Final Report Nvidia Code Sprint

Koen Buys

March 6, 2013

Abstract

This report documents the work done during the PCL Code sprint sponsored by Nvidia. The high-level purpose of this project is to implement and analyze various techniques for extracting humans body poses in dense 3D data at high speed with CUDA. The applications are numerous, from gaming to gesture interfacing. The code sprint component will be a focused implementation that produces fast and reusable code.

An adjusted open source implementation of the human pose detection algorithm by Shotton et al. was implemented using CUDA with depth images (and optionally RGB) as input. The human pose detection algorithm is based on the random decision forest approach and extended with bio-mechanical knowledge in order to work in more cluttered environments. In a second stage the training pipeline was made available to other people.

Chapter 1

System overview

The run-time goal of our system is to take in data from an RGB-D sensor (Microsoft Kinect or Asus Xtion Pro Live) and output the 3D locations of human body parts as predefined in a kinematic model.

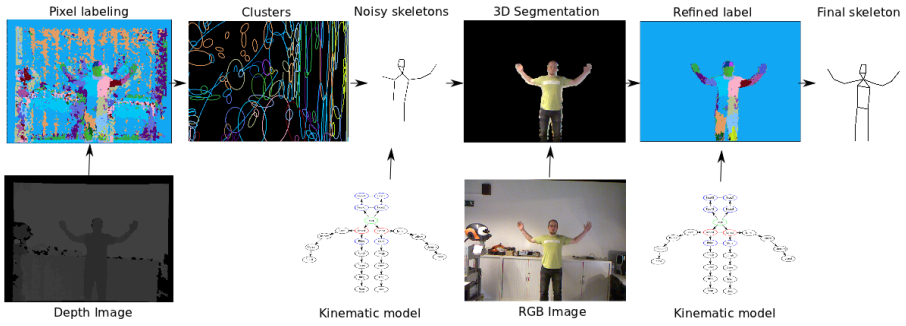


Figure 1.1: Run-time system flow diagram. A depth image is used to perform a noisy pixel labeling over the entire image, which is then clustered into human body parts from which a person is located and skeleton is extracted based on a kinematic model. This initial labeling and skeleton are used to bootstrap the on-line learning of a color and depth-based appearance model which then refines the pixel labeling and person segmentation, leading to more accurate and complete skeleton extraction. The output of the system includes a skeleton of the person, a part-based pixel labeling, and a segmentation of the person from the background.

At a high level, this process can be outlined as seen in Fig. 1.1. For a single pointcloud, every pixel is labeled as belonging to a single body part using a RDF approach and features like those in Shotton et al. Note that unlike Shotton et al., there is no background subtraction or fixed-pose initialization before the pixel labeling.

These initial pixel labelings are extremely noisy, so they are first smoothed and then clustered into more robust part estimates. From the statistics of the part estimates, a search for feasible kinematic trees is conducted, resulting in person detections and noisy skeletonization with missing body parts and poor localization.

To improve the skeleton estimates, we learn an appearance model on-line. The initial estimate is used as a seed for color and depth-based segmentation which retrieves missing body parts and better localizes existing parts. The kinematic trees are then re-estimated, resulting in a cleaner segmentation and localization. This process can be used for multiple people among clutter and occlusion.

In the sections below, we describe each component of the system in detail.

Chapter 2

Training Data

The initial pixel labeling, seen as the first step in Figure 1.1, is performed by a random decision forest based on depth features. In order to train these trees a large number of labeled training images is needed. In order to avoid the intensive labeling and recording of live people performing motions, we tackle the problem of generating training data by mapping real motion capture data onto a virtual model of a person.

In this section, we describe the data generation process. This process is modular and relies only on free open-source tools and data.

2.1 The human body model

We use the MakeHuman (MH) model, the model consists of a mesh structure with an underlying kinematic chain that can be parametrically morphed (by age, height, etc.) Although this model was not created for bio-mechanical applications, it is sufficiently accurate and flexible to model a large portion of the population. Previous work showed that matching this model onto a specific person can be done automatically, allowing for customization.

2.2 Pose information

To articulate the human model into realistic poses, we use the CMU MoCap database. This dataset contains a large range of human motions such as human interaction, sports, dancing, etc., captured using a Vicon system.

Two pre-processing steps are needed to make the data suitable. First, task-appropriate data must be selected. Since our target application was tracking people in indoor office or home environments, going about daily activities, we manually selected image sequences of every-day activities, and omitted extreme outlier actions such as break-dancing, reducing the dataset to 380,000 poses.

Second, the data must be sparsified. The original data was collected at 120Hz and contained many repeated poses. To accelerate training and avoid biasing the trees with unbalanced data, the data was greedily sparsified so that each pose was at least 5 degrees in joint angles from its nearest neighbor, leaving 80,000 poses.

Blender and OpenGL are used for skeleton mapping and rendering annotated depth images. Future users may adapt this modular process to train a part model that is better suited for their task, perhaps adding in objects or refining the body. Future work will add a sensor-dependent noise model into the depth maps.

Chapter 3

Pixel-wise body part labeling

The first step in the run-time algorithm is a pixel-wise classification of the entire image into body part labels, and is the core part implemented in CUDA. This section describes the features and decision forest, and the cluster-based method for learning the trees.

3.1 Features and Classifier

For a pixel x in image I, a feature is the difference in depth between two other randomly chosen points:

$$f_\Theta(I,x) = d_I(x + \frac{o_1}{d_I(x)}) - d_I(x + \frac{o_2}{d_I(x)}) \tag{3.1}$$

For each pixel, the feature vector $\hat{f}_{\hat{\Theta}}(I,x)$ contains 2000 features generated by randomly sampling offset pairs. For training, this feature vector is calculated for 2000 randomly chosen pixels on each body. The list of offset pairs, $\hat{\Theta}$, is common for all pixels. The feature vectors with their ground truth labels, $\hat{f}_{\hat{\Theta}}(I,x,c)$.

From these feature vectors, a randomized decision forest is learned. Decision forests are effective and can be implemented efficiently on a GPU. Each tree in the forest produces a pixel label estimate, $P_t(c|I,x)$, which are combined to give a forest estimate $P(c|I,x)$. Every pixel in the image is labeled, including the background.

3.2 Learning a decision forest

Random forest learning is efficiently performed in parallel by formulation as a MapReduce= problem, solved with a Hadoop Cluster. This formulation is portable to many available computing clusters, allowing the system to be re-trained as necessary. Training a single decision tree occurs as follows:

1. From the set of offset vectors $\Theta = (o_1, o_2)$ and feature thresholds τ, propose a random set of splitting candidates Φ.

2. Compute the information gain from splitting the data at each Φ_i.

3. Compute the splitting candidate Φ_i^* that maximizes information gain.

4. If the gain is high, split the data and recurse on the two subsets.

These steps are implemented as three MapReduce tasks:

- **Find the split** The Map task allows each feature to vote for a split point. The Reduce task accumulates the votes into histograms and computes the information gain. The best split point is written to the HDFS.

- **Split the data** The Map task uses the splitting threshold to map the feature vectors to odd or even values. The Reduce task then creates the two datasets, and a Cleanup task writes them to the HDFS.

- **In memory** It is much more efficient to work in memory on a node once the data is sufficiently divided to fit. The Map task gathers all of the distributed files from the HDFS into a single file, and the Reduce task splits the data in memory.

Generating the training data and learning a decision tree with 16 levels using Hadoop took 7 days for one body model in 80K poses on a 16 node cluster. The 16 cluster nodes were 8-core Xeon X5570 systems with 48GB RAM. The initial data generation took place on a single node with two GTX550ti GPUs.

By using the open infrastructure of MapReduce on a Hadoop platform, training can be conducted on non-homogeneous clusters and will automatically load balance, which allows our system to be retrained as necessary. However when they do feel the need to scale up because of the need for a larger training set, the implementation shifts smoothly to a cloud cluster on the publicly available Amazon Web Service cloud.

Chapter 4

Skeleton estimation and online learning

Up to this point, we have discussed the first step in Figure 1.1: pixel-wise labeling of the image from both a run-time and training perspective. Next, we explore the rest of the run-time pipeline, from extracting the skeleton and segmentation, to refining. Of note is the combination of RGB and depth for on-line learning. An example result of this skeleton extraction and refinement process is shown in Fig. 4.

4.1 Body Part Proposals

A breadth-first search is performed over pixels connected in the image (less than a given distance apart), with links created between them if the labels match and they are within a given distance in 3D. Only clusters with a sufficient number of pixels are retained. Performing this clustering in 3D is noticeably more robust than 2D.

Once all clusters with a minimal number of points are found for each of them the centroid, covariance are calculated and from this the eigenvectors and eigenvalues are calculated. All clusters are then evaluated a second time based on their first eigenvalue. If this value succeeds the feasible anthropometric

(a) (b) (c)

Figure 4.1: Refinement through online appearance model learning. (a) The original noisy labeling bootstraps online learning of a color and depth-based appearance model, leading to the much cleaner labeling in (b) (shown for the front-most person). (c) shows the image as a slightly-rotated point cloud, and the resulting skeleton of the front person.

values, the cluster is ignored as a good body part candidate. A lower limit however is not set, as this would not work under environments with clutter, causing occlusions or with partial self-occlusions.

4.2 Kinematic Tree Search

From the part proposals V we can generate proposals for all the people's locations and skeleton configurations S:

$$p(V|C, I) \rightarrow p(S|V) \tag{4.1}$$

Note that up to now, the people have not been segmented from the background.

Bringing kinematic constraints in the clusters happens in two phases. First local consistency between all links is evaluated. In the next step all global consistent kinematic trees are evaluated. This happens in a recursive pattern similar to a dynamic programming approach over all found local consistent links. First of all evaluated labeled clusters are stored based on size and label in a 2D matrix. Then according to the known kinematic chains all first step and second step edges of the kinematic graph are searched and the likelihood for each link is calculated and stored in the parent node. Including the second step link allows us to be redundant to some missing parts as some parts have a higher chance of not being correctly detected (like elbow and knee parts).

Once all local links are build we start evaluating the global consistency of each tree candidate. We do this by rooting the kinematic tree in both the lower head parts or taking the neck as root. This starting point proved to be the stablest one over the diversity of training poses and in the recorded data. All the global tree candidates are then evaluated based on number of parts connected in the tree, global error and normalized error over the number of parts. This gives the number of people in the scene combined with their pose as output and the labeled pixels. However due to interference from clutter, noise in the depth data or errors in the segmentation process not all pixels are grouped in the correct segment or are ignored due to mislabeling or an impossible local link. This causes the global kinematic chain to be incomplete, often without apparent reasons.

4.3 On-line learning

We now have detection and skeleton proposals for all people. However, these skeletons are inadequate in difficult scenarios such as moving cameras, high clutter and occlusions. In addition, the omission of background subtraction and the smaller training data set further reduce performance. The skeleton localizations that result are noisy and body parts are often missing, as in Fig. 1.1.

On the other hand, false positive part detections rarely occur, so this initial labeling can bootstrap on-line learning.

From the initial segmentation of a person, a model is learned of both their location in space and their appearance as given by the hue in the HSV space. Recently, Gulshan et al. showed that a person could be segmented using GrabCut given a loose bounding box prior and a number of seed points. However, their approach did not use depth and GrabCut took 6 seconds to run, which is

unacceptable for real-time tracking. Instead, we seed an Euclidean clustering based on the initial pixels' locations and hues. This can be made extremely efficient and effectively fills in missing pixels, as in Fig. 1.1. The segmentation features can be expanded as required by the application, for example by adding texture.

This new set of segmented pixels is then fed back into the part estimation and skeleton extraction process to produce a more robust estimate. By learning a model on-line, our algorithm exceeds the performance predicted by its training.

Chapter 5

Results

To test our algorithm, we collected 161 movies in indoor environments from a moving camera held approximately 1.4 - 1.8 meters off the ground. This scenario matches what would be seen in an indoor robotics scenario from a platform such as the PR2 robot.

There is no notable difference between the amount of female or male subjects found despite the fact that the RDF was trained on a single male subject. Due to the support of the kinematic chain and on-line learning, the algorithm can handle subjects and poses that where not explicitly trained for. This can be seen in Fig. 5.2 where a small young female subject is segmented out and annotated correctly in a cluttered environment. All the examples shown in this paper were annotated with the RDF trained for a single 25 year old slim male MakeHuman model.

(a) (b)

Figure 5.1: (a) Results of the NITE system versus our algorithm. The underlying image is a RGB-D frame shown in 3D and slightly rotated, leaving black shadows at depth discontinuities. The NITE results are shown as RGB axes for each joint. The results for our algorithm are shown as a white skeleton. Notice that our results show the head orientation, and provide more accurate shoulder and hip placement. (b) Final body part labeling and final segmentation by our algorithm.

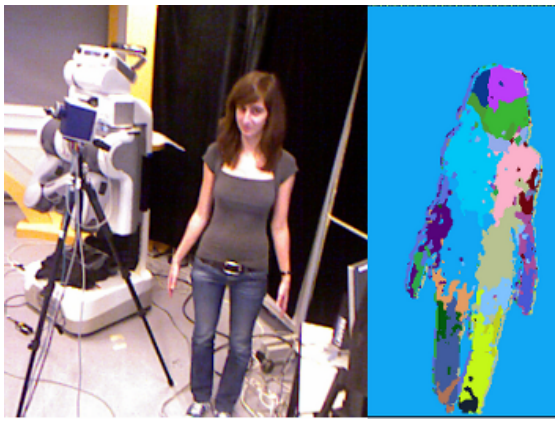
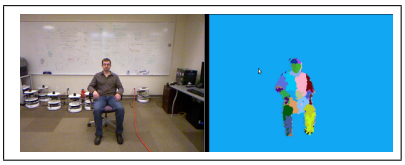
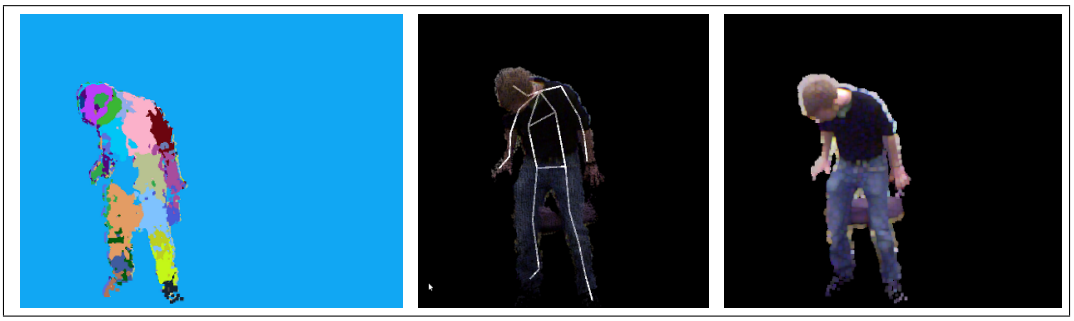


Figure 5.2: Body part labeling result of a female subject in a cluttered environment. The camera angle for this image is very different from the straight-on, chest-height angle generated in the training set.

(a)

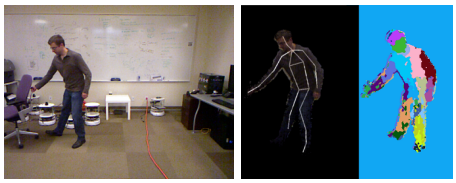


(b)



(c)

Figure 5.3: Results for poses that were not in the training set, but were still tracked adequately. The top row shows two sitting poses which are made more difficult by the presence of the chair. The bottom row shows someone about to sit in the chair.



(a) (b)

Figure 5.4: Result for a pose that was similar to a pose in the training dataset. (a) The original image. (b) The skeleton overlayed on the person segmented from the background, and the final body part labeling.

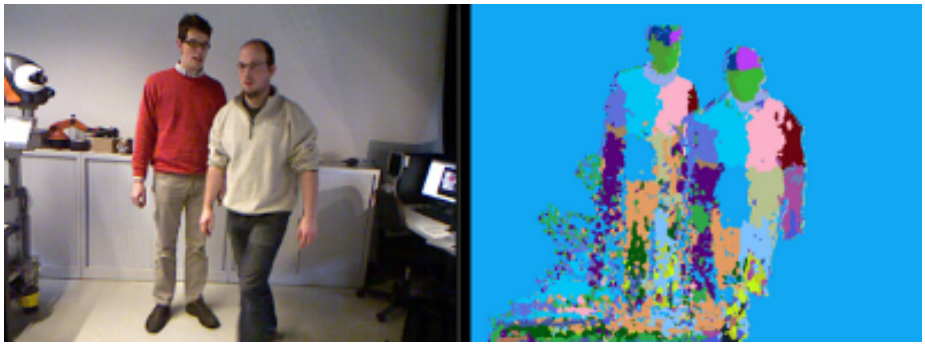


Figure 5.5: The system can successfully detect multiple people in close proximity. The matching pant color and close proximity of the rear person with the cabinet and floor has resulted some noise in the labeling.

Figure 5.6: Tracking a person in a highly cluttered environment with a high degree of occlusion.

Chapter 6

Tutorial

6.1 Configuring your PC to use your Nvidia GPU with PCL

In this tutorial we will learn how to check if your PC is suitable for use with the GPU methods provided within PCL. This tutorial has been tested on Ubuntu 11.04 and 12.04, let us know on the user mailing list if you have tested this on other distributions.

6.1.1 The explanation

In order to run the code you'll need a decent Nvidia GPU with Fermi or Kepler architecture you can check this by:

```
$ lspci | grep nVidia
```

This should indicate which GPU you have on your system, if you don't have an Nvidia GPU, we're sorry, but you won't be able to use PCL GPU. The output of this you can compare with this link on the Nvidia website, your card should mention compute capability of 2.x (Fermi) or 3.x (Kepler) or higher. If you want to run with a GUI, you can also run:

```
$ nvidia-settings
```

From either CLI or from your system settings menu. This should mention the same information.

First you need to test if your CPU is 32 or 64 bit, if it is 64-bit, it is preferred to work in this mode. For this you can run:

```
$ lscpu | grep op-mode
```

As a next step you will need a up to date version of the Cuda Toolkit. You can get this here, at the time of writing the latest version was 4.2 and the beta release of version 5 was available as well.

First you will need to install the latest video drivers, download the correct one from the site and run the install file, after this, download the toolkit and

install it. At the moment of writing this was version 295.41, please choose the most up to date one:

```
$ wget
    http://developer.download.nvidia.com/compute/cuda/4_2/rel/drivers/devdriver_4.2_linux_64_295.41.
```

Make the driver executable:

```
$ chmod +x devdriver_4.2_linux_64_295.41.run
```

Run the driver:

```
$ sudo ./devdriver_4.2_linux_64_295.41.run
```

You need to run this script without your X-server running. You can shut your X-server down as follows: Go to a terminal by pressing Ctrl-Alt-F1 and typing:

```
$ sudo service gdm stop
```

Once you have installed you GPU device driver you will also need to install the CUDA Toolkit:

```
$ wget
    http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/cudatoolkit_4.2.9_linux_64_ubu
$ chmod +x cudatoolkit_4.2.9_linux_64_ubuntu11.04.run
$ sudo ./cudatoolkit_4.2.9_linux_64_ubuntu11.04.run
```

You can get the SDK, but for PCL this is not needed, this provides you with general CUDA examples and some scripts to test the performance of your CPU as well as your hardware specifications.

CUDA only compiles with gcc 4.4 and lower, so if your default installed gcc is 4.5 or higher you'll need to install gcc 4.4:

```
$ sudo apt-get install gcc-4.4
```

Now you need to force your gcc to use this version, you can remove the older version, the other option is to create a symlink in your home folder and include that in the beginning of your path:

```
$ cd
$ mkdir bin
```

Add 'export PATH=$HOME/bin$:PATH as the last line to your /.bashrc file. Now create the symlinks in your bin folder:

```
$ cd ~/bin $ ln -s <your_gcc_installation> c++
$ ln -s <your_gcc_installation> cc
$ ln -s <your_gcc_installation> g++
$ ln -s <your_gcc_installation> gcc
```

If you use colorgcc these links all need to point to /usr/bin/colorgcc.

Now you can get the latest trunk version (or another one) of PCL and configure your installation to use the CUDA functions.

Go to your PCL root folder and do:

```
$ mkdir build; cd build
$ ccmake ..
```

Press c to configure ccmake, press t to toggle to the advanced mode as a number of options only appear in advanced mode. The latest CUDA algorithms are beeing kept in the GPU project, for this the BUILD_GPU option needs to be on and the BUILD_gpu_¡X¿ indicate the different GPU subprojects.

Press c again to configure for you options, press g to generate the makefiles and to exit. Now the makefiles have been generated successfully and can be executed by doing:

```
$ make
```

If you want to install your PCL installation for everybody to use:

```
$ make install
```

Now your installation is finished!

6.1.2 Tested Hardware

Tested the following methods with:

Method	**Hardware**	**FPS**
Kinfu	GTX680, Intel Xeon CPU E5620 @ 2.40Ghz x 8, 24Gb RAM	20-27
Kinfu	GTX480, Intel Xeon CPU W3550 @ 3.07GHz x 4, 7.8Gb RAM	40
Kinfu	C2070, Intel Xeon CPU E5620 @ 2.40Ghz x 8, 24Gb RAM	29
People pose detection	GTX680, Intel Xeon CPU E5620 @ 2.40Ghz x 8, 24Gb RAM	20-23
People pose detection	C2070, Intel Xeon CPU E5620 @ 2.40Ghz x 8, 24Gb RAM	10-20

6.2 Detecting people and their poses using Point-Cloud Library

In this tutorial we will learn how detect a person and its pose in a pointcloud based on the algorithm presented earlier.

This shows how to detect people with an Primesense device, the full version working on oni and pcd files can be found in trunk. The code assumes a organised and projectable pointcloud, and should work with other sensors then the Primesense device.

In order to run the code you'll need a decent Nvidia GPU with Fermi or Kepler architecture, have a look at the GPU installation tutorial to get up and running with your GPU installation.

6.2.1 The code

The full version of this code can be found in PCL trunk/gpu/people/tools, the following is a reduced version for the tutorial. This version can be found in doc/tutorials/content/sources/gpu/people_detect.

6.2.2 The explanation

Now, let's break down the code piece by piece. Starting from the main routine.

```
int main(int argc, char** argv)
{
  // selecting GPU and prining info
  int device = 0;
  pc::parse_argument (argc, argv, "-gpu", device);
  pcl::gpu::setDevice (device);
  pcl::gpu::printShortCudaDeviceInfo (device);

  // selecting data source
  boost::shared_ptr<pcl::Grabber> capture;
  capture.reset( new pcl::OpenNIGrabber() );

  //selecting tree files
  vector<string> tree_files;
  tree_files.push_back("Data/forest1/tree_20.txt");
  tree_files.push_back("Data/forest2/tree_20.txt");
  tree_files.push_back("Data/forest3/tree_20.txt");
  tree_files.push_back("Data/forest4/tree_20.txt");

  pc::parse_argument (argc, argv, "-tree0", tree_files[0]);
  pc::parse_argument (argc, argv, "-tree1", tree_files[1]);
  pc::parse_argument (argc, argv, "-tree2", tree_files[2]);
  pc::parse_argument (argc, argv, "-tree3", tree_files[3]);

  int num_trees = (int)tree_files.size();
  pc::parse_argument (argc, argv, "-numTrees", num_trees);

  tree_files.resize(num_trees);
  if (num_trees == 0 || num_trees > 4)
    return cout << "Invalid number of trees" << endl, -1;

  try
  {
    // loading trees
    typedef pcl::gpu::people::RDFBodyPartsDetector RDFBodyPartsDetector;
    RDFBodyPartsDetector::Ptr rdf(new RDFBodyPartsDetector(tree_files));
    PCL_INFO("Loaded files into rdf");

    // Create the app
    PeoplePCDApp app(*capture);
    app.people_detector_.rdf_detector_ = rdf;

    // executing
    app.startMainLoop ();
```

```
  }
  catch (const pcl::PCLException& e) { cout << "PCLException: " <<
      e.detailedMessage() << endl; }
  catch (const std::runtime_error& e) { cout << e.what() << endl; }
  catch (const std::bad_alloc& /*e*/) { cout << "Bad alloc" << endl; }
  catch (const std::exception& /*e*/) { cout << "Exception" << endl; }

  return 0;
}
```

First the GPU device is set, by default this is the first GPU found in the bus, but if you have multiple GPU's in your system, this allows you to select a specific one. Then a OpenNI Capture is made, see the OpenNI Grabber tutorial for more info on this.

```
  vector<string> tree_files;
  tree_files.push_back("Data/forest1/tree_20.txt");
  tree_files.push_back("Data/forest2/tree_20.txt");
  tree_files.push_back("Data/forest3/tree_20.txt");
  tree_files.push_back("Data/forest4/tree_20.txt");

  pc::parse_argument (argc, argv, "-tree0", tree_files[0]);
  pc::parse_argument (argc, argv, "-tree1", tree_files[1]);
  pc::parse_argument (argc, argv, "-tree2", tree_files[2]);
  pc::parse_argument (argc, argv, "-tree3", tree_files[3]);
```

The implementation is based on a similar approach as Shotton et al. and thus needs off-line learned random decision forests for labeling. The current implementation allows up to 4 decision trees to be loaded into the forest. This is done by giving it the names of the text files to load.

```
  int num_trees = (int)tree_files.size();
  pc::parse_argument (argc, argv, "-numTrees", num_trees);
```

An additional parameter allows you to configure the number of trees to be loaded.

```
    typedef pcl::gpu::people::RDFBodyPartsDetector RDFBodyPartsDetector;
    RDFBodyPartsDetector::Ptr rdf(new RDFBodyPartsDetector(tree_files));
    PCL_INFO("Loaded files into rdf");
```

Then the RDF object is created, loading the trees upon creation.

```
    // Create the app
    PeoplePCDApp app(*capture);
    app.people_detector_.rdf_detector_ = rdf;

    // executing
    app.startMainLoop ();
```

Now we create the application object, give it the pointer to the RDF object and start the loop. Now we'll have a look at the main loop.

```
void
startMainLoop ()
{
  cloud_cb_ = false;

  PCDGrabberBase* ispcd =
      dynamic_cast<pcl::PCDGrabberBase*>(&capture_);
  if (ispcd)
    cloud_cb_= true;

  typedef boost::shared_ptr<openni_wrapper::DepthImage>
      DepthImagePtr;
  typedef boost::shared_ptr<openni_wrapper::Image> ImagePtr;

  boost::function<void (const boost::shared_ptr<const
      PointCloud<PointXYZRGBA> >&)> func1 = boost::bind
      (&PeoplePCDApp::source_cb1, this, _1);
  boost::function<void (const ImagePtr&, const DepthImagePtr&, float
      constant)> func2 = boost::bind (&PeoplePCDApp::source_cb2,
      this, _1, _2, _3);
  boost::signals2::connection c = cloud_cb_ ?
      capture_.registerCallback (func1) : capture_.registerCallback
      (func2);

  {
    boost::unique_lock<boost::mutex> lock(data_ready_mutex_);

    try
    {
      capture_.start ();
      while (!exit_ && !final_view_.wasStopped())
      {
        bool has_data = data_ready_cond_.timed_wait(lock,
            boost::posix_time::millisec(100));
        if(has_data)
        {
          SampledScopeTime fps(time_ms_);

          if (cloud_cb_)
            process_return_ =
                people_detector_.process(cloud_host_.makeShared());
          else
            process_return_ = people_detector_.process(depth_device_,
                image_device_);

          ++counter_;
        }

        if(has_data && (process_return_ == 2))
          visualizeAndWrite();
      }
      final_view_.spinOnce (3);
    }
```

```
    catch (const std::bad_alloc& /*e*/) { cout << "Bad alloc" <<
        endl; }
    catch (const std::exception& /*e*/) { cout << "Exception" <<
        endl; }

    capture_.stop ();
  }
  c.disconnect();
}
```

This routine first connects a callback routine to the grabber and waits for valid data to arrive. Each time the data arrives it will call the process function of the people detector, this is a fully encapsulated method and will call the complete pipeline. Once the pipeline completed processing, the results can be fetched as public structs or methods from the people detector object. Have a look at doc.pointclouds.org for more documentation on the available structs and methods. The visualizeAndWrite method will illustrate one of the available methods of the people detector object:

```
void
visualizeAndWrite(bool write = false)
{
  const PeopleDetector::Labels& labels =
      people_detector_.rdf_detector_->getLabels();
  people::colorizeLabels(color_map_, labels, cmap_device_);

  int c;
  cmap_host_.width = cmap_device_.cols();
  cmap_host_.height = cmap_device_.rows();
  cmap_host_.points.resize(cmap_host_.width * cmap_host_.height);
  cmap_device_.download(cmap_host_.points, c);

  final_view_.showRGBImage<pcl::RGB>(cmap_host_);
  final_view_.spinOnce(1, true);

  if (cloud_cb_)
  {
    depth_host_.width = people_detector_.depth_device1_.cols();
    depth_host_.height = people_detector_.depth_device1_.rows();
    depth_host_.points.resize(depth_host_.width *
        depth_host_.height);
    people_detector_.depth_device1_.download(depth_host_.points, c);
  }

  depth_view_.showShortImage(&depth_host_.points[0],
      depth_host_.width, depth_host_.height, 0, 5000, true);
  depth_view_.spinOnce(1, true);

  if (write)
  {
    if (cloud_cb_)
      savePNGFile(make_name(counter_, "ii"), cloud_host_);
    else
```

```
        savePNGFile(make_name(counter_, "ii"), rgba_host_);
      savePNGFile(make_name(counter_, "c2"), cmap_host_);
      savePNGFile(make_name(counter_, "s2"), labels);
      savePNGFile(make_name(counter_, "d1"),
          people_detector_.depth_device1_);
      savePNGFile(make_name(counter_, "d2"),
          people_detector_.depth_device2_);
    }
  }
```

Line 144 calls the RDF getLabels method which returns the labels on the device, these however are a discrete enum of the labels and are visually hard to recognize, so these are converted to colors that illustrate each body part in line 145. At this point the results are still stored in the device memory and need to be copied to the CPU host memory, this is done in line 151. Afterwards the images are shown and stored to disk. Compiling and running the program

Add the following lines to your CMakeLists.txt file:

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

project(people_detect)

find_package(PCL 1.7 REQUIRED)

include_directories(${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})

#Searching CUDA
FIND_PACKAGE(CUDA)

#Include the FindCUDA script
INCLUDE(FindCUDA)

cuda_add_executable (people_detect src/people_detect.cpp)
target_link_libraries (people_detect ${PCL_LIBRARIES})
```

After you have made the executable, you can run it. Simply do:

```
$ ./people_detect
```