Deep Breath

Breath Pattern Detection with Intel 3D Camera

Nili Furman

Supervised by

Alon Zvirin

Yaron Honen

Based on

Realtime Breathing

Technion | Israel Institute of Technology

February 06, 2021

Abstract

Chest motion and respiratory volume abnormalities or sporadic breathing rate are often associated with thorax diseases hidden under the human eye radar. Those can range from light and passing conditions to fatal and lifestyle affecting illnesses, which in many cases stay unnoticeable or falsely diagnosed.

Current medical methods of chest motion abnormalities detection often rely on human medic eye observation which might be less accurate and comprehensive as computer vision, and often lead to mistreatment of patients who are unable to be physically present at the examination location due to various reasons: handicap, privacy or mostly - availability.

Our work in the GIP Lab in the Technion Institute strives to provide an innovative, precise and accessible-to-all tool to detect chest motion and breathing abnormalities as seeked by medical doctors and provide the findings as fast as possible, using a 3D camera.

Deep Breath - Breath Rate, Volume and Chest Motion Analyzer

Behind the Scenes

Human Breath

Human breath varies between children and adults, healthy and ill and many other characteristics, thus inspection must be adjusted. For example, a child’s breath rate is notably faster than adult’s one, and healthy breath patterns are different from abnormal, distinguishable by chest movement, volume, breath rate and even rhythm.

Tools and Developing Environment

Let us present the tools and environments used to develop the app. The following are the physical and virtual tools used, along with the relevant coding libraries and toolkits.

Intel Realsense Depth Camera D435

We used the Intel Realsense Depth Camera D435 as our 3D camera. It is a simple and not expensive tool that can be installed and used easily by all. This version of depth camera was introduced by Intel in January 2018, and provides best-in-class depth resolution, quality RGB, and high frame rate. For our purposes, it provides a steady frame rate of about 25 fps at 1280x720 pixel resolution in real-time, more than enough to detect subtle chest movements.

Intel Realsense SDK 2.0 $_{\textit{v.2.41.0}}$

The software development kit is open-source C++ code and available on Intel Realsense GitHub. It provides a convenient way to access the camera streams (depth and color), visualize them and configure the camera settings. Most of these features are provided in the **librealsense2** library (which is open source, as stated) and in the set of examples published by Intel Realsense.

Microsoft Visual Studio 2019

Microsoft Visual Studio is an IDE developed by Microsoft. It uses Microsoft software development platforms, has a code editor and an integrated debugger that serves both for source-level and machine-level. It also has Git support and provides various toolkits based on programming language and development purposes.

CMake

CMake is a cross-platform free and open-source software tool for managing the build process of software using a compiler-independent method. It supports directory hierarchies and applications that depend on multiple libraries. It is used in conjunction with native build environments such as Make, Qt Creator, Ninja, Apple's Xcode, and Microsoft Visual Studio. It has minimal dependencies, requiring only a C++ compiler on its own build system. (Wikipedia)

We used CMake in order to create a solution containing the open-source code of Intel Realsense, which provided us with a set of examples.

C++17

As mentioned above, the programming language used for coding is C++. Few changes were made to the C++ Standard Template Library, although some algorithms in the <algorithm> header were given support for explicit parallelization and some syntactic enhancements were made. (Wikipedia)

Visual Studio 2019 supports almost all of C++17.

Qt 5.15.1 *build tools:* ***mcsv 2019***

Qt is an innovative and developer friendly library which is available for open source uses. It provides a convenient set of tools to create and design cross-platform (and particularly Windows) applications that provide great user experience.

It also makes a great emphasis on performance and responsiveness, and thus, it is very suitable for real-time 3D applications.

Qt is also widely used and therefore is generously documented and has plenty of examples and additional free to use tools, such as the plotting widget used in the app.

Qt Plugin for Visual Studio was used as well.

OpenCV 4.3.1

OpenCV is an open-source library of programming functions mainly aimed at real-time computer vision, originally developed by Intel. It is written in C++ and its primary interfaces are with C++, but there are bindings to other languages as well.

If the library finds Intel's Integrated Performance Primitives on the system, it will use these proprietary optimized routines to accelerate itself. (Wikipedia)

We use OpenCV for frame processing: Color detection, thresholding and connected components algorithm, to find the stickers in the frame.

QCustomPlot

QCustomPlot is a Qt C++ widget for plotting and data visualization. It has no further dependencies and is well documented. This plotting library focuses on making good looking, publication quality 2D plots, graphs and charts, as well as offering high performance for realtime visualization applications. (QCustomPlot website)

plog ***by Sergey Podobry***

Plog is a C++ logging library that is designed to be as simple, small and flexible as possible. It is created as an alternative to existing large libraries and provides some unique features as CSV log format and wide string support. (plog GitHub)

Algorithm

The main goal of the app is analysis of chest movement and extraction of the BPM (breaths per minute) rate or breath volume change information during realtime video streaming, or from a pre-recorded file.

Breath pattern detection is achieved by applying image processing methods on sequences of color and depth frames.

The app is multithreaded and runs three major groups of threads:

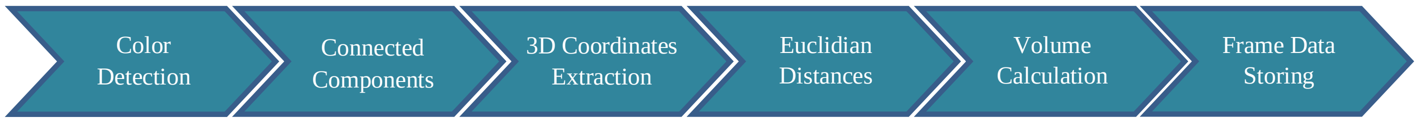
- GUI related (managed by Qt and OS)
- Main thread
- Frame polling and processing

As the latter holds the major algorithm implementation, we shall mainly discuss its logic in this section. Other thread interactions will be discussed in the Code Overview section.

The app's frame polling and processing loop awaits a user's request to run on a certain stream (either from an existing file or from the camera). After such a request is received, depth and color frames are polled using the device's wait mechanism in each iteration of the thread's loop, providing the synchronization objects allow it.

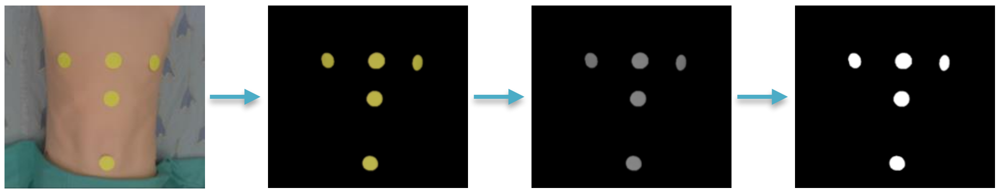
The frames are then aligned (`rs2::align::process(rs2::frameset), rs2_stream::RS2_STREAM_COLOR`) to its color frame to preserve its resolution. The two matched frames (a depth frame and a color frame) are then processed and a BPM (breaths per minute) measurement is performed. In addition, when the application starts, an array of 256 slots is initialized. This array, hereinafter referred to as the *frames array*, stores the data extracted from frames processed by the application.

Frame processing



1. Color detection

Color detection is done using the OpenCV library. The color frame matrix is screened by OpenCV's `inRange` method, to only retain values in a preset range. The range used corresponds to the color of stickers, as defined in the configuration file. The resulting matrix is then transformed to grayscale, and sequentially to a binary matrix, using a preset threshold.



2. Connected components

Connected components in the binary matrix are recognized by OpenCV's `connectedComponentsWithStats` method, which detects the connected components and returns their centroids. If areas in the background have colors similar to the stickers' color, these areas may appear white in the binary matrix and might be included in the set of connected components returned by `connectedComponentsWithStats`. The connected components returned are screened by an area threshold in order to mitigate this kind of noise. The threshold used is 50% of the area of the connected component with maximal size. Therefore, if large areas in the background appear in colors in the same range as the stickers, the application will not be able to screen them, and will produce unreliable results.

If the number of valid connected components found in a frame is lower than the number of stickers expected by the application, the frame will be discarded. Otherwise, each sticker is attributed with the corresponding connected component according to the component's center coordinates (x, y) and the stickers assumed alignment. The location of a sticker is defined as the center of the corresponding connected component.

3. 3D coordinates extraction

Using Intel's `depth_frame::get_distance` method, we extract the depth of each sticker's location (a pixel with x and y coordinates). Then, the `rs2_deproject_pixel_to_point`[1] method is used to extract the stickers' 3D coordinates in centimeters, based on their 2D pixel coordinates and their depth.

Under no special treatment, at times, the depth information for a certain area of the frame is absent (may be caused by non-optimal lighting or insufficient distance from the camera). This could be previously seen in Realtime Breathing: In this case, the 3D coordinates returned are invalid. When then dimension set in the configuration file is 3D, frames with invalid 3D coordinates are discarded. When then dimension is set to 2D, such frames are retained, since in this setting, the 3D coordinates are not used in further analysis.

Since we wish to achieve a sufficient set of data as possible for the volume measurements, in Deep Breath we apply an interpolation filter to the depth frame provided by Intel `rs2::spatial_filter` with `rs2_option::RS2_OPTION_HOLES_FILL` to approximate the missing coordinates.

4. Euclidian distances

The 3D Euclidian distance between every pair of stickers is calculated in centimeters, and the 2D Euclidian distance is calculated in units as defined in *2D measure units* in the configuration file (either pixels or cm). Additional frame metadata is extracted, such as color frame and depth frame timestamps (provided by the device). In the last stage of the process, the frame is tested to see if it is a duplicate of the previous frame: under 2D configuration, the test is based on the

[1] The usage of the `rs2_deproject_pixel_to_point` method referenced from rs-measure example provided by intel.

color timestamp, whereas per 3D configuration, it is based on both the color and the depth timestamps. If a frame has proven to be a duplicate, it is discarded.

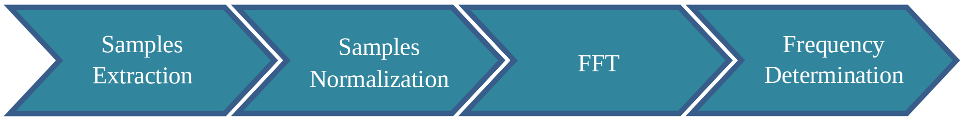
5. Volume Calculation

This stage is done in the volume calculation mode of the application. When all relevant locations are calculated in 3D, the volume is calculated based on its desired type: Either the volume of the tetrahedron formed by the three markers (as described further in this paper), or the Reimann Sums of surface trapped in the bounding box formed from the three markers.

6. Frame Data storing

After frame processing has finished, the frame **data** is stored in the next free cell of the frames data array. If there are no free cells, the new frame data will replace the oldest frame data in the array. This array storage method is cyclic, providing us efficiency in time and memory economical access to the data for further purposes.

BPM measurement



1. Samples extraction

Samples are extracted from the frames data array – each frame produces one sample. The value of a sample is the average distance (either 2D average distance or 3D average distance) or volume as calculated in the frame processing stage, according to the dimension and mode chosen. The pairs of markers participating in the average distance calculations are as defined in the *distances* section of the configuration, while the volume is calculated based on three markers and the volume type chosen, as explained further. The samples are given in the order of arrival of the corresponding frames.

2. Samples normalization

The samples are normalized and shifted to [-1, 1] range. We observed that with the addition of the normalization step, the application produces results significantly more reliable.

3. FFT

The normalized samples are processed by the FFT algorithm, to produce the components of the different frequencies which constitute the discrete signal.
The FFT implementation used is Alexander Thiemann's FFT Gist.
As this implementation requires the number of samples to be a power of 2, samples are padded with zero values when needed.

4. Frequency determination

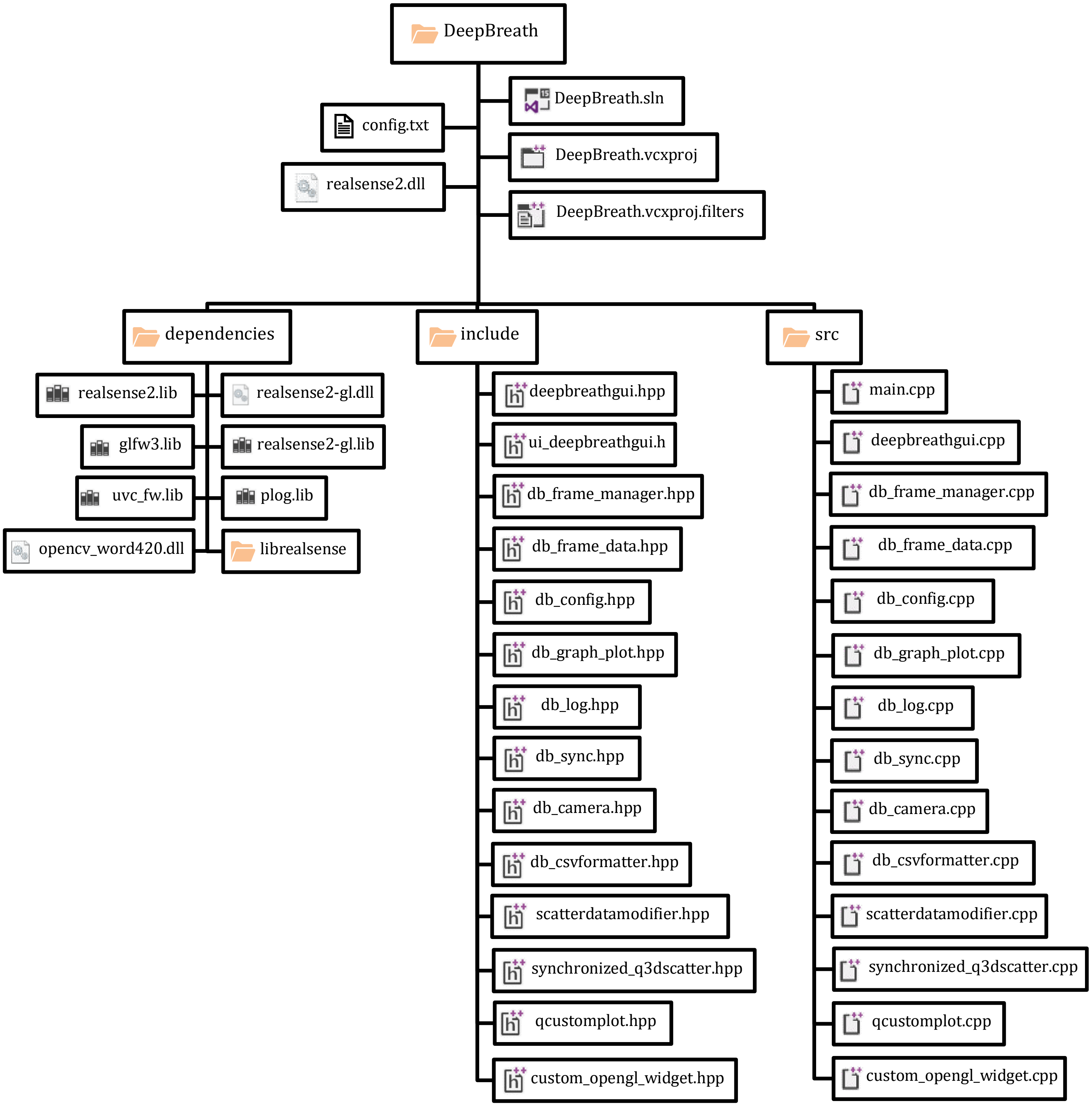
With the frequency components in hand, the frequency with the most dominant component is chosen, using the following formula to determine a component's size: $sqrt(Real^2 + Imaginary^2)$.
In order to screen low frequencies (which tend to hold large values), all frequencies corresponding to a BPM value lower than 5, are filtered out, taking under consideration that the breath rate of human beings is greater than 5.
The BPM value returned at each iteration is **60 * frequency**.

Correctness

The above-described algorithm (average distance or volume extraction per frame ⇢ samples normalization ⇢ FFT ⇢ dominant frequency and BPM determination) was tested in two different implementations for the average-distances-based calculation, one in C++ and the other in MATLAB. The resulting locations and distances as well as frequency and BPM values were roughly identical. In both cases the results were compared with manual measurements. Testing has proven the algorithmics to give reliable results with an average error (that is, the ration between manual and computer measurements) of less than 7%.
The correctness of the volume component addition was tested by comparison of previous known breath rates to distances calculations and was found to produce an average error of less than 1% compared to the distances-based calculations results on samples with ideal environment.
Nevertheless, it is important to state that although the breath rate and chest motion patterns are preserved in the volumetric method, there is no concrete indication of what the actual volumes calculated stand for, except for the differences between them which indeed indicate real change in chest volume (not necessarily lung volumes or air flow volume – the question of the connection between these should be forwarded to the medical team for further information) in cubic centimeters.

Solution and Code Overview

The code package is a standalone and contains all required resources for further development.



DeepBreath – Root Directory

The complete code package is located inside the DeepBreath folder. The root directory contains:

- *DeepBreath.sln* – Visual Studio 2019 solution file containing the project.
- *DeepBreath.vcxproj* – Visual Studio project file.
- *DeepBreath.vcxproj.filters* – Visual Studio project filters file of the project.
- *realsense2.dll* – Dynamic link library of the RealSense2 functionalities the project uses.
- *config.txt* – The config file the app parses (explained in the app's *Overview* section).
- src – Folder containing **our** source code.
- include – Folder containing **our** headers and Intel's provided header example.hpp.
- dependencies – Folder containing all libraries, sources, and headers our code uses.

src – Our Source Code

main.cpp

Contains the code to run the app and the frame polling thread.

Outline and main attributes:

I. `main()` function:

Qt App Initialization

1. Initialization of the app's window with relevant Qt objects.

Our Data Structures Initializations

2. Initialization of the Camera and FrameManager objects.

Polling Thread Initiation and Join

3. Thread initiation with a reference to the UI object passed as argument.
4. GUI show and application execution.
5. Disabling of the frame polling with booleans and condition variables.
6. Joining thread.

II. `poll_frames_thread()` thread function:

Initiation of Spatial Filter

1. Initialization of the spatial filter for holes filling. (Done only once as the thread is created in the main function)

Active Check Loop

2. Check if the app is still active.
3. Wait for the condition variable of frame polling to be notified.

Polling Loop

4. Get the Camera object instance and use it to fetch the frameset using a wait mechanism.
5. Align frameset to color frame to preserve resolution.
   Get color and depth frames.
6. Get the FrameManager object instance to process the frames that arrived.
   The FrameManager gets as argument the depth frame with the spatial filter applied to fill holes.
7. Render the frames.
8. Render the scatter view.
   To prevent a bottleneck (resulting in read access violation), scatter samples are rendered only once in two frameset arrivals (given they are not dumped, as mentioned earlier in the *Algorithm* section).
9. Calculate BPM.
   Render BPM and FPS on screen.
10. Log all newly collected data.

---

deepbreathgui.cpp

Source code of the GUI handlers implementation.

It is important to note that the code that handles user selections updates in the Config class, starts or stops camera streams (along with updating the synchronization objects) and resets the FrameManager and Graph objects are implemented within this file, in the relevant button click handlers.

db_frame_manager.cpp, db_frame_data.cpp

Source codes of the classes used in out algorithm, discussed below in the *Data Structures* section.

db_config.cpp, db_camera.cpp, db_graph_plot.cpp, db_log.cpp, db_csvformatter.cpp, db_sync.cpp

Source codes of other classes used for streaming and data visualization and logging based on user's selections, discussed further in the *Data Structures* section.

scatterdatamodifier.cpp, synchronized_q3dscatter.cpp, qcustomplot.cpp, custom_opengl_widget.cpp

Implementations of custom widgets of the GUI and their helpers.

include – Our Headers

deepbreathgui.hpp

Header file of the GUI handlers.

ui_deepbreathgui.h

A header defining and instantiating the UI and all its components (and widgets, along with custom ones). It was originally auto generated by the Qt tool for Visual Studio based on the basic form created in a Qt project in Qt Creator. Later in the development process, it was modified manually to support more complicated custom widgets without going through additional integration processes.

db_frame_manager.hpp, db_frame_data.hpp

Headers of the classes used in out algorithm, discussed below in the *Data Structures* section.

db_config.hpp, db_camera.hpp, db_graph_plot.hpp, db_log.hpp, db_csvformatter.hpp, db_sync.hpp

Headers of other classes used for streaming and data visualization and logging based on user's selections, discussed further in the *Data Structures* section.

scatterdatamodifier.cpp, synchronized_q3dscatter.cpp, qcustomplot.cpp, custom_opengl_widget.cpp

Headers of custom widgets of the GUI and their helpers.

dependencies

Contains all relevant libraries required for the project, plog code (both headers and sources), and all Intel RealSense (librealsense) relevant headers in their original hierarchy.

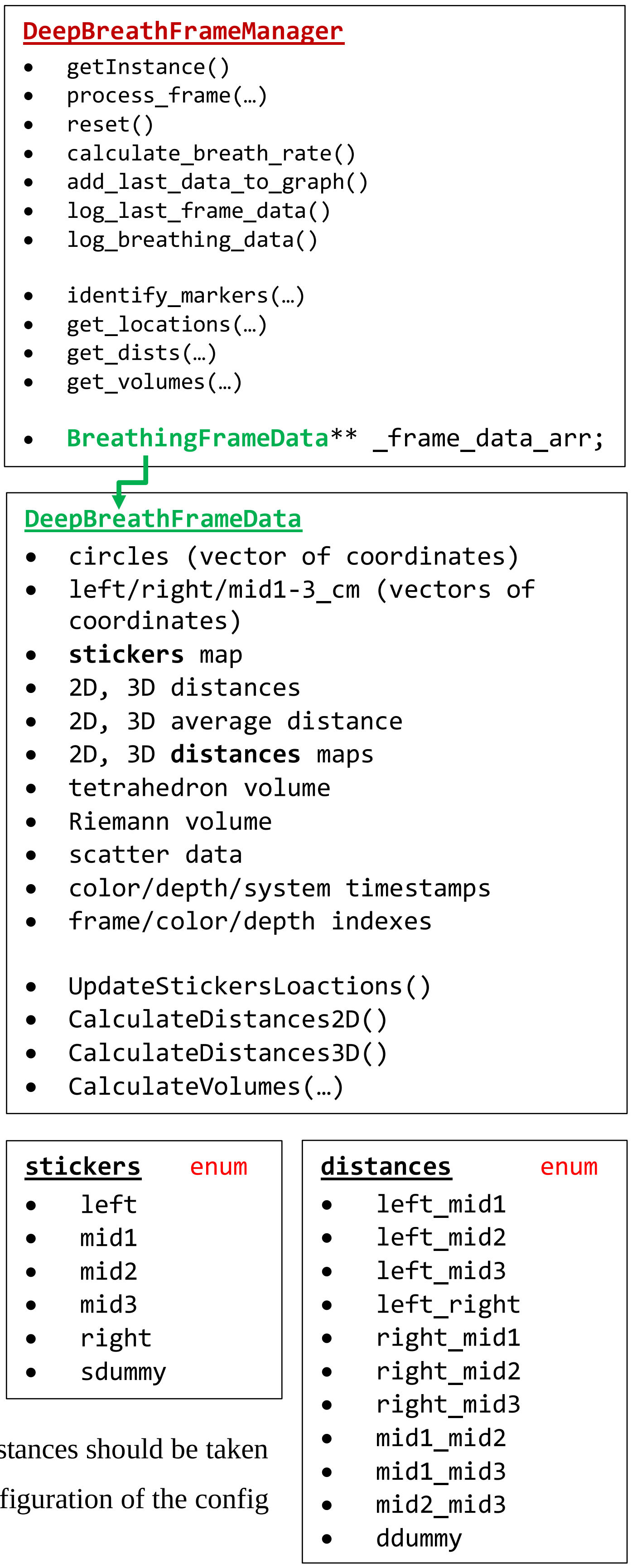
Data Structures

DeepBreathFrameManager

A singleton created and initialized once in the main code, and responsible for all frame managing: frame processing, frame storing, reset. In addition, it fetches relevant data for graph plotting and logging.

DeepBreathFrameData

Stores all data extracted from the frame: centroids of the stickers in pixels (‘circles’), coordinates in centimeters of all stickers, 2D and 3D calculated distances as well as the average of the chosen distances by user (and stored in Config class), calculated volume in the chosen method and the scatter data used for calculations, timestamps of the frame and time of the system clock by the frame arrival and frame index. This class is responsible for the update and calculation in practice, by its public methods, of the stickers’ locations, 2D and 3D distances and averages and volumes – when called by *DeepBreathFrameManager* in `process_frame()`. In addition, this class maintains a stickers map matching to each considered sticker the corresponding coordinates vector, and both 2D and 3D distances maps, matching to each considered distance the corresponding distance. These maps serve as indicators to which stickers and distances should be taken into account in calculations, according to the configuration of the config file.


```
DeepBreathFrameManager
•  getInstance()
•  process_frame(…)
•  reset()
•  calculate_breath_rate()
•  add_last_data_to_graph()
•  log_last_frame_data()
•  log_breathing_data()

•  identify_markers(…)
•  get_locations(…)
•  get_dists(…)
•  get_volumes(…)

•  BreathingFrameData** _frame_data_arr;

DeepBreathFrameData
•  circles (vector of coordinates)
•  left/right/mid1-3_cm (vectors of
   coordinates)
•  stickers map
•  2D, 3D distances
•  2D, 3D average distance
•  2D, 3D distances maps
•  tetrahedron volume
•  Riemann volume
•  scatter data
•  color/depth/system timestamps
•  frame/color/depth indexes

•  UpdateStickersLoactions()
•  CalculateDistances2D()
•  CalculateDistances3D()
•  CalculateVolumes(…)

stickers      enum
•  left
•  mid1
•  mid2
•  mid3
•  right
•  sdummy

distances     enum
•  left_mid1
•  left_mid2
•  left_mid3
•  left_right
•  right_mid1
•  right_mid2
•  right_mid3
•  mid1_mid2
•  mid1_mid3
•  mid2_mid3
•  ddummy
```

DeepBreathConfig

A singleton storing the user selections from the GUI. It also parses data from the config file, which provides a default selection on app startup. Created and initialized once upon UI object construction (`QDeepBreath::initDefaultSelection()`) and used in various cases by other classes when data from the config file needs to be accessed.

DeepBreathConfig

- `createInstance(…)`
- `getInstance()`
- **`dimension`**
- **`Graph mode`**
- **`stickers`** `included boolean map`
- **`distances`** `included boolean map`
- **`sticker_color`**
- **`volume_type`**
- `is_stickers boolean`

sticker color `enum`

- `YELLOW`
- `BLUE`
- `GREEN`
- `RED`

graph mode `enum`

- `DISTANCES`
- `LOCATION`
- `FOURIER`
- `NOGRAPH`

volume type `enum`

- `TETRAHEDRON`
- `REIMANN`

dimension `enum`

- `D2`
- `D3`

DeepBreathGraphPlot

Singleton created on stream start, manages graph plotting of the data in the app.

It updates the `QCustomPlot` graph widget every iteration of frame processing (if values are valid).

It gets its parameters from the *DeepBreathConfig* class, and *DeepBreathFrameManager* calls its `addData(…)` with relevant data passed as parameters for plotting.

It is recommended to check QCustomPlot documentation to understand better how this object works.

DeepBreathGraphPlot

- `createInstance(…)`
- `getInstance()`
- `reset()`
- `addData(…)`

- `graph_widget ptr`
- `First plot Boolean`
- `x,y min/max values for axes scaling`
- `min volume for better visualization of volume change`

DeepBreathLog

Global object used for measurements results logging into a .csv file.

It uses the plog library along with a custom CSV formatter *DeepBreathCSVFormatter* (see plog's documentation for more information) and logs only relevant data according to the user's selections as they are saved in *DeepBreathConfig*.

DeepBreathLog

- `init(…)`
- `stop()`

DeepBreathCamera

Singleton holding all Realsense objects handling the camera stream in one organized place.

Initialized once in the main function just after the initiation of the app.

```
DeepBreathCamera
•  getInstance()
```

DeepBreathSync

Synchronization global object.

Used for intactness of the streaming operations in the multithreaded environment.

Mutexes are needed to prevent the access to the condition variables in the critical section from other parts of the code, and therefore prevent race conditions and deadlocks during the frame polling state changes (i.e., stopping the camera).

```
DeepBreathSync
•  std::condition_variable cv_poll_frame
•  std::mutex m_poll_frame
•  std::condition_variable
   cv_end_poll_frame
•  std::mutex m_end_poll_frame
•  std::atomic<bool> is_poll_frame
•  std::atomic<bool> is_active
•  std::atomic<bool> is_end_poll_frame
```

As the condition variables are notified, the polling code can proceed based on the booleans states. Two major flags are used for the frame polling: `is_poll_frame` and `is_end_poll_frame`. The first allows to start every iteration of the frame polling and processing, while the latter indicates whether the procedure had fully ended – to allow starting and closing streams on user's operations without harming the app's state.

Finally, the `is_active` flag is required for thread killing on app exit.

Additional Information

1. Make sure your OpenCV environment variable is the same as used in the project: OPEN_CV. Change it either in your system or in the project if it is different.
2. Please compile the project only in release mode. For debug mode, a few properties must be set, according to the missing libraries. We chose not to support debug mode because running the app in debug mode often does not reflect its behavior in practice (for example, too long waiting time and too few processed frames for the calculations). However, it is possible to configure the debug mode, by setting the correct paths to the required libraries.
3. Deployment: Qt had a lot of dependencies and therefore requires deployment for a standalone application. It can be done on build by setting *Run Deployment Tool* to *Yes* in the

Qt Project Settings under *Configuration properties* in Visual Studio, or by running the following command in the command line after build:

```
C:/Qt/5.15.1/msvc2019_64/bin\windeployqt.exe" --list target –dir
release\ C:\git\DeepBreath\release\DeepBreath.exe
```

4. Please refer to the README files for exact steps of the environment setup.

The Windows App

Before We Start

To enable proper usage of the app features, there are a few things that must be set up before launching the app.

Intel RealSense 3D Camera

Intel RealSense D435 Depth Camera driver must be installed. Most concurrent computers that run an updated operating system might manage to find the driver online and install it on its own, but it is recommended to download it from Intel's download center (leads straight to the relevant page).

Libraries (lib) and Dynamic Link Libraries (dll)

The app requires a set of libraries to run appropriately. Please make sure all required libraries are installed and located in the same path as the DeepBreath.exe file:

realsense2.dll
opencv_world451.dll
All Qt5 dll's (including in the subfolders)

Please note that it is possible that your system is missing a few more Microsoft DLLs, such as:

MSVCP140.dll
CONCRT140.dll
VCRUNTIME140.dll

You will be notified about them when starting the app, and in this case, you should install Microsoft Visual C++ 2015 Redistributable[2].

Config File

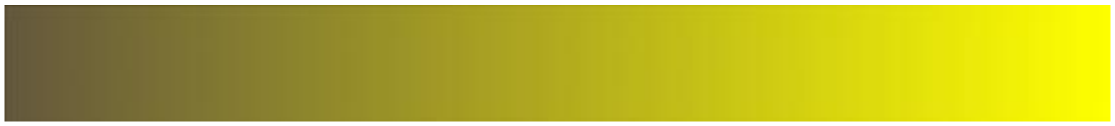
The app requires a config.txt file to run. This is a file containing the default configurations to use when running the app, written in a certain template to allow the app an intact parsing of the configurations chosen. The structure and usage of this file will be discussed widely in the app overview section.

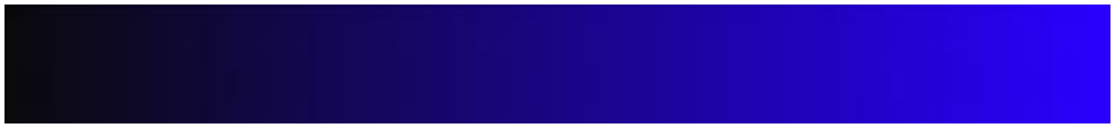
[2] Download and follow the instructions here.

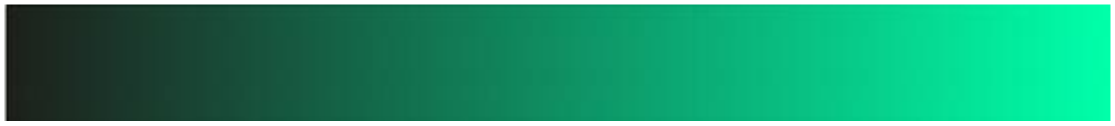
Stickers

The app detects stickers of a color chosen amongst yellow, blue, and green. For best performance it is recommended to use yellow stickers (as explained in the config and results sections).
The tones of the colors chosen can vary, but it is recommended to use sufficiently saturated and vibrant variations of the color chosen.

For your assistance, the following are (roughly) the ranges of colors that can be detected. Note that changes in screen colors projection may affect the perceived colors by the eye:


Yellow range gradient


Blue range gradient


Green range gradient

Background and Lighting

As mentioned, the app uses color detection to detect the stickers. Thus, it is very important to place the camera facing a background as uniform as possible, containing as few colors as possible and having a completely different hue from the stickers' color, preferably negating it. In addition, the stickers' detection might be affected by lighting conditions. Poorly lit scenes or concentrated light directly facing the stickers might affect their perception by the camera lens and cause misdetection leading to faulty results. Therefore, it is recommended to use well-lit spaces (preferably by white neutral light), sufficiently illuminating the patient and stickers while keeping the lighting low enough to preserve vision of color.

Inside the App - Overview

Config File

The app allows choosing different default configurations, as defined in the config.txt file. This file must be present in the same folder as the app's .exe file for the application to run properly. If the config file is not in the proper format, the behavior will be undefined.

Configuration may be changed by the user by opening the file (double-click) and changing the required fields. Following is an example of a properly defined config file:

```
#dimension: 2 for 2D or 3 for 3D. recommended: 2
2
#mode: D for distances, L for location, F for fourier, V for volume, N for no graph
D

#distances: set to y to include a certain distance. set to n otherwise. recommended:
left-mid3, right-mid3, mid2-mid3
left-mid1 n
left-mid2 n
left-mid3 y
left-right n
right-mid1 n
right-mid2 n
right-mid3 y
mid1-mid2 n
mid1-mid3 n
mid2-mid3 y

#location: set to y to track the location of a certain sticker. set to n otherwise
left n
mid1 n
mid2 y
mid3 y
right n

#number of stickers: 3, 4 or 5
4

#color of stickers: Y (yellow), B (blue) or G (green). recommended: Y
Y

#2D measure units: cm or pixel. recommended: pixel
pixel

#use stickers:
y

#volume calculation type(for V mode only): T for tetrahedron volume, R for reimann sums
R
```

Only `text highlighted in green` in the above example should be changed by the user.

The effect of each field in the configuration file is as follows:

- **Dimension**

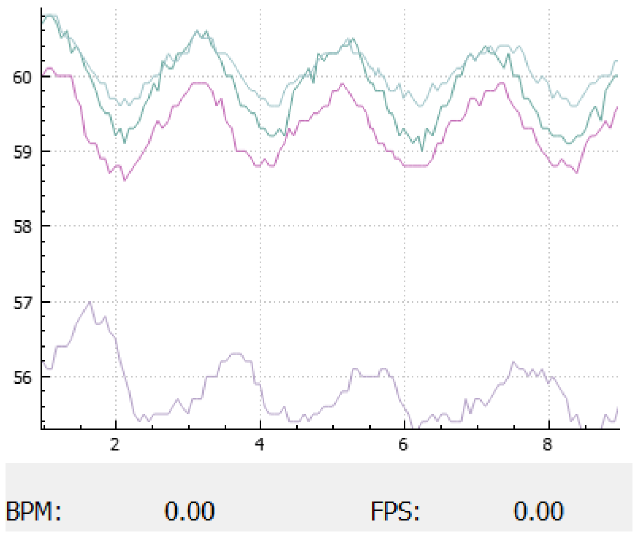
```
#dimension: 2 for 2D or 3 for 3D. recommended: 2
2
```

When measuring the distance between two stickers, the application may be set to measure the distance using 2 dimensions (data received only from color video), or 3 dimensions (taking depth in account as well). For the purpose of BPM measuring, 2D is recommended for better accuracy. In the example above the dimension is set to 2.
Permitted values: 2, 3.

- **Mode**

```
#mode: D for distances, L for location, F for fourier, V for volume,
N for no graph
D
```

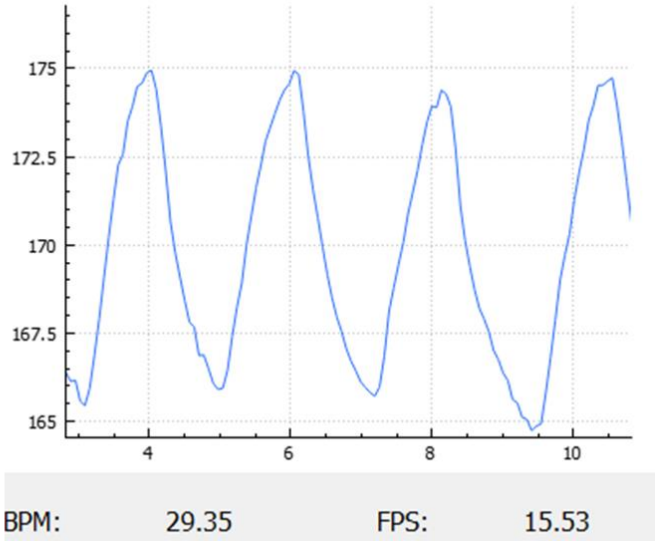
The app can be configured to 5 different modes, each displaying different output.
L – location mode: While in location mode, the application does not output a BPM value. When streaming from a live stream or from a file, a graph is generated, tracking the depth coordinate of different stickers in each frame. The stickers to be displayed in the graph are configured in the *location* field (follows below).



Example graph generated in L mode. In this example, 4 stickers were configured to y in the location field. The depth of each sticker, related to the camera location, in each frame, is displayed against the according timestamp. No BPM calculation.

In D, F, V and N modes the application measures BPM by tracking the change in the average distance between stickers, using the same method, and presented to the user in real time. However, in each mode, a different graph is generated alongside the BPM value.

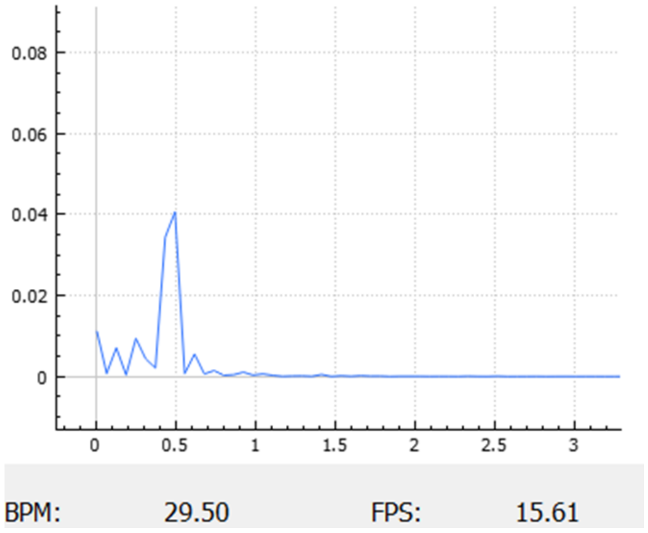
D – distances mode: When streaming from a live stream or from a file, a graph is generated, tracking the average distance between different stickers in each frame. The distances to be displayed in the graph (as well as taken in account for BPM measurements) are configured in the *distances* field (follows below).



BPM: 29.35 FPS: 15.53

Example graph generated in D mode. The average distance in each frame is displayed against the according timestamp.

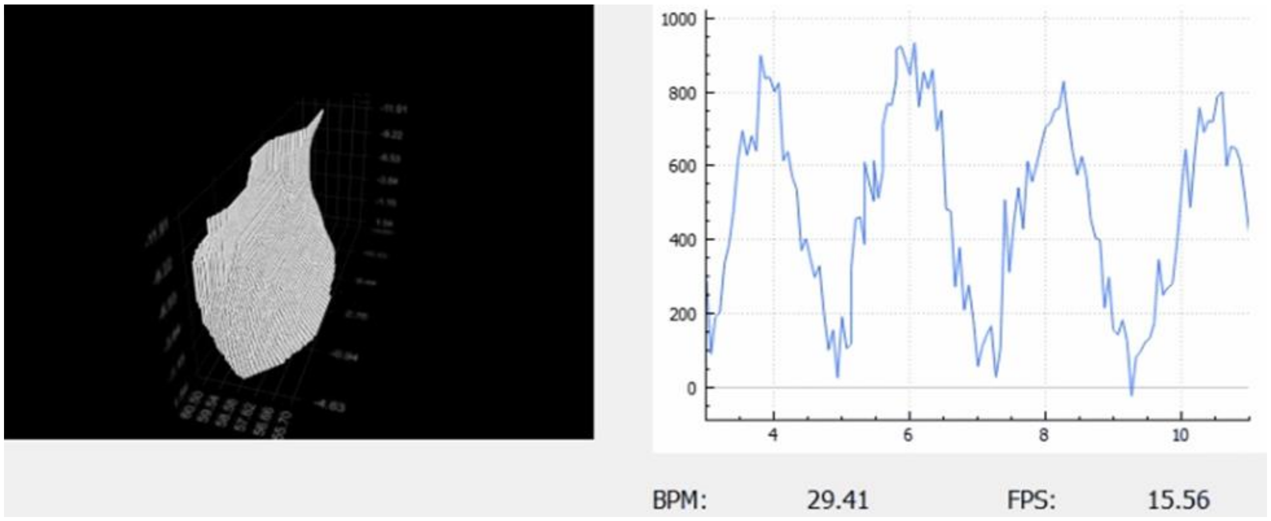
F – fourier mode: When streaming from a live stream or from a file, a graph is generated, displaying the values received by fft for each frequency, using the formula $Re^2 + Im^2$ for each frequency (Re and Im stand for the real part and the Imaginary part in accordance). Each iteration, the fft algorithm is applied to the last 256 samples collected, each sample containing the average distance measured in a frame. The distances taken in account in the samples are configured in the *distances* field (follows below).



BPM: 29.50 FPS: 15.61

Example graph generated in F mode. The value of each frequency is displayed in each iteration.

V – Volume mode: When streaming from a live stream or from a file, a graph is generated, tracking the volume of either the tetrahedron formed by 3 valid stickers in each frame or the surface tamed in the bounding box of the stickers using Reimann sums. The volumes that are displayed on the graph are taken in account for BPM measurements in this mode. In addition, a scatter of the points participating in the calculations is visualized as well.



BPM: 29.41 FPS: 15.56

Example output generated in V mode. Volume differences are shown on the y axis, and BPM is calculated based on the volumes.

N – No graph mode: In N mode, frequency and BPM are displayed and no additional data is presented.

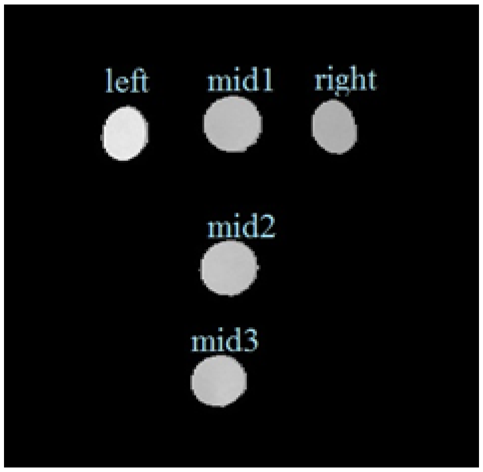
Permitted values: L, D, F and N.

- **Distances**

```
#distances: set to y to include a certain distance. set to n
otherwise. recommended: left-mid3, right-mid3, mid2-mid3
left-mid1 n
left-mid2 n
left-mid3 y
left-right n
right-mid1 n
right-mid2 n
right-mid3 y
mid1-mid2 n
mid1-mid3 n
mid2-mid3 y
```

In the distances field, each line represents a distance between two stickers.

The stickers are labeled by name, as described in the following scheme:



The distance between two stickers is marked by the names of the stickers, separated by a hyphen (-) character. Next to each distance, following a whitespace, appears a letter. By writing ***y*** next to a certain distance, the user instructs the application to take this distance into account when calculating the average distance in each iteration. The average distance is plotted in the distances graph generated in D mode and used for the frequency and BPM measurement as well (in modes D, F and N). In L mode, this field is disregarded.

In the config file example above, the application considers only 3 distances: left-mid3, right-mid3 and mid2-mid3 and ignores the rest.

Permitted values: y, n.

- **Location**

```
#location: set to y to track the location of a certain sticker.
left n
mid1 n
mid2 y
mid3 y
right n
```

In the location field, each line represents a sticker. The stickers are labeled by name, as described in the *distances* field section.

Next to each sticker`s name, following a whitespace, appears a letter. By writing ***y*** next to a certain sticker name, the user instructs the application to include this sticker when generating a location graph. In L mode, the graph generated will display the depth of every sticker marked with *y* in this field and ignore the rest. In modes D, F and N, this field is disregarded.

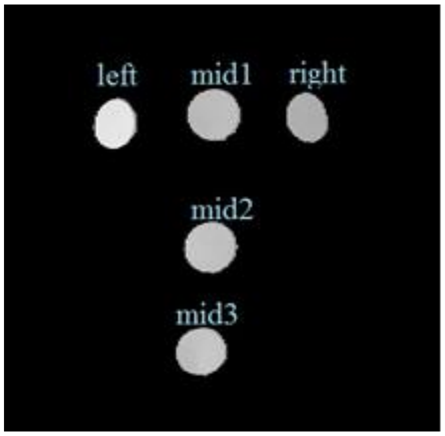
In the config file example above, the application disregards this field since *mode* is set to *D*. If changed to L mode, stickers mid2 and mid3 would be displayed in the location graph generated.

Permitted values: y, n.

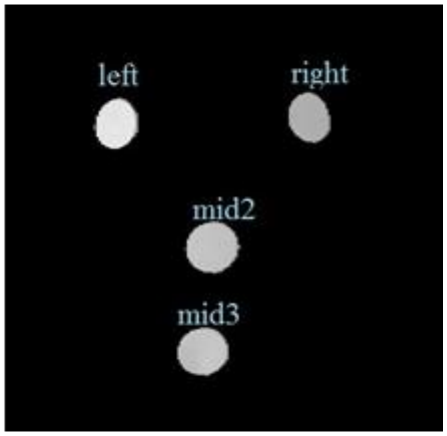
- **Number of stickers**

```
#number of stickers: 3, 4 or 5
5
```

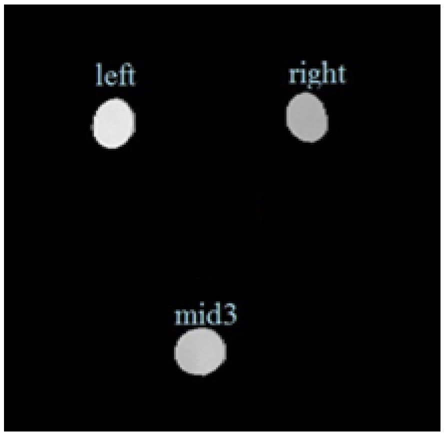
The application can analyze 3 different stickers layouts, with 3, 4 or 5 stickers:



Five stickers layout



Four stickers layout



Three stickers layout

The number in this field should be set according to the layout in use.

Permitted values: 3, 4, 5.

- **Color of stickers**

```
#color of stickers: Y (yellow), B (blue) or G (green). recommended: Y
Y
```

The application can detect stickers having 3 different colors: Y for yellow, B for blue and G for green. However, we recommend the use of yellow stickers due to higher accuracy in detection. While the tool was tested with different colors in the environment of private households, it was only tested on patients' videos (provided by the generous staff of Ichilov) with yellow stickers. Therefore, the hospital environment and lighting were not tested with other colors.

Permitted values: Y, B, G.

- **2D measure units**

```
#2D measure units: cm or pixel. recommended: pixel
pixel
```

When the dimension is set to *2*, the average distance can be measured either by centimeters (cm) or by pixels. When the dimension is set to *3*, only cm can be used (the field will be disregarded in that case, except from 2D distances appearing in the log file). For the purpose of frequency and BPM measurement, pixel units are recommended for higher accuracy.

Permitted values: pixel, cm.

- **Use stickers**

```
#use stickers:
y
```

An indicator whether to use stickers by default or not. Set to ‘n’ to use AI nipples and bellybutton recognition. (Not Implemented yet)

Permitted values: y, n.

- **Volume type**

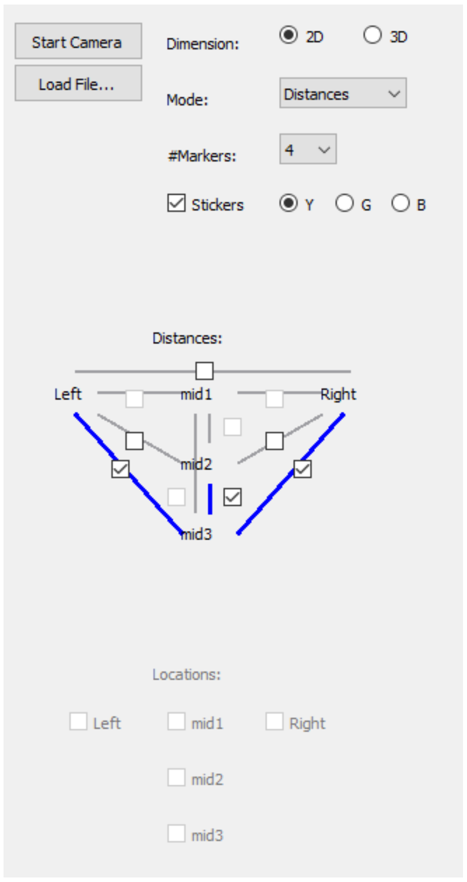
```
#volume calculation type (for V mode only): T for tetrahedron volume,
R for reimann sums
R
```

Used for volume calculation only. If volume mode is chosen, it can be calculated in two ways:

- By calculating the volume of the tetrahedron formed by the 3 valid stickers of the 3 stickers layout and the camera’s location.
- By calculating Riemann sums of the surface tamed in the bounding box of these stickers where the surface of origin is the plane of the camera.

Permitted values: R, T.

User Interface



Dimension: [✓ 2D] 3D
Mode: Distances
#Markers: 4
☑ Stickers [✓ Y] G B

The app's screen contains a set of configurations to choose between, all responsive and allow only valid combinations. Upon app's startup, the default configuration set in the config file is loaded.

There are two buttons that allow streaming: ***Start Camera*** and ***Load File…*** . If none of them is clicked, the app won't start streaming and other menu options will still be enabled and clickable.
It is possible to switch between camera streaming and file streaming anytime: If one of them is active, an alert dialog will pop and suggest to cancel the operation or continue.

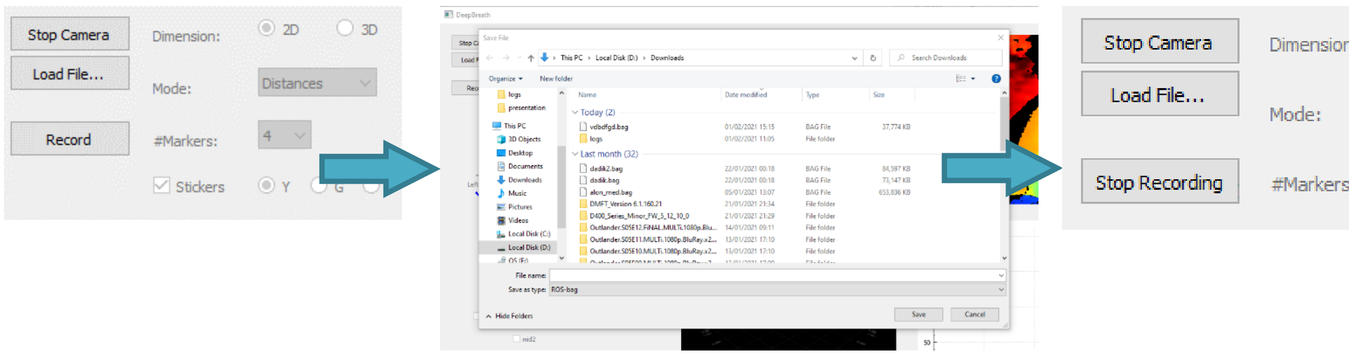
Start Camera
When checking this button, the application expects a live feed from the camera. The camera must be already plugged in to the computer via a suitable port before clicking the button. Consequentially, the live stream from the camera is shown on the app's screen (two streams are shown: color stream and depth stream), and a graph is shown in the graph widget according to the mode set in the configuration file. Below it, BPM and FPS values are shown as well (except for when mode is set to L) and are constantly updated. During the first seconds (up to 15), the BPM value shown is zero, since there is yet insufficient number of samples collected.
If Volume mode was selected, a scatter view of the participating points in the volumes' calculation is shown as well.
In addition, while camera stream is still open, all menu options are disabled to prevent user selections from changing during the streaming, the ***Start Camera*** button toggles to ***Stop Camera***, and a ***Record*** button appears under it.
Clicking on ***Stop Camera*** will stop the stream (and recording, if active) and enable back the menu.

Record

Upon clicking this button, a window opens, allowing the user to choose a location for the recording to be saved. The file will be saved in “.bag” format. After choosing the location, the recording is started, and the ***Record*** button toggles to ***Stop recording***.

Stop Recording

Clicking this button ends the recording. The application continues as before, using the camera’s feed. After a recording has ended, a new one can be started.



Name	Date modified	Type	Size
vdbdfgd.bag	01/02/2021 15:15	BAG File	37,774 KB
logs	01/02/2021 11:05	File folder	
dadik2.bag	22/01/2021 00:18	BAG File	84,597 KB
dadik.bag	22/01/2021 00:18	BAG File	73,147 KB
alon_med.bag	05/01/2021 13:07	BAG File	653,836 KB
DMFT_Version 6.1.160.21	21/01/2021 21:34	File folder	
D400_Series_Minor_FW_5_12_10_0	21/01/2021 21:29	File folder	
Outlander.S05E12.FiNAL.MULTi.1080p.Blu...	14/01/2021 09:11	File folder	
Outlander.S05E11.MULTi.1080p.BluRay.x2...	13/01/2021 17:10	File folder	
Outlander.S05E10.MULTi.1080p.BluRay.x2...	13/01/2021 17:10	File folder	

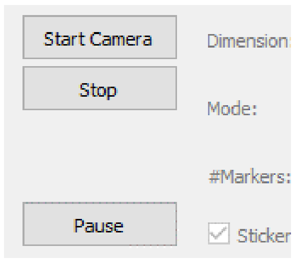
File name:
Save as type: ROS-bag

Load File…

Upon clicking this button, a file choosing dialog is opened, allowing the user to choose a previously recorded “.bag” file to be streamed. The file chosen must contain both streams used by the application (color and depth). Consequentially, the two streams from the file are shown on the app’s main screen. Equivalent to ***Start camera***, the application shows the streams, graph (and scatter view) and BPM and FPS values according to the configuration chosen, and configurations menu is disabled. In the same manner as above, the ***Load File…*** button toggles to ***Stop***.
When clicking ***Load File…*** , a ***Pause/Continue*** toggle button appears under it.

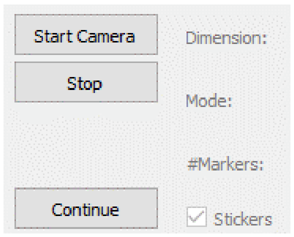
Pause

Upon clicking this button, the stream is paused. The video shown on the screen remains still, and no additional data is added to the graph.



Continue

Clicking this button resumes the stream from its pausing point. The streams shown on the screen, as well as the graph, continue as before.



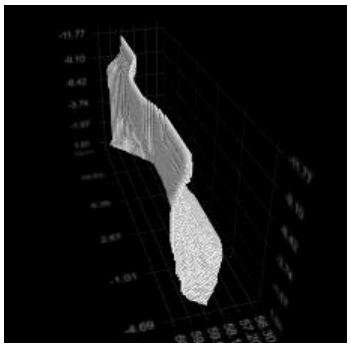
Graph Viewing

The graph widget on the bottom right of the screen displays the desired measurements according to the configurations in the config file. It is interactive and has plenty of viewing options:

- Using the mouse wheel, the graph can be scaled up or down.
- Using left button of the mouse, the graph can be moved.

Scatter View

The scatter widget on the bottom left of the screen displays the scatter of the points participating in the volumes' calculation (in Volume mode only). It is interactive as well: Using the right button of the mouse, the scatter can be rotated.



Log File

A separate log file is generated every time a stream is shown.

If *Start Camera* is clicked, a log file is created, and data is being written to the file until clicking *Stop Camera*. Using *Record/Stop recording* does not affect the logging process, and the log keeps updating. The generated log file appears in the directory *'logs'* in the application's root directory, and under the name format **live_camera_log_mode_dd-mm-yyyy_hh-mm-ss.csv**.

Upon clicking *Load File…* and choosing a file to be streamed, a log file is created too, and data is being written to the file either until clicking *Stop,* or until the file's stream had ended. Using *Pause* holds the arrival of new frames and therefore no new data is written to the log file until *Continue* is clicked. Then, writing to the log file is resumed from the point of pausing. The generated log file appears in the directory *'logs'* in the application's root directory, and under the name format **file_log_mode_dd-mm-yyyy_hh-mm-ss.csv**.

A single activation of the application may produce several log files, according to the number of files that were streamed, and the number of times *Show camera* was used.

Contents

The log file is in ".csv" format. The first line always contains titles – the names of the fields extracted every time a valid frame is received. Every configuration generates a log file based on user selections, while there are common fields for all logs and a generalization of the types of fields logged for each mode.

<u>The common data extracted is as follows</u>:

- **Frame idx** – a running index maintained by the application. Only valid frames, where all stickers were recognized are taken in account.
- **Color idx**, **Depth idx** – index of color frame and depth frame, correspondingly, supplied by the device.
- **Color timestamp, Depth timestamp –** timestamp of color frame and depth frame, correspondingly, supplied by the device. Time elapsed since the device was connected in millisecond.
- **System color timestamp** – time elapsed since the first frame (depth or color) arrival to the color frame timestamp arrival in seconds.
- **System depth timestamp** – time elapsed since the first frame (depth or color) arrival to the depth frame timestamp arrival in seconds.
- **System timestamp** – time elapsed since the application started in seconds.

<u>Data extracted based on user selections:</u>

Location Headers:

- **left (x y) [unit], right (x y) [unit], mid1 (x y) [unit], mid2 (x y) [unit], mid3 (x y) [unit]**
  Coordinates of the corresponding markers, given in the unit chosen in the config file (cm or pixel). These values are always logged but in different combinations. In all modes, only the active markers based on the *number of markers* will appear in the log file.
  Note: The field of the config file which is responsible for the unit is not controllable through the app since it does not have a significant meaning, and therefore as long as the config file has ‘pixel’ set in it, the unit will always be pixel in this case.
- **left (x y z) cm, right (x y z) cm, mid1 (x y z) cm, mid2 (x y z) cm, mid3 (x y z) cm**
  3D coordinates of the corresponding stickers, given in cm. In all modes, only the active markers based on the *number of markers* will appear in the log file.
- **left - mid1 2D distance [units], … , mid2 - mid3 2D distance [units]**
  2D Euclidian distance between the centers of two markers, given in [units] (cm or pixels, as explained above). These distances are always logged but in different combinations. In all modes except Locations, only distances selected or build-in for the mode are logged. In Locations mode, distances last selected in the configuration before choosing the Locations mode will be logged.
- **left - mid1 3D distance (cm), … , mid2 - mid3 3D distance (cm)**
  3D Euclidian distance between the centers of two stickers, given in cm. Logged in the same manner as the 2D distances, but only when selecting *3D* dimension in the configuration, in addition to the 2D values that are always logged.
- **2D average distance**
  The average 2D distance of all distances included in *distances* in the configuration, given in either cm or pixels, according to *2D measure units* set in the configuration file. This field is always logged.
- **3D average distance**
  The average 3D distance of all distances included in *distances* in the configuration, given in cm. Logged only when *3D* dimension is selected, in addition to the 2D value that is always logged.
- **Tetrahedron Volume / Reimann Volume**
  The calculated volume according to the *volume type* selected, in cm^3.

Logged only in *Volume* mode.

Note: The volumes logged are <u>not</u> the differences, but the actual values calculated for each frame. The reason behind this is the will to provide an easy way to test different fixed points and planes for volume calculations which might imply stronger connections between the absolute volumes calculated and their actual corresponding manual measurements.

- **FPS**

  the average frames rate (frames received per second). Last 256 frames are taken in account. Logged for all modes except Locations mode.
- **realSamplesNum**

  The number of samples (frames) currently stored by the application. BPM values are calculated based on these samples. The application may hold up to 256 samples at a time. When the number of real samples stored is significantly lower than 256, BPM values are unreliable and therefore not shown. Logged for all modes except Locations mode.
- **Frequency**

  The frequency of the breath rate, as measured after the corresponding frame was processed. Logged for all modes except Locations mode.
- **BPM** – the Breath Per Minute value, as measured after the corresponding frame was processed. BPM is calculated as 60 * frequency. Logged for all modes except Locations mode.

Results and Discussion

Testing environment

The application can be run with different configurations. In Realtime Breathing, we found that the following settings in the config file give the most reliable BPM values:

Dimension: 2
Distances: left-mid2, right-mid3, mid2-mid3
D2 units: pixel
Color of stickers: Y (yellow)
Number of stickers: 5

These are also the default values set in the config file.
As the volume measurements and BPM calculations based on it were found to be very similar (in the ideal environment), the following settings are also recommended:

Mode: V
D2 units: pixel
Color of stickers: Y (yellow)
Volume type: R

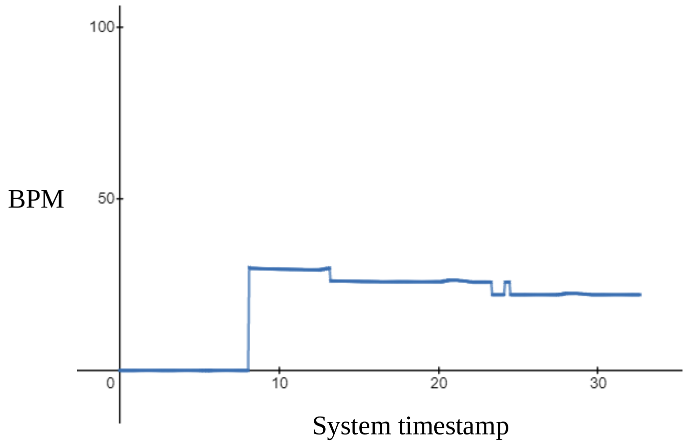
In the app, the user can select these settings by selecting Volume mode and Riemann Sums volume type. (All other relevant settings will be configured automatically and disabled to prevent errors)

The results given below were all measured using the latter recommended configuration. (For previous recommended settings, please see Realtime Breathing report)

As previously discussed, since the algorithm is based on color detection, objects in the background, which appear in the same color range as the stickers, may cause faulty results. Therefore, the results below were all measured using recordings or live streams with no significant noise in the background.

Additionally, for the application to give reliable BPM values, a certain number of samples must be collected first. It may take up to 15 seconds of streaming before this number is reached. For this reason, when comparing the application's values to the expected values, the first 10-15 seconds of each stream (an existing recording or a live stream) were disregarded.

The application outputs a BPM value in real time and updates the value several time per second. the graph below depicts the BPM values given by the application throughout the streaming of a one-minute file:



Breathing patterns of humans result in a BPM value that is not constant in time. In the absence of means to evaluate true BPM values for every point in time, the method we use to measure the application’s error is as follows:
For each stream (either a recorded file or a live stream), the average true BPM value is evaluated, and compared with the average BPM value given by the application throughout the run.
In addition, as implied earlier, we will discuss the general results of BPM measurements based on volumes calculations, and the possible approximation of the measured volumes to the real ones.

Streaming from an existing file

Samples from Ichilov that were used in Realtime Breathing were used in Deep Breath as well to assure there were no changes in the measurements of BPM values in the distances-based method. Although these samples showed good results in the previous methods, the BPM results achieved based on volumes in these samples were far from the original.
There were, though, some new samples from Ichilov (under favorable conditions) and old ideal samples shot in a ‘clear’ house environment that could provide fairly good approximations of the change in respiratory volume based on the knowledge of the tidal volume of human breath, but lacking the true values of the manual measurements it is hard to tell how strong the connection is.

Streaming live

The testing of a live stream performance was done in a private house environment. Currently, we are unable to perform live tests in a hospital's environment.
In an ideal environment, the results were almost identical between the previous method and the current. Even though, the measurements were still quite noisy.

Tetrahedron Volume

Before proceeding to the more accurate method of volume calculation, the first method tested as a proof-of-concept was tetrahedron volume differences between the frames.
Generally, in an ideal environment, BPM measurements in this method were somewhat indicative but not very stable (a few ‘glitches’ in BPM values could be noticed), which produced a relatively high error intermittently, while general results almost resemble the distances-based results. On the contrary, this method had generally produced better results on real samples from Ichilov (noisier samples).
Nevertheless, the change in volumes might still be indicative as mentioned before.

Riemann Sums Volume

In the ideal environment, BPM measurements in this method were very accurate, with an average error of less than 1% compared to the distances-based results, which might even be closer to zero as the comparison is done between two distinct runs of the videos analysis (which is done in realtime and therefore is less consistent even between two identical configurations).
Like in the Tetrahedron volume case, the change in volumes might be indicative as well.

Improvements

Multithreading

The demand on real time performance requires sacrifice of frame rate even on the strongest CPUs and GPUs. However, this can be overcome by manual support of multithreading. It had come to our knowledge that OpenCV supports multithreading, but the rendering of the frames and graphs and volumes calculations are heavy operations that affect the frame rate while run on a single thread. In addition, since the GUI is event based and is multithreaded itself, it is inevitable to run the frames polling and processing in a distinct thread to prevent UI blocking.

Currently the app runs on separate threads for each heavy operation as mentioned (whether it is implemented manually or supported by the libraries used).
Although volumes calculations cause a slight latency, we chose not to implement further multithreading, since the acquired samples under the current implementation are enough to supply satisfactory results while maintaining code simplicity.
All threads communications are done using condition variables (with wait mechanisms and locks) and flags, all maintained under a global synchronization object as described in earlier sections of this paper.

Methods Discarded

Using Intel's RealSense Viewer Source Code

At the very beginning of our work, our intention was to use as many available resources as possible and focus on algorithmics and real time processing improvement, namely using the existing code of Intel RealSense Viewer, and modifying it for our uses. Unfortunately, soon enough we encountered technical difficulties in modifying the existing code of the RealSense Viewer, as it is a very wide project poorly documented for these purposes.
As a result, we ended up paying efforts in constructing a very basic GUI for our app (with very little knowledge about it), as it was not the main goal of our application.
However, we did use the same GUI library as Intel (as we mentioned in the Tools and Developing Environment section) and were inspired by the viewer's and the examples' code, which allowed us faster and more stable work.

Hough Circles (vs. Color Detection and Connected Components)

We use color detection and connected components algorithm in order to detect the stickers properly, and even though noises are inevitable. At the stage of the algorithm planning and the implementation outline, we encountered the Hough Circles algorithm that is used to find circles in a picture and their centroids. During the development, we found that Hough Circles is not robust enough and is very prone to noises, not mentioning the fact that it is not color sensitive and quite expensive in time. Moreover, in order to detect the stickers properly with it, it is required to run the Hough Circles algorithm multiple times (roughly tens or hundreds of iterations[3], depends on the radius size) while its complexity is O(N^3)[4] – Not so efficient for real time processing.

Therefore, we decided to use more conventional tools to detect the stickers: simple color detection and Connected Components algorithm right afterwards. It is worth mentioning that both steps are carried out **only once**, what is more, their complexity is **linear**[5], and both steps are robust enough to complete the detection after handling the noises.

RGB (vs. HSV) Ranges Color Detection and Color Formats

In the beginning, we tried simple color detection with RGB common values to detect the yellow color of the stickers. At first, basic sticker detection did not work properly (or even at all), and that is where we began questioning the color ranges used in the inRange function of OpenCV.

Through further lookup through the net (references provided at the Refences section) we came into conclusion that HSV color ranges are more convenient to use due to their continuity in hues and values separately. (It is possible to generate a gradient from one color to another of desired intensities and all in between, as seen in the *Before We Start – Stickers* section)

In addition, it is important to note that not all color formats are supported either in librealsense applications or in OpenCV. We had gone through some trouble finding the appropriate formats that will allow us the interfacing between the camera's frames and OpenCV's functionalities.

Eventually, we found that streaming the color frame in RGB8 format (`RS2_FORMAT_RGB8`) will provide us the easiest and fastest way to convert the color frame 'data' – an matrix over RGB 8-bit – to OpenCV friendly format, that is, RGB 8-bit matrix as well that is later converted to a matrix over HSV color channels (also 8-bit).

Pointcloud vertices (and conversion to matrix)

As a means of providing depth data for the surface for which we calculate the volume by Riemann Sums, we tried to achieve all the vertices of the pointcloud of the depth frame. Namely, getting a vector of all the valid vertices of the depth frame. Soon enough we encountered a few drawbacks:

1. The vector is neither sorted nor can be easily converted to a matrix of defined width and

[3] https://stackoverflow.com/questions/38048265/houghcircles-cant-detect-circles-on-this-image
[4] https://www.cs.bgu.ac.il/~ben-shahar/Teaching/Computational-Vision/StudentProjects/ICBV052/ICBV-2005-2-BenYosef-Guy/circle_detection_using_folding_method.pdf
[5] http://graphstream-project.org/doc/Algorithms/Connected-Components/

height.
2. Manual conversion of the vertices to a matrix (of an approximated arrangement in a grid) costs (at least) another iteration over the vertices in addition to the summing process.
3. The sole operation of getting the vector is very heavy and causes a significant latency.
Therefore, while searching for a solution, we decided to stick to the method used for depth fetch from the depth frame in earlier stages of the algorithm – using `rs2_deproject_pixel_to_point` for each point. As this operation is heavy for each pixel as well, a satisfying solution was to delimit the deprojection area to the bounding box of the markers as it is fast and accurate enough.

Bugs

The application is not perfect, and it is possible that it has some unresolved issues. Let us discuss the open issues we have encountered and have not managed to fix them yet.

Camera streams rendering gets stuck

This is a new bug that appeared by the end of the development process. The GL widget responsible for rendering the streams freezes on the second frame arriving for rendering: If the color frame arrives first and the depth second, the widget rendering the depth might randomly freeze. This also happens regardless of the type of frame or even its actual arrival in the frameset: The bug also occurs when trying to render the exact same frame twice, on the second frame in line. This bug might be related to support issues of the Nvidia GPU drivers and Realsense SDK in Windows[6] or to Windows support in Qt libraries (Check for `Qt::WindowType::MSWindowsOwnDC` flag in the initiation of the widget).

Scatter widget read access violation + latency

When providing large sets of scatter points too frequently to the widget, it can free the buffer used by renderer while rendering has not ended yet. This happens because the widget does not provide a lock mechanism while rendering, and the partial solution for this consists of two steps:
1. Reimplementing the widget (with heritage) based on the open source, adding locks.
2. Reducing the rendering rate by 2 (in addition to dumping invalid frames)
The first step eliminates the read access violation as far as we have experienced, and the second reduces latency in a relatively fair price of smoothness and latency.

[6] https://forums.developer.nvidia.com/t/realsense-d435-with-jetson-tx2-depth-stop-working-freezes-after-time/83911

Even though, while this solution works for high-end hardware, it might not be sufficient for mid-class common hardware.

Possible UI bugs

It is very likely that the user interface is not completely polished. The user might experience some inconsistencies in the GUI behavior, such as enabling of values that should be disabled under certain configurations, unexpected crashes in certain irregular flows (especially on starting and stopping streams) and more. These bugs are usually very easy to fix but are quite hard to find since they happen in very specific flows that are not seen under the usual use case radar (and therefore they still exist).

Future work

Our project supplies a modular implementation of breath pattern analysis. Therefore, it is possible to take our work to new extents.

To begin with, it might be possible to remove the app's dependency on stickers and use image detection of the markers instead (of nipples and navel). The app's code provides a preparation for this, in the `identify_markers` method of *DeepBreathFrameManager* class.

In addition, phasing information might be extracted and visualized from the locations data provided by the app. Further information of the clinical significance of the locations is needed for this, though.

Finally, more improvements of the volume calculations can be made: Using edge detection to delimit the surface more accurately, finding a better fixed point for calculations and as a result finding a stronger connection between the values and the real volumes.

There is more that comes to mind regarding the expansion of the app's domain, and we hope that more research and development will realize any notion that may sharpen respiration research, abnormalities detection, and treatment.

A Few Personal Words

Once again, I want to thank my supervisors, Alon and Yaron, for their great patience and understanding.

As opposed to other times, this project had taken place amid the COVID-19 outbreak, affecting the fluent work in many aspects, from depriving us from frontal meetings to challenging my working environment and availability.

Nevertheless, we managed to overcome the hardships the last months brought us to face, and once again made it to a functional tool which might, even with little impact, change our futures for the best.

Another real sense (😊) of contribution arises in my heart from this project as well, and I am eager to see its value in the long term.

Special thanks,

Nili

References

Intel RealSense GitHub

Librealsense examples:

Sample Code for Intel® RealSense™ cameras

rs-imshow

rs-dnn

Displaying Camera Frames in Custom OpenGL Window

Iterating over pointcloud

rs2::frame to cv::mat

Qt

Qt rendering in custom widget

QCustomPlot

Realtime Data Demo

Examples from Forum

User Interactions

OpenCV Releases

Shape and color detection:

Detect RGB color interval with OpenCV and C++

Shape Detection & Tracking using Contours

Color Detection & Object Tracking

librealsense/cv-helpers.hpp at master · IntelRealSense/librealsense

Hough Circles Transformation:

Hough Circle Transform

Color Ranges:

Finding Lane Lines with Colour Thresholds - Joshua Owoyemi

ColorHexa - for visualization of color ranges

OpenCV better detection of red color? (Stackoverflow)

Detecting Blue Color in this image - OpenCV Q&A Forum

Detecting colors (Hsv Color Space) - Opencv with Python (Green color example)

HSV Color Picker

plog GitHub

Intel Realsense Wikipedia

OpenCV Wikipedia

Visual Studio Wikipedia