Reading time in analog clocks and watches

Universidade do Porto

Faculdade de Engenharia

FEUP

Visão por Computador
Grupo 10

5^o **ano** 1^o **semestre**
Mestrado Integrado em Engenharia Informática e Computação

Hugo Drumond	201102900	hugo.drumond@fe.up.pt
Inês Caldas	200904082	up200904082@fe.up.pt
Pedro Moura	201306843	up201306843@fe.up.pt

Conteúdo

1 Introdução

No âmbito da Unidade Curricular de Visão por Computador foi proposto elaborar um algoritmo que calcule as horas exibidas num relógio analógico usando as técnicas de processamento e análise de imagens estudadas nas aulas. Assim sendo, é feita a descrição e explicação do modelo implementado e de todos os passos intermédios que conduziram à solução. Além disso, são descritos aspetos mais técnicos, tais como: metodologia de testes e ferramentas/instruções de compilação. Ao longo do relatório, serão demonstradas imagens dos resultados obtidos para um dos relógios de teste.

2 Metodologia de testes

Usou-se um conjunto de imagens de modo a verificar se as horas eram detetadas corretamente. As horas:segundos:minutos são bem tirados nas 23 imagens de teste usadas. Apenas nas 3 imagens assinaladas como clockxy_tricky.ext é preciso fazer o ajuste manual dos parâmetros descritos no nome do ficheiro.

Figura 1: Clock "tipo"usado para demonstração dos resultados

3 O algoritmo

Em suma, o algoritmo desenvolvido encontra-se separado nas seguintes partes: identificação e isolamento da face do relógio; análise das linhas interiores a uma percentagem do raio do relógio; identificação das linhas que pertencem aos ponteiros; aplicação de um sub-algoritmo personalizado de clustering que faz o merge de linhas pertencentes a cada conjunto de ponteiros tirando partido da característica de simetria normalmente encontrada nos ponteiros; determinação das horas, minutos e segundos (caso exista).

3.1 Modelo e condições de funcionamento

Foram seguidas quase todas as simplificações indicadas na especificação do trabalho, exceto:

- the clock has just two hands: hours and minutes;
- the clock occupies a significant portion of the image;
- the clock display and hands are uniform in color;
- there are no other circular objects in the scene background.

3.2 Deteção da face do relógio

De modo a melhor a identificação do circulo do relógio é feito um pré-processamento através de um *Gaussian Blur*. Este algoritmo reduz algum ruído e detalhe da imagem, ajudando na identificação de *edges*, seja por diminuição ou melhor definição das formas geométricas. Posteriormente, aplicou-se o algoritmo *Hough Circles* até ser encontrado pelo menos um círculo, fazendo ajuste do argumento *Canny treshold* do *Hough Circles* até um máximo de 50 tentativas. De seguida, os círculos são ordenados por ordem descrente de raio. Se o círculo a ser testado não contiver ponteiros passa-se ao próximo. Após ser encontrada a face do relógio é feito o isolamento do círculo através da aplicação de uma máscara *circle* sobre a imagem original com os dados do raio e centro e é feito o recorte da imagem definindo a ROI.

Após a face do relógio ser deteta, é então necessário proceder-se à deteção dos ponteiros e do cálculo das horas, cujos algoritmos se encontram explicados nas secções que se seguem. É de referir que também se realizaram tentativas de identificação de faces de relógios aproximadas a um círculo através de contornos e *fills*. No entanto, essa implementação não introduziu melhorias significativas e seria necessário mais tempo para explorar soluções mais eficazes.

3.3 Deteção dos ponteiros

Na identificação dos ponteiros, começa-se por aplicar um *bilateral filter* com o objetivo de diminuir o ruído da imagem mas, ao mesmo tempo, tendo o cuidado de prevenir a eliminação de *edges*. De seguida, passa-se a matriz pelo *Canny* de modo a estabelecer as *edges* que serão usadas pelo algoritmo *HoughLinesP*.

Figura 2: Before bilateral

Figura 3: After bilateral

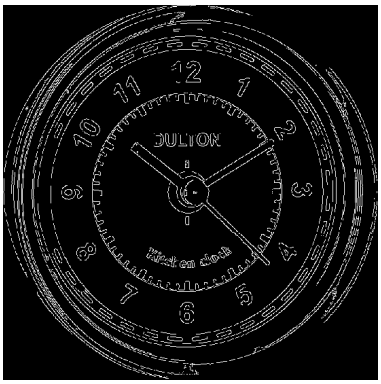

Figura 4: After canny

Posteriormente, é feito o cálculo de uma máscara *red* que deteta os pixeis na gama dos vermelhos presentes na imagem. Esta máscara é usada para separar o ponteiro vermelho dos segundos, caso exista. A máscara é obtida através da extração da gama de valores de vermelho dentro de duas *ranges* (*lower* e *upper*) e da aplicação de uma operação de dilatação de modo a não deixar partes das bordas na imagem. Esta técnica permite uma melhor deteção da sobreposição do ponteiro dos segundos com os outros ponteiros e, além disso, não exige que o ponteiro dos segundos seja sempre o maior, o que não é verdade muitas vezes. É possível, mesmo assim, ter outros elementos vermelhos na imagem e fazer a identificação do ponteiro dos segundos caso este não seja vermelho, mas neste caso, terá de ser o ponteiro mais longo.

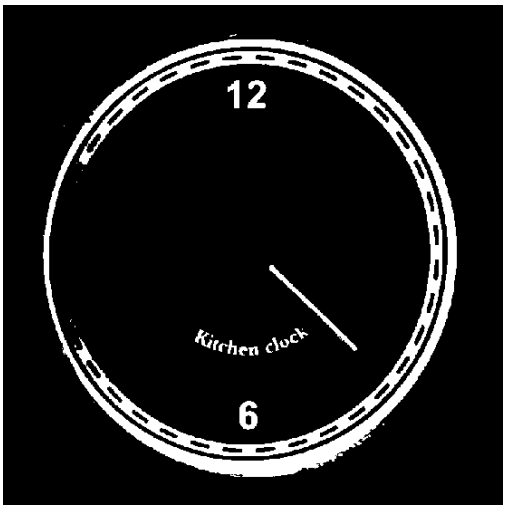

Figura 5: redmask sem dilation

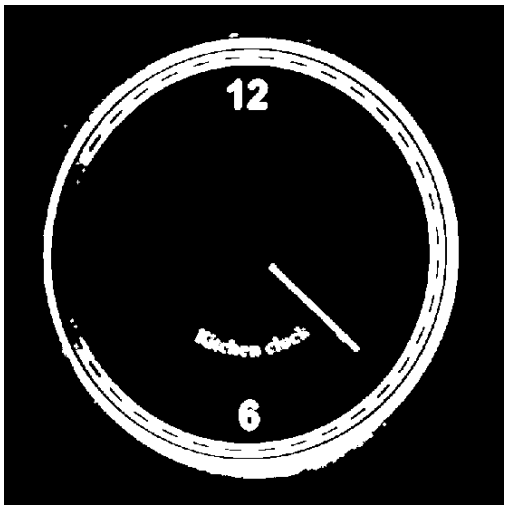

Figura 6: redmask com dilation

A identificação de ponteiros é testada inicialmente para a imagem obtida pela máscara *red* fazendo uso da seguinte função:

```
auto redMergedClockLines = iterativeLinesSearch(programData, clockCircle, redMask,
↪  1);
```

Esta função faz pequenos ajustes ao *HoughLinesP threshold*, e passa as linhas resultantes por uma função de filtragem(selectLinesCloseToCircleCenter) e de clustering/merging(clockPointerLinesMerge) até que se exceda as tentativas permitidas ou seja detetada pelo menos numberOfLinesToBreak. Se a função iterativeLinesSearch terminar com a condição de pelo menos numberOfLinesToBreak esta o maior conjunto de linhas até ao momento, descartando outras possíveis linhas que possam existir. No caso de terminar pela condição do maior número de iterações, esta retorna o maior conjunto de linhas encontradas ao longo do ciclo. O conjunto de linhas obtidas são retornados por ordem decrescente da norma das mesmas, facilitando assim o processo de associação de linhas a horas, minutos e segundos.

```
mergedClockLines = selectLinesCloseToCircleCenter(lines, clockCircle,
↪  DEFAULT_LINES_SELECTION_RADIUS_FACTOR);}

mergedClockLines = clockPointerLinesMerge(mergedClockLines,
↪  DEFAULT_LINES_MERGE_ANGLE, clockCircle);
```

Após este processo, caso seja detetada pelo menos uma Red Line, é escolhida a linha maior ficando o ponteiro dos segundos definido. Neste caso, aplica-se a máscara red à imagem para remover o ponteiro dos segundos e a nova imagem obtida é passada como argumento à função iterativeLinesSearch, para a deteção dos outros 2 ponteiros. Caso contrário, passa-se a imagem resultante da aplicação do Canny para a deteção dos 3 ponteiros (a existirem).

```
  // ...
  Mat imageHourSecond = result;
  if (hasRedLine) {
    result.setTo(cv::Scalar(0, 0, 0), redMask);
```

```
    imageShow("result after red mask", result);
  }
  // mergedClockLines is sorted in desc order of line norm
  auto mergedClockLines =
      iterativeLinesSearch(programData, clockCircle, imageHourSecond, 2);
  if (hasRedLine) {
    cout << "Encontrei red line!" << endl;
    if (mergedClockLines.size() >= 2) {
      return TimeLines(mergedClockLines[1], mergedClockLines[0], redLine);
    } else if (mergedClockLines.size() == 1) {
      return TimeLines(mergedClockLines[0], mergedClockLines[0], redLine);
    }
  } else {
    if (mergedClockLines.size() == 2) {
      return TimeLines(mergedClockLines[1], mergedClockLines[0]);
    } else if (mergedClockLines.size() == 1) {
      return TimeLines(mergedClockLines[0], mergedClockLines[0]);
    } else if (mergedClockLines.size() > 2) {
      return TimeLines(mergedClockLines[2], mergedClockLines[1],
                       mergedClockLines[0]);
    }
  }
  return TimeLines();
}
```

Para facilitar a determinação dos ponteiros e eliminar linhas que não são relevantes, criou-se uma função de filtragem, selectLinesCloseToCircleCenter, que remove todas as linhas que não tenham um ponto dentro e outro fora de um raio definido como uma percentagem do raio do *clock circle*. Esta função permite remover os "rabos"dos ponteiros (que podem levar a cálculos incorretos) e linhas que não se encontrem na zona esperada dos ponteiros.

Na função clockPointerLinesMerge são agrupadas as linhas que têm um ângulo inferior a linesMergeAngle. Para cada cluster, é calculada a maior norma e a média dos ângulos desse grupo. Estes valores são usados para definir uma nova linha que terá um ponto no centro do círculo e o outro dado por

```
Point2d directionVector = Point2d(cos(midAngle), sin(midAngle)) * maxNorm;
Point2d newPointB = directionVector + clockCircle.center;
```

O cálculo do ângulo médio é dado por:

$$vecAng = \text{atan2}(vec.y, vec.x)$$
$$midAngle = \text{atan2}(\sum \sin(vecAng), \sum \cos(vecAng))$$

Esta técnica de merging de clusters permite tirar proveito da simetria dos ponteiros. No entanto, há alguns *edge cases* que poderiam ser melhorados, tais como ponteiros bicudos do lado do centro com um grande ângulo. Foi feita uma tentativa de melhoramento, usando uma técnica de computação gráfica, *"Intersection of two lines in three-space"by Ronald Goldman, published in Graphics Gems, page 304* adaptada a 2d que caracteriza dois segmentos de retas: COLLINEAR_OVERLAPPING, COLLINEAR_DISJOINT, PARALLEL, SEGMENTS_INTERSECTING, SEGMENTS_NOT_INTERSECTING.

A ideia seria fazer merges distintos dada a diferente relação entre segmentos. No entanto, não houve tempo para maturar a solução e só foi possível fazer a implementação da função de caracterização:

```
SegmentsType segmentsAnalysis(Line &line1, Line &line2,
                              Point2d &intersectingPoint,
                              double doubleEqualityInterval);
```

O cálculo das horas é feito convertendo o ângulo clockwise das retas encontradas para os respetivos valores das horas, minutos e segundos. Neste cálculo foi tido em consideração as características geométricas dos ponteiros dos relógios de modo a remover erros por causa de falta de precisão das *edges*. Por exemplo, caso se tenha 50 minutos e o ponteiro das horas estiver para lá da uma mas muito perto desta hora (por exemplo 15% dos 100%, entre a uma e as duas) sabe-se que é mais provável o valor real ser 12:50 e não 1:50.

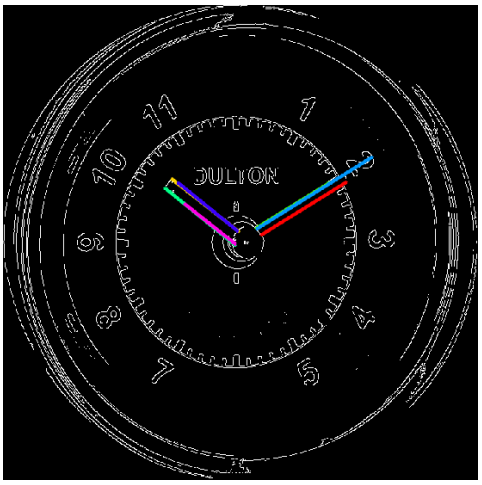

Figura 7: Linhas detetadas perto do centro, mas antes de serem agrupadas pelo algoritmo de clustering definido anteriormente

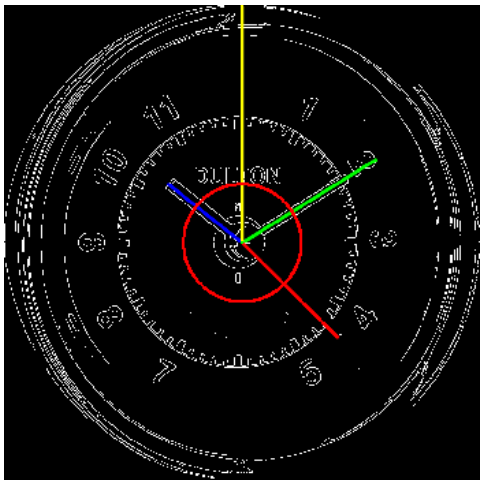

Figura 8: Resultado do cálculo das horas. O ponteiro das horas é identificado pela linha azul, o dos minutos a verde e dos segundos a vermelho.

4 Conclusão

A realização deste projeto demonstrou o grau de dificuldade na elaboração de aplicações que aplicam técnicas de visão por computador, mais concretamente de processamento e análise de imagens, mesmo em casos de estudos tão simples e limitados como o proposto para este projeto. A diversidade da natureza dos elementos que se pretende estudar e dos ambientes em que se inserem torna difícil a criação de um algoritmo que seja capaz de processar e analisar todos os casos com sucesso.

A aplicação proposta demonstrou um sucesso superior a 90% das imagens testadas dentro das restrições especificadas anteriormente, sendo este valor um pouco superior se o próprio utilizador alterar os thresholds na interface para se adequarem melhor à imagem em estudo.

Como possíveis melhorias à solução proposta poder-se-ia desenvolver algoritmos de segmentação de imagem mais complexos, fazendo uso de técnicas à base clustering ou histogramas que permitissem uma segmentação da face do relógio em contextos mais diversos. Seria também interessante remover algumas restrições ao problema como possibilitar uma maior variedade de ponteiros e faces de relógio.

A Ferramentas e Instruções de compilação

Utilizámos a linguagem C++14 e a biblioteca OpenCV. Existem 3 maneiras de compilar:

- makefile - Linux

```
make (debug|release)
```

- python - Linux

```
vcom.py build [-h] -s SOURCE_FILE [SOURCE_FILE ...] [-o OUTPUT_FILE
    ]
                [-a ARGUMENT [ARGUMENT ...]] [-r | -d] [-x] [-v]
```

- cmake - Linux e Windows

```
# For Windows
# Install OpenCV and extract it to the folder opencv/ in your
    project root
# Might be needed to change set(OpenCV_DIR "./opencv/build/x64/vc14
    /lib")
# Add the absolute path to "./opencv/build/x64/vc14/bin" or similar
     to
# Properties > Advanced System Settings > Environment Variables
    under path
https://sourceforge.net/projects/opencvlibrary/files/opencv-win
    /3.3.1/opencv-3.3.1-vc14.exe/download
# If using Visual studio 2017
# File > Open > CMake and open the CMakeLists.txt file
# Else install cmake
https://cmake.org/files/v3.9/cmake-3.9.4-win64-x64.msi
# And pick a generator from https://cmake.org/cmake/help/v3.9/
    manual/cmake-generators.7.html
mkdir cmake-build-debug && cd cmake-build-debug && cmake -
    DCMAKE_BUILD_TYPE=Debug -G "Visual Studio 14 2015 Win64" ../
# Then open the solution
```

B Interface

Após lançada a aplicação, é apresentada a seguinte interface ao utilizador:

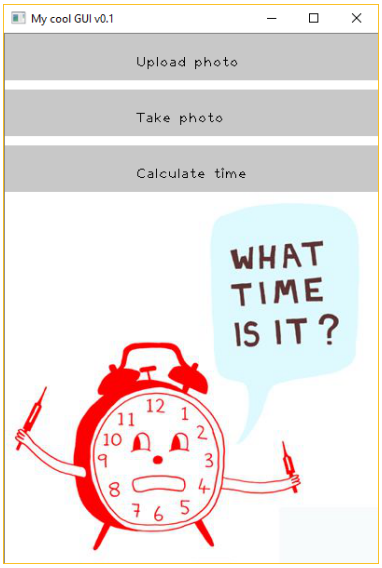

Figura 9: Interface da aplicação

O utilizador tem a possibilidade de:

- *Upload Photo* - especificar o caminho da imagem que pretende analisar pela linha de comandos;
- *Take Photo* - liga a câmera *default* e permite tirar uma foto;
- *Calculate Time* - corre o algoritmo de cálculo das horas, para a imagem escolhida.

No caso de utilizador escolher *Take Photo*, tem os seguintes comandos à sua disponibilidade:

- *p* - tira uma foto;
- *y* - aceita a foto obtida;
- *n* - descarta a foto obtida e permite tirar uma nova.

C Source Code

C.1 Clock Logic

```cpp
#define _USE_MATH_DEFINES
#include <algorithm>
#include <cmath>
#include <iostream>
#include <sstream>

#include <opencv2/highgui.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/imgproc.hpp>

#include "clock.hpp"

using namespace std;
using namespace cv;

// TODO : m.realease() in all Mat objects
const string DEFAULT_IMAGE_PATH = "../data/clock22.jpg";

const string WINDOW_NAME = "Clock Time Detection";
const string HOUGH_CIRCLES_CANNY_THRESHOLD_TRACKBAR_NAME =
    "HoughCircles Canny Threshold";
const string HOUGH_CIRCLES_ACCUMULATOR_THRESHOLD_TRACKBAR_NAME =
    "HoughCircles Accumulator Threshold";
const string HOUGH_LINES_P_TRACKBAR_NAME = "HoughLinesP Threshold";

const string BILATERAL_SIGMA_COLOR_TRACKBAR_NAME = "Bilateral Sigma Color";

const string BILATERAL_SIGMA_SPACE_TRACKBAR_NAME = "Bilateral Sigma Space";

const string CANNY_THRESHOLD1_TRACKBAR_NAME = "Canny Threshold 1";

const string CANNY_THRESHOLD2_TRACKBAR_NAME = "Canny Threshold 2";

const string CANNY_APERTURE_SIZE_TRACKBAR_NAME = "Canny Aperture Size";

const string DOUBLE_EQUALITY_INTERVAL_RADIUS_PERCENTAGE_TRACKBAR_NAME =
    "Double Equality Interval Radius Percentage";

constexpr int DEFAULT_HOUGH_CIRCLES_CANNY_THRESHOLD = 60;
constexpr int MIN_HOUGH_CIRCLES_CANNY_THRESHOLD = 1;
constexpr int MAX_HOUGH_CIRCLES_CANNY_THRESHOLD = 255;

constexpr int DEFAULT_HOUGH_CIRCLES_ACCUMULATOR_THRESHOLD = 100;
constexpr int MIN_HOUGH_CIRCLES_ACCUMULATOR_THRESHOLD = 1;
```

```
constexpr int MAX_HOUGH_CIRCLES_ACCUMULATOR_THRESHOLD = 255;

constexpr int DEFAULT_HOUGH_LINES_P_THRESHOLD = 83;
constexpr int MIN_HOUGH_LINES_P_THRESHOLD = 0;
constexpr int MAX_HOUGH_LINES_P_THRESHOLD = 155;

constexpr int MAX_BILATERAL_SIGMA = 300;
constexpr int DEFAULT_BILATERAL_SIGMA_COLOR = 25;
constexpr int DEFAULT_BILATERAL_SIGMA_SPACE = 50;

constexpr int MAX_CANNY_TRESHOLD = 255;
constexpr int DEFAULT_CANNY_THRESHOLD1 = 50;
constexpr int DEFAULT_CANNY_THRESHOLD2 = 120;
constexpr int MAX_CANNY_APERTURE_SIZE = 7;
constexpr int DEFAULT_CANNY_APERTURE_SIZE = 3;

constexpr double DEFAULT_LINES_MERGE_ANGLE =
    0.0874f;  // 5º rad, temp to sec 0.045f

constexpr double DEFAULT_LINES_SELECTION_RADIUS_FACTOR = 0.25;

struct ProgramData {
  Mat origImg;
  Mat grayImage;
  Mat imgCropped;
  Mat grayImageCropped;
  int houghCirclesCannyThreshold = DEFAULT_HOUGH_CIRCLES_CANNY_THRESHOLD;
  int houghCirclesAccumulatorThreshold =
      DEFAULT_HOUGH_CIRCLES_ACCUMULATOR_THRESHOLD;
  int houghLinesPThreshold = DEFAULT_HOUGH_LINES_P_THRESHOLD;
  int bilateralSigmaColor = DEFAULT_BILATERAL_SIGMA_COLOR;
  int bilateralSigmaSpace = DEFAULT_BILATERAL_SIGMA_SPACE;
  int cannyThreshold1 = DEFAULT_CANNY_THRESHOLD1;
  int cannyThreshold2 = DEFAULT_CANNY_THRESHOLD2;
  int cannyApertureSize = DEFAULT_CANNY_APERTURE_SIZE;
};

struct Circle {
  Point2d center;
  double radius;

  explicit Circle(Vec3f &circleData) {
    center.x = circleData[0];
    center.y = circleData[1];
    radius = circleData[2];
  }
};
```

```
struct Line {
  Point2d a;  // center point
  Point2d b;  // edge point

  Line() = default;

  explicit Line(const Point2d &p1, const Point2d &p2) {
    a.x = p1.x;
    a.y = p1.y;
    b.x = p2.x;
    b.y = p2.y;
  }
  explicit Line(Vec4d rawLine) {
    a.x = rawLine[0];
    a.y = rawLine[1];
    b.x = rawLine[2];
    b.y = rawLine[3];
  }
};

enum class TimeLinesType { HOUR_MINUTE = 0, HOUR_MINUTE_SECOND, UNKNOWN };

struct TimeLines {
  TimeLinesType timesLinesType;
  Line hour, minute, second;

  TimeLines() { timesLinesType = TimeLinesType::UNKNOWN; }

  TimeLines(Line hour_, Line minute_) {
    timesLinesType = TimeLinesType::HOUR_MINUTE;
    hour = hour_;
    minute = minute_;
  }

  TimeLines(Line hour_, Line minute_, Line second_) {
    timesLinesType = TimeLinesType::HOUR_MINUTE_SECOND;
    hour = hour_;
    minute = minute_;
    second = second_;
  }
};

struct TimeExtracted {
  int hour;
  int minute;
  int secs;

  TimeExtracted() : hour(0), minute(0), secs(0) {}
```

```
  explicit TimeExtracted(int h, int m) : hour(h), minute(m), secs(0) {}
  explicit TimeExtracted(int h, int m, int s) : hour(h), minute(m), secs(s) {}
};

enum class SegmentsType {
  COLLINEAR_OVERLAPPING = 0,
  COLLINEAR_DISJOINT,
  PARALLEL,
  SEGMENTS_INTERSECTING,
  SEGMENTS_NOT_INTERSECTING
};

string readCommandLine(int argc, char **argv, string const &defaultImagePath);

void readImage(Mat &image, ProgramData &programData);

void buildGui(TrackbarCallback callback, ProgramData &programData);

void clockTimeDetector(int, void *);

vector<Circle> getCircles(ProgramData &programData);

TimeLines getPointerLines(Mat &result, ProgramData &programData,
                          const Circle &clockCircle);

vector<Line> iterativeLinesSearch(ProgramData &programData,
                                  const Circle &clockCircle, Mat &image,
                                  size_t numberOfLinesToBreak);

void isolateClock(Circle &clockCircle, Mat &image, Mat &clock);

vector<Line> selectLinesCloseToCircleCenter(vector<Line> &lines,
                                            const Circle &circle,
                                            double radiusFactor);
vector<Line> clockPointerLinesMerge(vector<Line> clockLines,
                                    double linesMergeAngle,
                                    const Circle &clockCircle);

TimeExtracted extractTime(TimeLines &timeLines, const Circle &circle);

ostream &operator<<(ostream &ostr, const TimeExtracted &time);
double angleBetweenTwoLines(const Point2d &vec1, const Point2d &vec2,
                            bool toDegree = true);
// https://stackoverflow.com/questions/563198/how-do-you-detect-where-two-line-segments-inters
SegmentsType segmentsAnalysis(Line &line1, Line &line2,
                              Point2d &intersectingPoint,
                              double doubleEqualityInterval);
bool intervalsIntersect(double t0, double t1, double x0, double x1);
```

```
double crossProduct2d(Point2d &vec1, Point2d &vec2);
bool doubleIsZero(double value, double interval);
bool doubleEquality(double value, double reference, double interval);
Point2d calcLineVec(Line &line);
void calcPointDisplacement(Point2d &start, Point2d &displacementVector,
                           double displacementFactor, Point2d &displacedPoint);
double clockWiseAngleBetweenTwoVectors(const Point2d &vec1,
                                       const Point2d &vec2);
double getHourFromAngleDeg(double angle);
double getMinuteSecFromAngleDeg(double angle);
void bgr2gray(Mat &src, Mat &dst);
void gray2bgr(Mat &src, Mat &dst);
Scalar getDistinctColor(size_t index, size_t numberOfDistinctColors);
void normalizeHoughCirclesCannyThreshold(int &value);
void normalizeHoughCirclesAccumulatorThreshold(int &value);
void normalizeHoughLinesPThreshold(int &value);
void swapPoints(Line &line);
double medianHist(Mat grayImage);
template <class T>
constexpr const T &clamp(const T &v, const T &lo, const T &hi);
Mat calculateRedMask(Mat input);

void initiate(Mat &guiImage) {
  ProgramData programData = ProgramData();
  readImage(guiImage, programData);
  buildGui(clockTimeDetector, programData);
  clockTimeDetector(0, &programData);
  waitKey(0);
}

void clockTimeDetector(int, void *rawProgramData) {
  auto *programData = static_cast<ProgramData *>(rawProgramData);

  vector<Circle> circles = getCircles(*programData);
  if (circles.empty()) {
    return;
  }

  for (auto &clockCircle : circles) {
    isolateClock(clockCircle, programData->origImg, programData->imgCropped);
    bgr2gray(programData->imgCropped, programData->grayImageCropped);

    Mat SecPointerMask;
    Mat display;
    TimeLines timeLines = getPointerLines(display, *programData, clockCircle);

    if (timeLines.timesLinesType != TimeLinesType::UNKNOWN) {
      gray2bgr(display, display);
```

```
      TimeExtracted hoursExtracted = extractTime(timeLines, clockCircle);

      cout << endl << hoursExtracted << endl;

      line(display, timeLines.hour.a, timeLines.hour.b, Scalar(255, 0, 0), 3,
           LINE_AA);
      line(display, timeLines.minute.a, timeLines.minute.b, Scalar(0, 255, 0),
           3, LINE_AA);

      if (timeLines.timesLinesType == TimeLinesType::HOUR_MINUTE_SECOND) {
        line(display, timeLines.second.a, timeLines.second.b, Scalar(0, 0, 255),
             3, LINE_AA);
      }

      Point2d limitPoint;
      limitPoint.x = clockCircle.center.x;
      limitPoint.y = clockCircle.center.y - clockCircle.radius;
      Line midNightLine = Line(clockCircle.center, limitPoint);
      line(display, midNightLine.a, midNightLine.b, Scalar(0, 255, 255), 3,
           LINE_AA);

      // circle center
      circle(display, clockCircle.center, 3, Scalar(0, 255, 0), -1, 8, 0);
      // circle outline
      circle(display, clockCircle.center,
             DEFAULT_LINES_SELECTION_RADIUS_FACTOR * clockCircle.radius,
             Scalar(0, 0, 255), 3, 8, 0);

      stringstream ss;
      ss << hoursExtracted;
      putText(display, ss.str(),
              cvPoint(display.rows * 0.025f, display.cols * 0.070f),
              CV_FONT_HERSHEY_SIMPLEX, ((double)display.rows) / 500.0f,
              cvScalar(0, 0, 255), ((double)display.rows) / 500.0f, CV_AA);
      imageShow(WINDOW_NAME, display);

      break;
    }
  }
}

vector<Circle> getCircles(ProgramData &programData) {
  Mat blurredImage;
  GaussianBlur(programData.grayImage, blurredImage, Size(9, 9), 2, 2);

  int tries = 0;
  cout << endl;
```

```
vector<Circle> circles;
do {
  programData.houghCirclesCannyThreshold =
      max((DEFAULT_HOUGH_CIRCLES_CANNY_THRESHOLD - tries * 5),
          MIN_HOUGH_CIRCLES_CANNY_THRESHOLD);

  std::vector<Vec3f> raw_circles;

  /*
  Mat canny_output;
  int thresh = 200;
  vector<vector<Point> > contours;
  vector<Vec4i> hierarchy;
  /// Detect edges using canny
  Canny(programData.grayImage, canny_output, thresh, thresh * 2, 3);

  /// Find contours
  findContours(canny_output, contours, hierarchy, CV_RETR_TREE,
  CV_CHAIN_APPROX_SIMPLE, Point(0, 0));
  /// Draw contours
  Mat drawing = Mat::zeros(programData.grayImage.size(), CV_8UC3);

  RNG rng(12345);
  vector<Point> approx;
  cout << "con " << contours.size() << endl;

  for (int i = 0; i< contours.size(); i++)
  {
  Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255),
  rng.uniform(0, 255));

  approxPolyDP(contours[i], approx, 0.01*arcLength(contours[i], true), true);
  //drawContours(drawing, contours, i, color, 2, 8, hierarchy, 0, Point());
  if (approx.size() > 15) {
  drawContours(drawing, contours, i, color, 2, 8, hierarchy, 0, Point());
  }
  }

  /// Show in a window
  namedWindow("Contours", CV_WINDOW_AUTOSIZE);
  imshow("Contours", drawing);

  bgr2gray(drawing, drawing);
  namedWindow("Contours2", CV_WINDOW_AUTOSIZE);
  imshow("Contours2", drawing);
  */
  HoughCircles(blurredImage, raw_circles, HOUGH_GRADIENT, 1,
               blurredImage.rows / 8, programData.houghCirclesCannyThreshold,
```

```
                programData.houghCirclesAccumulatorThreshold, 0, 0);

    circles.clear();
    for (auto &raw_circle : raw_circles) {
      circles.push_back(Circle(raw_circle));
    }
    cout << "circles " << circles.size() << endl;
    if (!circles.empty()) {
      break;
    }
  } while (++tries < 50);

  std::sort(circles.begin(), circles.end(),
            [](Circle const &a, Circle const &b) -> bool {
              return a.radius > b.radius;
            });
  return circles;
}

TimeLines getPointerLines(Mat &result, ProgramData &programData,
                          const Circle &clockCircle) {
  imageShow("before bilateral", programData.grayImageCropped);

  bilateralFilter(programData.grayImageCropped, result,
                  programData.bilateralSigmaColor,
                  programData.bilateralSigmaSpace, BORDER_DEFAULT);

  imageShow("after bilateral", result);

  Canny(result, result, programData.cannyThreshold1,
        programData.cannyThreshold2, programData.cannyApertureSize);

  imageShow("after canny", result);

  // extract red pointer of seconds
  Mat redMask = calculateRedMask(programData.imgCropped);
  int dilation_size = 2;
  imageShow("redmask without dilation", redMask);
  Mat element = getStructuringElement(
      MORPH_CROSS, Size(2 * dilation_size + 1, 2 * dilation_size + 1),
      Point(dilation_size, dilation_size));
  /// Apply the dilation operation
  dilate(redMask, redMask, element);
  imageShow("redmask with dilation", redMask);

  auto redMergedClockLines =
      iterativeLinesSearch(programData, clockCircle, redMask, 1);
  bool hasRedLine = !redMergedClockLines.empty();
```

```
  Line redLine;
  if (hasRedLine) {
    redLine = redMergedClockLines[0];
  }

  Mat imageHourSecond = result;
  if (hasRedLine) {
    result.setTo(cv::Scalar(0, 0, 0), redMask);
    imageShow("result after red mask", result);
  }

  auto mergedClockLines =
      iterativeLinesSearch(programData, clockCircle, imageHourSecond, 2);
  if (hasRedLine) {
    cout << "Encontrei red line!" << endl;
    if (mergedClockLines.size() >= 2) {
      return TimeLines(mergedClockLines[1], mergedClockLines[0], redLine);
    } else if (mergedClockLines.size() == 1) {
      return TimeLines(mergedClockLines[0], mergedClockLines[0], redLine);
    }
  } else {
    if (mergedClockLines.size() == 2) {
      return TimeLines(mergedClockLines[1], mergedClockLines[0]);
    } else if (mergedClockLines.size() == 1) {
      return TimeLines(mergedClockLines[0], mergedClockLines[0]);
    } else if (mergedClockLines.size() > 2) {
      return TimeLines(mergedClockLines[2], mergedClockLines[1],
                       mergedClockLines[0]);
    }
  }

  return TimeLines();
}

vector<Line> iterativeLinesSearch(ProgramData &programData,
                                  const Circle &clockCircle, Mat &image,
                                  size_t numberOfLinesToBreak) {
  vector<Vec4d> rawLines;
  vector<Line> mergedClockLines;
  vector<Line> bestMergedClockLinesYet;
  int tries = 0;
  cout << endl;
  do {
    programData.houghLinesPThreshold =
        (DEFAULT_HOUGH_LINES_P_THRESHOLD + tries * 5) %
        MAX_HOUGH_LINES_P_THRESHOLD;
    HoughLinesP(image, rawLines, 1, CV_PI / 180,
                programData.houghLinesPThreshold, 30, 10);
```

```
    cout << "try: " << tries << ", rawLines.size() = " << rawLines.size()
         << endl;

    vector<Line> lines;

    for (auto &rawLine : rawLines) {
      lines.push_back(Line(rawLine));
    }

    mergedClockLines = selectLinesCloseToCircleCenter(
        lines, clockCircle, DEFAULT_LINES_SELECTION_RADIUS_FACTOR);

    Mat linesCloseMat;
    image.copyTo(linesCloseMat);
    gray2bgr(linesCloseMat, linesCloseMat);

    for (size_t i = 0; i < mergedClockLines.size(); ++i) {
      Line mergedLine = mergedClockLines[i];
      line(linesCloseMat, mergedLine.a, mergedLine.b,
           getDistinctColor(i, mergedClockLines.size()), 3, LINE_AA);
    }

    imageShow("Lines after selectLinesCloseToCircleCenter", linesCloseMat);

    cout << "mergedClockLines.size() = " << mergedClockLines.size()
         << ", after selectLinesCloseToCircleCenter" << endl;

    mergedClockLines = clockPointerLinesMerge(
        mergedClockLines, DEFAULT_LINES_MERGE_ANGLE, clockCircle);

    cout << "mergedClockLines.size() = " << mergedClockLines.size()
         << ", after clockPointerLinesMerge" << endl;

    if (mergedClockLines.size() > bestMergedClockLinesYet.size()) {
      bestMergedClockLinesYet = mergedClockLines;
    }
    if (mergedClockLines.size() >= numberOfLinesToBreak) break;
  } while (++tries < 50);

  std::sort(bestMergedClockLinesYet.begin(), bestMergedClockLinesYet.end(),
            [](Line const &l1, Line const &l2) -> bool {
              return norm(l1.b - l1.a) > norm(l2.b - l2.a);
            });

  return bestMergedClockLinesYet;
}
```

```
void isolateClock(Circle &clockCircle, Mat &image, Mat &clock) {
  cv::Mat mask = cv::Mat::zeros(image.rows, image.cols, CV_8UC1);
  auto clockCircleRadiusInt = static_cast<int>(clockCircle.radius);
  circle(mask, clockCircle.center, clockCircleRadiusInt, Scalar(255, 255, 255),
         -1, LINE_8, 0);
  Mat temp;
  image.copyTo(temp, mask);

  // Setup a rectangle to define your region of interest
  cv::Rect2d myROI(clockCircle.center.x - clockCircle.radius,
                   clockCircle.center.y - clockCircle.radius,
                   2.0 * clockCircle.radius, 2.0 * clockCircle.radius);
  clockCircle.center.x = clockCircle.radius;
  clockCircle.center.y = clockCircle.radius;

  // Crop the full image to that image contained by the rectangle myROI
  // Note that this doesn't copy the data
  clock = temp(myROI);
}

vector<Line> selectLinesCloseToCircleCenter(vector<Line> &lines,
                                            const Circle &circle,
                                            double radiusFactor) {
  double clock_radius_limit = radiusFactor * circle.radius;
  vector<Line> clockPointerLines;

  for (auto &line : lines) {
    Point2d vec1 = line.a - circle.center;
    Point2d vec2 = line.b - circle.center;

    if (norm(vec1) <= clock_radius_limit && norm(vec2) > clock_radius_limit) {
      clockPointerLines.push_back(line);
    }
    if (norm(vec2) <= clock_radius_limit && norm(vec1) > clock_radius_limit) {
      swapPoints(line);
      clockPointerLines.push_back(line);
    }
  }
  return clockPointerLines;
}

vector<Line> clockPointerLinesMerge(vector<Line> clockLines,
                                    double linesMergeAngle,
                                    const Circle &clockCircle) {
  vector<Line> result;

  if (clockLines.empty() || clockLines.size() == 1) {
    return clockLines;
```

```
}

for (size_t x = 0; x < clockLines.size() - 1; x++) {
  Line l1 = clockLines[x];

  Point2d vec1 = calcLineVec(l1);
  double maxNorm = norm(l1.b - clockCircle.center);
  double vec1Angle = atan2(vec1.y, vec1.x);
  double sumCos = cos(vec1Angle);
  double sumSin = sin(vec1Angle);
  for (size_t y = x + 1; y < clockLines.size(); y++) {
    Line l2 = clockLines[y];

    Point2d vec2 = calcLineVec(l2);

    double vec1Vec2Angle = angleBetweenTwoLines(vec1, vec2, false);
    cout << x << " - " << vec1Vec2Angle << " - " << linesMergeAngle << endl;
    if (vec1Vec2Angle < linesMergeAngle) {
      double vec2Angle = atan2(vec2.y, vec2.x);
      sumCos += cos(vec2Angle);
      sumSin += sin(vec2Angle);
      cout << x << " - joined " << endl;
      maxNorm = max(maxNorm, norm(l2.b - clockCircle.center));

      clockLines.erase(clockLines.begin() + y);
      y--;
    } else {
      cout << x << " - passed " << endl;
    }
  }
  clockLines.erase(clockLines.begin() + x);
  x--;

  double midAngle = atan2(sumSin, sumCos);
  cout << "midAngle: " << midAngle << " sumCos " << sumCos << " sumSin "
       << sumSin << endl;
  Point2d directionVector = Point2d(cos(midAngle), sin(midAngle)) * maxNorm;
  Point2d newPointB = directionVector + clockCircle.center;

  Line clockAvgPointer(clockCircle.center, newPointB);
  result.push_back(clockAvgPointer);
  if (clockLines.empty()) break;
}
if (clockLines.size() == 1) {
  Line l1 = clockLines[0];
  Point2d vec1 = calcLineVec(l1);
  double maxNorm = norm(l1.b - clockCircle.center);
  double vec1Angle = atan2(vec1.y, vec1.x);
```

```
    cout << "last Pointer: " << vec1Angle << endl;
    Point2d directionVector = Point2d(cos(vec1Angle), sin(vec1Angle)) * maxNorm;
    Point2d newPointB = directionVector + clockCircle.center;

    Line clockAvgPointer(clockCircle.center, newPointB);
    result.push_back(clockAvgPointer);
  }
  return result;
}

TimeExtracted extractTime(TimeLines &timeLines, const Circle &circle) {
  if (timeLines.timesLinesType == TimeLinesType::UNKNOWN)
    return TimeExtracted();

  // determine mid night clock pointer to help determine the correct hour and
  // minute
  Point2d limitPoint;
  limitPoint.x = circle.center.x;
  limitPoint.y = circle.center.y - circle.radius;
  Line midNightLine = Line(circle.center, limitPoint);
  Point2d vecReference = midNightLine.b - midNightLine.a;

  Line hourLine = timeLines.hour;
  Line minuteLine = timeLines.minute;

  Point2d hourVec = calcLineVec(hourLine);
  Point2d minuteVec = calcLineVec(minuteLine);

  double hourAng = clockWiseAngleBetweenTwoVectors(vecReference, hourVec);
  double minuteAng = clockWiseAngleBetweenTwoVectors(vecReference, minuteVec);

  double approximateHour = getHourFromAngleDeg(hourAng);
  double approximateMinute = getMinuteSecFromAngleDeg(minuteAng);
  double approximateSecond = 0.0;

  if (timeLines.timesLinesType == TimeLinesType::HOUR_MINUTE_SECOND) {
    Line secondLine = timeLines.second;
    Point2d secondVec = calcLineVec(secondLine);
    double secondAng = clockWiseAngleBetweenTwoVectors(vecReference, secondVec);
    approximateSecond = getMinuteSecFromAngleDeg(secondAng);
  }

  double hourDecimalPart = approximateHour - static_cast<int>(approximateHour);
  double minutePercentage = approximateMinute / 60.0;

  auto hourInt = static_cast<int>(approximateHour);
  auto minuteInt = static_cast<int>(approximateMinute);
  auto secondInt = static_cast<int>(approximateSecond);
```

```
  if (hourDecimalPart <= 0.15 && minutePercentage > 0.75) {
    hourInt = static_cast<int>(approximateHour) - 1;
  } else if (hourDecimalPart > 0.85 && minutePercentage < 0.25) {
    hourInt = (static_cast<int>(approximateHour) + 1) % 12;
  }

  if (hourInt == 0) {
    hourInt = 12;
  } else if (hourInt == -1) {
    hourInt = 11;
  }

  return TimeExtracted(hourInt, minuteInt, secondInt);
}

// TODO: define for hours and degree
double getHourFromAngleDeg(double angle) { return angle * 12.0 / 360.0; }

double getMinuteSecFromAngleDeg(double angle) { return angle * 60.0 / 360.0; }

ostream &operator<<(ostream &ostr, const TimeExtracted &time) {
  ostr << "Hours: " << time.hour << ":" << time.minute << ":" << time.secs;
  return ostr;
}

double clockWiseAngleBetweenTwoVectors(const Point2d &vec1,
                                       const Point2d &vec2) {
  if (vec1 == vec2) {
    return 0;
  }
  double dot = vec1.dot(vec2);
  double det = vec1.x * vec2.y - vec1.y * vec2.x;  // determinant
  double ang = atan2(det, dot);  // atan2(y, x) or atan2(sin, cos)
  if (ang < 0) {
    ang = 2 * M_PI + ang;
  }
  return (ang * 180.0) / M_PI;
}

// TODO: verify if one of the norms is zero...
double angleBetweenTwoLines(const Point2d &vec1, const Point2d &vec2,
                            bool toDegree) {
  if (vec1 == vec2) {
    return 0;
  }
  double ang = acos(vec1.dot(vec2) / (norm(vec1) * norm(vec2)));
```

```
  if (toDegree) {
    return (ang * 180.0) / M_PI;
  }

  return ang;
}

SegmentsType segmentsAnalysis(Line &line1, Line &line2,
                              Point2d &intersectingPoint,
                              double doubleEqualityInterval) {
  Point2d q = line2.a;
  Point2d p = line1.a;
  Point2d qMp = q - p;
  Point2d r = calcLineVec(line1);
  Point2d s = calcLineVec(line2);
  double rXs = crossProduct2d(r, s);
  double qMpXr = crossProduct2d(qMp, r);

  if (doubleIsZero(rXs, doubleEqualityInterval) &&
      doubleIsZero(qMpXr, doubleEqualityInterval)) {
    double rN = norm(r);
    double rNSquared = rN * rN;
    double t0 = qMp.dot(r) / rNSquared;
    double t1 = t0 + s.dot(r) / rNSquared;
    if (s.dot(r) < 0) {
      swap(t0, t1);
    }
    if (intervalsIntersect(t0, t1, 0, 1)) {
      return SegmentsType::COLLINEAR_OVERLAPPING;
    } else {
      return SegmentsType::COLLINEAR_DISJOINT;
    }
  }

  if (doubleIsZero(rXs, doubleEqualityInterval) &&
      !doubleIsZero(qMpXr, doubleEqualityInterval)) {
    return SegmentsType::PARALLEL;
  }

  if (!doubleIsZero(rXs, doubleEqualityInterval)) {
    double t = crossProduct2d(qMp, s) / rXs;
    double u = qMpXr / rXs;
    if (t >= 0 && t <= 1 && u >= 0 && u <= 1) {
      calcPointDisplacement(p, r, t, intersectingPoint);
      return SegmentsType::SEGMENTS_INTERSECTING;
    }
  }
```

```
  return SegmentsType::SEGMENTS_NOT_INTERSECTING;
}

bool intervalsIntersect(double t0, double t1, double x0, double x1) {
  return max(t0, x0) <= min(t1, x1);
}

double crossProduct2d(Point2d &vec1, Point2d &vec2) {
  return vec1.x * vec2.y - vec1.y * vec2.x;
}

Point2d calcLineVec(Line &line) { return line.b - line.a; }

bool doubleIsZero(double value, double interval) {
  return doubleEquality(value, 0, interval);
}

bool doubleEquality(double value, double reference, double interval) {
  return value > reference - interval && value < reference + interval;
}

void calcPointDisplacement(Point2d &start, Point2d &displacementVector,
                           double displacementFactor, Point2d &displacedPoint) {
  displacedPoint.x = start.x + displacementVector.x * displacementFactor;
  displacedPoint.y = start.y + displacementVector.y * displacementFactor;
}

void swapPoints(Line &line) { swap(line.a, line.b); }

/*
 * Calculates the median color of a image from hist
 * TODO: Canny betweem 0.66*[median value] and 1.33*[median value]
 */
double medianHist(Mat grayImage) {
  // Initialize parameters
  int histSize = 256;  // bin size
  float range[] = {0, 255};
  const float *ranges[] = {range};

  // Calculate histogram
  MatND hist;
  calcHist(&grayImage, 1, nullptr, Mat(), hist, 1, &histSize, ranges, true,
           false);

  double m = grayImage.rows * grayImage.cols / 2;
  int bin = 0;
  double med = -1.0;
```

```
  for (int i = 0; i < histSize && med < 0.0; ++i) {
    bin += cvRound(hist.at<float>(i));
    if (bin > m && med < 0.0) med = i;
  }

  cout << "Median " << endl;

  return med;
}

void readImage(Mat &image, ProgramData &programData) {
  image.copyTo(programData.origImg);
  bgr2gray(programData.origImg, programData.grayImage);

  if (programData.origImg.empty()) {
    throw std::invalid_argument("Invalid Input Image");
  }
}

void buildGui(TrackbarCallback callback, ProgramData &programData) {
  createWindow(WINDOW_NAME, programData.origImg.rows, programData.origImg.cols);

  createTrackbar(HOUGH_CIRCLES_CANNY_THRESHOLD_TRACKBAR_NAME, WINDOW_NAME,
                 &programData.houghCirclesCannyThreshold,
                 MAX_HOUGH_CIRCLES_CANNY_THRESHOLD, callback, &programData);
  createTrackbar(HOUGH_CIRCLES_ACCUMULATOR_THRESHOLD_TRACKBAR_NAME, WINDOW_NAME,
                 &programData.houghCirclesAccumulatorThreshold,
                 MAX_HOUGH_CIRCLES_ACCUMULATOR_THRESHOLD, callback,
                 &programData);
  createTrackbar(BILATERAL_SIGMA_COLOR_TRACKBAR_NAME, WINDOW_NAME,
                 &programData.bilateralSigmaColor, MAX_BILATERAL_SIGMA,
                 callback, &programData);
  createTrackbar(BILATERAL_SIGMA_SPACE_TRACKBAR_NAME, WINDOW_NAME,
                 &programData.bilateralSigmaSpace, MAX_BILATERAL_SIGMA,
                 callback, &programData);
  createTrackbar(CANNY_THRESHOLD1_TRACKBAR_NAME, WINDOW_NAME,
                 &programData.cannyThreshold1, MAX_CANNY_TRESHOLD, callback,
                 &programData);
  createTrackbar(CANNY_THRESHOLD2_TRACKBAR_NAME, WINDOW_NAME,
                 &programData.cannyThreshold2, MAX_CANNY_TRESHOLD, callback,
                 &programData);
}

void createWindow(string windowName, int rows, int cols) {
  if (rows > 400 || cols > 400) {
    namedWindow(windowName, WINDOW_NORMAL);
    resizeWindow(windowName, 400, 400);
  } else {
```

```
    namedWindow(windowName, WINDOW_AUTOSIZE);
  }
}

void imageShow(string windowName, Mat &image) {
  createWindow(windowName, image.rows, image.cols);
  imshow(windowName, image);
}

void normalizeHoughCirclesCannyThreshold(int &value) {
  clamp(value, MIN_HOUGH_CIRCLES_CANNY_THRESHOLD,
        MAX_HOUGH_CIRCLES_CANNY_THRESHOLD);
}

void normalizeHoughCirclesAccumulatorThreshold(int &value) {
  clamp(value, MIN_HOUGH_CIRCLES_ACCUMULATOR_THRESHOLD,
        MAX_HOUGH_CIRCLES_ACCUMULATOR_THRESHOLD);
}

void normalizeHoughLinesPThreshold(int &value) {
  clamp(value, MIN_HOUGH_LINES_P_THRESHOLD, MAX_HOUGH_LINES_P_THRESHOLD);
}

template <class T>
constexpr const T &clamp(const T &v, const T &lo, const T &hi) {
  return std::max(lo, std::min(v, hi));
}

void bgr2gray(Mat &src, Mat &dst) { cvtColor(src, dst, COLOR_BGR2GRAY); }

void gray2bgr(Mat &src, Mat &dst) { cvtColor(src, dst, COLOR_GRAY2BGR); }

Scalar getDistinctColor(size_t index, size_t numberOfDistinctColors) {
  Mat bgr;
  Mat hsv(1, 1, CV_8UC3,
          Scalar(static_cast<double>(index * 179 / numberOfDistinctColors), 255,
                 255));
  cvtColor(hsv, bgr, CV_HSV2BGR);
  return Scalar(bgr.data[0], bgr.data[1], bgr.data[2]);
}

Mat calculateRedMask(Mat input) {
  Mat hsv_image;

  cvtColor(input, hsv_image, cv::COLOR_BGR2HSV);
  Mat lower_red_hue_range;
  Mat upper_red_hue_range;
  inRange(hsv_image, cv::Scalar(0, 100, 100), cv::Scalar(10, 255, 255),
```

```
        lower_red_hue_range);
  inRange(hsv_image, cv::Scalar(150, 100, 100), cv::Scalar(179, 255, 255),
        upper_red_hue_range);

  Mat redMask = lower_red_hue_range | upper_red_hue_range;

  return redMask;
}
```

C.2 Interface

```
#define _USE_MATH_DEFINES
#include <algorithm>
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>

#include "clock.hpp"

using namespace cv;
using namespace std;

const string LOGO_PATH = "../data/logo.jpg";

const int DEFAULT_CAMERA = 0;

const int NO_OPER = -1;
const int UPLOAD_PHOTO_OPER = 0;
const int TAKE_VIDEO_PHOTO_OPER = 1;
const int CALCULATE_CLOCK_OPER = 2;

Mat3b canvas;
string buttonUploadText("Upload photo");
string buttonVideoText("Take photo");
string buttonClockText("Calculate time");
string winName = "My cool GUI v0.1";
int oper = NO_OPER;
int state = 0;

Rect buttonUpload;
Rect buttonVideo;
Rect buttonClock;

Mat input;

void askForFile();
int TakeOneFrame();
int calculateClock();
void callBackFunc(int event, int x, int y, int, void*);

void askForFile() {
  ifstream infile;

  cout << "Please enter the input file name> " << flush;
  while (true) {
```

```
    string imagePath;
    getline(cin, imagePath);
    input = imread(imagePath.c_str(), IMREAD_COLOR);

    if (!input.empty()) {
      imageShow("Image", input);
      state = 0;
      break;
    }
    cout << "Invalid file. Please enter a valid input file name> " << flush;
  }
}

int TakeOneFrame() {
  VideoCapture cap(DEFAULT_CAMERA);

  // Check if camera opened successfully
  if (!cap.isOpened()) {
    cout << "Error opening video stream or file" << endl;
    return -1;
  }

  Mat frame;

  while (true) {
    cap >> frame;
    frame.copyTo(input);
    imageShow("Video Feed", frame);
    if (char(waitKey(1)) == 'p') {
      imageShow("Image", input);
      bool retake = false;

      while (!retake) {
        if (char(waitKey(1)) == 'y') {  // Accept photo
          destroyWindow("Video Feed");
          state = 0;  // accept new input
          return 0;
        }

        else if (char(waitKey(1)) == 'n') {  // Retake photo
          retake = true;
          destroyWindow("Image");
        }
      }
    }
  }
}
```

```
int calculateClock() {
  state = 0;  // accept new input
  initiate(input);
  return 0;
}

void callBackFunc(int event, int x, int y, int, void*) {
  if (event == EVENT_LBUTTONDOWN) {
    if (buttonUpload.contains(Point(x, y))) {
      cout << "Clicked Upload!" << endl;
      auto buttonUploadCanvas = canvas(buttonUpload);
      rectangle(buttonUploadCanvas, buttonUpload, Scalar(0, 0, 255), 2);
      oper = UPLOAD_PHOTO_OPER;
      state = 1;
    }

    if (buttonVideo.contains(Point(x, y))) {
      cout << "Clicked Video!" << endl;
      auto buttonVideoCanvas = canvas(buttonVideo);
      rectangle(buttonVideoCanvas, buttonVideo, Scalar(0, 0, 255), 2);
      oper = TAKE_VIDEO_PHOTO_OPER;
      state = 1;
    }

    if (buttonClock.contains(Point(x, y))) {
      cout << "Clicked Clock!" << endl;
      auto buttonClockCanvas = canvas(buttonClock);
      rectangle(buttonClockCanvas, buttonClock, Scalar(0, 0, 255), 2);
      oper = CALCULATE_CLOCK_OPER;
      state = 1;
    }
  }
  if (event == EVENT_LBUTTONUP && state == 1) {
    rectangle(canvas, buttonUpload, Scalar(200, 200, 200), 2);
    rectangle(canvas, buttonVideo, Scalar(200, 200, 200), 2);
    rectangle(canvas, buttonClock, Scalar(200, 200, 200), 2);

    state = 2;
  }

  imshow(winName, canvas);
  waitKey(1);

  if (oper == UPLOAD_PHOTO_OPER && state == 2) {
    askForFile();
    oper = NO_OPER;
  } else if (oper == TAKE_VIDEO_PHOTO_OPER && state == 2) {
    TakeOneFrame();
```

```
    oper = NO_OPER;
  } else if (oper == CALCULATE_CLOCK_OPER && state == 2) {
    calculateClock();
    oper = NO_OPER;
  }
}

int main() {
  Mat img = imread(LOGO_PATH, IMREAD_COLOR);

  // Your button
  buttonUpload = Rect(0, 0, img.cols, 50);
  buttonVideo = Rect(0, 50 + 10, img.cols, 50);
  buttonClock = Rect(0, 100 + 20, img.cols, 50);

  // The canvas
  canvas = Mat3b(img.rows + 3 * buttonUpload.height + 2 * 10, img.cols,
                 Vec3b(255, 255, 255));

  // Draw the buttons
  //---------------------------------------------------
  canvas(buttonUpload) = Vec3b(200, 200, 200);
  putText(canvas(buttonUpload), buttonUploadText,
          Point2d(buttonUpload.width * 0.35, buttonUpload.height * 0.7),
          FONT_HERSHEY_PLAIN, 1, Scalar(0, 0, 0));

  canvas(buttonVideo) = Vec3b(200, 200, 200);
  putText(canvas(buttonVideo), buttonVideoText,
          Point2d(buttonVideo.width * 0.35, buttonVideo.height * 0.7),
          FONT_HERSHEY_PLAIN, 1, Scalar(0, 0, 0));

  canvas(buttonClock) = Vec3b(200, 200, 200);
  putText(canvas(buttonClock), buttonClockText,
          Point2d(buttonClock.width * 0.35, buttonClock.height * 0.7),
          FONT_HERSHEY_PLAIN, 1, Scalar(0, 0, 0));

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

  // Draw the image
  img.copyTo(
      canvas(Rect(0, 3 * buttonUpload.height + 2 * 10, img.cols, img.rows)));

  // Setup callback function
  namedWindow(winName);
  setMouseCallback(winName, callBackFunc);

  imshow(winName, canvas);
```

```
  while (true) {
    char key = static_cast<char>(cv::waitKey(30));
    if (key == 27) break;
  }

  return 0;
}
```