ARO Project 2021

Author: Andrew O'Kins
Date: 08/10/2021

Table of Contents

Introduction

The ARO Project is a program to find a sufficiently optimal series of images for spatial light modulators (SLMs) to generate a high intensity signal image from a camera. The program offers three algorithms to achieve this; sequential (brute-force), simple genetic algorithm, and micro genetic algorithm. Through the graphical user interface (GUI), a user can configure various parameters for the camera, SLMs, optimization process (how many generations, bin size for images, etc.), and various outputs made. The program also uses multithreading to increase efficiency towards finding results and enable the ability to abort optimization. There are two alternative builds of the program for using two different camera SDKs, one being Spinnaker and the other PICam. Both are developed and built within one Visual Studio project entitled “ARO_Project”.

The purpose of this document is to be a companion piece to the code implementation and elaborate on the project and its components as it currently is, to help in understanding and assist in further developments by providing a high-level look at the implementation and providing details that may be easily missed or misunderstood when looking over the code alone. The goal is to have this one document encapsulate everything to a degree to also help reduce the strain of searching through various files or being uncertain if a document is “up to date” with the others. Other areas of documentation may still be beneficial, but shouldn’t be required additional readings for what the project is at the time of this document’s writing. However it will assume that you have access to the code and that you are dedicating time to look over it as well.

Section 1: Libraries & Frameworks

Besides various standard libraries provided by C++ (string, algorithm, thread, etc.), there are other vital tools used in the project's implementation. Packages are used for handling both the graphical user interface (GUI), make tasks such as image processing or display easier, and to interface with the hardware being used (a camera and SLMs). The following is a brief overview of what is used to help introduce the project implementation.

For the GUI, the Microsoft Foundation Class (MFC) framework is used and is where a user will be able to configure then run/stop the optimization process. This is done with a group of dialogs within a main dialog that help encapsulate the various kinds of settings available and handle the properties/events that occur within. In the GUI presentation, these subdialogs are contained within a window that can be changed with selecting tabs The main dialog is where the control for selecting, starting, and stopping the optimization process can be found along with a few additional options such as save/load settings and enabling the usage of multithreading.

For interfacing with the spatial light modulators (SLMs), the Blink_PCIe SDK is used. To make the interactions more portable and easier to manage, all interactions with this SDK are contained within the SLMController class. An additional class SLM_Board helps in tracking the properties of each board that are found connected at startup. Provided LUT files can be located in C:\BLINK_PCIe\LUT_Files of ASLAnderson machine if not in the project contents itself.

Some tasks involving the managing of images are handled with the usage of OpenCV. This includes the method of saving images and providing separate window displays of the camera and SLMs to the user during optimization. These displays are encapsulated with CameraDisplay, but this library is also used in Optimization and various other parts of the project.

For the camera interactions (configuring and receiving image data) there are two libraries available with the build using one depending on the desired version. The first version is Spinnaker and is the older one used. The other is PICam and is a newer additional option in the project. These two SDKs are not used concurrently, with the implementation having distinct versions of CameraController and ImageController to wrap interactions with the SDKs. To change versions, you must build a separate executable with a change in CameraController.h in whether USE_PICAM and USE_SPINNAKER is defined. Attempting to use both will run into an explicit build error.

In the ASLAnderson machine at ASL, additional resources for Spinnaker can be found in "C:\Program Files\Point Grey Research\Spinnaker2013". For PICam it is in "C:\Program Files\Princeton Instruments\PICam". Examples for PICam implementation can be found in "C:\Users\Public\Documents\Princeton Instruments\Picam\Samples\source code".

For more details on the classes that use these SDKs and others, refer to Section 3: Classes.

Section 2: Program Walkthrough

In this section we will be giving a quick overview over the program's appearance and behavior as given by the user through the GUI. This is to help give an orientation to the layout before going deeper into the implementation. This guide will be going through the Spinnaker version (the only difference to the user may be the options available in Camera Settings).

When starting the program, two windows are created. One is a console window that gives helpful verbose/debug info while running, the other is the GUI window where the user will make interactions to configure, then start/stop the optimization process. Figure 2.1 shows the application windows at startup.

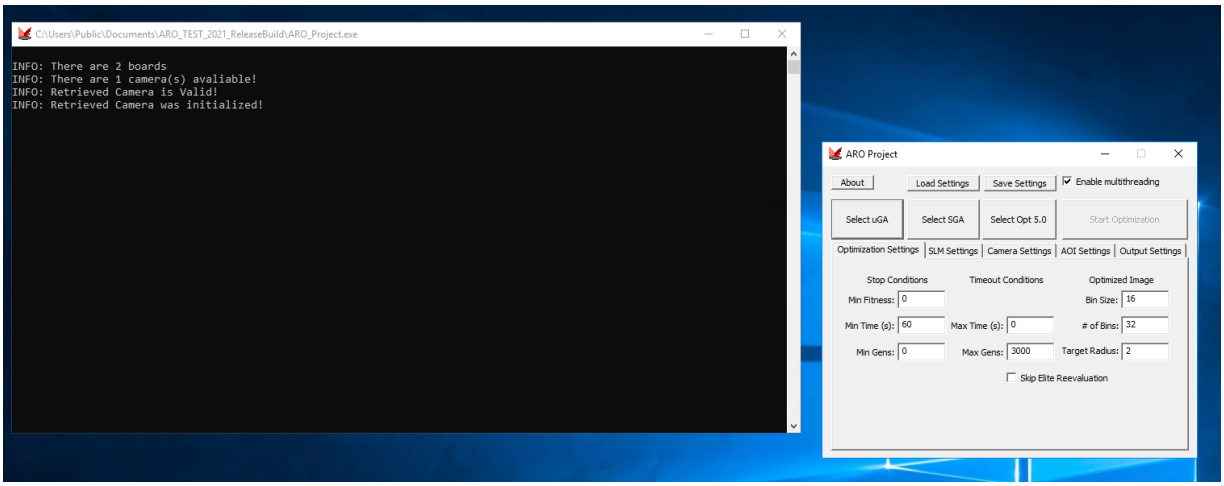

Figure 2.1. Main Windows, Console (left) and main GUI (right)

Since the console window does not provide any input, for the remainder of this guide we will be focusing on the GUI window which Figure 2.2 will provide a more focused view of. Note that the majority of the window contains a tabbed window with 5 tabs for the following categorized settings; Optimization, SLM, Camera, AOI (Area of Interest), and Output. Above this tab window are 7 buttons and 1 checkbox encompassed into 2 rows. The bottom row contains buttons for selecting an optimization and start/stop the selected optimization (initially the start/stop button is disabled as no algorithm has been chosen). The top row has an about button, load/save settings, and the toggle to enable the usage of multiple threads or not for our optimization. Note that the optimization is run on a separate worker thread from the GUI so that the stop button will work.

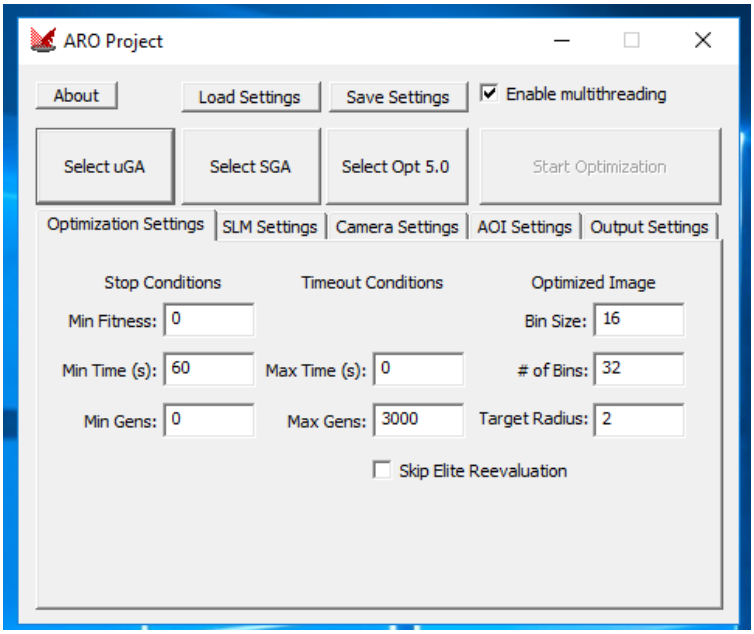

Figure 2.2. The Main GUI Window with Focus on Optimization Settings

The tab window is initially open to Optimization settings, and within it there is access to various fields relating to stop/timeout conditions for the genetic algorithms and fields for the desired optimized image dimensions. Below these fields is a toggle that when enabled will skip elites in a pool from being evaluated (as they already have been) to attempt speeding the optimization process.

Figure 2.3 provides a quick look into the about button. When clicked it produces a new child window that provides some details on the application and an OK button to close (the top-right close button of the window also works). This window must be closed in order to be able to refocus on the main GUI window again.

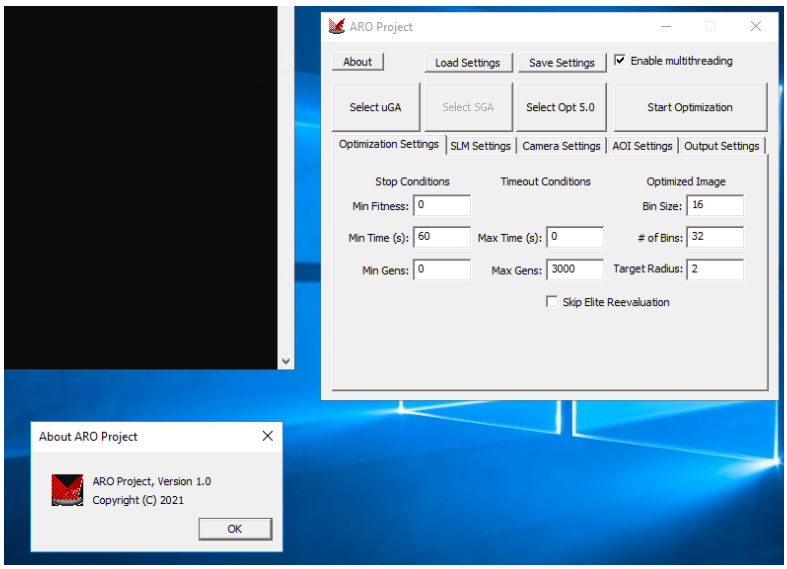

Figure 2.3. About Window Created by Clicking on About Button

The next tab is SLM Settings, within it are the available configurations for the Spatial Light Modulators (SLMs). Initially all the boards are powered off and given a LUT file path as “linear.LUT”. Figure 2.4.1 shows the current layout, note that there is a LUT file path display which is not editable, instead the method of changing LUT file is by clicking on the “Set LUT” button. Figure 2.4.2 shows the window created to select a LUT file, which by default filters out files to only show ones with a LUT extension (though the user can adjust to show all files if needed).

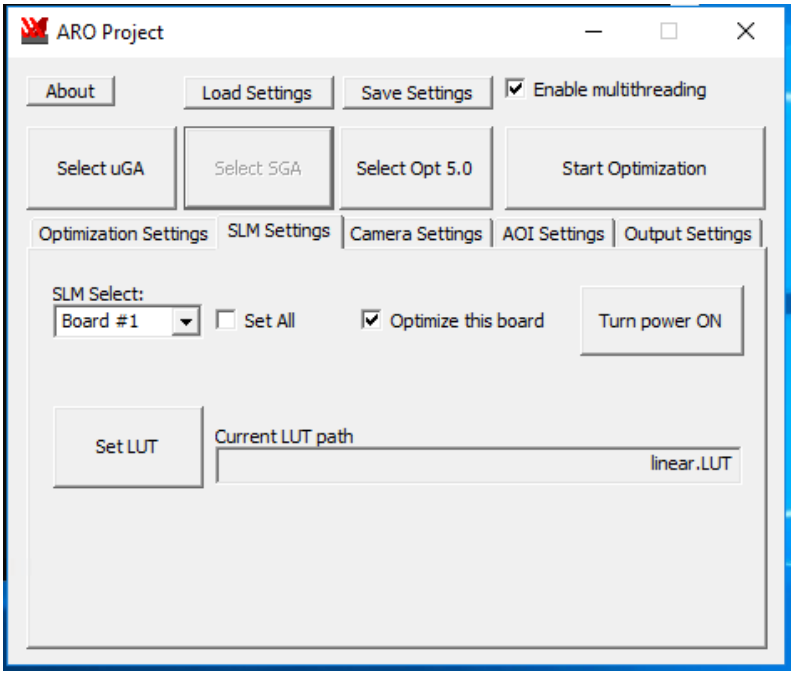

Figure 2.4.1. SLM Settings Window with Default Values

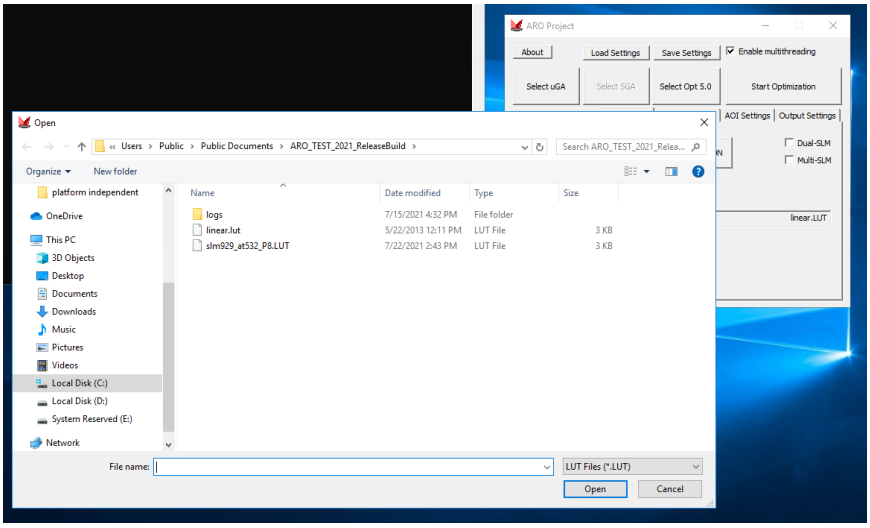

Figure 2.4.2. Selecting a LUT file to Set

In the top left is a SLM Select drop down list, this is updated at startup with the total number of boards connected and is for choosing which board to configure. If “Set All” is toggled, the current selection is ignored and all changes will be applied to all the boards instead. The power button is to power on/off the selected board, and the top-right toggles are for enabling optimization of more than one board and are mutually exclusive (either one is toggled or neither, it will not allow both).

For the currently selected board(s) there is a toggle to include it for the next optimization run. By default, the first board is enabled with the others (if there are any) disabled. The user may set additional boards to optimize, or change the set board by disabling the first board and enabling another one. If all the boards are disabled, the optimization will not run since there are no boards to optimize with.

The following two tabs (Camera and AOI) are the simplest as they feature relatively straightforward fields and buttons relating to configuring the Camera and its area of interest. The Camera settings offers the means of adjusting exposure time, frame rate (which may be separate from exposure and is how fast it acquires images), and gamma. The AOI Setting has fields for setting the area of interet’s dimensions and offset to choose where to get the image data from and two buttons to center the given dimensions into the Camera’s view or set the area of interest to be the entire view dimensions of the Camera.

The final tab is Output Settings and is shown in Figure 2.5. This window is where configurations made to the kinds of outputs made during or after optimization can be selected and where to output them to (the default location being a local logs folder). The toggles are separated into two columns; “Runtime Displays” and “File Output”.

The first column allows enabling/disabling the displays generated to show current best camera and SLM images (shown later when running the optimization). The second column provides selections to enable specific files to output to track performance (or save results) with the first toggle being an enable all option. When this toggle is checked, the following toggles are disabled to help communicate that their toggles would not change anything (as they are all enabled). At the bottom row there is a field to adjust the frequency that saving elites are made, this is only available if the toggle “Save Elites during Optimization” is enabled (the enable all option will also make this available).

The bottom of this window shows the settings for where to store the file outputs. The user may either edit directly in this field or use the “Browse” button to open a folder selection window and set the path through that.

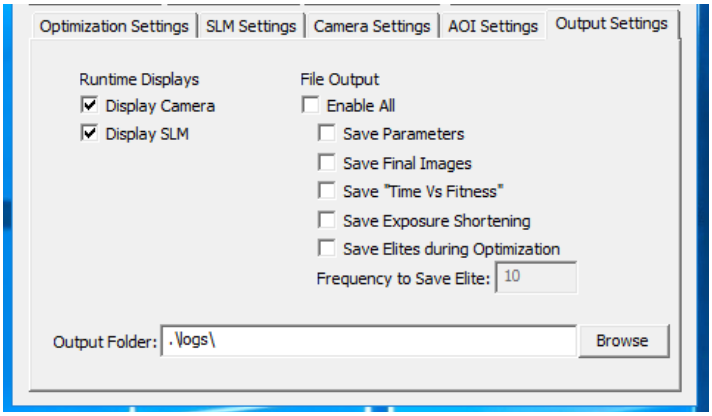

Figure 2.5. Output Settings

The final element of the GUI to show before demoing the optimization behavior is the tooltips feature. If the user wishes to have more details regarding a component, they simply hover over it and a text box below the cursor will appear. Figure 2.6 shows this in action with the “Frequency to Save Elite” field (note that you cannot see the cursor due to the capture software, it is located on the box containing the 10).

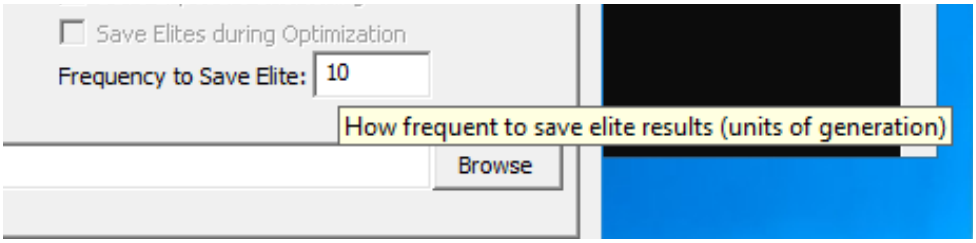

Figure 2.6. Tooltips Display for Multi-SLM with Green Highlight of Area Hovered.

With all major elements of the GUI covered, we will now show the user side experience of running an optimization algorithm. Figure 2.7 shows an example of what the user would see when performing an optimization (in this case SGA). The start button in the main GUI is now relabeled “STOP” and when pressed will attempt to safely end the optimization prematurely. The console window outputs information (in this demo I’ve stopped and started the optimization again and this is shown in the console log). With the display camera and display SLM toggled and in single SLM mode (dual and multi are not enabled in SLM settings), two additional windows are created to show the current best. These windows in Figure 2.7 are near the bottom (but can be moved around like the other windows) with the left one being the camera image and the other the SLM. Note that this was a test run where the laser was not active (only demoing the program itself) and so the camera image would be largely black (as there would be no light from the laser). When the optimization is finished, these windows are closed automatically and the stop button is reset to “Start Optimization”. Unlike when starting the program, the start button is still enabled as the previously selected algorithm is remembered as selected.

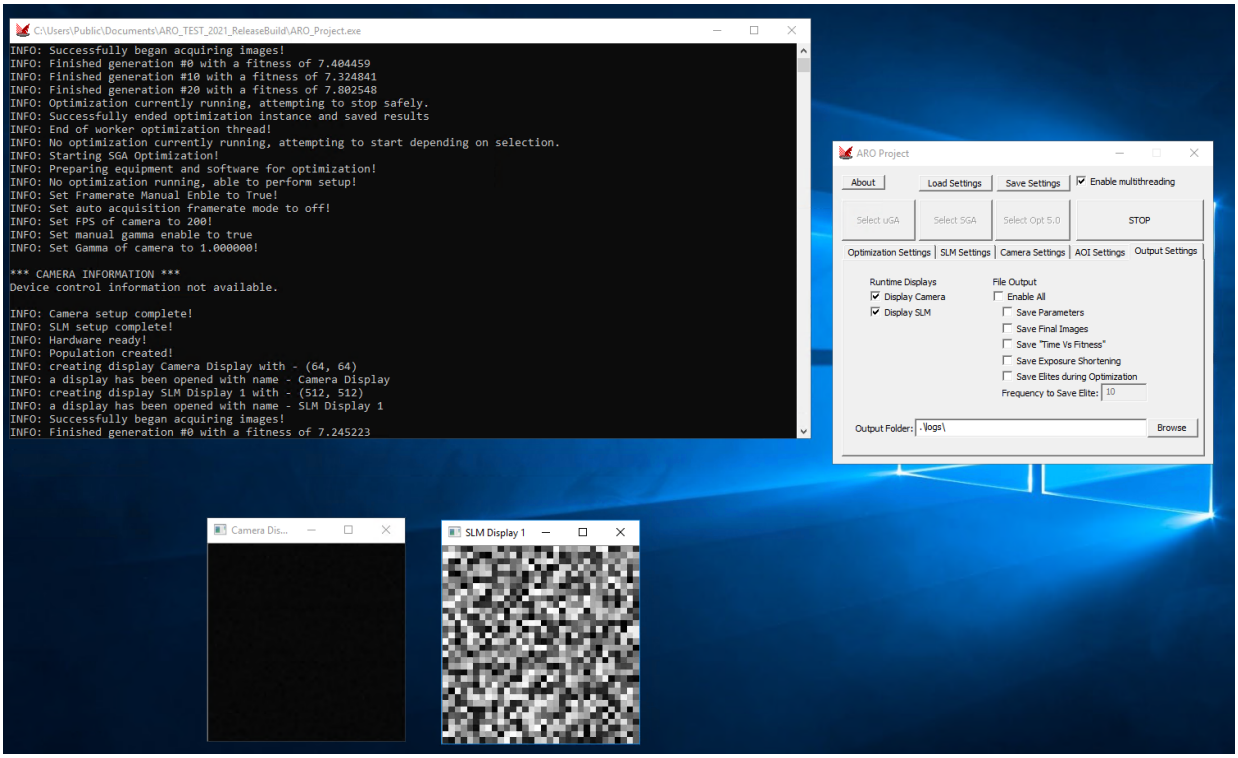

Figure 2.7. Test Demo of Optimization in Process with Newly Created Image Displays

Section 3: Classes

Refer to Figure 1 in Appendix A for a high level diagram that illustrates the relationships of the classes implemented within the project space.

This section elaborates on notable classes that are implemented in this project. Rather than giving details on every property and method, the goal is to provide a contextual explanation to the class' role so that when looking over the code you will have some initial background knowledge on what they are to do.

ARO_App

This class is the entry point for launching the application using MFC and relies on MainDialog. Unless there are some major changes to be made this can be largely left alone as the GUI will be configured through the dialog classes themselves.

Section 3-1: User Interface Dialogs Classes

This section covers the dialog for the GUI classes and their roles. The general structure of them having been constructed from the MFC setup and generation process provided by Visual Studio.

MainDialog

This dialog class is responsible for the main GUI window. Besides a few components it's main purpose is to manage application resources of the program and provide access to the sub dialogs through a tab window.

The most prominent element available in the MainDialog (besides the previously mentioned tab window) is the ability to select and start/stop the optimization process. The MainDialog is also where the option to save/load settings is available (this feature uses file i/o with preference to .cfg file extension and an order dependent and syntax specific file reading process that is implemented in the distinct file OutputSettings.cpp) and the option to enable multithreading or not. The Optimization class has a pointer to this dialog as a means of accessing the entire GUI to draw current values from and determine if the stop button has been pressed or not.

AOIControlDialog & CameraControlDialog

These two sub dialogs are what manages various configurations for the camera being used. AOIControlDialog is specific to setting the area of interest of the camera image to draw data from (rather than taking the entire view the camera has) with additional options to make setting AOI easier (centering and getting full image size buttons). CameraControlDialog provides the other options available to configuring the camera, currently that is the framerate, initial exposure time (separate from framerate), and gamma value. These values are used by the CameraController class and are accessed by it when starting the optimization process.

IMPORTANT: The CameraControlDialog's frame rate field is also accessed by SLMController to set the SLM framerate and attempt to match the camera's.

OptimizationControlDialog

This subdialog is where the configuration options for optimization parameters are available. The stop and timeout conditions are ignored by the BruteForce algorithm. The bin dimensions though are used by all 3 algorithms.

SLMControlDialog

This subdialog is where the user can configure the SLMs. It is also where the option to optimize more than one SLM is available. This dialog also provides the behavior in attempting to write a LUT file with a GUI-based response to if an error occurs to attempt a retry.

A notable element is the LUT path display which is updated to show the current board's LUT path, this component is ignored by the SLMController as it is assumed the SLM_Board's property is the most accurate (though they should be matching). To make this more clear, the field is disabled from user input.

OutputControlDialog

This subdialog manages the toggle of various outputs that the user may want to have made while the optimization process is running.

A note about the output is that all files when optimization is done should be named with a timestamp that should prevent the risk of overwriting multiple runs in the same output folder. Also the enable all option does not clear the values of the sub-toggles made within (just disabled from access).

Section 3-2: Utility Classes

These classes are used in the program as helpful tools. Note that in older documentation there was also a "Utility" class that contained various helpful static functions. This "Utility class" has been remade into a namespace containing the same functions.

ImageScaler

A class to contain the properties and scaling method to translate an image of one size to another. This is used to translate a desired image with a given resolution to one that will properly fit an SLM board.

CameraDisplay

A simple container to help in displaying a given image to a distinct window using the OpenCV library. You can set the dimensions of the window and update with a raw pointer to the data that is then to update the display. Although the name is "CameraDisplay", this class is

used to display both Camera and SLM images (as this class only interacts with image data and not the hardware).

TimeStampGenerator

This class when constructed gives how much time has elapsed since then. This is used by the optimization process to both compare against a stop condition and also to output time elapsed in performance outputs.

BetterRandom

This project's randomizer (instead of standard rand()) as a more evenly distributed randomizer. Used most notably in the genetic algorithms to generate random individuals and perform crossovers.

Section 3-3: Equipment Classes

These classes are what are responsible for accessing and managing the hardware equipment.

SLMController

The wrapper for interacting with the Blink_PCIe SDK for controlling the Spatial Light Modulators (SLMs). Wherever interactions with the SLMs are required this should be made accessible rather than interacting with Blink to increase portability of the project in case a different SDK is desired to be used in the future.

SLM_Board

Simple class to encapsulate tracking data for the state of the SLM boards connected. Includes (but not limited to) image dimensions of the board, a string for the current assigned LUT file, and the current power state (which isn't available from Blink directly, rather is updated whenever the power setting is updated in SLMController). SLMController holds a vector of pointers to these boards. During optimization, the Optimization class holds a vector of pointers to the boards that are set to be optimized that are taken from SLMController. It is important to not delete the boards after optimization as SLMController still refers to them (they are not a deep copy). The board_id property is the board index according to Blink and is important when optimizing to track the board when in the case that you don't have a continuous selection setup from board 1 (example being only optimizing board 2).

CameraController

The wrapper for interacting with the chosen camera SDK (Spinnaker or PICam). Regardless of SDK chosen, CameraController provides the means of configuring the camera accordingly and in acquiring the image data that is contained within ImageController.

At the start of an optimization, call *setupCamera()* to connect with the camera and apply current camera settings according to the UI. The *shutdownCamera()* is called to release

resources for the camera and should be called when done with the camera (which is not when done with optimization as it may be started again later in the GUI). The methods *startCamera()* and *stopCamera()* to manage the acquisition from running or not. These are called by the optimizations so that the camera isn't actively acquiring images when not optimizing.

For Spinnaker some hard coded settings are for acquisition with "Continuous Mode" and the buffer to "Newest Only" so that we can acquire the most recent image at a later point.

PICam operates somewhat differently, with different available parameters and setup. Unlike Spinnaker, it does not rely on a nodemap to access and set these parameters but instead a variety of PicamParameter defined enumerated values. The basic/default approach involves finite acquisition, for indefinite it requires setting the readout to 0 calling for wait on acquisition update for new data.

ImageController

This wraps all the interactions with the chosen camera SDK in the context of image data itself (this is most important for Spinnaker as it has its own ImagePtr class that is used). This class is used in the optimization to hold an image data that can be accessed or saved at a later point. The ImageController is also what provides the means of saving images. Like CameraController, there are alternate versions between Spinnaker and PICam.

Section 3-4: Optimization Classes

These classes are specifically related to the optimization algorithms. The child classes are not covered as those are responsible largely for implementing virtual methods which are different depending on the algorithm.

Individual

This class holds the properties and behaviors for a given individual in the genetic algorithms. It has a pointer to a vector that holds the image data to be written to an SLM and is being optimized and a variable to hold the resulting fitness determined by the average intensity within a radius. This fitness is initially set to -1 (something that should be impossible as absolute darkness would be 0) as a means of identifying if this individual has been evaluated or not.

Population

Holds a pool of individuals that represent a generation and provides the means of implementing the genetic steps of producing a new generation. The base Population class is virtual and relies on the implementation to be provided from the two child classes; SGA_Population and uGA_Population.

For the process of generating the next generation for SGA and uGA, refer to Figures 4 and 5 in Appendix A. These figures also note where multithreading is utilized (though multithreading if enabled is also used in the initialization of the first generation as well). These

threads are managed using the C++ standard thread library, and having in the optimization class a vector to hold them.

Optimization

Encapsulates the higher level properties and behaviors for an optimization algorithm along with the setup, output, and release of resources. The base Optimization class has 4 virtual methods that are relied on by the child classes to implement. The implementations of the optimization algorithms are made in child classes for each algorithm available (BruteForce, SGA, and uGA). For the genetic algorithm, the process of generating a pool and the next generation are dependent on the Population's implementation and are included in the associated child classes. For the genetic algorithms there is a population for each board (for example, in Dual-SLM mode there are two populations with 1 for each board separately).

The virtual methods are the following; *setupInstanceVariables*, *shutdownOptimizationInstance*, *runIndividual*, and *runOptimization*. The first two methods are to be responsible for initializing and releasing properties such as the population and current generation count. The method *runIndividual* is implemented by the GAs to perform the process of evaluating an individual in a pool (refer to Figure 2 in Appendix A), while for the BruteForce algorithm the *runIndvidual* is for optimizing a given board. The method runOptimization is a higher level function to encapsulate the total behavior of optimization and uses the previously mentioned methods to perform the algorithm.

For information on the process of optimization for the genetic algorithms, refer to Figures 2 and 3 in Appendix A for *runOptimization* and *runIndividual* respectively. These figures also note where multithreading is utilized during optimization. These threads are managed using the C++ standard thread library, and in the optimization class a vector to hold them for rejoining.

Section 4: Notes For Working on the Project

The goal of this section is to help starting on developing for this project by exploring areas that may not be most intuitive or familiar which may not be easily found through the code's implementation alone.

Section 4-1: Visual Studio Configuration

This project has been built using Visual Studio 2013, and with this a number of configurations have been made. Importantly is the linkage to the mentioned libraries in Section 1 which is also different between Debug and Release builds as the specific files used change from debug version to a release. For this project you will also require Blink_PCIe, OpenCV, and Spinnaker (or PICam) installed to access their libraries. The author of this document [Andrew O'Kins] had the experience of adding PICam and fixing linkage issues but not configuring all the settings, so this section is not a guarantee that everything is configured correctly if starting from scratch (sincere apologies if you are in that situation).

To modify the libraries being linked to, first open the project properties window by going under the Project tab. In Figure 4.1 this is "ARO_Project Properties…".

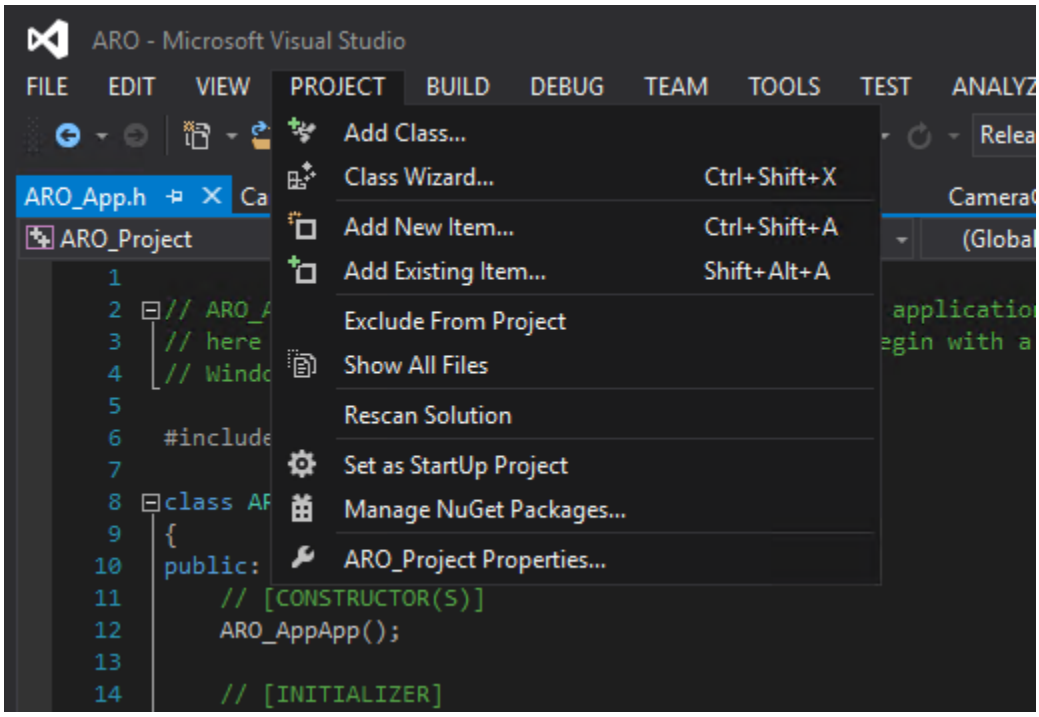

Figure 4.1 Project Properties Drop-Down Menu

The window displayed will provide the means of configuring the build properties for both Debug and Release version. When making changes to libraries, be sure to properly update both versions appropriately. Figure 4.2 shows the window with configuring debug and the VC++

directories properties selected. The highlighted row is where you will want to have included a path to the libraries being used.

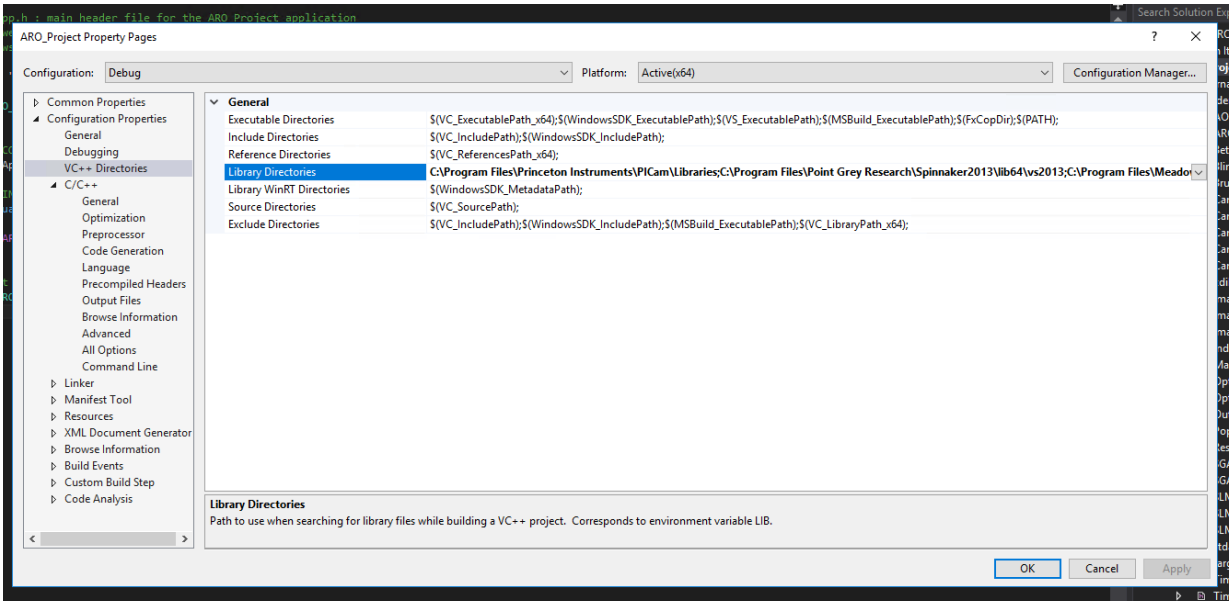

Figure 4.2 Debug Project Properties Focus on Library Directories Setting

There is more that will also need to be adjusted if you want to include new libraries and build. You will want to add the appropriate paths to the Include folder for accessing the files that you want (and will help intellisense know if you have your code right). This is under C/C++ and in General, Figure 4.3 shows this field is the top row.

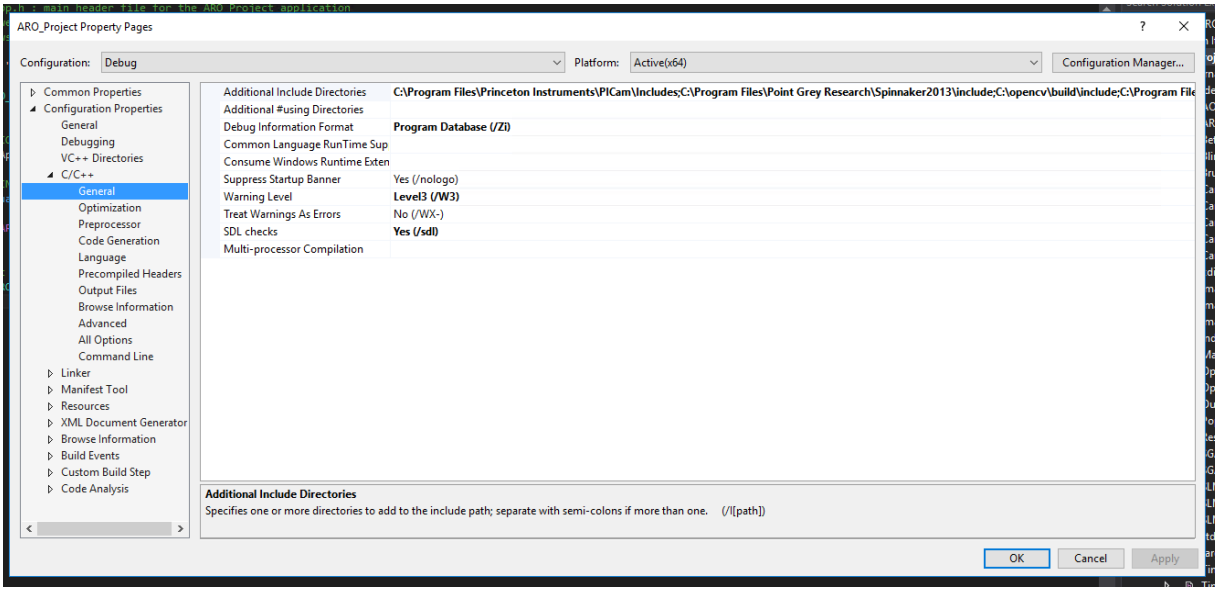

Figure 4.3. Debug Project Properties Focus on Include Directories Setting

It's not over yet! There's still a few places to set things up in. Another location is the linker where you will need to set it so that the linker will connect to the libraries too (so that it will actually compile right). These will have similar paths as the VC++ setting as shown in Figure 4.4.

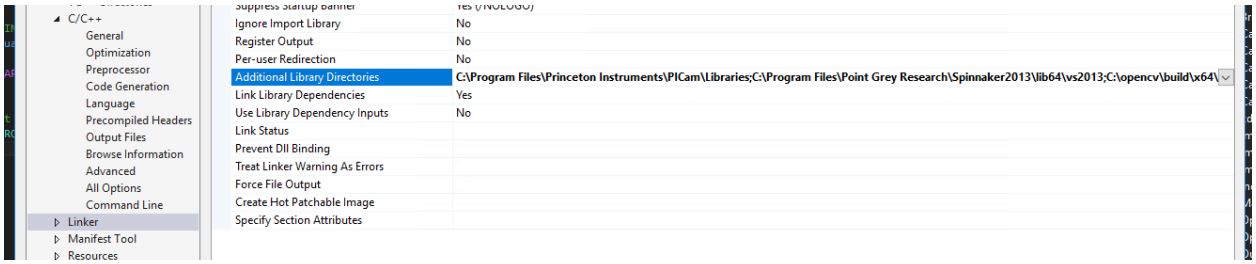

Figure 4.4 Linker Project Properties

The final area is to connect our dependencies to the appropriate .lib files that the executable will be using. This is under Linker -> Input and should be the top row. These aren't file paths but rather the file names. This is where you will want to make sure that the Debug and Release versions are set up correctly as there are library files for either debug or release version and having it misconfigured could lead to errors. Debug library files typically end with a 'd' in the name. Figure 4.5a shows an example of the Debug dependencies setup and Figure 4.5b shows the Release dependencies setup. Note the difference for OpenCV lib file being "opencv_core248.lib" for the Release configuration and "opencv_core248**d**.lib" for the Debug configuration.

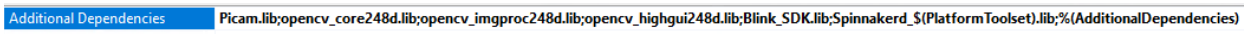

Figure 4.5a Dependencies for the Debug Build

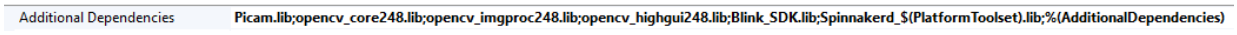

Figure 4.5b Dependencies for the Release Build

If these fields are properly, the libraries should now be properly set up and connected and you should be good to go with using them. You may need to restart Visual Studio to have intellisense work properly.

Section 4-2: Modifying the GUI

Editing GUI Components and Layout

Visual Studio provides a means of modifying the MFC GUI with a streamlined process by using the Resource View. Through the Resource View you can select a dialog to modify and then with simple drag and drop be able to move current elements within the dialog and add/remove elements. Adding/Removing elements is more involved but straightforward. When adding to the resource layout, drag the desired element from the toolbox window to the resource editor and be sure to edit the ID to something unique and identifiable under the Properties

window. Figure 4.6 gives an example of the resource view on one of the sub dialogs. The window to the right is the properties of a selected element. Notice to the left edge a 90-degree rotated button labeled “ToolBox”, click on this to get a list of component types that you can drag-and-drop to then add. To edit the ID, simply click on the ID field in the properties window and type it in (it will be added as a defined value in resource.h). For labels, change the Caption field.

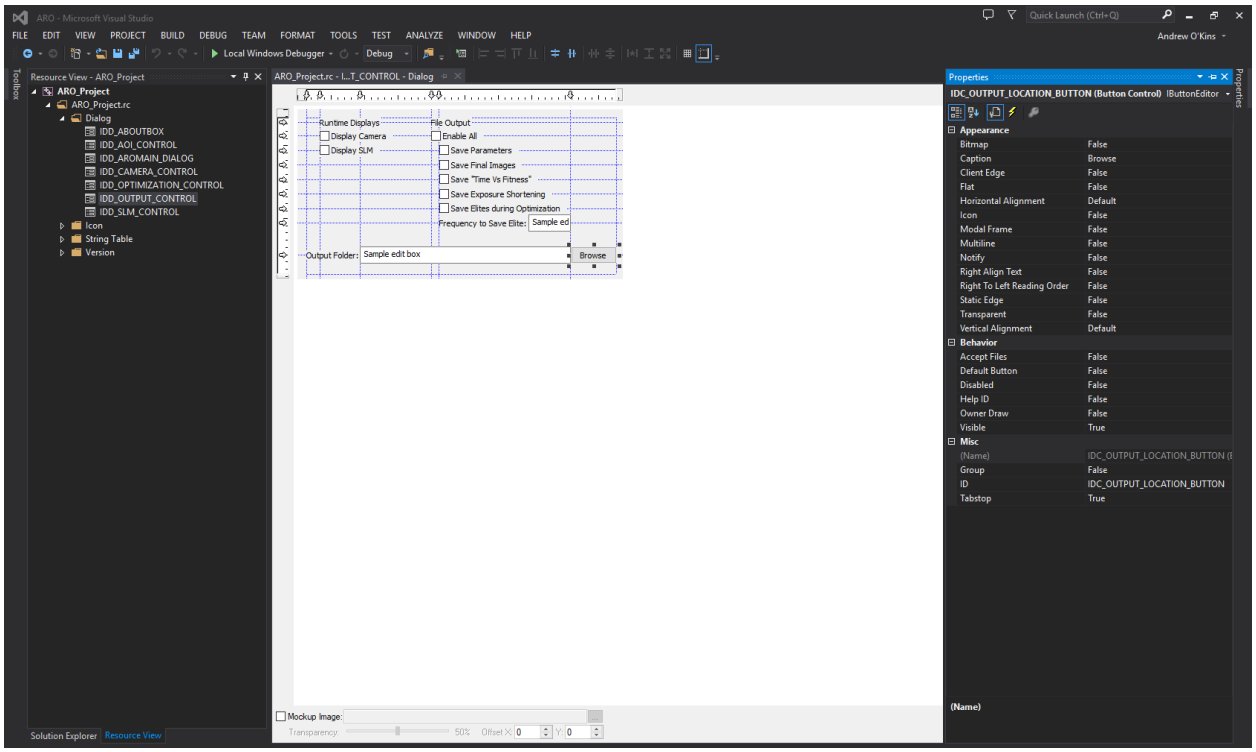

Figure 4.6 Resource View in Visual Studio

When ready to add the element as something that you can interact with in the code (such as a new checkbox), right click on the element to get a dropdown menu on your options which is shown in an example checkbox in Figure 4.7.

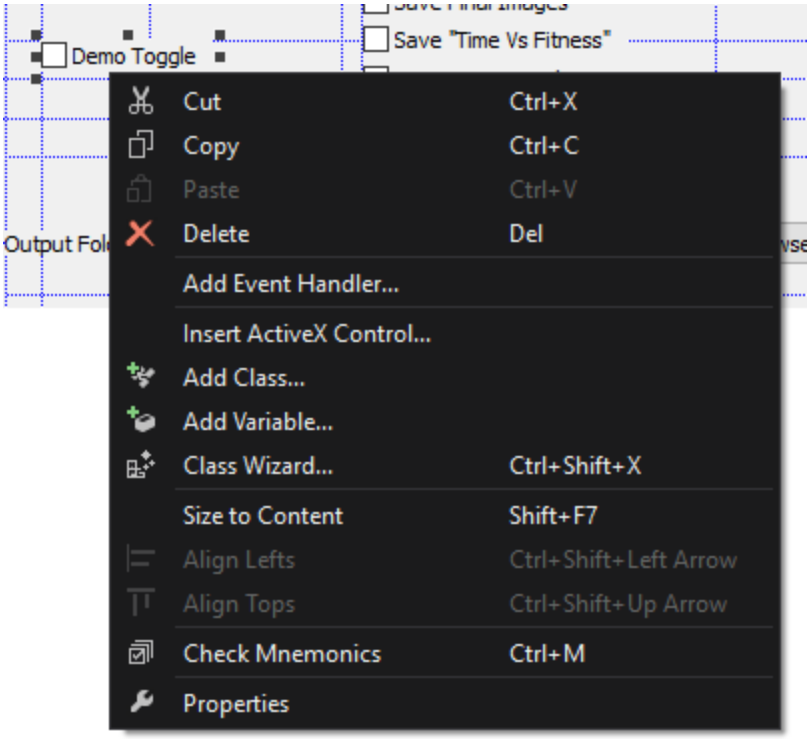

Figure 4.7 Available Options when Right-Clicking on an Element in Resource View

Two notable options to consider are “Add Variable…” and “Add Event Handler…”. If you select the add variable option, a window like in Figure 4.8 will appear and allow you to setup a variable that will be associated with the element you’ve created. Give it a variable name and ideally also a comment, typically Visual Studio will get the type and other parts as you need it so don’t worry too much about them unless you’re looking for something more specific or non-default. When you select Finish, the variable will be automatically added to you dialog and be ready to go. You will just need to give it a default value within *MainDialog::setDefaultUI()* and use it accordingly from then on.

Adding an event handler is useful for if you want to do something when the user does something (usually like when the element is clicked on). Figure 4.9 shows the add event handler window and is similar to the add variable except it generates an empty function for you to write code into that is called when the event occurs. A note to be careful with is that the event handler does not have to be within the same dialog class the associated component is in (or even a dialog class at all for that matter).

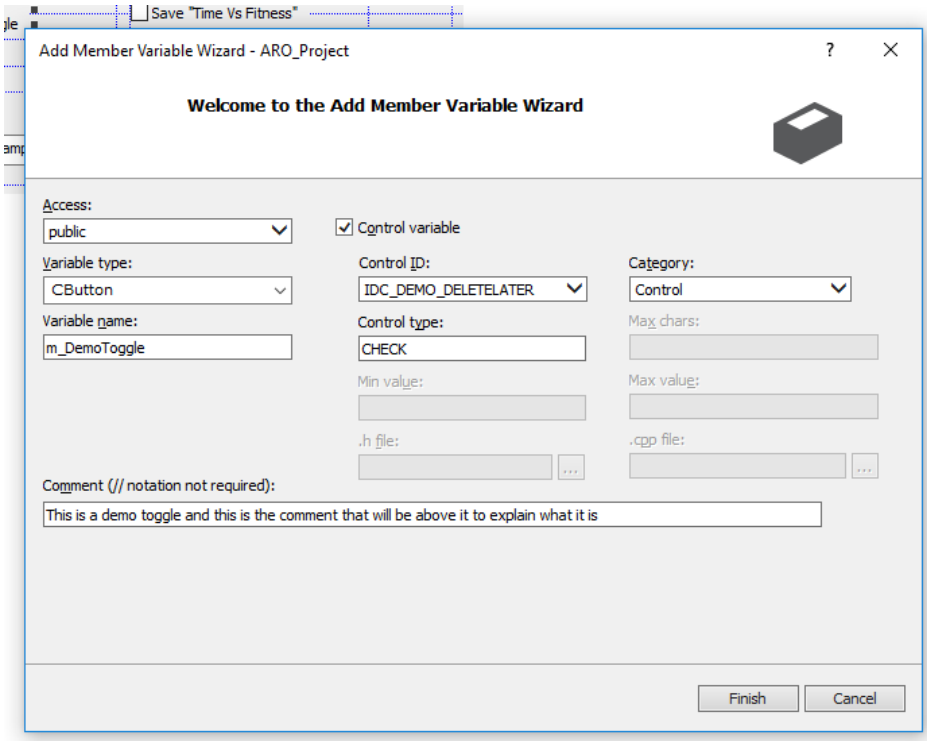

Figure 4.8 Adding Variable Window with Demo Toggle

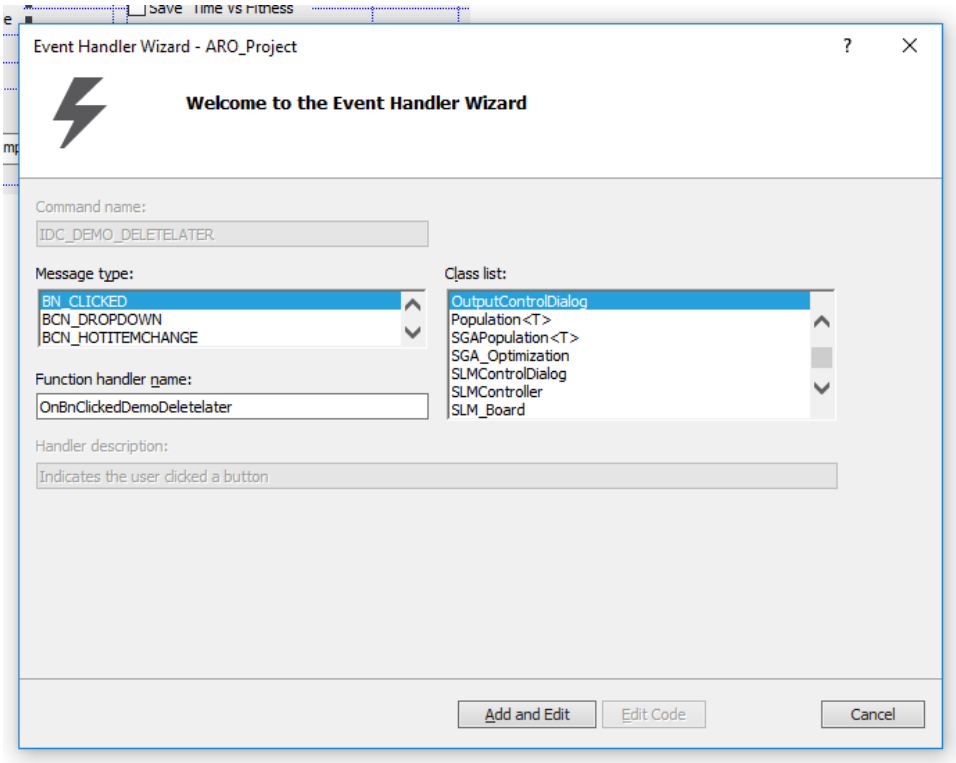

Figure 4.9 Adding Event Handler with Demo Toggle

Removing elements is equally straightforward, with the steps being to delete the component in both resource view and in the code (which may be more time consuming as it won't be auto-removed the same way it can be auto-added to the class). Note: when removing elements from resource view, it will not remove it from the code and vice-versa (unless you went into the resource file and edited it directly).

Whenever adding or removing elements, it is important to consider the save and load feature as you will have to add/remove the variables appropriately. It is also highly recommended to add a tooltip to provide additional details to this component. To do this, go into the implementation file for the dialog that the component is in and look under its *OnInitDialog*. To add the tooltip, use the m_mainToolTips property and call *AddTool* with arguments *GetDlgItem()* (with its argument being the ID of the component) and a c-string that will be what the tooltip says.

Creating a New Tab Window (subdialog class)

In the case of needing to have an entirely new category of settings (or something else in the same tab window), you will need to create a Dialog class that is then added to the tab list. This is a bit more involved but can be done with relative ease.

First you will need to create the resource dialog. Go into Resource View and add a new Dialog (right clicking in the Resource View and selecting "Insert Dialog" will work as shown in Figure 4.10.1 (or if you choose "Add Resource" a window like in Figure 4.10.2 will appear to choose Dialog from) as. This will create a new Dialog that we can use to add new elements too. The first recommended step is to right click on the dialog and select "Properties" to open up the Properties window for it (similar to a component) and set Border to "None" and update the name/ID to something more descriptive (in this example I'm calling it IDD_NEW_TABDIALOG). Should look something like Figure 4.11 now.

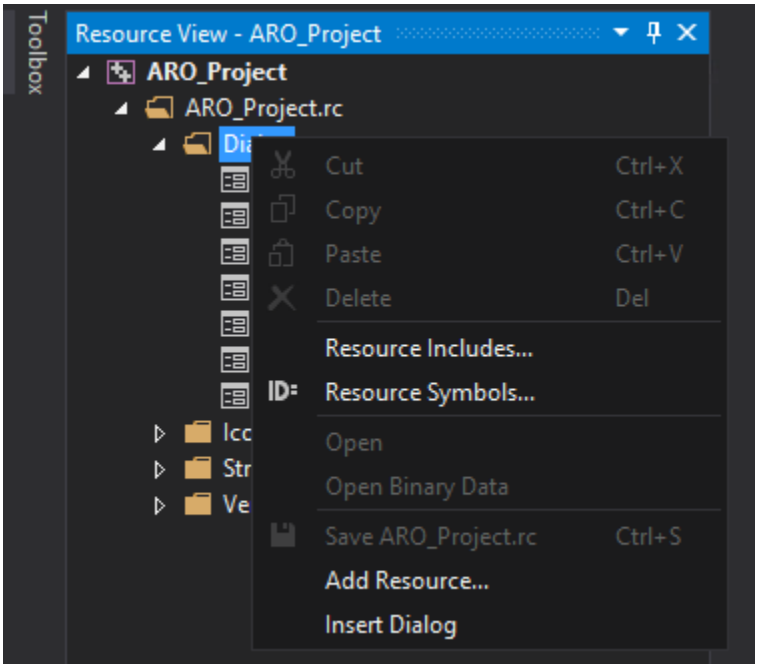

Figure 4.10.1. Inserting New Dialog into Resource View

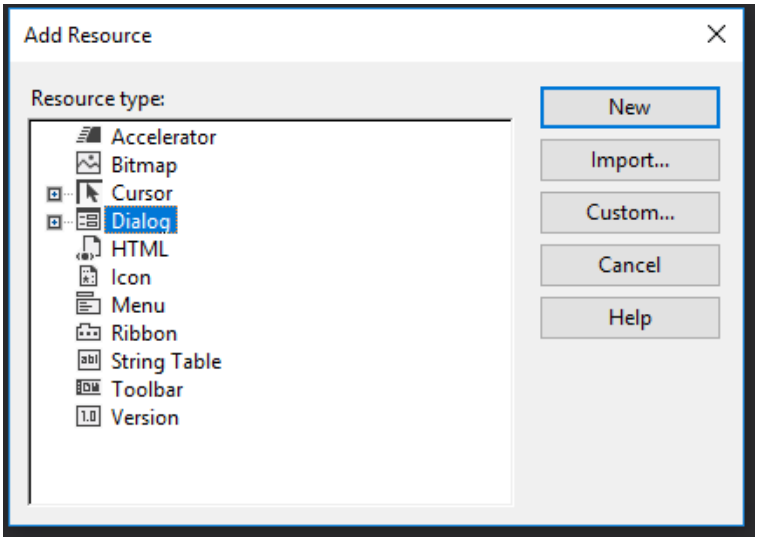

Figure 4.10.2. Add Resource Menu to Insert New Dialog From

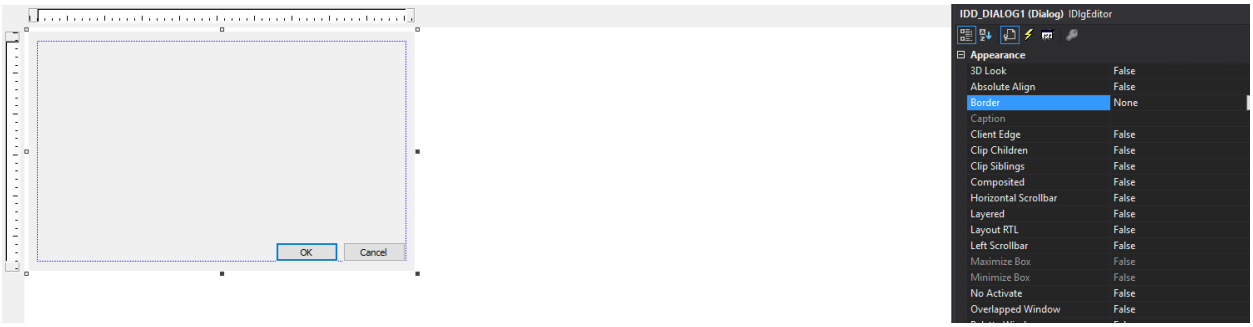

Figure 4.11 New Dialog with No Border

The next step is to now create a class that can use this new dialog window and be contained in the tab window. To do this, right click on the dialog and select “Add Class…”. Figure 4.12 shows the window that will appear, set the name of the class and when finished it will generate the header and implementation files for it automatically for you to work in. However to properly implement the class you will need to add some things. Most importantly you need to add the necessary includes for it to compile and use the resources. Figure 4.13 shows the includes made in the header of the new class and the general appearance of the class, you may need to restart Visual Studio for intellisense to not throw warnings with the new IDD. To include it into the main dialog you will also have to add the new header file into the MainDialog’s includes before adding it into the implementation.

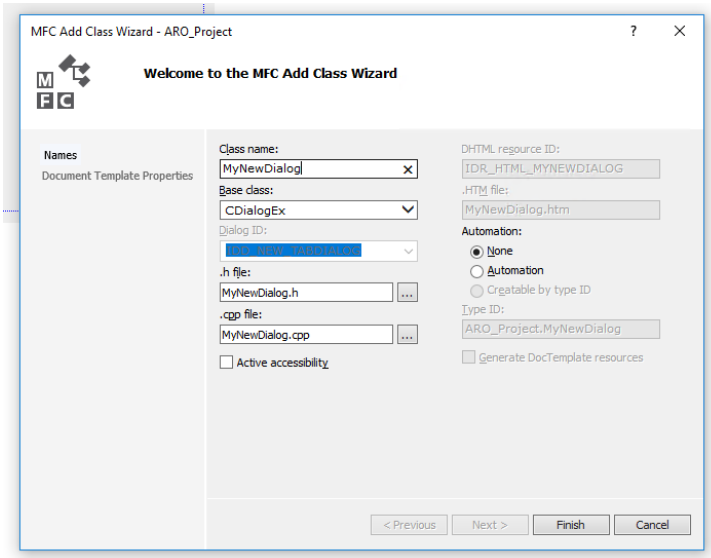

Figure 4.12 Creating a Dialog Class Window

```
#pragma once

#include "resource.h"
#include "afxwin.h"


// MyNewDialog dialog

class MyNewDialog : public CDialogEx
{
    DECLARE_DYNAMIC(MyNewDialog)

public:
    MyNewDialog(CWnd* pParent = NULL);   // standard constructor
    virtual ~MyNewDialog();

// Dialog Data
    enum { IDD = IDD_NEW_TABDIALOG };

protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

    DECLARE_MESSAGE_MAP()
};

```

Figure 4.13 The New Dialog Class

The next step is to incorporate this dialog into the tab window. Add the dialog as a public property of MainDialog’s, then go into MainDialog’s implementation file and into *OnInitDialog*. Add the tab heading into the headings array and increment the for loop to insert it into the m_TabControl property. Followed by creating the window with *Create([ID OF NEW DIALOG], &m_TabControl*) and use the rect variable to define the dimensions. The final critical

step is to add the dialog in *OnTcnSelchangeTab1* which is responsible for handling the tab showing the windows. Simple add a new case number for where the new dialog is and set it to show window and update m_pwndShow (failure to do so will give warning message when trying to run the program and the new window won't work).

For tooltips, this is implemented manually. Go into the header of your new dialog class and add a CToolTipCtrl property, declaration for *virtual BOOL PreTranslateMessage(MSG* pMsg)*, and for *virtual BOOL OnInitDialog()*. Then go into the implementation and add the following:

1. In constructor initialize the CToolTipCtrl property (pointer set it to a new one)
2. Implement the setup for the tooltip controller within *OnInitDialog*
    a. Call your tooltip controller's *Create* with *this* (pointer to your dialog's instance) as parent window.
    b. Add tools to your tooltips, using *GetDlgItem* with arguments of the IDs of your components and text for the tooltip.
    c. Call the tooltip controller's *Activate* with *this*
    d. Return *CDialogEx::OnInitDialog()* so it also does all the default stuff
3. Implement *PreTranslateMessage(MSG* pMsg)*
    a. Call your tooltip controllers *RelayEvent()* with argument being pMsg
    b. Return *Cdialog::PreTranslateMessage(pMsg)* so it does all the default stuff.
4. Properly deallocate the tooltip controller within the dialog's destructor (if pointer call delete).

The dialog should now be added to the program and have tooltips working. For more about adding/changing its components refer to the other subsections. You likely will also need to increase the width of the main dialog window so as to have room for the new tab.

Changing Default Values

Changing the default values (what the contents are in load up) for most of the GUI elements is pretty straightforward. All you have to do is go into the MainDialog's *setDefaultUI()* and set the value to the component there. This method is called within *OnInitDialog* and can be used to set the default values for all the sub-dialogs too.

Considering the Save / Load Feature and New Changes in Elements

The save and load ability is pretty simple, so it is simple to modify but also means that you will need to be careful and make sure it is kept up to date when changing what elements are in the GUI (especially adding new ones as it won't capture them). The save/load behaviors are kept separate in their own SettingsOutput.cpp file to help reduce the size of MainDialog.cpp (the main implementation file for the MainDialog class) for better navigation.

Like mentioned in Editing GUI Components and Layout, when removing elements you will have to be sure to remove references to the variables that no longer exist here. In the case of adding them, you will need to both add references to it in saving settings and loading them.

For loading settings, have the name of your variable (doesn't have to be the same as the code, just consistent) added in the conditions within *MainDialog::setValueByName()* and assign the input value accordingly to the element. I [Andrew O'Kins] recommend keeping the variables within a specific dialog grouped together to help in readability and tracking if there are variables missing or not.

For saving the settings, you will need to add it within the method *MainDialog::saveUItoFile()* following the syntax [variable name]=[value converted to string]. In the case of booleans (such as from checkboxes), I recommend using "true" and "false" for readability in the save file. As long as you keep things consistent within the syntax you should be good to go. If you wanted to add comments within the save file (such as categories), the comment syntax is starting the line with "#".

Section 4-3: Multithreading Management

Multithreading is made possible with two libraries, the standard thread library provided by C++11 and threading tools provided by MFC. The usage of threading in MFC is to launch the optimization and enable the ease of interactions with the GUI during the optimization for important features like the stop button. When the start button is pressed, it launches a thread with the method *optThreadMethod* and the main dialog passed in as the argument. This thread identifies the algorithm to use and runs it accordingly.

For the multithreading in the optimization algorithm, the threads are managed by having a member vector that holds the threads launched to later rejoin them all using the Utility method *rejoinClear().* An important assumption being made is that there is no risk of one of these threads to be running into an indefinite loop (thus locking everything up). This is because the only notable possibly indefinite loop is in the number of generations in the genetic algorithm (as it relies on stop conditions). In the case of this, the stop button (or setting the timeout conditions accordingly beforehand) should serve as an adequate solution.

Although it should not occur, in *runIndividual* for both SGA and uGA there is a check with a boolean usingHardware to make sure that a given individual thread is not starting to use hardware that another individual is currently using. This is done by setting usingHardware to true after the check and then setting it to false before unlocking the hardware mutex. The other locks don't have this additional check due to them being not as critical and this check is more of a debugging/confirmation that the mutex locks work.

This document will not go much further into how multithreading works, as it assumes the reader is either already familiar with parallelism with access to the code and C++ documentation (such as https://www.cplusplus.com/reference/thread/thread/). Otherwise this section would be iterating over trivial implementation and syntax. If the reader is not familiar, it is recommended to look over the C++ thread library and also for mutexes. A quick recap is that mutexes are a class that helps protect critical sections of the code by locking/unlocking when entering and exiting those sections. An important element of this usage is to make sure that you never not unlock after locking, as it would lead to a separate thread becoming stuck as it indefinitely waits for the mutex to unlock.

Section 4-4: Potential Areas of Future Work

This subsection's goal is to present areas of improvement in the project to consider in the perspective of the current (August 2021) author/programmer (Andrew O'Kins). Due to time and priorities these have not yet been addressed.

The small task is to reimplement the sorting algorithm used in the Population class for the current pool. This is currently using the insertion sort algorithm, which is generally not as efficient as something such as quicksort or mergesort. This may not bring significant improvement in the program due to the generally small size of the pool (30 elements at most with SGA), however it may be prudent to go for a better option that is available for future use.

Another area of work is a somewhat vague objective that is to find ways of optimizing the algorithm so that an evaluating individual thread is more lightweight and so faster. This may include attempting to reduce the number of local variables or unnecessary operations where found (the implementation of output settings is one way to attempt improving the efficiency by being able to reduce unneeded file outputs on a given run). One possibility may be finding a way to have the scaling be done outside of the hardware lock so that at least part of this image processing is done in parallel.

It may also be beneficial to do additional refactoring of the code such that there is less duplication between uGA and SGA as outside of the population class and output file names they behave nearly identical. Finding a way to encapsulate more of their behavior together (such as possibly their *runOptimization*) could make future changes that apply to both of them more manageable.

Appendix A: Diagrams

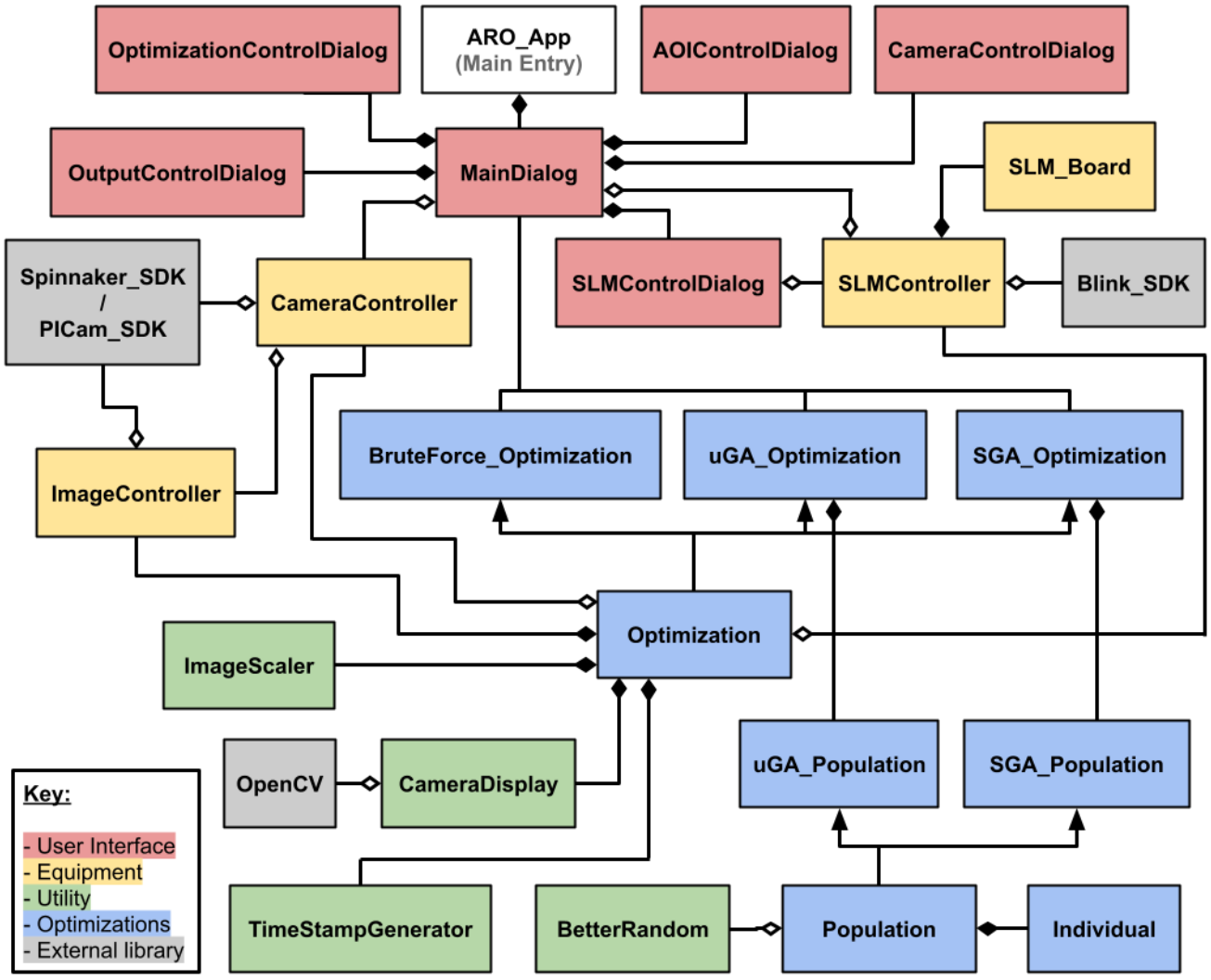

Figure 1. High-Level UML Diagram Illustrating Class Relationships

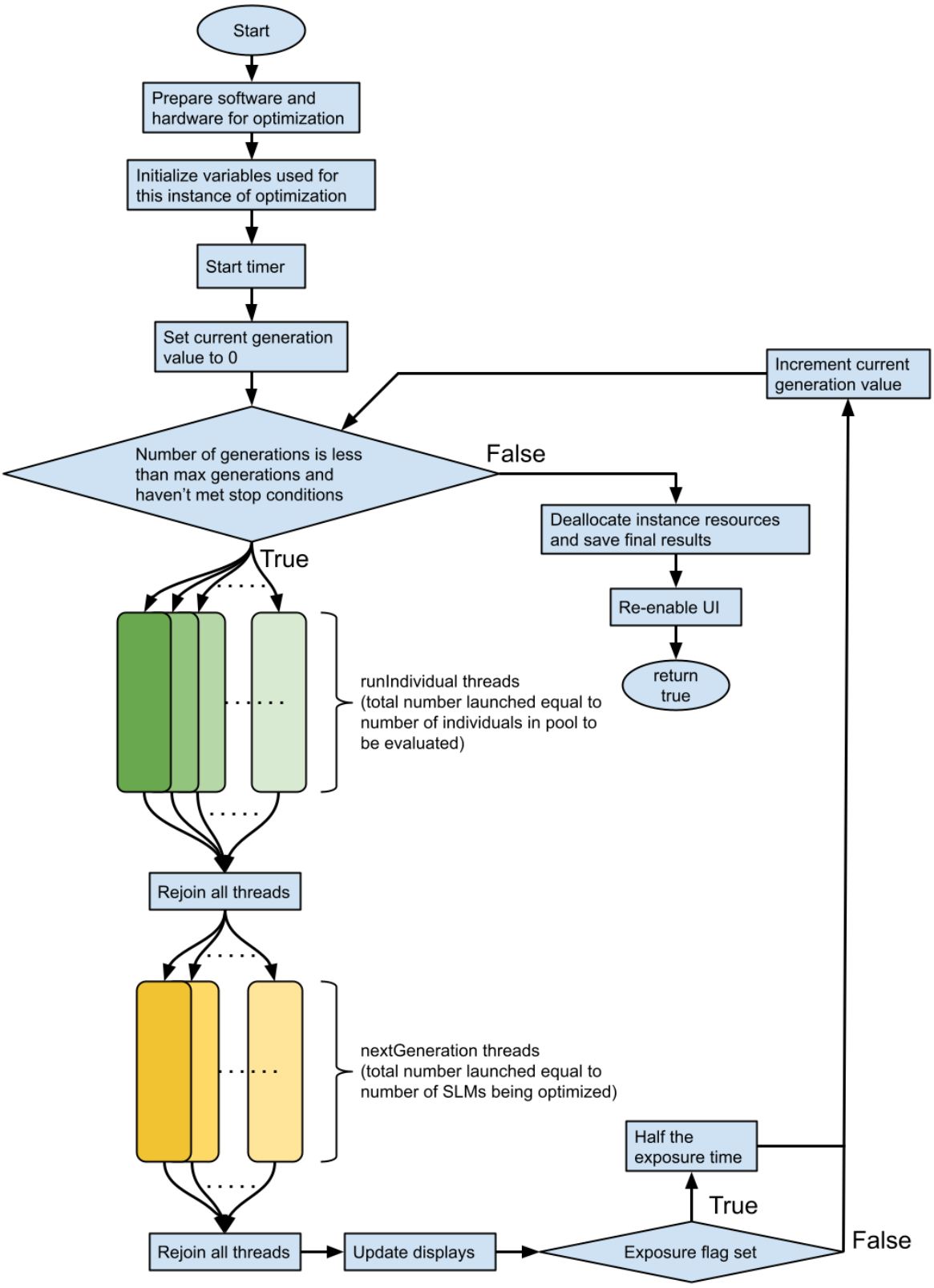

Figure 2. High-Level Flowchart of GA Optimization Process

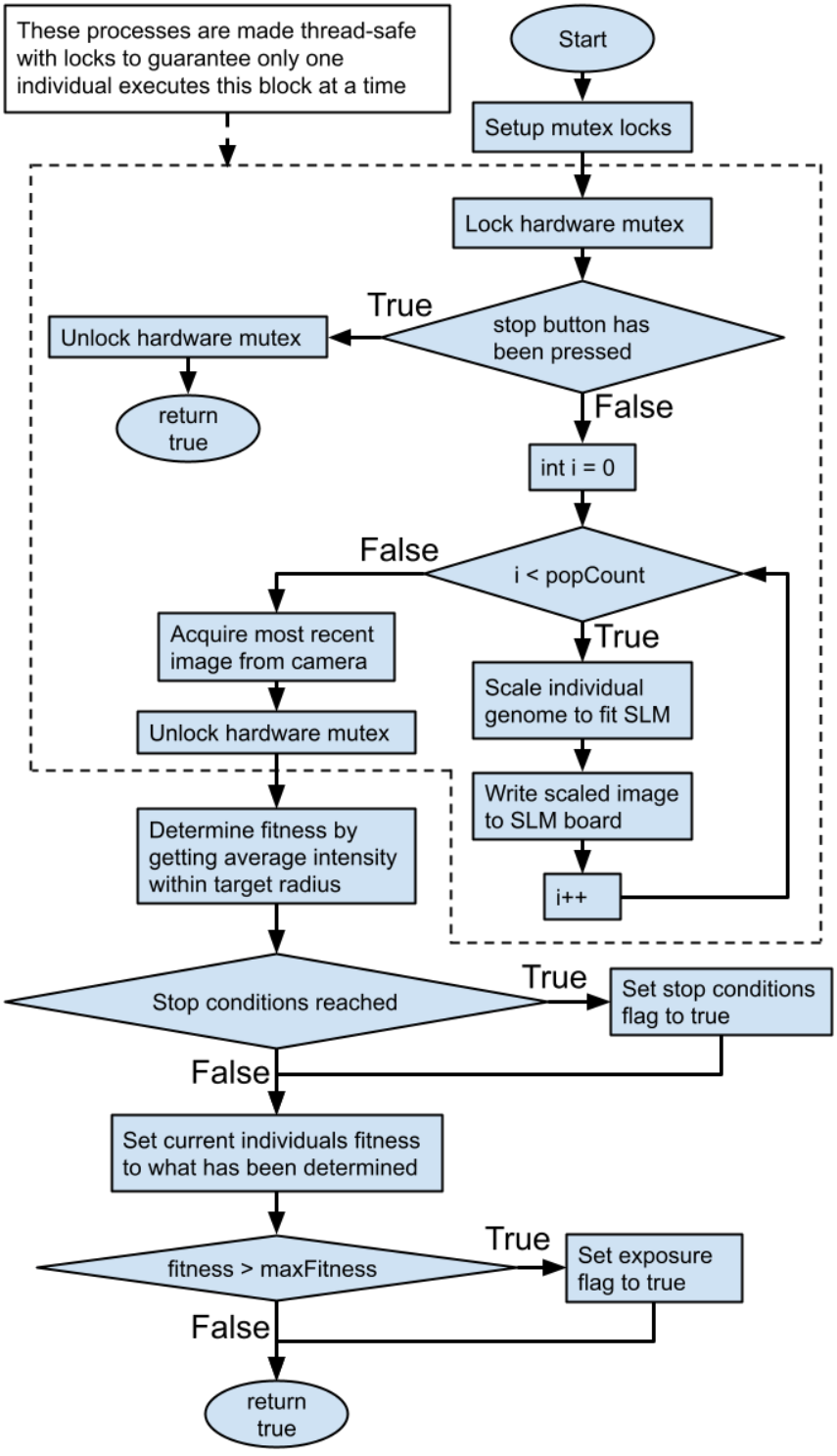

Figure 3. Flowchart of SGA and uGA Process of Evaluating an Individual

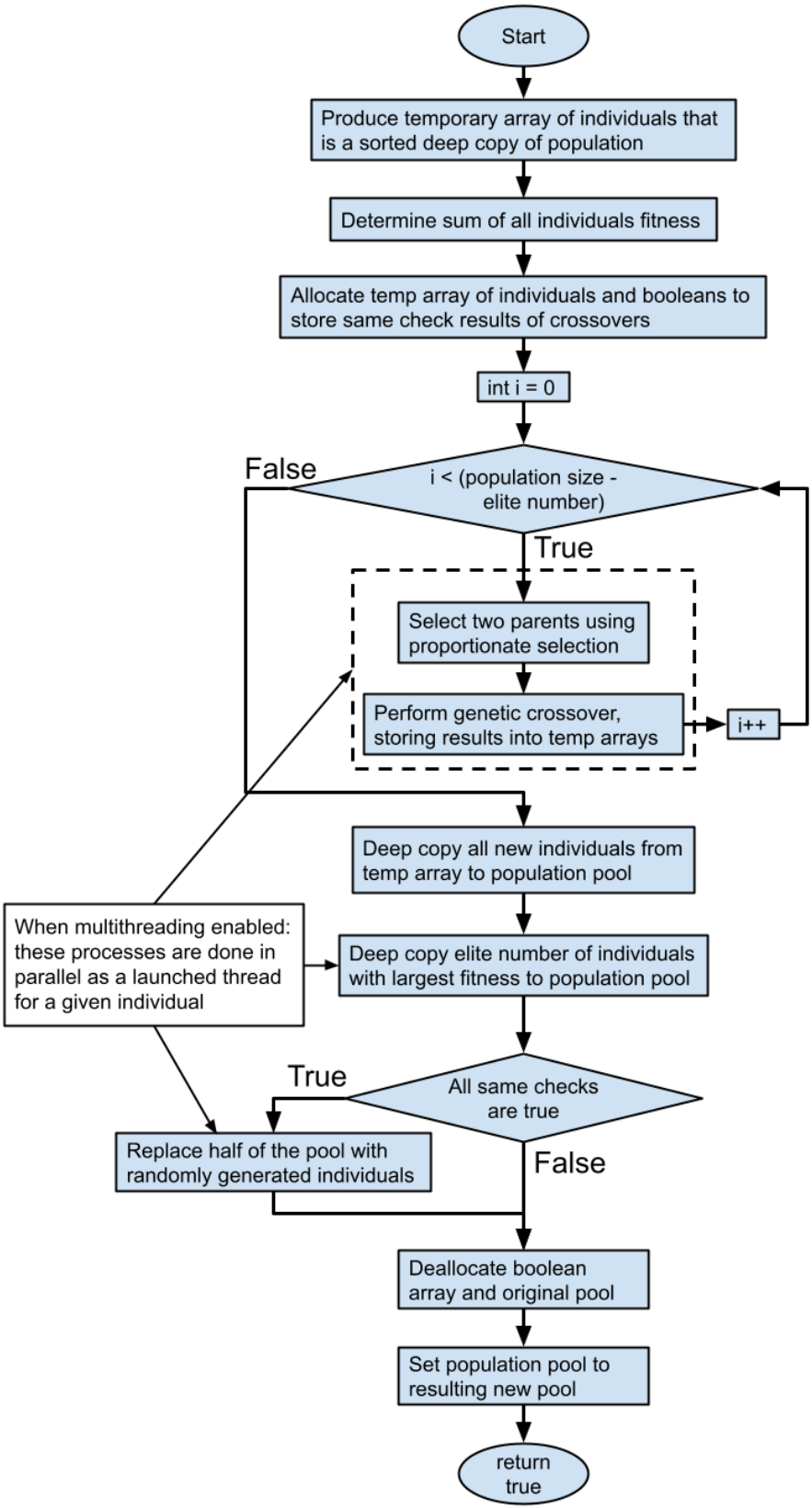

Figure 4. Flowchart of Generating New Generation in SGA

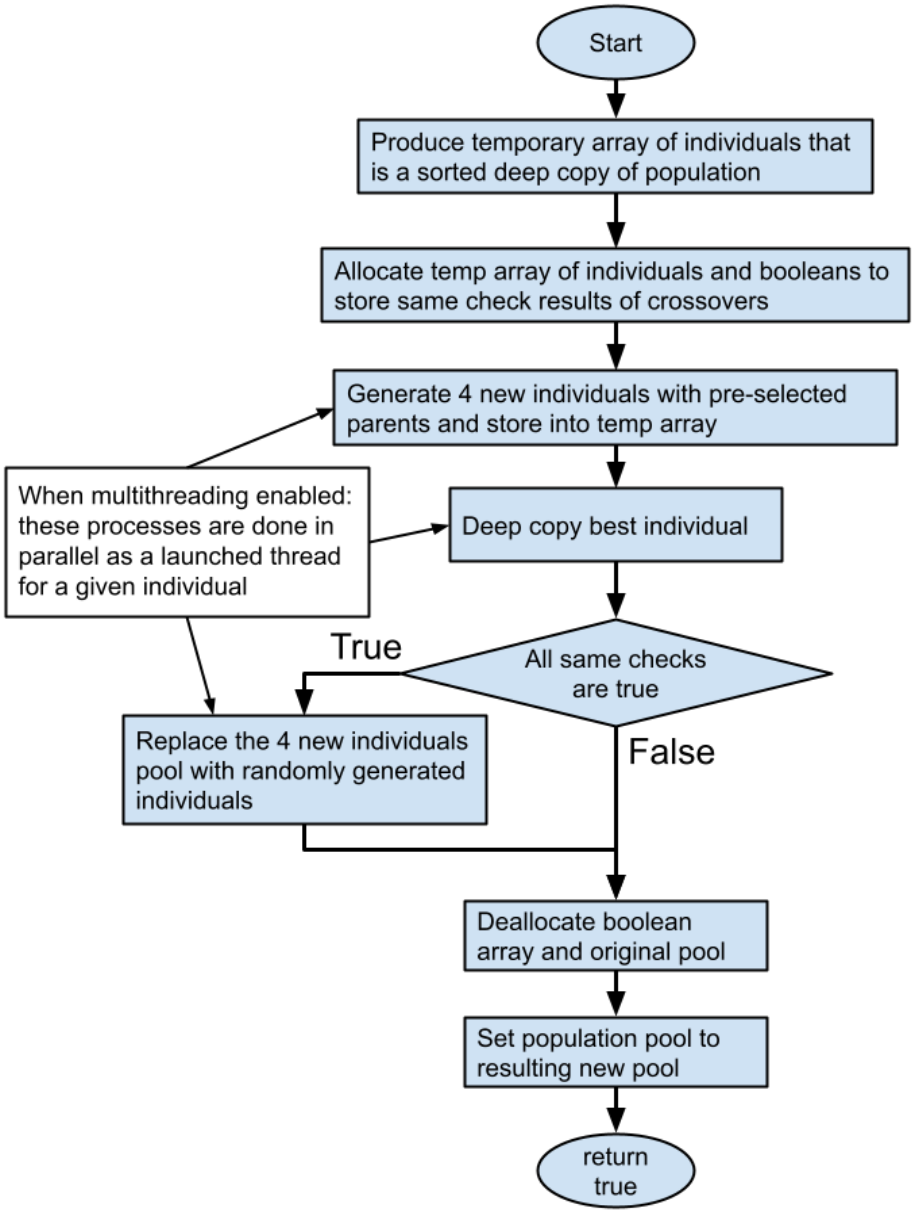

Figure 5. Flowchart of Generating New Generation in uGA

Appendix B: Additional Resources

Below are various links that will prove useful resources when working on the project:

Spinnaker C++ Programmer's Manual
http://softwareservices.flir.com/Spinnaker/latest/index.html

PICam 5.x Programmer's Manual
https://www.princetoninstruments.com/wp-content/uploads/2020/04/PICAM-5.x-Programmers-Manual-Issue-8-4411-0161-2.pdf

Meadowlark Optics Spatial Light Modulators User Manual
(contains documentation for Blink_PCIe SDK)
https://preview-assets-us-01.kc-usercontent.com/210eb585-dc2c-0008-b663-73b9ea5cff38/aab827a2-3e9c-4a2f-990d-70e0b6f328aa/1920%20PCIe%20User%20Manual.pdf

OpenCV Documentation & Tutorials
https://docs.opencv.org/4.5.2/index.html

Resources within this document (images and diagrams) should be made available in the project folder within the subfolder "Documentation Resources".

If you have additional questions regarding this document, you can contact the author [Andrew O'Kins] by email at andrewokins@gmail.com.