

Scientific Visualization and Virtual Reality

Practical assignments

Part 2: Visualization project

Jiří Iša

November 5th, 2005

Abstract

The practical assignment consists of two parts; the second part addresses the students ability to perform a larger project using VTK and the knowledge obtained during the Part 1 implementation.

In this article we briefly describe the probabilistic localization of the autonomous mobile robot, explain, why the visualization of the probabilistic distribution is usefull, why current visualization techniques are not sufficient and we discuss the provided implementation.

1 Introduction

For the Part 2 of the practical assignments students were expected either to select one of the predefined topics, or suggest the own one. This text is about the second possibility. Probabilistic autonomous mobile robot localization and its visualization were suggested as a topic and developed.

In this article we briefly describe the probabilistic localization, explain, why the visualization of the probabilistic distribution is usefull, why current visualization techniques are not sufficient and we discuss the provided implementation.

The Java wrappers of self-compiled VTK, version 4.4, on Linux operating system, were used to fullfill the assigned tasks.

2 Probabilistic robot localization

During the last years robotics experiences its new boom. Autonomous mobile robots are getting to our everyday lives. From vacuum cleaners through grass cutters to fully autonomous car-like vehicles (DARPA Grand Challenge by U. S. Army - <http://www.grandchallenge.org>, ELROB by German Bundeswehr - <http://www.elrob2006.org>).

These robots may either simply react to the environment, as in case of simple vacuum cleaners, or they can be provided with, or build itself, a map. In the second case it is crucial for the robot to know, where it is. Due to imperfection of the sensors and the inaccurate movement of the robot, errors in position prediction and/or calculation can accumulate quite easily and dangerously.

This is the point where Mr. Dieter Fox from University of Germany introduced the Markov probabilistic approach [1]. It has been, however, observed, that the suggested technique is too computationally expensive and cannot be used for on-line navigation in the real environment.

The improvement in the form of the Monte Carlo localization (MCL) was suggested soon afterwards [2] at Carnegie Mellon University and University of Bonn.

Both ideas are based on the fact, that the robot's position calculation may be based only on the previous position and the performed movement. It is not dependent on the situations more in the past. This is so called *Markov assumption*.

Both algorithms consists of two phases:

- prediction
- correction

During the prediction phase the probabilistic distribution for being at any point is changed according to the performed movement. Because it is known, that the robot has inaccurate actuators, this update has a form of Gaussian distribution.

During the correction phase the distribution is corrected according to the actual sensory input. Again the Gaussian noise on the sensors is expected.

3 The importance of the visualization

The Monte Carlo localization uses thousands of *samples*, that are moved and filtered according to the probabilistic movement and sensory model.

Because the Monte Carlo localization was introduced very soon after the original Markov localization, the visualization of the complete grid did not have a time to be developed.

Thanks to the *samples* nature of MCL the only visualization technique that is widely used is drawing the samples in the map.

Such an approach requires from the user to extract the probability from the density of the points by himself. This is not only highly inaccurate, but it may also be almost impossible, when there is a small area with too many occluding samples.

It also lacks the support for visualization of the probability distribution over the direction. And obviously the direction of the robot is a vital part of the robot navigation.

4 Implementation

As mentioned in the Introduction, the Scientific Visualization Library (VTK) was used for the visualization. It is available for the free download (<http://www.kitware.com>) and the source code is available under the GNU Public Licence (<http://www.gnu.org>). VTK is also offered in most major Linux distributions.

4.1 Scenario

To work with the probabilistic distribution, we have to have the area and its map first. For this task the playground of international robotic competition Eurobot 2006, as described on <http://www.eurobot.org>, was chosen.

The playground is 300cm long and 210cm wide. The robot might have up to 36cm in diameter, if circular. There is a wall at every side and few column obstacles on the playground.

The robot has to be equipped with some sensors, that allow it to correct its position estimate. The distance measurement to the obstacles in three directions was chosen.

4.2 Environment and robot simulation

After the search for a capable robotic simulator, that would allow the localization incorporation, inaccurate movement and noisy sensors, the own simulator was written.

Due to the scenario with only few types of the objects and one type of sensor (except the odometry) it appeared to be not too difficult.

The Markov localization approach as described in [1] was chosen as it was considered to be easier to implement. It also provides us with the probability in every point of interest and for every selected direction directly. It does not have to be estimated from *samples*.

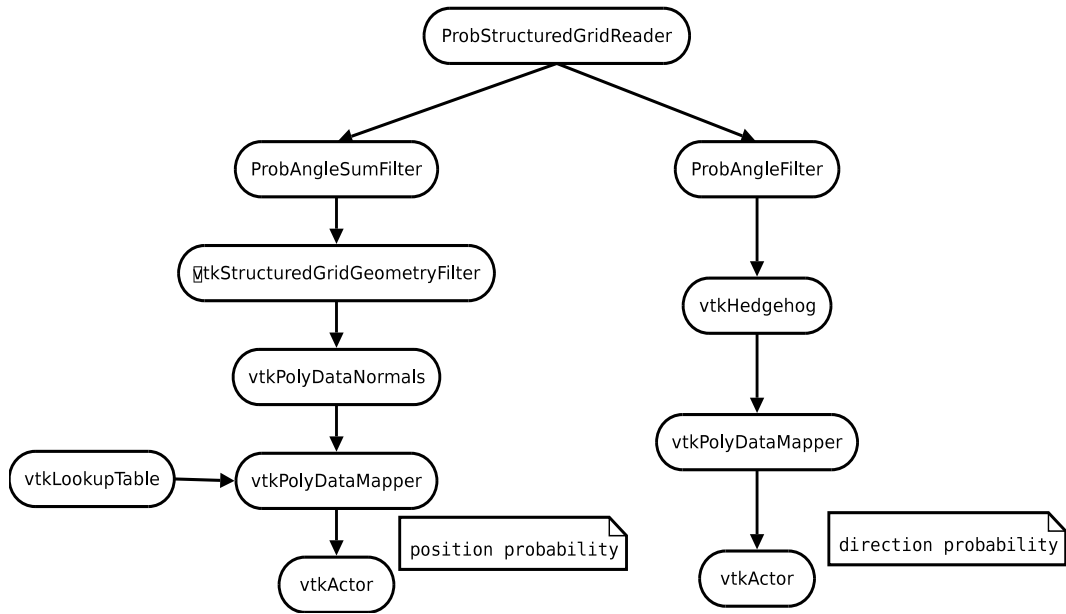


Figure 1: Probability distribution visualization pipeline

After some experimenting the points of interest and important angles selection was done as follows:

- Points of interest: The playground was covered with the resolution of three centimeters in both directions.
- Angle resolution: All possible angles were discretized to sixteen values.

This gives us roughly (not all positions are possible, as the robot could not go through the wall) 100x70x16 resolution.

Because the Markov localization performed even more time consuming, than expected, the robot's path was predefined and the values of probability distributions for two hundred timesteps were saved to the file. This file is then parsed during the visualization and shown to the user.

The simulator is not a part of the provided source code and it is not described further. It would need another article and the topic is pretty well covered in [1].

4.3 Visualization

4.3.1 *ProbStructuredGridReader*

As the name of the class suggests, it is responsible for input data reading and parsing.

Because the data obtained from the simulation have the nature of the structured grid, *vtkStructuredGrid* is used for data storage.

```

/*-----
* CREATE THE GRID
*-----
*/
vtkStructuredGrid grid = new vtkStructuredGrid();

grid.SetDimensions(X_RES, Y_RES, ANGLE_RES);
vtkFloatArray dataArray = new vtkFloatArray();
grid.GetPointData().SetScalars(dataArray);
  
```

```

/** Set up the points */
grid.SetPoints(new vtkPoints());
grid.GetPoints().SetNumberOfPoints(X_RES * Y_RES * ANGLE_RES);

for(int x = 0; x < X_RES; x++){
    for(int y = 0; y < Y_RES; y++){
        for(int angle = 0; angle < ANGLE_RES; angle++){
            grid.GetPoints().SetPoint(angle * X_RES * Y_RES + x * Y_RES + y,
                                      -105 + ROBOT_WIDTH + 3 * x,
                                      -150 + ROBOT_WIDTH + 3 * y,
                                      -Math.PI + angle * Math.PI/ 16);
        }
    }
}

/*-----
* fill the data grid with data
*-----
*/
...
        grid.GetPointData().GetScalars().InsertTuple1(offset, value);
...

```

4.3.2 ProbAngleSumFilter

The data from simulation contain probability distribution over position AND direction. We would like to visualize the probability distribution only over the position.

The probability of the position is defined as a sum over the probabilities of every direction in this position:

$$P(\text{position}) = \sum_{\text{every direction}} P(\text{position}, \text{direction})$$

vtkProbAngleSumFilter takes the structured grid provided by *ProbStructuredGridReader*, makes a sum over directions and creates new structured grid with the resolution X_RES x Y_RES x 1. The last, Z, dimension is used for the elevation of the point of the surface proportionally to the calculated probability of the position. The probability is also thresholded, nonzero values are enlarged by additive constant, scaled and stored as the scalar value.

```

//initialize the data array and points
vtkFloatArray dataArray = new vtkFloatArray();
dataArray.SetNumberOfValues(X_RES * Y_RES);
vtkStructuredGrid output = new vtkStructuredGrid();
output.GetPointData().SetScalars(dataArray);
output.SetDimensions(X_RES, Y_RES, 1);
vtkPoints points = new vtkPoints();
points.SetNumberOfPoints(X_RES * Y_RES);
output.SetPoints(points);
vtkDataArray inputScalars = input.GetPointData().GetScalars();
vtkDataArray outputScalars = output.GetPointData().GetScalars();

```

```

//fill in the values
for(int x = 0; x < X_RES; x++){
    for(int y = 0; y < Y_RES; y++){
        double sum = 0;

        for(int angle = 0; angle < ANGLE_RES; angle++){
            double value;
            value = inputScalars.GetTuple1(angle * X_RES * Y_RES + x * Y_RES + y);
            sum += value;
        }

        //the scalar value contrast is thresholded
        outputScalars.SetTuple1(x + y * X_RES, sum > 0.0001 ? 140 * sum : 0);
        points.SetPoint(x + y * X_RES,
            -105 + ROBOT_WIDTH + 3 * x,
            -150 + ROBOT_WIDTH + 3 * y,
            sum);
    }
}

```

4.3.3 ProbAngleFilter

The second branch of the visualization pipeline (see fig. 1) starts with the *ProbAngleFilter*.

It reads the data provided by *ProbStructuredGridReader* and creates the similar structure, this time with the resolution $X_RES \times Y_RES \times 1$. Vector values are used instead of scalars.

To every point vectors in every direction (every of the sixteen provided by the simulator) is placed. Its length is proportional to the probability of the given direction.

This is later used for the *hedgheg* visualization.

Improvement It has been observed that two different short *hedgheg* lines may connect one to another and form one long instead. This is highly misleading as it suggests high direction probability on the position, where it isn't.

The problem was solved using the last, elevation axis. Every *hedgheg* line is placed above the surface proportionally to the probability value. The higher probability, the higher (and closer to the viewer) the line is placed. Because lines may still connect, small random Gaussian noise is also added to the elevation coordinate.

```

//initialize points and vector array
vtkFloatArray dataVectorsArray = new vtkFloatArray();
vtkPoints points = new vtkPoints();

dataVectorsArray.SetNumberOfComponents(3);
dataVectorsArray.SetNumberOfTuples(X_RES * Y_RES * ANGLE_RES);
points.SetNumberOfPoints(X_RES * Y_RES * ANGLE_RES);
vtkDataArray inputScalars = input.GetPointData().GetScalars();

```

```

//fill in values
for(int x = 0; x < X_RES; x++){
    for(int y = 0; y < Y_RES; y++){
        double value;

        for(int angle = 0; angle < ANGLE_RES; angle++){
            value = inputScalars().GetTuple1(angle * X_RES * Y_RES + x * Y_RES + y);

            double currentAngle = -Math.PI + angle * 2 * Math.PI / ANGLE_RES;
            int offset = x + y * X_RES + angle * X_RES * Y_RES;

            points.InsertPoint(offset, -105 + 18 + 3 * x, -150 + 18 + 3 * y,
                30 + 20 * value + rnd.nextGaussian());
            dataVectorsArray.SetTuple3(offset,
                value * Math.cos(currentAngle),
                value * Math.sin(currentAngle),
                0);
        }
    }
}

```

```

out.GetStructuredGridOutput().SetPoints(points);
out.GetStructuredGridOutput().GetPointData().SetVectors(dataVectorsArray);

```

4.3.4 Further processing of probability distributions

So far we have created only the data grid. Now we need to convert it into polygons to be shown.

First the geometry of the structure describing the position probability is extracted using *vtkStructuredGridGeometryFilter*. Then the normals are calculated and the surface is smoothed little bit to hide the low resolution of the simulation using *vtkPolyDataNormals*.

```

/* Extract the surface geometry */
vtkStructuredGridGeometryFilter structured = new vtkStructuredGridGeometryFilter();
structured.SetExtent(0, X_RES, 0, Y_RES, 0, 1);
structured.SetInput(probAngleSumFilter.GetStructuredGridOutput());

/* Normals and smoothing */
vtkPolyDataNormals smooth = new vtkPolyDataNormals();
smooth.SetInput(structured.GetOutput());
smooth.SetFeatureAngle(90);

```

Then the color look-up table for the *vtkPolyDataMapper* is prepared. The color choice is made in such a way, that low value color, green, resembles the playground color and cannot be distinguished by the user from the playground surface itself. High value is set to red.

```

/* Colorization settings */
vtkLookupTable lut = new vtkLookupTable();
lut.SetHueRange(0.33, 0);

```

And all we have is passed to the new *vtkPolyDataMapper*:

```

vtkPolyDataMapper mapper = new vtkPolyDataMapper();
mapper.SetInput(smooth.GetOutput());
mapper.SetLookupTable(lut);
//The range is preset to disable automatic checking for maximum and minimum value.
mapper.SetScalarRange(0, 1);

```

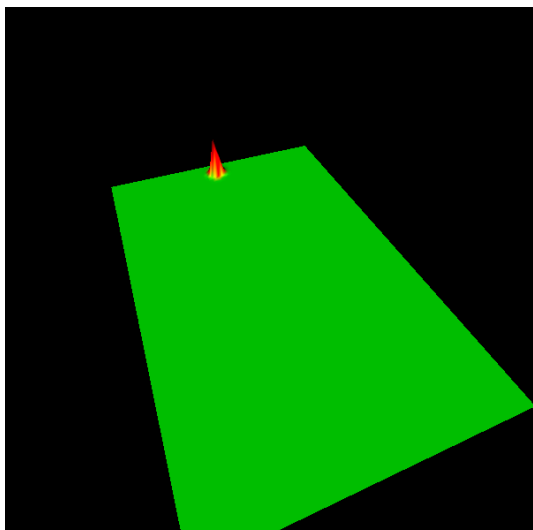


Figure 2: Position probability distribution

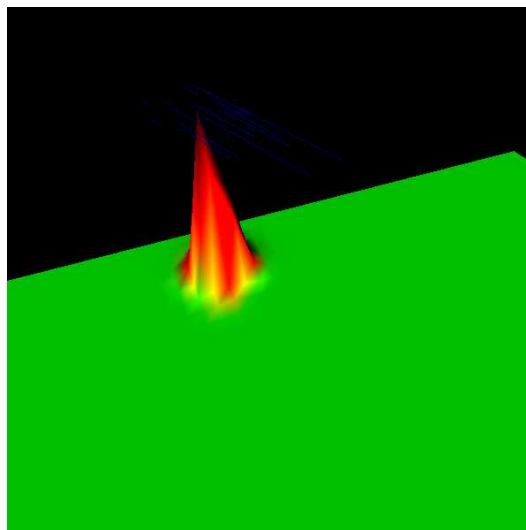


Figure 3: Position and direction probability distribution

The actor is created as usual. The scale of the z-axis is set high to enlarge the displacement of probability axes with small values.

```
/* Position probability distribution actor */
vtkActor probabilityDistribution = new vtkActor();
probabilityDistribution.SetMapper(mapper);
probabilityDistribution.SetScale(1, 1, 180);
```

As we can see on the fig. 2. there is still a lot to be done.

4.3.5 Further processing of the direction probability distribution

The direction probability distribution is one of the crucial points of the project. As mentioned before, the lack of direction visualization in current projects is one of the reasons to start this one.

Originally it was planned to use the color of the surface to map the most probable direction of the robot. However, this appeared to be the wrong way as the user's translation of the perception from color to direction is not intuitive and easy.

After some experimenting the *hedgehog* visualization was selected. It was being avoided for a long time, because the scene contains too many points and the simulation provides us with sixteen values for each of them. If the length of the *hedgehog* lines was proportional to the underlying probability, it could easily lead to the occlusion and the mess in the scene.

Surprisingly, this does not happen at all. Probabilities provided by the simulator are mostly zeros or very low, compared to the maximum value. This means that if we threshold these values, we can draw lines from all the points with probability over the threshold.

The only problematic thing happened to be the effect of lines connecting one to another. This has been solved by positioning different lines to the different latitudes, as described in the chapter 4.3.3.

Figure 3. shows that the result looks reasonably.

4.3.6 Environment visualization

The probability distribution might look nice, but it tells nothing to someone who does not know the map by heart.

Therefore the environment has to be modeled as well.

The implementation is largely similar to the code in *Practical assignments: Part 1* and code examples are omitted here. Only the visualization pipeline is shown on the Figure 4.

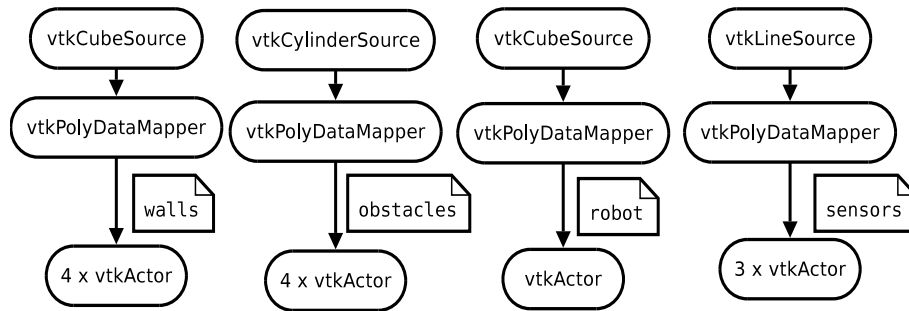


Figure 4: Visualization pipeline of the environment model

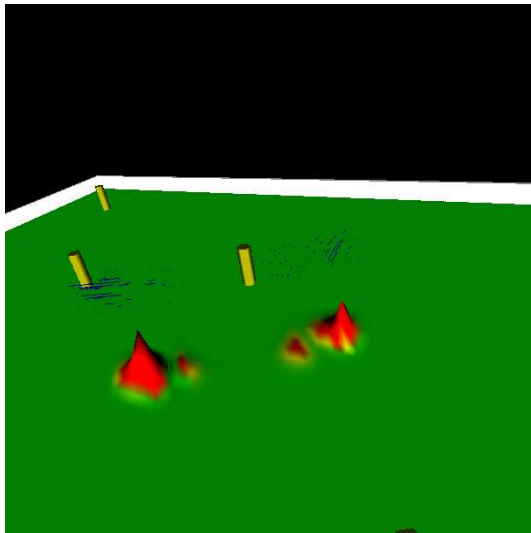


Figure 5: Now we see, what the robot thinks.
But where it really is?

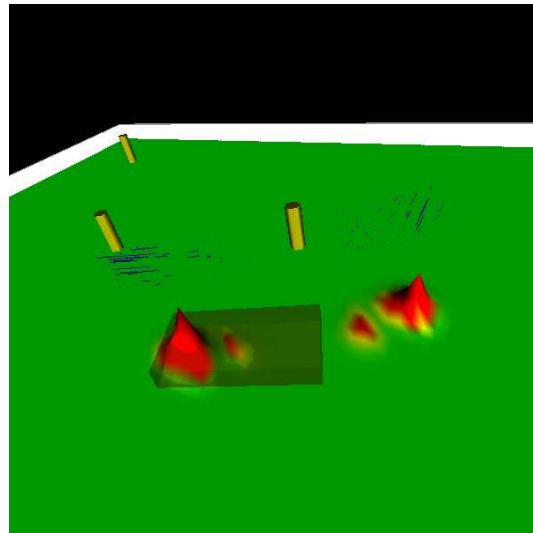


Figure 6: Here it is!

4.3.7 Robot visualization

We clearly cannot show the environment without showing the robot. If we do that, situations like on the Figure 5 occur. We know, what the robot thinks, but we do not know, where the robot is. We cannot see the accuracy of the localization method.

The robot visualization consists of two parts:

- position of the robot
- robot's orientation

The main problem we are facing is the fact, that the robot is most of the time at the same spot, where most of the interesting probability distribution events occur.

We might have a different plane for a robot and the probability distribution. However, in such a scenario it is nearly impossible for the user to match these planes because of the perspective.

The described problem might be nicely solved with the almost transparent robot placed to the same place as the probability distribution.

To express the orientation of the robot, its shape is used. In the simulation, the robot is considered round-shaped. For the visualization the robot is considered to be box-shaped. It resembles the car and intuitively it is obvious, where is front or back and where is the side. The problem of the front/back resolution is left over for the animation. Robot is expected to move mostly forward.

4.3.8 Sensors visualization

During the animation it became obvious, that it might be useful to understand the reason, why sometimes the probability goes higher, sometimes it goes lower and sometimes it "jumps" to the other location.

As the localization is strongly based on the sensory input, it could be expected that all these phenomena could be explained, if only we knew, what does the robot see.

Three simple lines in the direction of three robot's distance sensors were designed.

As we can see on the figure 7, we also obtained, as a sideeffect, the clue showing the facing direction of the robot.

4.3.9 Let's animate

The most important topic on probabilistic localization is, how it changes over time, how it depends on the sensory input, how it evolves.

The only and obvious way, how to visualize these effects is to animate the sequence. The simple animation loop, as in *Practical assignments: Part 1 - Starship Enterprise*, was implemented. Because of the high level of similarity the code is not listed here.

5 Discussion

5.1 Missing features

Currently the users interaction is limited very much. After the animation automatically starts, user has to wait until it ends and only after that the scene can be moved, rotated, etc.

The implementation has been designed with the better interaction in mind. Sadly enough, because of the issues discussed below, the plan for "play forward", "fast forward", "slow backward" and so on had to be dropped.

There are still clear remnants in the code. For instance the *ProbStructuredGridReader's* data are accessed through a bidirectional iterator and all time snapshots of the simulation are in the memory at the same moment. Even though in the current implementation only one snapshot is accessed at the moment and after it is left, it is never accessed again.

The idea, that could not have been implemented, was to have one thread for data loading, one for animation and one for interaction with the user.

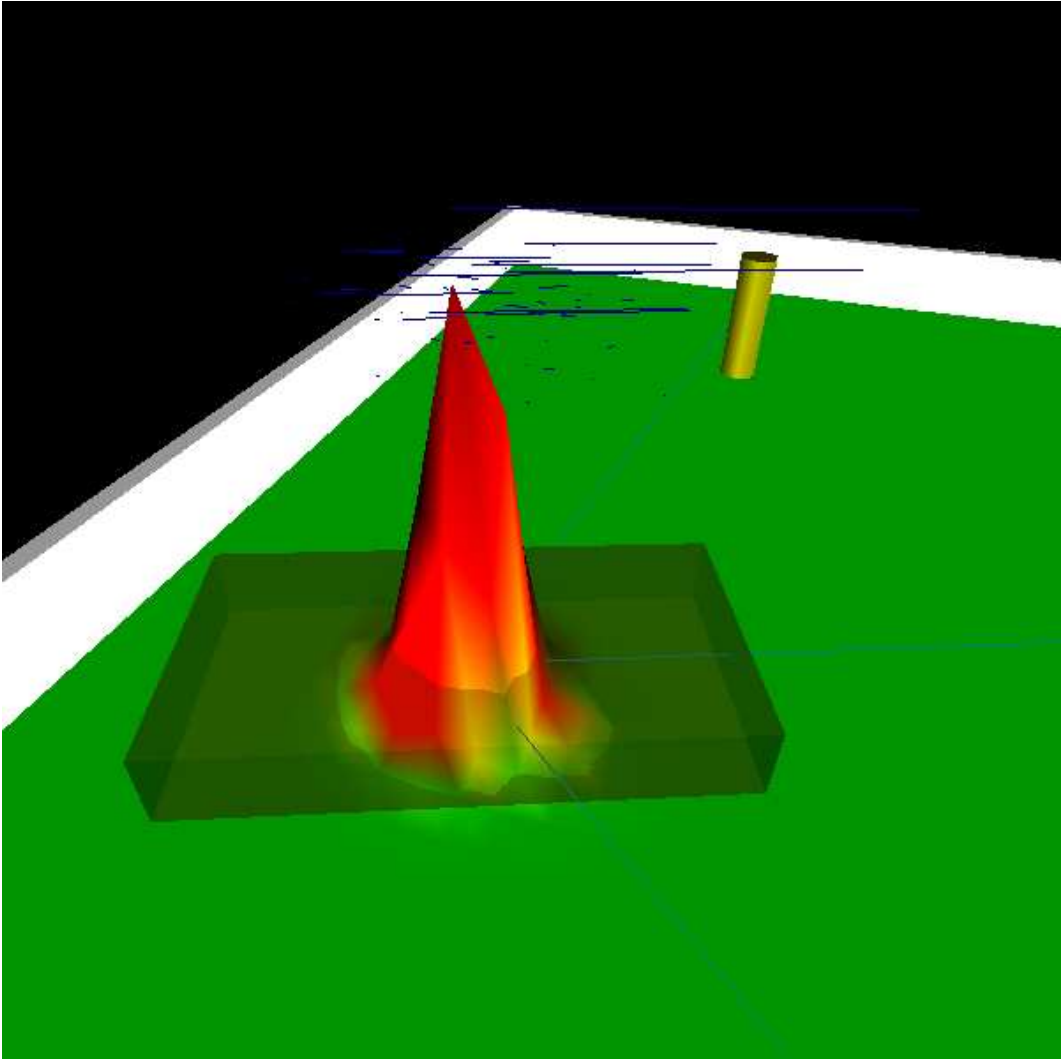


Figure 7: All visualization elements

5.2 Implementation issues and troubles

5.2.1 Overloading

Most, probably all, implementation troubles were caused by the initial choice to use the Java wrappers around the original C++ library.

According to the own experience and to the answer of one of the VTK authors (Ken Martin) on the dedicated VTK forum, overloading cannot be used any more. The ancestor C++ class cannot call methods in the Java descendant.

This appears in the code, where the own *reader* and *filters* are used.

The following code shows, how the *vtkProgrammableSource* is used instead.

```
//create the programmable source
vtkProgrammableSource src = new vtkProgrammableSource();
//instantiate a filter, that knows, where to get the data from and where to put it
ProbAngleSumFilter angleSumFilter = new ProbAngleSumFilter(dataIterator, src);
//inform the programmable source which method of which object to execute to get filled
src.SetExecuteMethod(angleSumFilter, "run");
```

As one more drawback of this method, the internal information of VTK classes, that takes care for checking, if the update is necessary, does not work any more. All instances have to be invalidated manually and asked for an update (again information from the forum):

```
//read next probability distribution snapshot from the simulation
reader.loadNext();

//update positions of the robot and sensors.
//These work good and don't have to be invalidated manually afterwards.
updateRobotPosition(robotActor, sensorActors, dataIterator);

//invalidate the programmable probability distribution source
src.Modified();
//force an update
src.Update();

//position probability surface
structured.Modified();
structured.Update();

//smoother
smooth.Modified();
smooth.Update();

//position probability mapper
mapper.Modified();
mapper.Update();

//position probability actor
probabilityDistribution.Modified();
```

The same has to be done for the branch of the pipeline with the direction probability processing.

5.2.2 Multithreading

The library crashes the application when two different Java threads access the same class. It does not even have to be the same instance of the object.

This is probably an issue of the method the wrapping is created and compiled, rather than the bug of the VTK library.

As a consequence there cannot run the interaction and animation thread at the same moment. Not even mentioning one more thread, that reads the data from file and stores them to VTK classes at the same moment.

This behavior is addressed nor in the documentation, nor in the dedicated VTK forum, nor anywhere else on the web.

6 Conclusion

The obtained visualization of the probabilistic robot localization fulfilled all the requirements placed on the probabilistic robot localization visualization. It, or the obtained video, could be suggested as the learning aid for lectures of mobile robotics.

The implemented simulator may be easily replaced by another one, using another method, like MCL. This way the behavior of the Monte Carlo approximation could be compared to the full Markov localization.

The interactivity issue described above still remains to be solved.

The VTK library appeared to be a great help. For every used method, except the own data preprocessing, there is a class already implemented.

However, usage of the Java wrappers should be strongly discouraged, except of the simple "load and show" projects, where there is no real need for multithreading.

References

- [1] Dieter Fox: *Markov Localization: A Probabilistic Framework for Mobile Robot Localization and Navigation*, Doctoral thesis, University of Bonn, Germany, (1998)
- [2] Frank Dellaert, Dieter Fox, Wolfram Burgard, Sebastian Thrun: *Monte Carlo Localization for Mobile Robots*, Carnegie Mellon University, Pittsburg PA, USA and University of Bonn, Germany, (2000)
- [3] Will Schroeder, Ken Martin, Bill Lorensen: *The Visualization Toolkit, 3rd edition*, Kitware Inc., ISBN: 1-930934-07-6, (2002)