DEVINE Documentation

Release 0.1.0

Ismael Balafrej, Julien Chouinard-Beaupre, Félix Labelle, Adam Le

Dec 18, 2018

Contents

Welcome,

In this document you will find various technical documentation of the DEVINE project such as installation process, in depth module information and even more!

CHAPTER 1

Architecture

1.1 Architecture

The DEVINE project is base on a distributed ROS system.

This allow the project to run on a real robot, while rendering heavy tasks like images segmentation and body tracking in another more perfomant computer.

1.1.1 Distributed Computing

As part of this project we experimented with running ROS nodes on multiple machines.

This solution was developed to suit our project's needs by allowing it to run on a remote server with its dependencies inside a container.

Initial network configuration

First, create a docker network. In this tutorial we will use subnet `172.20.0.0/16`, but you may need to change subnet so it does not conflict with existing networks. On each machine run:

```
$ docker network create --subnet 172.20.0.0/16 --gateway 172.20.0.1 devine
```

This will create a bridge interface named `br-${networkId}`. The network id can be recovered using `docker network ls`.

Bringing up the nodes

When bringing up the containers, assign them an ip (within the subnet) and a hostname. ROS nodes also need to be able to reach the rosmaster specified by the environment variable ROS_MASTER_URI.

For example, run on the first machine:

```
$ docker run -ti --rm --runtime=nvidia --network devine \
      --hostname machine1 --ip 172.20.0.16 \
      --add-host machine2:172.20.0.15 \
      -e ROS_MASTER_URI=http://172.20.0.16:11311 \
$ devine bash
```

On the second machine run:

```
$ docker run -ti --rm --runtime=nvidia --network devine \
      --hostname machine2 --ip 172.20.0.15 \
      --add-host machine1:172.20.0.16 \
      -e ROS_MASTER_URI=http://172.20.0.16:11311 \
$ devine bash
```

Tunneling

To link the containers we use ssh tunneling.

From machine1 run:

```
$ ssh -o Tunnel=ethernet -w 123:123 root@machine2
```

This will create a tap interface named tap123 on each side.

We connect these taps to the bridge. On each machine run:

```
$ ip link set tap123 master br-$(docker network ls -f name=devine | grep devine | awk
↪'{print $1}')
$ ip link set tap123 up
```

You can also check out the diagrams below in order to learn the basics on how each DEVINE modules interacts with each others:

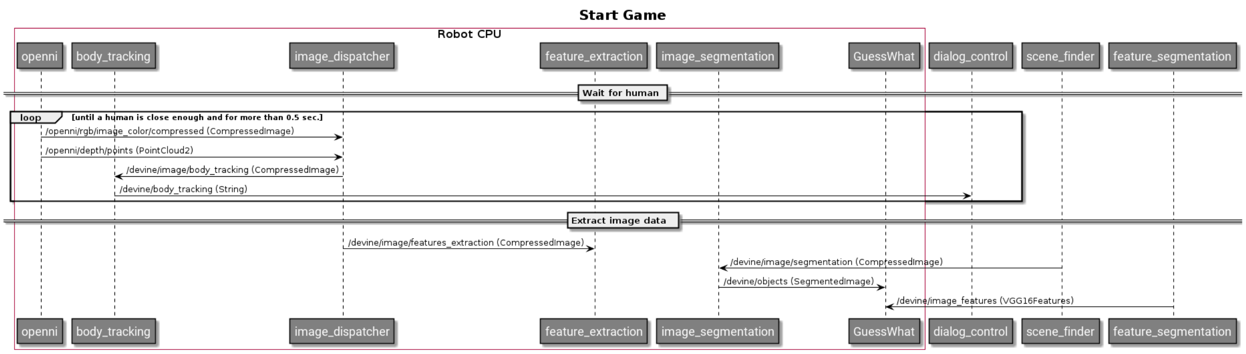
1.1.2 Image Pipeline

Being able to interface with GuessWhat?! and users requires taking inputs from the robot's Kinect 360 and processing them accordingly. The first link in the chain is the image dispatcher, which takes compressed images, validates that they are not blurred, and based on the game state, sends them onto the next node in the chain.

The next node to recieve the image, temporally, is the body tracking node. Using OpenPose we try to determine if a person is within range to begin a game. If it is the case, after the scene's picture is taken, the image is sent to the segmentation and feature extration nodes.

Interfacing with GuessWhat?! requires extracting: a list of all objects, bounding boxes around them and a feature vector (FC8 of a VGG16). Respectively the segmentatation and feature nodes are responsible for this.

Below is a UML showing the sequence of interactions between the different modules.



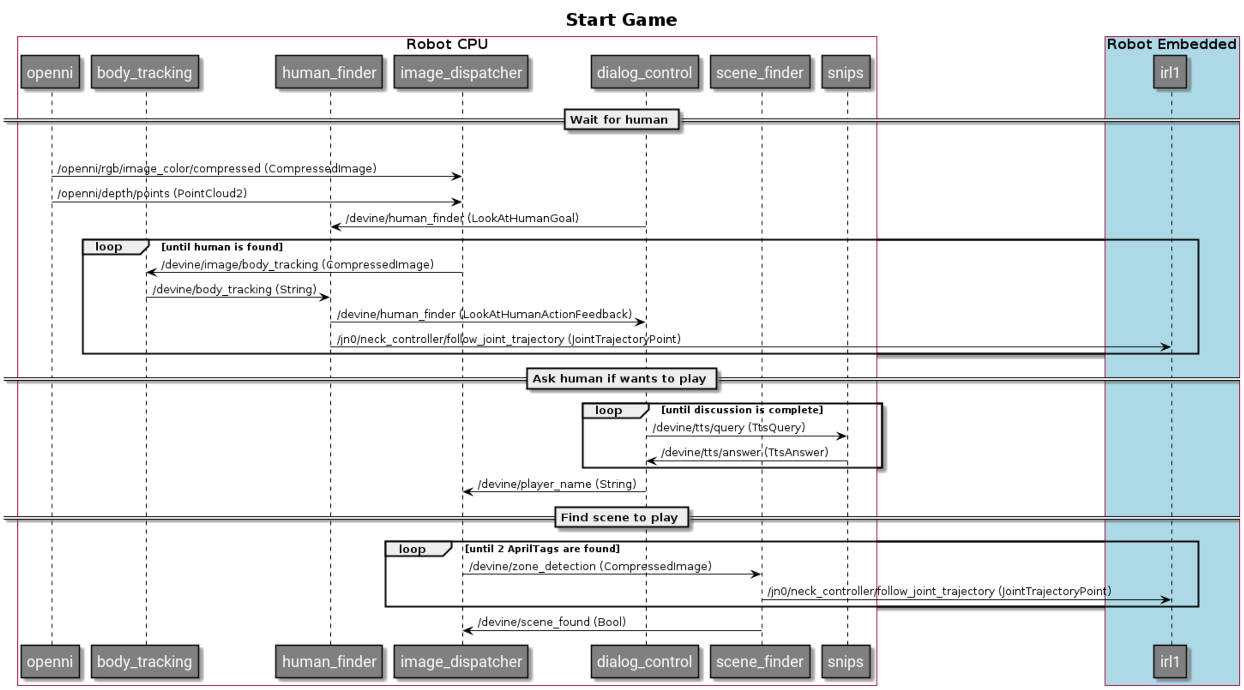
Additional Information

Specifics for each node can be found at the following links:

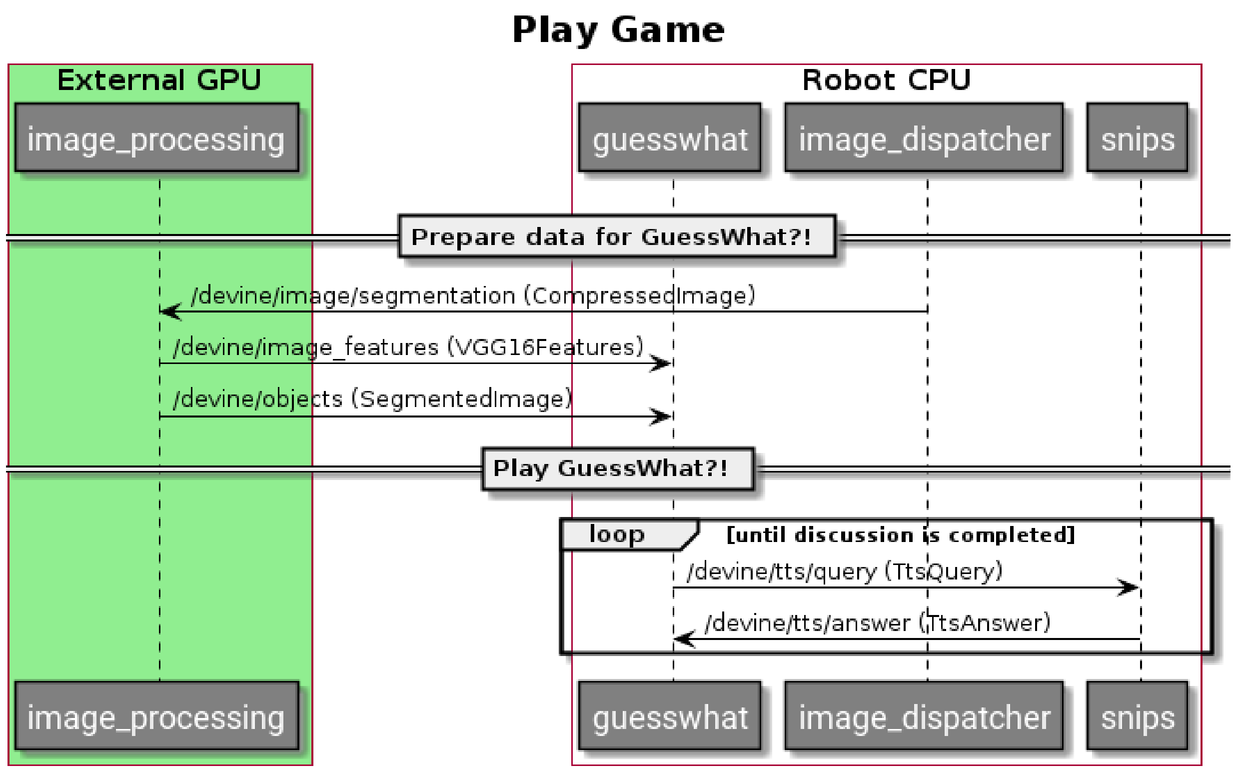
- *Image disptacher*
- *Segmentation*
- *Feature extraction*
- *Segmentation*
- *Bodytracking*
- *Depth mask*

1.1.3 UML Sequence Diagrams

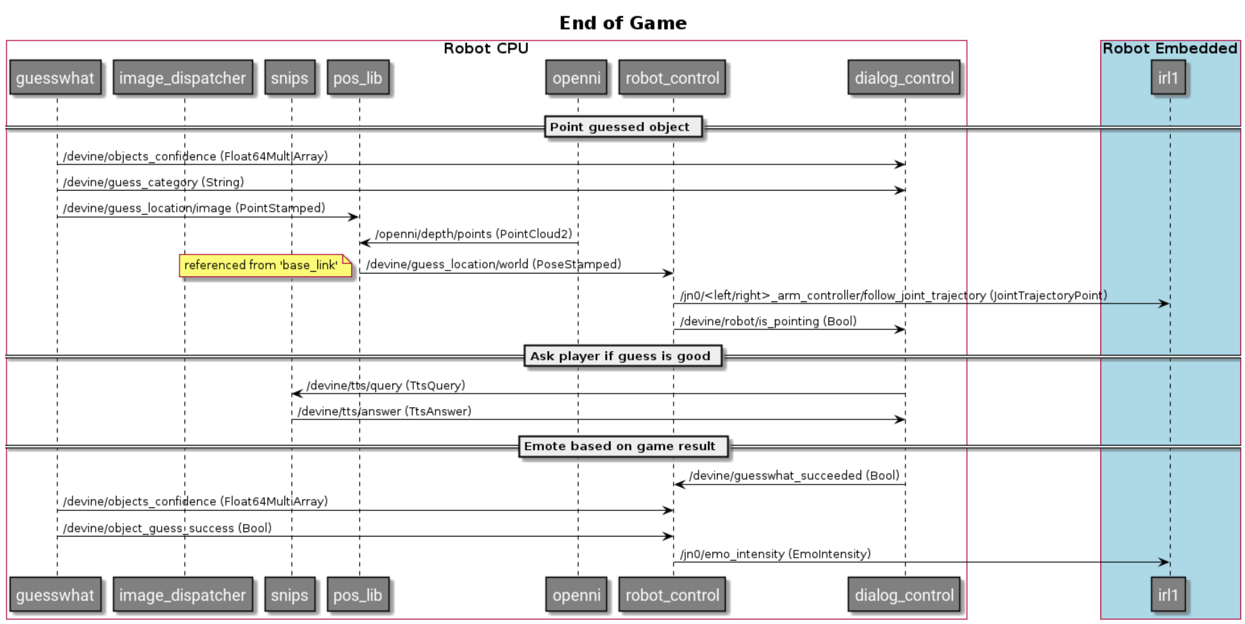
Start Game



Play Game



End of Game



CHAPTER 2

External Links

DEVINE website.

DEVINE GitHub.

CHAPTER 3

Installation

Global installation process for the project can be found here

3.1 Getting Started

DEVINE is a project with **many** dependencies such as ROS.

In this section, you can find links to different installation types that we support.

That being said, we **highly** recommand going with the *Docker* way.

3.1.1 Installation

Docker

Docker is an application which runs a program in an isolated environment with its dependencies, akin to a virtual machine. Docker is portable, lightweight and allows for compatibility.

How to get started

First, navigate to the docker folder.

Build the docker image for CPU use:

```
$ ./build.sh
```

Or build the docker image for GPU use:

```
$ ./build.sh --gpu
```

Theses commands will get the devine-base image and build the devine image.

Once the build is complete, you can validate by running `sudo docker images`. One docker should be named devine. With an image in hand, simply run the command to launch an instance of your docker image:

```
$ ./run.sh
```

You will arive in a ubuntu like terminal which has the same layout as the code base. To exit, use ctrl+d.

Note: both `run.sh` and `build.sh` have some arguments that can be set depending on your usage. Use the argument `--help` for more information.

Information about the DEVINE docker images

The DEVINE project uses two docker images:

- *devine-base*: contains all of the projects dependencies and can be rebuilt if necessary using `./base/build-base.sh`.
- *devine*: contains the actual code.

Separating the dependencies from the code speed up further DEVINE builds.

Useful commands

```
$ sudo docker container ls                                      # Lists all␣
↪containers currently running
$ sudo docker exec -it {containerId} bash                       # starts another␣
↪bash in a given docker container
$ docker cp {path/to/filename} {containerId}:{Destination/Path/} # copy a file into a␣
↪specific docker image
```

Ubuntu 16.04 LTS

We recommend you to install it on a fresh copy of Ubuntu 16.04 LTS.

The following steps will install all the dependencies for the DEVINE project.

1. Create a `catkin workspace` directory like explained in the ROS tutorial.
2. Create `src` directory under it.
3. Clone the DEVINE repository in `src/`. **Make sure not to rename the repository**
4. Navigate to `DEVINE/scripts`.
5. Run the following command:

```
$ ./install.sh {path/to/catkin/src} {path/to/devine/root}
```

GPU Usage - Optional

If you want to use your GPU instead of your CPU for the computation, follow the GPU setup bellow.

GPU Setup

Following the steps shown at *Ubuntu 16.04 LTS*, Tensorflow will use the CPU for all the computational problems. To make TensorFlow use your GPU, you need to do some more installation.

There is **many** ways to install TensorFlow / CUDA. This guide is only one of them.

As the writting of this documentation, TensorFlow GPU is officially supported for CUDA 9.0 with Nvidia drivers > 384.x and cuDNN >= 7.2

After these steps, you will have installed:

- CUDA 9.0 and it's dependencies
- cuDNN 7.3.0 and it's dependencies
- TensorFlow with GPU support and it's dependencies

Step 0 - Dependencies

You should have most of theses already.

```
$ sudo apt-get install build-essential cmake git unzip zip python-pip python3-pip␣
↪python-virtualenv swig python-wheel libcurl3-dev curl python-dev python3-dev python-
↪numpy python3-numpy
$ sudo apt-get install linux-headers-$(uname -r)
```

Step 1 - Cleanup

You need to make sure that you have nothing *Nvidia* or *CUDA* related installed on your machine.

You can follow theses steps if you want to uninstall *CUDA*, *Nvidia* and *Tensorflow* from your machine.

Do not worry, *Nvidia* drivers will be installed with *CUDA* later on.

- Remove all *Nvidia* and *CUDA* related installation

Danger: Be careful, the following steps are destructive and will uninstall and remove any Nvidia drivers installed

```
$ sudo apt-get purge nvidia*
$ sudo apt-get purge cuda* # You may need to manually purge them, for example sudo␣
↪apt-get purge cuda-cusparse-9-0
$ dpkg -l | grep '^rc' | awk '{print $2}' | grep cuda | sudo xargs dpkg --purge   #␣
↪verify the output first so you don't delete something else...
$ dpkg -l | grep '^rc' | awk '{print $2}' | grep nvidia | sudo xargs dpkg --purge #␣
↪verify the output first so you don't delete something else...
$ sudo apt-get autoremove
$ sudo apt-get autoclean
$ sudo rm -rf /usr/local/cuda*
```

- Uninstall any TensorFlow installation

```
$ pip uninstall tensorflow
$ pip uninstall tensorflow-gpu
```

- reboot!

```
$ sudo reboot
```

Step 1 - Install CUDA

You can download CUDA from *Nvidia* website and manually install it, but it is preferable to use their repository and install it using *Ubuntu*'s package manager.

- Download and install CUDA 9.0

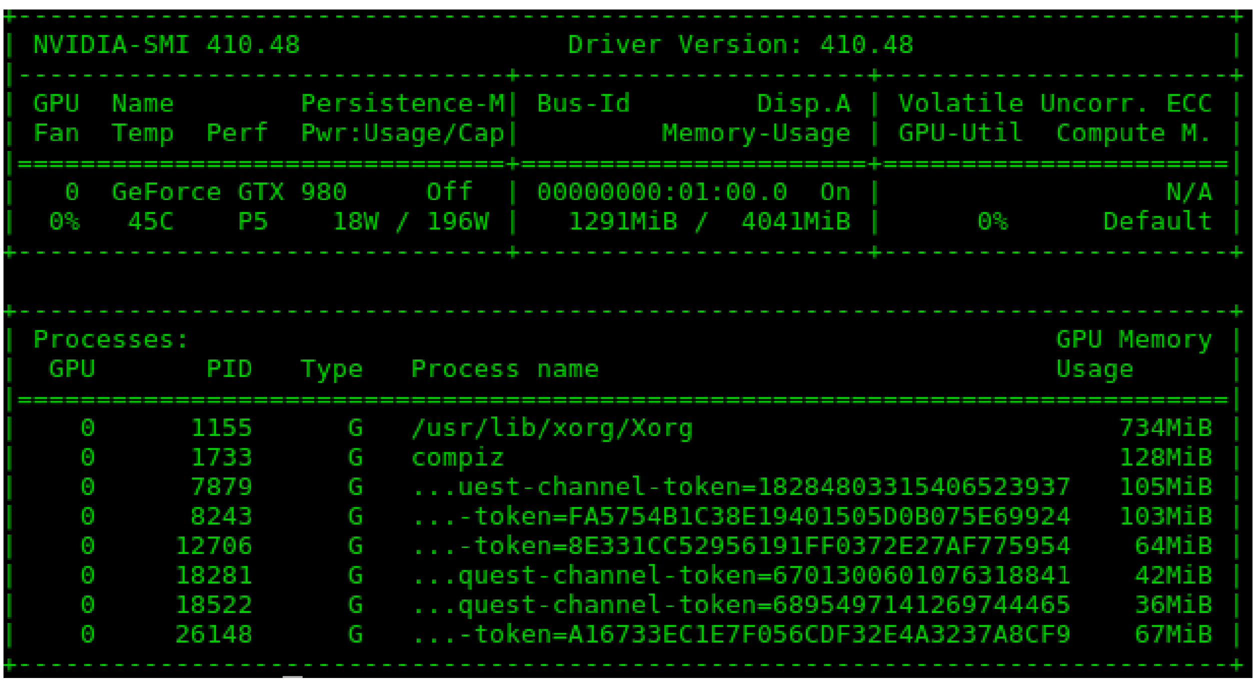
```
$ curl -O http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/
↪cuda-repo-ubuntu1604_9.0.176-1_amd64.deb
$ sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/
↪repos/ubuntu1604/x86_64/7fa2af80.pub
$ sudo dpkg -i ./cuda-repo-ubuntu1604_9.0.176-1_amd64.deb
$ sudo apt-get update
$ sudo apt-get install cuda-9-0 # this may take a while (~1.7G)
```

- reboot!

```
$ sudo reboot
```

- Verify installation

```
$ nvidia-smi # should return a list of GPUs with some metrics. Make sure the driver's␣
↪version shown on the top is > 384.x
```



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 410.48                 Driver Version: 410.48                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 980     Off  | 00000000:01:00.0  On |                  N/A |
|  0%   45C    P5    18W / 196W |   1291MiB /  4041MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1155      G   /usr/lib/xorg/Xorg                           734MiB |
|    0      1733      G   compiz                                       128MiB |
|    0      7879      G   ...uest-channel-token=18284803315406523937   105MiB |
|    0      8243      G   ....-token=FA5754B1C38E19401505D0B075E69924  103MiB |
|    0     12706      G   ....-token=8E331CC52956191FF0372E27AF775954   64MiB |
|    0     18281      G   ...quest-channel-token=6701300601076318841    42MiB |
|    0     18522      G   ...quest-channel-token=6895497141269744465    36MiB |
|    0     26148      G   ....-token=A16733EC1E7F056CDF32E4A3237A8CF9   67MiB |
+-----------------------------------------------------------------------------+
```

```
$ nvcc -V # should return the CUDA compiler version installed. Make sure the version␣
↪is 9.0

# example
```

(continues on next page)

(continued from previous page)

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2017 NVIDIA Corporation
Built on Fri_Sep__1_21:08:03_CDT_2017
Cuda compilation tools, release 9.0, V9.0.176
```

If you do not pass any verification steps, go back to *Step 1 - Cleanup*.

Step 2 - Install cuDNN

Download *cuDNN* 7.3.0 for *CUDA* 9.0 from *Nvidia*'s cuDNN archive.

You may need to create a account if you do not have one yet.

- Download and install

```
$ sudo tar -xzvf cudnn-9.0-linux-x64-v7.3.0.29.tgz
$ sudo cp cuda/include/cudnn.h /usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn*
```

- Update your bashrc.

In the case you have different *CUDA* version installed, change the folder to the *CUDA* version you want.

```
$ echo 'export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/cuda/lib64:/usr/local/
↪cuda/extras/CUPTI/lib64"' >> ~/.bashrc
$ echo 'export CUDA_HOME=/usr/local/cuda' >> ~/.bashrc
$ echo 'export PATH="$PATH:/usr/local/cuda/bin"' >> ~/.bashrc
$ . ~/.bashrc
```

Step 3 - Install TensorFlow GPU

- Uninstall *TensorFlow*

```
$ pip uninstall tensorflow
```

- Install *TensorFlow* with GPU support under *python3*

```
$ python3 -m pip install --user tensorflow-gpu
```

- Verify installation

```
$ python3
    import tensorflow as tf
    hello = tf.constant('Hello, TensorFlow!')
    sess = tf.Session() # You should see some information about your GPU in the output
    print(sess.run(hello))

# in another shell
$ nvidia-smi # you should see in the processe list python3
```

If you do not pass any verification steps, go back to *Step 1 - Cleanup*.

Step 5 - Profit

Have fun!

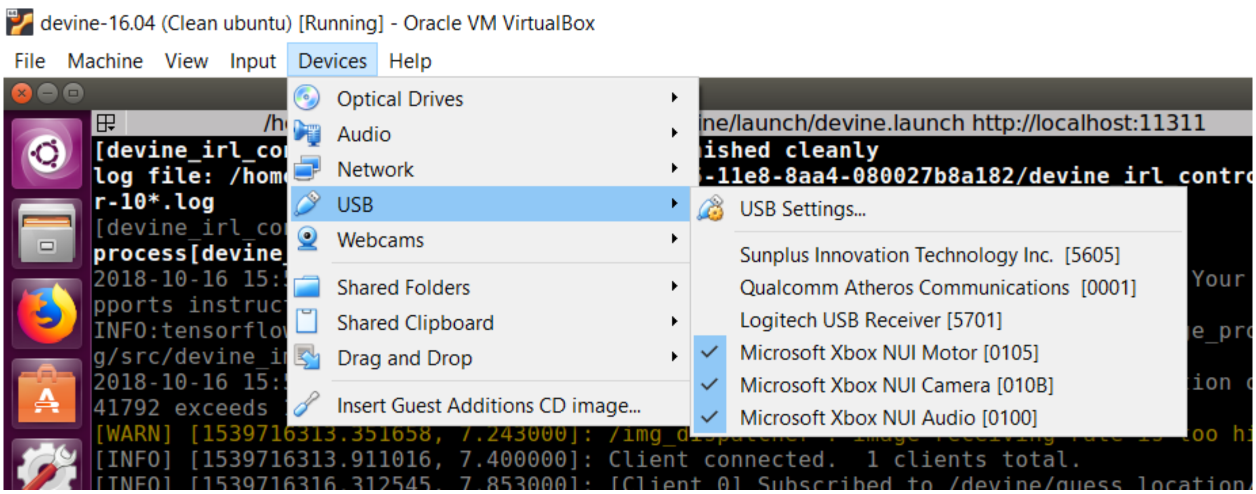
Virtual Box

The DEVINE project can be installed in a virtual machine.

To do so, make sure you have a VM with Ubuntu 16.04 installed, and follow the steps of installing *Ubuntu 16.04 LTS*.

Note about running the project in Virtual Box

To allow the Xbox Kinect connected physically to the host to communicate with the VM, you must link your USB devices from the host to the client:



There should be three devices to select for the Kinect:

- Microsoft Xbox NUI Motor
- Microsoft Xbox NUI Camera
- Microsoft Xbox NUI Audio

If you get an error while linking the devices, it may be possible that the device is busy by another process. The simplest way to solve that is to restart the client and restart the host.

You may also need to install Oracle VM VirtualBox Extension Pack in order to allow the use of **USB 2.0** in the settings of your VM.

3.1.2 Launching the project

The project uses a *devine.launch* file which can start all the required ROS nodes.

```
$ roslaunch devine devine.launch
```

By default, this will launch all the nodes. You can also specify which nodes to launch, like so:

```
$ roslaunch devine devine.launch launch_all:=false dashboard:=true
```

Also by default, the launch file is made to run on a real robot. To run in **simulation** only, you can change the *sim* argument:

```
$ roslaunch devine devine.launch sim:=true
```

CHAPTER 4

All DEVINE modules

4.1 Audio

4.1.1 Description

We use *SNIPS* as our voice assistant to interact with the robot with the voice.

4.1.2 ROS Installation

As *SNIPS* does not officially support Ubuntu Xenial, its intallation comes with a few caveat.

1. Run `$ sudo npm install -g snips-sam` to install SAM
2. Go to /usr/lib/node_modules/snips-sam/lib/session/ssh.js (or usr/local/lib/node_modules/snips-sam/lib/session/ssh.js) and change line 426 to [. . .] `echo "deb https://debian.snips.ai/ stretch stable main"` [. . .]
3. Install an upstream version of libc `$ sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test && sudo apt-get update && sudo apt-get upgrade -y libstdc++6`
4. Connect with `$ sam connect localhost`
5. `$ sam init`
6. If you get an error at this stage, add this line `your_username ALL=(ALL) NOPASSWD: ALL` at the end of your sudoers file with the command : `$ sudo visudo`, then try again from step 4.
7. Test the speaker with `$ sam test speaker`
8. Test the microphone with `$ sam test microphone`
9. If tests are not conclusive or quality is poor, try selecting a different speaker and microphone with : `$ sam setup audio`

10. Install our assistant `$ wget https://github.com/projetdevine/static/releases/download/v0.0.1/assistant.zip && sudo unzip -o assistant.zip -d /usr/share/snips`

Once the *SNIPS* team adds support for Ubuntu Xenial step 2 and 3 will not be necessary. Note that our assistant was tested for version 0.58.3 of the snips-platform-voice package.

4.1.3 Usage

```
$ roscore #start ROS master
$ rosrun devine_dialog snips.py __ns:=devine #run snips node
$ sam watch
$ rostopic echo /devine/tts/answer #listen to the answers
```

To send custom data to the topic used by snips, do :

`$ rosrun rqt_gui rqt_gui`

- Select topic : /devine/tts/query
- Select type : devine_dialog/TtsQuery
- Select a frequency
- Fill out the 'text' (ex: "Is the object blue ?"), 'uid' (ex: 1) and 'answer_type' (ex: 1) fields.

Or, run this command : `$ rostopic pub /devine/tts/query devine_dialog/TtsQuery '{text: "Is the object blue?", uid: 1, answer_type: 1}'`

4.2 Bodytracking

4.2.1 Description

Detecting people is an important part of our project. By detecting nearby humans, we can follow them using the robots eyes and find potential players. This functionality is provided by tf-pose-estimation.

The body tracking node outputs a JSON which contains a skeleton of all the people in a given image. It is is published on the *image/body_tracking* topic.

4.2.2 ROS Installation

Run the install script *install.sh*

4.2.3 Usage

```
$ rosrun devine_image_processing body_tracking.py __ns:=devine
```

4.3 Dashboard

4.3.1 Description

The dashboard is a web based project where we integrate all of the ROS nodes and gives us a centralized operation center. You can subscribe to any ROS topic and see what is being send on any topic and you can also send information to them. It's main goal is to allow us to verify that the whole DEVINE system works in harmony.

It can also be used to demo the project.

4.3.2 Usage

Once the project is installed on your machine, you can simply launch the dashboard like so:

```
$ roslaunch devine devine.launch launch_all:=false dashboard:=true
```

The process will listen and update whenever there is a change in the code.

4.3.3 Manual installation

```
$ sudo npm i -g webpack
$ npm install
$ pip3 install -r requirements.txt
$ sudo apt-get install ros-kinetic-rosbridge-server
```

4.3.4 Adding a view

Create an html layout for your view. E.g: *views/myview.html*. Or reuse one similar to yours.

include it in *views/index.html*, keep these class attributes *uk-width-expand command-view* and change the name attribute.

```
<div class="uk-width-expand command-view" name="myview" hidden>
    {% include 'myview.html' %}
</div>
```

Add it to the menu with a class attribute matching the name you used previously.

```
<li class="command-myview command-menu">My view</li>
```

Code your view in its own file (*src/myview.js*) and import it in *src/app.js*.

4.4 Depth mask

4.4.1 Description

To filter out extraneous objects in the background, the kinect's depth sensor is used to create a mask. This mask blacks out all objects further then 1.5m.

The body tracking node outputs the masked image. It is is published on the *sensor_msgs/CompressedImage* topic.

4.4.2 ROS Installation

Run the install script *install.sh*

4.4.3 Usage

```
$ rosrun devine_image_processing mask.py __ns:=devine
```

4.5 Feature extraction

4.5.1 Description

VGG-16 is used to extract image features which was in turn used by the question generator. It was coded using tensorflow and is available on github.

The feature extraction node outputs an array which contains the class of the object, which contains the FC8 layer's output. It is is published on the *features* topic.

4.5.2 ROS Installation

Run the install script *source install_package.sh*

4.5.3 Usage

```
$ rosrun devine_image_processing features_extraction.py __ns:=devine
```

4.6 GuessWhat

4.6.1 Description

This project makes use of the open source code provided alongside the original GuessWhat?! research. On our side, we add the strict minimum to have it act as a ROS node.

4.6.2 Installation

Since guesswhat is not yet a proper python module, it has to be added to your python path:

```
$ git clone --recursive https://github.com/GuessWhatGame/guesswhat.git /tmp/somewhere
$ export PYTHONPATH=/tmp/somewhere/src:$PYTHONPATH
```

Also install python dependencies:

```
$ pip3 install -r requirements.txt
```

Build this ROS package using:

```
$ catkin_make -C ~/catkin_ws
```

4.6.3 Usage

Roslaunch:

```
$ roslaunch devine devine.launch launch_all:=false guesswhat:=true
```

Monitor questions:

```
$ rostopic echo /devine/tts/query
  text: "is it a person ?"
  uid: 1234
  answer_type: 1
  ---
```

Send some test inputs:

```
$ cd example
$ python3 example.py
```

Reply:

```
$ rostopic pub /devine/tts/answer devine_dialog/TtsAnswer '{original_query: {text:
↪"is it a person ?", uid: 1234, answer_type: 1}, probability: 1.0, text: "yes"}'
```

4.7 Head Coordinator

4.7.1 Description

Scene finding:

We're using april tags and the apriltags2_ros library to find the scene where the objets are located. The head will rotate looking down until both tags are found, and then the image_dispatcher will proceed by taking a picture of the scene found.

4.7.2 Installation

1. Clone the apriltags2_ros repository in your catkin workspace, presumably ~/catkin_ws.

```
$ git clone git@github.com:dmalyuta/apriltags2_ros.git
```

2. Copy the settings available in *./src/head_coordinator/apriltags2_config* in the config directory of the newly cloned repository under *./apriltags2_ros/config*

3. Build the module using catkin_make:

```
$ catkin_make -C ~/catkin_ws
```

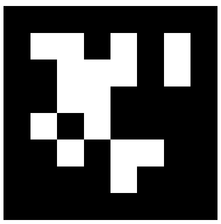
4.7.3 Usage

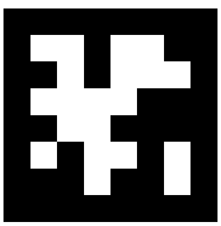
Using a kinect, place two 11cm by 11cm tag36h11 identified 0 and 1 in the top left and botom right corners of the scene you are trying to find.

```
$ roslaunch devine devine.launch launch_all:=false kinect:=true find_scene:=true
```

The robot's head should turn in order to find the scene when the *zone_detection* topic is triggered.

4.7.4 Example of april tags





These are examples of 36h11 tag ids #0 and #1. The tags must be 11cm wide when printed, and positioned respectively in the top left and bottom right corners. It's also preferable that they directly face the camera to have the best accuracy possible.

4.8 Image disptacher

4.8.1 Description

The image dispatcher is responsible for distributing images from the kinect to the various modules that need them in the correct order. It takes raw images, checks them for blur, applies the depth mask and sends the processed images to be segmented and have their features extracted.

4.8.2 ROS Installation

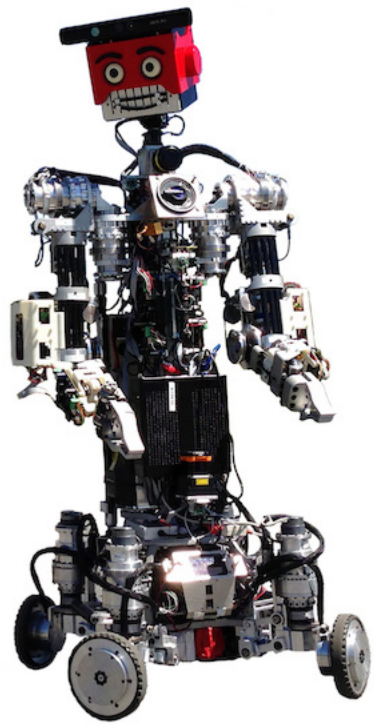
Run the install script *install.sh*

4.8.3 Usage

```
$ rosrun devine_image_processing image_dispatcher.py __ns:=devine
```

4.9 Robot Behavior

4.9.1 Description



We currently use robot IRL-1 from IntRoLab for our demonstrations. See official IRL-1 GitHub for more details.

4.9.2 Possible Mouvements

- **Point to position (x, y, z) with**
  - Right arm
  - Left arm
  - Head
- **Open and close**
  - Right gripper
  - Left gripper
- **SIMULATION ONLY, Do complex movements with arms and head:**
  - Happy (confidence >= threshold, success 1)
  - Satisfied (confidence < threshold, success 1)
  - Disappointed (confidence >= threshold, success 0)
  - Sad (confidence < threshold, success 0)
- **Facial expression**
  - Anger

- Joy
- Sad
- Surprise

4.9.3 Running Examples

Before running any examples, you need to:

1. Launch jn0 with RViz UI

```
$ roslaunch jn0_gazebo jn0_empty_world.launch # for simulation
$ roslaunch jn0_bringup jn0_standalone.launch # for real robot
```

2. Launch devine_irl_control nodes

```
$ roslaunch devine_irl_control devine_irl_control.launch sim:=true # for simulation
```

3. Load RViz configuration

```
File -> Open Config -> src/robot_control/launch/irl_point.rviz
```

You can now execute any of the examples:

- Point to position [x, y, z]

```
$ rosrun devine_irl_control example_point.py --point 0.6,0.3,0.5 --look 1,-0.6,0.5 __ns:=devine
# Position is referenced from base_link
```

- Do complex move (SIMULATION ONLY!!!)

```
$ rosrun devine_irl_control example_emotion.py -c 0 -s 0 __ns:=devine
```

4.9.4 Dependencies

See *package.xml* for dependencies.

4.9.5 Topics

Topics input and output from this module

In/Out	Topic	ROS Message
In	/devine/guess_location/world	geometry_msgs/PoseStamped *
In	/devine/robot/robot_look_at	
In	/devine/robot/head_joint_traj_point	trajectory_msgs/JointTrajectoryPoint
Out	/devine/robot/is_pointing	std_msgs/Bool
Out	/devine/robot/is_looking	
Out	/devine/robot/err_pointing	std_msgs/Float64MultiArray

* PoseStamped are relative to *base_link* (see *frame_id*)

4.9.6 Constants

File *irl_constant.py* contains

- Controllers names
- Joints names
- Joints limits

4.10 Segmentation

4.10.1 Description

We currently use Mask R-CNN to detect and segment the objects of our images. It was coded using tensorflow and trained using MSCOCO, which means that the classes it uses to segment objects are compatible with GuessWhat?!

The segmentation node outputs a *SegmentedImage* object which contains the class of the object, a box which delimits the object and a segmentation mask. It is is published on the *objects* topic.

4.10.2 ROS Installation

Run the install script *install.sh*

4.10.3 Usage

```
$ rosrun devine_image_processing segmentation.py __ns:=devine
```

4.11 Video

4.11.1 Description

We currently use a Microsoft Kinect for a Xbox 360 in combination with OpenNI to use it inside the ROS ecosystem.

4.11.2 Pre requirement Installation

1. Install OpenNI

```
$ sudo apt-get install ros-kinetic-openni-launch ros-kinetic-openni-camera ros-
↪kinetic-openni-description
$ sudo apt-get install ros-kinetic-compressed-image-transport #Image compression␣
↪plugin
```

2. Start OpenNI server

```
$ roslaunch devine devine.launch launch_all:=false kinect:=true dashboard:=true
```

3. View Data

You can use the dashboard (http://localhost:8080) or the image_view package:

```
$ rosrun image_view image_view image:=/openni/rgb/image_color #color
$ rosrun image_view image_view image:=/openni/rgb/image_mono #mono
$ rosrun image_view disparity_view image:=/openni/depth_registered/disparity
↪#disparity
```

4. Read the ROS OpenNI documentation for more info!

4.11.3 ROS Installation

1. Run the install script *./install_package.bash*
2. Build the module using *catkin_make*:

```
$ roscd
$ cd ..
$ catkin_make
```

CHAPTER 5

Tests

5.1 Tests

The tests are made using Python unittest.

5.1.1 Adding test cases

To add a test case, simply copy the *testcase_template.py* into your test folder, then import your test case into *test_suite.py*.

5.1.2 Running the unit tests with catkin

From your catkin workspace run the following:

```
$ catkin_make run_tests
```

This command will launch all the necessary nodes and run the tests.

Launching a single test case

Each *test_*.py* file corresponds to a test case.

Each one of these files can run individually like so:

```
$ python DEVINE/tests/src/devine_tests/*/test_*.py
```

CHAPTER 6

Cheat Sheet

6.1 ROS Cheat Sheet

Here you can see a couple of usefull ROS commands to help you out!

- `$ roscore`
  - Starts the ros core node, you need this before starting any other node.
- `$ rosrun {rosPackageName} {pythonFileContainingTheRosNode} [__ns:=namespace]`
  - Example: `$ rosrun devine_irl_control node_facial_expression.py __ns:=devine`
  - This will start the node specified inside the *node_facial_expression.py*
- `$ rostopic pub {/topic_name} std_msgs/{dataType} {Payload}`
  - Example: `$ rostopic pub /devine/objects_confidence std_msgs/Float64MultiArray "{layout: {dim: [{label: '', size: 0, stride: 0}], data_offset: 0}, data: [0,0.8, 0.7]}"`
  - This will publish the specified payload to the specified topic.
- `$ rostopic echo {topicName}`
  - Example: `$ rostopic echo /devine/robot/facial_expression`
  - This will listen and print out any messages on the specified topic.
- `$ roslaunch devine devine.launch`
  - This will launch **ALL** Devine nodes.
  - You can also use this to launch specific nodes like so `$ roslaunch devine devine.launch launch_all:=false dashboard:=true`
- `$ rosrun topic_tools throttle messages /openni/rgb/image_color/compressed 0.33 /devine/image/segmentation`

- Segments every 30 seconds

- `$ rosrun rqt_gui rqt_gui`
  - Starts a GUI with many usefull ROS development tools that enables you to subscribe and monitor ROS topics for example.
- `$ rosrun rqt_top rqt_top`
  - See the actually ressources consumed by your ROS environment.