# SwimTech

Swim Tech LLC

## Flow and Resistance Visualization

CSCI370 - Advanced Software Engineering Fall 2020

Nicholas Battaglino, Lucia Grande, Jaxon McNeil

# Table of Contents

## Introduction

SwimTech is a small Colorado based water-sports company. They work to build communities around water sports and exercise. Part of what SwimTech does is provide swimming classes for all skill levels. In these classes it is helpful to know as much as possible about the swimmer in order to improve technique. SwimTech has the capability to record swimmers in the water, but there is an opportunity to do more with these videos.

This was where our team came in. We were asked to take these swimming videos and add an overlay that will highlight the points of high resistance on the swimmer. This overlay needed to add a visualization for the flow of the water around the swimmer. Making this work by any method was our goal for the semester. This application is meant to help swimmers and coaches see room for improvement in an easy to interpret view. The application loads in the video and uses computer vision to track the swimmers movements. This positional information is then sent to a model that calculates the flow of water around the swimmer and the resistance of water against the swimmer. After the computation, an overlay is placed back on the original video. This final application is easy to use and can be incorporated into SwimTech’s classes fast and efficiently. Ultimately, the success of our project was dependent on our final product being intuitive and easy to use for non-technical users and providing as much visual information as possible to the end user.

## Requirements

**Functional requirements**

The ultimate goal was a video player application that processes a video of a swimmer and returns a video with an overlay of resistance and flow of the water.

The user interface (UI) allows for the user to select a video to be processed. This UI is minimal as there is no need to see anything while the video is being processed. This processed video is saved as a separate file in the end, and the user receives a done message. There is no need for playback off the final video to be handled by our application because that can be done by any video player of choice. Our list of UI requirements is listed below. We were able to accomplish each of these tasks and more.

- A basic UI with options to process raw footage
  - Select video to process
    - Underwater videos as input
    - Swimmer moving laterally across the field of view
  - Save video with overlay
    - Process the selected video with Computer Vision model
    - Generate overlay
    - Save video with overlay separately from the original
  - Playback can be handled by another application

For tracking the swimmer, The main requirement was to know where the swimmer is in the water. The important part from the tracking is to get positional information from the swimmer. Having a point for the head and hips allows the rest of the code to function. This information then needs to be processed in a way so that resistance and flow can be derived. Our list of model requirements is listed below.

- Use computer vision to track swimmer and then model
  - Need to know location of legs, arms, and body to send to the model
    - Track a swimmer’s movements as they move across the water
    - Pass this data to a drag/resistance calculation model
  - Do drag and resistance calculation in a fluid dynamics mode
    - Calculate points of high drag/resistance against the swimmer

The main goal for the overlay was to pack as much information onto the final video as possible. This should include all the results from the calculation, but at a minimum should display the flow, resistance, and drag on the swimmer. This needs to be represented with colorful readouts that are easy to interpret. There is no need for this to happen in real time. By taking longer, the software can be more accurate and display more information. Our list of overlay requirements is listed below.

- Display an overlay onto the original video
    - Take information from the model to make arrows or colored zones
        - Highlight areas of high drag/resistance against the swimmer
        - Video Processing does not have to be in real-time
    - Place graphic back on the swimmer
        - Overlayed information must be informative and easy to interpret
        - Show as much information as possible
            - Leverage non-real-time processing to show extra info if possible

**Non-functional requirements**

The application must also be user-friendly, and easy to expand on later. This means that for users, the graphics are high-quality and accurate. For SwimTech, the code must be documented and delivered in a manner that allows them to use and understand the product independently of our team. This means using excellent object oriented programming (OOP) principles. Additionally, we need to make regular use of GitHub to document our development process. The original list of non-functional requirements is listed below.

- The overlay should present information from the model in a clear, easy-to-understand manner.
    - Graphics should clearly indicate points of resistance or drag (arrows, highlights, etc), as well as any other relevant information derived from analysis.
- The code should be well documented.
- The code should use appropriate OOP design to allow for easy expansion.
- SwimTech's GitHub should be used to store all code during development.

## System Architecture

### The Design

To accomplish this, we decided to create an executable file that can be run from the command line to accept and process an input video to produce video output. The program is written in the 4.4.0 C++ version of openCV, and utilizes the openCV tracking and drawing libraries. The video input is processed frame by frame by a video processing module that analyzes the video to detect the swimmer's position and movement throughout the video based on regions identified for tracking by the user. The user will specifically select the head and the hips of the swimmer to be tracked, as these two regions are the two most identifiable points of a swimmer’s form. This information is used to give an (x, y) position on the frame so that resistance can be computed. Figure 1 demonstrates an initial idea of how the tracking is completed internally.

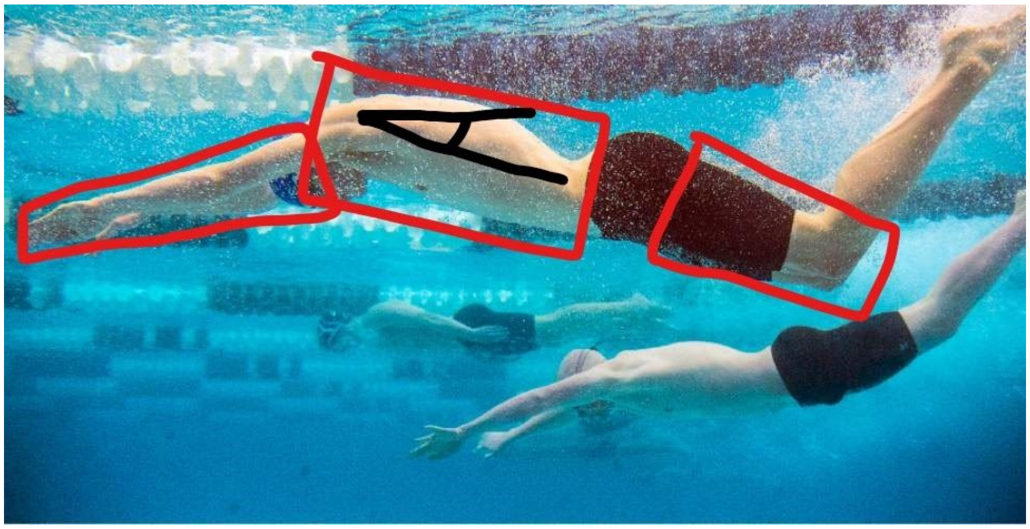


Figure 1: Initial swimmer tracking mockup.

From the tracking data, an overlay module then uses the body position to determine the swimmer’s body angle and location relative to the surface of the water to highlight the physical points of resistance throughout the video. The logic is, the steeper the angle, the more resistance through the water on the front of the swimmer. The larger the angle, the more open space behind the swimmer that will generate drag. Below, Figure 2 displays the graphical representation of this process. We do not need to calculate the amount of drag and resistance as a number to make this overlay. Instead we use openCV to highlight where the swimmer’s form could be more efficient based on the angle.

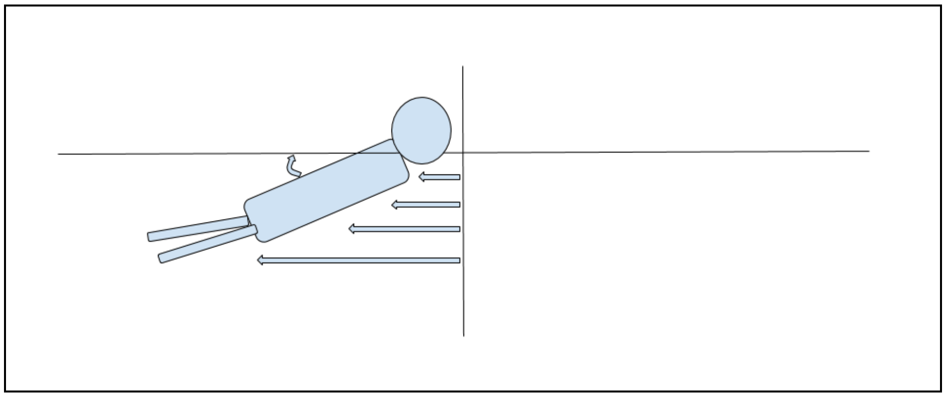


Figure 2: Initial Graphic explaining tracking and graphic calculation.

This information is passed back to the Computer Vision module for generating the overlaid visual onto the video. The overlay is processed frame by frame, so it is only dependent on the current image. The overlay consists of arrows in the front of the swimmer identifying resistance, flow lines around the swimmer, and the angle of the swimmer. The angle and arrows are colored on a scale determined by the angle. A steeper angle indicates bad form, causing the arrows to be red. Conversely, a smaller angle indicates good form, corresponding to a blue color. This overlay is placed on the swimmer using the location information from before when the swimmer was originally identified. Figure 3 shows the frame-by-frame process. For each frame, the tracker is updated based on the swimmer's location. The positional information is used to calculate the angle of the swimmer in the current frame as well as other variables necessary for the overlay. Finally the overlays are placed on the out frame and the frame is appended to our output video.

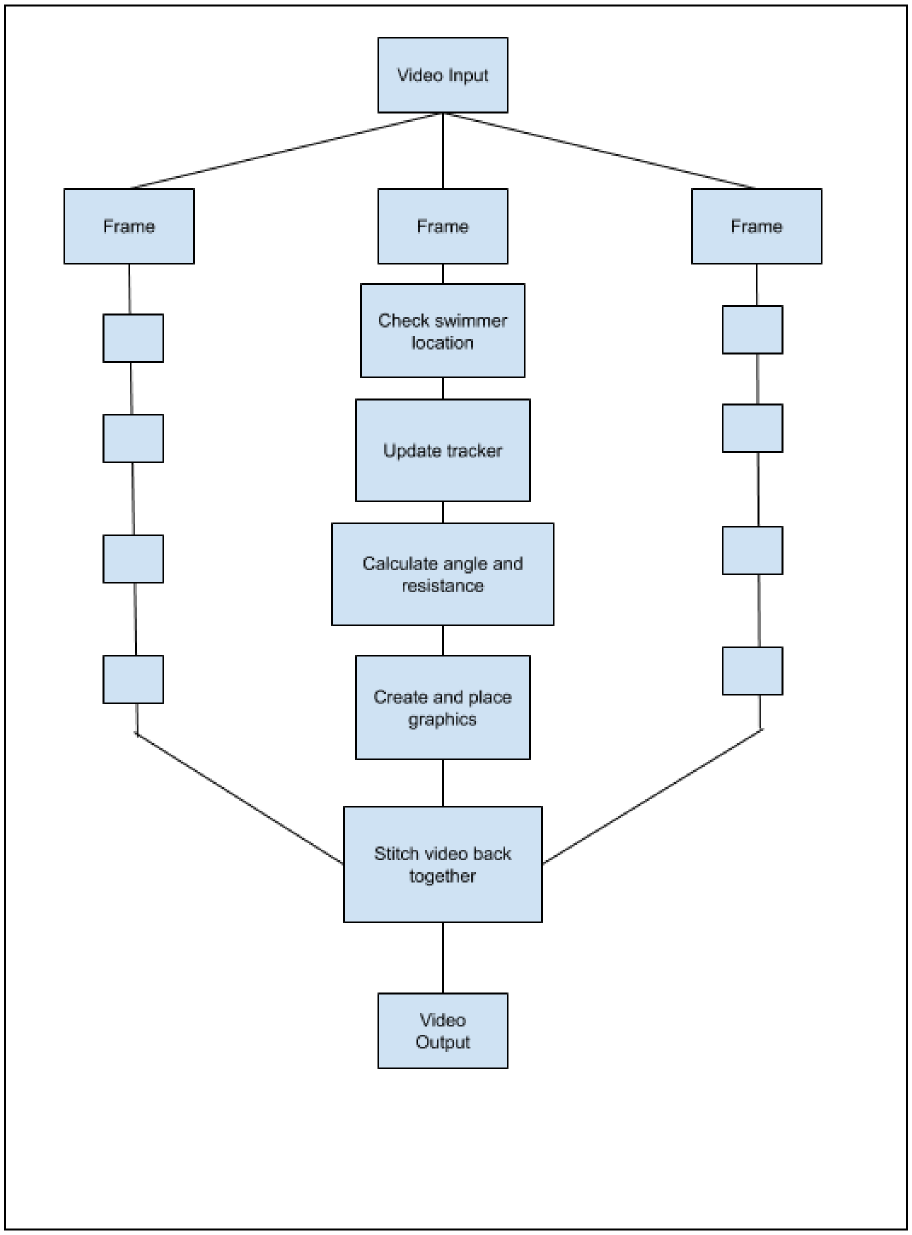

Video Input
Frame
Frame
Frame
Check swimmer location
Update tracker
Calculate angle and resistance
Create and place graphics
Stitch video back together
Video Output

Figure 3: Frame-by-frame process of design.

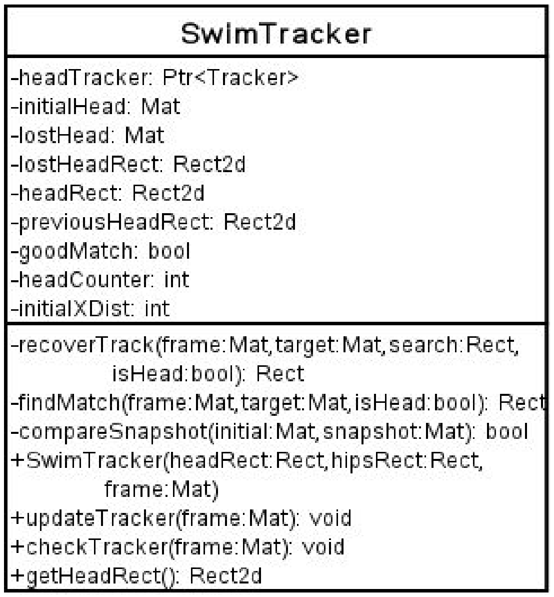
The user does not see any of the process. The user will only have to run the program and give the file path the original video. All the processing happens behind the scenes, so the user only sees the final output. The saved video is a separate output with the overlay on top of the original video, and should  appear in the same directory as the original video to be viewed later on a video player of choice. There is no need to have any other custom video player or GUI made because the user only needs to see the final output, which can be viewed with any regular video player.

## Technical Design

Our program breaks down into two major components: The SwimTracker class handles all tracking in the background while the Overlay class uses the generated positional information to place graphics.

### SwimTracker Class

The SwimTracker class is the core of the project and the most technically advanced piece. Its goal is to keep track of the swimmer in the water and return positional information. A simplified UML for SwimTrack is shown in figure 4. We built off of the given tracker class from OpenCV. The tracker class from OpenCV is based on two pieces of information, a region of interest (ROI) and a frame. When the tracker is created, it saves the image information inside the ROI. When given a new frame, the tracker updates the location of the ROI to what it is determined to be the best fit. This step is done with a combination of the image colors and a predicted location based on the previous movement. This serves as a great baseline but can be unreliable for the video input of a swimmer. The given tracker class works best when either the object or the camera is stationary. With videos of swimmers underwater, both the camera and target are moving. Our SwimTracker class attempts to overcome these challenges with some automatic error detection and correction. The project requires tracking both the head and the hips of the swimmer, and the SwimTracker class handles both so all the tracking can happen with one function call.



```
SwimTracker
-headTracker: Ptr<Tracker>
-initialHead: Mat
-lostHead: Mat
-lostHeadRect: Rect2d
-headRect: Rect2d
-previousHeadRect: Rect2d
-goodMatch: bool
-headCounter: int
-initialXDist: int
-recoverTrack(frame:Mat,target:Mat,search:Rect,
             isHead:bool): Rect
-findMatch(frame:Mat,target:Mat,isHead:bool): Rect
-compareSnapshot(initial:Mat,snapshot:Mat): bool
+SwimTracker(headRect:Rect,hipsRect:Rect,
            frame:Mat)
+updateTracker(frame:Mat): void
+checkTracker(frame:Mat): void
+getHeadRect(): Rect2d
```

Figure 4: of a UML for SwimTracker. Some duplicate variables were removed for head and hips. The full UML is in Appendix C

## TrackerKCF

Before the main body of code starts to loop over every frame, ROIs are selected by the user. These ROIs are passed to the constructor of SwimTracker to initialize the trackers based on the initial ROIs. The tracking ROIs are shown on figure 5. The tracker chosen from OpenCV to use was the Kernelized Correlation Filter (KCF) tracker. This option was chosen because of a reasonable balance of performance and run time. Other trackers were tried and were very accurate but cripplingly slow, or fast but prone to fail. KCF gives an acceptable run time with passable performance. For the length and variance of the video, some manual corrections are necessary and are described below.

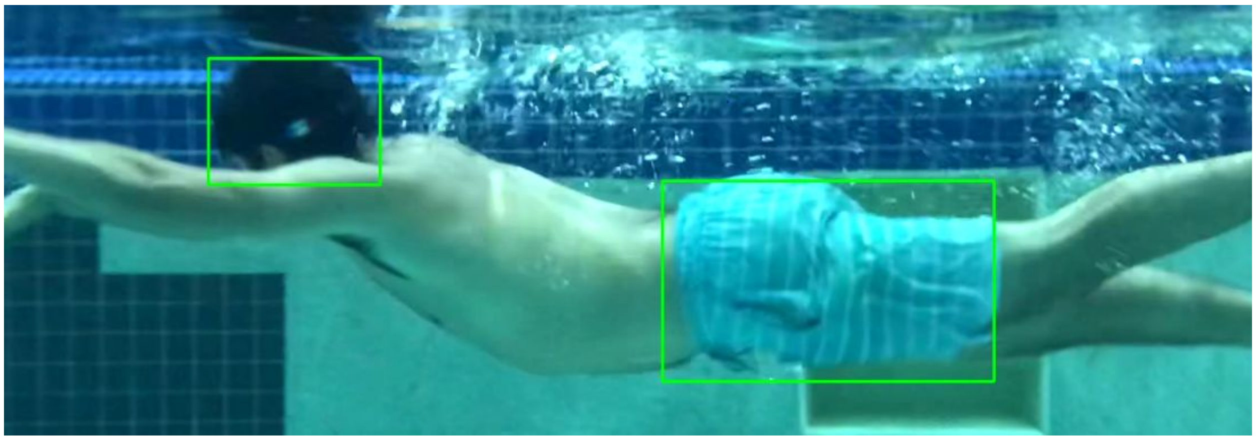


Figure 5: of two ROIs tracking the head and hips of the swimmer shown as two green boxes

## Failure detection

The first step in fixing the tracking is to detect when tracking has failed. When the given tracker fails it will not update the ROI and the object will move away from the old ROI. It is as simple as checking to see if the ROI has not moved to determine when tracking has failed. There is a small chance that the tracker is still working but the ROI just happens to be in the same place. While this case is unlikely, it is not damaging if it happens. In this event, the tracking is restarted. When tracking has failed a countdown is started. The countdown makes the SwimTracker class wait for a few frames to pass before attempting to reinstate the tracker. This is done because of the situations that tracking is likely to fail. When the head of the swimmer goes out of the water or is obscured with an arm there is no point trying to fix tracking right away. It is better to wait for a few frames so that the target is likely to be back in the frame. This method also saves processing power as time is not wasted on restarting the tracker when it is likely to fail again.

## Finding the new ROI

To fix the tracking, we need to find where the object is without asking the user for any help. The ultimate goal is to get an ROI that can be used to make a new tracker object. This process is done with the help of another OpenCV function called matchTemplate. This operation is a basic CV concept when you need to find an image in a larger image. This can be used for things like finding a ball on a field or a person in a landscape. Here, we are using it to find the head or hips in the frame of the swimming video.

When tracking fails, a snapshot is taken. This is the last known image that we try and find. The way matchTemplate works is by taking in a large frame and a smaller target image of this snapshot. It returns a probability map that is in the same format as an image. Each pixel is an integer that represents how good of a match was found. A white pixel is like the value 1 and means that a very good match was found. The small image slides across the larger one, and the similarity value is stored in the upper left point of the sliding ROI into the probability map at the corresponding location. An example probability map is shown in figure 6. From here, the maximum value in the probability map is found and that is the upper left point for the ROI that is the best match. The math performed on the image to make the probability map is based on the OpenCV function called Template Matching Correlation Coefficient (TM_CCOEFF) and is shown in figure 7. This was picked to be used for the matchTemplate function call because it was found to be reliable in finding the desired location of the target.

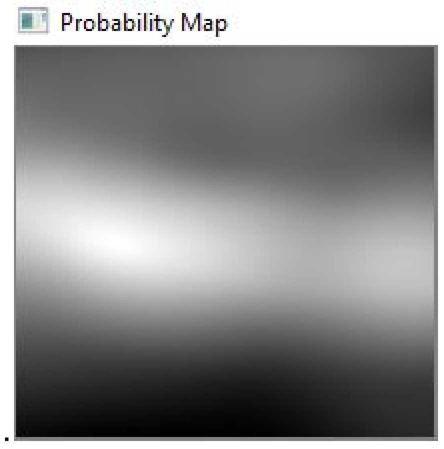


Figure 6: of a sample probability map

$$R(x,y) = \sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))$$

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \textstyle\sum_{x'',y''} T(x'',y'')$$
$$I'(x+x',y+y') = I(x+x',y+y') - 1/(w \cdot h) \cdot \textstyle\sum_{x'',y''} I(x+x'',y+y'')$$

Figure 7: of the equation for CCOEFF

## Construction of a subframe

One step to give this template matching a better chance is to give the function a smaller search frame to scan over. This subframe is prepared by a function that is called before the one that handles matchTemplate. In this function, a subframe is made from the larger frame. The subframe is twice the size of the target and centered on the target's last known location.  An example of this subframe is shown in figure 8. By only searching in a subframe, we accomplish two things: First is the accuracy is improved. It makes logical sense that the new location will be relatively close to the last known location, so we should only search near that. This prevents some random noise in the water that looks like a head to cause tracking to jump far away from the swimmer if the real head is obscured. Even if an inaccurate match is found, it still needs to be close to the original location and will still be relatively close to the right place. The second thing this does for us is save some time. The CCOEFF calculation in matchTemplate is very costly because it has a lot of math to run for each pixel. By cutting down the search frame, we reduce the number of unnecessary calculations that need to be done.

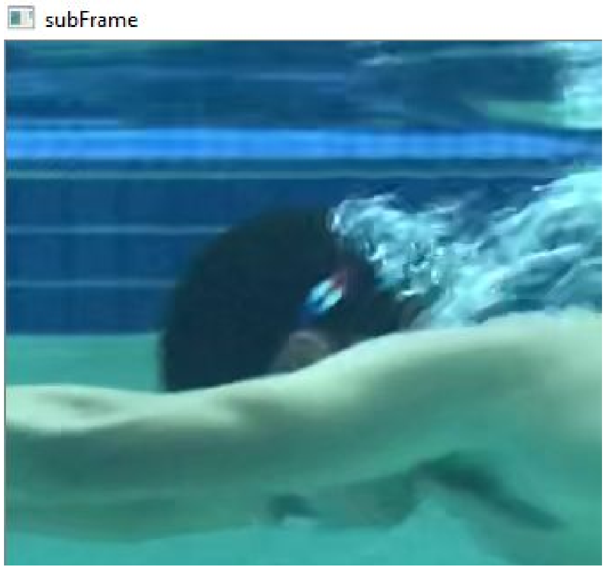


Figure 8: of an example subframe

**Original snapshot comparison**

Another slight advantage that we use to improve the chance of a successful re-tracking is the use of the original head and hips image. When the SwimTracker class is constructed, it saves the small image of the head and hips from the ROIs given. Any time tracking needs to be fixed, both the last known image and the original image will be run with the matchTemplate function. This means that we have two potential new ROIs. The one with the highest maximum value in the probability map is the one chosen to pass on. There is one last comparison to be done at this point before the new ROI is approved. The value from the probability map has to be higher than a set threshold score. This check means if both the new snapshot and the original image return a weak match, no update takes place. A flag is set so that no update is made based on the mediocre match.

**Snapshot rejection**

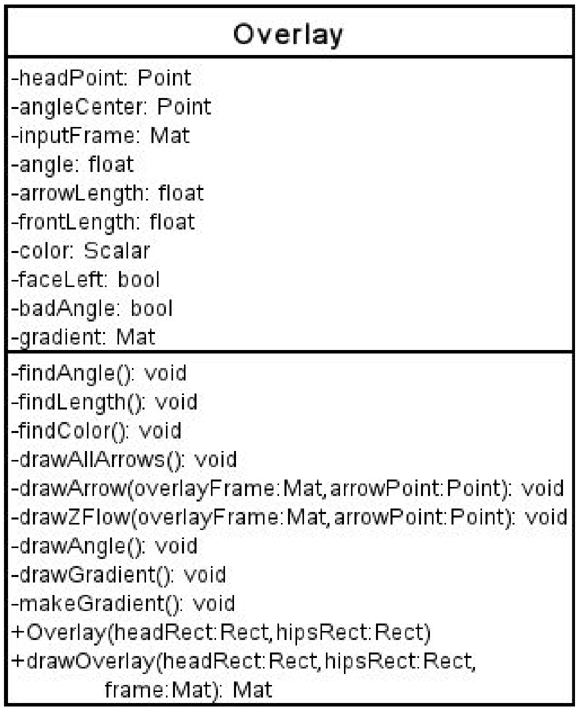
One step to ensure that the snapshot of the last known location is legitimate is by comparing the snapshot image to the original image. This step is done with the same logic as with the matchTemplate function except, the probability map is only one pixel. This is because the snapshot and the original image are the same sizes. Only one calculation needs to be done, and the resultant value is the score for how similar the images are to each other. The score needs to reach a certain threshold for the snapshot to be accepted. This is done because the snapshot image can be way off and then will continue to restart tracking based on a bad image. This step is a way to reject the snapshot and only look for the known constant of the original image. This is not done every time because there are cases where the angle of the video has changed so that the snapshot is better for restating the tracking.

**Head behind hips fix**

One last quick fix is to try and correct when the head has drifted behind the hips. This failure can happen to spite all the protections outlined above. It is easy to detect when the ROI for the head has started to overlap with the ROI for the hips. This should not happen for the geometry of a swimmer. If this is detected, the ROI is shifted to the right or the left based on the direction of the swimmer, and a new tracker is started. This step is a little bit of a shot in the dark because matchTemplate is not used, so there is no way to know that an accurate match is found. This consolation is acceptable because this is a failsafe that is called when every other measure has failed already. The length of the shift is found from the original relative position of the swimmer, so the distance should be close. If tracking fails again, the ROI will be closer, and the subframe should include the head, and normal tracking can be resumed.

## Overlay Class

The Overlay class is built to use the SwimTracker's generated positional information to produce a useful overlay for the input video. Similar to the SwimTracker class, this class is built around OpenCV. In this case, the class uses OpenCV's drawing tools to produce the output. This class isn't as sophisticated as the SwimTracker, it just uses trivial methods to communicate the information that the SwimTracker has produced. Figure 9 shows the UML class diagram for the Overlay class, and is further described below. See Appendix A (README) for more information on specific object members.



```
Overlay
-headPoint: Point
-angleCenter: Point
-inputFrame: Mat
-angle: float
-arrowLength: float
-frontLength: float
-color: Scalar
-faceLeft: bool
-badAngle: bool
-gradient: Mat
-findAngle(): void
-findLength(): void
-findColor(): void
-drawAllArrows(): void
-drawArrow(overlayFrame:Mat,arrowPoint:Point): void
-drawZFlow(overlayFrame:Mat,arrowPoint:Point): void
-drawAngle(): void
-drawGradient(): void
-makeGradient(): void
+Overlay(headRect:Rect,hipsRect:Rect)
+drawOverlay(headRect:Rect,hipsRect:Rect,
        frame:Mat): Mat
```

Figure 9: A UML diagram for the Overlay class. Constants and duplicate variables were removed. A full UML is in Appendix D.

### Calculations

The Overlay is initialized with the initial bounding boxes that the user inputs around the swimmer's head and hips. Then, in the main loop of the code, The drawOverlay method is called passing in each frame's head and hips rectangles (outputted from the SwimTracker) along with the frame itself. In the constructor and the drawOverlay method, a set of basic calculations are done that are used later on for drawing. First, upon instantiation of the Overlay class, the head and hips boxes are compared to determine if the swimmer is facing left or right. At each frame, this is used to first pick a point to represent the head and the hips. Using these, the angle of the swimmer is calculated. Finally, the angle is used to determine the length of the arrows and the color of the various overlays.

## Drawings

The next steps are fairly straightforward. The various draw methods are all called on the current frame. Each of these methods uses OpenCV drawing functions to generate graphics and place them on the input frame. Figure 10 and 11 below show the graphics that are used.

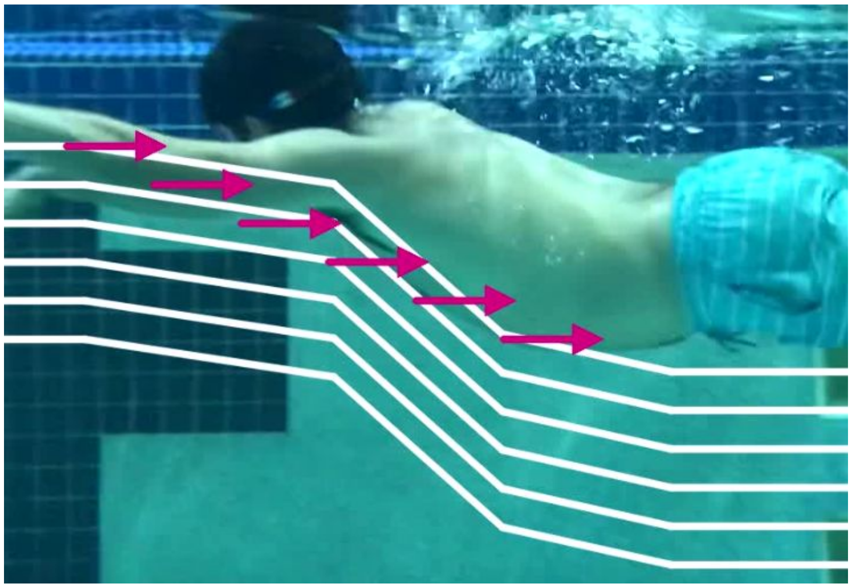


Figure 10: example of overlaid arrows and flowlines.

The drawAllArrows method is used to generate points on the line connecting the previously created head and hips points. The function drawArrow is called for each of these points and an arrow is drawn based on the previously calculated arrow length and color. In this function, the arrows are drawn with some degree of transparency. In the Overlay class there is a global variable that is stored as an angle threshold for opaque arrows. In the final submitted code, we set this to 20 degrees. In other words, when the swimmer’s body is greater than 20 degrees, the arrows are drawn with no transparency. As the angle approaches 0 degrees, the transparency linearly increases to full transparency. Flow lines are also drawn for each of the points generated by drawAllArrows. The drawFullZFlow method is used to do this. Each flow line is composed of five individually drawn lines. To accomplish this, there are variables set to define the horizontal from the line between the head and hips at which the joints of the flow line should be drawn. The vertical distance of the joints automatically scales based on the distance of the previously generated head and hips points. In the event that the hips point rises above the head point, a boolean (badAngle) is set to true and the drawAllArrows method omits any visualizations.

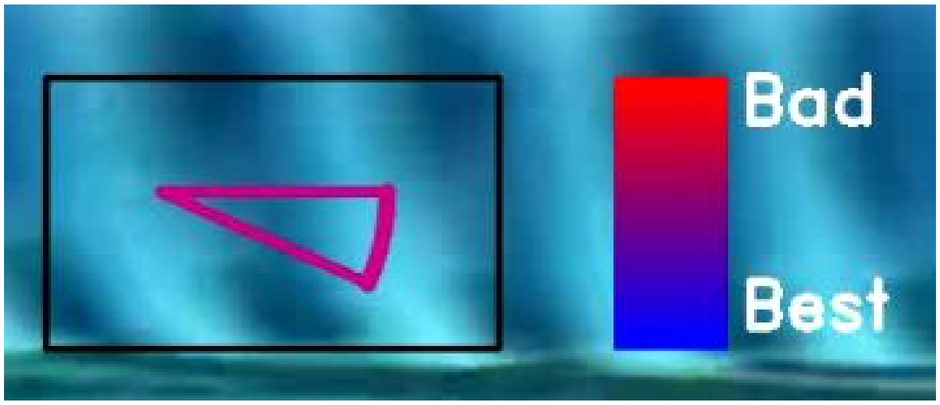


Figure 11: example of the angle and gradient

## Additional Graphics

The final two components of the Overlay are the angle representation and the gradient. These each have their own draw methods called on each frame. The angle is drawn in the top left corner using the previously generated color with a box enclosing it. The gradient is essentially a legend to show the scale of colors used for the other graphics. It was important that it was easy to tell points in the swimmer's technique that could be improved and a good color scale is an effective way to do this. Initially the scale went from red to green. This scale could possibly discriminate against users with some forms of colorblindness. In the final code, blue is representative of a good swimming angle with low resistance while red is a bad angle with lots of resistance. The actual gradient graphic is only generated once in the constructor using the makeGradient method. drawGradient is the method called for each frame to actually place the gradient onto the input frame.

Once all the graphics have been generated and placed onto the input frame, the frame is returned and stitched together with the other Overlaid frames for the final input video. Figure 12 shows the overall output with all the graphics displayed.

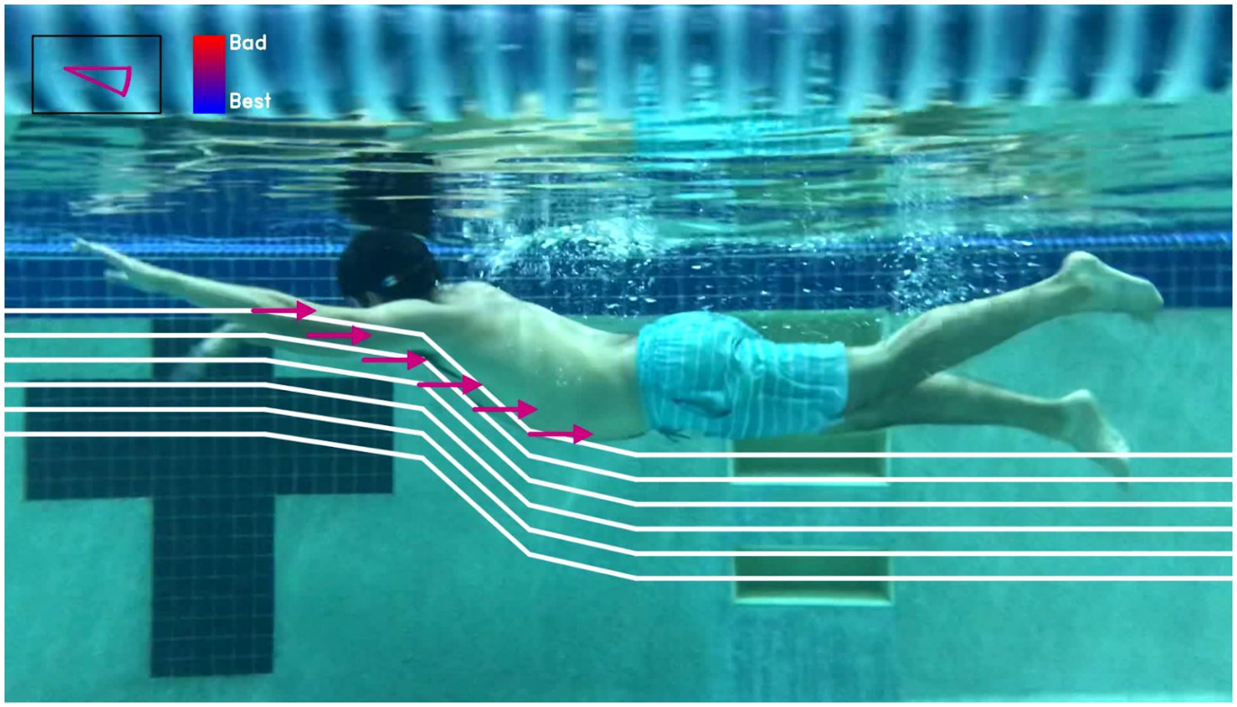


Figure 12: a full frame of the video with overlay

## Quality Assurance

Ensuring the quality of the software was a crucial part of development. This not only meant ensuring that the program met functional standards, but that it met ethical standards as well.  Our project needed to deliver high-quality outputs to the users. Swim coaches would be using or software poolside during swim lessons, so there could not be bugs or issues with the program that inhibit teaching. This output needed to be easy to understand and work for everyone. Our quality assurance plan was based around the Scrum mentality and constant review of our work, and involved both daily and regular assessments.

**Ethical Considerations**
To quickly cover the motivation for our project to achieve high quality, we can look at some of the ethics involved. We needed to consider who might be the users in the video and how we needed to ensure that our solution works for everyone. For SwimTech, we needed to make sure that we took the necessary steps to ensure that we delivered a useful solution that is in line with SwimTech's expectations. We also needed to ensure that our final product was complete and bug-free so we did not make more work for SwimTech.

**Definition of Done**
Our definition of done was well defined by the constraints handed to us by our client. Additionally, this definition is further fleshed out by the ethical considerations that we have outlined. Firstly, our program requires reliable tracking of the swimmer underwater. This means that the tracking algorithm that we implemented needed to work in a variety of conditions including different pools, lighting, and swimmers. The algorithm should not have worked worse for people of a certain gender, body type, or race. Reliable tracking is important because the rest of the program is dependent on this to work well. Next, our program needed to execute in a timely fashion. This software is meant to be used as a coaching tool, so the instructor and student shouldn't be waiting an unreasonable amount of time for the output to be generated. This, of course, meant that our tracking program not only needed to be reliable but also efficient. It was also important that the output created is easily understandable and informative. Both the instructor and student should be able to easily understand the output and use it to improve the student's swimming technique. Furthermore, our output was required to be visually appealing to fit in with professional standards. Finally, our actual code needed to be well documented as well as use object-oriented design so that it is extendable by SwimTech in the future.

**Overview of Plan**
To ensure that we abided by our ethical considerations, they were kept in mind throughout each phase of the design process. Since we executed our software quality assurance plan daily, as well as on a weekly pattern, these were great opportunities to check our program to ensure our progress aligned with our goals. There are some specific steps that we took during these meetings to do this. First, we ensured that we tested our code, namely our tracking algorithm, to confirm that it worked on a diverse set of inputs. During our regular coding sessions, this was as simple as changing out the input video between tests to ensure that it still worked. Since we coded in pairs or as a triad, the person(s) not in the driver seat could constantly review the readability of the code. On a less regular basis, we returned to the code and made sure the commenting and documentation was sufficient. We also reviewed and discussed the code's architecture to ensure that it made sense and was easily extendable. Finally, in testing our code, we made sure that the output was correct and accomplished the task that we were focusing on at that time.

**Daily Quality Checks**
The core of our plan was based on using the agile development process. We made tasks that we tracked with the free tool called Jira. All of our work related to a story we wrote, so we were easily able to keep track of tasks, as well as the overall project. We had daily stand-ups (DSUs) every Monday, Wednesday, and Friday. This allowed us to update each other if we have done some individual work, and also review what was accomplished, and decide together if the task was sufficiently completed. This meant that only stories that met the definition of done made it into the final project. At the DSUs we also spent some time looking over the code and thinking about what needed to be improved, in terms of performance, documentation, or general organization. We were then able to write a story for that item so we did not lose track of anything that needed to be done.

We tried to do all our coding in pair programming style, or all together. We had one person writing the code and one person focused on the details to help the coder. The other person is a little less focused on the micro aspects and spends more time thinking about the macro aspects of the project, or researched relevant topics to help guide the overall direction of the software development. Working this way allowed us to do a lot at once and made the best use of our time together. Pair or triad programming helped us catch most bugs or typos instantly, and made the code more readable, since we were constantly communicating.

In the process of coding, we were constantly checking our output video. In OpenCV, it is very easy to see what is happening, as we were able to display all the necessary intermediate frames to see what is happening at each step. This made debugging very convenient because we were able to see what was going wrong. Practicing running the code was essentially user interface testing. We interacted with the program to select different boxes for the swimmer as the user would, and gave varying inputs. We followed the instructions given as if we were unfamiliar with the software to make sure that our instructions were clear and easy to understand.

We also constantly monitored the runtime, and made small, incremental changes to see the impact on processing time. In the end, faster is better, so we continually compared the runtime from a week ago to the current build to see how it had changed. Adding more processing will increase the runtime, so we tested many options to find the process that added the least amount of overhead. Runtime was at the front of our minds whenever we did refactoring as we looked for opportunities to improve. The goal was to make sure that the processing time was less than 5 minutes for a video of a length less than 20 seconds.

**Regular Quality Assessments**

In addition to the daily checks, we had regular assessments that took a more in-depth look at the same areas of the project. They worked to assess the general functionality and progress of the project and ensured that it worked all together. At the end of each sprint, we set aside time for a meeting to discuss our work throughout the previous sprint, and what we plan on working on for the next sprint. In those meetings, we not only went over the logistics in Jira but also the software and code itself. This included fully running the program multiple times with varying inputs, checking the output for errors or bugs, and performing a full code review, and discussing any issues it currently has. Specifically, we were looking for any unexpected behaviors in both the code and the output, such as exceptions, early termination or incorrect tracking. When issues like these were found, they were added to our to-do list for the next sprint. We tested the software on a variety of videos and it needed to work for all of them in order to pass. By performing these regular performance checks of the code, we ensured that it maintains functionality and quality throughout development.

Outside of our testing, we also had regular meetings with our client to both discuss the progress of the project, but also to perform user-based testing. Since the client is both the user and the owner of the program, it was crucial that they not only understand it but also approve of the usability and functionality. Also, the software will be used in time-sensitive situations with non-technical people, so it was important that it is easy to use and meets the expected functionality set by the client. During our meetings with the client, we fully ran the program and walked them through the code itself, to show various outputs and cases. This is when we received feedback on the performance and usability of the program. In order for our software to pass these kind of user-based tests, the client needed to approve of the functionality and the overall output of the program at every stage of development, with the most crucial test being the final delivered product.

## Results

### Input Testing

Our testing primarily revolved around different inputs. Since our software will be used in a variety of different locations, with different swimmers and different strokes, we needed to ensure that it can handle those changes in conditions seamlessly. The ‘testing’ that we performed was essentially running different videos through the program and observing the behavior of the program itself, as well as the final output. The videos were provided by the client, and as such we did not have a wide variety of locations or swimmers. We were able to test our project with as many input videos as we received. While our set of input videos was not fully representative of the general population, we diversified our testing videos to our fullest extent. In addition, the program needed to be able to handle any type of stroke - butterfly, backstroke, breaststroke and freestyle all have different movements and techniques, so the program needs to be adaptable between strokes. We did have videos of all four strokes, so we were able to test the ability of the program to handle various body types and positions.

Before we received videos with all four strokes, most of our development was done with a video of a male swimming freestyle towards the left. While the swimmer stayed within an acceptable angle of the camera, the tracking maintained functionality and produced the expected output. However, when we tested on the new freestyle video with the swimmer heading towards the right, our graphics were incorrect. When we tested the other strokes, there were some issues with tracking due to interference of the hands and arms, and additionally, our calculations of resistance were incorrect when the head would drop below the hips, causing the graphics to not be displayed. The direction-related issues were addressed quickly and relatively easily in the code. However, the failing of the tracking algorithm was a larger issue that needed to be more carefully considered. In order to have more seamless tracking, the program is now able to recognize when it loses the original image, wait for a few frames, and then use a matching algorithm to find the most probable location of the lost image within a certain range. This allows for the head to leave the frame, and then be found again once it re-enters the video.

## User Acceptance Testing

An important aspect of our testing has been user-based acceptance testing. Our client is also the user, so the videos provided are a sort of requirement for user acceptance. In our meetings every week, we were able to update him and provide examples of output, providing a constant supply of feedback. Since our program successfully tracks and provides the expected information for the videos provided, we know that our software is acceptable by the client’s standards.

## Usability Testing

Our software is meant to be used poolside as a tool to assist in swim coaching, so our testing took into consideration the usability of the program in this context. We found that there are certain variables that directly impact the efficiency and correctness of our tracking algorithm. Firstly, a higher framerate has a direct correlation with the time it takes our program to resolve. This makes sense because our algorithm iterates through each frame, so a 60fps video will take roughly twice as long as a 30fps video of the same length. As a trade off to this, a higher framerate often means less blurry frames, which means more accurate tracking. Upon testing our program on some of the client’s given input videos, we found that blurry frames are very difficult to maintain tracking on. We took measures to recover lost tracks in these scenarios, but it is important to keep in mind that each of these measures negatively affect the performance of the program overall.

## Lessons Learned

In taking on this project, none of the members of our team had any experience with video processing of any type. We’ve progressed leaps and bounds from where we began, but the entire process was a major learning experience for the team. Overall, the main points learned have to do with details of OpenCV and some of the specific tools it contains. We had to do a lot of experimentation with several different methods to land on the tracking algorithm that we designed. At each step we had to make sure that we understood what we were doing with openCV first before we could bring it into our design. Each iteration of the program was a lesson in and of itself.

Additionally, we all learned many lessons in iterative software development. Using Agile throughout the process taught us how to efficiently plan and build software at a steady pace. This process also taught us how to communicate with a client and each other as a team. We had to make many revisions based on feedback from the client. This required us to change course many times and refactor our whole design to meet the updated requirements. Ultimately, our design was a success because we were able to adapt to the shifting needs of the client and still produce a quality product in the end. These are all useful skills that will be beneficial going forward in our software careers.

# Appendices

## A - README

////////////////////////
Libraries Used
/////////////////////////

The program is written in openCV, an open source computer vision and machine learning software library. Specifically, it is written in the C++ 4.4.0 version of openCV. Outside of the main openCV library, we also used the openCV tracking and drawing libraries, which need to be linked manually. Here is a tutorial on how to install openCV (and extended libraries) on Windows 10:
https://github.com/MicrocontrollersAndMore/OpenCV_3_Windows_10_Installation_Tutorial

A great way to learn more about openCV (and many of the functions used in the program) is through the tutorials at https://docs.opencv.org/master/d9/df8/tutorial_root.html. In addition, https://www.learnopencv.com has many great resources and tutorials.

//////////////////////////////////
How to run the code
//////////////////////////////////
----------------------------
For the Swim Coach
----------------------------

1. Make sure that the program is in the same folder as the video you want to use. Make sure that that the video is a .mov
2. Run the program
3. You will be prompted to enter the name of the video. Type in the name without the filetype extension
4. You will now see the first fame of the video and a control slider. Use the slider to select the first frame of the video. A good frame is one where the head and the hips can clearly be seen without any obstruction from the arms or the water
5. Press ESC when you are done
6. You will now see the frame you selected, and it is time to select the head and the hips.
7. Click and drag to max a box around the head of the swimmer. A big box that includes some extra water is better than a small box that cuts off part of the head. Press space to lock-in that box
8. Click and drag to make a box around the hips now. Try and include some of the swimsuit so the box is not only skin. Press space to lock this in.
9. Press ESC when you are done to run the program.

10. The program is running now, and you will see a message when it has completed the processing. Look for the output video in the same folder as from the start. It will have the same name but end in OUT

---------------------------------------
Tips for the swimmer + filmer
---------------------------------------
Just swim like normal, there is no need to adjust. One thing that can be done to help the tracking is in the choice of swimwear. It is best to pick a solid color for the swimsuit or swim cap. Avoid white, black, or blue. Other than those, any distinct solid color would help a lot. When making the original recording, try to move with the swimmer at the same rate. This will keep the swimmer in the same place in the frame so it is easier to track the change in the body position. Keep the angle of the video straight on the side of the swimmer so that no rotation of the camera is necessary. This helps the program see more of the swimmer.

//////////////////////////////////////////////////////
Code breakdown and explanation
//////////////////////////////////////////////////////

---------------------
Main body
---------------------
This is where everything starts: the input file is chosen, the reading of the video is initiated, the user selects the beginning frame and the initial tracking regions, and the processing begins.

The first section of the code primarily deals with obtaining the input video and starting locations via user input. The program currently runs on a provided video input, so the user must select the new video by typing in the name of the desired video. There is also an option to run the program in debug mode, which will allow the user to watch the video as its processing. To have the user select the regions of interest for tracking, a slider is created to allow the user to select the ideal starting frame (clear shot of head and hips). Then, they create the tracking boxes themselves. All of this comes with instructions shown via the command line.

Once all of that is done, the program then enters a loop that continues until the end of the video is reached. The loop processes one frame at a time by updating the trackers, checking the locations of the head and the hips, and then drawing the graphics onto each frame and saving it to the output video. All of the functions used in the loop are described below, in the SwimTracker Class and Overlay Class sections.

-------------------------
SwimTracker class
-------------------------
The basic overview of the SwimTracker class is that it is responsible for tracking the head and hips of the swimmer. From the main body of the code, the trackers are created when the

constructor is called. From this point on, SwimTracker handles everything and is only referenced to update the trackers on the new frame, check if tracking has failed, and read the useful positional information.

Class Methods:

The constructor:
OpenCV has its own tracking objects that it uses to track a single image in a frame. The SwimTracker class holds on to two of these trackers. The constructor saves pointers to these trackers as class variables. The constructor also saves the initial image of the head and hips for reference later.

Update tracker:
This is what is called for every frame of the video. Essentially, it calls the OpenCV function update on each of the trackers which moves the tracking box to the new location. It also holds onto the last known location of the box to help detect when tracking has failed, but that is in the check trackers function. Update trackers is also responsible for calling the function recover tracker when tracking has failed and there have been sufficient frames skipped. If not enough frames have been skipped, it just decrements the count down. If it is time to recover tracking, it will call recover track and replace the failed tracker with a new one if the match has been ranked as good enough. Update tracker will also check to see if the head tracker has failed and ended up behind the hips tracker. If this is the case, It will be corrected and reset onto the head.

Check tracker:
This is what is called by main to see if the tracking has failed and then work on fixing it. For both the head and the hips, check tracker will check if the box has moved by comparing the current location to the location from the last frame saved by the update tracker function. If they are the same, this means that tracking has failed. At this point, check tracker will take a snapshot of the head or hips to compare to later to fix the tracking. The snapshot is compared to the original head or hips image with the compare snapshot function and will be rejected if it is not good enough. At this point, the countdown is set so that update tracker will call recover tracker at the appropriate time.

Recover tracker:
The goal of recover tracker is to prep the frame so that the target can be found in the right area. A subframe is created to be twice the size of the image that is being searched for. This subframe is cut out of the full-frame. This prevents the tracker from jumping far away from where the head or hips should be. The subframe is also adjusted so that it does not go over the border of the full-frame. From here, find match is called. After we get the new location of the updated tracking box, the location is shifted inversely to the creation of the subframe so that it is correct, relevant to the main full frame.

Find match:

This is where we find a new location in the frame to track. This is based on the OpenCV function called matchTemplate. matchTemplate looks for a small given image in a big frame. We create a probability map that stores the match information. We call matchTemplate and pass in the subframe from before, the target image, and the probability map. matchTemplate moves the image over the frame and records how similar the image is in the probability map. This is done for both the snapshot and for the original image of the head or hips. Only the best match is saved. Then the match is validated against a baseline. If the score is not high enough, goodMatch is set to false and the tracker is not updated. If it is good enough, the location is returned up so that a new tracking object can be created.

Compare snapshot:
This is a small function that compares the snapshot to the original head or hips image. This prevents a snapshot from being taken of the water, and then start matching based on that. The logic here is very similar to that of find match. Except here, only one comparison is done so the probability map is only one by one. The similarity is again compared, and if the score is not high enough, the snapshot is rejected.

Constant Variables:

WAIT_FRAME is the number of frames to wait before fixing the tracking. If it is set to 3, tracking will be fixed on the third frame. For strokes that have the head out of the water for longer (butterfly/breaststroke), a higher number like 5 or 6 should be used. For strokes like freestyle or backstroke, a lower number like 2 or 3 is acceptable.

UPSHIFT_FOR_JUMP_FIX is the ratio of vertical to horizontal distance to move when fixed the “head hips overlap” problem. 1/6 works pretty well.

FIND_MATCH_THRESHOLD is the number threshold to decide if a tracking image should be accepted. If this is lowered, tracking is more likely to start again but, it might be wrong. If this is higher, tracking will wait for a better match but you will go more frames without any tracking.

SNAPSHOT_THRESHOLD is the same but for the snapshot. The higher this is, the more likely it will be that the snapshot is rejected. Then only the original image will be used.

----------------------
Overlay
----------------------
The Overlay class uses positional information taken from the SwimTracker class to place overlays over the original video. From the main body of the code, the overlay is created once and the drawOverlay method is used throughout the main loop to place the overlay frame by frame.

Class methods:

Overlay(Rect headRect, Rect hipsRect) //Constructor
The initially drawn bounding boxes of the heads and hips are passed into the constructor. These are then compared to determine the directions that the swimmer is facing and, thus, which way the overlays will be drawn.

Mat drawOverlay(Rect headRect, Rect hipsRect, Mat frame)
This method uses the current head and hips boxes from the SwimTracker as well as the current frame to place all the overlays on the frame. First, drawOverlay determines the swimmer's angle. This is used to determine the length of the arrows and the color of the overlays. The arrows, angle, and legend are then drawn on the given frame and the frame with the overlay is returned.

Private methods:
void findAngle()
This method finds the angle of the swimmer and stores it to the Overlay class's angle member.

void findLength()
This method calculates the arrow length based off the angle and stores it to the Overlay class's arrowLength member.

void findColor()
This method calculates the overlay color based off the angle and stores it to the Overlay class's color member.

void drawAllArrows()
This method iterates through points along the line that connects the Overlay's headPoint and hipsPoint members to draw water flow lines (via drawZflow) and arrows (via drawArrow).

void drawArrow(Mat overlayFrame,  Point arrowPoint)
This method draws an arrow pointing at the passed in arrowPoint. This arrow is drawn based on the previously calculated arrowLength, color, and the direction of the swimmer.

void drawZflow(Mat overlayFrame, Point arrowPoint)
NOT USED IN FINAL CODE. This is the original method used to draw flow lines. This method draws a flowline relative to the passed in arrowPoint. This flow line is drawn based on the previously calculated arrowLength, color, angle, and the direction of the swimmer.

void drawFullZflow(Mat overlayFrame, Point arrowPoint)
This method draws a flowline relative to the passed in arrowPoint. This flow line is drawn based on the previously calculated arrowLength, color, angle, and the direction of the swimmer.

Note: the flow lines are composed of 5 individually drawn lines.

void drawAngle()
This method places a representation of the swimmer's angle in the top right corner of the frame. The angle is the same color as the Overlay class's color member. This method also places a box around the drawn angle representation.

void drawGradient()
This method draws the gradient legend beside the angle. This legend is meant to give context to the scale of colors used to draw the arrows and angle representation (i.e. red is bad angle, blue is good).

void makeGradient();
This method generates the gradient when the Overlay class is first constructed. This is a separate method from drawGradient() so that the gradient doesn't have to be re-generated for each frame in the video.

Constant Variables:

const int BOX_OFFSET - This is the number of pixels to horizontally offset the line used to draw the flow and arrows from the corners of the hips and head trackers (i.e. this puts more space between the visualizations and the swimmer).

const float FRONT_MULTIPLIER - This a constant multiplier for the distance from the head and hips.

const int MIN_ARROW_LENGTH - This is the minimum length that the drawn arrows are allowed to be.

const int ARROW_ROC - This is a multiplier for the length of the arrows.

const int ARROW_ADJUST - This is a constant number to subtract from the angle when determining the length of the arrows.

const int MIN_STEP - This is the minimum distance between drawn arrows / flow lines.

const int STEP_SCALE - This is a multiplier for the distance between arrows.

const float SHIFT_SCALE - This is the maximum that the angle can be before the arrows are drawn fully opaque.

const int ARROW_HEAD_SIZE - This defines the size of the arrow head.

const int ARROW_OFFSET - This is another constant to offset the arrows from the line connecting the head and hips points.

const int ARROW_THICKNESS - This defines the thickness of the arrow lines.

const int Z_OFFSET - This is another constant to offset the flow lines from the line connecting the head and hips points.

const int Z_THICKNESS - This defines the thickness of the flow lines
const int Z_FULL = This defines the x offset of the outer-most joints of the flow lines.

const int ANGLE_BOX_THICKNESS - This defines the thickness of the box drawn around the angle representation.

const int ANGLE_THICKNESS - This defines the thickness of the curve of the angle representation.

const int ANGLE_LINE_THICKNESS - This defines the thickness of the lines of the angle representation.

const Scalar Z_LINES_COLOR = - This defines the color of the flow lines.

---------------------
Other files not used
---------------------
findColor was used to decompose the HSV elements of an image. This is not relevant to the final project.

Fishcpp was code found online that would do online detection in an image. It was unreliable for the shape of a swimmer and is not used in any way in the final project.

FRV was the original design of the project. It was based on a threshold image and many image processing techniques. It was very unreliable and abandoned. Nothing carried over to the final project.

Matching was a test bit of code to explore how we could find an image in another image. This is how findMatch works in the final project.

MT was code exploring of the multi tracking object works. The final project does not use the multi tracker but does receive input from the user in a very similar way

TT was a way to have the user draw tracking ROI but it was very bad. Not used in the final project.

**B - Room for Improvement**

The code has been designed to be modular. As described above, tracking and the overlay are broken into their separate parts so they can be changed without affecting the rest of the code. This will make it easy to expand and change later. This current state serves as a proof of concept that shows this is possible. The core elements are in place that prove we can track a swimmer and provide an overlay. How these actions take place can be changed and optimized later with future developments.

**User input**

The original design called for a seamless runtime that would require no extra input from the user other than what video they wanted. This had to be changed early on in the development. The current working version requires the user to select the starting frame of the video and highlight the initial location of the head and the hips of the swimmer. This should be easy to update to make truly contactless a reality. As long as the method generates positional information of the swimmer, the rest of the code should work the same. One way to do this would be machine learning (ML). One method would be unsupervised on only the videos. This option could work but, it seems unlikely. The better way would be to use supervised learning on videos that have labels for where the head and hips are in the frame. Both options would need a lot more data. This would be the most simple use of ML and would remove the need for users to provide the initial head and hips selection. If the ML could also find the best start frame, there would be no user input.

Another way to remove the need for user input is to have a physical marker on the body of the swimmer. This was discussed during development but decided against because of the lack of videos with any markings. This would hypothetically make tracking easier but would require some new technique that would need to be added. This would not necessarily guarantee 100% correctness and still might need the user to do some corrections.

**Performance**

Another big area for improvement is performance. Specifically reducing the runtime of the program so it is more usable in the poolside environment. OpenCV has many algorithms that can be used to do the tracking and detection of an image. These are costly matrix operations that are hard to fully understand. The algorithms that are in place now are worked best during initial development. If more correcting of the tracker were done by using detection on the side, we could use a cheaper tracker that is faster but less reliable. The opposite of that is using a better tracker that is slower but would need to be fixed less. Both of these directions could end up saving time. Some time improvement was done by searching in smaller areas of the frame. This got us some time improvement, but this could be optimized more.

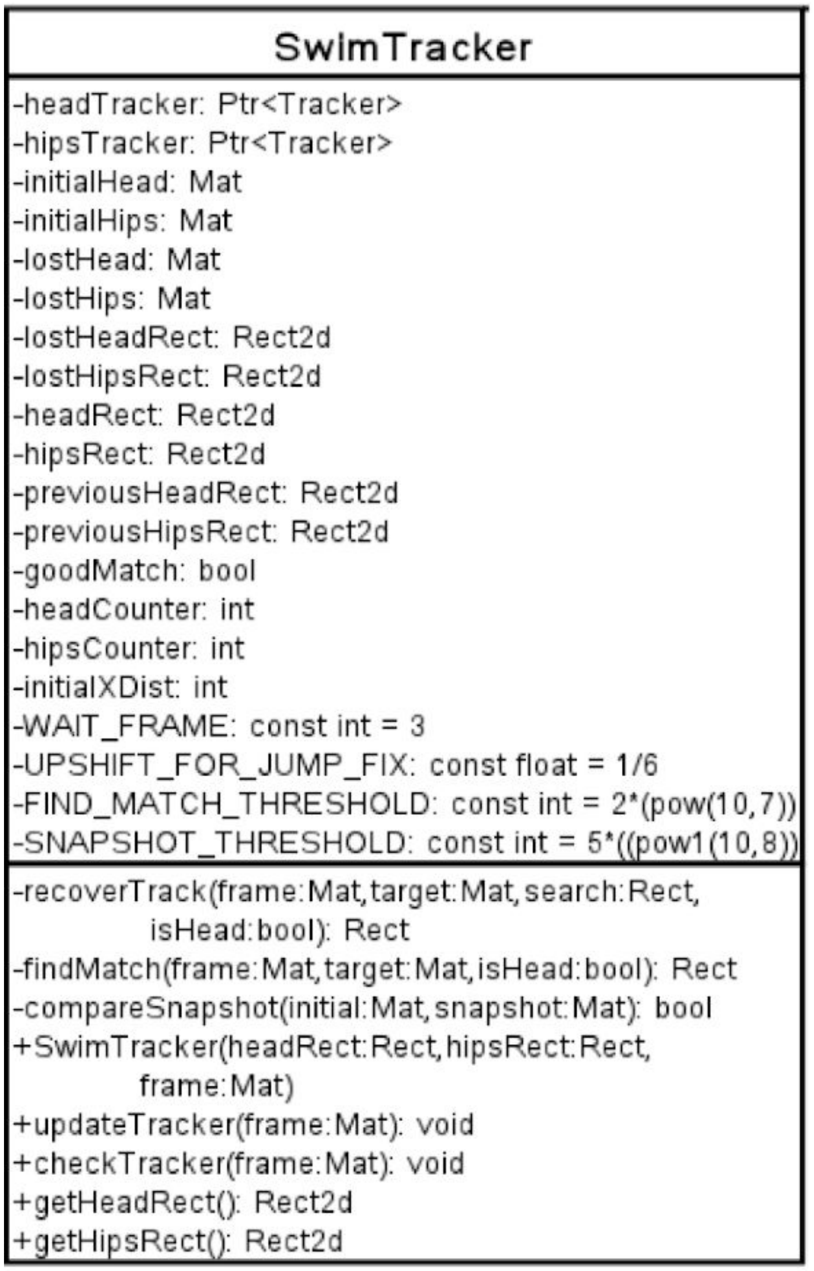
**Fluid model**

Right now, all the math is simple and based on the angle of the swimmer. The arms and the legs are also being ignored. It would be a good next step to expand this to use a better fluids model. This would mean that we could model the swimmer in the water and do some real math based on fluid dynamics. To do this we would need more detailed positional information about the swimmer to feed into the model. We imagine that we would need to bring in more libraries than only OpenCV and we would need much steadier and high-quality videos for this to become a reality. If we were able to do this, we could tune up graphics to see flow information and not just resistance. We could also overlay all the information about speed and drag as well.
Once the code gets to this point, ML could provide more insight into the calculation side. ML could try and determine what type of stroke is in the video and give better feedback for body position based on the stroke. The ML could take in information from the fluids model and compare that to the ideal simmer shape to give feedback. This would expand the use case for the software so that it can help experts get better and not just beginners improve basic form.

**User interface**

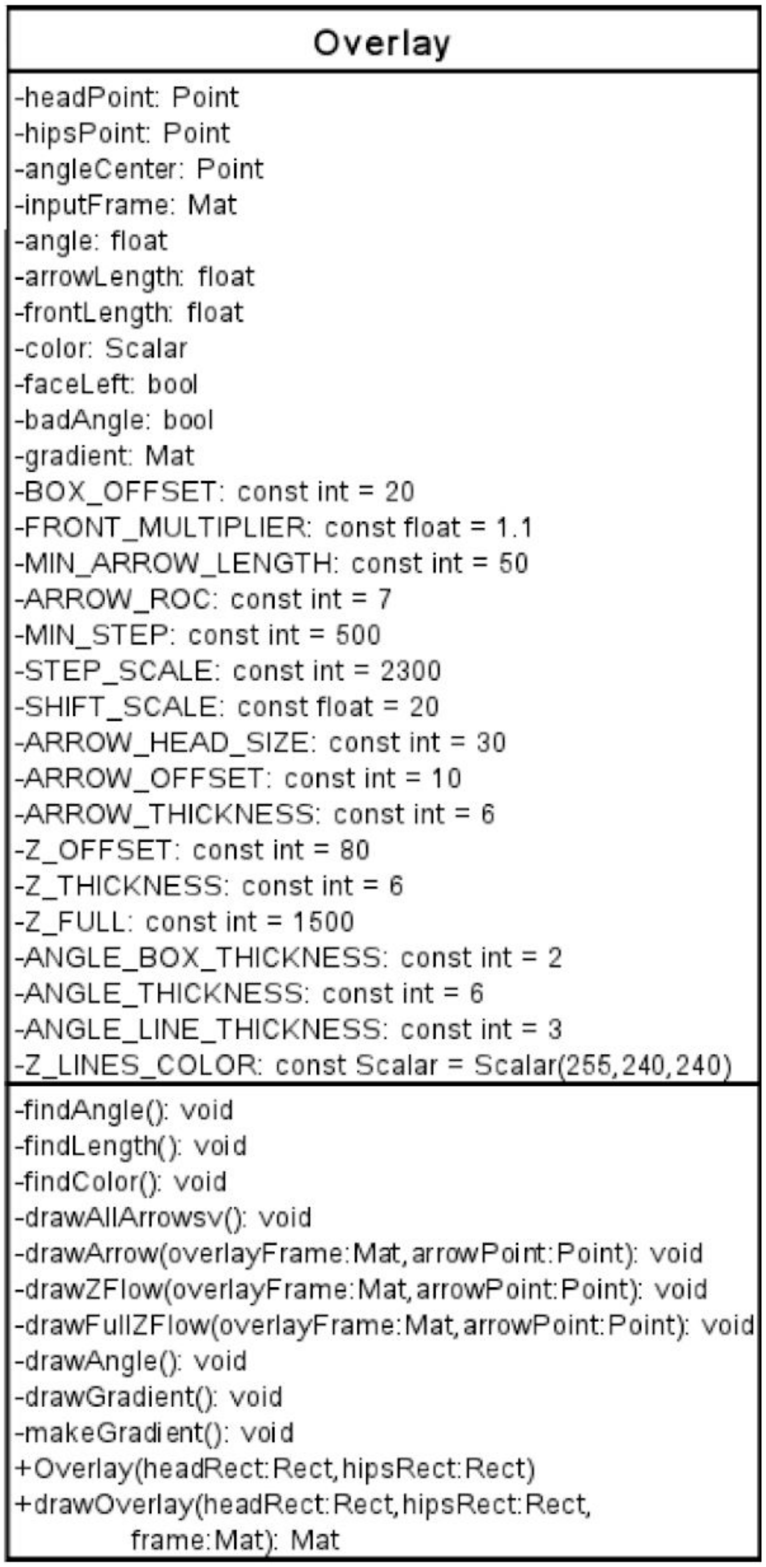
One thing we would have liked to do better on is the UI. This might be unnecessary if there is no longer a need for the user to give an initial input, but as it is now, we can do some tune-ups. We have a basic UI where most of the instructions come over the command line right now. It would be nice to have some better displays for the user when they receive the instructions for how to select the frame and position of the head and hips. We suspect that we would need more than only OpenCV to make the beautiful displays as envisioned. The part that would probably stick around even after removing the need for user inputs is some option for debug mode. This would allow for troubleshooting to see what is happening by displaying the frames as they are being processed. This would be a verbose option that would be selected at runtime. We are doing the same thing now by simply uncommenting the display options and they could be easily linked together to make this debug mode.

**C - Full SwimTracker UML**



```
SwimTracker
-headTracker: Ptr<Tracker>
-hipsTracker: Ptr<Tracker>
-initialHead: Mat
-initialHips: Mat
-lostHead: Mat
-lostHips: Mat
-lostHeadRect: Rect2d
-lostHipsRect: Rect2d
-headRect: Rect2d
-hipsRect: Rect2d
-previousHeadRect: Rect2d
-previousHipsRect: Rect2d
-goodMatch: bool
-headCounter: int
-hipsCounter: int
-initialXDist: int
-WAIT_FRAME: const int = 3
-UPSHIFT_FOR_JUMP_FIX: const float = 1/6
-FIND_MATCH_THRESHOLD: const int = 2*(pow(10,7))
-SNAPSHOT_THRESHOLD: const int = 5*((pow1(10,8))
-recoverTrack(frame:Mat,target:Mat,search:Rect,
              isHead:bool): Rect
-findMatch(frame:Mat,target:Mat,isHead:bool): Rect
-compareSnapshot(initial:Mat,snapshot:Mat): bool
+SwimTracker(headRect:Rect,hipsRect:Rect,
             frame:Mat)
+updateTracker(frame:Mat): void
+checkTracker(frame:Mat): void
+getHeadRect(): Rect2d
+getHipsRect(): Rect2d
```

## D - Full Overlay UML



```
Overlay
-headPoint: Point
-hipsPoint: Point
-angleCenter: Point
-inputFrame: Mat
-angle: float
-arrowLength: float
-frontLength: float
-color: Scalar
-faceLeft: bool
-badAngle: bool
-gradient: Mat
-BOX_OFFSET: const int = 20
-FRONT_MULTIPLIER: const float = 1.1
-MIN_ARROW_LENGTH: const int = 50
-ARROW_ROC: const int = 7
-MIN_STEP: const int = 500
-STEP_SCALE: const int = 2300
-SHIFT_SCALE: const float = 20
-ARROW_HEAD_SIZE: const int = 30
-ARROW_OFFSET: const int = 10
-ARROW_THICKNESS: const int = 6
-Z_OFFSET: const int = 80
-Z_THICKNESS: const int = 6
-Z_FULL: const int = 1500
-ANGLE_BOX_THICKNESS: const int = 2
-ANGLE_THICKNESS: const int = 6
-ANGLE_LINE_THICKNESS: const int = 3
-Z_LINES_COLOR: const Scalar = Scalar(255,240,240)
-findAngle(): void
-findLength(): void
-findColor(): void
-drawAllArrowsv(): void
-drawArrow(overlayFrame:Mat,arrowPoint:Point): void
-drawZFlow(overlayFrame:Mat,arrowPoint:Point): void
-drawFullZFlow(overlayFrame:Mat,arrowPoint:Point): void
-drawAngle(): void
-drawGradient(): void
-makeGradient(): void
+Overlay(headRect:Rect,hipsRect:Rect)
+drawOverlay(headRect:Rect,hipsRect:Rect,
            frame:Mat): Mat
```

## E - Alternate logo not used