text box. Generates drop down menu options
private: System::Void puzzleComboBox_Click(System::Object^  sender, System::EventArgs^  e) {

                                // This will be called the first time the box is clicked (when the initial prompt <enter name> is still displayed)
                                if (puzzleComboBox->Text->Equals("<enter game>") || puzzleComboBox->Text->Equals("KNOBPUZZLE1")) {
//CHANGEME

                                        // get rid of <enter name> prompt
                                        puzzleComboBox->Text = "";

                                        // clear out current drop down items
                                        puzzleComboBox->Items->Clear();

                                        // find all files that contain the word "KNOBPUZZLE"
                                        array<System::String^>^ fileNames = System::IO::Directory::GetFiles(
Constants::GAME_INPUT_DIRECTORY );
                                        List<System::String^>^ matches = gcnew List<System::String^>();
                                        for (int i = 0; i < fileNames->Length; i++) {
                                                System::String^ tmp = System::IO::Path::GetFileNameWithoutExtension(fileNames[i]);
                                                // find knobpuzzle files, and pull the name from them
                                                if (tmp->Contains("KNOBPUZZLE")) {   /// CHANGE THIS ONCE I CREATE A FOLDER
FOR GAME INPUTS
                                                        System::String^ delimStr = "_";
                                                        array<Char>^ delimiter = delimStr->ToCharArray();
                                                  array<System::String^>^ tokens = tmp->Split(delimiter);
                                                        tmp = tokens[0];
                                                        if (!matches->Contains(tmp)) {
                                                                matches->Add(tmp);
                                                        }
                                                }
                                        }
                                        // Now need to convert list back to an array
                                        array<System::String^>^ result = gcnew array<System::String^>(matches->Count);
                                        for (int i = 0; i < matches->Count; i++) {
                                                result[i] = matches[i];
                                        }

                                        // add puzzles to drop down list
                                        puzzleComboBox->Items->AddRange(result);
                                }
                        }

//----------------------------------------------------------------------------------------------------
```

```
// handle user clicking the help button
private: System::Void helpButton_Click(System::Object^  sender, System::EventArgs^  e) {

                                System::String^ fileName = Constants::HELP_FILE;

                                // make sure help file exists and has content
                                checkOrCreateFile(fileName);
                                array<System::String^>^ fileStrings = getStringArrayFromFile(fileName);
                                if (fileStrings->Length == 0) {
                                        MessageBox::Show("Error: can't find help information :(");
                                        return;
                                }
                                if (fileStrings[0]->Equals("Error")) {
                                        MessageBox::Show("Error: can't find help information :(");
                                        return;
                                }

                                // now open up the help file in notepad
                               System::Diagnostics::Process::Start("notepad.exe", fileName);

                        }

};
}
```

CALIBRATIONMAINPROMPT.H

```
/*
This form guides the user through the general calibration process. It launches the ColorCalibrationForm for the user to calibrate colors.
It then uses the newly calibrated colors and launches OpenCV to record the destination locations of each piece.
*/

#include <Windows.h>
#include <atlstr.h>
#include "stdafx.h"
#include <WinBase.h>
#include <WinUser.h>
#using <System.dll>
#include <stdlib.h>
#include  <stdio.h>
#include <vcclr.h>
#include <opencv2\opencv.hpp>          //includes all OpenCV headers
#include "Shape.h"

#include "Functions.h"
#include "ColorCalibrationForm.h"
#include "CalibrationTracking.h"

#pragma once

namespace ConsoleApplication4 {

        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;

        /// <summary>
```

```
/// Summary for CalibrationMainPrompt
/// </summary>
public ref class CalibrationMainPrompt : public System::Windows::Forms::Form
{
public:
	CalibrationMainPrompt(void)
	{
		InitializeComponent();
		this->STARTED = false;
		this->calibratingColors = true;
		this->calibratingLocations = false;
		this->waitingToPlacePieces = false;
		this->puzzle = gcnew KnobPuzzle();
		this->calibNextButton->Focus();
		this->colorForm = gcnew ConsoleApplication4::ColorCalibrationForm();
		this->myCalibrator = gcnew CalibrationTracking();

	}

protected:
	/// <summary>
	/// Clean up any resources being used.
	/// </summary>
	~CalibrationMainPrompt()
	{
		if (components)
		{
			delete components;
		}
	}

private: System::Windows::Forms::Label^  label1;
private: System::Windows::Forms::Label^  label2;
private: System::Windows::Forms::Button^  calibNextButton;
private: System::Windows::Forms::Label^  startColorsText;
private: System::Windows::Forms::Label^  placePiecesLabel;
private: System::Windows::Forms::Label^  pleaseWaitLabel;
private: bool calibratingColors;
private: bool calibratingLocations;
private: bool waitingToPlacePieces;
private: bool STARTED;
public: KnobPuzzle^ puzzle;
private: ConsoleApplication4::ColorCalibrationForm^ colorForm;
private: CalibrationTracking^ myCalibrator;
private: ThreadShell myThreadShell;

protected:

protected:

private:
	/// <summary>
	/// Required designer variable.
	/// </summary>
	System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
	/// <summary>
	/// Required method for Designer support - do not modify
	/// the contents of this method with the code editor.
	/// </summary>
	void InitializeComponent(void)
```

```
{
	this->label1 = (gcnew System::Windows::Forms::Label());
	this->label2 = (gcnew System::Windows::Forms::Label());
	this->calibNextButton = (gcnew System::Windows::Forms::Button());
	this->startColorsText = (gcnew System::Windows::Forms::Label());
	this->placePiecesLabel = (gcnew System::Windows::Forms::Label());
	this->pleaseWaitLabel = (gcnew System::Windows::Forms::Label());
	this->SuspendLayout();
	//
	// label1
	//
	this->label1->AutoSize = true;
	this->label1->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 12, System::Drawing::FontStyle::Bold,
System::Drawing::GraphicsUnit::Point,
		static_cast<System::Byte>(0)));
	this->label1->Location = System::Drawing::Point(13, 11);
	this->label1->Name = L"label1";
	this->label1->Size = System::Drawing::Size(95, 20);
	this->label1->TabIndex = 1;
	this->label1->Text = L"Calibration";
	//
	// label2
	//
	this->label2->AutoSize = true;
	this->label2->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 9.75F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
		static_cast<System::Byte>(0)));
	this->label2->Location = System::Drawing::Point(17, 44);
	this->label2->Name = L"label2";
	this->label2->Size = System::Drawing::Size(500, 32);
	this->label2->TabIndex = 2;
	this->label2->Text = L"To account for varying lighting conditions, calibrate color recognition before st"
		L"arting \r\na new game. If the board doesn\'t seem to be tracking correctly, try rec"
		L"alibrating. \r\n";
	//
	// calibNextButton
	//
	this->calibNextButton->Anchor = System::Windows::Forms::AnchorStyles::Right;
	this->calibNextButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 12,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
		static_cast<System::Byte>(0)));
	this->calibNextButton->Location = System::Drawing::Point(436, 122);
	this->calibNextButton->Name = L"calibNextButton";
	this->calibNextButton->Size = System::Drawing::Size(123, 37);
	this->calibNextButton->TabIndex = 3;
	this->calibNextButton->Text = L"Next...";
	this->calibNextButton->UseVisualStyleBackColor = true;
	this->calibNextButton->Click += gcnew System::EventHandler(this,
&CalibrationMainPrompt::calibNextButton_Click);
	//
	// startColorsText
	//
	this->startColorsText->AutoSize = true;
	this->startColorsText->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 12,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
		static_cast<System::Byte>(0)));
	this->startColorsText->Location = System::Drawing::Point(45, 122);
	this->startColorsText->Name = L"startColorsText";
	this->startColorsText->Size = System::Drawing::Size(261, 20);
	this->startColorsText->TabIndex = 4;
	this->startColorsText->Text = L"Click Next to begin calibrating colors";
	//
```

```
                                // placePiecesLabel
                                //
                                this->placePiecesLabel->AutoSize = true;
                                this->placePiecesLabel->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 12,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                                        static_cast<System::Byte>(0)));
                                this->placePiecesLabel->Location = System::Drawing::Point(12, 99);
                                this->placePiecesLabel->Name = L"placePiecesLabel";
                                this->placePiecesLabel->Size = System::Drawing::Size(412, 60);
                                this->placePiecesLabel->TabIndex = 5;
                                this->placePiecesLabel->Text = L"Please place all puzzle pieces in their assigned locations. \r\nPiece  locations wi"
                                        L"ll now be calibrated.\r\nWhen you are ready, click Next";
                                this->placePiecesLabel->Visible = false;
                                //
                                // pleaseWaitLabel
                                //
                                this->pleaseWaitLabel->AutoSize = true;
                                this->pleaseWaitLabel->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 12,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                                        static_cast<System::Byte>(0)));
                                this->pleaseWaitLabel->Location = System::Drawing::Point(55, 130);
                                this->pleaseWaitLabel->Name = L"pleaseWaitLabel";
                                this->pleaseWaitLabel->Size = System::Drawing::Size(307, 20);
                                this->pleaseWaitLabel->TabIndex = 6;
                                this->pleaseWaitLabel->Text = L"Please wait while locations are calibrated...";
                                this->pleaseWaitLabel->Visible = false;
                                //
                                // CalibrationMainPrompt
                                //
                                this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
                                this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
                                this->ClientSize = System::Drawing::Size(571, 176);
                                this->Controls->Add(this->pleaseWaitLabel);
                                this->Controls->Add(this->placePiecesLabel);
                                this->Controls->Add(this->startColorsText);
                                this->Controls->Add(this->calibNextButton);
                                this->Controls->Add(this->label2);
                                this->Controls->Add(this->label1);
                                this->Name = L"CalibrationMainPrompt";
                                this->Text = L"Calibration";
                                this->FormClosing += gcnew System::Windows::Forms::FormClosingEventHandler(this,
&CalibrationMainPrompt::CalibrationMainPrompt_FormClosing);
                                this->ResumeLayout(false);
                                this->PerformLayout();

                        }
#pragma endregion
                private: System::Void calibNextButton_Click(System::Object^  sender, System::EventArgs^  e) {

                                        if (this->puzzle->getPieceList()->Count <= 0) {
                                                System::Windows::Forms::MessageBox::Show("Error: cannot find puzzle piece information.
Please check game ID and try again");
                                                System::Console::WriteLine("CalibrationMainPrompt.h::calibNextButton_Click() : Error- puzzle
has no pieces. Exiting calibration");
                                                this->DialogResult = System::Windows::Forms::DialogResult::Cancel;
                                                this->Close();
                                                return;
                                        }

                                        // disable the next button
                                        this->calibNextButton->Enabled = false;
```

```
                // if we are currently in the color stage, do the following
                if (this->calibratingColors) {

                        // make the main calibration form invisible
                        this->Visible = false;


                        //pass puzzle class over to color form, launch it, and wait for it to return a dialogresult
                        this->colorForm->puzzle = this->puzzle;
                        System::Windows::Forms::DialogResult dialogResult = colorForm->ShowDialog();
                        if (dialogResult == System::Windows::Forms::DialogResult::OK) {
                                System::Console::WriteLine("CalibrationMainPrompt.h::calibNextButton_Click() :
colorFrom returned dialogue result OK");
                                delete colorForm;
                        }
                        else if (dialogResult == System::Windows::Forms::DialogResult::Cancel) {
                                System::Console::WriteLine("CalibrationMainPrompt.h::calibNextButton_Click() :
colorFrom returned dialogue result Cancel");
                                delete colorForm;
                                this->Close(); // cancel calibration; close form. This will result in DialogResult::Cancel
                                return;
                        }

                        // make this form visible again
                        this->Visible = true;

                        // switch text from 'startcolors' to 'please place pieces on board'
                        this->startColorsText->Visible = false;
                        this->placePiecesLabel->Visible = true;

                        // change current status from 'calibrating colors' to 'waiting for user to place pieces'
                        this->calibratingColors = false;
                        this->waitingToPlacePieces = true;


                        // since the calibrator will be running and running, set STARTED to true so we can be sure to close
it down.
                        this->STARTED = true;

                       //re enable the next button
                       this->calibNextButton->Enabled = true;

                        // set up the calibrator running. For now it will just show the gameboard.
                        // when waitingForUserToPlacePieces becomes false, it will start the location calibration.
                        this->myCalibrator = gcnew CalibrationTracking();
                        myCalibrator->setGame(this->puzzle);
                        myCalibrator->waitingForUserToPlacePieces = true;
                        // running it on a separate thread so we can still process the callbacks here.
                        //myCalibrator->startLocationCalibration();
                        this->myThreadShell.myThread = gcnew System::Threading::Thread(gcnew
System::Threading::ThreadStart(myCalibrator, &CalibrationTracking::startLocationCalibration));
                        this->myThreadShell.myThread->Start();

                       return;
                }

                // if we are currently waiting for the user to place pieces, and the user clicks the next button, then start to
calibrate locations:
                if (this->waitingToPlacePieces) {

                        // change instructions from 'please place pieces' to 'please wait for locations to be calibrated'
                        this->placePiecesLabel->Visible = false;
```

```
                                        //this->placePiecesLabel->Text = "Please wait while locations are calibrated...";
                                        //this->pleaseWaitLabel->Visible = true;
                                        System::Threading::Thread::Sleep(100); // wait just a moment so it changes the label before the calibrator takes focus

                                        // change current status from 'waiting for user to place pieces' to 'calibrating locations'
                                        this->waitingToPlacePieces = false;
                                        this->calibratingLocations = true;

                                        // set up a new CalibrationTracking^, pass it our puzzle, and ask it to find the locations
                                        //CalibrationTracking^ locationTracker = gcnew CalibrationTracking();
                                        //locationTracker->setGame(this->puzzle);
                                        //locationTracker->startLocationCalibration();
                                        myCalibrator->waitingForUserToPlacePieces = false;
                                        while (!myCalibrator->IS_STOPPED) {
                                                System::Threading::Thread::Sleep(30);
                                        }

                                        // here the user waits while it calibrates location

                                        // this was the last step, so set dialog result to OK to leave calibration
                                        this->DialogResult = System::Windows::Forms::DialogResult::OK;
                                        this->Close();
                                        return;
                                }

                                //re enable the next button
                                this->calibNextButton->Enabled = true;
                        }


// If this form is closed prematurely, close the spawned ColorCalibrationForm, which should stop any threads running there.
private: System::Void CalibrationMainPrompt_FormClosing(System::Object^  sender, System::Windows::Forms::FormClosingEventArgs^  e) {

                        if (this->STARTED == false) {
                                cv::destroyAllWindows();
                                this->DialogResult = System::Windows::Forms::DialogResult::Cancel;
                                return;
                        }

                        // end thread showing gameboard picture

                        if (!myCalibrator->IS_STOPPED) {
                                myCalibrator->Stop();
                                while (!this->myCalibrator->IS_STOPPED) {
                                        Console::WriteLine("calibrationMainPrompt::FormClosing():: Waiting for calibrator thread to end");

                                        System::Threading::Thread::Sleep(5);
                                }
                        }
                        this->myThreadShell.myThread->Abort();
                        this->myThreadShell.myThread->Join();
                        cv::destroyAllWindows();

                        if (this->colorForm->Enabled) {
                                this->colorForm->Close();
                                delete colorForm;
                        }
                        if (this->DialogResult != System::Windows::Forms::DialogResult::OK) {
                                this->DialogResult = System::Windows::Forms::DialogResult::Cancel;
                        }
```

```
                delete myCalibrator;
                cv::destroyAllWindows();
            }
};
}
```

COLORCALIBRATIONFORM.H

```
/*
This form guides the user through the color calibration process. It launches OpenCV tracking, and then steps through each piece in the puzzle
as the user calibrates.
*/

#include <Windows.h>
#include "stdafx.h"
#include <WinBase.h>
#include <WinUser.h>
#using <System.dll>
#include <stdlib.h>
#include  <stdio.h>
#include <vcclr.h>
#include "Functions.h"
#include "CalibrationTracking.h"


#pragma once

namespace ConsoleApplication4 {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    /// <summary>
    /// Summary for ColorCalibrationForm
    /// </summary>
    public ref class ColorCalibrationForm : public System::Windows::Forms::Form
    {
    public:
        ColorCalibrationForm(void)
        {
            InitializeComponent();

            // initialize our global variables here
            this->puzzle = gcnew KnobPuzzle(); // this holds our puzzle, which will be passed in by CalibrationMainPrompt.h
            this->pieceIndex = 0; // this tells us which piece we are currently calibrating
            this->piece = gcnew PuzzlePiece(); // this is the piece we are currently calibrating
            this->STARTED = false;
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
```

```
		~ColorCalibrationForm()
		{
			if (components)
			{
				delete components;
			}
		}
	private: System::Windows::Forms::Button^  okButton;
	protected:

	public: KnobPuzzle^ puzzle;
	private: int pieceIndex;
	private: PuzzlePiece^ piece;
	private: bool NEXT;
	private: bool STARTED;
	private: CalibrationTracking calibrator;
	private: ThreadShell myThreadShell;
	private: System::Windows::Forms::Label^  label1;
	public:
	private: System::Windows::Forms::Label^  label2;
	private: System::Windows::Forms::Label^  label3;
	private: System::Windows::Forms::Label^  currentPieceLabel;
	private: System::Windows::Forms::Button^  hintButton;

	protected:

	private:
		/// <summary>
		/// Required designer variable.
		/// </summary>
		System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
		/// <summary>
		/// Required method for Designer support - do not modify
		/// the contents of this method with the code editor.
		/// </summary>
		void InitializeComponent(void)
		{
			System::ComponentModel::ComponentResourceManager^  resources = (gcnew
System::ComponentModel::ComponentResourceManager(ColorCalibrationForm::typeid));
			this->okButton = (gcnew System::Windows::Forms::Button());
			this->label1 = (gcnew System::Windows::Forms::Label());
			this->label2 = (gcnew System::Windows::Forms::Label());
			this->label3 = (gcnew System::Windows::Forms::Label());
			this->currentPieceLabel = (gcnew System::Windows::Forms::Label());
			this->hintButton = (gcnew System::Windows::Forms::Button());
			this->SuspendLayout();
			//
			// okButton
			//
			this->okButton->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windows::Forms::AnchorStyles::Bottom |
System::Windows::Forms::AnchorStyles::Right));
			this->okButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 20.25F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
				static_cast<System::Byte>(0)));
			this->okButton->Location = System::Drawing::Point(277, 199);
			this->okButton->Name = L"okButton";
			this->okButton->Size = System::Drawing::Size(155, 75);
			this->okButton->TabIndex = 0;
			this->okButton->Text = L"OK";
```

```
            this->okButton->UseVisualStyleBackColor = true;
            this->okButton->Click += gcnew System::EventHandler(this, &ColorCalibrationForm::okButton_Click);
            //
            // label1
            //
            this->label1->AutoSize = true;
            this->label1->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 9.75F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                static_cast<System::Byte>(0)));
            this->label1->Location = System::Drawing::Point(13, 13);
            this->label1->Name = L"label1";
            this->label1->Size = System::Drawing::Size(194, 16);
            this->label1->TabIndex = 1;
            this->label1->Text = L"Calibrate Colors for Each Piece";
            //
            // label2
            //
            this->label2->AutoSize = true;
            this->label2->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 9.75F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                static_cast<System::Byte>(0)));
            this->label2->Location = System::Drawing::Point(16, 44);
            this->label2->Name = L"label2";
            this->label2->Size = System::Drawing::Size(401, 112);
            this->label2->TabIndex = 2;
            this->label2->Text = resources->GetString(L"label2.Text");
            //
            // label3
            //
            this->label3->AutoSize = true;
            this->label3->Location = System::Drawing::Point(19, 172);
            this->label3->Name = L"label3";
            this->label3->Size = System::Drawing::Size(74, 13);
            this->label3->TabIndex = 3;
            this->label3->Text = L"Current Piece:";
            //
            // currentPieceLabel
            //
            this->currentPieceLabel->AutoSize = true;
            this->currentPieceLabel->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 14.25F,
System::Drawing::FontStyle::Regular,
                System::Drawing::GraphicsUnit::Point, static_cast<System::Byte>(0)));
            this->currentPieceLabel->Location = System::Drawing::Point(100, 171);
            this->currentPieceLabel->Name = L"currentPieceLabel";
            this->currentPieceLabel->Size = System::Drawing::Size(145, 24);
            this->currentPieceLabel->TabIndex = 4;
            this->currentPieceLabel->Text = L"click ok to begin";
            //
            // hintButton
            //
            this->hintButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 9.75F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                static_cast<System::Byte>(0)));
            this->hintButton->Location = System::Drawing::Point(376, 13);
            this->hintButton->Name = L"hintButton";
            this->hintButton->Size = System::Drawing::Size(56, 28);
            this->hintButton->TabIndex = 5;
            this->hintButton->Text = L"Tips";
            this->hintButton->UseVisualStyleBackColor = true;
            this->hintButton->Click += gcnew System::EventHandler(this, &ColorCalibrationForm::hintButton_Click);
            //
            // ColorCalibrationForm
```

```
                        //
                        this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
                        this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
                        this->ClientSize = System::Drawing::Size(444, 286);
                        this->Controls->Add(this->hintButton);
                        this->Controls->Add(this->currentPieceLabel);
                        this->Controls->Add(this->label3);
                        this->Controls->Add(this->label2);
                        this->Controls->Add(this->label1);
                        this->Controls->Add(this->okButton);
                        this->Name = L"ColorCalibrationForm";
                        this->Text = L"Calibrate Colors";
                        this->FormClosing += gcnew System::Windows::Forms::FormClosingEventHandler(this,
&ColorCalibrationForm::ColorCalibrationForm_FormClosing);
                        this->Load += gcnew System::EventHandler(this, &ColorCalibrationForm::ColorCalibrationForm_Load);
                        this->ResumeLayout(false);
                        this->PerformLayout();

                }
#pragma endregion
        private: System::Void ColorCalibrationForm_Load(System::Object^  sender, System::EventArgs^  e) {
                        }


        private: System::Void okButton_Click(System::Object^  sender, System::EventArgs^  e) {

                        this->STARTED = true;
                        // make sure the puzzle has been properly assigned first
                        if (!this->puzzle->checkIsInitialized()) {
                                Console::WriteLine("ColorCalibrationForm.h::okButton_Click: Error - this form was not properly
initialized with a puzzle. Please give it a puzzle.");
                                this->DialogResult = System::Windows::Forms::DialogResult::Cancel;                      // if not,
return DialogResult::Cancel
                        }

                        // if the last piece has been calibrated, stop the calibrator, change the dialog result to OK and close this form
                        // note that the color calibration form is handling updating the colors in the PuzzlePiece instances
                        if (this->pieceIndex >= this->puzzle->getPieceList()->Count && this->puzzle->getPieceList()->Count != 0)
{
                                //MessageBox::Show("That was the last piece!");

                                // lock thread while we wait for calibration thread to end
                                HANDLE myMutex = CreateMutex(NULL, FALSE, (LPCWSTR) "calibration");
                                WaitForSingleObject(myMutex, INFINITE);

                                // tell calibrator to stop, and wait until it responds that it has done so
                                this->calibrator.Stop();
                                while (!this->calibrator.IS_STOPPED) {
                                        Console::WriteLine("Waiting for calibrator thread to end");
                                        System::Threading::Thread::Sleep(30);
                                }
                                // now abort the calibration thread and join it to the current one
                                this->myThreadShell.myThread->Abort();
                                this->myThreadShell.myThread->Join();

                                // release mutex and exit out of this form with the dialogresult::OK
                                ReleaseMutex(myMutex);
                                this->DialogResult = System::Windows::Forms::DialogResult::OK;
                                this->Close();
                                return;
                        }
```

```
                                        // if on the first piece, set up the new thread for the calibration. Thread will start tracking first piece.
                                        if (this->pieceIndex == 0) {
                                                // display current piece name on gui
                                                this->piece = this->puzzle->getPieceList()[this->pieceIndex];
                                                this->currentPieceLabel->Text = this->piece->getName();

                                                // setup calibrator thread with puzzle and start it (I'm wondering if it would be better to pass it the
puzzle piece to track instead - probably)
                                                this->calibrator.setGame(this->puzzle);
                                                this->myThreadShell.myThread = gcnew System::Threading::Thread(gcnew
System::Threading::ThreadStart(calibrator.returnHandle(), &CalibrationTracking::Start));
                                                this->myThreadShell.myThread->Start();
                                                // the first piece will start calibrating automatically

                                               // update piece index to next piece
                                                this->pieceIndex++;

                                                return;
                                        }

                                        // to iterate to next piece, tell calibrator thread to move to next piece, and then move to the next piece in this
thread as well
                                        if (this->puzzle->getPieceList()->Count != 0) {
                                                this->calibrator.Next();
                                                this->piece = this->puzzle->getPieceList()[this->pieceIndex];
                                                this->currentPieceLabel->Text = this->piece->getName();
                                                this->pieceIndex++;
                                                return;
                                        }

                                }

// if the form is closed prematurely, stop the calibrator thread
                                // already calibrated pieces will retain the new calibration information
 private: System::Void ColorCalibrationForm_FormClosing(System::Object^  sender, System::Windows::Forms::FormClosingEventArgs^  e) {

                        if (this->STARTED == false) {
                                // take down the gameboard window
                                cv::destroyAllWindows();
                                this->DialogResult = System::Windows::Forms::DialogResult::Cancel;
                                return;
                        }

                        // if the calibrator is already stopped, don't have to do anything
                        if (!this->calibrator.IS_STOPPED) {
                                Console::WriteLine("ColorCalibrationForm.h: this form is ending prematurely");
                                // if it isn't, go ahead and stop it
                                this->calibrator.Stop();
                                while (!this->calibrator.IS_STOPPED) {
                                        Console::WriteLine("Waiting for calibrator thread to end");
                                        System::Threading::Thread::Sleep(5);
                                }
                        }
                        // now abort the calibration thread and join it to the current one
                        this->myThreadShell.myThread->Abort();
                        this->myThreadShell.myThread->Join();
                        delete calibrator.returnHandle();
                        cv::destroyAllWindows();
                        if (this->DialogResult != System::Windows::Forms::DialogResult::OK) {
                                        this->DialogResult = System::Windows::Forms::DialogResult::Cancel;
                        }
                        cv::destroyAllWindows();
```

```
		}
private: System::Void hintButton_Click(System::Object^  sender, System::EventArgs^  e) {
                                        // pull all text from the help file into string array
                                        array<System::String^>^ fileStrings = getStringArrayFromFile(Constants::CALIBRATION_HELP_FILE);

                                       // if that didn't work, return an error
                                        if (fileStrings[0]->Equals("Error")) {
                                                MessageBox::Show("Error: can't find hint information :(");
                                                return;
                                        }

                                       // now cat all the strings together and show
                                        System::String^ final = "";
                                        for each (System::String^ line in fileStrings) {
                                                final = final + line + Environment::NewLine;
                                        }
                                        MessageBox::Show(final);
                        }
};
}
```

DISPLAYRESULTSFORM.H

```
/*
        This form displays results either from the current session (default) or from loaded data.
        Loaded data is pulled in via a selectOldResultsForm.
*/

#include "selectOldResultsForm.h"
#include "Functions.h"
#pragma once

namespace ConsoleApplication4 {

        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
        using namespace System::Data;
        using namespace System::Drawing;

        /// <summary>
        /// Summary for displayResultsForm
        /// </summary>
        public ref class displayResultsForm : public System::Windows::Forms::Form
        {
        public:
                displayResultsForm(void)
                {
                        InitializeComponent();
                        this->currentGame = "Unknown";
                        this->currentPlayer = "Unknown";
                        this->filesToBeDisplayed = gcnew List<System::String^>();
                        this->recordKeeper = gcnew ScoreKeeping();
                        //
                        //TODO: Add the constructor code here
                        //
                }

        protected:
                /// <summary>
                /// Clean up any resources being used.
```

```
		/// </summary>
		~displayResultsForm()
		{
			if (components)
			{
				delete components;
			}
		}

	public: System::String^ currentPlayer;
	public: System::String^ currentGame;
	public: ScoreKeeping^ recordKeeper;
	private: List<System::String^>^ filesToBeDisplayed;

	private: System::Windows::Forms::Button^  displayOldDataButton;
	private: System::Windows::Forms::Button^  doneButton;
	private: System::Windows::Forms::Label^  todaysSessionLabel;
	private: System::Windows::Forms::Label^  numGamesPlayedLabel;

	private: System::Data::DataSet^  myDataSet;
	private: System::Windows::Forms::TextBox^  mainTextBox;


	public:

	protected:

	private:
		/// <summary>
		/// Required designer variable.
		/// </summary>
		System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
		/// <summary>
		/// Required method for Designer support - do not modify
		/// the contents of this method with the code editor.
		/// </summary>
		void InitializeComponent(void)
		{
			this->displayOldDataButton = (gcnew System::Windows::Forms::Button());
			this->doneButton = (gcnew System::Windows::Forms::Button());
			this->todaysSessionLabel = (gcnew System::Windows::Forms::Label());
			this->numGamesPlayedLabel = (gcnew System::Windows::Forms::Label());
			this->myDataSet = (gcnew System::Data::DataSet());
			this->mainTextBox = (gcnew System::Windows::Forms::TextBox());
			(cli::safe_cast<System::ComponentModel::ISupportInitialize^  >(this->myDataSet))->BeginInit();
			this->SuspendLayout();
			//
			// displayOldDataButton
			//
			this->displayOldDataButton->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windows::Forms::AnchorStyles::Bottom |
System::Windows::Forms::AnchorStyles::Left));
			this->displayOldDataButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 15.75F,
System::Drawing::FontStyle::Regular,
				System::Drawing::GraphicsUnit::Point, static_cast<System::Byte>(0)));
			this->displayOldDataButton->Location = System::Drawing::Point(1, 466);
			this->displayOldDataButton->Name = L"displayOldDataButton";
			this->displayOldDataButton->Size = System::Drawing::Size(231, 43);
			this->displayOldDataButton->TabIndex = 0;
			this->displayOldDataButton->Text = L"Look at old data";
```

```
            this->displayOldDataButton->UseVisualStyleBackColor = true;
            this->displayOldDataButton->Click += gcnew System::EventHandler(this,
&displayResultsForm::displayOldDataButton_Click);
            //
            // doneButton
            //
            this->doneButton->Anchor =
static_cast<System::Windows::Forms::AnchorStyles>((System::Windows::Forms::AnchorStyles::Bottom |
System::Windows::Forms::AnchorStyles::Right));
            this->doneButton->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 18,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                static_cast<System::Byte>(0)));
            this->doneButton->Location = System::Drawing::Point(329, 466);
            this->doneButton->Name = L"doneButton";
            this->doneButton->Size = System::Drawing::Size(223, 43);
            this->doneButton->TabIndex = 1;
            this->doneButton->Text = L"Done";
            this->doneButton->UseVisualStyleBackColor = true;
            this->doneButton->Click += gcnew System::EventHandler(this, &displayResultsForm::doneButton_Click);
            //
            // todaysSessionLabel
            //
            this->todaysSessionLabel->AutoSize = true;
            this->todaysSessionLabel->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 15.75F,
System::Drawing::FontStyle::Regular,
                System::Drawing::GraphicsUnit::Point, static_cast<System::Byte>(0)));
            this->todaysSessionLabel->Location = System::Drawing::Point(13, 13);
            this->todaysSessionLabel->Name = L"todaysSessionLabel";
            this->todaysSessionLabel->Size = System::Drawing::Size(272, 25);
            this->todaysSessionLabel->TabIndex = 2;
            this->todaysSessionLabel->Text = L"Results of Current Session";
            //
            // numGamesPlayedLabel
            //
            this->numGamesPlayedLabel->AutoSize = true;
            this->numGamesPlayedLabel->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 14.25F,
System::Drawing::FontStyle::Regular,
                System::Drawing::GraphicsUnit::Point, static_cast<System::Byte>(0)));
            this->numGamesPlayedLabel->Location = System::Drawing::Point(18, 42);
            this->numGamesPlayedLabel->Name = L"numGamesPlayedLabel";
            this->numGamesPlayedLabel->Size = System::Drawing::Size(173, 24);
            this->numGamesPlayedLabel->TabIndex = 3;
            this->numGamesPlayedLabel->Text = L"# of games played: ";
            //
            // myDataSet
            //
            this->myDataSet->DataSetName = L"NewDataSet";
            //
            // mainTextBox
            //
            this->mainTextBox->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 12,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                static_cast<System::Byte>(0)));
            this->mainTextBox->Location = System::Drawing::Point(1, 69);
            this->mainTextBox->Multiline = true;
            this->mainTextBox->Name = L"mainTextBox";
            this->mainTextBox->ScrollBars = System::Windows::Forms::ScrollBars::Both;
            this->mainTextBox->Size = System::Drawing::Size(551, 391);
            this->mainTextBox->TabIndex = 4;
            //
            // displayResultsForm
            //
```

```
            this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(552, 509);
            this->Controls->Add(this->mainTextBox);
            this->Controls->Add(this->numGamesPlayedLabel);
            this->Controls->Add(this->todaysSessionLabel);
            this->Controls->Add(this->doneButton);
            this->Controls->Add(this->displayOldDataButton);
            this->Name = L"displayResultsForm";
            this->Text = L"Performance";
            this->Load += gcnew System::EventHandler(this, &displayResultsForm::displayResultsForm_Load);
            (cli::safe_cast<System::ComponentModel::ISupportInitialize^ >(this->myDataSet))->EndInit();
            this->ResumeLayout(false);
            this->PerformLayout();

        }
#pragma endregion

        // handle user clicking "look at old data"
    private: System::Void displayOldDataButton_Click(System::Object^ sender, System::EventArgs^ e) {

            // create form to select old records
            selectOldResultsForm^ oldResultsForm = gcnew selectOldResultsForm();
            // load form with current player and game and launch
            oldResultsForm->initialGame = this->currentGame;
            oldResultsForm->initialPlayer = this->currentPlayer;
        System::Windows::Forms::DialogResult dialogResult = oldResultsForm->ShowDialog();

            System::String^ selectedUser = "";
            System::String^ selectedGame = "";
            array<System::String^>^ selectedDates = gcnew array<System::String^>(0);

            // if the user selected OK on the form, then pull the selected player name, game, and dates
            if (dialogResult == System::Windows::Forms::DialogResult::OK) {
                selectedUser = oldResultsForm->selectedPlayer;
                selectedGame = oldResultsForm->selectedGame;
                selectedDates = oldResultsForm->selectedDates;
                // if no dates were chosen, return
                if (selectedDates->Length == 0) {
                    Console::WriteLine("displayResultsForm::displayOldDataButton_Click(): no dates were
chosen");
                    return;
                }
                this->currentGame = selectedGame;
                this->currentPlayer = selectedUser;
            }
            // if user hit cancel or there was an error, go ahead and return
            else if (dialogResult != System::Windows::Forms::DialogResult::OK) {
                Console::WriteLine("displayResultsForm::displayOldDataButton_Click(): user cancelled");
                return;
            }
            // find all of the files we need to pull based on selected user, game, and dates
            List<System::String^>^ filesNeeded = findRecordFiles(selectedUser, selectedGame, selectedDates);

            displayFiles(filesNeeded);
        }

    // if the user has selected old files, display their contents in the text box
private: void displayFiles(List<System::String^>^ files) {
             System::String^ resultString = "";

             // pull the contents of each file
```

```
                    for each(System::String^ file in files) {
                        array<System::String^>^ contents = getStringArrayFromFile(file);
                        if (contents[0]->Equals("ERROR")) {
                            continue;
                        }
                        for each (System::String^ line in contents) {
                            resultString = resultString + line + Environment::NewLine;
                        }
                    }
                    // display the loaded data
                    mainTextBox->Text = resultString;
                    todaysSessionLabel->Text = "Showing results for " + currentPlayer;
                    numGamesPlayedLabel->Text = currentGame;

                }

                // when the form loads up for the first time, build score display
        private: System::Void displayResultsForm_Load(System::Object^  sender, System::EventArgs^  e) {

                    int numberOfGames = 0;

                    // Pull result string from each game played this session
                    System::String^ displayString = "";
                    for (int i = this->recordKeeper->individualGamesList->Count -1; i >= 0; i--)   {
                        GamePlayedData^ game = this->recordKeeper->individualGamesList[i];
                        System::String^ tmpString = game->writeOut();
                        if (!tmpString->Contains("Error")) {
                            displayString = displayString + tmpString + Environment::NewLine;
                            numberOfGames++; // keep track of number of games completed
                        }
                    }
                    numGamesPlayedLabel->Text = "# of Games Played: " + numberOfGames;
                    mainTextBox->Text = displayString;
                }

// if user clicks 'Done', leave form
private: System::Void doneButton_Click(System::Object^  sender, System::EventArgs^  e) {
                this->DialogResult = System::Windows::Forms::DialogResult::OK;
            }
};
}
```

SELECTOLDRESULTSFORM.H

```
/*
        This form allows the user to select old data for display. It finds what dates are available for data based on the entered
        username and game, and the expected location of the data. The form will return the selected player, game, and dates (not the data).
*/

#include "Functions.h"
#pragma once

namespace ConsoleApplication4 {

        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
```

```
using namespace System::Data;
using namespace System::Drawing;

/// <summary>
/// Summary for selectOldResultsForm
/// </summary>
public ref class selectOldResultsForm : public System::Windows::Forms::Form
{
public:
	selectOldResultsForm(void)
	{
		InitializeComponent();
		this->initialGame = "ex. KNOBPUZZLE1";
		this->initialPlayer = "ex. Caleb";
		this->selectedDates = gcnew array<System::String^>(0);
		this->selectedGame = "";
		this->selectedPlayer = "";
		//
		//TODO: Add the constructor code here
		//
	}

protected:
	/// <summary>
	/// Clean up any resources being used.
	/// </summary>
	~selectOldResultsForm()
	{
		if (components)
		{
			delete components;
		}
	}

public: System::String^ initialPlayer;
public: System::String^ initialGame;
public: System::String^ selectedPlayer;
public: System::String^ selectedGame;

private: array<System::String^>^ availablePlayers;
public: array<System::String^>^ selectedDates;

private: System::Windows::Forms::Label^  label1;
protected:
private: System::Windows::Forms::Label^  label2;
private: System::Windows::Forms::Label^  label3;
private: System::Windows::Forms::ComboBox^  playerComboBox;
private: System::Windows::Forms::ComboBox^  gameComboBox;
private: System::Windows::Forms::ListBox^  dateListBox;
private: System::Windows::Forms::Label^  label4;
private: System::Windows::Forms::Label^  label5;
private: System::Windows::Forms::Button^  okButton;
private: System::Windows::Forms::Button^  cancelButton;


private:
	/// <summary>
	/// Required designer variable.
	/// </summary>
	System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
```

```
		/// <summary>
		/// Required method for Designer support - do not modify
		/// the contents of this method with the code editor.
		/// </summary>
		void InitializeComponent(void)
		{
			this->label1 = (gcnew System::Windows::Forms::Label());
			this->label2 = (gcnew System::Windows::Forms::Label());
			this->label3 = (gcnew System::Windows::Forms::Label());
			this->playerComboBox = (gcnew System::Windows::Forms::ComboBox());
			this->gameComboBox = (gcnew System::Windows::Forms::ComboBox());
			this->dateListBox = (gcnew System::Windows::Forms::ListBox());
			this->label4 = (gcnew System::Windows::Forms::Label());
			this->label5 = (gcnew System::Windows::Forms::Label());
			this->okButton = (gcnew System::Windows::Forms::Button());
			this->cancelButton = (gcnew System::Windows::Forms::Button());
			this->SuspendLayout();
			//
			// label1
			//
			this->label1->AutoSize = true;
			this->label1->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 18,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
				static_cast<System::Byte>(0)));
			this->label1->Location = System::Drawing::Point(27, 77);
			this->label1->Margin = System::Windows::Forms::Padding(6, 0, 6, 0);
			this->label1->Name = L"label1";
			this->label1->Size = System::Drawing::Size(87, 29);
			this->label1->TabIndex = 0;
			this->label1->Text = L"Player:";
			//
			// label2
			//
			this->label2->AutoSize = true;
			this->label2->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 18,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
				static_cast<System::Byte>(0)));
			this->label2->Location = System::Drawing::Point(27, 160);
			this->label2->Margin = System::Windows::Forms::Padding(6, 0, 6, 0);
			this->label2->Name = L"label2";
			this->label2->Size = System::Drawing::Size(84, 29);
			this->label2->TabIndex = 1;
			this->label2->Text = L"Game:";
			//
			// label3
			//
			this->label3->AutoSize = true;
			this->label3->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 18,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
				static_cast<System::Byte>(0)));
			this->label3->Location = System::Drawing::Point(355, 87);
			this->label3->Margin = System::Windows::Forms::Padding(6, 0, 6, 0);
			this->label3->Name = L"label3";
			this->label3->Size = System::Drawing::Size(97, 29);
			this->label3->TabIndex = 2;
			this->label3->Text = L"Date(s):";
			//
			// playerComboBox
			//
			this->playerComboBox->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 14.25F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
				static_cast<System::Byte>(0)));
```

```
            this->playerComboBox->FormattingEnabled = true;
            this->playerComboBox->Location = System::Drawing::Point(32, 122);
            this->playerComboBox->Margin = System::Windows::Forms::Padding(6);
            this->playerComboBox->Name = L"playerComboBox";
            this->playerComboBox->Size = System::Drawing::Size(294, 32);
            this->playerComboBox->TabIndex = 3;

            this->playerComboBox->TextChanged += gcnew System::EventHandler(this,
&selectOldResultsForm::playerComboBox_TextChanged);
            this->playerComboBox->Click += gcnew System::EventHandler(this,
&selectOldResultsForm::playerComboBox_Click);
            //
            // gameComboBox
            //
            this->gameComboBox->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 14.25F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                static_cast<System::Byte>(0)));
            this->gameComboBox->FormattingEnabled = true;
            this->gameComboBox->Location = System::Drawing::Point(32, 202);
            this->gameComboBox->Margin = System::Windows::Forms::Padding(6);
            this->gameComboBox->Name = L"gameComboBox";
            this->gameComboBox->Size = System::Drawing::Size(291, 32);
            this->gameComboBox->TabIndex = 4;
            this->gameComboBox->DropDown += gcnew System::EventHandler(this,
&selectOldResultsForm::gameComboBox_DropDown);
            this->gameComboBox->TextChanged += gcnew System::EventHandler(this,
&selectOldResultsForm::gameComboBox_TextChanged);
            this->gameComboBox->Click += gcnew System::EventHandler(this,
&selectOldResultsForm::gameComboBox_Click);
            //
            // dateListBox
            //
            this->dateListBox->FormattingEnabled = true;
            this->dateListBox->ItemHeight = 24;
            this->dateListBox->Location = System::Drawing::Point(461, 87);
            this->dateListBox->Name = L"dateListBox";
            this->dateListBox->SelectionMode = System::Windows::Forms::SelectionMode::MultiExtended;
            this->dateListBox->Size = System::Drawing::Size(309, 148);
            this->dateListBox->TabIndex = 6;
            //
            // label4
            //
            this->label4->AutoSize = true;
            this->label4->Location = System::Drawing::Point(28, 13);
            this->label4->Name = L"label4";
            this->label4->Size = System::Drawing::Size(560, 24);
            this->label4->TabIndex = 7;
            this->label4->Text = L"Please select a player and game. Available dates will be provided.";
            //
            // label5
            //
            this->label5->AutoSize = true;
            this->label5->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 11.25F,
System::Drawing::FontStyle::Regular, System::Drawing::GraphicsUnit::Point,
                static_cast<System::Byte>(0)));
            this->label5->Location = System::Drawing::Point(32, 41);
            this->label5->Name = L"label5";
            this->label5->Size = System::Drawing::Size(360, 18);
            this->label5->TabIndex = 8;
            this->label5->Text = L"*Mutiple dates may be selected by holding Shift or Ctrl";
            //
            // okButton
```

```
                                //
                                this->okButton->Location = System::Drawing::Point(261, 267);
                                this->okButton->Name = L"okButton";
                                this->okButton->Size = System::Drawing::Size(115, 41);
                                this->okButton->TabIndex = 9;
                                this->okButton->Text = L"OK";
                                this->okButton->UseVisualStyleBackColor = true;
                                this->okButton->Click += gcnew System::EventHandler(this, &selectOldResultsForm::okButton_Click);
                                //
                                // cancelButton
                                //
                                this->cancelButton->Location = System::Drawing::Point(382, 267);
                                this->cancelButton->Name = L"cancelButton";
                                this->cancelButton->Size = System::Drawing::Size(111, 41);
                                this->cancelButton->TabIndex = 10;
                                this->cancelButton->Text = L"Cancel";
                                this->cancelButton->UseVisualStyleBackColor = true;
                                this->cancelButton->Click += gcnew System::EventHandler(this, &selectOldResultsForm::cancelButton_Click);
                                //
                                // selectOldResultsForm
                                //
                                this->AutoScaleDimensions = System::Drawing::SizeF(11, 24);
                                this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
                                this->ClientSize = System::Drawing::Size(807, 320);
                                this->Controls->Add(this->cancelButton);
                                this->Controls->Add(this->okButton);
                                this->Controls->Add(this->label5);
                                this->Controls->Add(this->label4);
                                this->Controls->Add(this->dateListBox);
                                this->Controls->Add(this->gameComboBox);
                                this->Controls->Add(this->playerComboBox);
                                this->Controls->Add(this->label3);
                                this->Controls->Add(this->label2);
                                this->Controls->Add(this->label1);
                                this->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 14.25F, System::Drawing::FontStyle::Regular,
System::Drawing::GraphicsUnit::Point,
                                        static_cast<System::Byte>(0)));
                                this->Margin = System::Windows::Forms::Padding(6);
                                this->Name = L"selectOldResultsForm";
                                this->Text = L"Load Records";
                                this->Load += gcnew System::EventHandler(this, &selectOldResultsForm::selectOldResultsForm_Load);
                                this->ResumeLayout(false);
                                this->PerformLayout();

                        }
#pragma endregion

                        // this will run the first time the form opens up; when the parent code calls ShowDialog()
                private: System::Void selectOldResultsForm_Load(System::Object^  sender, System::EventArgs^  e) {
                                // current player and current game should have been passed into this form; autopopulate the relevent
textboxes with them
                                this->playerComboBox->Text = this->initialPlayer;
                                this->gameComboBox->Text = this->initialGame;

                                // find all of the available patients and save them for the drop down menu
                                array<System::String^>^ patientNames = System::IO::Directory::GetDirectories(
Constants::RESULTS_DIRECTORY );
                                for (int i = 0; i < patientNames->Length; i++) {
                                        patientNames[i] = System::IO::Path::GetFileNameWithoutExtension(patientNames[i]);
                                }
                                this->availablePlayers = patientNames;
                                playerComboBox->Items->AddRange(availablePlayers);
```

```
                        // find possible dates for the current user and game
                        findDatesFromUserAndGame();

                }

private: System::Void playerComboBox_Click(System::Object^  sender, System::EventArgs^  e) {
                }
private: System::Void gameComboBox_Click(System::Object^  sender, System::EventArgs^  e) {
                }

// if user clicks, okay, pull selected data from boxes and return dialogresult::Ok
private: System::Void okButton_Click(System::Object^  sender, System::EventArgs^  e) {

                        // if a date hasn't been selected, alert user and return
                        if (this->dateListBox->SelectedItems->Count == 0) {
                                System::Windows::Forms::MessageBox::Show("Please select Date(s)");
                                return;
                        }

                        // pull player name, game, and dates from text boxes and store as public variables
                        selectedPlayer = getPlayerName();
                        selectedGame = this->gameComboBox->Text;
                        // need to convert datelistbox type to Strings
                        System::Windows::Forms::ListBox::SelectedObjectCollection^ collection = this->dateListBox->SelectedItems;
                        selectedDates = gcnew array<System::String^>(collection->Count);
                        for (int i = 0; i < collection->Count; i++) {
                                System::String^ txt = dateListBox->GetItemText(collection[i]);
                                selectedDates[i] = txt;
                        }

                        // return dialogresult OK. Form should now close.
                        this->DialogResult = System::Windows::Forms::DialogResult::OK;
                }

// if user clicks cancel, return dialogresult::Cancel, which will close the form
private: System::Void cancelButton_Click(System::Object^  sender, System::EventArgs^  e) {
                        this->DialogResult = System::Windows::Forms::DialogResult::Cancel;
                }

                // find all files for the selected user, return their paths as strings
private: array<System::String^>^ findFileNamesFromUser() {

                        System::String^ player = getPlayerName();
                        System::String^ filePath = Constants::RESULTS_DIRECTORY + player;

                        // find all files in selected user's folder, if the folder exists
                        array<System::String^>^ fileNames = gcnew array<System::String^>(0);
                        if (System::IO::Directory::Exists(filePath)) {
                                fileNames = System::IO::Directory::GetFiles( filePath );
                        }
                        else if (!System::IO::Directory::Exists(filePath)) {
                                return fileNames;
                        }
                        // cut the paths and extensions off
                        for (int i = 0; i < fileNames->Length; i++) {
                                        fileNames[i] = System::IO::Path::GetFileNameWithoutExtension(fileNames[i]);
                        }
                        return fileNames;
                }

private: System::Void findDatesFromUserAndGame() {
```

```
                        // clear out all old dates in box
                        dateListBox->Items->Clear();

                        System::String^ player = getPlayerName();
                        System::String^ game = gameComboBox->Text;

                       // find all file names for user
                       array<System::String^>^ fileNames = findFileNamesFromUser();
                       if (fileNames->Length == 0 ) { return; }

                       System::String^ delimStr = "_";
                       array<Char>^ delimiter = delimStr->ToCharArray( );

                        // Parse each file name for game and date
                        for each (System::String^ file in fileNames) {
                                array<System::String^>^ tokens = file->Split(delimiter);
                                System::String^ gameToken = tokens[1];

                                // if the game doesn't match, move on
                                if (!gameToken->Equals(game)) { continue; }
                                System::String^ dateStr = tokens[3] + " " + tokens[4] + " " + tokens[2];
                                dateListBox->Items->Add(dateStr);
                        }
                }

                // if user uses the dropdown for the games, find the games for the given user and add to dropdown
private: System::Void gameComboBox_DropDown(System::Object^  sender, System::EventArgs^  e) {

                        // clear contents of drop down
                        gameComboBox->Items->Clear();

                       array<System::String^>^ fileNames = findFileNamesFromUser();

                       System::String^ delimStr = "_";
                       array<Char>^ delimiter = delimStr->ToCharArray( );

                       // now parse each one for game and date
                        for each (System::String^ file in fileNames) {
                                array<System::String^>^ tokens = file->Split(delimiter);
                                System::String^ game = tokens[1];
                                // now add game to drop down menu if it's not already there
                                if (!gameComboBox->Items->Contains(game)) {
                                        gameComboBox->Items->Add(game);
                                }
                        }
                }

// if user changes text in player box, recalculate possible games and dates
private: System::Void playerComboBox_TextChanged(System::Object^  sender, System::EventArgs^  e) {
                        // if player changed, reset dates to empty
                        dateListBox->Items->Clear();

                        // then refill dates if the game is present
                        findDatesFromUserAndGame();
                }

// if user changes text in game box, recalculate possible games and dates
private: System::Void gameComboBox_TextChanged(System::Object^  sender, System::EventArgs^  e) {
                        // if game changed, reset dates to empty
                        dateListBox->Items->Clear();
```

```
                        // then refill dates if the player is present
                        findDatesFromUserAndGame();
                }

private: System::String^ getPlayerName()
                {
                        return playerComboBox->Text->ToLower();
                }

};
}
```

GAMEBOARD.H

```
/* These classes hold the data for each individual gameboard. There is a GameBase class which has all the basic information in it,
and classes for individual game types can derive from it. Only the KnobPuzzle has been developed this year. An instance of the KnobPuzzle
class contains all information as to the name, shape, location, placement, etc. of each puzzle piece for a given board.
This instance will be passed all around through the program, to be used by tracking, scorekeeping, and the GUI
*/

#pragma once
#include <Windows.h>
#using <System.dll>

#ifndef FILE_H
#define FILE_H

using namespace System;
using namespace System::Collections::Generic;

// generic game class, with stop/start info etc. Different types of games can inherit from this class
public ref class GameBase {
public:

        GameBase()          { puzzleName = ""; puzzleType = ""; LevelOfDifficulty = 0; END_GAME = false;}

        // basic gets and sets
        virtual void setName(System::String^ Name) { this->puzzleName = Name; }
        System::String^ getName()                       { return puzzleName; }
        void setType(System::String^ type)     { this->puzzleType = type; }
        System::String^ getType()                       { return puzzleType; }
        void setLevelOfDifficulty(int level)   { this->LevelOfDifficulty = level; }
        int getLevelOfDifficulty()                      { return this->LevelOfDifficulty; }

        // set or reset END_GAME. This is a communication link between the main GUI and the RunTracking class
        void setEndGame()                               { this->END_GAME = true; }
        void resetEndGame()                                     { this->END_GAME = false; }

        // check if the game is over, or if there has been an error
        bool isEndGame()                        { return this->END_GAME; }
        void setError()                                 { this->Error = true; }
        bool checkIfError()                             { return this->Error; }

protected:

        HANDLE myMutex;
        bool Error;
        bool END_GAME;
        System::String^ puzzleName;
        System::String^ puzzleType;
```

```
	int LevelOfDifficulty;

};


//-----------------------------------------------------------------------
//-----------------------------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------------------


// Class specific to Knob Puzzle; Inherits from GameBase. initializes and manages list of puzzle pieces for a knob puzzle
public ref class KnobPuzzle : public GameBase
{

public:

	// constructors
	KnobPuzzle(void);
	KnobPuzzle(System::String^ code);
	KnobPuzzle(const KnobPuzzle^) {} // copy constructor 1 : pass in KnobPuzzle^
	KnobPuzzle(const KnobPuzzle%) {} // copy constructor 2 : pass in KnobPuzzle
	~KnobPuzzle(void); // make sure mutex gets released

	// access class data from outside
	int setGame(System::String^ code);

	// check if puzzle has been initialized. Overloaded.
	bool checkIsInitialized();   // any game loaded
	bool checkIsInitialized(System::String^ code);   // game with given name has been loaded
	bool checkIsInitialized(System::String^ code, int level);   // game with given name and level of difficulty has been loaded

	List<PuzzlePiece^>^ getPieceList() { return this->pieceList; }

	// write out current class data to file (normally newly calibrated values)
	int SaveCalibrationSettings();

protected:

private:
	// individual piece information
	List<PuzzlePiece^>^ pieceList;

	void Initialize();
	void LookUpGame(System::String^ code);
	int ParseShapeInformation(array<System::String^>^ tokens, PuzzlePiece^ piece);
	int WriteSettingsToFile();


};

#endif
```

GAMEBOARD.CPP

```
/* Definitions for functions belonging to the KnobPuzzle and GameBase classes. Ex. initializers, loading the game, writing out
calibration settings
*/

#include "stdafx.h"

#include "Functions.h"
```

```
using namespace System;
using namespace System::Collections::Generic;
using namespace System::Windows::Forms;

//-----------------------------------------------------------------------------------------------------
// set initial values for knob puzzle so we don't run into errors
void KnobPuzzle::Initialize()
{
	this->Error = false;  // error boolean will be set if something goes wrong somewhere
	this->puzzleName = gcnew System::String("KNOBPUZZLE");
	this->puzzleType = gcnew System::String("KNOBPUZZLE");
	this->LevelOfDifficulty = 0;
	this->pieceList = gcnew List<PuzzlePiece^>();
	this->myMutex = CreateMutex(NULL, FALSE, (LPCWSTR) "KnobPuzzle Class Mutex");
}

//-----------------------------------------------------------------------------------------------------
// initialize an empty knobpuzzle
KnobPuzzle::KnobPuzzle(void)
{
	Initialize();
}

//-----------------------------------------------------------------------------------------------------
KnobPuzzle::KnobPuzzle(System::String^ code)
{
	setGame(code);
}

// initialize a knob puzzle to a code. It will look into the game file for the matching code
int KnobPuzzle::setGame(System::String^ code) {

	// lock mutex for loading game data
	WaitForSingleObject(this->myMutex, INFINITE);
	this->Error = false; // if there was an error before, setGame might fix it

	Initialize(); // re-initialize all class data
	this->puzzleName = code; // update puzzle name
	LookUpGame(code); // this will fill up the knobpuzzle^ class properties

	// release mutex
	ReleaseMutex(myMutex);

	if (this->Error) {
		MessageBox::Show("KnobPuzzle::SetGame() - Couldn't initialize game. Check code and/or game file.");
		return -1;
	}
	return 0;
}
//-----------------------------------------------------------------------------------------------------
// I should really learn how to use destructors. I have no idea what this does.
KnobPuzzle::~KnobPuzzle(void)
{
}
//-----------------------------------------------------------------------------------------------------
// Look in game input file for knob puzzle matching given code. Pull all game information from file.
//*** CHANGES to input file must be dealt with there ******
void KnobPuzzle::LookUpGame(System::String^ code)
{
	// lock down thread while looking up game - just in case we use more threading in future
	WaitForSingleObject(this->myMutex, INFINITE);
```

```
	// find path for input file from game code
	System::String^ calibratedFile = getCalibratedInputPath(code);
	System::String^	defaultFile = getDefaultInputPath(code);
	System::String^ inputFile = calibratedFile;

	Console::WriteLine("KnobPuzzle::LookUpGame(): Calibrated Input File Path : " + calibratedFile + "\n Default Input File Path : " +
defaultFile);

	// if can't find calibrated file, then use default file instead
	if (!System::IO::File::Exists(calibratedFile)) {
		// if can't find calibrated file, then use default file instead
		Console::WriteLine("KnobPuzzle::LookUpGame() - Could not find calibrated file. Will use default instead.");
		inputFile = defaultFile;
		if (!System::IO::File::Exists(defaultFile)) {
			Console::WriteLine("KnobPuzzle::LookUpGame() - Could not find either calibrated or default game file");
			this->Error = true;
			ReleaseMutex(myMutex);
			return;
		}
	}

	// pull all strings from file
	array<System::String^>^ stringArray = getStringArrayFromFile(inputFile);
	if (stringArray == nullptr || stringArray[0]->Equals("ERROR")) {
		MessageBox::Show("KnobPuzzle::LookUpGame -Could not pull strings from Game file");
		this->Error = true;
		ReleaseMutex(myMutex);
		return;
	}

	// Initialize containers
	List<PuzzlePiece^>^ PieceList = gcnew List<PuzzlePiece^>(0);
	array<System::String^>^ tokens;
	int x, y, hmin, smin, vmin, hmax, smax, vmax;
	List<Int32>^ HSVmin;
	List<Int32>^ HSVmax;

	// go through each line in our section of input file. **THIS IS HARDCODED - changes to input file must be dealt with there
	// **NOTE** level of difficulty is not currently parsed. It's there for the future.
	int index = 0;
	System::String^ line = stringArray[index];
	Console::WriteLine("KnobPuzzle::LookUpGame() : parsing puzzle pieces from input file");
	while(!line->Contains("----") && index < stringArray->Length) {
		line = stringArray[index++];
		Console::WriteLine(line);
		if (line->Length == 0) {continue;} // if line empty, continue
		tokens = line->Split(); // break line into words

		// Pull PuzzlePiece^ tracking information
		if (tokens[0]->Equals("LOC") && tokens[3]->Equals("COLOR") && tokens->Length >= 11 ) {
			//Sample Format:::    LOC 1 1 COLOR 100 100 150 200 200 200 SQUARE
			System::String^ pieceName = tokens[10];
			bool try1 = Int32::TryParse(tokens[1], x);
			bool try2 = Int32::TryParse(tokens[2], y);
			bool try3 = Int32::TryParse(tokens[4], hmin);
			bool try4 = Int32::TryParse(tokens[5], smin);
			bool try5 = Int32::TryParse(tokens[6], vmin);
			bool try6 = Int32::TryParse(tokens[7], hmax);
			bool try7 = Int32::TryParse(tokens[8], smax);
			bool try8 = Int32::TryParse(tokens[9], vmax);
			// check if the parsing worked
			if (!try1 || !try2 || !try3 || !try4 || !try5 || !try6 || !try7 || !try8) {
```

```
                                        MessageBox::Show("Error: An inappropriate HSV value was found in line:\n" + line);
                                        Console::WriteLine("KnobPuzzle::LookUpGame() : Error: An inappropriate HSV value was found in
line:\n" + line);
                                        this->Error = true;
                                        ReleaseMutex(myMutex);
                                        return;
                                }
                                // check if the HSV values seem reasonable
                                if (hmin < 0 || smin < 0 || vmin < 0 || hmax > 256 || smax > 256 || vmax > 256) {
                                        MessageBox::Show("Error: HSV value error - min values must be 0 or greater, max values must be between
0 and 256. \n" + line);
                                        Console::WriteLine("KnobPuzzle::LookUpGame() : Error : HSV value - min values must be 0 or greater,
max values must be between 0 and 256.\n" + line);
                                        this->Error = true;
                                        ReleaseMutex(myMutex);
                                        return;
                                }
                                // plug tracking information into new puzzle piece
                                HSVmin = gcnew List<Int32>(3); // HSV min values
                                HSVmin->Add(hmin); HSVmin->Add(smin); HSVmin->Add(vmin);
                                HSVmax = gcnew List<Int32>(3); // HSV max values
                                HSVmax->Add(hmax); HSVmax->Add(smax); HSVmax->Add(vmax);
                                PuzzlePiece^ newPiece = gcnew PuzzlePiece(pieceName, HSVmin, HSVmax);
                                newPiece->setDestPos(x,y);   // destination location of piece

                                // now parse shape drawing information.
                                int success = ParseShapeInformation(tokens, newPiece); // this function automatically adds the shape info to the puzzle
piece
                                if (success != 0) {
                                        Console::WriteLine("KnobPuzzle::LookUpGame() : Error: Incorrect shape drawing information: \n Line: " +
line);
                                        MessageBox::Show("Error: Incorrect shape drawing information");
                                        this->Error = true;
                                        ReleaseMutex(myMutex);
                                        return;
                                }

                                // if all went well, add new piece to pieceList
                                PieceList->Add(newPiece);
                        }

                        // if the input line was missing values, throw error
                        else if (tokens[0]->Equals("LOC") && tokens[3]->Equals("COLOR") && tokens->Length < 11 ) {
                                Console::WriteLine("KnobPuzzle::LookUpGame() : Error: Incomplete piece information: \n Line: " + line);
                                MessageBox::Show("Error: Incomplete piece information: \n Line: " + line);
                                this->Error = true;
                                ReleaseMutex(myMutex);
                                return;
                        }
                }

                // if puzzle still has has 0 pieces after reading through file, throw error
                if (PieceList->Count == 0) {
                        Console::WriteLine("KnobPuzzle::LookUpGame() : Puzzle doesn't have pieces");
                        MessageBox::Show("Error: Puzzle doesn't have pieces");
                        this->Error = true;
                        ReleaseMutex(myMutex);
                        return;
                }

                // load piece data into mother class if all went well
                this->pieceList = PieceList;
```

```
		ReleaseMutex(myMutex);
}

//-------------------------------------------------------------------------------------------------------
// pull shape-drawing information from puzzle file line
int KnobPuzzle::ParseShapeInformation(array<System::String^>^ tokens, PuzzlePiece^ piece)
{
		System::String^ shapeType = piece->getName();
		int point_x, point_y, height, width, length, radius, side_length, bottom_length;
		bool try1 = true; bool try2 = true; bool try3 = true; bool try4 = true;

		if (shapeType->Equals("Circle")) {
			bool try1 = Int32::TryParse(tokens[11], point_x);
			bool try2 = Int32::TryParse(tokens[12], point_y);
			bool try3 = Int32::TryParse(tokens[13], radius);
			if (!try1 || !try2 || !try3 || !try4) {
				MessageBox::Show("KnobPuzzle::ParseShapeInformation(): Inappropriate shape drawing information for piece: " +
piece->getName());
				this->Error = true;
				return -1;
			}
			piece->setShapePoint(point_x, point_y);
			piece->setShapeRadius(radius);
		}
		else if (shapeType->Equals("Rectangle")) {
			bool try1 = Int32::TryParse(tokens[11], point_x);
			bool try2 = Int32::TryParse(tokens[12], point_y);
			bool try3 = Int32::TryParse(tokens[13], width);
			bool try4 = Int32::TryParse(tokens[14], height);
			if (!try1 || !try2 || !try3 || !try4) {
				MessageBox::Show("KnobPuzzle::ParseShapeInformation():: Inappropriate shape drawing information for piece: " +
piece->getName());
				this->Error = true;
				return -1;
			}
			piece->setShapePoint(point_x, point_y);
			piece->setShapeHeight(height);
			piece->setShapeWidth(width);
		}
		else if (shapeType->Equals("Square")) {
			bool try1 = Int32::TryParse(tokens[11], point_x);
			bool try2 = Int32::TryParse(tokens[12], point_y);
			bool try3 = Int32::TryParse(tokens[13], width);
			if (!try1 || !try2 || !try3 || !try4) {
				MessageBox::Show("KnobPuzzle::ParseShapeInformation():: Inappropriate shape drawing information for piece: " +
piece->getName());
				this->Error = true;
				return -1;
			}
			piece->setShapePoint(point_x, point_y);
			piece->setShapeWidth(width);
		}
		else if (shapeType->Equals("Triangle") || shapeType->Equals("Pentagon")) {
			bool try1 = Int32::TryParse(tokens[11], point_x);
			bool try2 = Int32::TryParse(tokens[12], point_y);
			bool try3 = Int32::TryParse(tokens[13], length);
			if (!try1 || !try2 || !try3 || !try4) {
				MessageBox::Show("KnobPuzzle::ParseShapeInformation():: Inappropriate shape drawing information for piece: " +
piece->getName());
				this->Error = true;
				return -1;
```

```
                }
                piece->setShapePoint(point_x, point_y);
                piece->setShapeLength(length);
        }
        else if (shapeType->Equals("Isosceles")) {
                //bool try1 = Int32::TryParse(tokens[11], point_x);
                //bool try2 = Int32::TryParse(tokens[12], point_y);
                //bool try3 = Int32::TryParse(tokens[13], side_length);
                //bool try4 = Int32::TryParse(tokens[14], bottom_length);
                //if (!try1 || !try2 || !try3 || !try4) {
                //      MessageBox::Show("KnobPuzzle::ParseShapeInformation():: Inappropriate shape drawing information for piece: " +
piece->getName());
                //      this->Error = true;
                //      return -1;
                //}
                //piece->setShapePoint(point_x, point_y);
                //piece->setShapeHeight(side_length);
                //piece->setShapeWidth(bottom_length);
        }
        else if (shapeType->Equals("House")) {
                //bool try1 = Int32::TryParse(tokens[11], point_x);
                //bool try2 = Int32::TryParse(tokens[12], point_y);
                //bool try3 = Int32::TryParse(tokens[13], side_length);
                //bool try4 = Int32::TryParse(tokens[14], bottom_length);
                //if (!try1 || !try2 || !try3 || !try4) {
                //      MessageBox::Show("KnobPuzzle::ParseShapeInformation():: Inappropriate shape drawing information for piece: " +
piece->getName());
                //      this->Error = true;
                //      return -1;
                //}
                //piece->setShapePoint(point_x, point_y);
                //piece->setShapeHeight(side_length);
                //piece->setShapeWidth(bottom_length);
        }
        else if (shapeType->Equals("Tree")) {
                // CURRENTLY THIS IS HARDCODED INTO THE SHAPE DRAWING FUNCTION
                //bool try1 = Int32::TryParse(tokens[11], point_x);
                //bool try2 = Int32::TryParse(tokens[12], point_y);
                //bool try3 = Int32::TryParse(tokens[13], radius);
                //int corner_x, corner_y;
                ////piece->setShapePoint(point_x, point_y);
                //bool try1 = Int32::TryParse(tokens[11], corner_x);
                //bool try2 = Int32::TryParse(tokens[12], corner_y);
                //bool try3 = Int32::TryParse(tokens[13], width);
                //bool try4 = Int32::TryParse(tokens[14], height);
                //if (!try1 || !try2 || !try3 || !try4) {
                //      MessageBox::Show("KnobPuzzle::ParseShapeInformation():: Inappropriate shape drawing information for piece: " +
piece->getName());
                //      this->Error = true;
                //      return -1;
                //}
                //piece->setShapePoint(point_x, point_y);
                //piece->setShapeHeight(height);
                //piece->setShapeWidth(width);
        }
        else if (shapeType->Equals("Door")) {
        }
        else if (shapeType->Equals("Sun")) {
        }
        else {
                        MessageBox::Show("KnobPuzzle::ParseShapeInformation():: Not a recognized shape: " + piece->getName());
                        this->Error = true;
```

```
                        return -1;
                }
        // if any of the above parsing failed, return an error
        if (!try1 || !try2 || !try3 ||!try4) { return -1; }

        return 0;
}

//-------------------------------------------------------------------------------------------------------//
// user saves calibration settings
int KnobPuzzle::SaveCalibrationSettings() {
        int success = WriteSettingsToFile();
        return success;
}

//-------------------------------------------------------------------------------------------------
// Write the current KnobPuzzle calibration settings to the calibration file. THIS FUNCTION NEEDS TO BE REVAMPED TO CONSTRUCT NEW FILES WITHOUT TEMPLATE
int KnobPuzzle::WriteSettingsToFile() {

        // lock everything down to this thread
        myMutex = CreateMutex(NULL, FALSE, (LPCWSTR) "Writing settings to File");
        WaitForSingleObject(myMutex, INFINITE);

        System::Diagnostics::Debug::WriteLine("Saving calibration settings to file");
        Console::WriteLine("KnobPuzzle::WriteSettingsToFile(): Saving calibration settings to file");
        int success = 0;

        // Find default code file to use as template for changes
        System::String^ inputFile = getDefaultInputPath(this->getName());

        // if default code file is missing, use calibrated as template instead
        if (!System::IO::File::Exists(inputFile)) {
                System::Windows::Forms::MessageBox::Show("KnobPuzzle::WriteSettingsToFile(): Warning - can't find default input file.");
                inputFile = getCalibratedInputPath(this->getName());
                if (!System::IO::File::Exists(inputFile)) {
                        System::Windows::Forms::MessageBox::Show("KnobPuzzle::WriteSettingsToFile(): Could not find calibrated or default game file");
                        this->Error = true;
                        ReleaseMutex(myMutex);
                        return -1;
                }
        }
        //*** IN FUTURE WOULD LIKE TO HAVE CODE RECONSTRUCT FILE FROM BASE UP; NOT NEED TEMPLATE. That will keep things more consistent
        // pull in all strings from code file to serve as template for changes
        array<System::String^>^ stringArray = getStringArrayFromFile(inputFile);
        if (stringArray[0]->Equals("ERROR")) {
                System::Windows::Forms::MessageBox::Show("KnobPuzzle::WriteSettingsToFile(): Could not load game file");
                this->Error = true;
                ReleaseMutex(myMutex);
                return -1;
        }

        System::String^ line = "";

        System::Diagnostics::Debug::WriteLine("Constructing new file strings");
        Console::WriteLine("Constructing new file strings");
        int puzzlePieceIndex = 0;

        // loop through pieces and write them out
        for (int i = 0; i < stringArray->Length ; i++) {
```

```
                line = stringArray[i];
                //if original line does not begin with LOC , then continue to next line
                if (line->Length < 4 || !line->Substring(0,4)->Equals("LOC ")) {continue;}

                // get the relevent information from the current puzzle piece
                PuzzlePiece^ currentPiece = this->getPieceList()[puzzlePieceIndex];
                int xdest = currentPiece->getXDest();
                int ydest = currentPiece->getYDest();
                List<int>^ HSVmin = currentPiece->getHSVmin();
                List<int>^ HSVmax = currentPiece->getHSVmax();
                int Hmin = HSVmin[0];
                int Smin = HSVmin[1];
                int Vmin = HSVmin[2];
                int Hmax = HSVmax[0];
                int Smax = HSVmax[1];
                int Vmax = HSVmax[2];
                int shapePointX = currentPiece->getShapePointX();
                int shapePointY = currentPiece->getShapePointY();
                System::String^ name = currentPiece->getName();

                // cat puzzle piece information together into a single line in the proper format
                System::String^ constructor = "LOC " + xdest + " " + ydest + " COLOR " + Hmin + " " + Smin + " " + Vmin
                                                                        + " " + Hmax + " " + Smax + " " + Vmax + " " + name + " " +
shapePointX  + " " + shapePointY;
                if (name->Equals("Circle")) {
                        constructor = constructor  + " " + currentPiece->getShapeRadius();
                }
                else if (name->Equals("Square")) {
                        constructor = constructor  + " " + currentPiece->getShapeWidth();
                }
                else if (name->Equals("Rectangle")) {
                        constructor = constructor  + " " + currentPiece->getShapeWidth() + " " + currentPiece->getShapeHeight();
                }
                else if (name->Equals("Triangle") || name->Equals("Pentagon")) {
                        constructor = constructor  + " " + currentPiece->getShapeLength();
                }
                // copy the line over the old line in the array of file strings
                stringArray[i] = constructor;
                System::Diagnostics::Debug::WriteLine(stringArray[i]);

                // go to next puzzle piece
                puzzlePieceIndex++;
        }
        // write out final string array to calibrated file
        writeStringArrayToFile(stringArray, getCalibratedInputPath(this->getName()));
        Console::WriteLine("GameBoard.cpp::WriteSettingsToFile() : writing new settings to file : " + getCalibratedInputPath(this->getName()));

        // unlock thread
        ReleaseMutex(myMutex);

        return success;
}

//-----------------------------------------------------------------------------------------------
// Check if the puzzle has been initialized and has pieces and is the right game
bool KnobPuzzle::checkIsInitialized(System::String^ code) {
        // check for errors or no pieces, and make sure name matches
        if (!this->Error && this->pieceList->Count > 0 && this->puzzleName->Equals(code)) { return true; }
        else { return false; }
}
//-----------------------------------------------------------------------------------------------
// Check if the puzzle has been initialized and has pieces, is the right game and right level
```

```
bool KnobPuzzle::checkIsInitialized(System::String^ code, int level) {
	// check for errors or no pieces, and make sure name matches
	if (!this->Error && this->pieceList->Count > 0 && this->puzzleName->Equals(code) && this->getLevelOfDifficulty() == level) { return
true; }
	else { return false; }
}

//-----------------------------------------------------------------------------------------------------
// Check if the puzzle has been initialized and has pieces
bool KnobPuzzle::checkIsInitialized() {
	// see if the error is set or if there is 0 pieces (bad)
	if (!this->Error && this->pieceList->Count > 0) { return true; }
	else { return false; }
}
```

PUZZLEPIECE.H

```
/* This class hold the data for an individual puzzle piece. This class is currently tailored for the KnobPuzzle, but extensions could be made.
Basic information included is the shape of the piece, its color, its destination coordinates, and the time at which it was placed.
*/

#pragma once

#include <Windows.h>
#using <System.dll>

using namespace System;
using namespace System::Collections::Generic;

public ref class PuzzlePiece
{
private:

	HANDLE myMutex;
	int x_pos, y_pos;
	int x_dest, y_dest;
	List<int>^ HSV_min;
	List<int>^ HSV_max;

	// shape drawing data (add as necessary)
	int shape_point_x;
	int shape_point_y;
	int shape_width;
	int shape_height;
	int shape_length;
	int shape_radius;

	void Initialize();

public:

	DateTime timePlaced;
	bool isPlaced;
	System::String^ name;

	PuzzlePiece(void);
	PuzzlePiece(System::String^ piece_name);
	PuzzlePiece(System::String^ piece_name, List<int>^ HSVmin, List<int>^ HSVmax);
	PuzzlePiece(System::String^ piece_name, List<int>^ HSVmin, List<int>^ HSVmax, int xdest, int ydest);

	~PuzzlePiece(void);
```

```
        // Get and Set Timing/Gameplay Data

        void setTimePlacedToNow();
        DateTime getTimePlaced() { return timePlaced; }
        void setTimePlaced(DateTime tim) { this->timePlaced = tim; }


        // Get and Set Tracking Data

        int getXPos()       {return x_pos;}
        int getXDest()      { return x_dest;}
        void setXPos(int x)  {x_pos = x;}
        void setXDest(int x) {x_dest = x;}

        int getYPos()       {return y_pos;}
        int getYDest()    { return y_dest;}
        void setYPos(int y)  {y_pos = y;}
        void setYDest(int y) {y_dest = y;}

        void setPos(int x, int y)     { x_pos = x; y_pos = y; }
        void setDestPos(int x, int y) { x_dest = x; y_dest = y; }

        List<int>^ getHSVmin()        {return HSV_min;}
        void setHSVmin(List<int>^ min) {HSV_min = min;}

        List<int>^ getHSVmax()        {return HSV_max;}
        void setHSVmax(List<int>^ max) {HSV_max = max;}

        System::String^ getName()      {return name;}
        void setName(System::String^ t) {name = t;}

        // Get and Set Shape Drawing Data

        void setShapePoint(int x, int y) { shape_point_x = x; shape_point_y = y; }
        int getShapePointX()   { return shape_point_x; }
        int getShapePointY()   { return shape_point_y; }

        void setShapeHeight(int h);
        void setShapeWidth(int w);
        void setShapeLength(int l);
        void setShapeRadius(int r);
        int getShapeHeight() { return shape_height; }
        int getShapeWidth()  { return shape_width;  }
        int getShapeLength() { return shape_length; }
        int getShapeRadius() { return shape_radius; }
};
```

PUZZLEPIECE.CPP

```
/* Definitions for functions belonging to the PuzzlePiece class
*/

#include "stdafx.h"

#include "PuzzlePiece.h"
```

```
using namespace System;

PuzzlePiece::PuzzlePiece(void)
{
	Initialize();

}

void PuzzlePiece::Initialize()
{
	setName("N/A");
	array<int>^ input = { 0,0,0 };
	List<int>^ HSVmintmp = gcnew List<int>((IEnumerable<int>^) input);
	List<int>^ HSVmaxtmp = gcnew List<int>((IEnumerable<int>^) input);
	setHSVmin(HSVmintmp);
	setHSVmax(HSVmaxtmp);
	shape_point_x = 0;
	shape_point_y = 0;
	shape_width = 0;
	shape_height = 0;
	shape_length = 0;
	shape_radius = 0;
	isPlaced = false;
}

PuzzlePiece::~PuzzlePiece(void)
{
}

PuzzlePiece::PuzzlePiece(System::String^ piece_name)
{
	Initialize();
	setName(piece_name);
}

PuzzlePiece::PuzzlePiece(System::String^ piece_name, List<int>^ HSVmin, List<int>^ HSVmax)
{
	Initialize();
	setName(piece_name);
	setHSVmin(HSVmin);
	setHSVmax(HSVmax);
}

PuzzlePiece::PuzzlePiece(System::String^ piece_name, List<int>^ HSVmin, List<int>^ HSVmax, int xdest, int ydest)
{
	Initialize();
	setName(piece_name);
	setHSVmin(HSVmin);
	setHSVmax(HSVmax);
	setXDest(xdest);
	setYDest(ydest);
}

void PuzzlePiece::setTimePlacedToNow()
{
	this->isPlaced = true;
   DateTime saveNow = DateTime::Now;
	this->timePlaced = saveNow;
}
```

```
	//Set Shape Drawing Data
void PuzzlePiece::setShapeHeight(int h) {
		if (this->getName()->Equals("Rectangle")) { this->shape_height = h; }
		else { System::Console::WriteLine("Error: PuzzlePiece.h - Tried to set a height for a non-rectangle"); }
	}
void PuzzlePiece::setShapeWidth(int w) {
		if (this->getName()->Equals("Rectangle") || this->getName()->Equals("Square")) { this->shape_width = w; }
		else { System::Console::WriteLine("Error: PuzzlePiece.h - Tried to set a width for something other than a rectangle or square"); }
	}
void PuzzlePiece::setShapeLength(int l) {
		if (this->getName()->Equals("Triangle") || this->getName()->Equals("Pentagon")) { this->shape_length = l; }
		else { System::Console::WriteLine("Error: PuzzlePiece.h - Tried to set a length for something other than a triangle or pentagon");
}
	}
void PuzzlePiece::setShapeRadius(int r) {
		if (this->getName()->Equals("Circle")) { this->shape_radius = r; }
		else { System::Console::WriteLine("Error: PuzzlePiece.h - Tried to set a radius for a non-circle"); }
	}
```

TRACKEDPIECE.H

```
#pragma once
#include <Windows.h>
#include <string>
#include <opencv2\opencv.hpp>
#include <string>
#include <stack>
#include <process.h>
#using <System.dll>
#include <stdio.h>
#include <deque>
#include <vcclr.h>


using namespace std;
using namespace cv;

class TrackedPiece
{
public: bool timeLock;
		bool isTimeLocked() { return timeLock; }
		void setTimeLock() { timeLock = true; }
private:

	HANDLE myMutex;
	std::string name;
	int x_pos, y_pos;
	int x_dest, y_dest;
	int lastxPos, lastyPos;
	Scalar HSV_min, HSV_max;
	Scalar color;

	// drawing data
	int shape_point_x;
	int shape_point_y;
	int shape_width;
	int shape_height;
	int shape_length;
	int shape_radius;
```

```
        // Set to true when piece has consistently moved recently.
        // i.e. the user has picked up and is trying to place the piece.
        bool isMoving;

        bool isPlacedCorrectly;

        // Set to true when piece should be flashing on screen
        bool flashing;

        // Set true when piece should be turned off.
        bool turn_off;

        bool dimmed;



        // max number of values allowed in movementHistory deque
        static const unsigned int MAX_DEQUE_SIZE = 6;
        // number of tiemr ticks that piece has been moving to trigger a flashing hint
        static const unsigned int NUM_TRUES_TRIGGER_FLASH = 2;
        // max number of values allowed in placementHistory deque
        static const unsigned int MAX_PLACEMENT_DEQUE_SIZE = 8;
        // number of timer ticks that piece was detected in the right spot to trigger that it
        // has been placed correctly
        static const unsigned int NUM_TRUES_PLACED_CORRECTLY = 3;

        static const unsigned int PLACED_THRESH = 15;

        // Holds movement history of pieces each time the timer checks for movement.
        // Holds a max of MAX_DEQUE_SIZE elements
        deque<bool> movementHistory;

        // Holds a history of if the piece is sitting in its correct x,y location on the puzzle board
        deque<bool> placementHistory;

public:
        // position of shape on screen
        int xScreenPos;
        int yScreenPos;

        // True when piece is drawn on screen, false when it is not (for flashing shape)
        bool on;

        TrackedPiece(void);

        TrackedPiece(std::string);
        TrackedPiece(std::string, Scalar, Scalar);
        TrackedPiece(std::string, Scalar, Scalar, int, int); // (name, HSVmin, HSVmac, Xdest, Ydest)

        ~TrackedPiece(void);

        // Used for setting the isMoving boolean.
        // Returns true if if the piece has moved (according to movement threshold value) several
        // times in the last several timer ticks. This should tell that the user has picked up a piece
        // and is trying to place it.
        // returns false if there has not been consistent recent movement.
        int checkForMovement(bool justMoved);

        // Checks if piece has consistently been sitting in correct x,y position. If so, sets isPlacedCorrectly boolean to true.
        bool checkIfPlacedCorrectly();
```

```
bool getIsPlacedCorrectly() {return isPlacedCorrectly;}

void toggle(Mat &image);
void dim(Mat &image);
void turnOff(Mat &image);
void turnOn(Mat &image);

bool isFlashing() {return flashing;}
bool isTurnedOff() {return turn_off;}
bool isTurnedOn() {return !turn_off;}
bool isDimmed() {return dimmed;}

bool isOn() {return on;}

void clearStatus();
void setTurnOff(bool value) {turn_off = value;}
void setDimmed(bool value) {dimmed = value;}

int getXPos() {return x_pos;}
void setXPos(int x) {x_pos = x;}

int getXDest() {return x_dest;}
void setXDest(int x) {x_dest = x;}

int getYPos() {return y_pos;}
void setYPos(int y) {y_pos = y;}

int getYDest() {return y_dest;}
void setYDest(int y) {y_dest = y;}

int setPos(int x, int y) { x_pos = x; y_pos = y; }
int setDest(int x, int y) { x_dest = x; y_dest = y; }

int getLastxPos() {return lastxPos;}
void setLastxPos(int pos) {lastxPos = pos;}

int getLastyPos() {return lastyPos;}
void setLastyPos(int pos) {lastyPos = pos;}

Scalar getHSVmin() {return HSV_min;}
void setHSVmin(Scalar min) {HSV_min = min;}

Scalar getHSVmax() {return HSV_max;}
void setHSVmax(Scalar max) {HSV_max = max;}

std::string getName(){return name;}
void setName(std::string t){name = t;}

Scalar getColor() {return color;}
void setColor(Scalar c) {color = c;}

// shape drawing variables
void setShapePoint(int x, int y) { this->shape_point_x = x; this->shape_point_y = y; }
int getShapePointX()   { return shape_point_x; }
int getShapePointY()   { return shape_point_y; }

void setShapeHeight(int h);
void setShapeWidth(int w);
void setShapeLength(int l);
void setShapeRadius(int r);
int getShapeHeight() { return shape_height; }
int getShapeWidth()  { return shape_width;  }
```

```
        int getShapeLength() { return shape_length; }
        int getShapeRadius() { return shape_radius; }

};

//void on_trackbar( int, void* );
//void createTrackbarWindow();
//void erodeAndDilate(Mat &image);
//string intToStdString(int number);
//void drawObject(vector<TrackedPiece> pieces, Mat &frame);
//void trackFilteredObject(TrackedPiece &piece, Mat &cameraFeed, Mat &threshold_image);
//void trackTrackedPiece(TrackedPiece &piece, Mat &camera_feed, Mat &HSV_image, Mat &threshold_image);
//int startTrack();
```

TRACKEDPIECE.CPP

```
#include "stdafx.h"
#include <Windows.h>
#include "TrackedPiece.h"
#include <string>
#include <stack>
#include <process.h>
#using <System.dll>
#include <stdio.h>
#include <algorithm>
#include "Shape.h"
#include <cmath>

using namespace std;

TrackedPiece::TrackedPiece(void)
{
        setName("N/A");
        setHSVmin(Scalar(0,0,0));
        setHSVmax(Scalar(0,0,0));
        timeLock = false;
        isMoving = false;
        isPlacedCorrectly = false;
        flashing = false;
        turn_off = false;
        dimmed = false;
}


TrackedPiece::~TrackedPiece(void)
{
}


TrackedPiece::TrackedPiece(std::string piece_name)
{
        setName(piece_name);
        timeLock = false;
        isMoving = false;
        isPlacedCorrectly = false;
        flashing = false;
        turn_off = false;
        dimmed = false;
}
```

```
TrackedPiece::TrackedPiece(std::string piece_name, Scalar HSVmin, Scalar HSVmax)
{
	setName(piece_name);
	setHSVmin(HSVmin);
	setHSVmax(HSVmax);
	timeLock = false;
	isMoving = false;
	isPlacedCorrectly = false;
	flashing = false;
	turn_off = false;
	dimmed = false;
}

TrackedPiece::TrackedPiece(std::string piece_name, Scalar HSVmin, Scalar HSVmax, int xdest, int ydest)
{
	setName(piece_name);
	setHSVmin(HSVmin);
	setHSVmax(HSVmax);
	setXDest(xdest);
	setYDest(ydest);
	timeLock = false;
	isMoving = false;
	isPlacedCorrectly = false;
	flashing = false;
	turn_off = false;
	dimmed = false;
}

int TrackedPiece::checkForMovement(bool justMoved)
{
	//Return value tells what all the other pieces should be set to do
	//
	//  0 - Do nothing
	//  1 - Turn On
	//  2 - Dim
	//  3 - Turn off


	// add to movement history
	movementHistory.push_back(justMoved);
	// check if max size reached
	if(movementHistory.size() > MAX_DEQUE_SIZE)
		movementHistory.pop_front();

	// check for consistent movement
	int numTrues = count(movementHistory.begin(), movementHistory.end(), true);
	//cout << "Num trues " << name << ": " << numTrues << endl;

	if (numTrues >= 3 )
	{
		cout << name << " piece being placed." << endl;
		flashing = true;	// starts flashing
		return 1;
	}
	else
	{
		flashing = false;	// stops flashing
		return 0;
	}

	/*  This was for having hints occur incrementally
	if (numTrues >= 4 )
```

```
		{
			//clearStatus();
			std::cout << name << " **TURN OFF**" << endl;
			flashing = true;
			return 3;
		}

		else if (numTrues >= 3)

		{
			//clearStatus();
			cout << name << " **DIM**" << endl;
			flashing = true;
			return 2;
		}
		else if (numTrues >= NUM_TRUES_TRIGGER_FLASH)
		{
			//clearStatus();
			cout << name << " piece being placed." << endl;
			flashing = true;	// starts flashing
			return 1;
		}
		else
		{
			flashing = false;	// stops flashing
			return 0;
		}
		*/

}

bool TrackedPiece::checkIfPlacedCorrectly()
{
		if (abs(x_dest-x_pos) < PLACED_THRESH && abs(y_dest-y_pos) < PLACED_THRESH)
			placementHistory.push_back(true);
		else
			placementHistory.push_back(false);

		// check if max size reached
		if(placementHistory.size() > MAX_PLACEMENT_DEQUE_SIZE)
			placementHistory.pop_front();

		// check for consistent placement in correct position
		int numTrues = count(placementHistory.begin(), placementHistory.end(), true);
		if (numTrues >= NUM_TRUES_PLACED_CORRECTLY) {
			isPlacedCorrectly = true;
			cout << name << " placed correctly!" << endl;
			return true;
		} else {
			isPlacedCorrectly = false;
			return false;
		}
		// else placed correctly = false?
		// Once it has been  placed, do we want it to be able to be unplaced?
}

void TrackedPiece::toggle(Mat &image)
{
		Shape shapes(&image);
		if(isOn())
		{
			// turn off
```

```
                shapes.Draw_Shape(*this, 0);
                on = false;
        }
        else
        {
                // turn on
                shapes.Draw_Shape(*this, 1);
                on = true;
        }
        imshow("Puzzle Board Window", image);
}

void TrackedPiece::turnOff(Mat &image)
{
        Shape shapes(&image);
        shapes.Draw_Shape(*this, 0);
        on = false;
        imshow("Puzzle Board Window", image);
}

void TrackedPiece::turnOn(Mat &image)
{
        Shape shapes(&image);
        if(!isOn())
        {
                shapes.Draw_Shape(*this, 1);
                on = true;
        }
        imshow("Puzzle Board Window", image);
}

void TrackedPiece::dim(Mat &image)
{
        Shape shapes(&image);
        shapes.Draw_Shape(*this, 0.35);
        on = false;
        imshow("Puzzle Board Window", image);
}

void TrackedPiece::clearStatus()
{
        turn_off = false;
        dimmed = false;
        flashing = false;
}

        //Set Shape Drawing Data
void TrackedPiece::setShapeHeight(int h) {
                if (this->getName() == "Rectangle") { this->shape_height = h; }
                else { System::Console::WriteLine("Error: TrackedPiece.cpp - Tried to set a height for a non-rectangle"); }
        }
void TrackedPiece::setShapeWidth(int w) {
                if (this->getName() == "Rectangle" || this->getName() == "Square") { this->shape_width = w; }
                else { System::Console::WriteLine("Error: TrackedPiece.cpp - Tried to set a width for something other than a rectangle or
square"); }
        }
void TrackedPiece::setShapeLength(int l) {
                if (this->getName() == "Triangle" || this->getName() == "Pentagon") { this->shape_length = l; }
                else { System::Console::WriteLine("Error: TrackedPiece.cpp - Tried to set a length for something other than a triangle or
pentagon"); }
        }
void TrackedPiece::setShapeRadius(int r) {
```

```
		if (this->getName() == "Circle") { this->shape_radius = r; }
		else { System::Console::WriteLine("Error: TrackedPiece.cpp - Tried to set a radius for a non-circle"); }
	}
```

SCOREKEEPING.H

```
/*
These classes record and compile the performance data for each session (i.e. users, games, times pieces were placed)
It also controls file IO for saving performance data.
*/

#pragma once
#include "stdafx.h"
#include "GameBoard.h"

#ifndef FILE_G
#define FILE_G

using namespace System;
using namespace System::Collections::Generic;


// GamePlayedData and GamePlayed are circularly dependent, so declare them both beforehand
ref class GamePlayedData;
ref class GamePlayed;

// Keeps stats of a single game. Is compiled into a GamePlayedData instance, and then is no longer used.
ref class GamePlayed
{
public:

	GamePlayed();
	~GamePlayed();
	GamePlayed(KnobPuzzle^ Puzzle);

	void setGame(KnobPuzzle^ Puzzle);
	void setPlayer(System::String^ name) { this->player = name; }
	bool NOT_COMPLETED;

	System::String^ getType() { return this->gameType; }
	System::String^ getName() { return this->gameName; }
	System::String^ getPlayer() { return this->player; }
	GamePlayedData^ getGameData() { return this->gameData; }
	KnobPuzzle^ getGame() { return this->game; }
	int getLevelOfDifficulty() { return this->levelOfDifficulty; }

	// tell GamePlayed to pull the current date/time to record as start or end time
	void setStartTimeToNow(); // tell GamePlayed to pull the current date/time to record as start time
	void setTimeCompletedToNow();

	void gameEndedEarly();

	int getTimeForCompletion() { return this->timeForCompletion; }

	// functions for compiling data
	int compileData(); // pull information from puzzle pieces to fill arrays. Can only do this once
	double getAverageTimeBetweenPieces() { return this->avgTimeBetweenPieces; }
	DateTime getTimeStarted() { return this->timeStarted; }
	List<int>^ getTimesOfPlacement() { return this->timesOfPlacement; }
	List<int>^ getTimesBetweenPlacements() { return this->timeBetweenPlacements; }
	List<System::String^>^ getOrderOfPiecesPlayed() { return this->orderOfPiecesPlayed; }
```

```
private:

        KnobPuzzle^ game;                       // class holding all the puzzle data. ONLY HAS KNOBPUZZLE RIGHT NOW
        System::String^ gameType;  // type of game, e.g. knobpuzzle, blockpuzzle, snake...
        System::String^ gameName;  // name of the game. e.g. KNOBPUZZLE1
        System::String^ player;    // this is the name of the player
        int levelOfDifficulty;

        GamePlayedData^ gameData;
        bool ALREADY_COMPILED;

        DateTime timeStarted; // datetime object with time that the puzzle was started
        DateTime timeCompleted; // datetime object with time that the puzzle was ended

        int timeForCompletion; // time between start and end of game in seconds
        double avgTimeBetweenPieces;

        // please note that these three arrays are like a separated dictionary -
        // each entry in OrderOfPiecesPlayed should correspond to the matching index in the other two arrays.
        // BE VERY CAREFUL WITH THIS
        List<int>^ timesOfPlacement; // actual times of placement (minus start time) in seconds
        List<int>^ timeBetweenPlacements; // time it took to place each piece in seconds
        List<System::String^>^ orderOfPiecesPlayed; // names of pieces, from first placed to last placed

        void Initialize();
};

//-------------------------------------------------------------------------------------------------------//-------------------------------------------------------------------------------------------------------
---

// a simplified class that holds the data for a given game. This controls how file output appears, and scoring data is displayed.
ref class GamePlayedData {

public:
        GamePlayedData();
        ~GamePlayedData() {}
        GamePlayedData(GamePlayed^ inputGame);
        void Initialize();

        bool isSet; // can only be set once; once isSet is true (data compiled), shouldn't be able to change anything
        bool NOT_COMPLETED; // set if the game was ended prematurely

        System::String^ playerName;
        System::String^ gameName;

        int levelOfDifficulty;
        int averageTimeBetweenPieces;
        int timeForCompletion;

        List<int>^ timesOfPlacement; // actual times of placement (minus start time) in seconds
        List<int>^ timeBetweenPlacements; // time it took to place each piece in seconds
        List<System::String^>^ orderOfPiecesPlayed; // from first placed to last placed

        // the following refer to the time the game was started
        System::String^ month;
        System::String^ day;
        System::String^ year;
        System::String^ seconds;
        System::String^ minutes;
        System::String^ hours;
```

```
		System::String^ writeOut(); // returns a string of the data
		System::String^ buildFileName();

		int Save(); // writes the data to a file

};


//-------------------------------------------------------------------------------------------------//-------------------------------------------------------------------------------------------------

// Keeps running stats of the whole session ( >= 1 game). Stores data via GamePlayedData^ objects
public ref class ScoreKeeping
{
public:
		HANDLE myMutex;
		List<GamePlayedData^>^ individualGamesList;

		ScoreKeeping();
		void AddNewGame(GamePlayedData^ newGame) { this->individualGamesList->Add(newGame); }
		System::String^ showFinalResults();

private:
		//GamePlayed^ calculateAverageForGame(System::String^ gameName);
};

#endif
```

SCOREKEEPING.CPP

```
/*
		Definitions for functions belonging to the GamePlayed, GamePlayedData, and ScoreKeeping classes
*/

#include "stdafx.h"
//#include <Windows.h>
//#include <string>
//#include <stack>
//#include <process.h>
//#using <System.dll>
//#include <stdio.h>
#include "Functions.h"
#include "GameBoard.h"

using namespace System;
using namespace System::Collections::Generic;

void GamePlayed::Initialize()
{
		this->gameType = "Unknown";
		this->gameName = "Unknown";
		this->player = "Unknown";
		this->timesOfPlacement = gcnew List<int>();
		this->orderOfPiecesPlayed = gcnew List<System::String^>();
		this->avgTimeBetweenPieces = 0;
		this->timeForCompletion = 0;
		this->game = gcnew KnobPuzzle();
		this->ALREADY_COMPILED = false;
		this->gameData = gcnew GamePlayedData();
```

```
	this->NOT_COMPLETED = false;
}

GamePlayed::GamePlayed()
{
	Initialize();
}
//---------------------------------------------------------------------------------------------------

GamePlayed::~GamePlayed()
{
}
//---------------------------------------------------------------------------------------------------

GamePlayed::GamePlayed(KnobPuzzle^ Puzzle)
{
	Initialize();
	setGame(Puzzle);
}
//---------------------------------------------------------------------------------------------------

void GamePlayed::setGame(KnobPuzzle^ Puzzle)
{
	// the Input puzzle is just a reference, so it can be changed at any time. This means we should NEVER use
	// this->game's knobpuzzle outside of GamePlayed. All data variables should be set once when the game is completed and not touched
again
	// if game has already been compiled, return. The KnobPuzzle reference might have changed since then - dangerous.
	if (ALREADY_COMPILED) {
		Console::WriteLine("GamePlayed.cpp::compileData():: Error - GamePlayed instance has already been compiled - can't change");
		return;
	}
	this->game = Puzzle;
	this->gameType = Puzzle->getType();
	this->gameName = Puzzle->getName();
}

//---------------------------------------------------------------------------------------------------
void GamePlayed::setStartTimeToNow()
{
   DateTime saveNow = DateTime::Now;
	this->timeStarted = saveNow;
}

//---------------------------------------------------------------------------------------------------
void GamePlayed::setTimeCompletedToNow()
{
   DateTime saveNow = DateTime::Now;
	this->timeCompleted = saveNow;
}

//---------------------------------------------------------------------------------------------------
// Call compileData() once directly after completing a game. It will compile the stats, and then prohibit any more changes to class data (read-only).
int GamePlayed::compileData()
{
	// if game has already been compiled, return. The KnobPuzzle reference might have changed since then - dangerous.
	if (ALREADY_COMPILED) {
		Console::WriteLine("GamePlayed.cpp::compileData():: Error - tried to recompile GamePlayed Data");
		return -1;
	}
	// make sure the knobpuzzle was initialized
	if (!this->game->checkIsInitialized()) {
		Console::WriteLine("GamePlayed.cpp::compileData():: Error - KnobPuzzle was never initialized in this GamePlayed instance.");
```

```
		return -1;
	}
	// check if the start time was successfully recorded
	if (this->timeStarted.Equals(DateTime::MinValue)) {
		Console::WriteLine("GamePlayed.cpp::compileData():: Error - Start time was never recorded for GamePlayed^ instance.");
		return -1;
	}

	// pull level of difficulty
	this->levelOfDifficulty = this->game->getLevelOfDifficulty();

	// calculate seconds it took to finish the game
	this->timeForCompletion = secondsBetweenTwoDateTimes(this->timeStarted, this->timeCompleted);

	// find times placed information from puzzle pieces
	List<int>^ timesPlaced = gcnew List<int>();
	Dictionary< System::String^, int >^ pieceDict= gcnew Dictionary< System::String^, int >();
	int tim = 0;
	Console::WriteLine("Compiling Data: showing the times that each piece was placed");
	for each (PuzzlePiece^ piece in this->game->getPieceList()) {
		tim = secondsBetweenTwoDateTimes(this->timeStarted, piece->getTimePlaced()); // calc seconds between beginning of game and when piece was placed
		Console::WriteLine(piece->getName() + "   : " + tim);
		timesPlaced->Add(tim);
		pieceDict->Add(piece->getName(),tim); // add piece and time to dictionary (dictionary is for convenience for next step)
	}

	// make two lists to hold sorted time values, and their corresponding piece names
	List<System::String^>^ sortedKeys = gcnew List<System::String^>();
	List<int>^ sortedVals = gcnew List<int>();

	// sort times from smallest to largest
	for (int i = 0; i < timesPlaced->Count; i++) {
		System::String^ minStr = "";
		int minVal = 9999999;

		// find smallest time entry in dictionary
		for each (System::String^ key in pieceDict->Keys) {
			if (pieceDict[key] < minVal) {
				minStr = key;
				minVal = pieceDict[key];
			}
		}
		pieceDict->Remove(minStr); // remove piece from dictionary so that it isn't rerecorded
		sortedKeys->Add(minStr);
		sortedVals->Add(minVal);
	}
	this->orderOfPiecesPlayed = sortedKeys;
	this->timesOfPlacement = sortedVals;

	// now use sorted values to make a 3rd list - the times between each piece and the one before it
	List<int>^ timesBetweenPieces = gcnew List<int>();
	int temp = 0;
	for each (int tim in sortedVals) {
		timesBetweenPieces->Add(tim - temp);
		temp = tim;
	}
	this->timeBetweenPlacements = timesBetweenPieces;

	// calculate average time taken between pieces
	this->avgTimeBetweenPieces = averageListOfInts(timesBetweenPieces);
```

```
		Console::WriteLine("Testing the sorted data:");
		for (int i = 0; i < sortedKeys->Count; i++) {
			Console::WriteLine(sortedKeys[i] + "   " + sortedVals[i] + "   time between placement: " + timeBetweenPlacements[i]);
		}
		// set 'already compiled' to true, so that data can never be compiled again. From hereonout, the gameData instance will be returned.
		ALREADY_COMPILED = true;
		this->game = gcnew KnobPuzzle(); // destroy reference to the input KnobPuzzle just in case
		this->game->setError();
		this->gameData = gcnew GamePlayedData(this);
		return 0;
}
//-------------------------------------------------------------------------------------------------
// if the game was ended early, void out unplaced pieces instead of compiling the data
void GamePlayed::gameEndedEarly() {

		this->NOT_COMPLETED = true;

		List<int>^ timesPlaced = gcnew List<int>();
		Dictionary< System::String^, int >^ pieceDict= gcnew Dictionary< System::String^, int >();
		int tim = 0;
		Console::WriteLine("GamePlayed::gameEndedEarly(): negating time placed information for pieces not placed");
		for each (PuzzlePiece^ piece in this->game->getPieceList()) {
			if (!piece->isPlaced) {
				timesPlaced->Add(-1);
				Console::WriteLine(piece->getName() + "   : NOT PLACED");
				continue;
			}
			// calc seconds between beginning of game and when piece was placed
			tim = secondsBetweenTwoDateTimes(this->timeStarted, piece->getTimePlaced());
			Console::WriteLine(piece->getName() + "   : " + tim);
			timesPlaced->Add(tim);
			pieceDict->Add(piece->getName(),tim); // add piece and time to dictionary
		}
}


//-------------------------------------//-------------------------------------//-------------------------------------//-------------------------------------//-------------------------
-------------


//-------------------------------------//-------------------------------------//-------------------------------------//-------------------------------------//-------------------------
-------------

// GAMEPLAYED DATA
//-------------------------------------------------------------------------------------------------

GamePlayedData::GamePlayedData() {
		Initialize();
}
//-------------------------------------------------------------------------------------------------
GamePlayedData::GamePlayedData(GamePlayed^ inputGame) {

		// clear everything out real quick
		Initialize();
		this->NOT_COMPLETED = inputGame->NOT_COMPLETED;

		// copy all data over from GamePlayed^ instance
		this->playerName = inputGame->getPlayer();
		this->gameName = inputGame->getName();
		this->levelOfDifficulty = inputGame->getLevelOfDifficulty();

		DateTime timeStarted = inputGame->getTimeStarted();
```

```
		this->month = timeStarted.ToString("MMMM");
		this->day = timeStarted.Day.ToString();
		this->year = timeStarted.Year.ToString();
		this->seconds = timeStarted.Second.ToString();
		this->minutes = timeStarted.Minute.ToString();
		this->hours = timeStarted.Hour.ToString();

		// if the game was not completed, return here, with just the basic start information
		if (this->NOT_COMPLETED) {
			this->isSet = true;
			return;
		}

		this->averageTimeBetweenPieces = inputGame->getAverageTimeBetweenPieces();
		this->timeForCompletion = inputGame->getTimeForCompletion();

		this->timesOfPlacement = inputGame->getTimesOfPlacement();
		this->timeBetweenPlacements = inputGame->getTimesBetweenPlacements();
		this->orderOfPiecesPlayed = inputGame->getOrderOfPiecesPlayed();

		// all set, so be done!
		this->isSet = true;

}
//-----------------------------------------------------------------------------------------------
// set up all the containers
void GamePlayedData::Initialize() {

		this->isSet = false;
		this->NOT_COMPLETED = false;
		playerName = "";	gameName = "";	levelOfDifficulty = 0;
		averageTimeBetweenPieces = 0;		timeForCompletion = 0;

		List<int>^ timesOfPlacement = gcnew List<int>();
		List<int>^ timeBetweenPlacements = gcnew List<int>();
		List<System::String^>^ orderOfPiecesPlayed = gcnew List<System::String^>();

		// the following refer to the time the game was started
		month = "";		day = ""; year = ""; seconds = "";		minutes = ""; hours = "";
}
//-----------------------------------------------------------------------------------------------
System::String^ GamePlayedData::writeOut() {

		if (!this->isSet) {
			Console::WriteLine("GamePlayedData::writeOut(): data hasn't been set yet");
			return "Error";
		}
		if (this->NOT_COMPLETED) {
			Console::WriteLine("GamePlayedData::WriteOut(): Error : game was not completed");
			System::String^ resultString = "";
			resultString = "Player : " + this->playerName + Environment::NewLine + "Game : " + this->gameName +
Environment::NewLine;
			System::String^ tim = day + " " + month  + " " + year + " "  + hours + ":" + minutes;
			resultString = resultString + "Time Started : " + tim + Environment::NewLine;
			resultString = resultString + "GAME NOT COMPLETED";
			return resultString;
		}

		System::String^ resultString = "";
		resultString = "Player : " + this->playerName + Environment::NewLine + "Game : " + this->gameName + Environment::NewLine;
		resultString = resultString + "Level of Difficulty : " + this->levelOfDifficulty + Environment::NewLine;
		System::String^ tim = day + " " + month  + " " + year + " "  + hours + ":" + minutes;
```

```
		resultString = resultString + "Time Started : " + tim + Environment::NewLine;
		resultString = resultString + "Time for Completion (s): " + this->timeForCompletion+ Environment::NewLine;
		resultString = resultString + "Average Time Between Pieces: " + this->averageTimeBetweenPieces + Environment::NewLine +
Environment::NewLine;
		for (int i = 0; i < this->orderOfPiecesPlayed->Count; i++)
		{
			System::String^ val1 = "Piece : " + this->orderOfPiecesPlayed[i] + Environment::NewLine;
			System::String^ val2 = "	Time of Placement (s) :    " + this->timesOfPlacement[i] + Environment::NewLine;
			System::String^ val3 = "	Time it Took to Place (s) : " + this->timeBetweenPlacements[i] + Environment::NewLine;
			resultString = resultString + System::String::Format("{0}{1}{2}", val1, val2, val3);
		}

		return resultString;
}
//-----------------------------------------------------------------------------------------------
// output file for saved performance data will be labeled: mainpath/PlayerName/PlayerName_GameName_YYYY_MMMM_dd.txt
System::String^ GamePlayedData::buildFileName() {
		if (!isSet) {
			return "Error";
		}
		System::String^ mainString = buildOutputFileName(playerName, gameName, month, day, year);

		return mainString;

}
//-----------------------------------------------------------------------------------------------
int GamePlayedData::Save()
{
		// build file name to save to
		System::String^ outputFile = this->buildFileName();
		System::Windows::Forms::MessageBox::Show(outputFile);

		// pull results for this game
		System::String^ finalResults = this->writeOut();
		if (finalResults->Contains("Error")) { return -1; }
		// check if there is already results for that day. If so, append data
		if (System::IO::File::Exists(outputFile)) {
			Console::WriteLine("GamePlayedData::Save(): appending lines to file");
			// add a couple spaces before new results for padding
			finalResults = "\r\n\r\n" + finalResults;
			array<System::String^>^ tmpArray = gcnew array<System::String^>(1);
			tmpArray[0] = finalResults;
			// append new data to existing file
			int success = appendStringArrayToFile(tmpArray, outputFile);
		}
		// if file doesn't exist yet, create new file with a brief header and put the data there
		if (!System::IO::File::Exists(outputFile)) {
			// pull today's date and construct a header string for the file, with date and Player's name
			DateTime saveNow = DateTime::Now;
			DateTime today = saveNow.ToLocalTime();
			System::String^ introString = "Player: " + this->playerName + "\r\nSession: " + month + " " + day + ", " + year + " " + hours +
Environment::NewLine;
			// put both strings in an array to write out to file
			array<System::String^>^ tmpArray = gcnew array<System::String^>(2);
			tmpArray[0] = introString; tmpArray[1] = finalResults;
			int success = writeStringArrayToFile(tmpArray, outputFile);
			if (success != 0) {
				Console::WriteLine("GamePlayedData::Save(): Error - couldn't write output to file " + outputFile);
				return -1;
			}
		}
		return 0;
```

```
}

//-------------------------------------//-------------------------------------//-------------------------------------//-------------------------------------//-------------------------------------

//-------------------------------------//-------------------------------------//-------------------------------------//-------------------------------------//-------------------------------------

// SCOREKEEPING



// Initialize a blank scorekeeper
//-----------------------------------------------------------------------------------------------------
ScoreKeeping::ScoreKeeping() {
	this->individualGamesList = gcnew List<GamePlayedData^>();
}

//-----------------------------------------------------------------------------------------------------
// return a string for printing out all results, to be displayed from the GUI.
System::String^ ScoreKeeping::showFinalResults()
{
	System::String^ finalString = "Performance and Progress\n\n";

	List<System::String^>^ individualGameStrings = gcnew List<System::String^>();  // to hold each game's result string
	System::String^ individualResult = "";

	// print the results for each game and then tack them all together
	for each (GamePlayedData^ game in this->individualGamesList)
	{
		individualResult = game->writeOut();
		if (!individualResult->Contains("Error")) {individualGameStrings->Add(individualResult); }
	}
	finalString = finalString + System::String::Join("\n", individualGameStrings);

	return finalString;
}
```

RUNTRACKING.H

```
/* This class controls the operation of OpenCV.  It starts OpenCV running, monitors/controls tracking, gathers time data, and shuts
OpenCV down once the game is completed or stopped.  An instance of this class is created in "Functions.cpp - initializeTracking()" when the user
hits
the Run button on the GUI
*/
#pragma once

#include "stdafx.h"
```

```
#include <vcclr.h>
#include <opencv2\opencv.hpp>		//includes all OpenCV headers
#include "Shape.h"

#define _CRTDBG_MAP_ALLOC

class RunTracking
{

 public:

			RunTracking() { Initialize(); }
			~RunTracking() {delete sound_player;}

	virtual void Initialize();
	virtual int Start();
	virtual void Stop() { STOP = true; }
	virtual void setGame(KnobPuzzle^ game) {this->Game = game; this->gameRecord->setGame(game);}
			gcroot<KnobPuzzle^> getGame() {return this->Game;}
			bool checkIfAllPiecesCorrect();
			gcroot<GamePlayed^> returnScore() { return this->gameRecord;}

			virtual void setPlayer(System::String^ name) { this->gameRecord->setPlayer(name); }


protected:

			// all the game and piece information is passed in via a KnobPuzzle class instance.
			// gcroot appears to magically transfer my classes from managed->unmanaged without consequences. Don't ask how.
	gcroot<KnobPuzzle^> Game;

	// when openCV is terminated (gameover), this instance of GamePlayed will be added to the overall scorekeeping class for the gui
	gcroot<GamePlayed^> gameRecord;

	bool STOP;
			Shape shapes;

	virtual int startTrack();
	virtual void endTrack();
	virtual void trackTrackedPiece(TrackedPiece &piece, Mat &camera_feed, Mat &HSV_image, Mat &threshold_image);
	virtual void trackFilteredObject(TrackedPiece &piece, Mat &cameraFeed, Mat &threshold_image);
	void createTrackbarWindow();
			void createTrackbarWindow(TrackedPiece &tmp);
	void erodeAndDilate(Mat &image);
	void drawObject(vector<TrackedPiece> pieces, Mat &frame);
	void drawPuzzleBoard(Mat &image);

			int loadTrackedPieces();

			//handling placement of pieces
			void processPlacementOfPiece(TrackedPiece trackedpiece);

			//// test cases
			//virtual void Test1();
			//virtual void Test2();
			//virtual void Test3();
			//virtual void Test4();
			//virtual void Test5();
			//virtual void Test6();

private:
```

```
                    // Making global for now, since can't get the timer callback function to work as a member function
                    //vector<TrackedPiece> pieces;

                    long StartTime;

    std::string original_window;
    std::string trackbar_window;
    std::string filtered_window;
    std::string puzzle_window;

                    // hard coded test case. 0 = not a test
                    //int TestCase;

    //default capture width and height
    int FRAME_WIDTH;
    int FRAME_HEIGHT;
    //max number of objects to be detected in frame
    int MAX_NUM_OBJECTS;
    //minimum and maximum object area
    int MIN_OBJECT_AREA;
    int MAX_OBJECT_AREA;

    int H_min;
    int H_max;
    int S_min;
    int S_max;
    int V_min;
    int V_max;

                    // Could not get to work as member variables
                    //static VOID CALLBACK RunTracking::static_timerTick(  _In_  HWND hwnd, _In_  UINT uMsg, _In_  UINT_PTR idEvent,
_In_  DWORD dwTime);
                    //VOID CALLBACK RunTracking::timerTick(  _In_  HWND hwnd, _In_  UINT uMsg, _In_  UINT_PTR idEvent, _In_
DWORD dwTime);

    RunTracking(const RunTracking&);  // Not implemented.
    void operator=(const RunTracking&);  // Not implemented.

                    // Used to play sounds.
                    SoundEffectPlayer* sound_player;

                    // sound filenames
                    string sound_game_start;
                    string sound_game_completed;

};
```

RUNTRACKING.CPP

```
/*  This file includes all the functions that are directly related to the maintenance of the RunTracking class,
e.g. initializing, starting, ending.  Tracking functions are located in "Tracking.cpp"
*/

#include "stdafx.h"

#include <opencv2\opencv.hpp>
#include "Functions.h"
#include "RunTracking.h"

void RunTracking::Initialize() {
```

```
                //this->TestCase = Constants::TESTNUMBER;

	this->original_window = "Original Capture";
	this->trackbar_window = "Trackbar Window";
	this->filtered_window = "Filtered Image";
	this->puzzle_window = "Puzzle Board Window";

	//default capture width and height
	this->FRAME_WIDTH = Constants::DEFAULT_FRAME_WIDTH;
	this->FRAME_HEIGHT = Constants::DEFAULT_FRAME_HEIGHT;

	//max number of objects to be detected in frame
	this->MAX_NUM_OBJECTS= Constants::DEFAULT_MAX_OBJECTS_IN_FRAME;

	//minimum and maximum object area
	this->MIN_OBJECT_AREA = Constants::DEFAULT_MIN_OBJECT_AREA;
	this->MAX_OBJECT_AREA = FRAME_HEIGHT*FRAME_WIDTH/1.5;
	this->H_min = Constants::DEFAULT_H_MIN;
	this->H_max = Constants::DEFAULT_H_MAX;
	this->S_min = Constants::DEFAULT_S_MIN;
	this->S_max = Constants::DEFAULT_S_MAX;
	this->V_min = Constants::DEFAULT_V_MIN;
	this->V_max = Constants::DEFAULT_V_MAX;

                // initialize gameRecording and set its start time to now. All scores will be measured against this start time
                this->gameRecord = gcnew GamePlayed();
                this->gameRecord->setStartTimeToNow();
                this->STOP = false;

                // sound file names
                sound_game_start = "guitar_start.mp3";
                sound_game_completed = "guitar_end.mp3";

                sound_player = new SoundEffectPlayer();

                //Can play game start sound here
                sound_player->play_Sound(sound_game_start);

}
//-------------------------------------------------------------------------------------------------------

// start running opencv
int RunTracking::Start() {

                ////IF test is selected, go to test
                //if (this->TestCase != 0) {
                //	switch(this->TestCase) {
                //			case 1:  Test1();	break;
                //			case 2:  Test2();	break;
                //			case 3:  Test3();	break;
                //			case 4:  Test4();	break;
                //			case 5:  Test5();	break;
                //			case 6:  Test6();	break;
                //		// add more tests here as necessary
                //			default:
                //				System::String^ errorStr = "Error: Cannot find test case " + this->TestCase;
                //				System::Windows::Forms::MessageBox::Show(errorStr);
                //	}
                //}
                //else {
                int success = startTrack();
                return success;
```

```
                        //}
}
//-------------------------------------------------------------------------------------------------

// when tracking is ended, destroy windows and compile data
void RunTracking::endTrack() {

                        System::Console::WriteLine("RunTracking::EndTrack() : Exiting RunTracking");
                        cv::destroyAllWindows();

                        // if game was exited early, tell gameRecord to handle what we have
                        if (this->Game->isEndGame()) {
                                this->gameRecord->gameEndedEarly();
                                return;
                        }

                        // if game was completed, compile record for game
      System::Windows::Forms::MessageBox::Show("Congratulations, YOU WON!");
                        //this->gameRecord->setTimeCompletedToNow();
                        this->gameRecord->compileData();

                        return;
}

//-------------------------------------------------------------------------------------------------
// called when the tracker decides a piece was placed
void RunTracking::processPlacementOfPiece(TrackedPiece trackedpiece)
{
            // find corresponding PuzzlePiece^ in KnobPuzzleInstance, and set it's time placed to now.
            for each (PuzzlePiece^ piece in this->Game->getPieceList()) {
                        if (piece->getName()->Equals(stdStringToSystemString(trackedpiece.getName()))) {
                                    piece->setTimePlacedToNow();
                                    Console::WriteLine("RunTracking.cpp: processPlacementOfPiece() : piece placed!! - " + piece->getName());
                        }
            }
}
```

TRACKING.CPP

```
// Multiple_Object_Tracking.cpp : Defines the entry point for the console application.
//

// This project has been modified from the original version written by Kyle Hounslow:

//-------------------------------------------------------------------------------------------------------------------
//Written by  Kyle Hounslow 2013

//Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software")
//, to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense,
//and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

//The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
//-------------------------------------------------------------------------------------------------------------------

/* This  file defines all of the tracking functions and algorithms used for tracking during normal gameplay (not calibration). Most
functions belong to the RunTracking class.
*/

#include "stdafx.h"
```

```
#include <Windows.h>          // for timer
#include <WinBase.h>          //for sleep()
#include <vcclr.h>
#include <iostream>
#include <string>
#include <vector>
#include <opencv2\opencv.hpp>          //includes all OpenCV headers
#include "Functions.h"
#include "Shape.h"
#include "RunTracking.h"
#include <msclr\marshal_cppstd.h> //to convert managed string to std::string


#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>

using namespace cv;
using namespace std;
using namespace msclr::interop;

//
// Global for now, should not be though
vector<TrackedPiece> pieces;

Mat puzzle_board;                                    //Puzzle board image for drawing shapes on
int difficulty;                                              //Difficulty pulled from the gui
                                                                                    //  there should be a better way to access this but
                                                                                    //  I'm not sure how to do that.

//-------------------------------------------------------------------------------------------------------------
void on_trackbar( int, void* )
{//This function gets called whenever a
            // trackbar position is changed
}

//-------------------------------------------------------------------------------------------------------------
// load up the tracked pieces vector from the PuzzlePieces imported with the KnobPuzzle
int RunTracking::loadTrackedPieces() {
            pieces.clear();
            // if there are no pieces to load, return an error
            if (this->Game->getPieceList()->Count < 1) {
                        Console::WriteLine("RunTracking::loadTrackedPieces(): no pieces to load!");
                        return -1;
            }
            // otherwise, convert each PuzzlePiece to a TrackedPiece and push it onto the vector
             for (int i = 0; i < this->Game->getPieceList()->Count; i++) {
                         pieces.push_back(puzzlePieceToTrackedPiece(this->Game->getPieceList()[i]));
             }
             return 0;
 }

//-------------------------------------------------------------------------------------------------------------
void RunTracking::createTrackbarWindow()
{
            namedWindow(trackbar_window);
            createTrackbar( "H_MIN", trackbar_window, &H_min, H_max, on_trackbar );
            createTrackbar( "H_MAX", trackbar_window, &H_max, H_max, on_trackbar );
            createTrackbar( "S_MIN", trackbar_window, &S_min, S_max, on_trackbar );
            createTrackbar( "S_MAX", trackbar_window, &S_max, S_max, on_trackbar );
            createTrackbar( "V_MIN", trackbar_window, &V_min, V_max, on_trackbar );
            createTrackbar( "V_MAX", trackbar_window, &V_max, V_max, on_trackbar );
```

```
}

//---------------------------------------------------------------------------------------------------
void RunTracking::erodeAndDilate(Mat &image)
{
	//create structuring element that will be used to "dilate" and "erode" image.
	//the element chosen here is a 3px by 3px rectangle

	Mat erodeElement = getStructuringElement( MORPH_RECT,Size(3,3));
	//dilate with larger element so make sure object is nicely visible
	Mat dilateElement = getStructuringElement( MORPH_RECT,Size(8,8));

	erode(image,image,erodeElement);
	erode(image,image,erodeElement);


	dilate(image,image,dilateElement);
	dilate(image,image,dilateElement);
}

//---------------------------------------------------------------------------------------------------
void RunTracking::drawObject(vector<TrackedPiece> pieces, Mat &frame){

	for(int i =0; i<pieces.size(); i++){

		cv::circle(frame,cv::Point(pieces.at(i).getXPos(),pieces.at(i).getYPos()),10,cv::Scalar(0,0,255));
		cv::putText(frame,intToStdString(pieces.at(i).getXPos())+ " , " +
intToStdString(pieces.at(i).getYPos()),cv::Point(pieces.at(i).getXPos(),pieces.at(i).getYPos()+20),1,1,Scalar(0,255,0));
		cv::putText(frame,pieces.at(i).getName(),cv::Point(pieces.at(i).getXPos(),pieces.at(i).getYPos()-30),1,2,pieces.at(i).getColor());
	}
}

//---------------------------------------------------------------------------------------------------
void RunTracking::trackFilteredObject(TrackedPiece &piece, Mat &cameraFeed, Mat &threshold_image){

	vector <TrackedPiece> pieces; // IS THIS THE SAME AS THE GLOBAL VARIABLE ABOVE???

	Mat temp;
	threshold_image.copyTo(temp);
	//these two vectors needed for output of findContours
	vector< vector<Point> > contours;
	vector<Vec4i> hierarchy;
	//find contours of filtered image using openCV findContours function
	findContours(temp,contours,hierarchy,CV_RETR_CCOMP,CV_CHAIN_APPROX_SIMPLE );
	//use moments method to find our filtered object
	bool objectFound = false;
	if (hierarchy.size() > 0) {
		int numObjects = hierarchy.size();
//		cout << "Num objects: " << numObjects << endl;
//		cout << "Max Num objects: " << MAX_NUM_OBJECTS << endl;
		// threshholed to calculate movement
		const int thresh = 40;
		//saves max area of each contour detected so only the largest one will be tracked
		double maxArea = 0;
		// temporary piece for contours found
		TrackedPiece tmp;

		//if number of objects greater than MAX_NUM_OBJECTS we have a noisy filter
		if(numObjects < MAX_NUM_OBJECTS){
			// for each object (contour) detected
			for (int index = 0; index >= 0; index = hierarchy[index][0]) {
```

```
                                        // get the moment of the contour
                                        Moments moment = moments((cv::Mat)contours[index]);
                                        // get the area from the moment
                                        double area = moment.m00;
//                                      cout << "Area " << index << " is: " << area << endl;

                                        // if the area is less than MIN_OBJECT_AREA then it is probably just noise
                                        // it must also be large than the max area found so far since we only want the largest area.
                                        if(area > MIN_OBJECT_AREA && area > maxArea){
                                                // set new max area
                                                maxArea = area;
                                                // Clear previous objects found so only one (the biggest) is detected
                                                pieces.clear();

                                                int xPos = moment.m10/area;
                                                int yPos = moment.m01/area;

                                                tmp.setXPos(xPos);
                                                tmp.setYPos(yPos);
                                                tmp.setName(piece.getName());
                                                tmp.setColor(piece.getColor());

                                                //cout << piece.getName() << ": x: " << xPos << " y: " << yPos << endl;
                                                //cout << "LastPos: x: " << piece.getLastxPos() << " y: " << piece.getLastyPos() << endl;

                                                pieces.push_back(tmp);

                                                objectFound = true;
                                        }
                                }

                                //let user know you found an object and check for movement
                                if(objectFound ==true){

                                        // Update piece location (tmp piece should now be biggest contour found)
                                        piece.setXPos(tmp.getXPos());
                                        piece.setYPos(tmp.getYPos());

                                        /*
                                         * Movement checking moved to timerTick
                                         *
                                        // Check for movement (tmp piece should now be biggest contour found)
                                        if(tmp.getXPos() > (piece.getLastxPos() + thresh) || tmp.getXPos() < (piece.getLastxPos() - thresh))
                                        {
                                                piece.setLastxPos(tmp.getXPos());
                                                cout << piece.getName() << ": X movement" << endl;
                                        }
                                        if(tmp.getYPos() > (piece.getLastyPos() + thresh) || tmp.getYPos() < (piece.getLastyPos() - thresh))
                                        {
                                                piece.setLastyPos(tmp.getYPos());
                                                cout << piece.getName() << ": Y movement." << endl;
                                        }
                                        */

                                        //draw object location on screen
                                        drawObject(pieces,cameraFeed);}

                        }else putText(cameraFeed,"TOO MUCH NOISE! ADJUST FILTER",Point(0,50),1,2,Scalar(0,0,255),2);
                }
        }

//-----------------------------------------------------------------------------------------------
```

```
void RunTracking::trackTrackedPiece(TrackedPiece &piece, Mat &camera_feed, Mat &HSV_image, Mat &threshold_image)
{
	//convert to binary image with white = in range specified
	inRange(HSV_image, piece.getHSVmin(), piece.getHSVmax(), threshold_image);
	erodeAndDilate(threshold_image);
	trackFilteredObject(piece, camera_feed, threshold_image);
}

//---------------------------------------------------------------------------------------------------
// draw the puzzleboard background by pulling shape information from the TrackedPiece vector
void RunTracking::drawPuzzleBoard(Mat &image)
{
	//Shape shapes(&image);
	shapes.setImage(&image);
	shapes.Clear_To_Black(); // Must clear to black first, otherwise get exception
	shapes.Clear_To_Gray();
	for (unsigned int i = 0; i < pieces.size(); i++)
	{
		shapes.Draw_Shape(pieces[i], 1);
	}
}

//---------------------------------------------------------------------------------------------------
// This should probably be a member function of some class, but I wasn't sure where it should go.
bool checkIfAllCorrect()
{
	for(int i = 0; i < pieces.size(); i++)
	{
		// If a piece is found that isn't placed correctly, return false
		if(!pieces[i].getIsPlacedCorrectly())
			return false;
	}
	return true;
}

//---------------------------------------------------------------------------------------------------
VOID CALLBACK timerTick(  _In_  HWND hwnd, _In_  UINT uMsg, _In_  UINT_PTR idEvent, _In_  DWORD dwTime)
{
	int thresh = 20;

	for(int i = 0; i < pieces.size(); ++i)
		pieces[i].clearStatus();

	for(int i = 0; i < pieces.size(); ++i)
	{
		int status = 0;
		if(pieces[i].getXPos() > (pieces[i].getLastxPos() + thresh) || pieces[i].getXPos() < (pieces[i].getLastxPos() - thresh))
		{
			pieces[i].setLastxPos(pieces[i].getXPos());
			status = pieces[i].checkForMovement(true);
			//cout << pieces[i].getName() << ": X movement" << endl;
		}

		else if(pieces[i].getYPos() > (pieces[i].getLastyPos() + thresh) || pieces[i].getYPos() < (pieces[i].getLastyPos() - thresh))
		{
			pieces[i].setLastyPos(pieces[i].getYPos());
			status = pieces[i].checkForMovement(true);
			//cout << pieces[i].getName() << ": Y movement." << endl;
		}

		else { // No movement
			status = pieces[i].checkForMovement(false);
```

```
		pieces[i].checkIfPlacedCorrectly();
		bool allCorrect = checkIfAllCorrect();
		if (allCorrect) {
			cout << "All pieces placed correctly!" << endl;
		}
}

if (status != 0)
{
		//EASY
		if(difficulty == 1)
		{
			//Turn off all other pieces
			for(int j = 0; j < pieces.size(); j++)
			if (i != j)
			{
				pieces[j].clearStatus();
				pieces[j].setTurnOff(true);
			}
		}

		//MEDIUM
		if(difficulty == 2)
		{
			//Dim all other pieces
			for(int j = 0; j < pieces.size(); j++)
			if (i != j)
			{
				pieces[j].clearStatus();
				pieces[j].setDimmed(true);
			}
		}

		//HARD
		if(difficulty == 3)
		{
			//I don't think anything has to happen here
		}
}



//Depending of the status returned above, this will change
//if all the other pieces are turned off, turned on, etc...
/*
if (status == 1)
{
		//Turn on all other pieces
		for(int j = 0; j < pieces.size(); j++)
			if (i != j)
			{
				pieces[j].clearStatus();
			}
}
else if (status == 2)
{
		//Dim all other pieces
		for(int j = 0; j < pieces.size(); j++)
			if (i != j)
			{
				pieces[j].clearStatus();
```

```
                                        pieces[j].setDimmed(true);
                                }
                        }
                        else if (status == 3)
                        {
                                //Turn off all other pieces
                                for(int j = 0; j < pieces.size(); j++)
                                        if (i != j)
                                        {
                                                pieces[j].clearStatus();
                                                pieces[j].setTurnOff(true);
                                        }
                        }
                        */
                }

                //cout << "Timer tick." << endl;
}

//-------------------------------------------------------------------------------------------------------
VOID CALLBACK timerFlash(  _In_  HWND hwnd, _In_  UINT uMsg, _In_  UINT_PTR idEvent, _In_  DWORD dwTime)
{
                for(int i = 0; i < pieces.size(); ++i)
                {
                        //First check if the piece should be flashing
                        if (pieces[i].isFlashing())
                        {
                                pieces[i].toggle(puzzle_board);
                        }
                        //Then check if it should be dimmed
                        else if(pieces[i].isDimmed())
                        {
                                pieces[i].dim(puzzle_board);
                        }
                        //Then check if it should be turned off
                        else if (pieces[i].isTurnedOff())
                        {
                                cout << "TURNING OFF " << pieces[i].getName() << endl;
                                pieces[i].turnOff(puzzle_board);
                        }
                        //If its not doing anything special, then make sure that it is turned on
                        else
                        {
                                pieces[i].turnOn(puzzle_board);
                        }
                }
}
/*
 * Can't get callback to work as a member function, so making pieces global for now
 *
void* ptr;

VOID CALLBACK RunTracking::static_timerTick(  _In_  HWND hwnd, _In_  UINT uMsg, _In_  UINT_PTR idEvent, _In_  DWORD dwTime)
{
                RunTracking* pThis = reinterpret_cast<RunTracking*>(ptr);
                pThis->timerTick(hwnd, uMsg, idEvent, dwTime);
}
*/

//-------------------------------------------------------------------------------------------------------
int RunTracking::startTrack()
{
```

```
	//Get diffculty
	difficulty = this->Game->getLevelOfDifficulty();

	// set timer to periodically check piece movement
	UINT timer_ms = Constants::TIMER_TICK;
	HWND hwnd1 = NULL;
	UINT_PTR myTimer = SetTimer(hwnd1, 1, timer_ms, timerTick);

	// set timer to flash shapes
	UINT timer_flash = Constants::FLASH_DELAY;
	HWND hwnd2 = NULL;
	UINT_PTR myTimer2 = SetTimer(hwnd2, 1, timer_flash, timerFlash);
	//SetTimer(NULL, 1, timer_flash, 2timerFlash);

	// TRYING TO IMPORT CALIBRATED PIECES HERE::
	int loadResult = this->loadTrackedPieces();

	// if import failed, return error
	if (loadResult != 0) {
		Console::WriteLine("RunTracking::startTrack():: loadTrackedPieces() failed. Exiting.");
		KillTimer(hwnd1, myTimer); // kill the timers
		KillTimer(hwnd2, myTimer2);
		return -1;
	}

	VideoCapture capture;

	capture.open(0);	//0 is default video device, 1 is other/USB camera
	//set height and width of capture frame
	capture.set(CV_CAP_PROP_FRAME_WIDTH,FRAME_WIDTH);
	capture.set(CV_CAP_PROP_FRAME_HEIGHT,FRAME_HEIGHT);

	if (!capture.isOpened())
	{
		cout << "RunTracking::startTrack():: Cannot open camera." << endl;
		KillTimer(hwnd1, myTimer); // kill the timers
		KillTimer(hwnd2, myTimer2);
		System::Windows::Forms::MessageBox::Show("Can't access the camera!");
		return -1;
	}
	cout << "RunTracking::startTrack():: Camera opened" << endl;

	Mat camera_feed;		//raw camera image
	Mat HSV_image;			//camera image converted to HSV
	Mat threshold_image;		//image after HSV is filtered and processed


	//if (calibrate_mode)
	//{
	//	createTrackbarWindow();
	//}

	// Moved to member variables of RunTracking class
//	TrackedPiece yellow = TrackedPiece("Tennis Ball", Scalar(25,44,160), Scalar(77,95,256));
//	TrackedPiece red_circle = TrackedPiece("Circle", Scalar(165, 107, 25), Scalar(185, 233, 256));
//	TrackedPiece green_rectangle = TrackedPiece("Rectangle", Scalar(74, 75, 50), Scalar(88, 214, 256));
//	TrackedPiece yellow_pentagon = TrackedPiece("Pentagon", Scalar(16, 47, 47), Scalar(32, 200, 256));
//	TrackedPiece white_square = TrackedPiece("Square", Scalar(77, 0, 168), Scalar(158, 63, 256));

	//Mat puzzle;					//Puzzle board image for drawing shapes on

	namedWindow(puzzle_window, WINDOW_NORMAL);
```

```
drawPuzzleBoard(puzzle_board);
imshow(puzzle_window, puzzle_board);


// Move puzzle board window to correct position on game board monitor
// (Puzzle board monitor needs to be set up to the left of first monitor.)
moveWindow(puzzle_window, -1600, 0);
cvSetWindowProperty(puzzle_window.c_str(), CV_WND_PROP_FULLSCREEN, CV_WINDOW_FULLSCREEN);  // Makes full screen
moveWindow(original_window, 640, 0);
moveWindow(filtered_window, 640, 512);

// go into infinite loop of reading camera input and tracking pieces
while(1)
{
	capture.read(camera_feed);

	//namedWindow(original_window);

	//namedWindow(filtered_window);
	try {
		cvtColor(camera_feed, HSV_image, CV_BGR2HSV);
	} catch (System::Exception ^e){
	   System::Console::WriteLine("Tracking.cpp::StartTrack():: Exception " + e->GetType()->ToString() + "  at cvtColor()");
		KillTimer(hwnd1, myTimer); // kill the timers
		KillTimer(hwnd2, myTimer2);
		return -1;
	}

		for (int i = 0; i < pieces.size(); ++i)
		{
			trackTrackedPiece(pieces[i], camera_feed, HSV_image, threshold_image);
		}
//		imshow(filtered_window, threshold_image);

	imshow(original_window, camera_feed);

	waitKey(30);


	// check if individual pieces are placed correctly
	//allCorrect = true;
	// check if all pieces are placed correctly
	bool allCorrect = checkIfAllCorrect();
	if (allCorrect) {
		System::Console::WriteLine("Tracking.cpp::startTrack() : All pieces placed correctly!");
	}

	// ADDED TEMP WORKAROUND - game will end if all pieces have been placed at least once., whether
	// or not they are still correcly placed.
	int numberCorrectlyPlaced = 0;
	for(int i = 0; i < pieces.size(); i++)
	{
		// if piece is already placed, continue
		if (pieces[i].isTimeLocked()) {
			numberCorrectlyPlaced++;
			continue;
		}
		if (!pieces[i].isTimeLocked()) { allCorrect = false; }
		// otherwise check. If this is the first time finding it's correct, then process placement
		//TODO: This checkIfPlacedCorrectly should not be called every loop iteration.
		//			It needs to run only on a regular timer to be consistent with how fast it responds.
		bool correct = pieces[i].getIsPlacedCorrectly();
```

```
                        if (correct && !pieces[i].isTimeLocked()) {
                                // Play placed correctly sound here
                                sound_player->playRandomPiecePlacedSound();

                                processPlacementOfPiece(pieces[i]);
                                pieces[i].setTimeLock();
                        }
                }
                //if (numberCorrectlyPlaced == pieces.size()) { allCorrect = true; }

                if (this->Game->isEndGame() || allCorrect) {
                        if(allCorrect)
                        {
                                //play game completed sound
                                this->gameRecord->setTimeCompletedToNow();
                                sound_player->play_Sound(sound_game_completed);
                        }
                        KillTimer(hwnd1, myTimer); // kill the timers
                        KillTimer(hwnd2, myTimer2);
                        destroyAllWindows(); // shut everything down.
                        endTrack();
                        return 0;
                }
        }
        return 0;
}
```

CALIBRATIONTRACKING.H

```
/* This class controls the operation of OpenCV.  It starts OpenCV running, monitors/controls tracking, gathers time data, and shuts
OpenCV down once the game is completed or stopped.  An instance of this class is created in "StartOpenCV.cpp" when the user hits
the Run button on the GUI
*/
#pragma once

#include "stdafx.h"
#include <vcclr.h>
#include <opencv2\opencv.hpp>       //includes all OpenCV headers
#include "GameBoard.h"
#include "Functions.h"

ref class CalibrationTracking
{

 public:

                bool IS_STOPPED;
                bool waitingForUserToPlacePieces;
                bool doneWithLocationTracking;

                CalibrationTracking() { Initialize(); }

                CalibrationTracking^ returnHandle() { return this; }
    virtual void Initialize();
    virtual void Start();
                virtual void startLocationCalibration();
    virtual void Stop();
                virtual void Next() { NEXT = true; }
```

```
    virtual void setGame(KnobPuzzle^ game) {this->Game = game;}
                virtual void setPieceToTrack(PuzzlePiece^ piece) {this->pieceBeingTracked = piece; }

protected:

    bool STOP;
                bool NEXT;
                int iterator;

    KnobPuzzle^ Game;
                PuzzlePiece^ pieceBeingTracked;

                virtual int startTrackColor();
                virtual int startTrackLocation();
    virtual void endTrack();
                virtual void nextPiece();
                virtual void savePieceInformation();
    virtual void trackTrackedPiece(TrackedPiece &piece, Mat &camera_feed, Mat &HSV_image, Mat &threshold_image);
    virtual List<int>^ findPieceLocation(TrackedPiece &piece, Mat &camera_feed, Mat &HSV_image, Mat &threshold_image);
    virtual List<int>^ trackFilteredObject(TrackedPiece &piece, Mat &cameraFeed, Mat &threshold_image);
    //virtual List<int>^ trackFilteredObject2(TrackedPiece &piece, Mat &cameraFeed, Mat &threshold_image);
                void createTrackbarWindow(TrackedPiece &tmp);
    void erodeAndDilate(Mat &image);
    void drawObject(vector<TrackedPiece> pieces, Mat &frame);


private:

                HANDLE myMutex;

                // window names
    System::String^ original_window;
    System::String^ trackbar_window;
    System::String^ filtered_window;
    System::String^ puzzle_window;

    //default capture width and height
    int FRAME_WIDTH;
    int FRAME_HEIGHT;
    //max number of objects to be detected in frame
    int MAX_NUM_OBJECTS;
    //minimum and maximum object area
    int MIN_OBJECT_AREA;
    int MAX_OBJECT_AREA;

                // min&max calibration values
    int H_min;
    int H_max;
    int S_min;
    int S_max;
    int V_min;
    int V_max;

                // temporary calibration values
                int calibrate_H_min;
    int calibrate_H_max;
    int calibrate_S_min;
    int calibrate_S_max;
    int calibrate_V_min;
    int calibrate_V_max;

                void drawBoard();
```

```
};
```

CALIBRATIONTRACKING.CPP

```
/*  This file includes all the functions that are directly related to the maintenance of the CalibrationTracking class,
e.g. initializing, starting, ending.  Tracking functions are located in "Tracking.cpp"
*/

#include "stdafx.h"

#include <opencv2\opencv.hpp>
#include <vcclr.h>
#include <iostream>
#include <string>
#include <vector>
#include "TrackedPiece.h"
#include "Functions.h"
#include "CalibrationTracking.h"
#include "Shape.h"

using namespace cv;
using namespace std;


// initialize all variables upon creation of class
void CalibrationTracking::Initialize() {

                        // window names
        this->original_window = "Original Capture";
        this->trackbar_window = "Trackbar Window";
        this->filtered_window = "Filtered Image";
        this->puzzle_window = "Puzzle Board Window";

        //default capture width and height
        this->FRAME_WIDTH = Constants::DEFAULT_FRAME_WIDTH;
        this->FRAME_HEIGHT = Constants::DEFAULT_FRAME_HEIGHT;

        //max number of objects to be detected in frame
        this->MAX_NUM_OBJECTS= Constants::DEFAULT_MAX_OBJECTS_IN_FRAME;

        //minimum and maximum object area & HSV
        this->MIN_OBJECT_AREA = Constants::DEFAULT_MIN_OBJECT_AREA;
        this->MAX_OBJECT_AREA = FRAME_HEIGHT*FRAME_WIDTH/1.5;
        this->H_min = Constants::DEFAULT_H_MIN;
        this->H_max = Constants::DEFAULT_H_MAX;
        this->S_min = Constants::DEFAULT_S_MIN;
        this->S_max = Constants::DEFAULT_S_MAX;
        this->V_min = Constants::DEFAULT_V_MIN;
        this->V_max = Constants::DEFAULT_V_MAX;

                        // holder for temporary HSV trackbar values
        this->calibrate_H_min = Constants::DEFAULT_H_MIN;
        this->calibrate_H_max = Constants::DEFAULT_H_MAX;
        this->calibrate_S_min = Constants::DEFAULT_S_MIN;
        this->calibrate_S_max = Constants::DEFAULT_S_MAX;
        this->calibrate_V_min = Constants::DEFAULT_V_MIN;
        this->calibrate_V_max = Constants::DEFAULT_V_MAX;

                        // management variables
                        this->STOP = false;
```

```
		this->NEXT = false;
		this->iterator = 0;
		this->waitingForUserToPlacePieces = false;
		this->doneWithLocationTracking = false;


		this->myMutex = CreateMutex(NULL, FALSE, (LPCWSTR) "calibration");
}
//-----------------------------------------------------------------------------------------------------

// start color tracking algorithm
void CalibrationTracking::Start() {
	// lets lock onto this thread just for safety sake
	startTrackColor();
}


//-----------------------------------------------------------------------------------------------------

// start location tracking algorithm
void CalibrationTracking::startLocationCalibration() {

	// display the puzzle board
//	cv::Mat board = displayPuzzleBoard();
//	imshow("game_board", board);
	drawBoard();
	moveWindow("game_board", -1600, 0);
	cvSetWindowProperty("game_board", CV_WND_PROP_FULLSCREEN, CV_WINDOW_FULLSCREEN);		// Makes full screen

	// wait for user to place pieces (gui form will send signal when done)
	while (this->waitingForUserToPlacePieces) {
		waitKey(70);
		if (this->STOP) {
			endTrack();
			return;
		}
	}

	// track locations
	startTrackLocation();
	cv::destroyAllWindows();
}
//-----------------------------------------------------------------------------------------------------

void CalibrationTracking::savePieceInformation() {

		// changing global data here, so lock down thread
		WaitForSingleObject(myMutex, INFINITE);

		// pull data from the calibration values and plug back into puzzle piece
		List<int>^ HSV_min_list = gcnew List<int>();
		List<int>^ HSV_max_list = gcnew List<int>();
		HSV_min_list->Add(this->calibrate_H_min); HSV_min_list->Add(this->calibrate_S_min); HSV_min_list->Add(this-
>calibrate_V_min);
		HSV_max_list->Add(this->calibrate_H_max); HSV_max_list->Add(this->calibrate_S_max); HSV_max_list->Add(this-
>calibrate_V_max);
		this->Game->getPieceList()[this->iterator]->setHSVmin(HSV_min_list);
		this->Game->getPieceList()[this->iterator]->setHSVmax(HSV_max_list);

		ReleaseMutex(myMutex);
}
```

```
// move to the next piece for calibration
void CalibrationTracking::nextPiece() {

                        // lock down thread
                        WaitForSingleObject(myMutex, INFINITE);

                        //destroy old trackbar window
                        destroyWindow(systemStringToStdString(trackbar_window));

                        // update iterator. If iterator has passed the # of pieces, end calibration
                        this->iterator = this->iterator + 1;
                        if (this->iterator >= this->Game->getPieceList()->Count) {
                                endTrack();
                                ReleaseMutex(myMutex);
                                return;
                        }

                        // pull new piece and convert to trackedpiece
                        TrackedPiece tmp = puzzlePieceToTrackedPiece(this->Game->getPieceList()[this->iterator]);
                        createTrackbarWindow(tmp);              // create new trackbar window based on new initial values
                        this->NEXT = false;
                        ReleaseMutex(myMutex);
}

void CalibrationTracking::Stop() {
            STOP = true;
            //WaitForSingleObject(myMutex, INFINITE);
            //cv::destroyAllWindows();
            //ReleaseMutex(myMutex);
}

//-------------------------------------------------------------------------------------------------

// end tracking, 'clean up' game.  this instance of the class will now end (though that might change in the future)
void CalibrationTracking::endTrack() {
                        WaitForSingleObject(myMutex, INFINITE);
                        System::Console::WriteLine("CalibrationTracking::EndTrack() : Exiting CalibrationTracking");
                        cv::destroyAllWindows();
                        destroyWindow(systemStringToStdString(original_window));
                        destroyWindow(systemStringToStdString(trackbar_window));
                        destroyWindow(systemStringToStdString(filtered_window));
                        destroyWindow(systemStringToStdString(puzzle_window));
                        this->IS_STOPPED = true;
                        ReleaseMutex(myMutex);
}


////-------------------------------------------------------------------------------------------------

void CalibrationTracking::createTrackbarWindow(TrackedPiece &tmp)
{

                        namedWindow(systemStringToStdString(trackbar_window));

      this->calibrate_H_min = tmp.getHSVmin()[0];
      this->calibrate_H_max = tmp.getHSVmax()[0];
      this->calibrate_S_min = tmp.getHSVmin()[1];
      this->calibrate_S_max = tmp.getHSVmax()[1];
      this->calibrate_V_min = tmp.getHSVmin()[2];
      this->calibrate_V_max = tmp.getHSVmax()[2];
```

```
                    // now have to cast everything back to int* (I DO NOT TRUST THIS COMPLETELY; I FEAR IT MIGHT CAUSE CRASHES OR MEMORY LEAKS)
                    pin_ptr<int> pinned_H_min = &this->calibrate_H_min;
                    pin_ptr<int> pinned_H_max = &this->calibrate_H_max;
                    pin_ptr<int> pinned_S_min = &this->calibrate_S_min;
                    pin_ptr<int> pinned_S_max = &this->calibrate_S_max;
                    pin_ptr<int> pinned_V_min = &this->calibrate_V_min;
                    pin_ptr<int> pinned_V_max = &this->calibrate_V_max;
                    //System::String^ values = H_min2 + " " + H_max2 + " " + S_min2 + " " + S_max2 + " " + V_min2 + " " + V_max2;
                    // int createTrackbar(const string& trackbarname, const string& winname, int* value, int count, TrackbarCallback onChange=0, void* userdata=0)
                    // value is the the location of the sliding thing, and count is the max value of the whole slider (min is always 0)
                    createTrackbar( "H_MIN", systemStringToStdString(trackbar_window),  pinned_H_min, H_max, on_trackbar );
                    createTrackbar( "H_MAX", systemStringToStdString(trackbar_window),  pinned_H_max, H_max, on_trackbar );
                    createTrackbar( "S_MIN", systemStringToStdString(trackbar_window),  pinned_S_min, S_max, on_trackbar );
                    createTrackbar( "S_MAX", systemStringToStdString(trackbar_window), pinned_S_max, S_max, on_trackbar );
                    createTrackbar( "V_MIN", systemStringToStdString(trackbar_window), pinned_V_min, V_max, on_trackbar );
                    createTrackbar( "V_MAX", systemStringToStdString(trackbar_window), pinned_V_max, V_max, on_trackbar );
}


////-----------------------------------------------------------------------------------------------------------
// this can be combined with the runtracking version
void CalibrationTracking::erodeAndDilate(Mat &image)
{
        //create structuring element that will be used to "dilate" and "erode" image.
        //the element chosen here is a 3px by 3px rectangle

        Mat erodeElement = getStructuringElement( MORPH_RECT,Size(3,3));
        //dilate with larger element so make sure object is nicely visible
        Mat dilateElement = getStructuringElement( MORPH_RECT,Size(8,8));

        erode(image,image,erodeElement);
        erode(image,image,erodeElement);


        dilate(image,image,dilateElement);
        dilate(image,image,dilateElement);
}

////-----------------------------------------------------------------------------------------------------------

void CalibrationTracking::drawObject(vector<TrackedPiece> pieces, Mat &frame){

        for(int i =0; i<pieces.size(); i++){

                cv::circle(frame,cv::Point(pieces.at(i).getXPos(),pieces.at(i).getYPos()),10,cv::Scalar(0,0,255));
                cv::putText(frame,intToStdString(pieces.at(i).getXPos())+ " , " + intToStdString(pieces.at(i).getYPos()),cv::Point(pieces.at(i).getXPos(),pieces.at(i).getYPos()+20),1,1,Scalar(0,255,0));
                cv::putText(frame,pieces.at(i).getName(),cv::Point(pieces.at(i).getXPos(),pieces.at(i).getYPos()-30),1,2,pieces.at(i).getColor());
        }
}

////-----------------------------------------------------------------------------------------------------------

List<int>^ CalibrationTracking::trackFilteredObject(TrackedPiece &piece, Mat &cameraFeed, Mat &threshold_image){

        vector <TrackedPiece> pieces;
        List<int>^ returnList = gcnew List<int>;
        //returnList->Add(0); returnList->Add(0);
        Mat temp;
        threshold_image.copyTo(temp);
```

```
	//these two vectors needed for output of findContours
	vector< vector<Point> > contours;
	vector<Vec4i> hierarchy;
	//find contours of filtered image using openCV findContours function
	findContours(temp,contours,hierarchy,CV_RETR_CCOMP,CV_CHAIN_APPROX_SIMPLE );
	//use moments method to find our filtered object
	bool objectFound = false;
	if (hierarchy.size() > 0) {
		int numObjects = hierarchy.size();
//		cout << "Num objects: " << numObjects << endl;
//		cout << "Max Num objects: " << MAX_NUM_OBJECTS << endl;
		// threshholed to calculate movement
		const int thresh = 40;
		//saves max area of each contour detected so only the largest one will be tracked
		double maxArea = 0;
		// temporary piece for contours found
		TrackedPiece tmp;

		//if number of objects greater than MAX_NUM_OBJECTS we have a noisy filter
		if(numObjects < MAX_NUM_OBJECTS){
			// for each object (contour) detected
			for (int index = 0; index >= 0; index = hierarchy[index][0]) {

				// get the moment of the contour
				Moments moment = moments((cv::Mat)contours[index]);
				// get the area from the moment
				double area = moment.m00;
//				cout << "Area " << index << " is: " << area << endl;

				// if the area is less than MIN_OBJECT_AREA then it is probably just noise
				// it must also be large than the max area found so far since we only want the largest area.
				if(area > MIN_OBJECT_AREA && area > maxArea){
					// set new max area
					maxArea = area;
					// Clear previous objects found so only one (the biggest) is detected
					pieces.clear();

					int xPos = moment.m10/area;
					int yPos = moment.m01/area;
					returnList->Add(xPos);
					returnList->Add(yPos);

					tmp.setXPos(xPos);
					tmp.setYPos(yPos);
					tmp.setName(piece.getName());
					tmp.setColor(piece.getColor());

					//cout << piece.getName() << ": x: " << xPos << " y: " << yPos << endl;
					//cout << "LastPos: x: " << piece.getLastxPos() << " y: " << piece.getLastyPos() << endl;

					pieces.push_back(tmp);

					objectFound = true;
				}
			}

			//let user know you found an object and check for movement
			if(objectFound ==true){

				// Update piece location (tmp piece should now be biggest contour found)
				piece.setXPos(tmp.getXPos());
				piece.setYPos(tmp.getYPos());
```

```
                                        //draw object location on screen
                                        drawObject(pieces,cameraFeed);}

                        }else
                        {
                                putText(cameraFeed,"TOO MUCH NOISE! ADJUST FILTER",Point(0,50),1,2,Scalar(0,0,255),2);
                        }
        }
        return returnList;
}

////-----------------------------------------------------------------------------------------------------

void CalibrationTracking::trackTrackedPiece(TrackedPiece &piece, Mat &camera_feed, Mat &HSV_image, Mat &threshold_image)
{
        //convert to binary image with white = in range specified
        inRange(HSV_image, piece.getHSVmin(), piece.getHSVmax(), threshold_image);
        erodeAndDilate(threshold_image);
        trackFilteredObject(piece, camera_feed, threshold_image);

}

////-----------------------------------------------------------------------------------------------------

List<int>^ CalibrationTracking::findPieceLocation(TrackedPiece &piece, Mat &camera_feed, Mat &HSV_image, Mat &threshold_image)
{
        //convert to binary image with white = in range specified
        inRange(HSV_image, piece.getHSVmin(), piece.getHSVmax(), threshold_image);
        erodeAndDilate(threshold_image);
        List<int>^ returnedList = trackFilteredObject(piece, camera_feed, threshold_image);
        //System::String^ tmp = returnedList[0] + " " + returnedList[1];
        //System::Windows::Forms::MessageBox::Show(tmp);
        return returnedList;

}

////-----------------------------------------------------------------------------------------------------

int CalibrationTracking::startTrackLocation()
{

        VideoCapture capture;
        capture.open(0);     //0 is default video device, 1 is other/USB camera
        //set height and width of capture frame
        capture.set(CV_CAP_PROP_FRAME_WIDTH,FRAME_WIDTH);
        capture.set(CV_CAP_PROP_FRAME_HEIGHT,FRAME_HEIGHT);

        if (!capture.isOpened())
        {
                Console::WriteLine("CalibrationTracking::startTrackLocation():: Error - Cannot open camera.");
                return 1;
        }
        Console::WriteLine("CalibrationTracking::startTrackLocation():: Camera Opened");

        //// set up filtered and original windows
        namedWindow(systemStringToStdString(original_window));
        namedWindow(systemStringToStdString(filtered_window));

        Mat camera_feed;              //raw camera image
        Mat HSV_image;                        //camera image converted to HSV
        Mat threshold_image;          //image after HSV is filtered and processed
```

```
	List<int>^ Xcoords;
	List<int>^ Ycoords;

	// loop through each puzzle piece individually
	for each (PuzzlePiece^ currentPiece in this->Game->getPieceList()) {
		Xcoords = gcnew List<int>();
		Ycoords = gcnew List<int>();

		// for each puzzle piece, find location coordinates 20 times
		for (int i = 0; i < 20; i++)
		{
			capture.read(camera_feed);

			cvtColor(camera_feed, HSV_image, CV_BGR2HSV);

			// track for calibration
			TrackedPiece tmp = puzzlePieceToTrackedPiece(currentPiece);
			List<int>^ tmpList = findPieceLocation(tmp, camera_feed, HSV_image, threshold_image);
			if (tmpList->Count == 2) {
				Xcoords->Add(tmpList[0]);
				Ycoords->Add(tmpList[1]);
			}

			// show updated windows
			imshow(systemStringToStdString(filtered_window), threshold_image);
			imshow(systemStringToStdString(original_window), camera_feed);

			waitKey(30);
		}
		// average those coordinates
		// if no coordinates were found, will return (0,0)
		if (Xcoords->Count !=0 && Ycoords->Count != 0) {
			double x = averageListOfInts(Xcoords);
			double y = averageListOfInts(Ycoords);
			// now set the X and Y destinations of that piece using these averaged coordinates
			currentPiece->setDestPos(x,y);
		}
		else {
			currentPiece->setDestPos(0,0);
		}
	}

	// release unmanaged components (I'm not sure if this actually works)
	cv::destroyAllWindows();
	endTrack();
	camera_feed.release();
	HSV_image.release();
	threshold_image.release();
	capture.release();

	return 0;
}


////-----------------------------------------------------------------------------------------------------

int CalibrationTracking::startTrackColor()
{

	VideoCapture capture;
```

```
	capture.open(0);	//0 is default video device, 1 is other/USB camera
	//set height and width of capture frame
	capture.set(CV_CAP_PROP_FRAME_WIDTH,FRAME_WIDTH);
	capture.set(CV_CAP_PROP_FRAME_HEIGHT,FRAME_HEIGHT);

	if (!capture.isOpened())
	{
		cout << "Cannot open camera." << endl;
		return 1;
	}
	cout << "Camera opened" << endl;

	Mat camera_feed;		//raw camera image
	Mat HSV_image;			//camera image converted to HSV
	Mat threshold_image;		//image after HSV is filtered and processed

	TrackedPiece tmp = puzzlePieceToTrackedPiece(this->Game->getPieceList()[this->iterator]);
	createTrackbarWindow(tmp);

	// display the puzzle board
	drawBoard();

	this->iterator = 0;

	//// set up filtered and original windows
	std::string originalwindow = systemStringToStdString(original_window);
	std::string filteredwindow = systemStringToStdString(filtered_window);
	namedWindow(originalwindow);
	moveWindow(originalwindow, 640, 0);
	namedWindow(filteredwindow);
	moveWindow(filteredwindow, 640, 512);

	while(1)
	{
		capture.read(camera_feed);

		cvtColor(camera_feed, HSV_image, CV_BGR2HSV);

		// track for calibration
		TrackedPiece tmp = TrackedPiece(systemStringToStdString(this->Game->getPieceList()[this->iterator]->getName()),
Scalar(calibrate_H_min, calibrate_S_min, calibrate_V_min), Scalar(calibrate_H_max, calibrate_S_max, calibrate_V_max));
		trackTrackedPiece(tmp, camera_feed, HSV_image, threshold_image);
		imshow(systemStringToStdString(filtered_window), threshold_image);

		imshow(systemStringToStdString(original_window), camera_feed);

		waitKey(30);
		if (STOP) {
			savePieceInformation();
			endTrack();
			cv::destroyAllWindows();
			camera_feed.release();
			HSV_image.release();
			threshold_image.release();
			capture.release();
			break;
		}
		if (NEXT) {
			savePieceInformation();
			nextPiece();
		}
	}
```

```
		cv::destroyAllWindows();
		camera_feed.release();
		HSV_image.release();
		threshold_image.release();
		capture.release();
		return 0;
}

void CalibrationTracking::drawBoard() {
		// display the puzzle board
		cv::Mat puzzleBoard;
		vector<TrackedPiece> trackedPieces = vector<TrackedPiece>(this->Game->getPieceList()->Count);
		for each (PuzzlePiece^ managedPiece in this->Game->getPieceList()) {
			trackedPieces.push_back(puzzlePieceToTrackedPiece(managedPiece));
		}

		puzzleBoard = displayPuzzleBoard(puzzleBoard, trackedPieces);

		imshow("game_board", puzzleBoard);
		cv::moveWindow("game_board", -1600, 0);
		cvSetWindowProperty("game_board", CV_WND_PROP_FULLSCREEN, CV_WINDOW_FULLSCREEN);		// Makes full screen
}
```

SHAPE.H

```
#pragma once

#ifndef SHAPE_H
#define SHAPE_H

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include "TrackedPiece.h"
//#include "Functions.h"

#define SCREEN_WIDTH 1600
#define SCREEN_HEIGHT 900

using namespace cv;
using namespace std;

ref class Constants;

enum shapeType{	Circle,
		Square,
		Triangle,
		Pentagon,
		Rectangular,	//"Rectangle" was previously defined in some Windows file...
		Arrow,
		Star};

class Shape
{
public:
	//-----Constructors & Destructor-----
	Shape(Mat* img);
			Shape() {}
	~Shape(void);
			void setImage(Mat* img);
```

```
                    void endImage();

    //-----Accessors
    inline Mat* get_image() const {return image;}
    inline Point get_start() const {return start;}
    inline Point get_end() const {return end;}
    inline Point get_startingPoint() const {return startingPoint;}
    inline int get_height() const {return height;}
    inline int get_width() const {return width;}


    //-----Mutators-----
    //THIS IS NOT DELETE SHAPE; it delets all images and leaves image black
    inline void Clear_To_Black() {*image = Mat::zeros(SCREEN_HEIGHT, SCREEN_WIDTH, CV_8UC3);}
                    inline void Clear_To_Gray() {*image =
Scalar(Constants::BACKGROUND_COLOR,Constants::BACKGROUND_COLOR,Constants::BACKGROUND_COLOR);}
    inline void set_color(Scalar new_color) {color = new_color;}
                    inline void set_startingPoint(Point strt) {startingPoint = strt;}
                    inline void set_height(int num) {height = num;}
                    inline void set_width(int num) {width = num;}

    //-----Drawing Functions------
                    void Draw_Shape(TrackedPiece piece, double dim_factor);
    void Draw_Circle(Point middle, int radius, int thickness = 1, int lineType = 8);
    void Draw_Rectangle(Point corner, int wid, int heig, int thinkness = 1, int lineType = 8);
    void Draw_Square(Point corner, int wid, int thickness = 1, int lineType = 8);
    void Draw_Triangle(Point top, int length, int thickness = 1, int lineType = 8);
    void Draw_Pentagon(Point top, int length, int thickness = 1, int lineType = 8);
    void Draw_Star(Point top, int length, int thickness = 1, int lineType = 8);
    void Draw_Arrow(Point begin, Point end, int thickness = 1, int lineType = 8);
                    void Draw_Isosceles(int thickness = 1, int lineType = 8);
                    void Draw_House(int thickness = 1, int lineType = 8);
                    void Draw_Tree(int thickness = 1, int lineType = 8);
                    void Draw_Door(int thickness = 1, int lineType = 8);
                    void Draw_Sun(int thickness = 1, int lineType = 8);

    //-----Other Methods------
    void setColor(Scalar BGR_val) {color = BGR_val;}


private:
    Mat* image;
    Point start;
    Point end;
    Point startingPoint;
    int height;
    int width;
    int line_thickness;
    Scalar color;
    shapeType type;


};

#endif //#ifndef SHAPE_
```

SHAPE.CPP

```
#include "stdafx.h"
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
```

```
#include <iostream>
#include "Shape.h"
#include "Functions.h"

using namespace cv;
using namespace std;

Shape::Shape(Mat* img)
{
	image = img;
	color = Scalar(0,0,255);
}


Shape::~Shape(void)
{
}

void Shape::setImage(Mat* img) {
		image = img;
		color = Scalar(0,0,255);
}

// close puzzle board window
void Shape::endImage() {
		//cv::destroyAllWindows();
		//cvReleaseImage(this->image);
		this->image->release();
}

//------------Drawing Methods------------

// Pick which shape to draw
void Shape::Draw_Shape(TrackedPiece piece, double dim_factor)
{
		std::string shapeType = piece.getName();
		if (shapeType == "Circle") {
			// Making circle slightly orange so it is not mistaken for puzzle piece
			setColor(Scalar(0, 0 * dim_factor, 255 * dim_factor));
			Draw_Circle(Point(piece.getShapePointX(),piece.getShapePointY()), piece.getShapeRadius(),
Constants::SHAPE_LINE_WIDTH);
		}
		else if (shapeType == "Rectangle") {
			setColor(Scalar(0, 255 * dim_factor, 0));
			Draw_Rectangle(Point(piece.getShapePointX(),piece.getShapePointY()), piece.getShapeWidth(),piece.getShapeHeight(),
Constants::SHAPE_LINE_WIDTH);
		}
		else if (shapeType == "Square") {
			setColor(Scalar(255 * dim_factor, 0, 0));
			Draw_Square(Point(piece.getShapePointX(),piece.getShapePointY()), piece.getShapeWidth(),
Constants::SHAPE_LINE_WIDTH);
		}
		else if (shapeType == "Triangle") {
			setColor(Scalar(255 * dim_factor, 0, 255 * dim_factor));
			Draw_Triangle(Point(piece.getShapePointX(),piece.getShapePointY()), piece.getShapeLength(),
Constants::SHAPE_LINE_WIDTH);
		}
		else if (shapeType == "Pentagon") {
			setColor(Scalar(0, 255 * dim_factor, 255 * dim_factor));
			Draw_Pentagon(Point(piece.getShapePointX(),piece.getShapePointY()), piece.getShapeLength(),
Constants::SHAPE_LINE_WIDTH);
		}
```

```
	else if (shapeType == "Isosceles") {
		setColor(Scalar(255 * dim_factor, 0, 255 * dim_factor));
		Draw_Isosceles(Constants::SHAPE_LINE_WIDTH);
	}
	else if (shapeType == "House") {
		setColor(Scalar(0, 0, 255 * dim_factor));
		Draw_House(Constants::SHAPE_LINE_WIDTH);
	}
	else if (shapeType == "Tree") {
		setColor(Scalar(0, 255 * dim_factor, 0));
		Draw_Tree(Constants::SHAPE_LINE_WIDTH);
	}
	else if (shapeType == "Door") {
		setColor(Scalar(255 * dim_factor, 0, 0));
		Draw_Door(Constants::SHAPE_LINE_WIDTH);
	}
		else if (shapeType == "Sun") {
		setColor(Scalar(0, 255 * dim_factor, 255 * dim_factor));
		Draw_Sun(Constants::SHAPE_LINE_WIDTH);
	}
	else {
		System::String^ str = "Error: cannot draw piece \"" + stdStringToSystemString(shapeType) + "\". Not a recognized shape.";
		//System::Windows::Forms::MessageBox::Show(str);
		System::Console::WriteLine(str);
	}
}


//Circle
void Shape::Draw_Circle(Point middle, int radius, int thickness, int lineType)
{
	type = Circle;
	startingPoint = middle;
	width = radius * 2;
	height = radius * 2;
	line_thickness = thickness;
	start = Point(-1,-1);
	end = Point(-1,-1);

	circle(*image,
		middle,
		radius,
		color,
		thickness,
		lineType);
}

//Rectangle
void Shape::Draw_Rectangle(Point corner, int wid, int heig, int thickness, int lineType)
{
	type = Rectangular;
	startingPoint = corner;
	width = wid;
	height = heig;
	line_thickness = thickness;
	start = Point(-1,-1);
	end = Point(-1,-1);

	rectangle(*image,
		startingPoint,
		Point(corner.x+wid, corner.y+heig),
		color,
```

```
                    thickness,
                    lineType);
}

//Square
void Shape::Draw_Square(Point corner, int wid, int thickness, int lineType)
{
    Draw_Rectangle(corner, wid, wid, thickness, lineType);
    type = Square;
}

//Triangle
// length is length of one side of equilateral triangle
void Shape::Draw_Triangle(Point top, int length, int thickness, int lineType)
{
    type=Triangle;
    startingPoint = top;
    width = length;
    height = length * (sqrt(3)/2);

    line_thickness = thickness;
    start = Point(-1,-1);
    end = Point(-1,-1);

    Point triangle_points[1][3];
    triangle_points[0][0] = Point(top.x, top.y);
    triangle_points[0][1] = Point(top.x - (length/2), top.y + height);
    triangle_points[0][2] = Point(top.x + (length/2), top.y + height);


    if (thickness == -1)
    {
        const Point* ppt[1] = {triangle_points[0]};
        int npt[] = {3};
        fillPoly(*image,
                    ppt,
                    npt,
                    1,
                    color,
                    lineType);
    }
    else
    {
        line(*image,
            Point(triangle_points[0][0].x, triangle_points[0][0].y),
            Point(triangle_points[0][1].x, triangle_points[0][1].y),
            color,
            thickness,
            lineType);

        line(*image,
            Point(triangle_points[0][1].x, triangle_points[0][1].y),
            Point(triangle_points[0][2].x, triangle_points[0][2].y),
            color,
            thickness,
            lineType);

        line(*image,
            Point(triangle_points[0][2].x, triangle_points[0][2].y),
            Point(triangle_points[0][0].x, triangle_points[0][0].y),
            color,
            thickness,
```

```
                lineType);
        }
}

//Pentagon
void Shape::Draw_Pentagon(Point top, int length, int thickness, int lineType)
{
        type = Pentagon;
        startingPoint = top;
        width = 1.61803398875*length;
        height = 1.538909039*length;
        start = Point(-1,-1);
        end = Point(-1,-1);

        line_thickness = thickness;

        Point p1 = Point(top.x + (0.80901699437*length), top.y + (0.58778525229*length));
        Point p2 = Point(p1.x - (0.30901699436*length), top.y + height);
        Point p3 = Point(p2.x - length, p2.y);
        Point p4 = Point(top.x - (0.80901699437*length), p1.y);

        Point pentagon_points[1][5];
        pentagon_points[0][0] = top;
        pentagon_points[0][1] = p1;
        pentagon_points[0][2] = p2;
        pentagon_points[0][3] = p3;
        pentagon_points[0][4] = p4;

        if (thickness == -1)
        {
                const Point* ppt[1] = {pentagon_points[0]};
                int npt[] = { 5 };
                fillPoly(*image,
                                ppt,
                                npt,
                                1,
                                color,
                                lineType);
        }
        else
        {
                line(*image,
                        Point(pentagon_points[0][0].x, pentagon_points[0][0].y),
                        Point(pentagon_points[0][1].x, pentagon_points[0][1].y),
                        color,
                        thickness,
                        lineType);

                line(*image,
                        Point(pentagon_points[0][1].x, pentagon_points[0][1].y),
                        Point(pentagon_points[0][2].x, pentagon_points[0][2].y),
                        color,
                        thickness,
                        lineType);

                line(*image,
                        Point(pentagon_points[0][2].x, pentagon_points[0][2].y),
                        Point(pentagon_points[0][3].x, pentagon_points[0][3].y),
                        color,
                        thickness,
                        lineType);
```

```
                line(*image,
                        Point(pentagon_points[0][3].x, pentagon_points[0][3].y),
                        Point(pentagon_points[0][4].x, pentagon_points[0][4].y),
                        color,
                        thickness,
                        lineType);

                line(*image,
                        Point(pentagon_points[0][4].x, pentagon_points[0][4].y),
                        Point(pentagon_points[0][0].x, pentagon_points[0][0].y),
                        color,
                        thickness,
                        lineType);
        }

}

//Star
void Shape::Draw_Star(Point top, int length, int thickness, int lineType)
{
        type = Star;
        startingPoint = top;
        width = 1.61803398875*length;
        height = 1.538909039*length;
        start = Point(-1,-1);
        end = Point(-1,-1);

        line_thickness = thickness;

        Point p1 = Point(top.x + (0.80901699437*length), top.y + (0.58778525229*length));
        Point p2 = Point(p1.x - (0.30901699436*length), top.y + height);
        Point p3 = Point(p2.x - length, p2.y);
        Point p4 = Point(top.x - (0.80901699437*length), p1.y);

        Point pentagon_points[1][5];
        pentagon_points[0][0] = top;
        pentagon_points[0][1] = p2;
        pentagon_points[0][2] = p4;
        pentagon_points[0][3] = p1;
        pentagon_points[0][4] = p3;

        if (thickness == -1)
        {
                const Point* ppt[1] = {pentagon_points[0]};
                int npt[] = { 5 };
                fillPoly(*image,
                                ppt,
                                npt,
                                1,
                                color,
                                lineType);
                Shape temp_shape = Shape(image);
                temp_shape.Draw_Pentagon( Point( top.x, top.y + (length * 1.175) ), -(length * 0.38), -1 );

        }
        else
        {
                line(*image,
                        Point(pentagon_points[0][0].x, pentagon_points[0][0].y),
                        Point(pentagon_points[0][1].x, pentagon_points[0][1].y),
                        color,
                        thickness,
```

```
                    lineType);

            line(*image,
                    Point(pentagon_points[0][1].x, pentagon_points[0][1].y),
                    Point(pentagon_points[0][2].x, pentagon_points[0][2].y),
                    color,
                    thickness,
                    lineType);

            line(*image,
                    Point(pentagon_points[0][2].x, pentagon_points[0][2].y),
                    Point(pentagon_points[0][3].x, pentagon_points[0][3].y),
                    color,
                    thickness,
                    lineType);

            line(*image,
                    Point(pentagon_points[0][3].x, pentagon_points[0][3].y),
                    Point(pentagon_points[0][4].x, pentagon_points[0][4].y),
                    color,
                    thickness,
                    lineType);

            line(*image,
                    Point(pentagon_points[0][4].x, pentagon_points[0][4].y),
                    Point(pentagon_points[0][0].x, pentagon_points[0][0].y),
                    color,
                    thickness,
                    lineType);
        }

}

//Arrow
void Shape::Draw_Arrow(Point begin, Point end, int thickness, int lineType)
{

}

//Isosceles
void Shape::Draw_Isosceles(int thickness, int lineType)
{
   Point top = Point(790,70);
   width = 420;
   height = 220;

   line_thickness = thickness;
   start = Point(-1,-1);
   end = Point(-1,-1);

        Point triangle_points[1][3];
   triangle_points[0][0] = Point(top.x, top.y);
   triangle_points[0][1] = Point(top.x - (width/2), top.y + height);
   triangle_points[0][2] = Point(top.x + (width/2), top.y + height);


   if (thickness == -1)
   {
      const Point* ppt[1] = {triangle_points[0]};
      int npt[] = {3};
      fillPoly(*image,
            ppt,
```

```
                npt,
                1,
                color,
                lineType);
        }
        else
        {
            line(*image,
                                                    Point(triangle_points[0][0].x, triangle_points[0][0].y),
                                        Point(triangle_points[0][1].x, triangle_points[0][1].y),
                color,
                thickness,
                lineType);

            line(*image,
                    Point(triangle_points[0][1].x, triangle_points[0][1].y),
                    Point(triangle_points[0][2].x, triangle_points[0][2].y),
                    color,
                    thickness,
                    lineType);

            line(*image,
                    Point(triangle_points[0][2].x, triangle_points[0][2].y),
                    Point(triangle_points[0][0].x, triangle_points[0][0].y),
                    color,
                    thickness,
                    lineType);
        }
}

//House
void Shape::Draw_House(int thickness, int lineType)
{
            //Top Rectangle
            Point startingPoint = Point(582,355);
    int wid = 420;
    int heig = 100;
    line_thickness = thickness;
    start = Point(-1,-1);
    end = Point(-1,-1);

    rectangle(*image,
                                                startingPoint,
                                                Point(startingPoint.x+wid, startingPoint.y+heig),
                                                color,
                                                thickness,
                                                lineType);

            //Left Rectangle
            startingPoint = Point(582,355);
    wid = 85;
    heig = 403;
    line_thickness = thickness;
    start = Point(-1,-1);
    end = Point(-1,-1);

    rectangle(*image,
                                                startingPoint,
                                                Point(startingPoint.x+wid, startingPoint.y+heig),
                                                color,
                                                thickness,
                                                lineType);
```

```
        //Right Rectangle
        startingPoint = Point(917,355);
    wid = 85;
    heig = 403;
    line_thickness = thickness;
    start = Point(-1,-1);
    end = Point(-1,-1);

    rectangle(*image,
                                        startingPoint,
                                        Point(startingPoint.x+wid, startingPoint.y+heig),
                                        color,
                                        thickness,
                                        lineType);

}

//Tree
void Shape::Draw_Tree(int thickness, int lineType)
{
    //Draw the Rectangle
        Point startingPoint = Point(230,585);
    int wid = 85;
    int heig = 165;
    line_thickness = thickness;
    start = Point(-1,-1);
    end = Point(-1,-1);

    rectangle(*image,
                                        startingPoint,
                                        Point(startingPoint.x+wid, startingPoint.y+heig),
                                        color,
                                        thickness,
                                        lineType);
        //Draw the Circle
        Point middle = Point(272,455);
    int radius = 130;
    line_thickness = thickness;
    start = Point(-1,-1);
    end = Point(-1,-1);

    circle(*image,
        middle,
        radius,
        color,
        thickness,
        lineType);
}

//Door
void Shape::Draw_Door(int thickness, int lineType)
{
    Point startingPoint = Point(722,515);
    int wid = 139;
    int heig = 224;
    line_thickness = thickness;
    start = Point(-1,-1);
    end = Point(-1,-1);

    rectangle(*image,
                                        startingPoint,
```

```
                                                    Point(startingPoint.x+wid, startingPoint.y+heig),
                                                    color,
                                                    thickness,
                                                    lineType);
}

void Shape::Draw_Sun(int thickness, int lineType)
{
        Point middle = Point(1316,193);
   int radius = 115;
   line_thickness = thickness;
   start = Point(-1,-1);
   end = Point(-1,-1);

   circle(*image,
        middle,
        radius,
        color,
        thickness,
        lineType);
}
```

SOUNDEFFECTPLAYER.H

```
#pragma once

#include <dshow.h>
#include <string>

class SoundEffectPlayer
{
public:
        SoundEffectPlayer(void);
        ~SoundEffectPlayer(void);
        int play_Sound(std::string filename);
        int playRandomPiecePlacedSound();
        std::string getSound(int sound_num);
private:
        IGraphBuilder *pGraph;
   IMediaControl *pControl;
   IMediaEvent   *pEvent;
        HRESULT hr;
        static const int NUM_SOUNDS = 7; //Number of piece placed sounds
        std::string sound_piece_placed1;
        std::string sound_piece_placed2;
        std::string sound_piece_placed3;
        std::string sound_piece_placed4;
        std::string sound_piece_placed5;
        std::string sound_piece_placed6;
        std::string sound_piece_placed7;
};
```

SOUNDEFFECTPLAYER.CPP

```
#include "stdafx.h"
#include "SoundEffectPlayer.h"
#include <cstdlib>
#include <time.h>
```

```
#include <msclr\marshal_cppstd.h> //to convert managed string to std::string

using namespace msclr::interop;

SoundEffectPlayer::SoundEffectPlayer(void)
{
   // Initialize the COM library.
   hr = CoInitialize(NULL);
   if (FAILED(hr))
   {
      printf("ERROR - Could not initialize COM library");
   }
            // seed random number generator used to play random sounds
            srand(time(NULL));

            sound_piece_placed1 = "guitar1.mp3";
            sound_piece_placed2 = "guitar2.mp3";
            sound_piece_placed3 = "guitar3.mp3";
            sound_piece_placed4 = "guitar4.mp3";
            sound_piece_placed5 = "guitar5.mp3";
            sound_piece_placed6 = "guitar6.mp3";
            sound_piece_placed7 = "guitar7.mp3";
}


SoundEffectPlayer::~SoundEffectPlayer(void)
{
            pControl->Release();
   pEvent->Release();
   pGraph->Release();
   CoUninitialize();
}

//-----------------------------------------------------------------------------------------------------------
// Plays the audio file 'filename.' The file must be in Sounds directory which is two levels up from execution directory.
int SoundEffectPlayer::play_Sound(std::string filename)
{
            System::String^ soundfile = Constants::SOUNDS_DIRECTORY;
            string stdsoundfile = marshal_as<std::string>(soundfile);
            stdsoundfile += filename;

            // Conver string to wide-character string
            std::wstring stemp = std::wstring(stdsoundfile.begin(), stdsoundfile.end());
            LPCWSTR sw = stemp.c_str();

   pGraph = NULL;
   pControl = NULL;
   pEvent = NULL;

   // Create the filter graph manager and query for interfaces.
   hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
                  IID_IGraphBuilder, (void **)&pGraph);
   if (FAILED(hr))
   {
      printf("ERROR - Could not create the Filter Graph Manager.");
   }

            hr = pGraph->QueryInterface(IID_IMediaControl, (void **)&pControl);
   hr = pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent);

   // Build the graph.
   hr = pGraph->RenderFile(sw, NULL);
```

```
    if (SUCCEEDED(hr))
    {
        // Run the graph.
        hr = pControl->Run();
/*
        if (SUCCEEDED(hr))
        {
            // Wait for completion.
            long evCode;
            pEvent->WaitForCompletion(INFINITE, &evCode);

            // Note: Do not use INFINITE in a real application, because it
            // can block indefinitely.
        }
*/

    }
            return 0;
}

int SoundEffectPlayer::playRandomPiecePlacedSound()
{
            int rand_num = rand() % NUM_SOUNDS + 1; //random number from 1 - NUM_SOUNDS
            switch(rand_num) {
                        case 1: play_Sound(sound_piece_placed1);
                                    break;
                        case 2: play_Sound(sound_piece_placed2);
                                    break;
                        case 3: play_Sound(sound_piece_placed3);
                                    break;
                        case 4: play_Sound(sound_piece_placed4);
                                    break;
                        case 5: play_Sound(sound_piece_placed5);
                                    break;
                        case 6: play_Sound(sound_piece_placed6);
                                    break;
                        case 7: play_Sound(sound_piece_placed7);
                                    break;
                        default:
                                    printf("playRandomPiecePlacedSound(): Sound not available");
                                    return -1;
            }
            return 0;
}

string SoundEffectPlayer::getSound(int sound_num)
{
                        switch(sound_num) {
                        case 1: return sound_piece_placed1;
                                    break;
                        case 2: return sound_piece_placed1;
                                    break;
                        case 3: return sound_piece_placed1;
                                    break;
                        case 4: return sound_piece_placed1;
                                    break;
                        case 5: return sound_piece_placed1;
                                    break;
                        case 6: return sound_piece_placed1;
                                    break;
                        case 7: return sound_piece_placed1;
                                    break;
```

```
		default:
			printf("getSound(): Sound not available.");
			return "";
		}
}
```

FUNCTIONS.H

```
#pragma once

#include "stdafx.h"

#include <opencv2\opencv.hpp>
#include "TrackedPiece.h"
#include "PuzzlePiece.h"
#include "GameBoard.h"
#include "ScoreKeeping.h"

#ifndef GUARD_J
#define GUARD_J

using namespace System;
using namespace System::Collections::Generic;

// Define any constants that will be repeated or that may be changed. EX) int x = Constants::CONSTANT_X
ref class Constants {
public:
	static const int TESTNUMBER = 0;	// chose a test number to run. 0 = not a test

	//OPENCV related defaults ---

	//OpenCV default capture width and height
	static const int DEFAULT_FRAME_WIDTH = 640;
	static const int DEFAULT_FRAME_HEIGHT = 480;

   //max number of objects to be detected in frame
   static const int DEFAULT_MAX_OBJECTS_IN_FRAME =20;
   //minimum and maximum object area
   static const int DEFAULT_MIN_OBJECT_AREA = 2000;

	// HSV default values (0-256)
   static const int DEFAULT_H_MIN = 0;
   static const int DEFAULT_H_MAX = 256;
   static const int DEFAULT_S_MIN = 0;
   static const int DEFAULT_S_MAX = 256;
   static const int DEFAULT_V_MIN = 0;
   static const int DEFAULT_V_MAX= 256;

	// shape drawing stuff
	static const int BACKGROUND_COLOR = 60;
	static const int SHAPE_LINE_WIDTH = 15;

	static const int FLASH_DELAY = 400;
	static const int TIMER_TICK = 350;

	// Hardcoded file paths
	//static System::String^ GAME_INPUT_DIRECTORY = System::Windows::Forms::Application::StartupPath + "/../";
	//static System::String^ HELP_FILE = System::Windows::Forms::Application::StartupPath + "/../Help.txt";
	//static System::String^ CALIBRATION_HELP_FILE = System::Windows::Forms::Application::StartupPath +
"/../CalibrationHelp.txt";
```

```
		//static System::String^ RESULTS_DIRECTORY = System::Windows::Forms::Application::StartupPath +
"/PatientPerformanceData/";
		//static System::String^ SOUNDS_DIRECTORY = System::Windows::Forms::Application::StartupPath + "/../../Sounds"
		static System::String^ GAME_INPUT_DIRECTORY = "C:/PuzzleAssembly";
		static System::String^ HELP_FILE = "C:/PuzzleAssembly" + "/Help.txt";
		static System::String^ CALIBRATION_HELP_FILE = "C:/PuzzleAssembly" + "/CalibrationHelp.txt";
		static System::String^ RESULTS_DIRECTORY = "C:/PuzzleAssembly" + "/PatientPerformanceData/";
		static System::String^ SOUNDS_DIRECTORY = "C:/PuzzleAssembly" + "/Sounds";

};

ref class Globals {
  public:
		static int difficultylevel;
};
//--- FROM FUNCTIONS.CPP----

// Start up a game using RunTracking
GamePlayedData^ initializeTracking(KnobPuzzle^ %Game, System::String^ userName);

// display the puzzle board background
cv::Mat displayPuzzleBoard(cv::Mat matName, vector<TrackedPiece>);

// Unmanaged <--> Managed Conversions
List<int>^ scalarToList(cv::Scalar scalar);
TrackedPiece puzzlePieceToTrackedPiece(PuzzlePiece^ puzzlePiece);
PuzzlePiece^ trackedPieceToPuzzlePiece(TrackedPiece trackedPiece);

// Game code input/puzzle class functions
System::String^ searchPuzzleType(System::String^ code);
array<System::String^>^ getStringArrayFromFile(System::String^ inputFile);
int checkOrCreateFile(System::String^ fileName);
int writeStringArrayToFile(array<System::String^>^ inputArrray, System::String^ fileName);
int appendStringArrayToFile(array<System::String^>^ inputArray, System::String^ fileName);

System::String^ getCalibratedInputPath(System::String^ code);
System::String^ getDefaultInputPath(System::String^ code);

// Timekeeping
int secondsBetweenTwoDateTimes(DateTime startTime, DateTime endTime);
//double getElapsedSeconds(long startTime);

// performance data IO
List<System::String^>^ findRecordFiles(System::String^ player, System::String^ game, array<System::String^>^ days);
GamePlayed^ fileLinesToGamePlayed(array<System::String^>^ fileLines);
System::String^ buildOutputFileName(System::String^ player, System::String^ game, System::String^ month, System::String^ day, System::String^
year);

// Miscellaneous
std::string intToStdString(int number);
System::String^ stdStringToSystemString(std::string str);
std::string systemStringToStdString(System::String^ str);
double averageListOfInts(List<int>^ inputList);

// workaround hack to declare a thread as a global variable in a form
ref class ThreadShell {
public:
		ThreadShell() { Started = false; }
		bool Started;
		System::Threading::Thread^ myThread;
};
```

```
// ---FROM TRACKING.CPP----

void on_trackbar( int, void* );  // this one won't compile as part of RunTracking - no idea why


#endif
```

FUNCTIONS.CPP

```
#include "stdafx.h"

#include "Functions.h"
#include "RunTracking.h"

using namespace System;
using namespace System::Collections::Generic;
using namespace System::Windows::Forms;
using namespace cv;

//-----------------------------------------------------------------------------------------------------------

// Start tracking via a RunTracking instance, and then return the results of the game. Currently only designed for a KnobPuzzle game
GamePlayedData^ initializeTracking(KnobPuzzle^ %Game, System::String^ userName)
{
	GamePlayedData^ gameResults = gcnew GamePlayedData();
	// Initialize OpenCV running class (RunTracking) and load with puzzle
	{
		RunTracking* newTracker = new RunTracking();
		newTracker->setGame(Game);
		newTracker->setPlayer(userName);
		int success = newTracker->Start();

		// if the game had an error, return an empty GamePlayedData
		if (success != 0) {
			Console::WriteLine("Functions::initializeTracking():: the tracker returned an error.");
			delete newTracker;
			return gameResults;
		}

		// Once game is over, pull the game results
		gameResults = newTracker->returnScore()->getGameData();

		// add game results to main scorekeeper instance. Show game results.
		System::String^ results = gameResults->writeOut();
		if (results->Contains("Error")) {
			delete newTracker;
			return gameResults;
		}

		MessageBox::Show(results);

		// see if user wants to save results
		System::Windows::Forms::DialogResult dialogResult = MessageBox::Show("Save game results for " + userName + "?",
"Warning", MessageBoxButtons::YesNo, MessageBoxIcon::Warning);
		// if user says yes, save the settings to the hardcoded location (user doesn't select)
		if(dialogResult == System::Windows::Forms::DialogResult::Yes)
		{
			gameResults->Save();
		}
```

```
                delete newTracker;
        }
        return gameResults;
}
//-----------------------------------------------------------------------------------------------------

// match the game code to the type (as a string). Simple matching; game code should always have game type in it
System::String^ searchPuzzleType(System::String^ code)
{
        System::String^ type = "";
        // Here I would search some database/list sort of thing for the type of puzzle. Or it starts the code.  Ex.
        if (code->Contains("KNOBPUZZLE")) { // tehcnically only knobpuzzles work for the current iteration of the system
                type = gcnew System::String("KnobPuzzle");
        }
        if (code->Contains("BLOCKPUZZLE")) {
                type = gcnew System::String("BlockPuzzle");
        }
        if (code->Contains("SNAKE")) {
                type = gcnew System::String("Snake");
        }
        return type;
}


//-----------------------------------------------------------------------------------------------------
// chunk together calibrated input path or default input path
System::String^ getCalibratedInputPath(System::String^ code) {
                                        System::String^ str = Constants::GAME_INPUT_DIRECTORY + code + ".txt";
                                        return str; }
System::String^ getDefaultInputPath(System::String^ code) {
                                        System::String^ str = Constants::GAME_INPUT_DIRECTORY + code + "_Default" + ".txt";
                                        return str; }

//-----------------------------------------------------------------------------------------------------
// Pull all strings from a file into an array of System::Strings^
array<System::String^>^ getStringArrayFromFile(System::String^ inputFile) {

        array<System::String^>^ lines;
        if (System::IO::File::Exists(inputFile)) { Console::WriteLine("Functions: getStringArrayFromFile(): Found File: \n" + inputFile); }
        else {
                lines[0] = gcnew System::String("ERROR");
                return lines;
        }
        // Read in all lines of file into an array 'lines'
        try {
                lines = System::IO::File::ReadAllLines(inputFile);
                Console::WriteLine("getStringArrayFromFile(): Reading in input file \n" + inputFile);
        }
        // return error if there's a problem
        catch (System::Exception^ e) {
                Console::WriteLine("getStringArrayFromFile(): Error reading input file: \n" + inputFile);
                System::Diagnostics::Debug::WriteLine(e);
                Console::WriteLine(e);
                lines = gcnew array<System::String^>(1);
                lines[0] = gcnew System::String("ERROR");
                return lines;
        }
        return lines;

}
//-----------------------------------------------------------------------------------------------------
// quick function to see if a file already exists. If it doesn't, try to create it.
```

```
int checkOrCreateFile(System::String^ fileName) {
	// if file doesn't exist yet, create it
	if (!System::IO::File::Exists(fileName)) {
		try {
			System::IO::FileStream^ fs = System::IO::File::Create(fileName);
			fs->Close();
		}

		catch (System::Exception^ e) {
			Console::WriteLine("checkOrCreateFile(): Error creating file " + fileName);
			return -1;
		}
	}
	return 0;
}
//-----------------------------------------------------------------------------------------------
// append an array of strings to a file
int appendStringArrayToFile(array<System::String^>^ inputArray, System::String^ fileName) {

	// make sure file is created
	if (checkOrCreateFile(fileName) != 0) {
		Console::WriteLine("writeStringArrayToFile(): file was not created : " + fileName);
		return -1;
	}

   // Add all new text onto end of file
   try
   {
      System::IO::File::AppendAllLines(fileName, inputArray);
   }
	catch (System::Exception^ e)
	{
		System::Diagnostics::Debug::WriteLine(e);
		Console::WriteLine("writeStringArrayToFile(): Error - can't write lines to file:");
		Console::WriteLine(e);
		return -1;
	}

	return 0;
}
//-----------------------------------------------------------------------------------------------
// write an array of strings to a given file
int writeStringArrayToFile(array<System::String^>^ inputArray, System::String^ fileName) {

	// make sure file is created
	if (checkOrCreateFile(fileName) != 0) {
		Console::WriteLine("writeStringArrayToFile(): file was not created : " + fileName);
		return -1;
	}

   // Write all lines to file
   try
   {
      System::IO::File::WriteAllLines(fileName, inputArray);
   }
	catch (System::Exception^ e)
	{
		System::Diagnostics::Debug::WriteLine(e);
		Console::WriteLine("writeStringArrayToFile(): Error - can't write lines to file:");
		Console::WriteLine(e);
		return -1;
	}
```

```
		return 0;
}

//-----------------------------------------------------------------------------------------------------------
// convert an integer into a std::string
std::string intToStdString(int number){
		std::stringstream ss;
		ss << number;
		return ss.str();
}
//-----------------------------------------------------------------------------------------------------------

// convert std::string to System::String^
System::String^ stdStringToSystemString(std::string str) {
		System::String^ MyString = gcnew System::String(str.c_str());
		return MyString;
}
//-----------------------------------------------------------------------------------------------------------

// convert System::String^ to std::string
std::string systemStringToStdString(System::String^ str)
{
		if (str->Equals("")) {
			return "";
		}
    using System::Runtime::InteropServices::Marshal;
    System::IntPtr pointer = Marshal::StringToHGlobalAnsi(str);
    char* charPointer = reinterpret_cast<char*>(pointer.ToPointer());
    std::string returnString(charPointer, str->Length);
    Marshal::FreeHGlobal(pointer);
    return returnString;
}

//-----------------------------------------------------------------------------------------------------------

// take average of a list of ints. If no integers given, will return 0
double averageListOfInts(List<int>^ inputList) {
		double sum = 0;
		for each (int num in inputList) {
			sum += num;
		}
		if (inputList->Count != 0) {
			double average = sum/inputList->Count;
			return average;
		}
		else { return 0; }
}

//-----------------------------------------------------------------------------------------------------------
// convert a cv::scalar into a list of 3 ints (for use in managed code)
List<int>^ scalarToList(cv::Scalar scalar) {
		List<int>^ myList = gcnew List<int>(0);
		myList->Add(scalar[0]); myList->Add(scalar[1]); myList->Add(scalar[2]);
		return myList;
}
//-----------------------------------------------------------------------------------------------------------

//// get elapsed seconds from a start time based on number of DateTime ticks
//double getElapsedSeconds(long startTime) {
//		DateTime tim = DateTime::Now;
//		long placeTime = tim.Ticks - startTime; // 10,000 ticks in a millisecond, 1000 milliseconds in a second
```

```
//		TimeSpan^ elapsed = gcnew TimeSpan(placeTime);
//		return elapsed->TotalSeconds;
//}

//-----------------------------------------------------------------------------------------------------------

// get elapsed seconds between two DateTimes

int secondsBetweenTwoDateTimes(DateTime startTime, DateTime endTime) {
		TimeSpan span = endTime.Subtract(startTime);
		return span.TotalSeconds;
}

//-----------------------------------------------------------------------------------------------------------
// Convert a managed PuzzlePiece to an unmanaged TrackedPiece
TrackedPiece puzzlePieceToTrackedPiece(PuzzlePiece^ puzzlePiece) {
		// pull name
		System::String^ name = puzzlePiece->getName();
		// Puzzle piece HSV lists go [H, S, V]
		int H_min = puzzlePiece->getHSVmin()[0];
		int H_max = puzzlePiece->getHSVmax()[0];
		int S_min = puzzlePiece->getHSVmin()[1];
		int S_max = puzzlePiece->getHSVmax()[1];
		int V_min = puzzlePiece->getHSVmin()[2];
		int V_max = puzzlePiece->getHSVmax()[2];
		// create new Tracked Piece with these results and return
		TrackedPiece result = TrackedPiece(systemStringToStdString(name), Scalar(H_min, S_min, V_min), Scalar(H_max, S_max,
V_max),puzzlePiece->getXDest(),puzzlePiece->getYDest());

		// set all drawing data
		result.setShapePoint(puzzlePiece->getShapePointX(), puzzlePiece->getShapePointY());
		if (name->Equals("Square") || name->Equals("Rectangle"))
				{ result.setShapeWidth(puzzlePiece->getShapeWidth()); }
		if (name->Equals("Rectangle"))
				{ result.setShapeHeight(puzzlePiece->getShapeHeight()); }
		if (name->Equals("Pentagon") || name->Equals("Triangle"))
				{ result.setShapeLength(puzzlePiece->getShapeLength()); }
		if (name->Equals("Circle"))
				{ result.setShapeRadius(puzzlePiece->getShapeRadius()); }

		return result;
}
//-----------------------------------------------------------------------------------------------------------
// Convert a managed PuzzlePiece to an unmanaged TrackedPiece
PuzzlePiece^ trackedPieceToPuzzlePiece(TrackedPiece trackedPiece) {
		// pull name
		std::string name = trackedPiece.getName();
		// Tracked piece HSV scalars go [H, S, V]
		int H_min = trackedPiece.getHSVmin()[0];
		int H_max = trackedPiece.getHSVmax()[0];
		int S_min = trackedPiece.getHSVmin()[1];
		int S_max = trackedPiece.getHSVmax()[1];
		int V_min = trackedPiece.getHSVmin()[2];
		int V_max = trackedPiece.getHSVmax()[2];
		List<int>^ HSV_min;
		List<int>^ HSV_max;
		// recreate HSV list<int>^s
		HSV_min->Add(H_min); HSV_min->Add(S_min); HSV_min->Add(V_min);
		HSV_max->Add(H_max); HSV_max->Add(S_max); HSV_max->Add(V_max);
		// create new Puzzle Piece with these results and return
		PuzzlePiece^ result = gcnew PuzzlePiece(stdStringToSystemString(name), HSV_min, HSV_max, trackedPiece.getXDest(),
trackedPiece.getYDest());
```

```
		// set all drawing data
		result->setShapePoint(trackedPiece.getShapePointX(), trackedPiece.getShapePointY());
		if (name == "Circle") { result->setShapeRadius(trackedPiece.getShapeRadius()); }
		if (name == "Square" || name == "Rectangle") { result->setShapeWidth(trackedPiece.getShapeWidth()); }
		if (name == "Rectangle") { result->setShapeHeight(trackedPiece.getShapeHeight()); }
		if (name == "Pentagon" || name == "Triangle" ) { result->setShapeLength(trackedPiece.getShapeLength()); }

		return result;
}
//---------------------------------------------------------------------------------------------------
// build the path to the performance data file for a given player, game and date
System::String^ buildOutputFileName(System::String^ player, System::String^ game, System::String^ month, System::String^ day, System::String^
year) {
		// build path
		System::String^ pathStr = Constants::RESULTS_DIRECTORY + player + "\\";
		// build filename
		System::String^ fileStr = player + "_" + game + "_" + year + "_" + month + "_" + day + ".txt";
		System::String^ mainString = pathStr + fileStr;
		return mainString;
}

//---------------------------------------------------------------------------------------------------
// find all files matching the given player, game, and date
List<System::String^>^ findRecordFiles(System::String^ player, System::String^ game, array<System::String^>^ days) {

		// find player's results directory
		List<System::String^>^ results = gcnew List<System::String^>();
		System::String^ dirPath = Constants::RESULTS_DIRECTORY + player;
		if (!System::IO::Directory::Exists(dirPath)) {
			Console::WriteLine("Functions::findRecordFiles():: could not find directory for " + player);
			return results;
		}

		System::String^ delimStr = "_";
		array<Char>^ delimiter = delimStr->ToCharArray( );
		System::String^ month; System::String^ da; System::String^ year;

		// construct each file path for each date and check if the file exists
		for each (System::String^ day in days) {
			array<System::String^>^ tokens = day->Split(); // break up date string
			if (tokens->Length < 3) {
				Console::WriteLine("Functions::findRecordFiles():: date incorrectly formatted : " + day);
				continue; }  // if there aren't 3 parts (month day year) to date, continue
			month = tokens[0];
			da = tokens[1];
			year = tokens[2];

			// reconstruct file path
			System::String^ finalPath = buildOutputFileName(player, game, month, da, year);
				Console::WriteLine("Functions::findRecordFiles():: looking for file " + finalPath);
			// check if it exists
			if (System::IO::File::Exists(finalPath)) {   // check if it exists
				Console::WriteLine("Functions::findRecordFiles():: found file " + finalPath);
				results->Add(finalPath);
			}
		}
		return results;
}
//---------------------------------------------------------------------------------------------------
// Parse given file lines into a GamePlayed^ instance
GamePlayed^ fileLinesToGamePlayed(array<System::String^>^ fileLines) {
```

```
	GamePlayed^ result = gcnew GamePlayed();

		System::String^ line = fileLines[0];
		System::String^ gameName = "";
		System::String^ playerName = "";
		System::String^ timeForCompletion;
		System::String^ averageTimeForPieces;
		System::String^ month;
		System::String^ year;
		System::String^ day;
		System::String^ tim;
		List<System::String^>^ pieceNames = gcnew List<System::String^>();
		List<System::String^>^ timesToPlace = gcnew List<System::String^>();
		List<System::String^>^ timesOfPlacement = gcnew List<System::String^>();

		int index = 0;
		while(index < fileLines->Length) {
			line = fileLines[index++];
			Console::WriteLine(line);
			if (line->Contains("Game:")) {
				array<System::String^>^ tokens = line->Split();
				gameName = tokens[-1];
			}
			if (line->Contains("Player:") ) {
				array<System::String^>^ tokens = line->Split();
				playerName = tokens[-1];
			}
			if (line->Contains("Time for Completion (s):")) {
				array<System::String^>^ tokens = line->Split();
				timeForCompletion = tokens[-1];
			}
			if (line->Contains("Average Time")) {
				array<System::String^>^ tokens = line->Split();
				averageTimeForPieces = tokens[-1];
			}
			if (line->Contains("Time Started:")) {
				array<System::String^>^ tokens = line->Split();
				tim = tokens[-1];
				year = tokens[-2];
				day = tokens[-3];
				month = tokens[-4];
			}
			if (line->Contains("Piece")) {
				// pull piece name
				array<System::String^>^ tokens = line->Split();
				pieceNames->Add(tokens[-1]);
				// move ot next line and pull time of placement
				line = fileLines[index++];
				tokens = line->Split();
				timesOfPlacement->Add(tokens[-1]);
				// move to next line and pull time to place
				line = fileLines[index++];
				tokens = line->Split();
				timesToPlace->Add(tokens[-1]);
			}
		}
		return result;
}

// construct and display the puzzle board background using the vector of TrackedPieces
cv::Mat displayPuzzleBoard(cv::Mat matName, vector<TrackedPiece> pieces) {
```

```
	Shape shapes;
	shapes.setImage(&matName);
	shapes.Clear_To_Black(); // Must clear to black first, otherwise get exception
	shapes.Clear_To_Gray();	// then turn it all gray
	// now draw out each piece
	for (unsigned int i = 0; i < pieces.size(); i++)
	{
		shapes.Draw_Shape(pieces[i], 1);
	}
	return matName;
}
```