

Tutorial sobre Player/Stage

Práctica 2: Robótica

Técnicas de los sistemas Inteligentes

Curso 2012/213

Contenidos

- Explicación del contenido de un fichero .world.
- Conceptos básicos ficheros .cfg player.
- Ejemplos de dos programas en C++
 - Evitación de obstáculo simple
 - Campo repulsivo proporcional.

Stage: simulación basada en modelos

Fichero .world

- Una lista de modelos que describen lo que hay en la simulación
 - El entorno básico
 - Robots
 - Otros objetos (obstáculos...)

Sintaxis básica

```
define <nombre_modelo> model
(  
  ....#propiedades y valores  
)
```

Model: clase fundamental

Objetos del mundo

- Cualquier objeto tiene una serie de propiedades básicas:
 - posición, tamaño, velocidad, color, visibilidad para otros objetos.
- Stage facilita una "clase" fundamental de la que todos los objetos heredan
 - Un objeto del mundo es siempre una instancia de la clase model
 - No tienen por qué usarse todas las propiedades disponibles
 - Tienen valores por defecto.
- Documentación http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__model.html

Model

Summary and default values

```
model
(
  pose [ 0.0 0.0 0.0 0.0 ]
  size [ 0.1 0.1 0.1 ]
  origin [ 0.0 0.0 0.0 0.0 ]
  velocity [ 0.0 0.0 0.0 0.0 ]

  color "red"
  color_rgba [ 0.0 0.0 0.0 1.0 ]
  bitmap ""
  ctrl ""

  # determine how the model appears in various sensors
  fiducial_return 0
  fiducial_key 0
  obstacle_return 1
  ranger_return 1
  blob_return 1
  laser_return LaserVisible
  gripper_return 0
  gravity_return 0
  sticky_return 0

  # GUI properties
  gui_nose 0
  gui_grid 0
  gui_outline 1
  gui_move 0 (1 if the model has no parents);

  boundary 0
  mass 10.0
  map_resolution 0.1
  say ""
  alwayson 0
)
```

Colores en
/etc/X11/rgb.txt

Estructura fichero .world

1. Inclusión de ficheros
2. Propiedades específicas de la simulación
3. Configuración ventana de usuario
4. Configuración mapa de entorno
5. Definición de objetos de simulación basada en modelos.

```
include <nombredefichero>.inc
....
quit_time 3600
resolution 0.02
....

window(
    size [ 635.000 666.000 ] # in pixels
    scale 37.481 # pixels per meter
    center [ -0.019 -0.282 ]
    rotate [ 0 0 ]
)

floorplan(
    name "cave"
    size [16.000 16.000 0.800]
    pose [0 0 0 0]
    bitmap "bitmaps/cave.png"
)

pioneer2dx
(
    name "r0"
    pose [ -7 -7 0 45 ]
    sicklaser()
    localization "gps"
    localization_origin [ 0 0 0 0 ]
)
```

Estructura fichero .world

1. Inclusión de ficheros

- Definir objetos y guardarlos en ficheros .inc
- Reutilización de objetos

fichero "sicklaser.inc"

```
define sicklaser laser
(
    # laser-specific properties
    # factory settings for LMS200
    range_max 8.0
    fov 180.0
    samples 361
    #samples 90 # still useful but much faster to compute
    # generic model properties
    color "blue"
    size [ 0.156 0.155 0.19 ]
    # dimensions from LMS200 data sheet
)
```

include sick.inc

```
....
quit_time 3600
resolution 0.02
....

window(
    size [ 635.000 666.000 ] # in pixels
    scale 37.481 # pixels per meter
    center [ -0.019 -0.282 ]
    rotate [ 0 0 ]
)

floorplan(
    name "cave"
    size [16.000 16.000 0.800]
    pose [0 0 0 0]
    bitmap "bitmaps/cave.png"
)

pioneer2dx
(
    name "r0"
    pose [ -7 -7 0 45 ]
    sicklaser()
    localization "gps"
    localization_origin [ 0 0 0 0 ]
)
```

Estructura fichero .world

2. Propiedades específicas

- En general son opcionales, valen sus valores por defecto
- Descripción en:
- http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__world.html

```
include <nombredefichero>.inc
```

```
quit_time 3600  
resolution 0.02
```

```
....
```

```
window(  
    size [ 635.000 666.000 ] # in pixels  
    scale 37.481 # pixels per meter  
    center [ -0.019 -0.282 ]  
    rotate [ 0 0 ]  
)
```

```
floorplan(  
    name "cave"  
    size [16.000 16.000 0.800]  
    pose [0 0 0 0]  
    bitmap "bitmaps/cave.png"  
)
```

```
pioneer2dx
```

```
(  
    name "r0"  
    pose [ -7 -7 0 45 ]  
    sicklaser()  
    localization "gps"  
    localization_origin [ 0 0 0 0 ]  
)
```


Estructura fichero .world

3. Configuración ventana de usuario

– **size:** tamaño de la ventana de simulación en *pixels*.

- **size [ancho alto]**

– **scale:** ¿cuántos metros corresponden a cada pixel?

- **Valor óptimo=round(W/M)**
- **W:tamaño ventana**
- **M:tamaño mapa**

– Descripción detallada en:

- http://playerstage.sourceforge.net/doc/Stage-3.2.1/group_worldgui.html

```
include <nombredefichero>.inc
....
quit_time 3600
resolution 0.02
....
```

```
window(
    size [ 635.000 666.000 ] # in pixels
    scale 37.481 # pixels per meter
    center [ -0.019 -0.282 ]
    rotate [ 0 0 ]
)
```

```
floorplan(
    name "cave"
    size [16.000 16.000 0.800]
    pose [0 0 0 0]
    bitmap "bitmaps/cave.png"
)
pioneer2dx
(
    name "r0"
    pose [ -7 -7 0 45 ]
    sicklaser()
    localization "gps"
    localization_origin [ 0 0 0 0 ]
)
```

Estructura fichero .world

- Configuración mapa de entorno
 - Es un objeto como cualquier otro de la simulación

fichero map.inc

```
define floorplan model(  
    # sombre, sensible, artistic  
    color "gray30"  
    # most maps will need a bounding box  
    boundary 1  
    gui_nose 0  
    gui_grid 0  
    gui_move 0  
    gui_outline 0  
    gripper_return 0  
    fiducial_return 0  
    laser_return 1  
)
```

fichero simple.world

```
include map.inc  
....  
quit_time 3600  
resolution 0.02  
....  
  
window(  
    size [ 635.000 666.000 ] # in pixels  
    scale 37.481 # pixels per meter  
    center [ -0.019 -0.282 ]  
    rotate [ 0 0 ]  
)  
  
floorplan(  
    name "cave"  
    size [16.000 16.000 0.800]  
    pose [0 0 0 0]  
    bitmap "bitmaps/cave.png"  
)  
  
pioneer2dx  
(  
    name "r0"  
    pose [ -7 -7 0 45 ]  
    sicklaser()  
    localization "gps"  
    localization_origin [ 0 0 0 0 ]  
)
```

Estructura fichero .world

- Configuración mapa de entorno
 - Es un objeto como cualquier otro de la simulación

fichero map.inc

```
define floorplan model(  
    # sombre, sensible, artistic  
    color "gray30"  
    # most maps will need a bounding box  
    boundary 1  
    gui_nose 0  
    gui_grid 0  
    gui_move 0  
    gui_outline 0  
    gripper_return 0  
    fiducial_return 0  
    laser_return 1  
)
```

- **color:** dice a Player/Stage qué color tienen el modelo (gris sombreado)
- **boundary:** si hay o no una caja rodeando al modelo. Si ponemos 1, evitamos que el robot se salga del mundo.
- **gui_nose:** muestra una línea indicando la orientación del modelo.
- **gui_grid:** sobrepone una retícula
- **gui_move:** si podemos hacer "drag&drop"
- **gui_outline:** si remarcamos o no el objeto.
- **<algun_sensor>_return:** si permitimos o no que un sensor reaccione al modelo.
 - fiducial_return, laser_return, obstacle_return,...
- **gripper_return:** puede o no ser cogido por una pinza.

Estructura fichero .world

- Configuración mapa de entorno
 - Es un objeto como cualquier otro de la simulación

fichero map.inc

```
define floorplan model(  
  # sombre, sensible, artistic  
  color "gray30"  
  # most maps will need a bounding box  
  boundary 1  
  gui_nose 0  
  gui_grid 0  
  gui_move 0  
  gui_outline 0  
  gripper_return 0  
  fiducial_return 0  
  laser_return 1  
)
```

fichero simple.world

```
include map.inc  
....  
quit_time 3600  
resolution 0.02  
....  
  
window(  
  size [ 635.000 666.000 ] # in pixels  
  scale 37.481 # pixels per meter  
  center [ -0.019 -0.282 ]  
  rotate [ 0 0 ]  
)  
  
floorplan(  
  name "cave"  
  size [16.000 16.000 0.800]  
  pose [0 0 0 0]  
  bitmap "bitmaps/cave.png"  
)  
  
pioneer2dx  
(  
  name "r0"  
  pose [ -7 -7 0 45 ]  
  sicklaser()  
  localization "gps"  
  localization_origin [ 0 0  
)
```

Declaramos que usamos un objeto "de clase" floorplan y con nombre "cave"

Definimos propiedades específicas de "cave"

16 m de ancho
16 m de largo
0.8 m de alto

Entre ellas, la imagen que usaremos como entorno (bmp,jpeg,gif,png)

Estructura fichero .world

5. Definición de objetos de simulación basada en modelos.

- Objetos en general
 - Según la clase "model"
- Robots
 - A partir de modelos específicos para
 - Sensores
 - Efectores (Devices en Player/Stage)

```
include <nombredefichero>.inc
....
quit_time 3600
resolution 0.02
....

window(
    size [ 635.000 666.000 ] # in pixels
    scale 37.481 # pixels per meter
    center [ -0.019 -0.282 ]
    rotate [ 0 0 ]
)

floorplan(
    name "cave"
    size [16.000 16.000 0.800]
    pose [0 0 0 0]
    bitmap "bitmaps/cave.png"
)

pioneer2dx
(
    name "r0"
    pose [ -7 -7 0 45 ]
    sicklaser()
    localization "gps"
    localization_origin [ 0 0 0 0 ]
)
```

Estructura fichero .world

5. Definición de objetos de simulación basada en modelos.

— Objetos en general

- Según la clase "model"
- Lata naranja
- Brick azul
- Poblamos el mundo

La estructura block se usa también para definir partes de cuerpos de robots

z [suelo techo]
suelo: distancia desde el suelo
altura = techo -suelo

```
define lata_naranja model
(
  bitmap "bitmaps/circle.png"
  size [0.15 0.15 0.15]
  color "orange"
)
```

Definimos una lata como un cilindro a partir de una imagen de un círculo y obligamos a que tenga el tamaño 0.15mx0.15mx0.15m

```
define brick_azul model
(
  block
  (
    points 4
    point[0] [1 0]
    point[1] [1 1]
    point[2] [0 1]
    point[3] [0 0]
    z [0 1]
  )
  size [0.1 0.2 0.2]
  color "DarkBlue"
)
```

Definimos un brick como un bloque cuadrado de lado 1 y lo "estiramos" para que se ajuste a 0.1mx0.2mx0.2m

Poblamos el mundo con objetos en distintas posiciones

```
lata_naranja(name "orange1" pose [-1 -5 0 0])
lata_naranja(name "orange2" pose [-2 -5 0 0])
lata_naranja(name "orange3" pose [-3 -5 0 0])
brick_azul(name "carton1" pose [-2 -4 0 0])
brick_azul(name "carton2" pose [-2 -3 0 0])
brick_azul(name "carton3" pose [-2 -2 0 0])
```

Modelos específicos para sensors y devices

http://playerstage.sourceforge.net/doc/Stage-3.2.1/group__model.html

Hardware de robots

- Hardware robots

- Sensores

- Rango
 - Ubicación
 - Propioceptivos
 - Otros

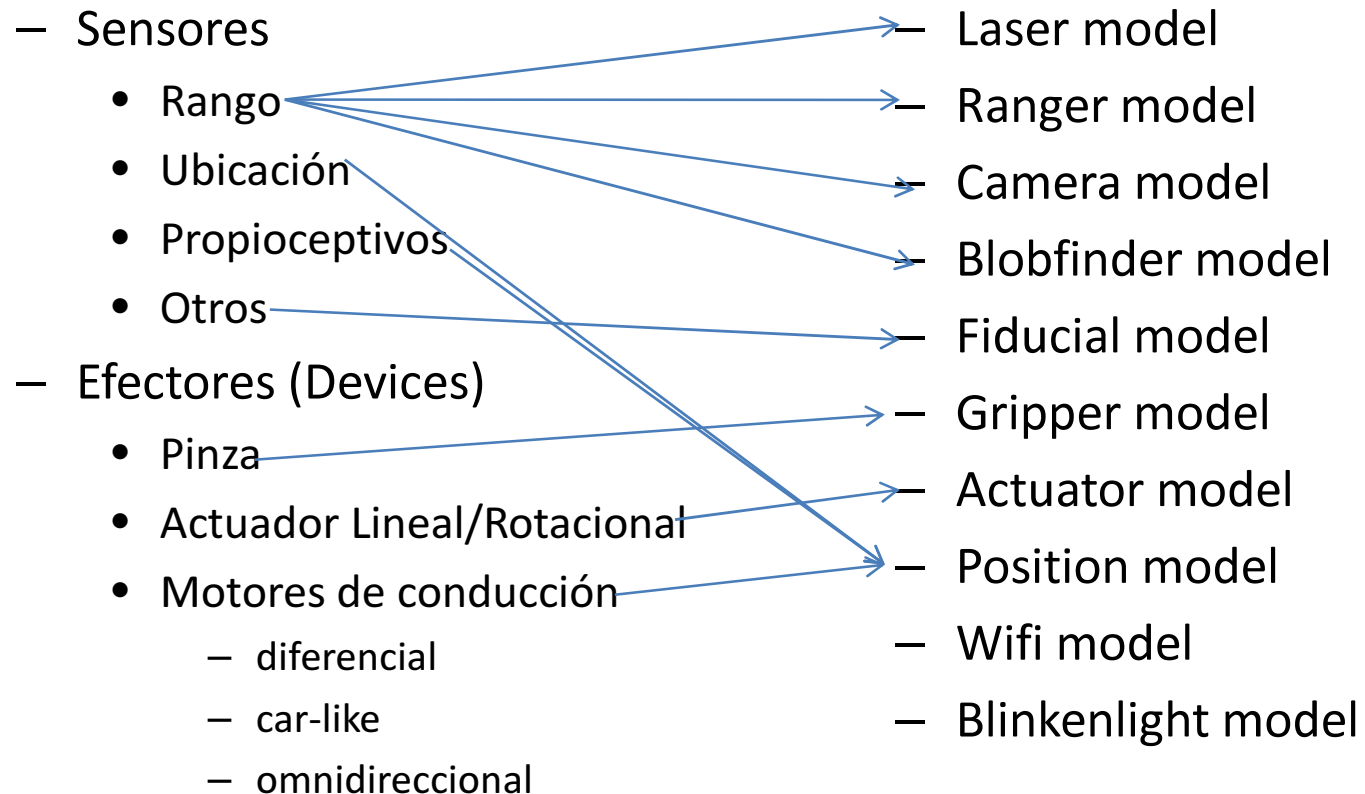
- Efectores (Devices)

- Pinza
 - Actuador Lineal/Rotacional
 - Motores de conducción
 - diferencial
 - car-like
 - omnidireccional

Modelos en Stage

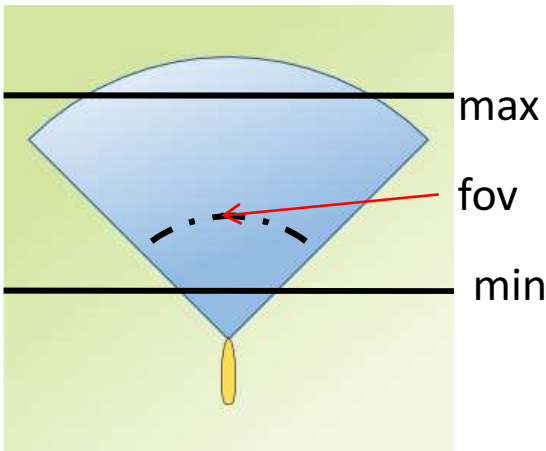
- Model

- Laser model
 - Ranger model
 - Camera model
 - Blobfinder model
 - Fiducial model
 - Gripper model
 - Actuator model
 - Position model
 - Wifi model
 - Blinkenlight model



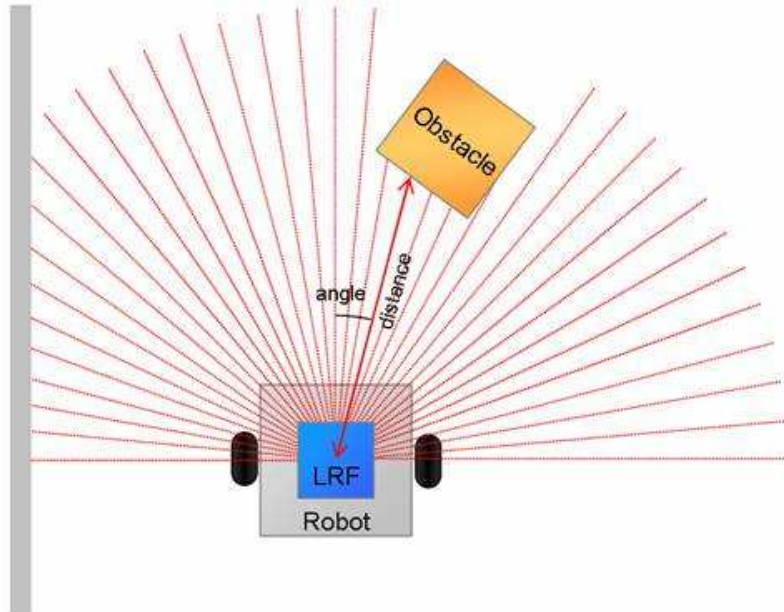
Modelo ranger

- Doc: http://playerstage.sourceforge.net/doc/Stage-3.2.1/group_model_ranger.html
 - Simula sensores de detección de obstáculos
 - Sonars, sensores infrarrojos.
 - Localiza modelos cuyo `ranger_return = 1`
 - Relación con Player: Interface "sonar"
- Parámetros más importantes:
 - **scount** <int>: número de sensores del modelo
 - **spose[ranger_number] [x y yaw]**: dice al simulador donde está cada sensor, respecto al centro geométrico del robot
 - **ssize [x y]**: tamaño de cada sensor
 - **sview [min max fov]**:
 - distancia mínima y máxima detectada en METROS y
 - campo de vision (field of view) en GRADOS



Modelo laser

- Doc:
http://playerstage.sourceforge.net/doc/Stage-3.2.1/group_model_laser.html
- Caso especial de ranger en el que sólo hay un sensor, pero con un mayor campo de visión.
- Localiza modelos cuyo laser_return = 1
- Plaver: interface "laser"



- Parámetros más importantes:
- **samples:** Número de muestras, lecturas de rango, que toma
 - Se comporta como varios sensores de rango, todos con la misma posición.
 - cada lectura tiene un "yaw", orientación, distintos.
 - Cada muestra se distribuye proporcionalmente en el ángulo de visión
 - Si el ángulo es 180° y hay 180 muestras, están separadas 1°.
- **range_max:** máxima distancia a la que puede detectar
- **fov:** campo de visión en RADIANTES

Modelo position (el más importante)

- Doc:
http://playerstage.sourceforge.net/doc/Stage-3.2.1/group_model_position.html
- Simula:
 - Sensores propioceptivos (odometría)
 - Localización
 - Modos de dirección
 - No hay ruedas, cadenas,....
- Colisiona con modelos cuyo `obstacle_return = 1`
- Player: Interface "position2d"
 - informa donde está el robot y
 - facilita comandos de movimiento.
- Parámetros más importantes:
- **drive:**
 - "diff": conducción diferencial.
 - "car": conducción modo coche
 - "omni": conducción omnidireccional
- **localization:**
 - "gps" sabe siempre con exactitud donde está
 - "odom" calcula internamente la posición con un modelo odométrico simple (sujeto a error)

¿Cómo defino un robot?

1. Siempre a partir de un model position que lo recoge todo

1. Definir el cuerpo

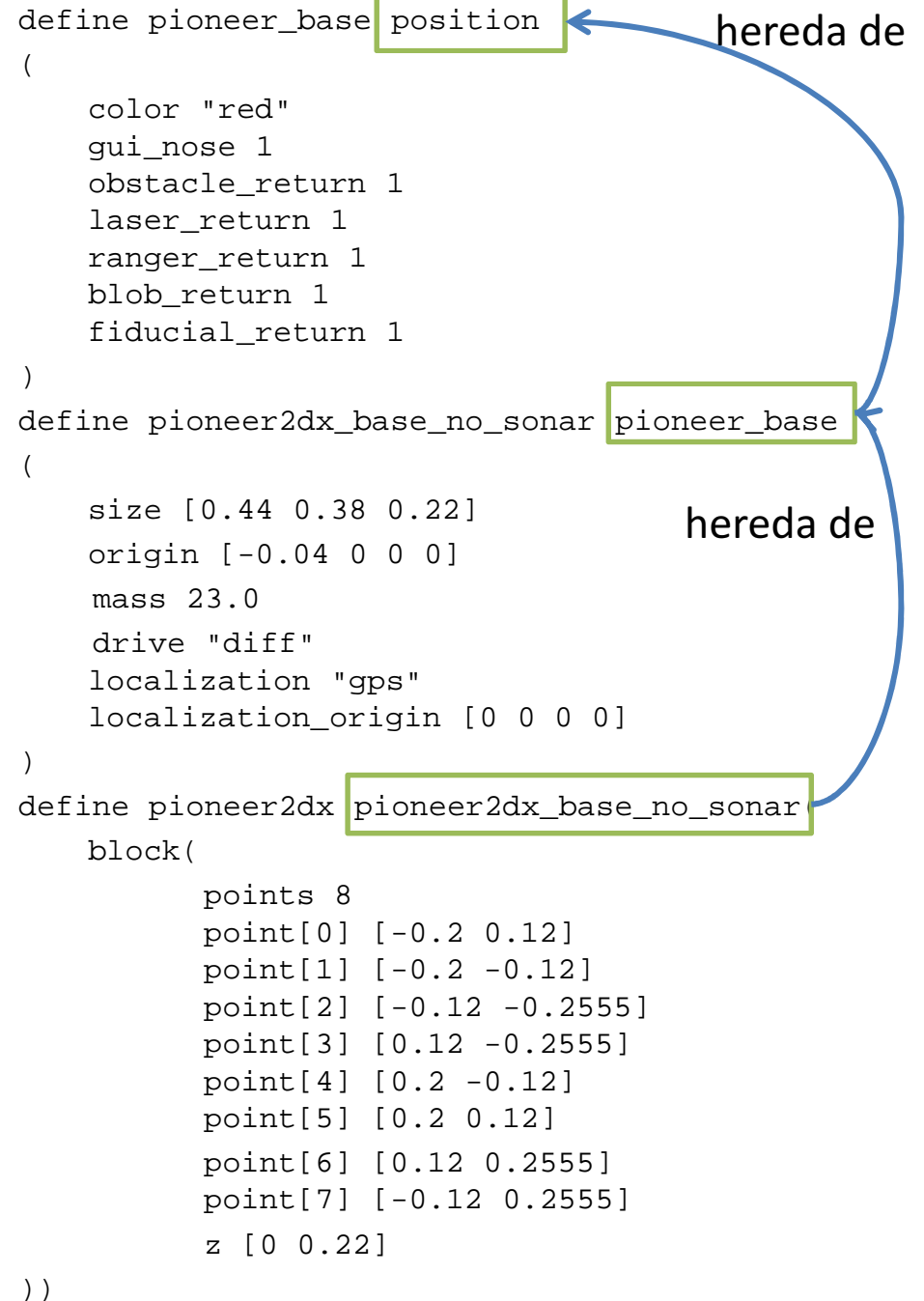
- A partir de forma, tamaño y posición

2. Definir modos de dirección y localización

3. Definir los sensores

4. Adjuntarlos al modelo "position".

```
define pioneer_base position
(
    color "red"
    gui_nose 1
    obstacle_return 1
    laser_return 1
    ranger_return 1
    blob_return 1
    fiducial_return 1
)
define pioneer2dx_base_no_sonar pioneer_base
(
    size [0.44 0.38 0.22]
    origin [-0.04 0 0 0]
    mass 23.0
    drive "diff"
    localization "gps"
    localization_origin [0 0 0 0]
)
define pioneer2dx pioneer2dx_base_no_sonar
    block(
        points 8
        point[0] [-0.2 0.12]
        point[1] [-0.2 -0.12]
        point[2] [-0.12 -0.2555]
        point[3] [0.12 -0.2555]
        point[4] [0.2 -0.12]
        point[5] [0.2 0.12]
        point[6] [0.12 0.2555]
        point[7] [-0.12 0.2555]
        z [0 0.22]
    )
)
```



fichero "ejemplo-robot-pioneer.inc"

¿Cómo defino un robot?

1. Siempre a partir de un model position que lo recoge todo

1. Definir el cuerpo

- A partir de forma, tamaño y posición

2. Definir modos de dirección y localización

3. Definir los sensores

4. Adjuntarlos al modelo "position".

```
define pioneer_base position
(
    color "red"
    gui_nose 1
    obstacle_return 1
    laser_return 1
    ranger_return 1
    blob_return 1
    fiducial_return 1
)

define pioneer2dx_base_no_sonar pioneer_base
(
    size [0.44 0.38 0.22]
    origin [-0.04 0 0 0]
    mass 23.0
    drive "diff"
    localization "gps"
    localization_origin [0 0 0 0]
)

define pioneer2dx pioneer2dx_base_no_sonar
block(
    points 8
    point[0] [-0.2 0.12]
    point[1] [-0.2 -0.12]
    point[2] [-0.12 -0.2555]
    point[3] [0.12 -0.2555]
    point[4] [0.2 -0.12]
    point[5] [0.2 0.12]
    point[6] [0.12 0.2555]
    point[7] [-0.12 0.2555]
    z [0 0.22]
)
))
```

El cuerpo es rojo, muestra su orientación (nose) pueden colisionar con él, refleja laser y ranger. Detectable por detectores de color y fiduciales.

hereda de

fichero "ejemplo-robot-pioneer.inc"

¿Cómo defino un robot?

1. Siempre a partir de un model position que lo recoge todo

1. Definir el cuerpo

- A partir de forma, tamaño y posición

2. Definir modos de dirección y localización

3. Definir los sensores

4. Adjuntarlos al modelo "position".

```
define pioneer_base position
(
    color "red"
    gui_nose 1
    obstacle_return 1
    laser_return 1
    ranger_return 1
    blob_return 1
    fiducial_return 1
)
define pioneer2dx_base_no_sonar pioneer_base
(
    size [0.44 0.38 0.22]
    origin [-0.04 0 0 0]
    mass 23.0
    drive "diff"
    localization "gps"
    localization_origin [0 0 0 0]
)
define pioneer2dx pioneer2dx_base_no_sonar
    block(
        points 8
        point[0] [-0.2 0.12]
        point[1] [-0.2 -0.12]
        point[2] [-0.12 -0.2555]
        point[3] [0.12 -0.2555]
        point[4] [0.2 -0.12]
        point[5] [0.2 0.12]
        point[6] [0.12 0.2555]
        point[7] [-0.12 0.2555]
        z [0 0.22]
    )
)
```

hereda de

El cuerpo tiene un tamaño de 44cmx38cmx22xm y su origen es "casi el centro". La conducción es diferencial y la información sobre localización es perfecta. Su posición inicial es el origen de coordenadas del mundo.

fichero "ejemplo-robot-pioneer.inc"

¿Cómo defino un robot?

1. Siempre a partir de un model position que lo recoge todo

1. Definir el cuerpo

- A partir de forma, tamaño y posición

2. Definir modos de dirección y localización

3. Definir los sensores

4. Adjuntarlos al modelo "position".

```
define pioneer_base position
(
    color "red"
    gui_nose 1
    obstacle_return 1
    laser_return 1
    ranger_return 1
    blob_return 1
    fiducial_return 1
)
define pioneer2dx_base_no_sonar pioneer_base
(
    size [0.44 0.38 0.22]
    origin [-0.04 0 0 0]
    mass 23.0
    drive "diff"
    localization "gps"
    localization_origin [0 0 0 0]
)
define pioneer2dx pioneer2dx_base_no_sonar
    block(
        points 8
        point[0] [-0.2 0.12]
        point[1] [-0.2 -0.12]
        point[2] [-0.12 -0.2555]
        point[3] [0.12 -0.2555]
        point[4] [0.2 -0.12]
        point[5] [0.2 0.12]
        point[6] [0.12 0.2555]
        point[7] [-0.12 0.2555]
        z [0 0.22]
    )
)
```

hereda de

hereda de

El cuerpo es un octógono de 0.22m de alto

fichero "ejemplo-robot-pioneer.inc"

¿Cómo defino un robot?

1. Siempre a partir de un model position que lo recoge todo

1. Definir el cuerpo

- A partir de forma, tamaño y posición

2. Definir modos de dirección y localización

3. Definir los sensores

4. Adjuntarlos al modelo "position"

1. o a un modelo que hereda de "position".

fichero "sicklaser.inc"

```
define sicklaser laser
(
  range_max 8.0
  fov 180.0
  samples 361
  color "blue"
  size [ 0.156 0.155 0.19 ]
)
```

```
include sicklaser.inc
include ejemplo-robot-pioneer.inc
....
....
```

```
window(....)
```

```
floorplan(...)
```

```
define mi_ranger ranger (....)
define mi_camara camera (....)
pioneer2dx
```

```
(
  name "r0"
  pose [ -7 -7 0 45 ]
  sicklaser()
  mi_ranger()
  mi_camara()
)
```

```
localization "gps"
localization_origin [ 0 0 0 0 ]
```

Podemos "machacar" propiedades para esta instancia concreta

¿Cómo defino varios robots?

1. Por ejemplo, declarando varios objetos del mismo tipo.
 1. Pero con nombre y posiciones distintas

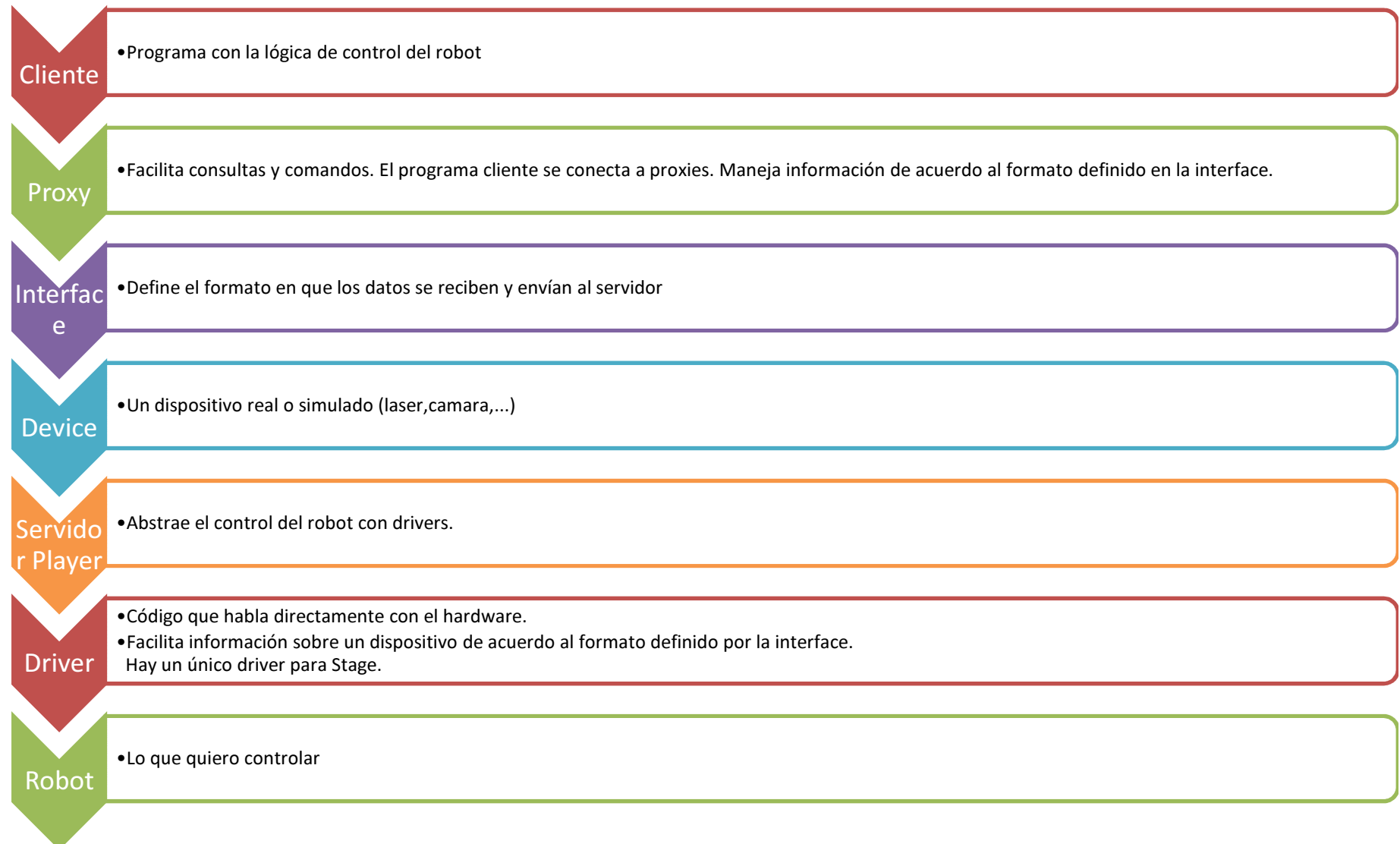
fichero "sicklaser.inc"

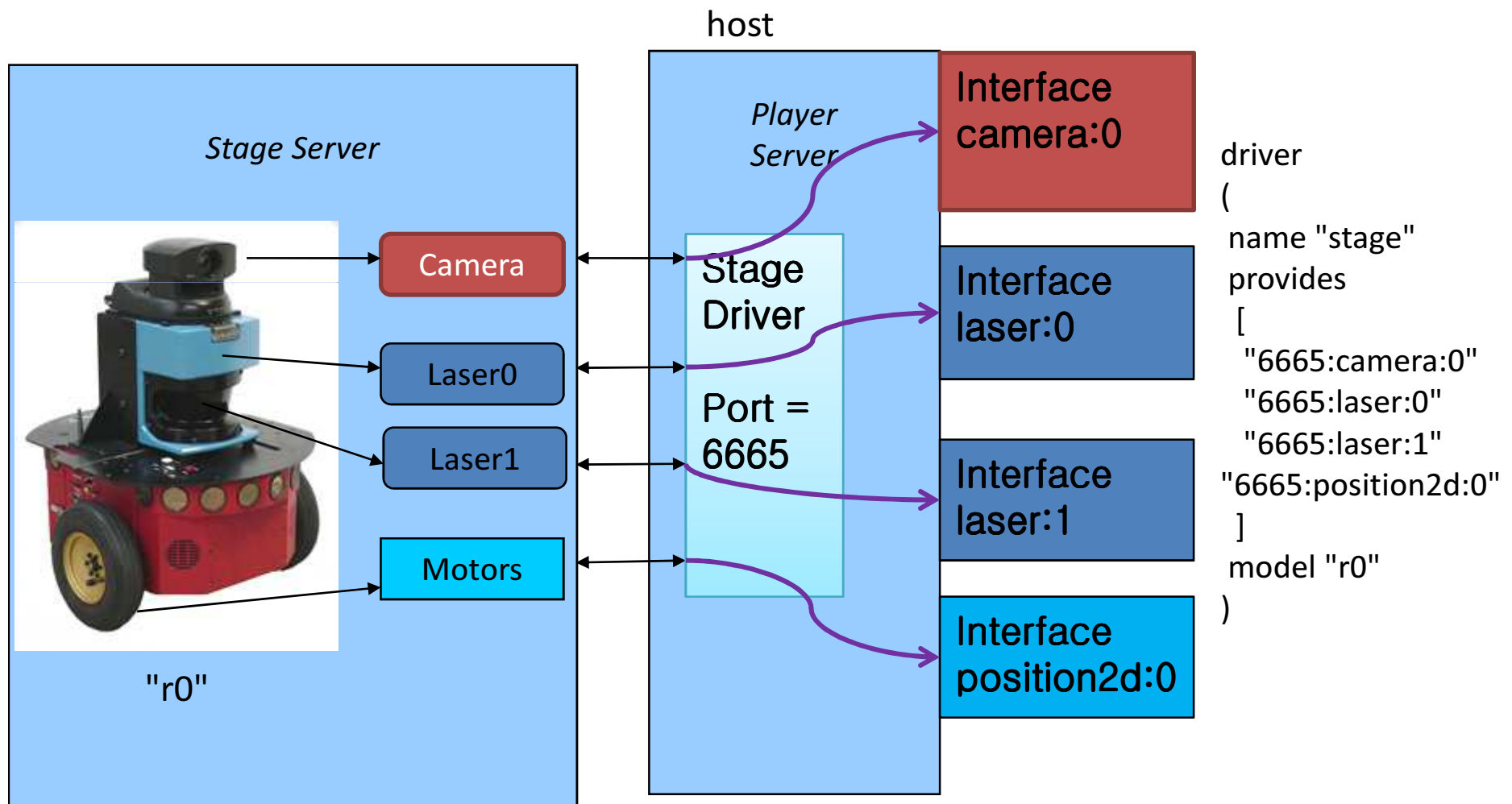
```
define sicklaser laser
(
    range_max 8.0
    fov 180.0
    samples 361
    color "blue"
    size [ 0.156 0.155 0.19 ]
)

(...)
pioneer2dx
(
    name "r0"
    pose [ -7 -7 0 45 ]
    sicklaser()
    localization "gps"
    localization_origin [ 0 0 0 0 ]
)

pioneer2dx
(
    name "r1"
    pose [ 7 7 0 45 ]
    sicklaser()
    localization "gps"
    localization_origin [ 0 0 0 0 ]
)
```


Player: drivers, interfaces, devices, proxies.





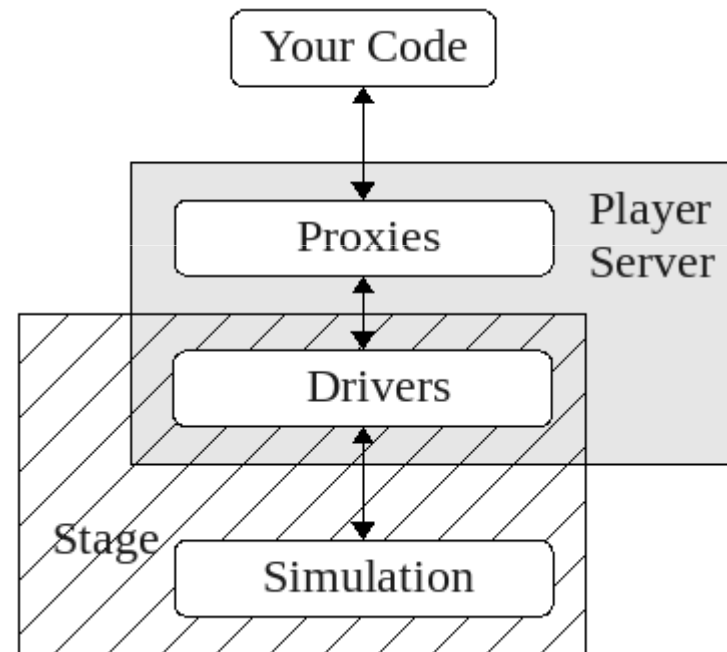
Fichero .cfg

- Siempre hay que:
 - Definir un driver con el que se informa a Player
 - Que se va a usar la simulación con stage
 - El fichero de simulación
- Y después:
 - Definir un driver por cada robot en el mundo.

```
driver(  
  name "stage"  
  provides [ "simulation:0" ]  
  plugin "stageplugin"  
  worldfile "ejemplo-simple.world"  
  
)  
# Create a Stage driver and attach  
# position2d and laser interfaces  
# to the model "r0"  
driver(  
  name "stage"  
  provides [  
    "6665:position2d:0"  
    "6665:laser:0"  
  ]  
  model "r0" )
```

Escribir un programa de control

- Proxy:
 - Una clase que permite
 - conectarse con Player,
 - pedir o enviar información a un driver con



Información sobre proxies.

- Doc de los proxies:
 - http://playerstage.sourceforge.net/doc/Player-3.0.2/player/classPlayerCc_1_1ClientProxy.html
 - Buscar el proxy que se corresponda con el dispositivo que queremos manejar
 - Por ejemplo:
 - Si usamos dispositivo Position2d, buscar la documentación sobre Position2dProxy
 - Si usamos Laser, buscar LaserProxy
 - etc...

Un programa ejemplo (evita-obstaculos-simple.cc)

```
4
5 #include <iostream>
6 #include <libplayerc++/playerc++.h>
7
8 using namespace PlayerCc;
9
10
11 int main(int argc, char *argv[])
12 {
13
14     PlayerClient    robot("localhost");
15     LaserProxy      lp(&robot,0);
16     Position2dProxy pp(&robot,0);
17
18
19     lp.RequestGeom();
20
21     for(;;)
22     {
23         double turnrate, speed;
24
25         // Lee información de los proxies
26         robot.Read();
27
28         // muestra en pantalla la muestra central del l
29         std::cout << lp[180] << std::endl;
30
31         // evitación de obstáculos simples
32         if (lp.GetMinLeft() < lp.GetMinRight()) //si el
33             turnrate = dtor(-10); // gira 10 grados por s
34         else
35             turnrate = dtor(10); //gira 10 grados por seg
36
37         if(lp[180] < 0.500) //si delante hay un obstácu
38             speed = 0; //pone la velocidad a 0
39         else
40             speed = 0.100; //pone velocidad de crucero
41
42         // envia comando a los motores.
43         pp.SetSpeed(speed, turnrate);
44     }
45 }
46
```

Siempre Incluir esta librería.

Doc en:

http://playerstage.sourceforge.net/doc/Player-3.0.2/player/group_player_clientlib_cplusplus.html

Para evitar conflictos de nombres.

Un programa ejemplo (evita-obstaculos-simple.cc)

```
4
5  #include <iostream>
6  #include <libplayerc++/playerc++.h>
7
8  using namespace PlayerCc;
9
10
11  int main(int argc, char *argv[])
12  {
13
14      PlayerClient  robot("localhost");
15      LaserProxy    lp(&robot,0);
16      Position2dProxy pp(&robot,0);
17
18
19      lp.RequestGeom();
20
21      for(;;)
22      {
23          double turnrate, speed;
24
25          // Lee información de los proxies
26          robot.Read();
27
28          // muestra en pantalla la muestra central del l
29          std::cout << lp[180] << std::endl;
30
31          // evitación de obstáculos simples
32          if (lp.GetMinLeft() < lp.GetMinRight()) //si el
33              turnrate = dtor(-10); // gira 10 grados por s
34          else
35              turnrate = dtor(10); //gira 10 grados por seg
36
37          if(lp[180] < 0.500) //si delante hay un obstácu
38              speed = 0;      //pone la velocidad a 0
39          else
40              speed = 0.100;  //pone velocidad de crucero
41
42          // envia comando a los motores.
43          pp.SetSpeed(speed, turnrate);
44      }
45  }
```

- Lo primero: las conexiones
 - Crea un objeto "robot" que se conecta automáticamente con el host "localhost" donde estará ejecutándose Player.
- Puede usarse
 - PlayerClient robot1("localhost",6665)
 - PlayerClient robot2("localhost", 6666)
 -

Un programa ejemplo (evita-obstaculos-simple.cc)

```
4
5  #include <iostream>
6  #include <libplayerc++/playerc++.h>
7
8  using namespace PlayerCc;
9
10
11  int main(int argc, char *argv[])
12  {
13
14      PlayerClient  robot("localhost");
15      LaserProxy    lp(&robot,0);
16      Position2dProxy pp(&robot,0);
17
18
19      lp.RequestGeom();
20
21      for(;;)
22      {
23          double turnrate, speed;
24
25          // Lee información de los proxies
26          robot.Read();
27
28          // muestra en pantalla la muestra central del 1
29          std::cout << lp[180] << std::endl;
30
31          // evitación de obstáculos simples
32          if (lp.GetMinLeft() < lp.GetMinRight()) //si el
33              turnrate = dtor(-10); // gira 10 grados por s
34          else
35              turnrate = dtor(10); //gira 10 grados por seg
36
37          if(lp[180] < 0.500) //si delante hay un obstácu
38              speed = 0;      //pone la velocidad a 0
39          else
40              speed = 0.100;  //pone velocidad de crucero
41
42          // envia comando a los motores.
43          pp.SetSpeed(speed, turnrate);
44      }
45  }
```

- Crea un objeto "lp" de la clase LaserProxy.
- las características de este objeto están definidas en el fichero sicklaser.inc.
- Se conecta directamente a laser:0.
- Recibe como argumento un puntero a un robot, es decir, a un PlayerClient.
- Si hubiera más índices, como
 - laser:1
 - laser:2
- Poner
 - LaserProxy lp1(&robot,1)
- Lo mismo para Postion2dProxy
- las características de pp están definidas en el fichero ejemplo-pioneer.inc

Un programa ejemplo (evita-obstaculos-simple.cc)

```
4
5  #include <iostream>
6  #include <libplayerc++/playerc++.h>
7
8  using namespace PlayerCc;
9
10
11  int main(int argc, char *argv[])
12  {
13
14      PlayerClient    robot("localhost");
15      LaserProxy      lp(&robot,0);
16      Position2dProxy pp(&robot,0);
17
18      lp.RequestGeom();
19
20      for(;;)
21      {
22          double turnrate, speed;
23
24          // Lee información de los proxies
25          robot.Read();
26
27          // muestra en pantalla la muestra central del l
28          std::cout << lp[180] << std::endl;
29
30          // evitación de obstáculos simples
31          if (lp.GetMinLeft() < lp.GetMinRight()) //si el
32              turnrate = dtor(-10); // gira 10 grados por s
33          else
34              turnrate = dtor(10); //gira 10 grados por seg
35
36          if(lp[180] < 0.500) //si delante hay un obstácu
37              speed = 0;      //pone la velocidad a 0
38          else
39              speed = 0.100;  //pone velocidad de crucero
40
41          // envia comando a los motores.
42          pp.SetSpeed(speed, turnrate);
43      }
44  }
45
46
```

- Llama al método RequestGeom() del objeto lp.
- Sirve para poder actualizar internamente, es decir, "traer" desde el servidor, toda la información sobre la geometría del laser (que se definió en el fichero sick-laser.inc)
- Es conveniente que, para cada dispositivo que se controle, las primeras líneas de código "soliciten" la geometría.
- Podría usarse también pp.RequestGeom()

Un programa ejemplo (evita-obstaculos-simple.cc)

```
4
5  #include <iostream>
6  #include <libplayerc++/playerc++.h>
7
8  using namespace PlayerCc;
9
10
11  int main(int argc, char *argv[])
12  {
13
14      PlayerClient    robot("localhost");
15      LaserProxy      lp(&robot,0);
16      Position2dProxy pp(&robot,0);
17
18
19      lp.RequestGeom();
20
21      for(;;)
22      {
23          double turnrate, speed;
24
25          // Lee información de los proxys
26          robot.Read();
27
28          // muestra en pantalla la muestra central del 1
29          std::cout << lp[180] << std::endl;
30
31          // evitación de obstáculos simples
32          if (lp.GetMinLeft() < lp.GetMinRight()) //si el
33              turnrate = dtor(-10); // gira 10 grados por s
34          else
35              turnrate = dtor(10); //gira 10 grados por seg
36
37          if(lp[180] < 0.500) //si delante hay un obstácu
38              speed = 0;      //pone la velocidad a 0
39          else
40              speed = 0.100;  //pone velocidad de cruce
41
42          // envia comando a los motores.
43          pp.SetSpeed(speed, turnrate);
44      }
45  }
```

•Ciclo de control fundamental de **todo programa** escrito para Player, basado en el ya conocido

- Perceive
- Reason
- Act.

•**Lectura** de toda la información proporcionada por sensores o cualquier otro dispositivo.

•**La función Read es fundamental.** Observar que se aplica al robot, no a dispositivo. Es la primera función que tiene que haber en el ciclo.

•**Lógica de control (Reasoning).**

- Su objetivo es determinar un valor para **speed** y **turnrate**.
- Observar que estamos en un robot de control diferencial y esta es la manera de controlar los motores.

•**Actuación**

- Envía al robot una orden, mediante el proxy tipo Position2dProxy:
 - Velocidad lineal = **speed**
 - Velocidad angular = **turnrate**

Un programa ejemplo (evita-obstaculos-simple.cc)

```
4
5 #include <iostream>
6 #include <libplayerc++/playerc++.h>
7
8 using namespace PlayerCc;
9
10
11 int main(int argc, char *argv[])
12 {
13
14     PlayerClient    robot("localhost");
15     LaserProxy      lp(&robot,0);
16     Position2dProxy pp(&robot,0);
17
18
19     lp.RequestGeom();
20
21     for(;;)
22     {
23         double turnrate, speed;
24
25         // Lee información de los proximos
26         robot.Read();
27
28         // muestra en pantalla la muestra central del 1
29         std::cout << lp[180] << std::endl;
30
31         // evitación de obstáculos simples
32         if (lp.GetMinLeft() < lp.GetMinRight()) //si el
33             turnrate = dtor(-10); // gira 10 grados por s
34         else
35             turnrate = dtor(10); //gira 10 grados por seg
36
37         if(lp[180] < 0.500) //si delante hay un obstácu
38             speed = 0; //pone la velocidad a 0
39         else
40             speed = 0.100; //pone velocidad de cruce
41
42         // envia comando a los motores.
43         pp.SetSpeed(speed, turnrate);
44     }
45 }
46
```

•¿Qué hace el programa?

•Es un programa para evitar obstáculos con una técnica muy simple.

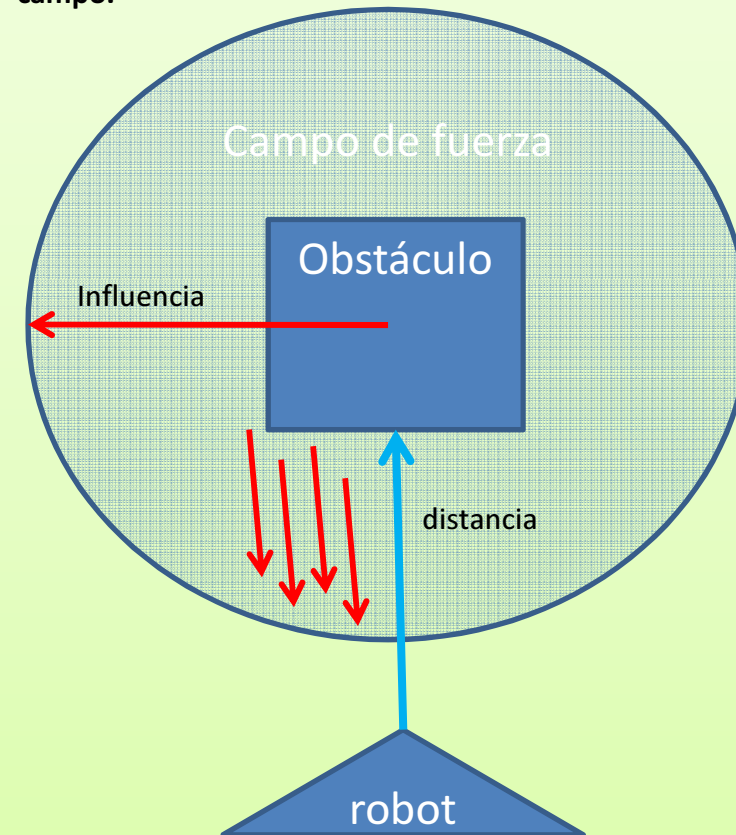
1. Después de leer los datos del robot.
2. Muestra el valor de la muestra central del láser
 1. la variable lp "se comporta" como un array. Cada muestra está en la posición lp[i]
3. Comprueba las distancias mínimas detectadas por el laser en su sector izquierdo y en el derecho.
 1. Ver http://playerstage.sourceforge.net/doc/Player-3.0.2/player/classPlayerCc_1_1LaserProxy.html para conocer qué hacen GetMinLeft y GetMinRight.
 2. Si hay un obstáculo más cerca a la izda, establece la velocidad angular en -10º/seg, es decir, giro a la izda (dtor es una función que transforma grados en radianes, rtod es la inversa)
 3. Si hay un obstáculo más cerca a la decha, gira a la izda. a una velocidad de 10º/seg.
4. Comprueba si justo delante (lp[180]) hay un obstáculo a menos de 0.5 metros.
 1. Si lo hay establece la variable de velocidad a 0, es decir, se para.
 2. Si no hay obstáculo delante, sigue a una velocidad de 0.1 m/sg.
5. Envía orden de marcha a los motores con velocidad lineal speed y velocidad angular turnrate.
 1. Ver los métodos aplicables a un objeto Position2dProxy en http://playerstage.sourceforge.net/doc/Player-3.0.2/player/classPlayerCc_1_1Position2dProxy.html

Otro programa ejemplo (repulsivo-simple.cc)

```
1
2 #include <iostream>
3 #include <libplayerc++/playerc++.h>
4 #include <cmath>
5
6 void repulsivo (double &mag, double &dir,
7               _double distancia, double influencia)
8 {
9     if (distancia <= influencia) {
10         dir = -180;
11         mag = (influencia-distancia)/influencia;
12     }
13     else {
14         dir = 0;
15         mag = 0;
16     }
17 }
18
19 int main(int argc, char *argv[])
20 {
21     using namespace PlayerCc;
22
23     PlayerClient  robot("localhost");
24     LaserProxy    lp(&robot,0);
25     Position2dProxy pp(&robot,0);
26
27     const double D = 2.0; //metros de influencia máxima del
28     const double maxSpeed = 0.3; //velocidad máxima en m/s
29     double magnitud; //factor multiplicativo [0,1] de la ve
30     double direction; //ángulo de desplazamiento, direction;
31     double turnrate, speed; //velocidad angular y lineal fir
32     lp.RequestGeom();
33     for(;;)
34     {
35         // lee de los proxies
36         robot.Read();
37         // muestra la muestra central del laser.
38         std::cout << lp[180] << std::endl;
39         speed = maxSpeed;
40         double angulo = 0;
41         repulsivo(magnitud, direction, lp[180],D);
42         speed = ((magnitud == 0) ? maxSpeed : maxSpeed*magnit
43         angulo = dtor(direction);
44         pp.SetCarlike(speed, angulo);
45     }
```

•¿Qué hace el programa?

- Es un programa para conseguir que un robot tenga un comportamiento repulsivo ante un obstáculo.
- La idea es simular un campo de fuerza lineal al rededor del obstáculo que repela al robot.
- El robot saldrá del campo con una dirección de -180° (irá marcha atrás) a una velocidad proporcional a la fuerza del campo.



Otro programa ejemplo (repulsivo-simple.cc)

```
1
2 #include <iostream>
3 #include <libplayerc++/playerc++.h>
4 #include <cmath>
5
6 void repulsivo (double &mag, double &dir,
7               _double distancia, double influencia)
8 {
9     if (distancia <= influencia) {
10         dir = -180;
11         mag = (influencia-distancia)/influencia;
12     }
13     else {
14         dir = 0;
15         mag = 0;
16     }
17 }
18
19 int main(int argc, char *argv[])
20 {
21     using namespace PlayerCc;
22
23     PlayerClient  robot("localhost");
24     LaserProxy    lp(&robot,0);
25     Position2dProxy pp(&robot,0);
26
27     const double D = 2.0; //metros de influencia máxima del
28     const double maxSpeed = 0.3; //velocidad máxima en m/s
29     double magnitud; //factor multiplicativo [0,1] de la ve
30     double direction; //ángulo de desplazamiento, direction;
31     double turnrate, speed; //velocidad angular y lineal fir
32     lp.RequestGeom();
33     for(;;)
34     {
35         // lee de los proxies
36         robot.Read();
37         // muestra la muestra central del laser.
38         std::cout << lp[180] << std::endl;
39         speed = maxSpeed;
40         double angulo = 0;
41         repulsivo(magnitud, direction, lp[180],D);
42         speed = ((magnitud == 0) ? maxSpeed : maxSpeed*magnit
43         angulo = dtor(direction);
44         pp.SetCarlike(speed, angulo);
45     }
```

•¿Qué hace la función?

•Entradas:

- **distancia:** distancia del robot al obstáculo (normalmente obtenida por la lectura del laser)
- **influencia:** radio (metros) de influencia del campo desde el centro del obstáculo.

•Salidas:

- **mag:** magnitud del campo, cuanto más cerca se esté del centro, más fuerte es el campo.
 - la magnitud del campo es la fuerza repulsiva del obstáculo y se puede corresponder con un factor con el que multiplicar la velocidad a la que el robot debe salir del campo.
 - **dir:** orientación del campo (en grados) que se corresponde con la orientación que debe tener el robot para salir del campo de fuerza
- si el robot está dentro de la influencia del campo (distancia <= influencia), la función devuelve
- dir = -180°, es decir, salir marcha atrás
 - mag = valor **normalizado (en [0,1])** proporcional a la cercanía del robot al obstáculo. Es usado fuera de la función como factor multiplicativo de la velocidad.
- si no está en el campo de influencia devuelve dir = mag = 0.a

Otro programa ejemplo (repulsivo-simple.cc)

```
1
2 #include <iostream>
3 #include <libplayerc++/playerc++.h>
4 #include <cmath>
5
6 void repulsivo (double &mag, double &dir,
7               _double distancia, double influencia)
8 {
9     if (distancia <= influencia) {
10         dir = -180;
11         mag = (influencia-distancia)/influencia;
12     }
13     else {
14         dir = 0;
15         mag = 0;
16     }
17 }
18
19 int main(int argc, char *argv[])
20 {
21     using namespace PlayerCc;
22
23     PlayerClient  robot("localhost");
24     LaserProxy    lp(&robot,0);
25     Position2dProxy pp(&robot,0);
26
27     const double D = 2.0; //metros de influencia máxima del
28     const double maxSpeed = 0.3; //velocidad máxima en m/s
29     double magnitud; //factor multiplicativo [0,1] de la ve
30     double direction; //ángulo de desplazamiento, direction;
31     double turnrate, speed; //velocidad angular y lineal fir
32     lp.RequestGeom();
33     for(;;)
34     {
35         // lee de los proxies
36         robot.Read();
37         // muestra la muestra central del laser.
38         std::cout << lp[180] << std::endl;
39         speed = maxSpeed;
40         double angulo = 0;
41         repulsivo(magnitud, direction, lp[180],D);
42         speed = ((magnitud == 0) ? maxSpeed : maxSpeed*magnit
43         angulo = dtor(direction);
44         pp.SetCarlike(speed, angulo);
45     }
```

•¿Qué hace el programa

1. Lee información de los sensores.
2. Envía a la función repulsivo el valor de la muestra central del láser como representando la distancia al obstáculo.
3. Toma el valor de magnitud y direction devueltos por la función repulsivo.
4. Determina el valor final de speed.
5. Convierte el valor del ángulo a radianes
6. Envía la orden de moverse tipo "carlike" al robot.
 1. No olvidar que para este caso hay que definir en **Stage** el modelo de robot como **drive "car"**
 2. Tampoco olvidar que estamos en un ciclo continuo, en cada iteración el programa lee la distancia a que se encuentra el robot del obstáculo y actualiza continuamente los valores de speed y angulo adecuadamente.