Realtime Breathing

Breath Pattern Detection With Intel 3D Camera

Nili Furman

Maayan Ehrenberg

Supervised by

Alon Zvirin

Yaron Honen

Technion | Israel Institute of Technology

May 06, 2020

Abstract

Chest motion abnormalities and 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 also 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.

Realtime Breathing - Breath Rate 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

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 2017

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.
We used Visual Studio 2017 due to compatibility issues with the Realsense toolkit.

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 2017 supports almost all of C++17.

Dear Imgui

Dear ImGui is a bloat-free graphical user interface library for C++. It is fast, portable, renderer agnostic and self-contained (no external dependencies).

Dear ImGui is designed to enable fast iterations and to empower programmers to create content creation tools and visualization / debug tools (as opposed to UI for the average end-user). It favors simplicity and productivity toward this goal, and lacks certain features normally found in more high-level libraries. (Dear Imgui GitHub)

Thus, it is very suitable for real-time 3D applications, and it is indeed the library used by Intel Realsense itself in the Realsense Viewer and many of the examples provided by Intel.

OpenCV 4.2.0

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.

CvPlot ***by Profactor GmbH***

OpenCV plotting library, used for graph plotting. It allows us to conveniently plot clean, good-looking graphs of the desired data while providing a simple set of tools to move and enlarge the plotted graph.

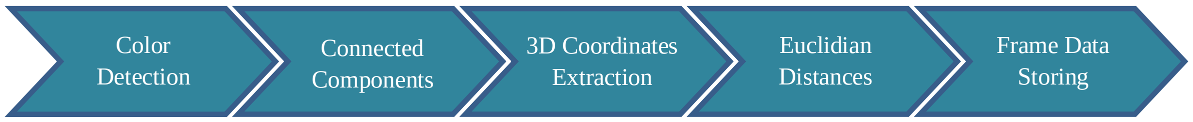
Algorithm

The main goal of the app is analysis of chest movement and extraction of the BPM (breaths per minute) rate 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. 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.

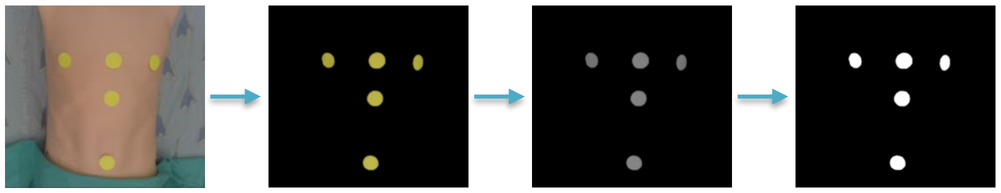
The app's main loop awaits a user's request to run on a certain stream (either from an existing file or from the camera). After such a request is received, depth and color frames are polled using the device's wait mechanism in each iteration of the main loop. The frames are then aligned (rs2::align_to_color). The two matched frames (a depth frame and a color frame) are then processed and a BPM (breaths per minute) measurement is performed.

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.

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). In this case, the 3D coordinates returned are invalid. When then dimension set in the configuration file is 3D, frames with invalid

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

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.

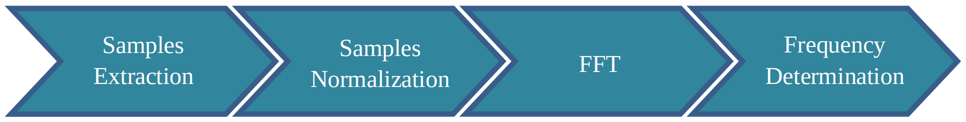
4. Euclidian distances

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 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. 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 as calculated in the frame processing stage. The field extracted is either 2D average distance or 3D average distance, according to the dimension set in the configuration file. The pairs of stickers taken into account in the average distance are as defined in the *distances* section of the configuration file. 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.

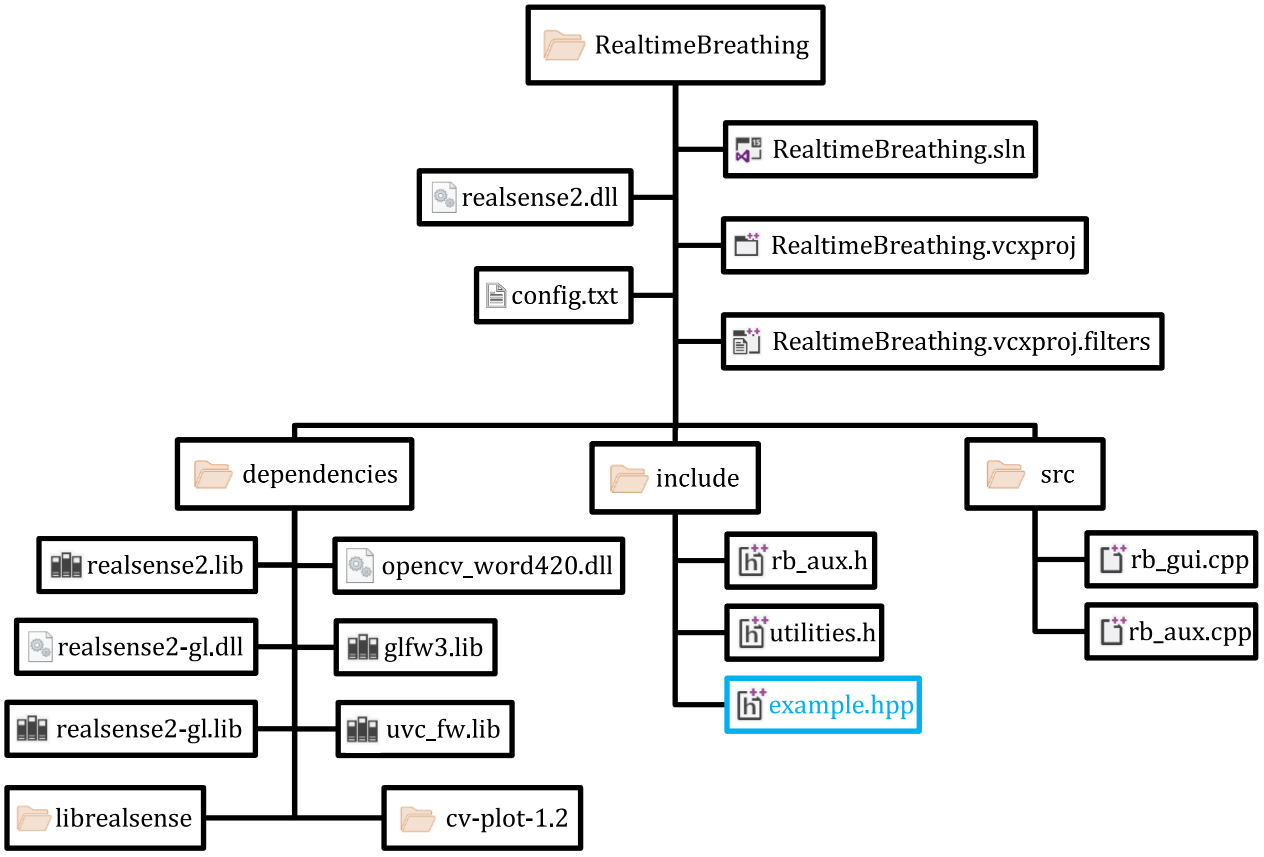
The BPM value returned at each iteration is **60 * frequency**.

Correctness

The above described algorithmics (average distance extraction per frame ⇢ samples normalization ⇢ FFT ⇢ dominant frequency and BPM determination) was tested in two different implementations, 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%.

Solution Overview

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



RealtimeBreathing – Root Directory

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

- *RealtimeBreathing.sln* – Visual Studio 2017 solution file containing the project.
- *RealtimeBreathing.vcxproj* – Visual Studio project file.
- *RealtimeBreathing.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

rb_gui.cpp

contains the main code.

Outline and main attributes:

——————————— Window and Rendering ———————————

1. Initialization of the app's window, ImGui library and rendering helpers.

——————————— Camera Related Initializations ———————————

2. Creation of a pipeline and a configuration file of the camera and align objects.

——————————— Our Data Structures Initializations ———————————

3. Initialization of Config, FrameManager and GraphPlot.
   Definitions of variables.
   Definition of the frameset variable.

——————————— Main Loop ———————————

4. The main loop. Runs while the app is alive.
   Initialization of rendering flags, frames, and checkboxes.
5. Treatment of user's choice of streaming, log file opening and closing, record, pause and continue buttons.
   Starting the pipeline, enabling streaming if needed, and disabling it when needed.
   Waiting for frameset.
6. Aligning the depth frame to the color frame in the frameset.
7. Getting color and depth frames from the frameset.
8. FrameManager processes the frames.
9. Frames rendering.
10. Graph plotting.

———————————————————————————

rb_aux.cpp

contains auxiliary functions and all implementations of the functions in our headers.

include – Our Headers

rb_aux.h

Definitions and declarations of primary data structures used for processing and plotting.

utilities.h

Definitions and declarations of helper data structures and functions.

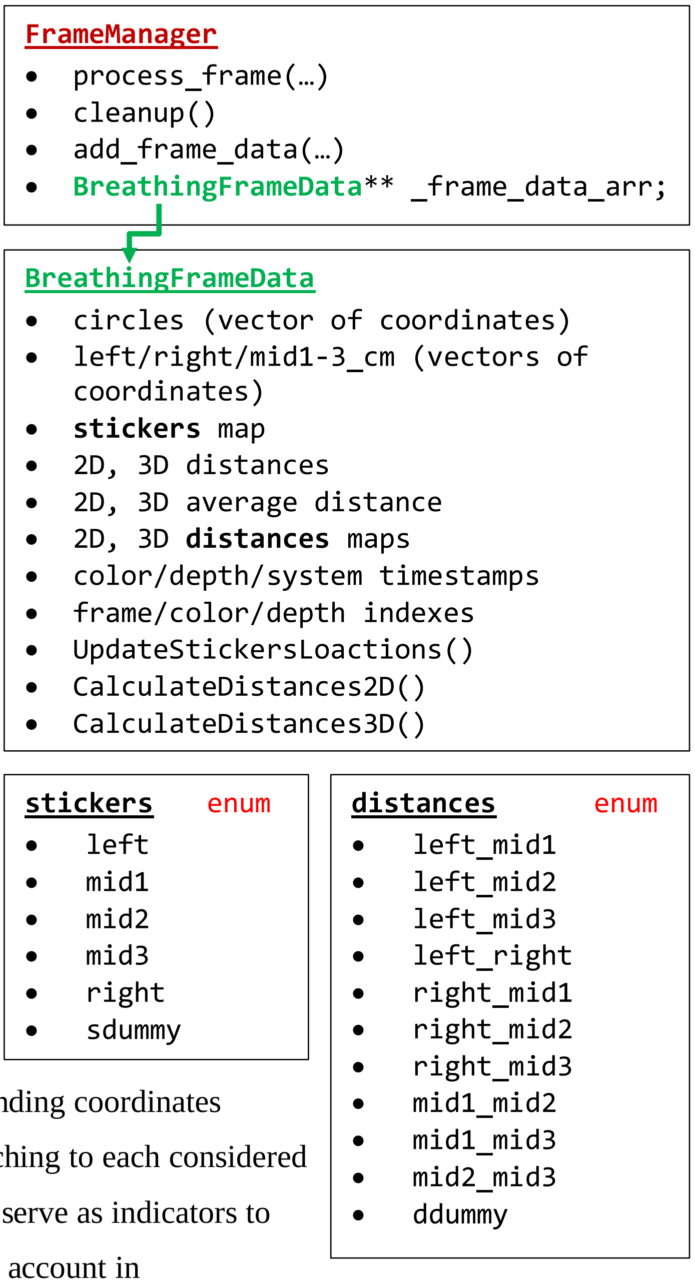
dependencies

Contains all relevant libraries required for the project, CvPlot entire code (both headers and sources), and all Intel RealSense (librealsense) relevant headers in their original hierarchy. It also includes a ready OpenCV 4.2.0 DLL in case you need it for the app package.

Data Structures

FrameManager

Created and initialized once in the main code, and responsible for all frame managing: frame processing, frame storing, cleanup. In addition, it fetches relevant data for graph plotting.



BreathingFrameData

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 configured distances in the config file, 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 – when called by *FrameManager* 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.

Config

Class storing the data parsed from the config file. Created and initialized once in the main code and used in various cases by other classes when data parsed from the config file needs to be accessed.

Config
- **dimension**
- **Graph mode**
- **stickers** included boolean map
- **distances** included boolean map
- **sticker_color**

GraphPlot

Manages graph plotting of the app.

Creates a new window for the graph upon stream start and updates it every iteration of frame processing (if values are valid). It gets its parameters from the *Config* class upon initialization, and from *FrameManager* for plotting. This class's plotting methods are based on CvPlot library, so it is recommended to view the documentation of the library in order to understand those methods.

GraphPlot
- *CvPlot::Window** window
- *CvPlot::Axes* axes
- Beginning time
- **graph_mode**
- **dimension**
- First plot Boolean
- Private plotting methods for all modes
- GraphPlot(…) constructor
- reset(…)
- plot(…)

sticker_color enum
- YELLOW
- BLUE
- GREEN
- RED

graph_mode enum
- DISTANCES
- LOCATION
- FOURIER
- NOGRAPH

dimension enum
- D2
- D3

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.

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 RealtimeBreathing.exe file:

realsense2.dll
opencv_world420.dll

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

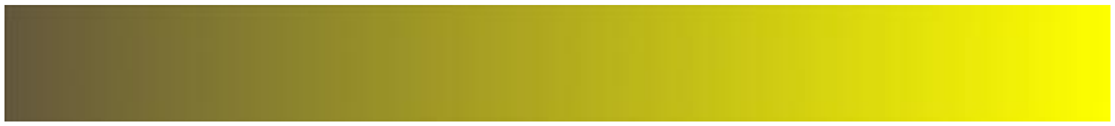
The app requires a config.txt file to run. This is a file containing configurations to use when running the app, written in a certain template to allow the app an intact parsing of the configurations chosen. The structure and usage of this file will be discussed widely in the app overview section.

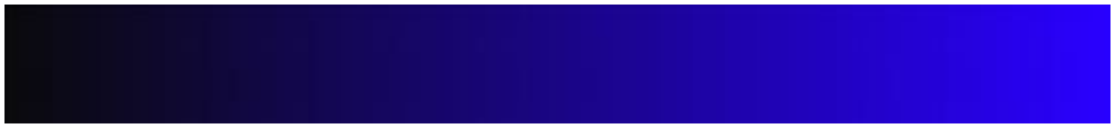
[2] Download and follow the instructions here.

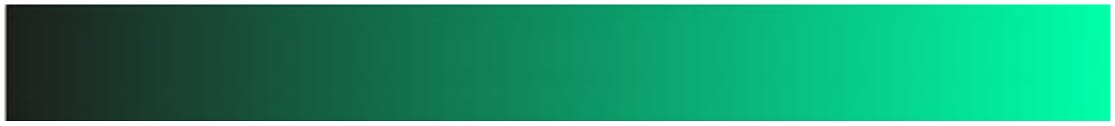
Stickers

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 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, 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.
left n
mid1 n
mid2 y
mid3 y
right n

#number of stickers: 4 or 5
5

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

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

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**

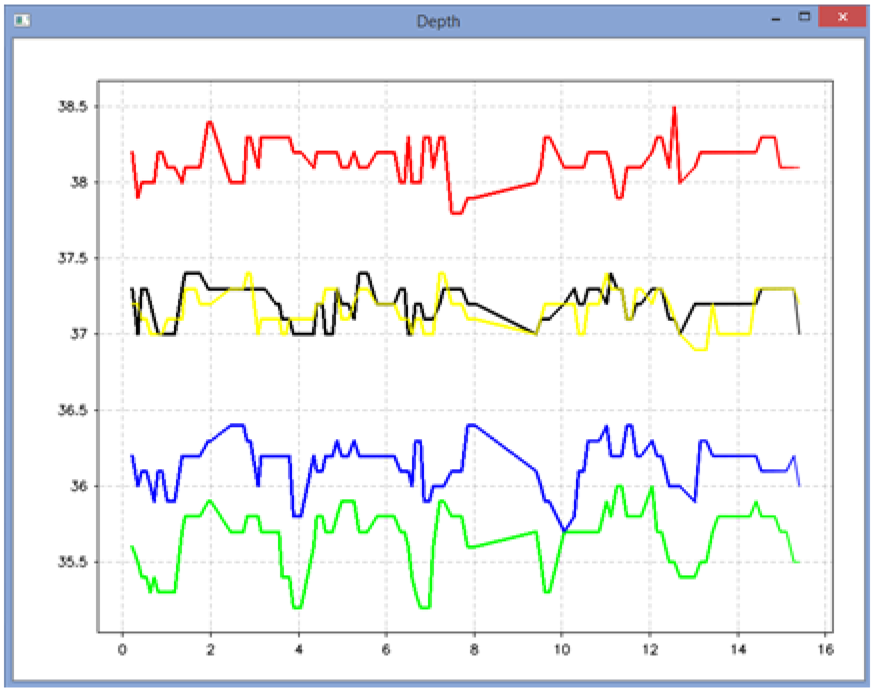
```
#dimension: 2 for 2D or 3 for 3D. recommended: 2
2
```

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, N for no graph
D
```

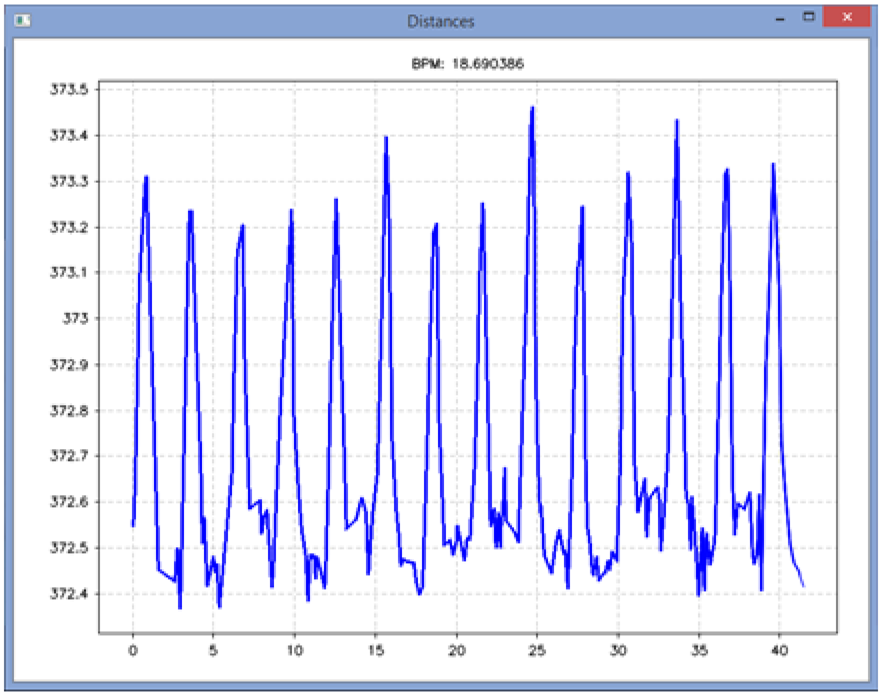
The app can be configured to 4 different modes, each displaying different output.
L – location mode: While in location mode, the application does not output a BPM value. When streaming from a live stream or from a file, a graph is generated, tracking the depth coordinate of different stickers in each 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, all 5 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.

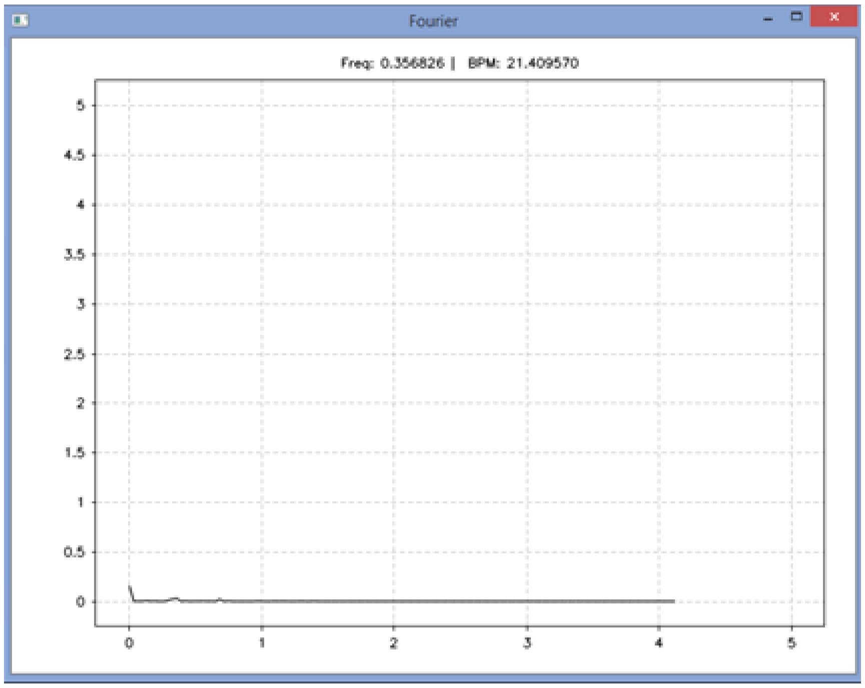
In D, F and N modes the application measures BPM by tracking the change in the average distance between stickers, using the same method, and presented to the user in real time. However, in each mode, a different graph is generated alongside the BPM value.

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).



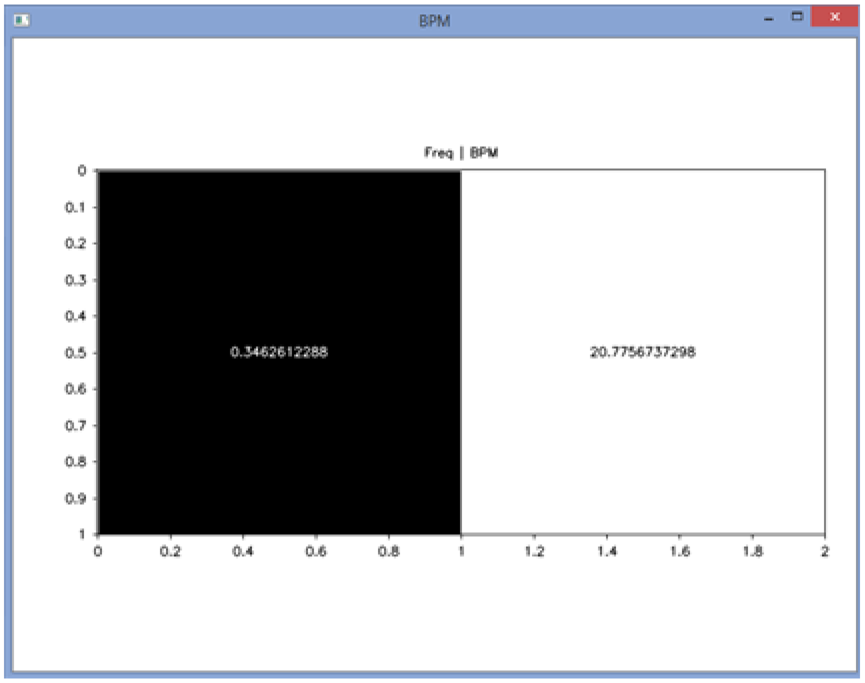
Example graph generated in D mode. The average distance in each frame is displayed against the according timestamp.

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).



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

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



Example output generated in N mode. Most dominant frequency and BPM values are displayed.

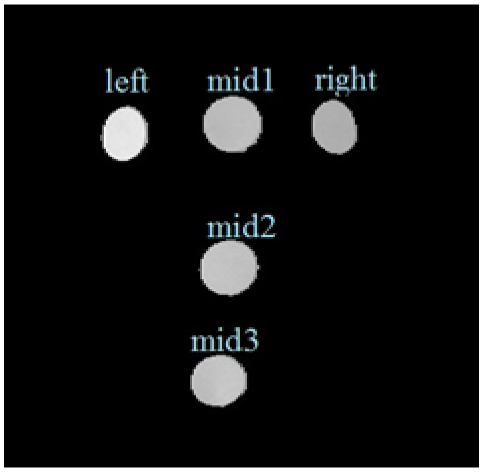
Permitted values: L, D, F and N.

- **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.

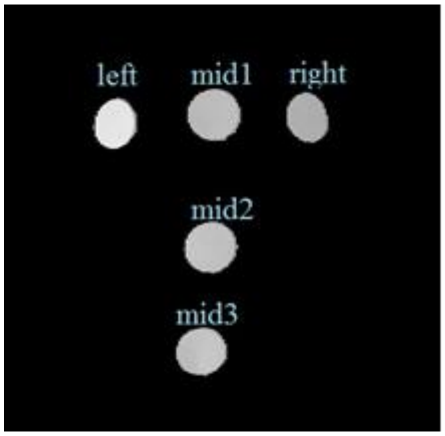
In the config file example above, the application disregards this field since *mode* is set to *D*. If changed to L mode, stickers mid2 and mid3 would be displayed in the location graph generated.

Permitted values: y, n.

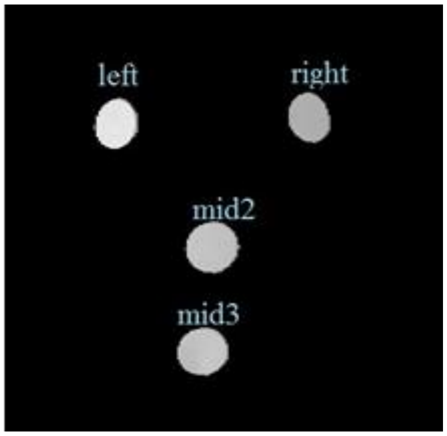
- **Number of stickers**

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

The application can analyze 2 different stickers layouts, with 4 stickers or 5:



Five stickers layout



Four stickers layout

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

Permitted values: 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**

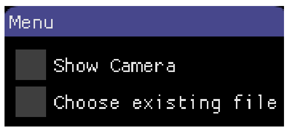
```
#2D measure units: cm or pixel. recommended: pixel
pixel
```

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.

User Interface

The app's opening screen contains two buttons: ***Choose existing file*** and ***Show camera***. Whenever both buttons are checked, as well as when none of them are checked, the application receives no stream.



Show 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 main screen (two streams are shown: color stream and depth stream).

In addition, a separate window opens, presenting a graph according to the mode set in the configuration file. This window displays BPM values as well (except for when mode is set to L) and is constantly updated. During the first seconds (up to 15), the BPM value shown is zero, since there is yet insufficient number of samples collected.

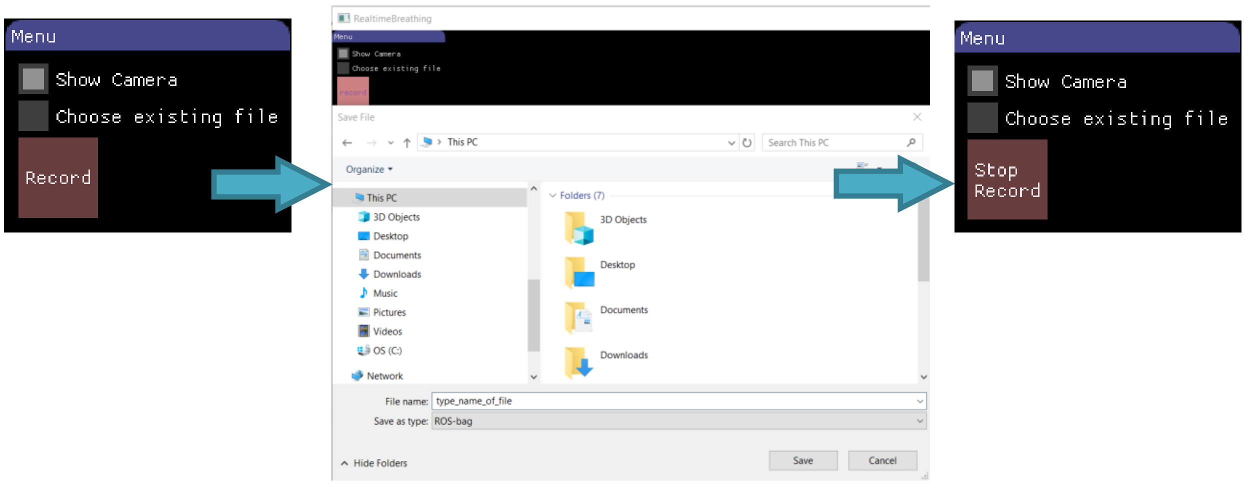
While ***Show camera*** is checked, a ***Record/Stop Record*** button appears in the main window.

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 ***Stop record*** button appears below.

Stop record

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.



File name: type_name_of_file
Save as type: ROS-bag

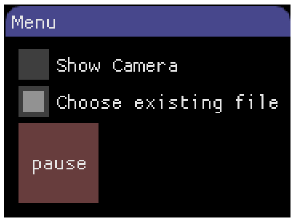
Choose existing file

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 ***Show camera***, a new window appears, presenting a graph according to the mode set in the configuration file, as well as BPM values (except for when mode is set to L). Just as in ***show camera***, during the first seconds the BPM value shown is zero.

While ***Choose existing file*** is checked, a ***Pause/Continue*** button appears in the main window.

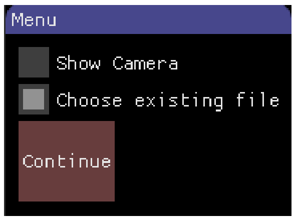
Pause

Upon clicking this button, the stream is paused. The video shown in the main 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 in the main screen, as well as the graph, continue as before.



Graph Viewing

The new window opened upon every stream start is, as already mentioned, showing a graph of the desired measurements according to the configurations in the config file. This graph has plenty of viewing options:

- Using the mouse wheel or pressed right mouse button, the graph can be scaled up or down.
- Using pressed scroll button (wheel) of the mouse, the graph can be moved.
- Graph can be reset by double clicking the right mouse button.

Log File

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

If *Show camera* is checked, a log file is created, and data is being written to the file until unchecking *Show camera*. Using *Record/Stop record* does not affect the logging process, and the log keeps updating. The generated log file appears in the application's directory under the name format **live_camera_log_dd-mm-yyyy_hh-mm-ss.csv**.

Upon checking *Choose existing file* and choosing a file to be streamed, a log file is created too and data is being written to the file either until *Choose existing file* is unchecked, 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 application's directory under the name format **file_log_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.

The following lines may contain one of the followings:

- An empty line – may indicate a frame was discarded since it is a duplication of the previous frame, or since not all stickers were recognized.
- *Warning: illegal 3D coordinates! frame was dumped.* – this message indicates a frame was discarded due to illegal 3D coordinates.
- *frames array cleanup...* – this message indicates all samples currently stored by the application were removed. This may happen after several consecutive frames were discarded, either for fault of illegal 3D coordinates, or for fault of frames duplication.
- A line containing values in all 44 fields – such a line is written for every received valid frame.

The data extracted is as follows:

- **Frame_index** – 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) in seconds.
- **System depth timestamp** – time elapsed since the first frame (depth or color) in seconds.
- **System timestamp** – time elapsed since the application started in seconds.
- **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 sticker, given in cm.
- **left (x y) pixels, right (x y) pixels, mid1 (x y) pixels, mid2 (x y) pixels, mid3 (x y) pixels** - 2D coordinates of the corresponding sticker, given in pixels.
- **left - mid1 2D distance [units], … , mid2 - mid3 2D distance [units]** – 2D Euclidian distance between the centers of two stickers, given in [units] (units may be either cm or pixels, according to *2D measure units* set in the configuration file).
- **left - mid1 3D distance (cm), … , mid2 - mid3 3D distance (cm)** – 3D Euclidian distance between the centers of two stickers, given in cm.
- **2D average distance** – the average 2D distance of all distances included in *distances* in the configuration file, given in either cm or pixels, according to *2D measure units* set in the configuration file.
- **3D average distance** – the average 3D distance of all distances included in *distances* in the configuration file, given in cm.
- **FPS** – the average frames rate (frames received per second). Last 256 frames are taken in account.
- **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.
- **Frequency –** the frequency of the breath rate, as measured after the corresponding frame was processed.
- **BPM** – the Breath Per Minute value, as measured after the corresponding frame was processed. BPM is calculated as 60 * frequency.

Results and Discussion

Testing environment

While the application can be run with different configurations, we found that the following setting gives the most reliable BPM values:

Dimension: 2

Distances: left-mid2, right-mid3, mid2-mid3

D2 units: pixel

Color of stickers: Y (yellow)

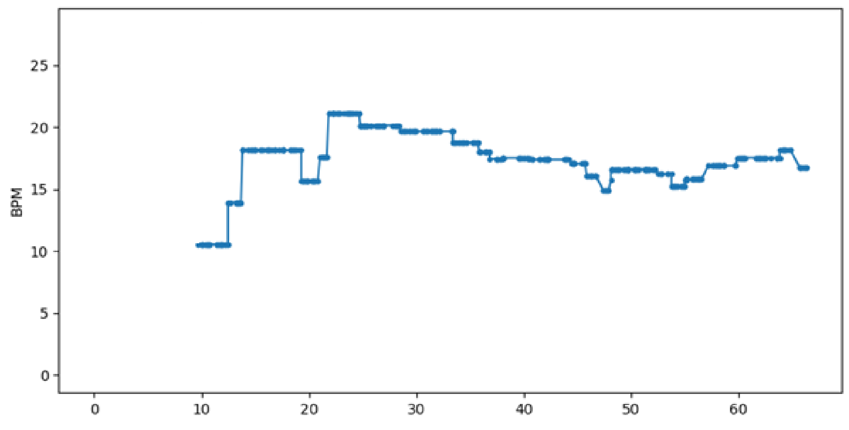
Number of stickers: 5

These are also the default values set in the config file.
The results given below were all measured using this recommended configuration.

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.

Streaming from an existing file

All files used in these tests were provided by Ichilov and were recorded in the hospital's environment.
We tested several files and received an average deviation of 1.85 BPM. On average, 6.24% error (percentage of the true BPM).

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.
We tested live streaming in several different breathing rates and received an average deviation of 1.01 BPM. On average, 3.49% error (percentage of the true BPM).

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-

[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/

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).

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 are heavy operations that affect the frame rate while run on a single thread.

For comparison, the average real time frame rate (on a modern 8-core CPU and strong GPU) of the application full processing **without** graph rendering is 25 fps, which drops significantly to 16 fps on average **with** graph rendering. Upon file read, which is lower a priori, it drops even to a single-digit frame rate, which is unfavorable at least.

The solution to these might be (and actually is, based on running a testing and buggy implementation) manual implementation of multithreading: creating separate threads for the renderers, and the graph renderer in particular.

Nevertheless, we chose not to implement multithreading, since the acquired samples under the current implementation (of a single thread) are enough to supply satisfactory results while maintaining code simplicity.

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, by calculating the difference in volume of the polyhedron defined by the stickers, it might be possible to exploit variations in respiratory volume.

In addition, the depth dimension of the 3D camera might provide information about phasing abnormalities in breathing, when looking at the locations of the stickers (which are already provided by the app).

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

We want to thank our supervisors, Alon and Yaron, for their great patience towards us and our questions and troubles, and for their great support in technical and logical issues we faced.

We started the project with little to no knowledge of almost all fields regarding the project and grew wiser each day. We feel a real sense of contribution from this project, and we hope it will satisfy those who will use it, and those who will follow us.

Special thanks,

Nili and Maayan

References

Intel RealSense GitHub

Librealsense examples:

Sample Code for Intel® RealSense™ cameras

rs-imshow

rs-dnn

Dear Imgui GitHub

For GUI Implementation in the app, contains a set of examples and fine documentation.

buttons and windows:

demo window

Quick GUI for MBED projects using Dear ImGui and Serial - jaeblog

OpenCV 4.2.0 GitHub

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

cv-plot GitHub

Cpp-text-table GitHub

Intel Realsense Wikipedia

OpenCV Wikipedia

Visual Studio Wikipedia