CCTag

AliceVision

Oct 18, 2022

INSTALL

This library provides the code for the detection of CCTag markers made up of concentric circles [CGGG16]. CCTag markers are a robust, highly accurate fiducial system that can be robustly localized in the image even under challenging conditions. The library can efficiently detect the position of the image of the (common) circle center and identify the marker based on the different ratio of their crown sizes.

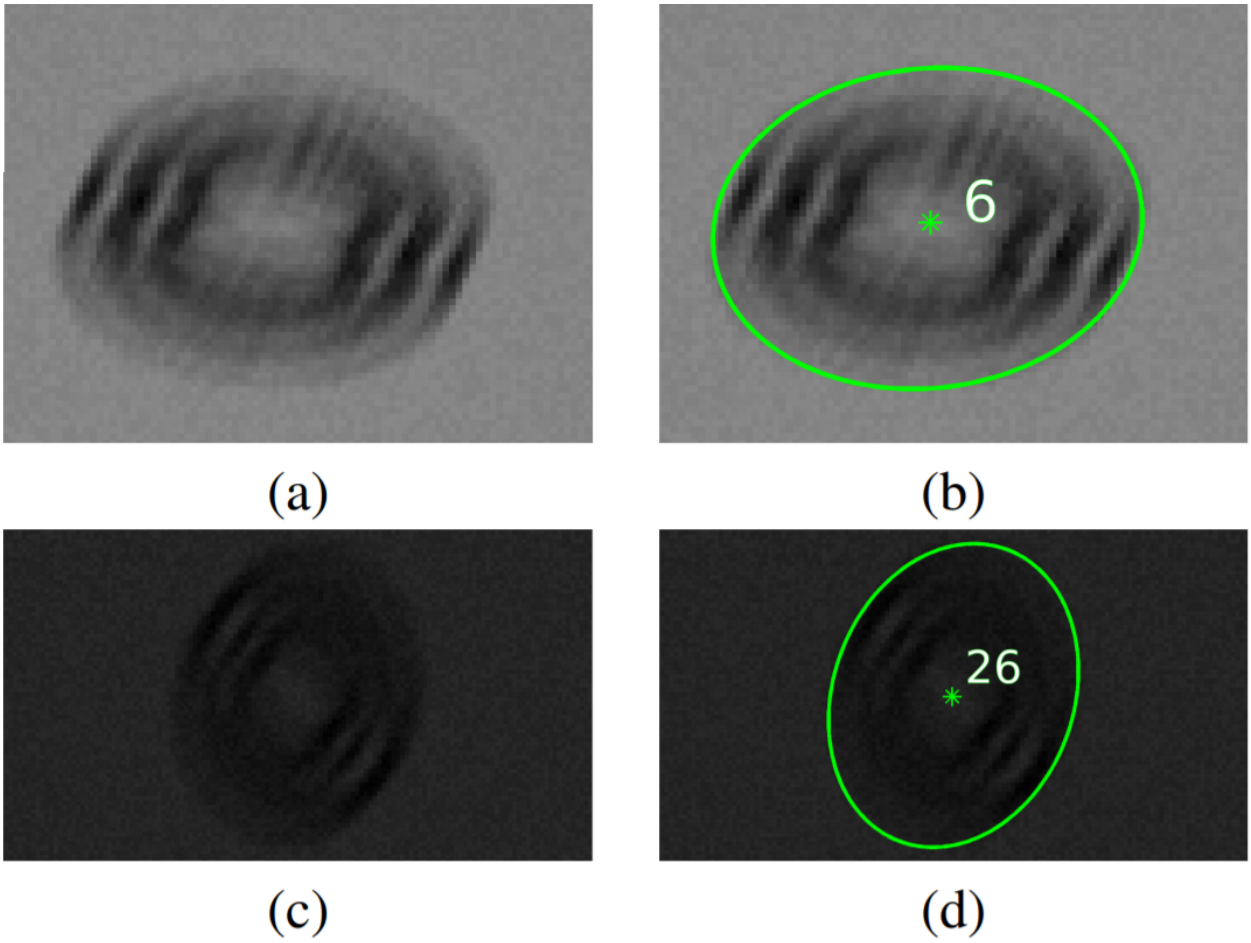
An example of three different CCTag markers with three crowns. Each marker can be uniquely identified thanks to the thickness of each crown, which encodes the information of the marker, typically a unique ID.

The implementation is done in both CPU and GPU (Cuda-enabled cards). The GPU implementation can reach real-time performances on full HD images.

CHAPTER

ONE

EXAMPLE OF DETECTION IN CHALLENGING CONDITIONS



Examples of synthetic images of circular fiducials under very challenging shooting conditions i.e., perturbed, in particular, by a (unidirectional) motion blur of magnitude 15px. The markers are correctly detected and identified (b,d) with an accuracy of 0.54px and 0.36px resp. in (a) and (c) for the estimated imaged center of the outer ellipse whose semi-major axis (in green) is equal to 31.9px and 34.5px resp.

CHAPTER

TWO

COMPARISON WITH ARTOOLKITPLUS

The video shows the effectiveness and the robustness of the detection compared to the ARToolKitPlus [WS07]. ARTKPlus is, among all the available open-source solutions, one achieving better performances in terms of detection rate and computational time.

In the video, 4 markers for each solution are placed on a plane at known positions, so that the relevant plane-induced homography can be estimated. For ARTKPlus once all the markers are detected and identified, the homography is estimated using all the detected marker corners following a DLT approach [HZ04] (note that the homography can be then estimated even if only one marker is detected). For CCTAG, the image of the four centres of the concentric circles is used to compute the homography.

The image placed in between the markers can be then rectified in order to visually assess the quality of the estimated homography. Thanks to the accurate estimation of the image of the four centres of the concentric circles provided by CCTag, the homography can be robustly estimated and the rectified image is not affected by any significant jittering, whereas the rectified image computed with the ARTKPlus homography is more unstable. Moreover, the video shows that the proposed method allows detecting the marker even in very challenging conditions, such as severe motion blur and sudden illumination changes.

2.1 Requirements

2.1.1 Hardware

CCTag has a CPU and a GPU implementation. The GPU implementation requires an NVIDIA GPU card with a CUDA compute capability >= 3.5. You can check your NVIDIA GPU card CC support here or on the NVIDIA dev page. If you do not have a NVIDIA card you will still able to compile and use the CPU version.

Here are the minimum hardware requirements for CCTag:

Minimum requirements	
Operating systems	Windows x64, Linux, macOS
CPU	Recent Intel or AMD cpus
RAM Memory	8 GB
Hard Drive	No particular requirements
GPU	NVIDIA CUDA-enabled GPU (compute capability >= 3.5)

2.1.2 Software

CCTag depends on the following libraries:

- Eigen3 >= 3.3.4
- Boost >= 1.66
- OpenCV >= 3.1
- TBB >= 2021.5.0

> **Warning:** In order to have Cuda support on Windows, at least Eigen 3.3.9 is required

---

CCTag can be installed from the following package managers.

2.2 vcpkg

vcpkg is a cross-platform (Windows, Linux and MacOS), open-source package manager created by Microsoft.

Since v1.0.0 of the library it is possible to build and install the library through vcpkg on Linux, Windows and MacOS by running:

```
vcpkg install cctag[cuda,apps]
```

where `cuda` and `apps` are the options to build the library with the cuda support and the sample applications, respectively.

2.3 conan

conan is a decentralized and multi-platform package manager.

Since v1.0.1, you can install CCTag from conan by running:

```
conan install cctag/1.0.1@
```

where `1.0.1@` is the version you want to install. See *CCTag as third party* for how to use CCTag as third party in your project.

2.4 Building the library

2.4.1 Building tools

Required tools:

- CMake >= 3.14 to build the code
- Git

- C/C++ compiler supporting the C++14 standard (gcc >= 5, clang >= 3.4, msvc >= 2017)

Optional tool:

- CUDA >= 9.0

Note: On Windows, there are compatibility issues to build the GPU part due to conflicts between msvc/nvcc/thrust/eigen/boost.

2.4.2 Dependencies

vcpkg

vcpkg can be used to install all the dependencies on all the supported platforms. This is particularly useful on Windows. To install the dependencies:

```
vcpkg install
        boost-accumulators
        boost-algorithm
        boost-container
        boost-date-time
        boost-exception
        boost-filesystem
        boost-iterator
        boost-lexical-cast
        boost-math
        boost-mpl
        boost-multi-array
        boost-ptr-container
        boost-program-options
        boost-serialization
        boost-spirit
        boost-static-assert
        boost-stacktrace
        boost-test
        boost-thread
        boost-throw-exception
        boost-timer
        boost-type-traits
        boost-unordered
        opencv
        tbb
        eigen3
```

You can add the flag `--triplet` to specify the architecture and the version you want to build. For example:

- `--triplet x64-windows` will build the dynamic version for Windows 64 bit
- `--triplet x64-windows-static` will build the static version for Windows 64 bit
- `--triplet x64-linux-dynamic` will build the dynamic version for Linux 64 bit

and so on. More information can be found here

Linux

On Linux you can install from the package manager:

For Ubuntu/Debian package system:

```
sudo apt-get install g++ git-all libpng12-dev libjpeg-dev libeigen3-dev libboost-all-dev␣
↪libtbb-dev
```

For CentOS package system:

```
sudo yum install gcc-c++ git libpng-devel libjpeg-turbo-devel eigen3-devel boost-devel  ␣
↪    tbb-devel
```

MacOS

On MacOs using Homebrew install the following packages:

```
brew install git libpng libjpeg eigen boost tbb
```

2.4.3 Getting the sources

```
git clone https://github.com/alicevision/CCTag.git
```

2.4.4 CMake configuration

From CCTag root folder you can run cmake:

```
mkdir build && cd build
cmake ..
make -j `nproc`
```

On Windows add `-G "Visual Studio 16 2019" -A x64` to generate the Visual Studio solution according to your VS version (see CMake documentation).

If you are using the dependencies built with VCPKG you need to pass `-DCMAKE_TOOLCHAIN_FILE=path/to/vcpkg/scripts/buildsystems/vcpkg.cmake` at cmake step to let it know where to find the dependencies.

Otherwise you can specify the path where each dependency can be found (if not installed in system folders) by passing its related path. For example, for OpenCV you can pass `-DOpenCV_DIR=path/to/opencv/install/share/OpenCV/` to tell where the `OpenCVConfig.cmake` file can be found.

CMake options

CMake configuration can be controlled by changing the values of the following variables (here with their default value)

- `CCTAG_WITH_CUDA:BOOL=ON` to enable/disable the Cuda implementation
- `BUILD_SHARED_LIBS:BOOL=ON` to enable/disable the building shared libraries
- `CCTAG_ENABLE_SIMD_AVX2:BOOL=OFF` to enable/disable the AVX2 optimizations
- `CCTAG_BUILD_TESTS:BOOL=OFF` to enable/disable the building of the unit tests
- `CCTAG_BUILD_APPS:BOOL=ON` to enable/disable the building of applications
- `CCTAG_BUILD_DOC:BOOL=OFF` to enable/disable building this documentation

So if you do not want to build the Cuda part, you have to pass `-DCCTAG_WITH_CUDA:BOOL=OFF` and so on.

2.5 CCTag as third party

When you install CCTag a file `CCTagConfig.cmake` is installed in `<install_prefix>/lib/cmake/CCTag/` that allows you to import the library in your CMake project. In your `CMakeLists.txt` file you can add the dependency in this way:

```
# Find the package from the CCTagConfig.cmake
# in <prefix>/lib/cmake/CCTag/. Under the namespace CCTag::
# it exposes the target CCTag that allows you to compile
# and link with the library
find_package(CCTag CONFIG REQUIRED)
...
# suppose you want to try it out in a executable
add_executable(cctagtest yourfile.cpp)
# add link to the library
target_link_libraries(cctagtest PUBLIC CCTag::CCTag)
```

Then, in order to build just pass the location of `CCTagConfig.cmake` from the cmake command line:

```
cmake .. -DCCTag_DIR=$CCTAG_INSTALL/lib/cmake/CCTag/
```

If you are using conan for your project then you need to add cctag to your `conanfile.txt`:

```
[requires]
cctag/1.0.1

[generators]
CMakeToolchain
CMakeDeps
```

and when building you may need to follow these steps:

```
mkdir build
cd build
conan install .. -s build_type=Release
cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
cmake --build . --config Release
```

2.6 Docker image

A docker image can be built using the Ubuntu based `Dockerfile`, which is based on nvidia/cuda image (https://hub.docker.com/r/nvidia/cuda/)

2.6.1 Building the dependency image

We provide a `Dockerfile_deps` containing a cuda image with all the necessary CCTag dependencies installed.

A parameter `CUDA_TAG` can be passed when building the image to select the cuda version. Similarly, `OS_TAG` can be passed to select the Ubuntu version. By default, `CUDA_TAG=10.2` and `OS_TAG=18.04`

For example to create the dependency image based on ubuntu 18.04 with cuda 8.0 for development, use

```
docker build --build-arg CUDA_TAG=8.0 --tag alicevision/cctag-deps:cuda8.0-ubuntu18.04 -
↪f Dockerfile_deps .
```

The complete list of available tags can be found on the nvidia dockerhub page

2.6.2 Building the CCTag image

Once you built the dependency image, you can build the cctag image in the same manner using `Dockerfile`:

```
docker build --tag alicevision/cctag:cuda8.0-ubuntu18.04 .
```

2.6.3 Running the CCTag image

In order to run the image nvidia docker is needed: see the installation instruction. Once installed, the docker can be run, e.g., in interactive mode with

```
docker run -it --runtime=nvidia alicevision/cctag:cuda8.0-ubuntu18.04
```

2.6.4 Official images on DockeHub

Check the docker hub CCTag repository for the available images.

2.7 Library usage

Detecting the markers requires three main entities:

- the class *cctag::CCTag* modeling a single marker
- and the functions *cctag::cctagDetection()* to process the images and get the list of detected markers.
- the struc *cctag::Parameters* that control the detection algorithm through the various parameters that it exposes.

2.7.1 Detection

Here is a minimal sample of code that enable CCTag detection on an image:

```
// set up the parameters
const std::size_t nCrowns{3};
cctag::Parameters params(nCrowns);
// if you want to use GPU
params.setUseCuda(true);

// load the image e.g. from file
cv::Mat src = cv::imread(image_filename);
cv::Mat graySrc;
cv::cvtColor(src, graySrc, CV_BGR2GRAY);

// choose a cuda pipe
const int pipeId{0};

// an arbitrary id for the frame
const int frameId{0};

// process the image
boost::ptr_list<ICCTag> markers{};
cctagDetection(markers, pipeId, frameId, graySrc, params);
```

@TODO maybe explain what cuda pipe means

2.7.2 Process detected markers

Here is a simple example on how to process the detected markers. The function `drawMarkers` takes the list of detected markers and it overlay their information on the original image. From the list of markers, if the detected marker is valid it draws the center of the marker, its ID and the outer ellipse *cctag::numerical::geometry::Ellipse*, all in green. If the marker is not valid, draw the center and the ID in red.

```
void drawMarkers(const boost::ptr_list<ICCTag>& markers, cv::Mat& image)
{
    // drawing settings
    const int radius{10};
    const int fontSize{3};
    const int thickness{3};
    const int fontFace{cv::FONT_HERSHEY_SIMPLEX};

    for(const auto& marker : markers)
    {
        // center of the marker
        const cv::Point center = cv::Point(marker.x(), marker.y());
        const auto rescaledOuterEllipse = marker.rescaledOuterEllipse();

        // check the status and draw accordingly, green for valid, red otherwise
        if(marker.getStatus() == status::id_reliable)
        {
            const cv::Scalar color = cv::Scalar(0, 255, 0, 255);
            // draw the center
```

(continues on next page)

(continued from previous page)

```
            cv::circle(image, center, radius, color, thickness);
            // write the marker ID
            cv::putText(image, std::to_string(marker.id()), center, fontFace, fontSize,␣
↪color, thickness);
            // draw external ellipse
            cv::ellipse(image,
                        center,
                        cv::Size(rescaledOuterEllipse.a(), rescaledOuterEllipse.b()),
                        rescaledOuterEllipse.angle() * 180 / boost::math::constants::pi
↪<double>(),
                        0,
                        360,
                        color,
                        thickness);
        }
        else
        {
            // the same for invalid markers but in red
            const cv::Scalar color = cv::Scalar(0, 0, 255, 255);
            cv::circle(image, center, radius, color, thickness);
            cv::putText(image, std::to_string(marker.id()), center, fontFace, fontSize,␣
↪color, thickness);
        }
    }
}
```

Here is an example of possible result:

2.8 API References

2.8.1 Main Classes

struct **Parameters**

Structure containing all the major parameters using in the *CCTag* detection algorithms.

Public Functions

explicit **Parameters**(std::size_t nCrowns = kDefaultNCrowns)

The constructor, normally the most interesting parameter is the number of crowns.

Parameters
nCrowns – The number of crowns that the markers to detect are made up of.

template<class **Archive**>
inline void **serialize**(*Archive* &ar, unsigned int version)

Serialize the parameter settings.

Template Parameters
Archive – The class to use to store the data.

Parameters

- **ar** – **[inout]** The object where to store the data.
- **version** – **[in]** The serialization version.

void **setDebugDir**(const std::string &debugDir)

Set the debug directory where debug data is stored.

Parameters
debugDir – **[in]**

void **setUseCuda**(bool val)

Whether to use the Cuda implementation or not.

Note: Ignored if the code is not built with Cuda support.

Parameters
val – **[in]** `true` to use the Cuda implementation, `false` to use the CPU.

class **CCTag** : public cctag::ICCTag

Class modeling the *CCTag* marker containing the position of the marker in the image, its ID and its status.

Public Functions

inline virtual float **x**() const override

Ger the x coordinate of the center of the marker.

Returns
x coordinate of the center.

inline virtual float **y**() const override

Ger the x coordinate of the center of the marker.

Returns
x coordinate of the center.

inline virtual const cctag::numerical::geometry::*Ellipse* &**rescaledOuterEllipse**() const override

Get the rescaled outer ellipse of the marker. The rescaled outerEllipse is in the coordinate system of the input image, while the internal ellipse is relative to a pyramid level.

Returns
the outer ellipse.

inline virtual MarkerID **id**() const override

Get marker ID.

Returns
the marker ID.

inline virtual int **getStatus**() const override

Get the status of the marker.

Returns
the status of the marker.

2.8.2 Functions

void **cctag::cctagDetection**(boost::ptr_list<ICCTag> &markers, int pipeId, std::size_t frame, const cv::Mat &graySrc, std::size_t nRings, logtime::*Mgmt* *durations, const std::string &parameterFilename, const std::string &cctagBankFilename)

Perform the *CCTag* detection on a gray scale image.

Parameters

- **markers** – **[out]** Detected markers. WARNING: only markers with status == 1 are valid ones. (status available via getStatus())
- **pipeId** – **[in]** Choose between several CUDA pipeline instances
- **frame** – **[in]** A frame number. Can be anything (e.g. 0).
- **graySrc** – **[in]** Gray scale input image.
- **nRings** – **[in]** Number of *CCTag* rings.
- **durations** – **[in]** Optional object to store execution times.
- **parameterFilename** – **[in]** Path to a parameter file. If not provided default parameters will be used.
- **cctagBankFilename** – **[in]** Path to the cctag bank. If not provided, radii will be the ones associated to the CCTags contained in the markersToPrint folder.

void **cctag::cctagDetection**(boost::ptr_list<ICCTag> &markers, int pipeId, std::size_t frame, const cv::Mat &graySrc, const cctag::*Parameters* &params, logtime::*Mgmt* *durations = nullptr, const CCTagMarkersBank *pBank = nullptr)

Perform the *CCTag* detection on a gray scale image.

Parameters

- **markers** – **[out]** Detected markers. WARNING: only markers with status == 1 are valid ones. (status available via getStatus())
- **pipeId** – **[in]** Choose between several CUDA pipeline instances
- **frame** – **[in]** A frame number. Can be anything (e.g. 0).
- **graySrc** – **[in]** Gray scale input image.
- **params** – **[in]** *Parameters* for the detection.
- **durations** – **[in]** Optional object to store execution times.

- **pBank** – **[in]** Path to the cctag bank. If not provided, radii will be the ones associated to the CCTags contained in the markersToPrint folder.

2.8.3 Utility Classes

class **Ellipse**

It models an ellipse with standard form $\frac{x^2-x_c}{a^2} + \frac{y^2-y_c}{b^2} = 1$, centered in `_center` (x_c, x_y) and rotated clock-wise by `_angle` wrt the x-axis. Note that, arbitrarly, the representation with the major axis aligned with the y-axis is chosen.

Subclassed by cctag::numerical::geometry::Circle

Public Functions

Ellipse() = default

Default constructor, set all parameters to zero.

explicit **Ellipse**(const Matrix &matrix)

Build an ellipse from a 3x3 matrix representing the ellipse as a conic.

Note: By default, the representation with the major axis aligned with the y-axis is chosen.

Parameters

matrix – **[in]** The 3x3 matrix representing the ellipse.

Ellipse(const Point2d<Eigen::Vector3f> &center, float a, float b, float angle)

Build an ellipse from a set of parameters.

Parameters

- **center** – **[in]** The center of the conic.
- **a** – **[in]** The length of the semi-axis x.
- **b** – **[in]** The length of the semi-axis y.
- **angle** – **[in]** The orientation of the ellipse wrt the x-axis as a clock-wise angle in radians.

inline const Matrix &**matrix**() const

Return the matrix representation of the ellipse.

Returns

3x3 matrix representation of the ellipse.

inline Matrix &**matrix**()

Return the matrix representation of the ellipse.

Returns

3x3 matrix representation of the ellipse.

inline const Point2d<Eigen::Vector3f> &**center**() const

Return the center of the ellipse.

Returns

3 element vector with the homogeneous coordinates of the ellipse.

inline Point2d<Eigen::Vector3f> &**center**()

Return the center of the ellipse.

Returns
3 element vector with the homogeneous coordinates of the ellipse.

inline float **a**() const

Return the length of the x-semi axis of the ellipse.

Returns
the length of the x-semi axis of the ellipse.

inline float **b**() const

Return the length of the y-semi axis of the ellipse.

Returns
the length of the y-semi axis of the ellipse.

inline float **angle**() const

Return the orientation of the ellipse.

Returns
the clock-wise orientation angle in radians of the ellipse wrt the x-axis

void **setA**(float a)

Set the length of the x-semi axis of the ellipse.

Parameters
a – **[in]** the length of the x-semi axis.

void **setB**(float b)

Set the length of the y-semi axis of the ellipse.

Parameters
b – **[in]** the length of the y-semi axis.

void **setAngle**(float angle)

Set the orientation angle of the ellipse.

Parameters
angle – **[in]** the clock-wise orientation angle in radians.

void **setCenter**(const Point2d<Eigen::Vector3f> &center)

Set the center of the ellipse.

Parameters
center – **[in]** the new center of the ellipse.

void **setMatrix**(const Matrix &matrix)

Update the ellipse from a matrix representing a conic.

Parameters
matrix – **[in]** 3x3 matric representing the ellipse.

void **setParameters**(const Point2d<Eigen::Vector3f> &center, float a, float b, float angle)

Update the ellipse from its parameters.

Parameters

- **center** – **[in]** The center of the conic.
- **a** – **[in]** The length of the semi-axis x.

- **b** – **[in]** The length of the semi-axis y.
- **angle** – **[in]** The orientation of the ellipse wrt the x-axis as a clock-wise angle in radians.

Ellipse **transform**(const Matrix &mT) const

Return a new ellipse obtained by applying a transformation to the ellipse.

Parameters

mT – **[in]** a 3x3 matrix representing the transformation.

Returns

the transformed ellipse.

void **getCanonicForm**(Matrix &mCanonic, Matrix &mTprimal, Matrix &mTdual) const

Compute the canonical form of the conic, along with its transformation.

Parameters

- **mCanonic** – **[out]** 3x3 diagonal matrix representing the ellipse in canonical form.
- **mTprimal** – **[out]** 3x3 transformation matrix such that C = `mTprimal.transpose()` * `mCanonic` * `mTprimal`
- **mTdual** – **[out]** 3x3 inverse transformation matrix (= `mTprimal.inv()`)

Friends

friend std::ostream &**operator<<**(std::ostream &os, const *Ellipse* &e)

Print the ellipse in matrix form in Matlab notation.

Parameters

- **os** – **[inout]** the stream where to output the ellipse.
- **e** – **[in]** the ellipse

Returns

the stream with appended the matrix representation of the ellipse.

struct **Mgmt**

class **Measurement**

2.9 Markers usage

You can find the pdf of the marker to use in the `markersToPrint` of the project root directory.

2.9.1 Print the markers

We recommend to print the markers on a hard, flat and matt surface.

The size of the marker can be chosen considering the minimum size of the marker image that can be detected. The image of the marker should be roughly no less than 30 pixel of radius for the external ring. The size of the actual marker to print can be computed considering the distance of the camera w.r.t the marker, the focal length and the resolution of the image.

To **roughly** estimate the (minimum) radius R of the marker to print you can use the formula:

$$R = \frac{m\,u}{f} d$$

where:

- m is minimum size in pixel for the radius (e.g. 30 pixel)
- u is the pixel size in mm (that can be found on the specs of the camera)
- f is the focal length in mm
- d is the distance between the camera and the marker.

For example, for the marker to have a $m = 75$ pixels radius using a camera with a pixel size of $u = 0.00434$ mm and a focal length of $f = 24$ mm and seeing the marker from a distance of $d = 5$ m, the estimated radius of the actual marker to print is $R = \frac{75 * 0.00434}{24}\ 5000 = 67.81$ mm.

2.9.2 Generate the markers

In the `markersToPrint` directory you can also find a python program `generate.py` to generate the svg file of the markers. You can customize the size and print the id of the marker on the corner.

Here is the usage and options:

```
usage: generate.py [-h] [--rings N] [--outdir dir] [--margin N] [--radius N]
                   [--addId] [--addCross] [--generatePng] [--generatePdf]
                   [--whiteBackground]

Generate the svg file for the markers.

optional arguments:
  -h, --help          show this help message and exit
  --rings N           the number of rings (possible values {3, 4}, default: 3)
  --outdir dir        the directory where to save the files (default: ./)
  --margin N          the margin to add around the external ring (default: 400)
  --radius N          the radius of the outer circle (default: 500)
  --addId             add the marker id on the top left corner
  --addCross          add a small cross in the center of the marker
  --generatePng       also generate a png file
  --generatePdf       also generate a pdf file
  --whiteBackground   set the background (outside the marker) to white instead
                      of transparent
```

For example, calling:

```
./generate.py --outdir markers3 --margin 100 --addId
```

it will create a directory `markers3` where it saves an svg file for each marker with a margin around the marker of 100 and with the ID of the marker printed on the top left corner.

To generate pdf and/or png file, use the flags `--generatePdf` and `--generatePng` .

2.10 About

2.10.1 License

CCTag is licensed under MPLv2 license.

More info about the license and what you can do with the code can be found at tldrlegal website

2.10.2 Contact us

You can contact us on the public mailing list at alicevision@googlegroups.com

You can also contact us privately at alicevision-team@googlegroups.com

2.10.3 Cite us

If you want to cite this work in your publication, please use the following

```
@inproceedings{calvet2016Detection,
  TITLE = {{Detection and Accurate Localization of Circular Fiducials under Highly␣
↪Challenging Conditions}},
  AUTHOR = {Calvet, Lilian and Gurdjos, Pierre and Griwodz, Carsten and Gasparini,␣
↪Simone},
  BOOKTITLE = {{Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern␣
↪Recognition (CVPR)}},
  ADDRESS = {Las Vegas, United States},
  PAGES = {562 - 570},
  YEAR = {2016},
  MONTH = Jun,
  DOI = {10.1109/CVPR.2016.67}
}
```

2.10.4 Acknowledgements

This has been developed in the context of the European project POPART founded by European Union's Horizon 2020 research and innovation programme under grant agreement No 644874.

Additional contributions for performance optimizations have been funded by the Norwegian RCN FORNY2020 project FLEXCAM.

2.11 Bibliography

BIBLIOGRAPHY

[CGC12] L. Calvet, P. Gurdjos, and V. Charvillat. Camera tracking using concentric circle markers: paradigms and algorithms. In *2012 19th IEEE International Conference on Image Processing*. IEEE, September 2012. doi:10.1109/icip.2012.6467121.

[CGGG16] Lilian Calvet, Pierre Gurdjos, Carsten Griwodz, and Simone Gasparini. Detection and Accurate Localization of Circular Fiducials under Highly Challenging Conditions. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 562 – 570. Las Vegas, United States, June 2016. URL: https://hal.archives-ouvertes.fr/hal-01420665/document, doi:10.1109/CVPR.2016.67.

[HZ04] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2004.

[WS07] Daniel Wagner and Dieter Schmalstieg. ARToolKitPlus for Pose Tracking on Mobile Devices. In *Computer Vision Winter Workshop (CVWW)*, 139–146. 2007. URL: http://www.icg.tu-graz.ac.at/Members/daniel/ARToolKitPlusMobilePoseTracking.

INDEX

C