Khepera Project
In context of maze solving

By

Xavier Bouthillier

Bachelor Project

In

Informatic bachelor
in Albert-Ludwigs-Universität Freiburg
as exchange student
from Université de Montréal

Freiburg im Breisgau, Deutschland

Abstract

In this project, I developed a plant and the required tools to command the Khepera III robot in a physical environment.

The main goal of this document is to provide a description of the tools and how to use them to drive the Khepera robot throughout a maze. Therefore, this is not only the report of my project, but also an instruction manual.

You will find here descriptions of two different types of detectors, the description of a simple navigation system for the Khepera and the description of the maze plant customized for this project.

I really hope this project will be useful in the future. If any questions are left unanswered by this document, I will gladly answer them.
xavier.bouthillier@gmail.com

Contents

1. Introduction

What is Khepera III

The Khepera III is a small differential wheeled robot (diameter of 5.5cm). It has eleven infrared sensors, nine around it and two on the bottom. Additionally, it has five ultrasonic sensors around it. It can communicate with Bluetooth and is equipped with a battery that provides it approximately four hours of autonomy. There is a commercialized device called KoreBot II that can provide the Khepera III limited computing capacity.

Kind of research that will make use of it

As the research in the lab is directed toward expensive machine learning algorithms, the limited computing capacity of the KoreBot II was not interesting. It would be more efficient to use the computing power of a dedicated computer and communicate with the Khepera III through Bluetooth.

The actual researches of Manuel Blum do not explicitly concern problems of mobile robotics. Thus, it has been decided that the first environment in which the Khepera III will evolve would not make use of the sensors, but of an external detector that keeps track of the robot position and orientation. The main idea is simply to solve a physical problem based on the knowledge gained from a virtual training on this problem, in other words, from a simulation.

Classical maze problem

The CL^2 system already includes a maze plant where it is possible to solve the shortest-path problem from a given position to a given target. As I concentrated on building the modules needed to detect and pilot the Khepera III, I used this plant so I do not have to worry about the problem itself (solving a maze).

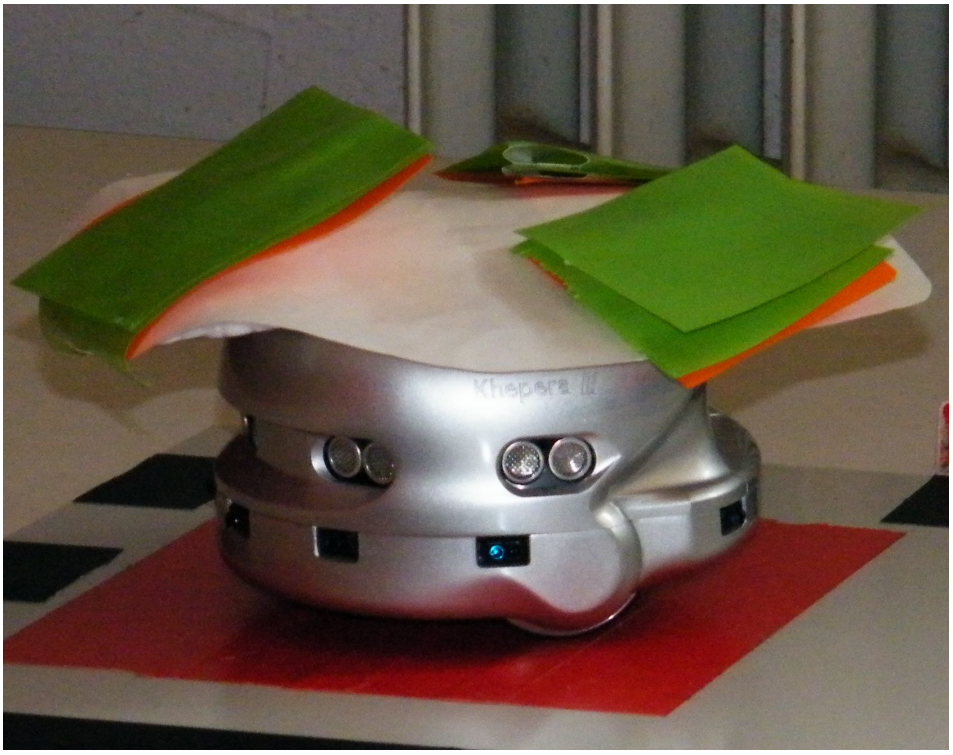


Figure 1.1: The Khepera III with its little hat for detection

In a first time, I built a module to detect the position and the orientation of the Khepera III with the help of a camera hanging from the ceiling above the maze. In a second time, I created a navigation system for the robot with the help of the position detector. In a last step, I built a module to get a virtual discretized description of a physical maze. I finally linked the detectors with the maze plant that I customized a little bit.

In the first chapter of this document, I will explain the modules built for the detectors inside of the Tapir system. In the second chapter, I will explain every classes and modules I had to build for the CL^2. Then in the third chapter, I will explain how to set up and run an experiment with the Khepera III and a maze using the tools explained in previous chapters.

2. Tapir modules

2.1 Introduction

The Tapir system is used to detect objects within images. I used it to detect the Khepera III position and orientation and to build the discretized description of the maze as well. For the maze description, you can see it as the detection of every block (and their type) of the maze seen as a grid.

I first built a detector based on the histogram detector. This detector proved to be too unstable and difficult to manage when ambient light change. Because of this, I developed a simple color detector that proved to be more stable and manageable in this environment.

Then, I developed a simple colored grid detector to discretize an image and get the mean color of each block. Based on this detector, I built the maze detector that is designed to detect more specifically given colors according to the maze definition (wall color, empty space color, etc). Finally, I built a module to output multiple objects over shared memory, as I needed this for the maze output.

2.2 Orientation histogram detector

2.2.1 Introduction

I first built a detector to detect the robot position and orientation. To do so, I used a histogram detector to detect three dots and calculate the angle of the height of the triangle. In order to work properly, two points must be clearly closer to form a triangle that is near to be isosceles. (figure 2.1)

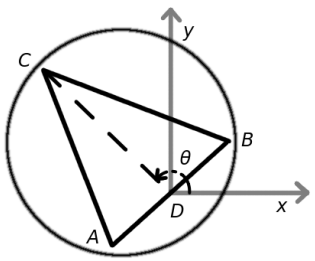
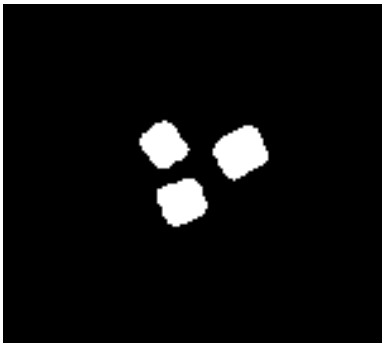


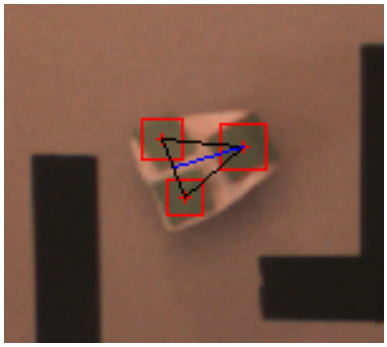
Figure 2.1: The orientation θ is the angle between the line $\overline{CD}$ and the x axis. It ranges counterclockwise from 0 to 2π.

This detector has been split and renamed by Thomas Lampe. To use a similar detector, use the HistogramDetector and the Triangle-

Figure 2.2: Detection of the position and orientation



(a) Detection of the points – The points are detected and their size are evaluted to discriminate them from noise



(b) Detection of the orientation – The points positions are evaluated and the orientation is calculated from point C to D (see figure 2.1)

Tracker. See the section 2.2.5 for more information.

2.2.2 Detection of the object and its orientation

The detector first detects blobs with the help of the histogram detector (figure 2.2a). When this is done, it iterate through the blobs and selects the three biggest ones. It then selects the two closer points and calculates the line from D to C to get the angle θ (figure 2.1), which is the orientation of the object (figure 2.2b).

The position of the object is the mean position of the three points.

2.2.3 Improvements

The detector is weak to noise. If one of the dots is not correctly detected and get smaller than some noisy blobs, the noisy blob is detected as one of the dots and the calculated position is wrong.

We could keep the three points in memory and take the closest points at the next detection. It could still be unstable if the movement is fast and noise is closer than the next position. The best option might be to weight the distances with the size of the blobs.

The detector was to unstable because of small changes in ambient light. Also, we needed to remove the Khepera from under the camera every time we restarted the detector, because the histogram should

not contain colors of the Khepera. So I decided to build a new detector that would detect specific colors. This way, the detector would be more reliable or could be more easily reconfigured during ambient light changes. Also, we could also leave the Khepera under the camera all the time.

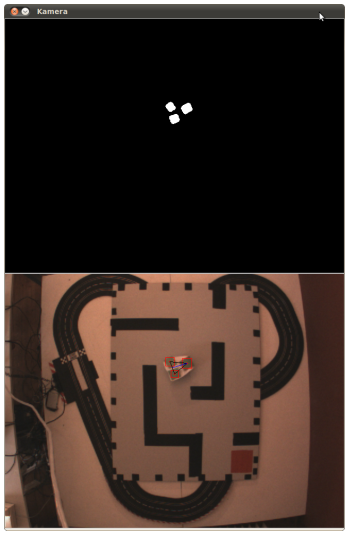


Figure 2.3: Interface example of the Orientation histogram detector – The detected points (white on black) are above and the calculated triangle and orientation are underneath

2.2.4 How to use

1. Find a color that is well detected with the detector in the desired environment. Square of colored paper seems to fare better than colored dots with pencils on white paper
2. Put 3 dots of the found color on the object you want to detect. Inclined surface reflects less strong light and thus is easier to detect.
3. Get the object out of the vision.
4. Start the detector and wait a few second until it stops adding frames to the histogram.
5. Add the object to the environment.
6. It should be detector properly. If not, try modifying the ambiant light or adding more frames with the ' ' (space) key while the robot is removed from the view.

Parameters for configuration file

examples_init
number of frames added to the histogram at the beginning

examples_renew
number of frames added to the histogram when pressing +

box
draw box around all blobs in the image

area
(min,max) define minimum and maximum size to detect objects (objects smaller than min or bigger than max are not detected)

filename
filename where to save histogram (optional)

autoload
load the histogram saved in filename at the beginning

Keys

'+'
add 'examples_renew' frames to the histogram

' ' (space)
add frames to the histogram until ' ' (space) is pressed again.

's'
save the histogram in 'filename'

'l'
load the histogram from 'filename'

2.2.5 Edit

During the refactoring of the modules to make them compatible with the new Tapir interface, Thomas Lampe have separated the detection and blob selection process. He placed the selection process in TriangleTracker, which makes more sense since it is a tracking process. The detector remains the one on which I built the orientation histogram detector; HistogramDetector. The major improvements to stabilise more the detection of the triangle will now lie in TriangleTracker rather than in HistogramDetector.

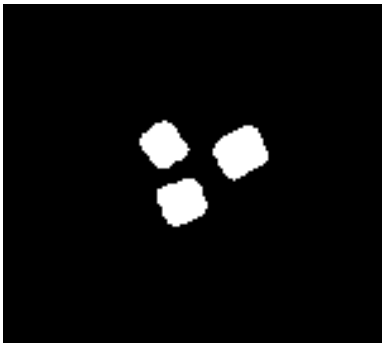
2.3 Orientation color detector

2.3.1 Introduction

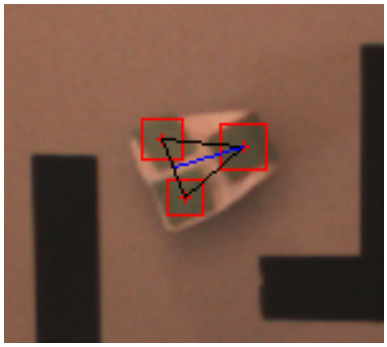
As the first detector was not stable enough, I built a second detector that detects dots within a given color range. Three dots are selected and it then calculates the angle of the height of the resulting triangle. In order to work properly, two points must be clearly closer to form a triangle that is near to be isosceles. (figure 2.4)

This detector has been split and renamed by Thomas Lampe. To

Figure 2.5: Detection of the position and orientation



(a) Detection of the points – The points are detected and their size are evaluted to discriminate them from noise



(b) Detection of the orientation – The points positions are evaluated and the orientation is calculated from point C to D (see figure 2.4)

use a similar detector, use the ColorDetector and the TriangleTracker. See the section 2.3.6 for more information.

2.3.2 Detection of the object and its orientation.

The detector enhance the colors given the saturation and brightness values. It then detects blobs using the opencv function cvInRangeS() (figure 2.5a). Given the mean color $c = (R, G, B)$ and the range $r = (R, G, B)$, it detects the colors $d = (R, G, B)$ such that $c - r < d < c + r$. Once the blobs are detected, it selects the three biggest blobs. The two closer blobs are used to calculate the line from middle point D to C (figure 2.4). The angle theta is the orientation of the object (figure 2.5b).

The position of the object is the mean position of the three points

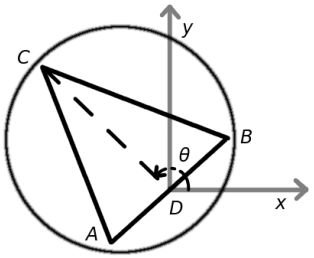


Figure 2.4: The orientation θ is the angle between the line $\overline{CD}$ and the x axis. It ranges counterclockwise from 0 to 2π.

2.3.3 Improvements

There is less noise than in the previous detector, but it is still weak to noise. If one of the dots is not correctly detected and get smaller than some noisy blobs, the noisy blob is detected as one of the dots and the calculated position is not good.

We could keep the three points in memory and take the closest points at the next detection. It could still be unstable if the movement is fast and noise is closer than the next position. The best option

may be to weight the distances with the size of the blobs.

It would be usefull to add a third frame where we show the range of selected color. Given the mean color $c = (R, G, B)$ and the range $r = (R, G, B)$, it could be a gradation from $c - r$ to $c + r$.

2.3.4 In comparison with histogram detector

The drawback compared to histogram detector is that we need to find the best color range at the beginning of every set of experiment, because the ambient light may be different from a day to another. When this is done, it is a lot more stable than histogram detector but it is still vulnerable to change in ambient light. In overall, the changes in the color range at the beginning of a new set of experiment are small or inexistent.

The big advantage compared to Histogram detector is that we do not need the get the object out of the vision every time we start again the detector.

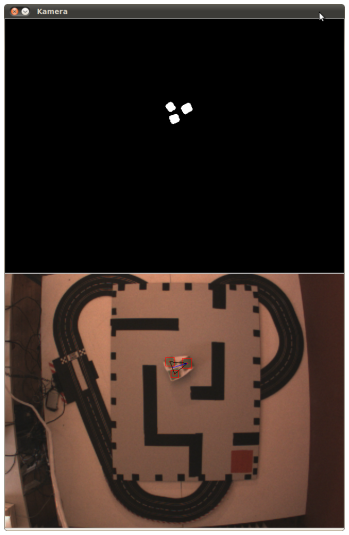


Figure 2.6: Interface example of the Orientation color detector – The detected points (white on black) are above and the calculated triangle and orientation are underneath

2.3.5 How to use

1. Before choosing a color, you may test which one is easier to detect with this detector, given the colors of the environment. Square of colored paper seems to fare better than colored dots with pencils on white paper. Take a look at the keys section 2.3.5 to modify the color and range value so you can find the best ones for you colored object.

2. Put 3 dots of the found color on the object you want to detect. Inclined surface reflects less strong light and thus is easier to detect.

3. Start the detector. The object can be already in the vision or not, it does not matter for this detector.

4. Use the keys to change the color range in order to find the best color range for a stable detection. You may have to play with it times to times due to different sun lights.

5. Print the color range to the screen, remember it and write it down in the configuration file so you do not have to modify it at each start.

Parameters for configuration file

box
bool, draw box for detected blobs (default true)

area
int int, (min,max) define minimum and maximum size to detect objects (objects smaller than min or bigger than max are not detected)

color
int int int, R G B (mean color)

wide
int int int, R G B (ranges)

saturation
int, level of saturation (default 0)

brightness
int, level of brightness (default 0)

Keys

'4' R+ 1

'r' R- 1

'5' G+ 1

't' G- 1

'6' B+ 1

'y' B- 1

'7' Rrange+ 1

'u' Rrange- 1

'8' Grange+ 1

'i' Grange- 1

'9' Brange+ 1

'o' Brange- 1

'b' change brightness mode

upkey Brightness+ 1

downkey Brightness- 1

's' change saturation mode

upkey Saturation+ 1

downkey Saturation- 1

'p' Print color/range or brightness or saturation regarding the current mode

'e' Get out of brightness/saturation mode (to color/range mode)

2.3.6 Edit

During the refactoring of the modules to make them compatible with the new Tapir interface, Thomas Lampe have separated the detection and blob selection process. He placed the selection process in TriangleTracker, which makes more sense since it is a tracking process. The detector was renamed ColorDetector. The major improvements to stabilise more the detection of the triangle will now lie in the TriangleTracker rather than in the ColorDetector.

2.4 Colored grid detector

2.4.1 Introduction

I needed to get a discretized description of the environment. To do so, I built a detector that discretizes the image and calculates the mean color of every block of the grid. The detector uses a given list of colors and finds for every block the closest color of this list to its mean. This can be used to assign types to blocks as in a maze, each different colors represents a different type. (walls, open space, target, etc).

2.4.2 Detection and drawing

Here is the algorithm described in chronological order.

1. The detector enhances the colors given the saturation and brightness values.
2. It calculates the mean color of every block and save it in a grid.
3. It draws the mean color as a thick border inside blocks of the upper image if print_block_color=**true**
4. It calculates the closest color from the mean block color
5. It draws the block closest color in the lower image
6. It add blocks to the detected objects :
   - x, y (upper left corner)
   - angle=0, size=color index
7. It draws the grid on the upper and lower image
8. If in get_block_color mode, it draws the mean color of a given block in the lower image.

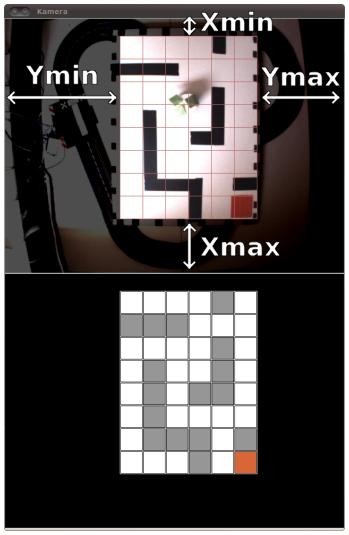


Figure 2.7: Interface of Colored grid detector – The subframe can be adjusted to a specific region of the image using the x_{min}, x_{max}, y_{min} and y_{max} values. Only this region is considered by the detector.

2.4.3 Improvements

This is not a major problem, I would even say less serious than minor, but the grid is incorrectly drawn when we change the min max values of the subframe. It does not happen on start, only on live changes. It is sometimes annoying, but it does not impact the performance of the detector, neither the usability of it.

It could be useful to change the blocks information mapping (point 6. of section 2.4.2). The color index is saved in the size attribute, but we could save the actual size of the block, the length of the side, in this attribute. Still, for rectangular blocks there would miss an attribute to save both side lengths. Even though, the color index could be saved in the angle attribute, which is an attribute that, I think, will never be useful in this context.

2.4.4 How to use

1. Set the x_{min}, x_{max}, y_{min} and y_{max} values using the arrow keys to narrow down the subframe around the environement (maze) (see figure 2.7). Be sure that the terminal has the focus. Print on terminal the values by pressing the 'p' key, write it down and save them in the configuration file.

2. Use the get_block_color mode to get the color values (RGB) of specific blocks, so you can give more accurate colors to detect in the configuration file (see figure 2.8).

3. You can adjust the brightness and saturation values to get better results.

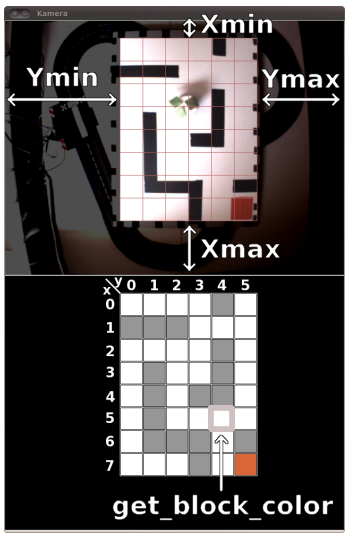


Figure 2.8: Get_object_color mode – This mode is really useful to get mean color value of a given block. The mean color is also drawn as you can see in the image at the end of the arrow.

Parameters for configuration file

grid_x
int, Number of blocks in x

grid_y
int, Number of blocks in y (default = grid_x)

x_min
int, Lower limit of the detection frame on the image

x_max
int, Upper limit of the detection frame on the image

y_min
int, Left limit of the detection frame on the image

y_max
int, Right limit of the detection frame on the image

print_block_color
bool, Draws the mean color of the block on the upper image

get_block_color
bool, Enable the get_block_color on start

get_color_x
int, Initial x position of get_block_color block

get_color_y
int, Initial y position of get_block_color block

saturation
int, Modify saturation of the image

brightness
int, Modify brightness of the image

colors
int,int,int|int,int,int|..., given colors to detect in RGB form

Keys

Press 'l', 'u', 'b', 's' or 'c' to enter a mode and 'e' to exit it. You can then use the arrow and the 'p' key for the functionality of each mode.

'l' change minimum (left,up) bounds mode
- **left** y_{min} -= 2
- **right** y_{min} += 2
- **up** x_{min} -= 2
- **down** x_{min} += 2
- **'p'** print x_{min}, x_{max}, y_{min} and y_{max}

'u' change maximum (right,down) bounds mode
- **left** y_{max} -= 2
- **right** y_{max} += 2
- **up** x_{max} -= 2
- **down** x_{max} += 2
- **'p'** print x_{min}, x_{max}, y_{min} and y_{max}

'b' change brightness mode
- **left** -5 to brightness
- **down** -5 to brightness
- **right** +5 to brightness
- **up** +5 to brightness
- **'p'** print brightness value

's' change saturation mode
- **left** -2 to saturation
- **down** -2 to saturation
- **right** +2 to saturation
- **up** +2 to saturation
- **'p'** print saturation value

'c' get_block_color mode

left	move get_block_color block to the left
right	move get_block_color block to the right
down	move get_block_color block down
up	move get_block_color block up
'p'	print get_block_color block position and comparisons (distance) with mean color of the block and the given colors

'e' exit current mode

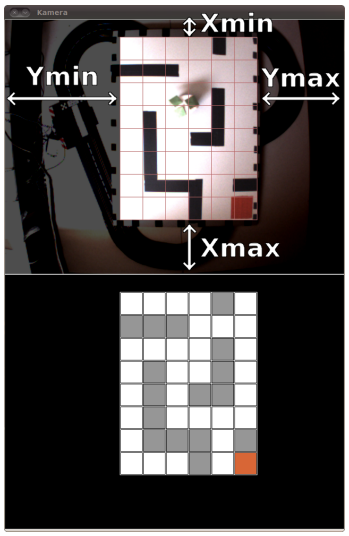


Figure 2.9: Interface of maze detector – The subframe can be adjusted to a specific region of the image using the x_{min}, x_{max}, y_{min} and y_{max} values. Only this region is considered by the detector.

2.5 Maze Detector

2.5.1 Introduction

The colored grid detector worked well to detect mean colors of the blocks, but it was too unstable in some circumstances, around the target for example. The mean color sometimes alternates quickly between two or three mean colors. To fix this, I built a subdetector that has two methods of detection. First, it detects the mean color and the closest mean color as for the Colored grid detector, but if the closest mean color is not close enough (distance bigger than max_distance_to_color) then the detected color is the closest one between the wall color (by default black) and the ground color (by default white).

2.5.2 Detection and drawing

Here is the algorithm described in chronological order.

1. The detector enhance the colors given the saturation and brightness values.
2. It calculates the mean color of every block.
3. It draws the mean color as a thick border inside blocks of the upper image if print_block_color=**true**.
4. It calculates the closest color from the mean block color.
5. Set block color:
   (a) If the closest color is closest than a given threshold, block color is set to the closest color value.

(b) Else, block color is set to closest color between wall color and ground color.

6. It draws the block selected color in the lower image
7. It adds blocks to the detected objects
   (a) x, y (upper left corner)
   (b) angle=0, size=color index (ground=0,wall=1)
8. It draws the grid on the upper and lower image
9. If in get_block_color mode, it draws the mean color of the block in the lower image (get_block_x,get_block_y).

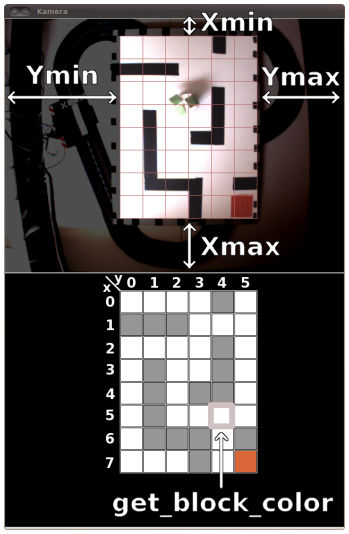


Figure 2.10: Get_object_color mode – This mode is really useful to get mean color value of a given block. The mean color is also drawn as you can see in the image at the end of the arrow.

2.5.3 Improvements

The same improvements of the colored grid detector apply to this one. See section 2.4.3 for more information.

2.5.4 How to use

1. Find colors that are easy to detect. Typically black and white for the walls and the free space, and a basic color like red for the target. An easy way to build it is with red and black tape on a white board. Ground color is assigned to index 0 and wall color to index 1, the other colors given in the configuration file start their index at 2.
2. You can build the maze before, but it is easier too build it while the detector is on, so you can keep track of how well it is detected.
3. Once the camera is placed above the maze or the empty board, set the x_{min}, x_{max}, y_{min} and y_{max} values using the arrows to narrow down the subframe around the maze (see figure 2.9). Be sure that the terminal has the focus. Print on terminal the values by pressing the 'p' key. Write it down and save the values in the configuration file.
4. Use get_block_color mode to get the color values (RGB) of the wall, the ground and the target (see figure 2.10). Wall and ground colors are black and white by default.
5. You can adjust the brightness and saturation values to get better results.

Parameters for configuration file

grid_x
int, Number of blocks in x.

grid_y
int, Number of blocks in y. (default = grid_x)

x_min
int, Lower limit of the detection frame on the image.

x_max
int, Upper limit of the detection frame on the image.

y_min
int, Left limit of the detection frame on the image

y_max
int, Right limit of the detection frame on the image

print_block_color
bool, Draws the mean color of the block on the upper image

get_block_color
bool, Enable the get_block_color on start

get_color_x
int, Initial x position of get_block_color block

get_color_y
int, Initial y position of get_block_color block

saturation
int, Modify saturation of the image

brightness
int, Modify brightness of the image

colors
int,int,int|int,int,int|..., given colors to detect in RGB form

ground
int,int,int, color of the ground (default = 255,255,255)(white)

walls
int,int,int, color of the walls (default = 0,0,0)(black)

max_distance_to_color
: int, maximal distance to color before the block mean color is evaluated as ground or wall.

Keys

Press 'l', 'u', 'b', 's' or 'c' to enter a mode and 'e' to exit it. You can then use the arrow and the 'p' key for the functionality of each mode.

'l' change minimum (left,up) bounds mode
- **left** y_{min} -= 2
- **right** y_{min} += 2
- **up** x_{min} -= 2
- **down** x_{min} += 2
- **'p'** print x_{min}, x_{max}, y_{min} and y_{max}

'u' change maximum (right,down) bounds mode
- **left** y_{max} -= 2
- **right** y_{max} += 2
- **up** x_{max} -= 2
- **down** x_{max} += 2
- **'p'** print x_{min}, x_{max}, y_{min} and y_{max}

'b' change brightness mode
- **left** -5 to brightness
- **down** -5 to brightness
- **right** +5 to brightness
- **up** +5 to brightness
- **'p'** print brightness value

's' change saturation mode
- **left** -2 to saturation
- **down** -2 to saturation
- **right** +2 to saturation
- **up** +2 to saturation
- **'p'** print saturation value

'c' get_block_color mode

left	move get_block_color block to the left
right	move get_block_color block to the right
down	move get_block_color block down
up	move get_block_color block up
'p'	print get_block_color block position and comparisons (distance) with mean color of the block and the given colors

'e' exit current mode

2.6 Multiple object shared memory

2.6.1 Introduction

The Tapir shared memory output module only share one object, but the previously build detectors, colored grid detector and maze detector need to share multiple objects at the same time.

The objects will be saved in a shared memory segment at a given index (key). This address (key) can be used in another program to fetch the information.

2.6.2 Allocating and saving shared memory segment

Here is the algorithm described in chronological order.

1. It allocates a shared memory segment with shmget during initialization.
2. It assigns an index of the shared memory segment for the list of mutexes. (see section 2.6.3)
3. It assigns an index of the shared memory segment for the list of objects. (see section 2.6.3)
4. During send (set value in memory), for each object_i in objects:
   (a) It locks the mutex_i of the object_i.
   (b) Saves the object_i.
   (c) Releases the mutex_i.

2.6.3 Memory structure

On the machine I used, sizeof(mutex) was 40 bytes and sizeof(double) 8 bytes. I could have used only one mutex for all objects, but it would have slowed down the process. In overall, I never needed to share more than 100 objects. For every objects, there is 5 double, for the slower version it would take:

$$100 * 5 * 8Byte + 40Byte \simeq 4000Byte = 4KB$$

For the faster version, we add a mutex for every object, thus it takes:

$$100 * (5 * 8Byte + 40Byte) = 8000Byte = 8KB$$

It takes 2 times more space, but 8KB is still really small, so it is not a problem. Faster is better.

n = number of objects

Indices	**Items**	**Attributes**
0	mutex_1	mutex
$\vdots$	$\vdots$	$\vdots$
n	mutex_n	mutex
$n+1$	object_1	x
$n+2$	object_1	y
$n+3$	object_1	size
$n+4$	object_1	angle
$n+5$	object_1	runtime
$\vdots$	$\vdots$	$\vdots$
$n+(n-1)*5+1=(n-1)*6+2$	object_n	x
$n+(n-1)*5+2=(n-1)*6+3$	object_n	y
$n+(n-1)*5+3=(n-1)*6+4$	object_n	size
$n+(n-1)*5+4=(n-1)*6+5$	object_n	angle
$n+(n-1)*5+5=6*n$	object_n	runtime

2.6.4 How to use

Define the number of object that needs to be send and write it in the configuration file. Define a unique key_id and key_path combination that will be used in the tapir detector in CLsquare.

Parameters for configuration file

key_path pathname for the key of shared memory segment (read about ftok())

key_id key for the shared memory segment (read about ftok())

nb_objects the number of objects to be send

3. CL^2 modules and Maze-Khepera plant

3.1 Introduction

The maze plant inside CL^2 already procure a virtual environment to solve the maze problem. It was straightforward to customize it in order to command the Khepera III throughout a maze.

Once the detectors were built, I needed to pass the information from the detector to the plant. I built a wrapper around the maze data class to read the maze structure from the detector rather than from a static file.

In order to pilot the Khepera in the maze, I built a navigation class that takes x_i, y_i coordinates in parameters and guides the robot with the help of an orientation detector.

3.2 Multiple object shared memory (class)

3.2.1 Introduction

The Tapir detector sends multiple objects for the maze detection. I needed to build a TapirDetector class to receive those objects from shared memory.

The saved objects will be read at the memory address given by the pathname and the keyid.

3.2.2 Connecting to and reading shared memory segment

Here is the algorithm described in chronological order.

1. It connects to a shared memory segment with shmget.
2. It assigns an index of the shared memory segment for the list of mutexes. (see section 3.2.3)
3. It assigns an index of the shared memory segment for the list of objects. (see section 3.2.3)
4. During update (read value in memory), for each $\_mboxobject_i$ in objects:
   (a) It locks the mutex_i of the object_i.
   (b) Reads the object_i.
   (c) Releases the mutex_i.

3.2.3 Memory structure

For more information, see section 2.6.3

n = number of objects

Indices	**Items**	**Attributes**
0	mutex_1	mutex
$\vdots$	$\vdots$	$\vdots$
n	mutex_n	mutex
$n+1$	object_1	x
$n+2$	object_1	y
$n+3$	object_1	size
$n+4$	object_1	angle
$n+5$	object_1	runtime
$\vdots$	$\vdots$	$\vdots$
$n+(n-1)*5+1=(n-1)*6+2$	object_n	x
$n+(n-1)*5+2=(n-1)*6+3$	object_n	y
$n+(n-1)*5+3=(n-1)*6+4$	object_n	size
$n+(n-1)*5+4=(n-1)*6+5$	object_n	angle
$n+(n-1)*5+5=6*n$	object_n	runtime

3.2.4 How to use

Set the number of objects, key_path and key_id as set in the configuration file in tapir.

Parameters for configuration file

key_path pathname for the key of shared memory segment (read about ftok())

key_id key for the shared memory segment (read about ftok())

nb_objects the number of objects to be read

3.3 Maze-Khepera data (class)

3.3.1 Introduction

From the detector, we have a discrete description of the maze, but it is still unreadable for the maze plant. To make use if it, I wrapped the MazeData class that defines the maze structure used by the maze plant. The wrapper takes in parameter the Tapirdetector, reads the blocks from the detector and writes a maze definition file that MazaData can read.

3.3.2 Convert detector information to maze data

Here is the algorithm described in chronological order.

1. It first get the maze description from TapirDetector.
2. It calculates the maze boundaries (x_{min}, x_{max}, y_{min}, y_{max}, x_{extend} and y_{extend}):
   (a) Min and max are calculated regarding minimum and maximum positions of the blocks. (Do not forget that the detector can detect a subframe of the image, thus min and max values are important)
   (b) Since we have upper-left position of the blocks (see point 6. of section 2.4.2), x_{min} and y_{min} correspond to exact x_{min} and y_{min} values but $\tilde{x}_{max}$ and $\tilde{y}_{max}$ are the upper-left position of the max block. Thus, real x_{max} and y_{max} are $\tilde{x}_{max} + gridwidth_x$ and $\tilde{y}_{max} + gridwidth_y$.
   (c) x_{extend}, $y_{extend} = grid_x$, $grid_y$ (number of different x and y positions)
3. It builds a grid to save the maze info.

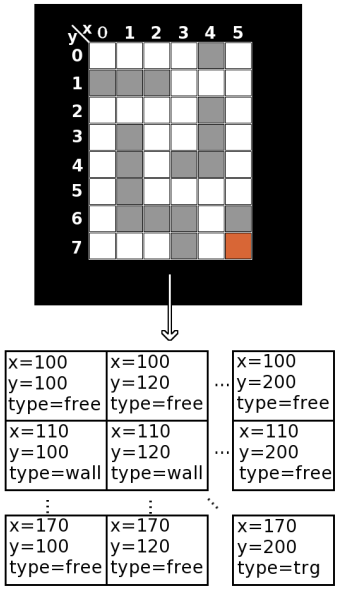


Figure 3.1: Structure of the list of detected objects – Each block of the grid is represented as a blob with the attributes x, y and *type* which are mapped to the attributes x, y, *size* of the opencv blob object. x and y are left-upper values on the image, noted as x_i, y_i, not to confuse with the grid values x_g, y_g. The objects are listed from left to right, from top to bottom.

4. It loops through the blocks and save the type information in the grid.

5. Once the grid is build, it saves it in a file as a maze definition.

6. MazeData is constructed using the saved file maze.

3.3.3 Improvements

The maze is saved in a grid before writing it in a file. We could save the blocks directly from the list to the file without making a grid, but this way we might not be able to save it serially from the list to the file. The objects might not be in the order needed to print them in the file. As it is right now, the objects are already in-order when they are send from the MazeDetector, so we could save it directly in a file without saving it in a grid. But by doing this, an in-order list of objects would be a prerequisite. By saving the blocks in a grid to be sure that we have the blocks in-order, we could, for whatever reason, send objects in random order and it would work as well.

We could have transferred the data directly to MazeData without saving it in a maze definition file. We could have inherited from MazeData to do so. But we would need to modify some part of MazeData as they are not private but protected. Also and most of all, MazeReward and MazeGraphic rely on the maze definition file to build their maze definition. They could use the new MazeKheperaData, but it is better if we can avoid modifying more modules.

Whenever someone wants to modify the mapping of the block information (x=x,y=y,size=color index) or add new types for blocks (trap for example), the lines to modify are in the loop of the save_maze() function. See the improvements subsection 2.4.3 of colored grid detector for more information on this.

3.3.4 How to use

The maze detector must be on. You need to pass the corresponding configuration file for the detector to the class and give a file name to save the maze in the constructor; MazeKheperaData(tapir_conf_file, filename).

You can then use the functions get_RX() and get_RY() to get the (X_i,Y_i) values of the image from the (x_g, y_g) coordinates of the grid.

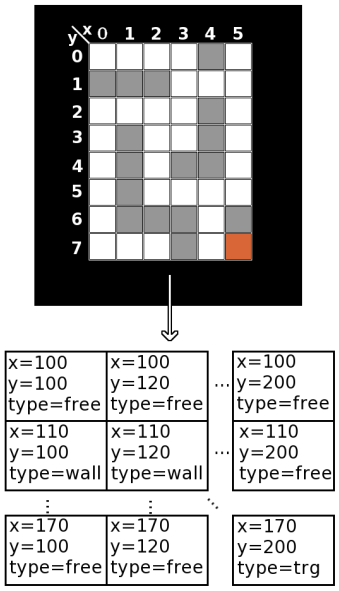


Figure 3.2: Example of min and max values –
In this example, we have:
$x_{min} = 100$
$y_{min} = 100$
$\tilde{x}_{max} = 170$
$\tilde{y}_{max} = 200$
$x_{max} = 180$
$y_{max} = 220$
$x_{extend} = 8$
$y_{extend} = 6$

Also, you can use get_centered_RX() and get_centered_RY() to get the (x_i, y_i) in the center of the block in the image from the (x_g, y_g) coordinates of the grid.

3.4 Maze reward (module)

3.4.1 Introduction

If we use the plant as a reward module, the mainloop of CLsquare try to destroy the plant 2 times at the end of the experiment. It tries to destroy the plant, and latter the reward module, but the reward module is the same as the plant, so the program crash because the plant is already destroyed.

So I decided to build a dummy reward module. It returns reward 1 whatever are the state and the action. It loads the maze from maze.def to know where is the goal so it can evaluate the function is_terminal(current_state).

3.4.2 How to use

Be sure that the maze definition file is named maze.def. The MazeKhepera plant should do this by default. There is nothing to do other than this.

3.5 Directed Khepera (class)

3.5.1 Introduction

This class aims to pilot the Khepera III to a given (x, y) position. To do so, we get the position and orientation of the Khepera from a camera hanging from the ceiling. It needs to calculate the needed orientation to drive to a (x, y) position and sends a command to the Khepera via a serial port to make it turn or drive forward.

3.5.2 Communication

It first opens a port for serial port communication. The serial port communicates via Bluetooth. In Ubuntu, it is usually /dev/rfcomm0.

To write commands to the buffer, it is imperative to add a carriage '\r' at the end. If not, the Khepera will not recognize the command.

The buffer is read char by char. The robot always answers to commands. The buffer is emptied after every command that is sent.

3.5.3 Navigation

1. It first get the position of the robot including the orientation.
2. If the detection is not successful and position is unknown, it sends a command to immobilize the robot
3. It then calculates the desired angle $\varphi = \text{atan2}(y_t - y_{target}, x_t - x_{target})$
4. It add π to the calculated angle and the robot orientation to get range $(0, 2\pi)$ rather than $(-\pi, \pi)$
5. Angle error $\alpha = \varphi - \theta$
6. If $|\alpha| < \epsilon_\theta$ (if φ fall in green region in figure 3.3) it sends a forward command
7. If $\epsilon_\theta < |\alpha| < \pi$ (if φ fall in red region in figure 3.3) sends a turn right command
8. Else (φ fall in blue region in figure 3.3) sends a turn left command
9. Do this while $(x_t - x_{target})^2 + (y_t - y_{target})^2 > \epsilon_e^2$

ϵ_θ and ϵ_e are statically defined in the class.

The turn right and turn left commands are graded on the level of angle error. To simplify calculations, negative angle errors, are inverted by adding 2π before calculating turning speed. The function of the speed, sp, is the following, where sp_{min} and sp_{max} are respectively the minimal and maximal speed of the robot:

$$Sp = \left(\frac{1}{\pi - \epsilon_\theta}\right)(\alpha * (sp_{max} - sp_{min}) + \pi * sp_{min} - \epsilon_\theta * sp_{max})$$

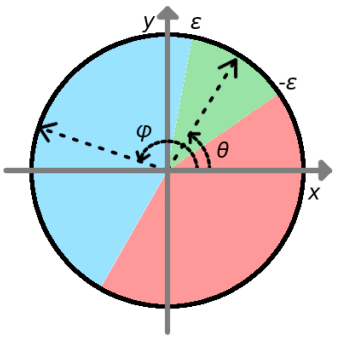


Figure 3.3: Navigation based on desired angle – The desired angle φ is the orientation needed to access the (x_{target}, y_{target}) position in straight line. θ is the current orientation of the robot. If φ falls in the blue region regarding θ, then the robot should turn left. If φ falls in the green region regarding θ, then the robot should go forward. Finally, if φ falls in the red region regarding θ, the robot should turn right.

3.5.4 Improvements

There is no command for the robot for a definite angle rotation. It would be a great enhancement for the rapidity and accuracy of the robot movements if we could find a way to make it precise turns of a given angle with a single command. There is a command to make

precise turn of the wheels, but it doesnt interpolate directly to a complete robot rotation of a precise angle. We would need to calculate how much the Khepera moves for a precise rotation of the wheels.

The update of the tapir detector and the time between read/write commands on the buffer of the serial port are really slow. The tapir detector seems to update values at a rate between 1 to 2 times per second, which is incredibly slow. It would be really helpful to speed up both processes to get higher accuracy in the navigation.

3.5.5 How to use

The tapir detector must be on. Set parameters of the tapir detector in khepera_tapir.cfg. To learn how to set up the Bluetooth connection, take a look at the section 4.3.

Constructor parameters

PORT port of the serialport. Usually /dev/rfcomm0 in Ubuntu.

max_speed, min_speed (optional) Be carefull if you set those parameters. The update time of the detector is slow and serial port communication too, so if the max_speed is too high, there might be overruns.

Init parameters

manipulate If true, the robot enters an infinite loop where you can send commands. Write quit to end, it then returns false and CLsquare ends.

delta_t The sleep time between each loop of the navigation algorithm. Delta_t seems to need to be equal or higher than 1.0 second because of the time that takes Tapir.update() and write/read serial port commands.

3.6 Maze-Khepera plant (module)

3.6.1 Introduction

We want to find the optimal paths through a maze and pilot the Khepera from its position to the target. To do so, the plant needs the

maze information through the Tapir detector. The action chosen by the controller will be send to the DirectedKhepera that will navigate the robot to the target point with the help of an orientation detector (Histogram detector or color detector with triangle tracker).

3.6.2 Gather information

It first loads the maze with the help of MazeKheperaData. I don't know why, but if the detector run once, then the information will still be accessible via shared memory even If the detector is down.

If the parameter with_robot is set to true, it will construct and initialize the robot. Otherwise, it will run as a classic MazePlant without Khepera. If the parameter test_robot_block is set to true, the program enters an infinite loop and the user is asked to enter x, y position corresponding to the grid positions. This can be used to debug maze block positions. This option might be useless now that the plant is working properly.

3.6.3 Pilot the Khepera

The action is chosen by the controller (as in MazePlant) and is represented by the next position (x_g, y_g) in the grid. The position is translated by MazeKheperaData to an (x_i, y_i) position in the image which correspond to the center of the grid block. This position (x_i, y_i) is send to DirectedKhepera that navigates the Khepera to this position.

3.6.4 Improvements

The most annoying thing with this plant is when we need to abort the experiment while the robot is moving. We always need to keep a second clsquare running with the option manipulate_robot=true to be able to send a stop command. It would be nice if the plant could catch an abort and send automatically a stop command to the Khepera.

3.6.5 How to use

1. Set the maze detector. Be sure that the detection is correct.
2. Start frameview if it is not already running.

3. ./CLSquare your_config_file.cls

4. If with_robot=true, stop the maze detector when asked for and start the Khepera detector.

Parameters for configuration file

with_robot Use the Khepera robot during the experiment (default false)

manipulate_robot Manipulate the Khepera with manual commands (D,l0,l0)

test_robot_blocks Send manually the Khepera to grid positions (x_g, y_g)

The manipulate_robot and test_robot_blocks parameters prevent experiments. CL^2 stay locked in the init function and quit when the infinite loop of manipulate_robot or test_robot_blocks is broken.

4. How to make an experiment

4.1 Introduction

In this chapter I will briefly explain how to set up the tools and a maze for an experiment. I will first explain how to set up the maze and the maze detector. Then, I will explain how to set up the Khepera. You might have problems with the Bluetooth connections, this is sometimes the longest part to set up. Finally, I will explain how to run experiments with CL^2.

4.2 Maze

1. Find colors that are easy to detect. Typically black and white for the walls and the free space, and a basic color like red for the target. An easy way to build it is with red and black tape on a white board.

2. You can build the maze before, but it is easier to build it while the detector is on, so you can keep track of how well it is detected.

3. Once the camera is placed above the maze, set the x_{min}, x_{max}, y_{min} and y_{max} values using the arrows with the MazeDetector to narrow down the subframe around the maze. Be sure that the terminal as the focus. Print on screen the values, write them down and save them in the configuration file.

4. Use get_block_color functions to get the color values (RGB) of the wall, the ground and the target. Wall and ground colors are black and white by default.

5. You can adjust the brightness and saturation values to get better results

6. You are now done with the maze

For more information, take a look at the sections 2.4, 2.5 and 2.6.

4.3 Khepera

1. Find a color that is easy to detect given the colors of the maze. Different textures or materials may have different effectiveness. Drawing color on a white paper seems really ineffective.
2. If you use the OrientationHistogramDetector (HistogramDetector and TriangleTracker), you will need to get the Khepera out of the camera view every time you start the detector. If you use the OrientationColorDetector (ColorDetector and TriangleTracker), you can leave the robot in the view of the detector when you start it, it is not a problem.
3. Put the Khepera on.
4. In Ubuntu, open the Bluetooth administration panel. If Khepera does not appear in the detected devices click on search. If it still does not appear, the battery may not be full enough. Connect the Khepera on the charge and try again. It should appear if it is on and plugged on the charge.
5. Click on pair (key) if not paired. The access key is 0000
6. Click on setup to setup the serial port.
7. If it works, write down the port. (/dev/rfcomm0 most of the time)
8. If it does not work. Try the following :
   `http://www.k-team.com/forum/index.php?topic=553.0`
9. To test the connection and to be able to stop the Khepera if the CL^2 experiment fails, run ./CLsquare manipulate_robot.cls
10. If the connection fails, try to use minicom as it is described here:
    `http://www.k-team.com/forum/index.php?topic=553.0`
11. Once the connection is correctly established, try moving the robot with commands D,l5000,l-5000. You can change values 5000 and -5000 for any values you want between -20000 and 20000.

12. The Khepera is now ready.

For more information, take a look at the sections 2.2, 2.3 and 3.5.

4.4 CL^2

1. Set starting positions in Maze.init.
2. Set khepera_tapir.cfg and tapir.cfg (maze).
3. Set train.cls file, you can leave the plant part empty, you do not want to train with the robot.
4. Set with_robot.cls file. Be sure that with_robot=true is in the Plant module.
5. Start the maze detector.
6. Start frameview in clsquare ./bin/frameview &
7. Start training with ./CLsquare train.cls
8. When training is done, you can start experimenting with Khepera. Before starting the experiment, be sure to have a CL^2 running with manipulate.cls so you can stop the robot if any problem arises. For example if CL^2 crash or if the robot fall out of the maze. You should always leave it running when you are experimenting with the Khepera. (D,l0,l0 to stop the robot)
9. When training is done, start the experiment with Khepera with ./CLsquare with_robot.cls
10. When asked for, stop MazeDetector and start the detector for the khepera.
11. The Khepera should now be on its way to the target.

5. Conclusion

I have explained different modules for detection. There was first two detectors for the orientation, from which the second one, the color detector, proved to be more stable. I then explained how the colored grid detector and the maze detector work in order to detect a maze made of black tape on a white board.

To use the detectors, I needed to build new classes for the Maze-Khepera plant. From the major ones, a class, the MazeKheperaData, wraps the maze data structure to enable reading from detector rather than static file. But the most important one is the DirectedKhepera class that enables to move the Khepera to specific (x_i, y_i) positions by communicating with it via Bluetooth.

The system works pretty well with no major flaws. The robot sometimes gets out of the maze but it is quite unusual. Even though, there are still lots of improvements that could be done, as the detectors sometimes get unstable and the robot navigates really slowly.

The project could be pushed farther. We could add the possibility to work with varying target, varying in intensity for a weighted reward or varying in positions. Also, we could enhance the plant and make it continuous. The Khepera would then have more flexibility for its movements and they may look more natural. There are plenty of possibilities with this setup. I hope the structure I made will be useful to build up new ones on it.

List of Figures