OpenTL: An object-oriented library for model-based visual tracking

Technische Universität München - Fakultät für Informatik
Chair for Robotics and Embedded Sytems

The Visual Tracking Team
<opentl-dev@mailknoll.in.tum.de>

July 31, 2009

Contents

1 Installing *OpenTL* **2**
1.1 Hardware requirements . . . 2
1.2 Installation Steps for Microsoft Windows XP . . . 3
1.3 Installation Steps for Ubuntu Jaunty . . . 6
1.4 Software run-time dependencies . . . 7
1.5 Software compile-time dependencies . . . 7
1.6 Using CMake to create a Visual Studio Solution . . . 8

2 Tutorial application: a color-based particle filter **10**
2.1 Step 1: Camera input and video output . . . 10
2.1.1 FireWire camera input . . . 11
2.1.2 USB camera input . . . 13
2.2 Step 2: Pose representation and screen projection . . . 14
2.3 Step 3: Shape and appearance model . . . 18
2.4 Step 4: Set up the color-based likelihood . . . 20
2.5 Step 5: Set up the particle filter and track the object . . . 26
2.6 Step 6: Tracking multiple targets . . . 28
2.7 Step 7: Measurement fusion with another modality . . . 29

3 The modular structure of *OpenTL* **33**

Abstract

OpenTL is a general-purpose software library for model-based object tracking, written in C++.

OpenTL allows simultaneous tracking of multiple targets, with different degrees of freedom, using multiple cameras and visual modalities, as well as representing all of the available model information in a seamless and user-friendly class interface, for a wide variety of algorithms and systems.

OpenTL is implemented in a hierarchical and modular fashion, where the most relevant features are derived from common abstractions in order to achieve a highly modular, scalable and parallelizable architecture. Within this framework, several state-of-the-art computer vision and tracking methodologies can be readily implemented and tested.

In this report, an overview of the *OpenTL* architecture will be given, as well as a step-by-step tutorial for learning how to build the first object tracking application.

Chapter 1

Installing *OpenTL*

1.1 Hardware requirements

In order to use the *OpenTL* library, your hardware should meet the minimum system requirements shown in table 1.1. Altough the library was developed platform-independently, it currently supports installation on Windows XP and Ubuntu Linux (Jaunty). The processor(s) must be able to deal with MMX, SSE, SSE2 instructions, which is the case for almost all modern CPUs.

Some processing steps of the library are directly performed on the graphics card (GPU) in order to obtain real-time performance. Depending on the subset of delivered library modules a modern graphics card will be required, which conforms at least to the OpenGL specification 2.1 and provides the following OpenGL extensions:

- GL_EXT_gpu_shader4
- GL_ARB_texture_rectangle
- GL_ARB_draw_buffers

These GPU extensions are usually provided by a GeForce series 8 (8600 GT and upper) card. The supported extensions of any OpenGL-capable graphics card can be determined by looking at the output of the *glewinfo* command, that is

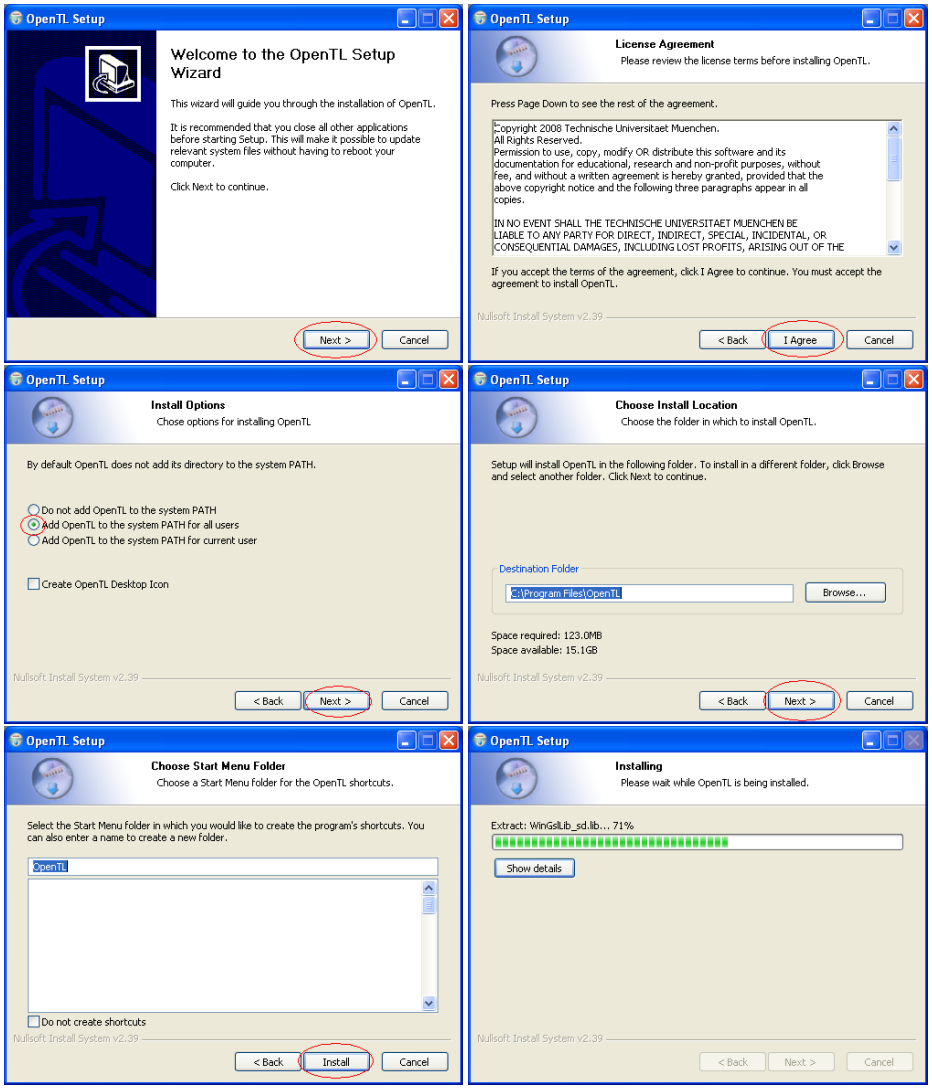
	Minimum	Recommended
Processor	Intel 2 Ghz single core	Intel > 2 Ghz Core Duo
Memory	1GB	2GB
Graphics card	NVIDIA GeForce 8600	NVIDIA GeForce 9800
Operating Systems	Microsoft Windows XP/Vista 32bit or Ubuntu Jaunty Linux (32/64bit)	
Free disk space	300MB (*OpenTL* and 3rd-party dependencies)	
Internet connection	For installation only (download of 3rd-party dependencies)	

Table 1.1: System hardware requirements

included in the Glew library package that -only for Windows- is also shipped within the *OpenTL* installer.

1.2 Installation Steps for Microsoft Windows XP

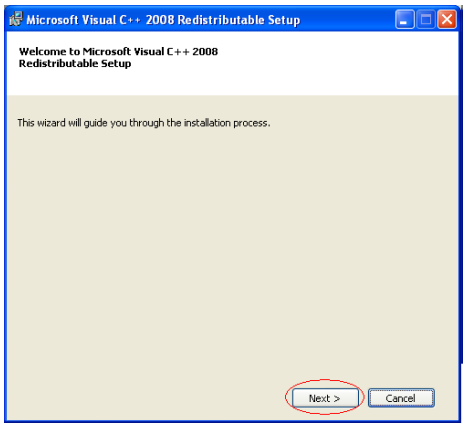
In order to install *OpenTL* on Windows XP, run the *OpenTL-0.8.0-Windows-x86.exe* executable, which contains an installer that guides you through the installation process. Each step of the installer is depicted below.



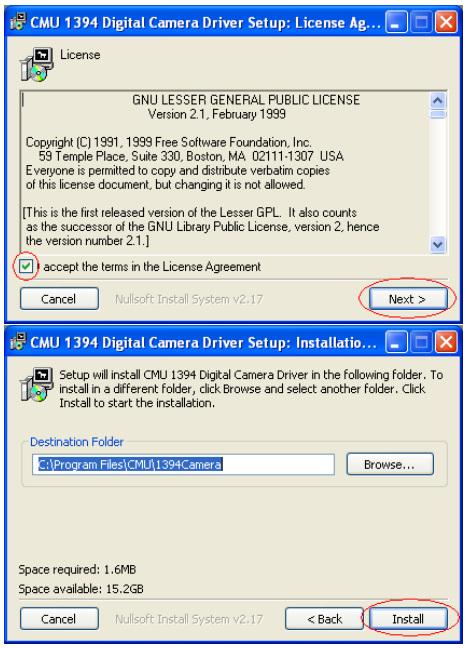
Now the *OpenTL* software is installed on your system, and Setup will ask to

install the required 3rd-party dependencies. In case you have already installed any of these dependencies, you can at any time press "Cancel" in order to skip a specific dependency.

The first dependency installed is the Visual Studio Redistributable package[1].



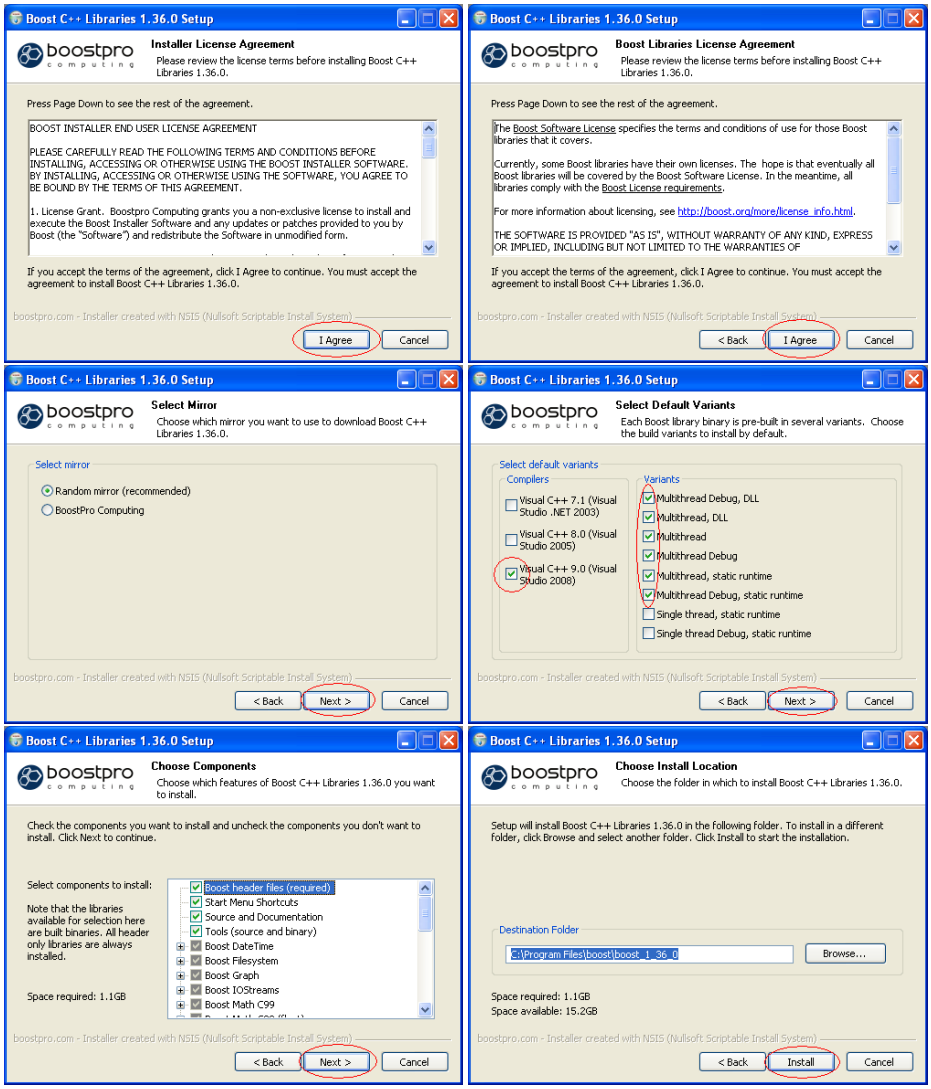
The next dialog will ask you to install the CMU Firewire camera drivers.



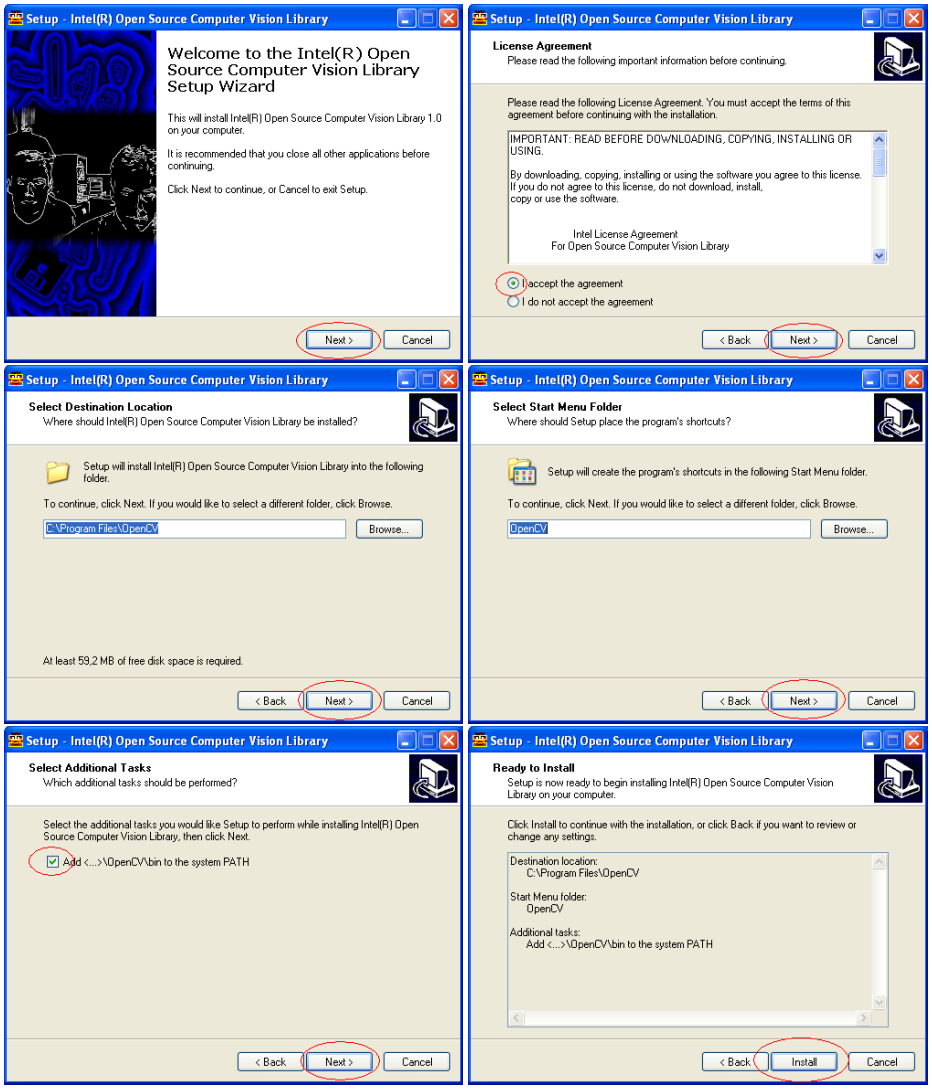
Afterwards, the Setup dialog will ask you to install the *Boost C++* library. Please make sure to select the right options, which are depicted in the screen-

[1]Note: you can skip this package if you already have Visual Studio 2008 installed.

shots below.



Finally, the last Setup dialog will ask you to install the OpenCV library.



After the Setup installation, your system should be ready to run the pre-compiled *OpenTL* tutorial applications.

However, in order to develop and compile code using the library, you will also need to install the compile-time dependencies of section 1.5.

1.3 Installation Steps for Ubuntu Jaunty

The *OpenTL* package for Linux is provided as a Debian package (OpenTL-0.8.0-Linux-*.deb). In order to install the library the dependencies have to be installed first. This can be done by executing the following command as *root*

[t]

Dependency	Version	Default install path	Shipped
Boost	1.36	PrgFiles\Boost	yes(as downloader)
CMU1394 drivers	6.4.5	PrgFiles\CMU\1394Camera	yes(as installer)
OpenCV	1.0	PrgFiles\OpenCV	yes(as installer)
Glew	1.4.0	*OpenTL*\externals	yes
Freeglut	2.4.0	*OpenTL*\externals	yes
DevIL		*OpenTL*\externals	yes
WinGSL		*OpenTL*\externals	yes
NVIDIA Cg Toolkit	optional	NONE	optional
OpenGL	>= 2.1.1	with GPU driver	no, GPU driver

Table 1.2: Run-time dependencies (Windows)

user:

```
apt-get install freeglut3-dev libboost-serialization1.37-dev \
libboost-thread1.37-dev, libc6-dev libcv-dev libcvaux-dev \
libdc1394-22-dev libdevil-dev libgcc1 libglew1.5-dev \
libgsl0-dev libhighgui-dev libraw1394-dev libstdc++6
```

Afterwards, the provided *OpenTL* Debian archive can be installed by executing the following command (still as *root*):

```
dpkg -i OpenTL-0.8.0-Linux*.deb
```

In order to access the camera device, please make also sure that the permissions of the `/dev/raw*` and `/dev/video*` files are appropriately set for the developing user.

1.4 Software run-time dependencies

Figure 1.2 shows the run-time dependencies for Windows and Figure 1.3 shows the ones for Linux.

On Windows, all required dependencies are either included in the *OpenTL* installer, or will be downloaded during the installation process. The Linux version relies on dependencies provided by the Ubuntu software repository. For the installation process, see section 1.2(respectively, 1.3)[2].

1.5 Software compile-time dependencies

Required build and repository tools (Windows)

- Microsoft Visual Studio 2008: C/C++ programming environment used for developing *OpenTL* and its applications

[2]Note for Linux Users: Install the `*-dev` packages, if you plan to develop with *OpenTL*.

Dependency	Version	Shipped with *OpenTL*
Boost	1.37	no, in Ubuntu repository
OpenCV	1.0	no, in Ubuntu repository
Glew	1.4.0	no, in Ubuntu repository
Freeglut	2.4.0	no, in Ubuntu repository
DevIL		no, in Ubuntu repository
GSL		no, in Ubuntu repository
libdc1394	2.x	no, in Ubuntu repository
libraw1394		no, in Ubuntu repository
NVIDIA Cg Toolkit		no, installed with GPU driver (NVIDIA)
OpenGL	>= 2.1.1	no, installed with GPU driver (NVIDIA)

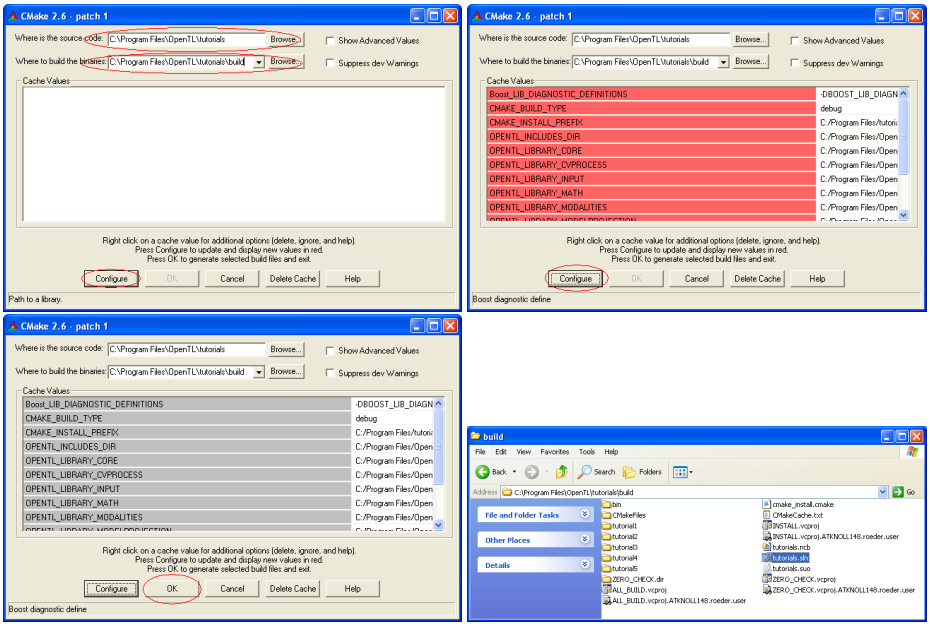
Table 1.3: Run-time dependencies (Linux)

- CMake 2.6.2: Tool to generate platform-independent Makefiles and Projects for both Windows and Linux compilers

Required build and repository tools (Linux)

- GCC: C++ compiler under Linux
- CMake 2.6.2: (see above)

1.6 Using CMake to create a Visual Studio Solution



In order to build a Visual Studio *Solution* with CMake, the following main steps must be performed after installing *OpenTL* (ref. to the 4 pictures above):

1. Specify where the source code and output build directories for the tutorial applications can be found. Usually the latter is a `build/` subfolder within the former.
2. Click on `Configure` in order to configure the CMake internal variables (containing paths to library sub-modules, external dependencies, etc.)
3. If the previous step has been successfully done, click `OK` in order to generate the makefiles and Visual Studio solution
4. The generated solution file should be in the `build` directory above specified, with the same name of the folder (`tutorials.sln`)

Chapter 2

Tutorial application: a color-based particle filter

OpenTL provides all of the desirable *building blocks* for visual tracking, concerning:

1. Prior models
2. Computer vision
3. Bayesian tracking

In order to introduce the user to the basic features of *OpenTL*, we start with a relatively simple application: a color-based particle filter. This tutorial describes how to achieve the goal in five main steps, meanwhile introducing the relevant library classes and methods.

2.1 Step 1: Camera input and video output

The very first step for a visual tracking application concerns grabbing frames from the camera, and displaying the video signal on-line (Fig. 2.1).

OpenTL currently supports IEEE-1394 FireWire camera inputs as well as standard Webcams, from one or multiple devices, both for Linux and Windows OS. In particular, webcam inputs are handled through the OpenCV camera functions.

In *OpenTL* we provide a main abstraction `opentl::input::ImageSensor`, from which we derive the following classes:

- `opentl::input::FirewireCamera`: IEEE-1394 FireWire cameras, for both Windows and Linux OS
  - `opentl::input::LinuxDC1394Camera`

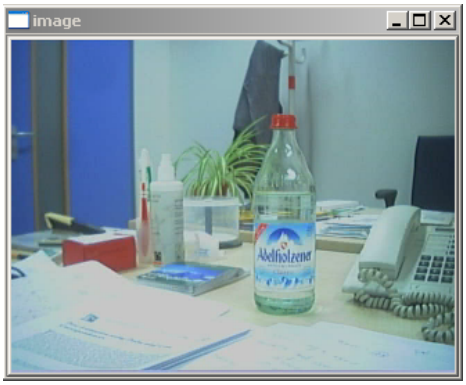


Figure 2.1: Tutorial 1: capture and display on-line the camera stream.

  - `opentl::input::WindowsCMU1394Camera`

- `opentl::input::OpenCVCamera`: standard Webcams, handled through the OpenCV capture functions

In order to describe both type of cameras, for this tutorial we provide two versions. In the tutorial code, they can be selected at startup, by specifying a command line parameter. For the tutorial applications using cameras, one command line option has been added to switch the type of camera to use, such that the user may specify the camera type at start up by

`./tutorial# -camType <GUPPY|SONY|OPENCV>`

In order to display all available command line options for a tutorial application, the user may call the application with the `help` option:

`./tutorial# -help`

The output for *tutorial 1*, when starting with the `help` option, is shown in Figure 2.2

```
Command line options:
Option          Default          Current          Description
-------------------------------------------------------------------------------
-help          (*)false          true             display a help message
-camType         GUPPY           GUPPY            camera type to use <OPENCV | GUPPY | SONY>

(+): mandatory argument
(*): argument was set on commandline
```

Figure 2.2: Output of tutorial 1, when calling with the `help` option

2.1.1 FireWire camera input

The abstract class `opentl::input::FirewireCamera` from the `opentl::input` namespace provides this platform-independency, so that we can write the same

application code for both OS. At compile time, it is automatically decided which inherited class (`LinuxDC1394Camera` or `WindowsCMU1394Camera`) to instantiate, and link to the code.

In particular, under Windows this class uses the *CMU library* drivers, whereas on Linux the standard `libdc1394` is used.

The first instruction

```
opentl::input::FirewireCamera::getNumberOfFirewireCameras();
```

retrieves the number of currently connected FireWire cameras of our machine. Afterwards, we select one device (for instance, number 0) and create the camera class

```
input::FirewireCamera::createFirewireCamera(0);
```

We notice that both instructions so far are static code; after creating the camera, we can use the internal class methods. The next call

```
camera->open();
```

initializes the device, and

```
camera->captureStart();
```

starts the acquisition engine. We suppose so far to use a camera with `FORMAT 7` pixel coding, which requires a *de-bayering* mechanism, in order to pack input pixels into the standard RGB image format. This is the default choice, but of course there are several other possibilities; however, in this first tutorial we only cover the basic functionalities, and refer the user to the `FireWireCamera.h` header file for more details.

If we desire to set an automatic color adjustment we can use

```
camera->enableWhiteBalanceAuto(true);
```

as well as automatic gain adjustment (to the environment light)

```
camera->enableGainAuto(true);
```

Next, we declare an image which will contain the camera input frame. In *OpenTL*, images are handled through the class `Image` of the `core::cvdata` namespace, which is built on top of the OpenCV `IplImage` data structure, and contains the necessary facilities for allocation, accessing and copying.

```
Image(  int initWidth, int initHeight,
        ColorChannels initChannels = RGB,
        BitsPerChannel bpc = BPC8U);
```

The first two arguments here specify image resolution, the third one is the number of channels, and the last the number of bits per channel, that completely reflects all OpenCV image types: for example, `RGB` are 3-channel color images,

and `BPC8U` specifies 8 bits, unsigned (corresponding to unsigned char values). For a complete list of available types, please refer to the `Image.h` header file.

The camera input class already contains an image with the corresponding resolution and pixel format, whose pointer we can obtain with the function `FirewireCamera::getImage()`.

Moreover, in order to record the input to a video, we declare a video writer (OpenCV class `CvVideoWriter`)

```
cvCreateVideoWriter(  "video.avi",
                      codec, 30.0,
                      cvSize(camera->getImageWidth(),
                      camera->getImageHeight()), 1
```

where `codec` is a suitable output video format; in our example we use the XViD codec, given by the OpenCV macro `CV_FOURCC('X','V','I','D')`.

Now we start the main on-line loop, that performs the following operations:

1. Acquire a new frame

```
camera->captureNext();
```

2. Get the camera image (as pointer)

```
srcImage = camera->getImage();
```

3. Display it on screen

```
cvShowImage("image",srcImage->getIplImage());
```

4. Add it to the output video

```
cvWriteFrame(video, srcImage->getIplImage());
```

The loop can be exited at any time by pressing the `esc` key.

2.1.2 USB camera input

In a second version, we consider a simple Webcam device, that can be handled by the OpenCV-HighGUI functionalities, through the `CvCapture` data structure.

For this purpose, the class `opentl::input::OpenCVCamera` contains most of the same methods and data of the `ImageSensor` abstraction; we do not repeat here the previous code description, but only point out the main differences.

- the constructor for `OpenCVCamera` is direct (unlike the virtual constructor `opentl::input::OpenCVCamera(0)`)
- we do not yet provide automatic gain, and color balance adjustments, for USB cameras

- in the `OpenCVCamera` class, usually the image is flipped upside down, so we currently need to flip it back manually, by using

```
cvFlip(srcImage->getIplImage())
```

- at the end, we call `delete camera;`, which in turn calls the virtual destructor `deleteFirewireCamera()`.

For the rest, no other modifications are done.

2.2 Step 2: Pose representation and screen projection

Now we will learn how to specify and use the degrees of freedom of the object model.

For this purpose, we start with a simple task: to project a few points from object space to camera screen, under a given pose representation (Fig. 2.3).

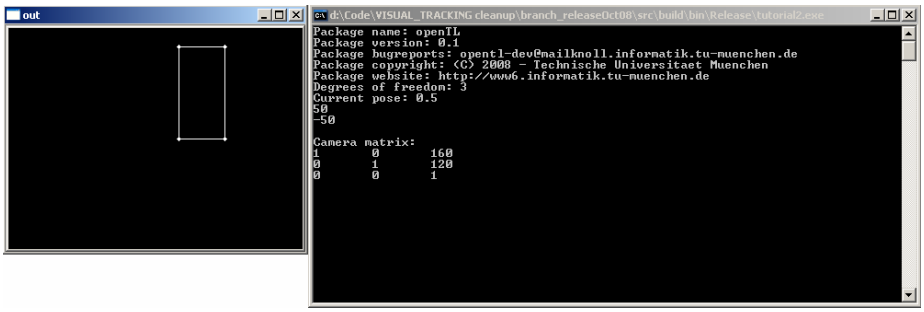


Figure 2.3: Tutorial 2: Draw a simple shape model at a given pose (2D translation and scale).

The first step consists of declaring the pose class: we choose a simple 2D pose, consisting of: uniform scale s, and translations t_x and t_y); the corresponding transformation matrix is given by

$$T = \begin{bmatrix} s & 0 & 0 & t_x \\ 0 & s & 0 & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

This is accomplished by declaring a variable of the class `Pose2d1ScaleTranslation`

```
opentl::core::cvdata::Pose2d1ScaleTranslation pose(
opentl::core::cvdata::Pose::COMPOSITIONAL);
```

Notice that this class belongs to the `opentl::core::cvdata` namespace, and in particular to the `POSE_2D` subset of types, which is the set of planar,

uncalibrated transformations. Furthermore, the *update mode* has to be specified in the constructor. Here one my choose between `Pose::COMPOSITIONAL` and `Pose::ADDITIVE`, which corresponds to an exponential representation for the former and an additive for the latter one.

Afterwards, we need to insert this pose representation into a *state*, which may include also other parameters such as velocity, acceleration, etc. (in this case, pose is the only state variable). For this purpose, we need the class `State`, which is part of the `opentl::core` namespace.

Moreover, in order to access states in a thread-safe way, each state is created through a shared pointer from the *Boost* library as

```
boost::shared_ptr<opentl::core::State> state;
```

The state must be initialized with the pose type and data, and we accomplish this by using the above defined pose template

```
state.reset(new opentl::core::State(&pose));
```

where the two non-specified flags (false, by deafult), specify the presence of velocity and acceleration variables in the state (here they are not present).

We can read the overall number of state parameters (which is 3) with

```
state->getPoseDegOfFreedom()
```

In order to set the current pose parameters, we declare a `vector` with 3 components, from the `math::` namespace. For example,

```
math::Vector pose_data(dof);
pose_data[0] = -50;
pose_data[1] = 50;
pose_data[2] = -50;
```

and we set the state with

```
state->setPoseData(&pose_data);
```

This instruction tells to set the *whole* state from a single vector containing the pose data, with the proper dimension (pose+velocity+acceleration dimensions). In our case, no velocity or acceleration components are present, therefore `pose_data` has the right dimension. Note, that the scale factors are represented in a logarithmic way in *OpenTL*, s.t. a scale parameter of -50 is mapped to the finally applied scale factor s by

$$s = 1.01^{pose\ parameter}$$

Next, we need to declare the camera model. This is done by using the `SensorModel` class from the `opentl::models` namespace

```
opentl::models::SensorModel sensor;
```

Although *OpenTL* currently handles only camera input, for sake of generality we denote this class as sensor model. A general, calibrated camera setup requires two informations for each camera:

- Internal parameters (acquisition model)
- External parameters (relative world location)

However, for our simple 2D transform, an uncalibrated model can be provided, where no external parameters are given (world coordinates coincides with camera coordinates), and the internal parameters are only the horizontal and vertical image resolution (no focal length is needed).

The latter is accomplished by the following

```
sensor.setK(focus, xres, yres);
```

where `xres,yres` are the image resolutions, and `focus` is set to 1. This command sets the internal matrix K, in this case given by

$$K = \begin{bmatrix} 1 & 0 & 0 & r_x/2 \\ 0 & 1 & 0 & r_y/2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

The base frame origin of the coordinate system then resides at the center of the image, with y pointing down and x pointing to the right.

The overall mapping, from object to screen homogeneous coordinates, is always given by

$$\begin{aligned} \bar{y} &= K \cdot T \cdot \bar{x} \\ y &= \begin{bmatrix} \frac{\bar{y}_1}{\bar{y}_3} & \frac{\bar{y}_2}{\bar{y}_3} \end{bmatrix}^T \end{aligned} \tag{2.3}$$

In our framework, multiple cameras can be present, therefore we need to put them again in a `std::vector` of pointers (no shared pointers are required for cameras, since they are read-only input classes).

```
std::vector<models::SensorModel *> sensVector;
```

Now we can declare the main class of this tutorial application: the `Warp`. This class is part of the `modelprojection` namespace, and it has always a unique instance throughout the application. Its semantics consists of managing the relationship (mapping) between all object and camera spaces. In other words, it acts as a "dispatcher", that projects points from each target to each camera, according to the current state, and compute screen Jacobians as well (if required).

```
opentl::modelprojection::Warp warp(sensVector);
```

The Warp class keeps a pointer to the sensors inside, and gets each time the pose parameters for a given object, both when updating projection matrices, and when computing individual point projections and derivatives.

Whenever the state vector changes (or at the beginning), the `warp` class requires updating the internal camera matrices (and Jacobians, if required), before projecting individual points on the screen. This is obtained by calling the `warpUpdate` function

```
warp.warpUpdate(state.get());
```

with the current state vector.

Now we take a few points in homogeneous object coordinates (`math::Vector4`), corresponding to the four corners of a square

```
math::Vector4 objP1(-50, -100, 0, 1);
math::Vector4 objP2(50, -100, 0, 1);
math::Vector4 objP3(50, 100, 0, 1);
math::Vector4 objP4(-50, 100, 0, 1);
```

where we set the z coordinate to 0, since we have a planar mapping. This is an arbitrary choice for 2D poses, since any non-zero value is just ignored. The corresponding screen projections, for the warp function, are of type `math::Vector2` (inhomogeneous screen coordinates).

```
math::Vector2 screenP1, screenP2, screenP3, screenP4;
```

In order to do the screen projection, we call the main function `warpPoint`:

```
void warpPoint(  cvdata::Pose& objPose,
                 const math::Vector4* objPoint,
                 math::Vector2* screenPoint,
                 std::vector<math::Vector2>* jacobian = NULL,
                 int cameraIdx = 0, int linkIdx = 0);
```

In this function, the first argument is a generic `Pose` class (passed by reference), which in our case has the type `core::cvdata::Pose2dRotoTranslation`. The second argument is a pointer to the object point in homogeneous coordinates (`math::Vector4`); output screen coordinates (`math::Vector2`) are also passed by pointer, and the screen Jacobian is a $(2 \times dof)$ matrix

$$J \equiv \frac{\partial y}{\partial p}$$

which is an optional argument for this function. Finally, `cameraIdx` specifies the viewing camera (for a multi-camera setup), and `linkIdx` the skeleton part which this point belongs to (for articulated, multi-body structures). By default they are set to 0.

We call this function 4 times, for each object point

```
warp.warpPoint(*state->getPose(), &objP1, &screenP1);
warp.warpPoint(*state->getPose(), &objP2, &screenP2);
warp.warpPoint(*state->getPose(), &objP3, &screenP3);
warp.warpPoint(*state->getPose(), &objP4, &screenP4);
```

In order to plot results, we declare a color image with the same resolution of the (virtual) sensor device

```
opentl::core::cvdata::Image img(xres,yres);
```

Afterwards, we cast the screen points to a `CvPoint` structure

```
CvPoint pt1 = cvPoint(cvRound(screenP1[0]), cvRound(screenP1[1]));
```

and similarly for the others. This allows to use the OpenCV drawing functions `cvCircle` and `cvLine`, that we use to draw the shape. For example, we have

```
cvLine(img.getIplImage(), pt1, pt2, CV_RGB(255,255,255), 1);
```

In this call, we use the method `getIplImage()` in order to retrieve the internal pointer to the OpenCV data structure

```
IplImage* getIplImage();
```

Similarly, also matrices and vectors can access the internal `CvMat` pointer, by calling `Matrix::getCvMat()`, `Vector::getCvMat()`.

At the end, we should get an output that should be comparable to the left image in Figure 2.3.

2.3 Step 3: Shape and appearance model

In the previous step we learned how to represent an object pose through the `Pose` class, and to project individual points from object to camera space, through the `Warp` class.

Now we consider the full object shape, represented by a set of connected primitives (polygons, edges and vertices) of a CAD model. This provides a wire-frame representation of the object (OpenGL-style), that however contains no information about the surface appearance, which is also needed by the color-based application that we are building.

Therefore, in this tutorial step we will take from the on-line camera input a simple but complete shape and appearance model, consisting of a rectangular shape and a reference image; this model will be obtained by simple mouse operations(Fig. 2.4).

The main new class introduced here is called `ShapeAppearance`, which is part of the `opentl::core::cvdata` namespace:

```
opentl::core::cvdata::ShapeAppearance shapeApp;
```

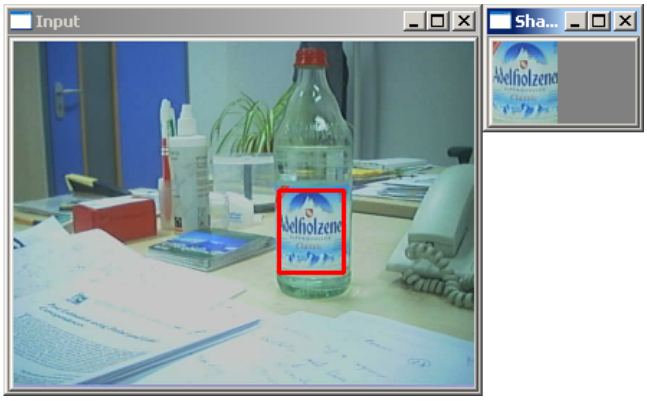


Figure 2.4: Tutorial 3: From the on-line camera input (left) take a reference shape and appearance model (right).

We begin as always by declaring the main OpenCV window, on which we set up a mouse callback function through the HighGUI functionalities

```
cvNamedWindow( "Input", 1 );
cvSetMouseCallback( "Input", mouseCallback, 0 );
```

where `mouseCallback` is a function that takes care of mouse events happening over the window, in particular left button press and release, and dragging pointer over the window. As a result, this callback provides the rectangular region of the camera stream, from which we want to get the shape and appearance model.

In order to interface the callback with the main code, we need to declare some global variables: the input image (`cvdata::Image * image`), the output selection (`CvRect selection`) and a few static variables for the callback itself like button press/release flags.

Afterwards, we set up the camera and run the input as in Step 1. In order to set up the appearance model, we declare an additional color image

```
core::cvdata::Image * appearance = NULL;
```

that will be allocated at the right moment, with the selected rectangle size. In the main loop, the input image is updated with the camera input, while the mouse callback handles the user interaction with the display window, until a rectangle has been dragged onto the window.

This events sets the flag `ready_shape` to `true`, so that the model can be taken. At this point, we declare the shape model as a rectangle, with the same size of the selected region, but centered about the origin $(0, 0, 0, 1)$ in object coordinates. This is done by the `initRectangle` function.

```
shapeApp.initRectangle(selection.width, selection.height);
```

Afterwards, we allocate the appearance image with the same size, and copy the reference image under the selected area, by using the OpenCV `cvSetImageROI`

and `cvCopy` functions. This image must now be inserted into the `shapeApp` class. Since an object appearance can be specified by multiple reference images (that can also be used as texture maps), we put the image in a `std::vector`

```
std::vector<opentl::core::cvdata::Image *> imageList(1, appearance);
```

Moreover, each image is identified and retrieved through its name, in order to avoid confusion arising from numerical indexes. We give it here the name `MyAppearance`

```
sprintf(strbuf, "MyAppearance");
std::vector<char *> namesVec;
namesVec.push_back(strbuf);
```

and finally insert it into the model

```
shapeApp.initAppearance(&imageList, &namesVec);
```

Finally, we can verify the internal appearance model by retrieving this image by name

```
shapeApp.getMaterialByName("MyAppearance")
```

Notice that, for sake of generality, we call the internal appearance images *material*, since they may also represent textures (if a texure map is given).

2.4 Step 4: Set up the color-based likelihood

So far, we obtained a model of the object to be tracked, and we learned how the mapping between object and camera space works.

The next step is then how to compare the model with the current image, under a given pose hypothesis: this is the *likelihood* function

$$P(z|s)$$

where s is the object state, and z are the *measurement* data, associated to the object. This function expresses the probability of the observed data, if the "true" state were s; in other words, it models our sensory processing system in a probabilistic form, where the uncertainty may arise from many sources, namely signal noise, modeling errors of any type, etc.

In our case, the camera provides raw sensory data (color images), containing a large amount of information that can be pre-processed in many different ways.

Therefore, in computer vision many different likelihood models have been developed, that can be roughly classified according to:

1. The visual modality (edges, colors, motion, ...)
2. The level of abstraction of the resulting data, namely: pixel-, feature- or state-space data

In order to understand this tutorial step, we briefly recall then our `Modality` abstraction from the `opentl::modalities` namespace, that unifies all visual modalities at any processing level, through the following abstract methods:

1. `preProcess()`: Model- and state-independent image processing, producing data suitable for the subsequent matching
2. `sampleModelFeatures()`: Collect visible features from the shape and appearance model, at predicted pose s^- (also called *off-line* features)
3. `matchPixel/Feature/ObjectLevel()`: Match model and image data, and produce output at one of the three levels: pixel maps, associated features, or a Maximum-Likelihood state estimate
4. `updateModelFeatures()`: From image data, after updating the state s, update also the model features (or better, the *on-line* version of these features)

Here, we consider measurements arising from a color-based modality (color histograms in Hue-Saturation space), which is processed at feature-level; therefore, our measurement data is a color histogram, to be compared to the corresponding model histogram via a suitable distance metric E (for example, the Bhattacharyya distance).

After the metric has been defined, in order to state the probabilistic model $P(z|s)$ we also need to know the *covariance* of the residual noise, R, so that finally we have a simple, Gaussian model

$$P(z|s) \propto \exp\left(-\frac{E^2}{2R^2}\right)$$

For more details about this modality and the likelihood function, please refer to the paper [**?**]. In *OpenTL*, we have a dedicated class for this kind of processing, called `ColourHist2D`, which inherits from the `Modality` abstraction:

1. `preProcess()`: Convert the camera image from RGB to HSV color space
2. `sampleModelFeatures()`: Collect the HSV histogram from the reference appearance model (off-line)
3. `matchFeatureLevel()`: Collect the underlying image histogram under the predicted pose, and compute the Bhattacharyya coefficient
4. `updateModelFeatures()`: From image data, after updating the state s, update also an on-line reference histogram

In Fig. 2.5, we see an example of the color-based likelihood evaluation.

The first part of the code for this tutorial step is the same as step 3 (set up camera input, mouse callback, and obtain the shape and appearance model), that we do not repeat here.

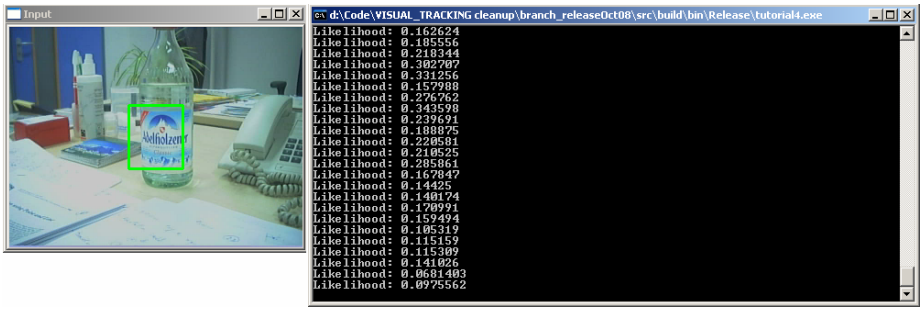


Figure 2.5: Tutorial 4: Color-based likelihood evaluation between the reference model (see Fig. 2.4) and the current image, under a fixed rectangular area (green rectangle). The likelihood is computed in real-time (left), while moving the camera around the object.

Afterwards, we need to encapsulate all model information (shape, appearance and degrees of freedom) in a unique container class, called `ObjModel`

```
ObjModel obj(&shapeApp,
core::cvdata::CvData::POSE_2D_1SCALE_TRANSLATION);
```

that is part of the `opentl::core::models` namespace. In the constructor, we pass the shape and appearance information, plus the degrees of freedom, which this time are directly given as data type `POSE_2D_1SCALE_TRANSLATION`, instead of using a template instance of the corresponding `Pose` class.

This constructor has also a third argument, namely the motion model (we will see it in the next tutorial step) of the class `models::Motion::MotionType` that by default is set to `BROWNIAN`: this is a simple Brownian model, where only the pose is part of the state, while velocity is a white Gaussian noise (process noise).

As already mentioned, the state of an object concerns its temporal informations like the pose, and possible velocities and accelerations. To provide a convenient way of handling informations about tracked objects, the information about them is stored within a `Target`-class, taking care of concrete single instances of tracked objects, which thus is related to the respective object model. The `Target`-class is part of the `opentl::models` namespace, handles all informations about a single object (like its states, etc.) and due to thread-safety issues is declared as a *Boost* shared pointer. Since possibly multiple targets are tracked simulatneously, we need to declare a vector of targets by:

```
std::vector<boost::shared_ptr<models::Target> > targetVec;
```

Since the targets depend on the object model, instances of a concrete target type are directly created from the `ObjModel` class:

```
obj.addNewTargets(1, &targetVec);
```

which inserts 1 new target in the target vector, of the corresponding type of the `obj` class.

As we learned from Step 2, in order to map from screen to camera space we also need a `Warp` class, and a `SensorModel`; this is done in the usual way. Here we also introduce a new class: `SensorData`, from the `opentl::models` namespace. This is an abstract container for the sensor raw data (in our case an `Image`), that is used by the `Modality` classes for processing.

```
opentl::models::SensorData sensData(srcImage);
```

which, at construction time, needs to know which type of data is produced by the sensor (in this case, and `Image` template). Moreover, we also link the particular image pointer to the class, through the method `SensorData::setData`

```
sensData.setData(srcImage,false);
```

so that the camera image stored in `srcImage` can be directly used by this class (without need for copying data). This kind of abstraction is meant for future developments of *OpenTL*, if other sensor types will be introduced (for example laser ranges, infrared sensors etc.).

Now we can declare the color-based modality class `ColourHist2D`. For this purpose, we first need to se some parameters; since each modality may have many parameters for all of its processing steps, of very different type and semantics, we introduce the abstract data container class `opentl::core::util::ParameterContainer`, as a general base class for the modality abstraction, which is differently defined for each modality. The `ParameterContainer` in general distinguishes between *Online* and *Offline* parameters, where the former are allowed to be changed during processing, and the change of the latter forces the user to do a reinitialization of the modality.

For color histograms, we specify the following relevant parameters (for feature-level matching):

- `ColourFormat preProcess_DestColorFormat`: output color space for the pre-processing step (HSV in our case)
- `int sampleModelFeatures_Bins1,2,3`: number of bins for each dimension of the histogram (H,S,V respectively)
- `double matchFeatVariance`: this is the variance R, for the feature-level likelihood above defined
- `bool matchFeatWithMdetect`: Flag, specifying whether to use the off-line histogram for matching
- `bool matchFeatWithMtrack`: whether to use the on-line histogram for matching
- `double matchFeatMtrackWeight`: weight for the on-line model histogram

- `double matchFeatMdetectWeight`: weight for the off-line model histogram (similarly for Mtrack)

In order to set up the Modality, we first instantiate the class by

```
ColourHist2D * pColHisto =
new modalities::ColourHist2D(warp, camIdx)
```

where `camIdx` is the camera number (for multi-camera setups, we have a modality instance for each camera). Afterwards, the parameters may be set to the desired values by:

```
pColHisto->setParameter<paramT>
(opentl::modalities::ColourHist2D::param_name, value);
```

with *paramT* the type of the parameter and *param_name* the name of the parameter as specified in the list above. Note that if the user does not set a value for a parameter at this stage, the modality will be initialized with predefined default parameters. Finnally to make the parameters valid and conclude the initialization, the user calls

```
pColHisto->init();
```

The next class we need is the likelihood:

```
modalities::Likelihood likelihood;
```

This class handles all of the visual modalities and data fusion classes, in order to produce the target-related measurements for the correction step.

In particular, this class has two main methods that are used by different kind of Bayesian tackers

- `implicitModel`: $P(z|s)$, with P the likelihood function
- `explicitModel`: $z = h(s) + v$, with v the explicit measurement noise, and h the expected measurement

For a particle filter, we require in particular the `implicitModel`, as will be described in the next tutorial step; instead, for Extended Kalman Filters the `explicitModel` provides the measurement residuals $E = z - h(s)$ (also called *innovations*) and the covariance R of v (supposed to be a Gaussian noise).

In order to use these methods, we need to connect the likelihood class to the visual modalities and data fusion, and *OpenTL* can do it in a flexible way, by building a *measurement tree*. For the present tutorial we use only one modality, that we pass as a template at construction time

```
likelihood.addChild(pColHisto, modalities::Modality::FEATURE_LEVEL);
```

which internally clones and stores the Modality class template, through its `Modality::clone()` method. The specification `Modality::FEATURE_LEVEL` tells

the likelihood to internally call a feature-level matching: `matchFeatureLevel` The output likelihoods from `implicitModel` are stored in a vector, and multiple states (arising from multiple targets and/or particle hypotheses) can be simultaneously evaluated. In a multi-thread version of this class, this will provide a high degree of parallelism with a tremendous performance impact; however, this is not the only place in *OpenTL* where parallel computing can take place, since also multiple modalities and cameras for the same target can be evaluated in a multi-thread fashion.

Some modalities require an off-line sampling of visual features from the shape and apperance model. This is the case for `ColorHist2D`, where the method `sampleModelFeatures` collects a reference histogram from the appearance model; in particular, we need to call this function from the `likelihood` class, which internally hosts the clone of our Modality, and calls its method

```
likelihood.sampleModelFeatures(targetVec);
```

For threading issues, the number of particles is divided into partitions. In the present case threading is out of scope, and therefore we set the number of active partitions in the for the target to 1 and set the active state vector to the first one by:

```
targetVec[0]->resizeActivePartitions(1);
targetVec[0]->setActiveState(0, targetVec[0]->getState(0));
```

Before starting the camera loop again, we reset the object pose to the center of the image

```
poseVec[0] = 1;
poseVec[1] = 0;
poseVec[2] = 0;
targetVec[0]->getState(0).get()->setPoseData(&poseVec);
```

and update the internal transformation matrices with the `warpUpdate` method

```
warp->warpUpdate(targetVec[0]->getState(0).get());
```

Now we can re-start the camera loop, in order to compare the incoming images with the reference model, under the given pose, and get the likelihood value. The first step for this purpose, as we explained at the beginning, is image pre-processing

```
pColHisto->preProcess(&sensData);
```

where the sensor data are used (which contain a pointer to the camera image). Then, the likelihood is obtained by simply calling

```
likelihood.implicitModel(&targetVec, 0);
```

where 0 is the number of the partition. In order to retrieve the value of the active sample we call

```
targetVec[0]->getActiveWeight(0)
```

, display the value on the output console and observe that the maximum value should be obtained when the interior of the projected rectangle (green box) is the same as the appearance image.

Finally, in order to show the model at the given pose, we introduce another static function, which is part of the `output` namespace:

```
output::Output::drawPose(  cvdata::Image* img,
                           cvdata::CvData* visFeat,
                           object::Warp* warp,
                           cvdata::Pose* pose,
                           int thickness = 1,
                           CvScalar color = cvScalarAll(255.0),
                           bool drawEdgeNormals=false,
                           bool drawEdges=true
                           int camIdx = 0);
```

The first argument is the output image, the second specifies the kind of data we wish to visualize (normally the `object::ShapeAppearance`); then we need the `Warp` class and the specific `Pose` data, and finally some data related to the line style, color, whether to draw screen edges and normals (the latter are useful for debugging purposes), and the camera view (for multi-camera contexts).

2.5 Step 5: Set up the particle filter and track the object

In this part of this tutorial "closes the loop": based on the visual measurement so far described, we want to update our knowledge about the current object state, in a Bayesian fashion (prediction-correction).

For this purpose, *OpenTL* offers several filters that can be used in different situations, such as: single or multiple targets, linear/nonlinear measurement or dynamics, Gaussian/non-Gaussian noise distributions, and different measurement levels (pixel/feature/object). These filters are provided inside the `tracker::` namespace, and all are derived from the common abstraction `Tracker`, which has 3 virtual methods

1. `init()`: Initialize the filter with a prior distribution (usually Gaussian, or uniform)

2. `predict()`: From the previous state posterior, generate the prior distribution using the dynamical model (`models::Motion`)

3. `correct()`: From the current prior, perform a measurement (calling on of the `likelihood` methods `explicitModel`, `implicitModel`) and update the new posterior. Afterwards obtain a meaningful, unique output state for output purposes (e.g. the Kalman Filter mean, or the particle average)

In particular, for the present color-based task we have a highly nonlinear measurement model (histogram distance), for which no Jacobian matrices are available. This case is suitable for a SIR (sample-importance-resampling) particle filter, that we can instantiate right after the `likelihood` declaration, with

```
SIRParticle particle(&likelihood, warp);
```

The constructor accepts the `likelihood` class, of which internally instantiates a clone (that means, the likelihood class that we declared so far now is used as a *template* for construction). Therefore, if we need to access the `Modality` methods, we need to do it indirectly, from the `Tracker` class: in particular, in order to sample off-line the reference histogram from the appearance image, we call

```
particle.sampleModelFeatures(targetVec);
```

which calls the internal `likelihood` method, which in turn calls the internal `ColourHist2D` method. To initialize the particle filter, the number of particle hypotheses have to be specified, which is done in a per target based fashion, by first specifying a vector containing the number of particles for each target by

```
std::vector<int> nOfParticles(1,200)
```

where in our example, with 3 degrees of freedom and a single target, we set it to 200. Furthermore, an initial state for each target has to be given which in our case is the predefined position of the selected rectangle. The final initialization is done by

```
particle.init(targetVec, &initStates, NULL, &nOfParticles);
```

with `initStates` a vector of *Boost* shared pointers of states:

```
std::vector<boost::shared_ptr<core::State> > initStates;
```

We can see a snapshot of the previously defined object, tracked in real-time with the color-based particle filter, in Fig. 2.6.

Now, in order to track the object, we call in the main camera loop the above mentioned methods

```
pColHisto->preProcess(&sensData);
particle.predict(targetVec);
particle.correct(targetVec);
```

where the `preProcess` method must be called before any other processing step.

Once again, we notice that `preProcess` is the only method that should be called from the original Modality template `pColHisto`, and not from the `Tracker` or `Likelihood` classes.

This is meant for efficiency reasons, since the same modality can be used in order to instantiate more tracking pipelines, that share a common image preprocessing step; the output of this step is, in fact, also shared by all tracking pipelines that have been instantiated from the same template.

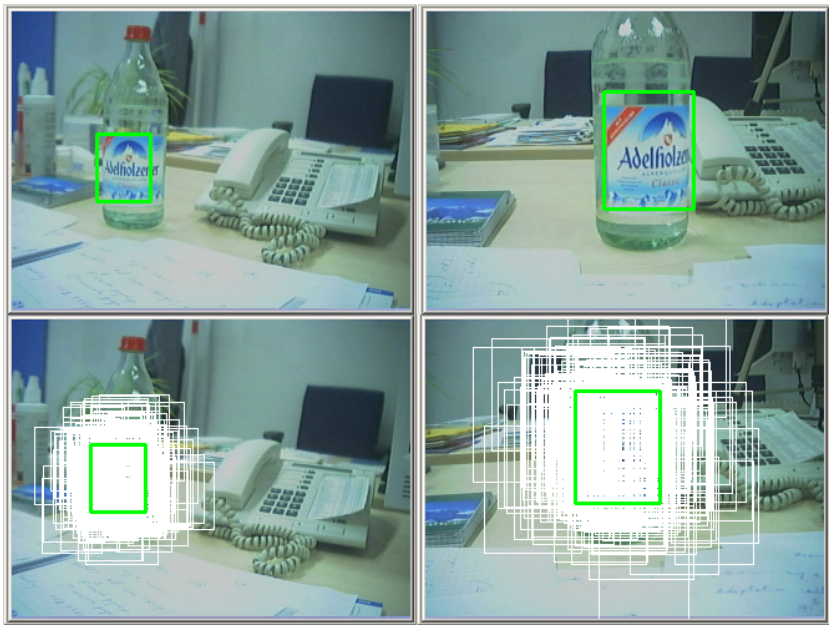


Figure 2.6: Tutorial 5: Color-based particle filter. Top row: the object with shape and appearance model defined in the previous steps is tracked on-line from different viewpoints. Bottom row: the same frames with superimposed particles, representing the estimated posterior distribution (the green rectangle represents the *weighted average* of the particle set).

Afterwards, we can display the output of our tracker by taking the internal output state of the respective target

```
targetVec[0]->getOutputState()
```

that corresponds to the weighted average of the particle states for target 0 (in a multi-target context, we may need this index as well). The function that provides visualization of our model is `output::Output::drawPose`, as described in Step 2.4.

For debugging purposes, we can also draw the entire particle set, by calling the same function on each particle p state `particle.getParticles(p)`.

2.6 Step 6: Tracking multiple targets

Part 6 of this *OpenTL* tutorial extends the tracker of step 5 to a multiple target version. *OpenTL* supports the tracking of multiple target due to the use of vectors for all kind of objects used for tracking including:

- targets

- shapes
- appearances (reference images)
- object models

One can make - as described in step 3 - a rectangular selection around the object to track. This selection can be done in this step more than once. Multiple objects in the scene can be selected and tracked with only one particle filter.

For each selection of the user a new reference image (appearance), a new shape (with the dimension of the selection and the selected reference image), a new object model (with motion model, and pose representation), and a new target is created. Everything else stays the same.

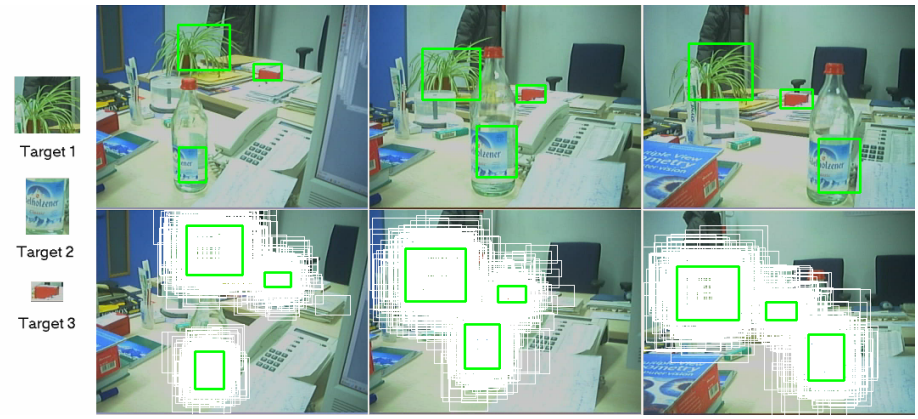


Figure 2.7: Tutorial 6: Tracking multiple targets using color-statistics and particle filtering. The left row shows three targets that have been selected with the mouse. The upper colomn shows the result of the tracking in different frames of a video sequence. The lower column shows the respective hypothesis of the particle filter.

2.7 Step 7: Measurement fusion with another modality

The next step consists of adding a second modality to the processing tree in order to provide a fusion of different modalities. We'll use the `ContourPointsGPU` modality, which is an intensity edge based modality, implementing the abstraction in the following way:

1. `preProcess`: Extract the edges of the image by applying an appropriate method. At the moment one may choose between the *Sobel*-Filter, *Canny*-Edge map, or a combined approach of both methods.

2. `sampleModelFeatures`: project the model at the given pose into the image, find the visible edges from the camera view and uniformely sample those edges with points for the matching step.

3. `matchFeatureLevel`: match the projected visible edges against the edges found in the `preProcess`-step, by doing a search along the normels of the sample points retrieved from the `sampleModelFeatures`-step. Here we restrict to searching the closest edge along the normal only, yielding a single hypothesis.

Furthermore, like all modalities, it is part of the `opentl::modalities` namespace. It is also notable, that the ContourPointsGPU modality, is a *feature-level* modality and, as can be seen from the name, some computations are done on the GPU. At the moment, those are the visibility testing and sample point generation. In order to add this modality to the processing tree, we first instantiate it by

```
ContourPointsGPU * pContourPoints =
new ContourPointsGPU(objectModels, warp, &scene, camIdx);
```

where `scene` is an object of the `modelprojection::GLScene` class. This class is responsible for handling the virtual OpenGL representation of the real scene, which is needed for GPU-programming purposes as well as more efficient visualizations. The `GLScene` class is instantiated by

```
GLScene scene(objectModels, &sceneParams);
```

with `sceneParams` a struct of parameters for the scene. We here only mention the most important two parameters for the moment, which are the dimensions of the scene:

`construct_initWindowSizeX/Y`

Afterwards, like for the *ColourHistogram* modality, the modality specific parameters are defined. We set the following parameters for the `ContourPointsGPU` modality:

- `bool matchFeatLevel_useFixedCov`: use a fixed value for the covariance
- `double matchFeatLevel_fixedCov`: the fixed value for the covariance
- `FeatDetectFilterType preProcess_featureFilterType`: `CANNY`, `SOBEL` or `CANNY_AND_SOBEL`
- `double matchF_angleThreshold`: maximum angle between matched edge and projected edge (this is only possible for `CANNY` based filters)
- `double matchF_missingAssocRate`: Expected percentage of unassociated contour points
- `bool matchF_robustFlag`: Use robust object level algorithm or not

- `std::vector<unsigned int> matchF_edgeSearchlengthAlongNormalinPixel`: For each link, the number of pixels to search along the normals
- `bool matchF_nearestNeighbor`: use the nearest neighboring edge pixel as measurement
- `bool matchF_enableDebugOutput`: enable the generation of debug output images
- `bool generate_visCheck`: generate the visibility checking class automatically

Like for the `ColourHist2D` modality, to finalize the initialization of the modality we call

```
pContourPoints->init();
```

Now we can add the `ContourPointsGPU` to the likelihood computation by

```
likelihood.addChild(pContourPoints,
                    core::cvdata::FEATURE_LEVEL, 0.5);
```

note, that 0.5 reflects the contributing weight of the modality to the likelihood computation and therefore also has to be set to 0.5 for the `ColourHist2D` modality at this place. Internally, the joint likelihood is computed by first multiplying the single likelihoods of each modality by the assigned weights, and finally doing a product of the resultant values. Within *OpenTL* this is called a *dynamic fusion* at feature level. Moreover, for this step we only use the single target version of our tutorial application (up to step 5). Fig. 2.8 shows the output results of the application.

Figure 2.8: Tutorial 7: Tracking a single target using dynamic fusion of color-statistics and intensity-edges with particle filtering. The upper row shows the estimated output state in green on the left hand side and on the right hand side additionally all other used particles in white. The bottom row shows the selected appearance image on the left hand side, and the edge image with the matched normal points for one particle on the right hand side.

Chapter 3

The modular structure of *OpenTL*

In this Chapter, we briefly overview the *OpenTL* modular architecture, which has been in more detail presented at the CoTeSys Workshop [**?**], as well as in [**?**].

Fig. 3.1 provides a sketch of the layered module architecture of *OpenTL*, consisting of about 200 C++ classes distributed in 11 functional modules. Shown are the top-down dependencies across layers.

In the implementation, all classes have been organized according to the respective module *namespaces*. In particular, the topmost namespace is `opentl`, and all sub-spaces are hereafter described.

1. **Layer**: matrix computations

   The base layer contains facilities for algebra and matrix computation and manipulation, as well as general math utilities.

   - *Math module*: `opentl::math` namespace contains all matrix data types and computations, derived from a common abstraction Matrix. The Math module does not depend on any other module inside *OpenTL*, but it depends on external libraries (e.g. OpenCV).

2. **Layer**: base data structures and pose representations

   The second layer contains almost all of the data structures and pose representation classes, providing the core functionalities of *OpenTL*.

   - *Core module*: `opentl::core` contains the base data types and processing, including geometric transformations and Jacobians computation, image data, and visual feature structures. It is organized into three subspaces:
     - `opentl::core::cvdata`: Most of the *OpenTL* data structures are defined here. All of them inherit from the base abstraction

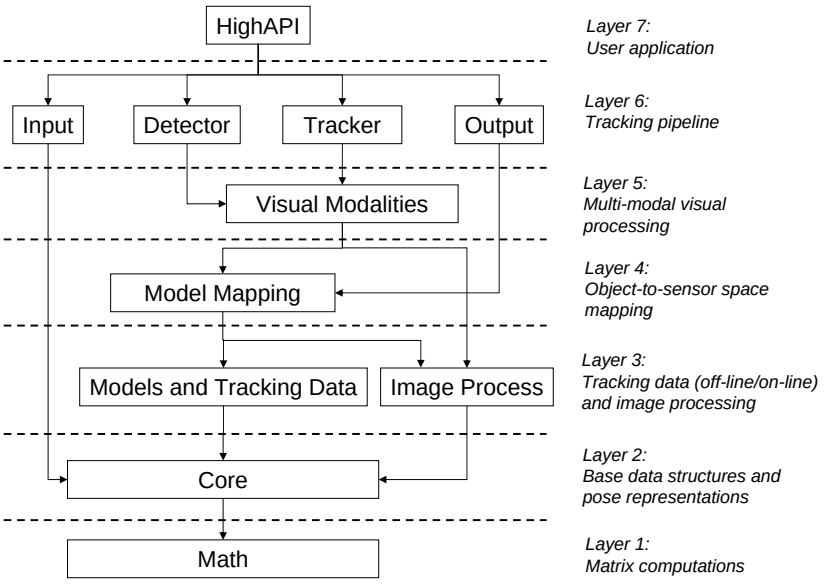


Figure 3.1: The *OpenTL* module architecture.

CvData, for the most different purposes: state-space representations (which in turn inherit from a general Pose abstraction); image data; shape and appearance model; visual features of the most variable nature and descriptor (inheriting from a common abstraction)

- `opentl::core::util`: File interface utilities, including an abstract data serialization class (input/output) and directory management
- `opentl::core::parser`: Specialized parsers for object models (COLLADA format) and command-line options for running executable application code

The `opentl::core` module depends on `opentl::math` for the base matrix and vector data.

3. **Layer**: tracking models (off-line/on-line) and image processing

   Layer 3 holds model-free image processing facilities, as well as object data on a higher level than the core module (i.e. target-related); here also GPU *shaders* for model-free image processing are provided.

   - *Models and tracking data*: In this module, the main namespace `models` contains classes for two main purposes:

  - Object models: shape, appearance, degrees of freedom, dynamics
  - Data processed within the tracking pipeline (Layer 6): sensor data, measurements, state hypotheses

  It depends on the `opentl::core` main data structures (for example shape and appearance data).

- *Model-free data processing*: The namespace `opentl::cvprocess` consists of model-free image and sensory data processing methods. Many of these methods have been implemented in an efficient way using the GPU shader language (GLSL), and the OpenGL contexts for rendering (*on-screen* and *off-screen*) are here provided.

  All of the functions implemented in this module do not make use of prior models, nor of any state hypothesis: examples include edge detection, color conversion, camera image de-bayering, invariant keypoints detection, etc. This module also depends on the core data structures.

4. **Layer**: object-to-sensor space mapping

   The fourth layer consists of classes mapping between object and sensor (in particular, image) spaces. Here are also included advanced GPU-based facilities, for example a sampler of visible model edges from any given viewpoint.

   - *Model projection module*: The `opentl::modelprojection` namespace provides classes for object-to/from-image mapping facilities: geometric features projection and derivatives, back-projection using depth maps, as well as GPU-based model rendering and features sampling.

     It depends on `opentl::models` (shape/appearance and state), and on `opentl::cvprocess` (base GPU shaders).

5. **Layer**: multi-modal visual processing

   Layer 5 contains the visual modalities for tracking: they perform all model-based processing operations required for data association and fusion (both over multiple modalities and targets), and deliver output measurements to the trackers/detectors of the upper layer.

   - *Visual modalities*: The `opentl::modalities` namespace includes a common abstraction for model- and state-based measurement processing, related to the most different visual modality (color, template, edges, etc.). In particular, the common functions are:
     (a) pre-processing
     (b) model features sampling
     (c) static/dynamic data association
     (d) measurement Likelihood, residual and Jacobian computations

(e) multi-modal and multi-camera data fusion

(f) model features update (after state update)

This module depends on `opentl::modelprojection`, because it needs to map features between object and image space in order to perform matching, sampling and update operations. It also depends on `opentl::cvprocess`, because each modality makes use of specific image processing.

6. **Layer**: tracking pipeline

   Here the main *tracking pipeline* is realized, through the main abstractions *tracker* and *detector*, as well as sensory input and output visualization.

   - *Input module*: Common abstraction for input sensor devices (e.g. FireWire, USB and Ethernet cameras), providing open/init/close and data acquisition methods. It depends on `opentl::core`, because of the `Image` data type.
   - *Detector module*: Common abstraction for object detection (model-based and model-free).

     Its purpose is to find new targets (=initial states) as well as remove lost tracks, without any prior information about number and location of the new targets, eventually using knowledge of the already existing targets.
   - *Tracker module*: Here several Bayesian trackers, including Gaussian-based filters (such as the Extended Kalman Filter) and Monte-Carlo filters (particle or MCMC filters) are implemented under the same abstraction - prediction, measurement, data association and fusion, correction.

     It depends on Modalities, because the measurement is performed by calling the Likelihood, which in turn calls the modality processing tree.
   - *Output module*: Classes for output visualization (e.g. model rendering), post-processing (e.g. track loss detection) and simple control tasks (e.g. pan-tilt unit controller through a serial port).

     It depends on *ModelProjection*, because of the OpenGL rendering and mapping facilities.

7. **Layer**: User application interface

   The HighAPI module in the topmost layer is finally meant to encapsulate the tracking pipeline in a more compact and user-friendly API, with an easier system and parameter specification.

   - *HighAPI module*: High-level interface to the tracking pipeline (currently work in progress).