

Signals, Instruments, and Systems – W7

**High-Fidelity Simulation of
Real-Time Embedded
Systems**

Outline

- A few words about simulation!
- Webots
 - Generalities
 - Sensors/actuators
 - Webots API
- Programming embedded systems
 - Buffers
 - Timing
 - An example using the camera in Webots
- Quick note about communication

Simulation: why?

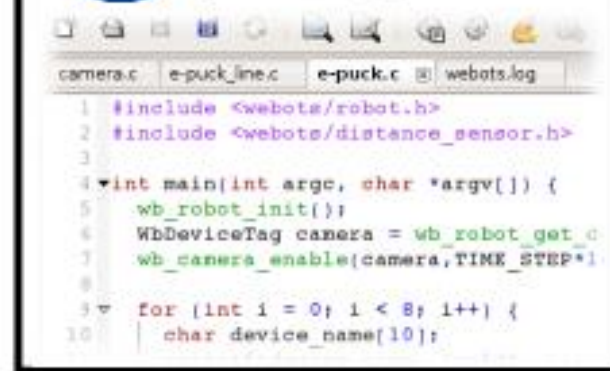
- Hardware prototyping is time-consuming, expensive and even sometimes risky
- Real hardware platforms (e.g., robots) might be expensive
- Quickly change the experimental setup
- Often easier for monitoring experiments (and evaluating specific metrics)
- Sometimes faster than real-time
 - Useful for using numerical optimization schemes (genetic algorithms, particle swarm optimizations, etc.)
 - Enable systematic search of the parameter space

Webots robotic simulator

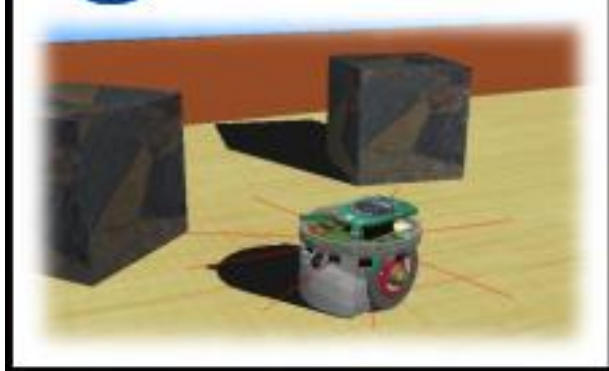
1 model



2 program



3 simulate



4 transfer



In this course, we will focus on steps **2**, **3** only.

Webots features

- Realistic (physics-based) robotics simulator
- Fast prototyping
- Experiment without real hardware
- Can *usually* run faster than real-time
- Available sensors: distance sensors, light sensors, cameras, accelerometers, touch sensors, position sensors, GNSS, receivers, force sensors, etc.
- Available actuators: linear and rotational motors, grippers, LEDs, emitters, etc.

How does it look like (3)?



e-puck mobile robot performing obstacle avoidance
and line following

Webots GUI

scene tree

world view

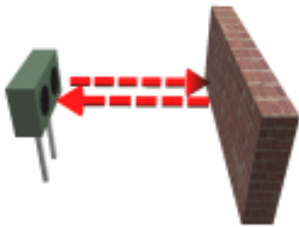
editor

The screenshot displays the Webots GUI with the following components:

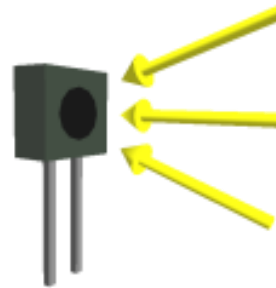
- Scene Tree (Left):** A hierarchical list of objects in the simulation, including 'WorldInfo', 'Viewpoint', 'Background', 'PointLight', 'SquareArena', and several 'DEF short_rock' objects, along with 'DEF E_PUCK DifferentialWheels'.
- World View (Center):** A 3D rendering of a robot on a yellowish ground with several grey rocks. A red circle highlights the '0.70x' simulation speed indicator in the top toolbar.
- Code Editor (Right):** A C++ source file named 'e-puck.c' containing code for robot initialization and sensor enablement. The code includes headers for robot, differential_wheels, distance_sensor, light_sensor, camera, and accelerometer, and defines a main function that sets up these sensors.
- Console (Bottom):** A terminal window showing the compilation process using 'make' and 'gcc', resulting in the executable 'e-puck'.

console

Modeled sensors



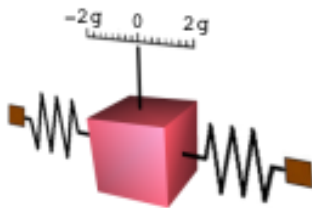
distance sensor



light sensor



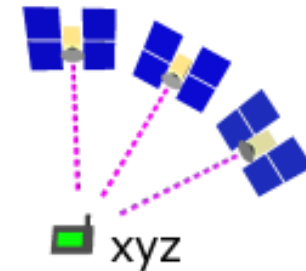
camera



accelerometer



touch/pressure
sensor



GNSS

...and battery sensor, compass, gyroscope, torque sensor, etc.

Modeled actuators



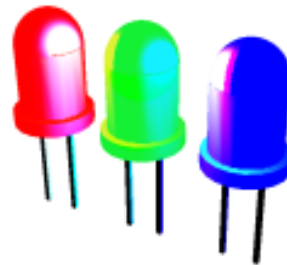
motor (rotational / linear)



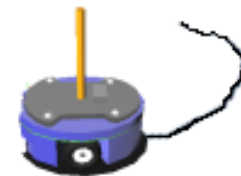
gripper



connector
(docking systems)



LED

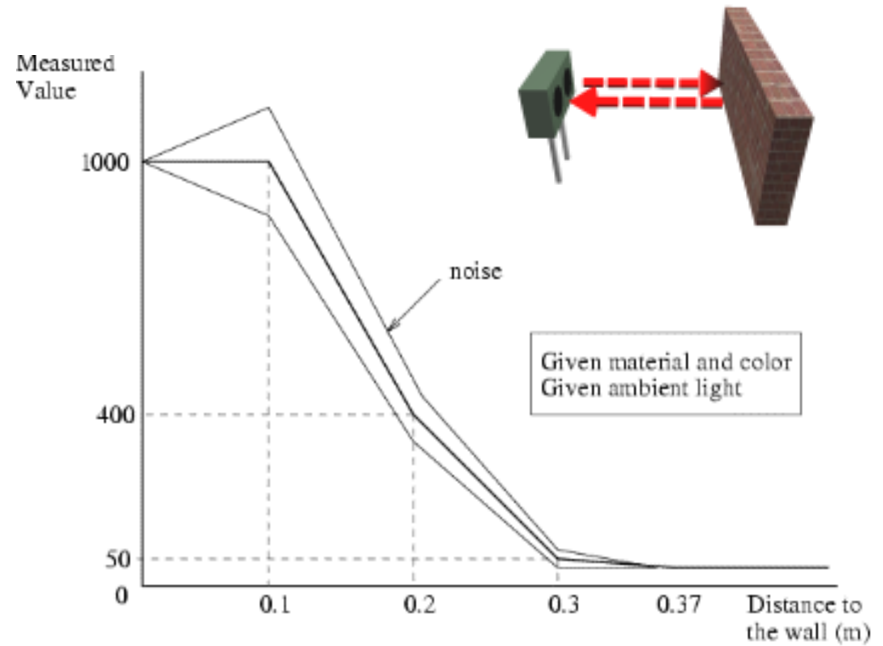


pen

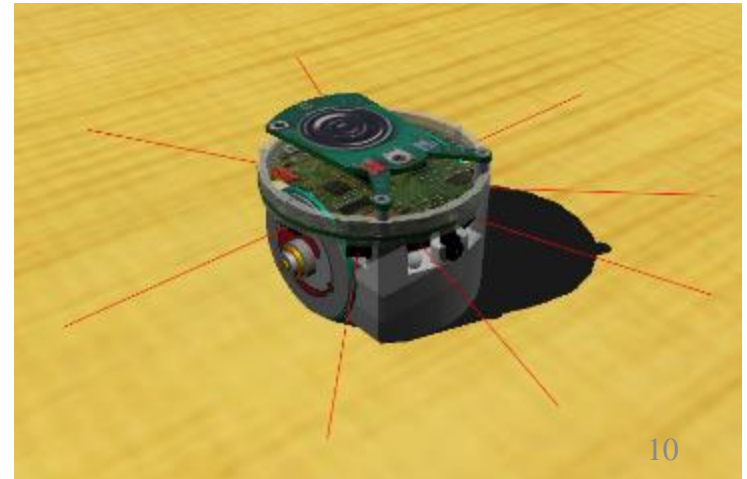
... and propeller, emitter & receiver, etc.

Modeling sensors

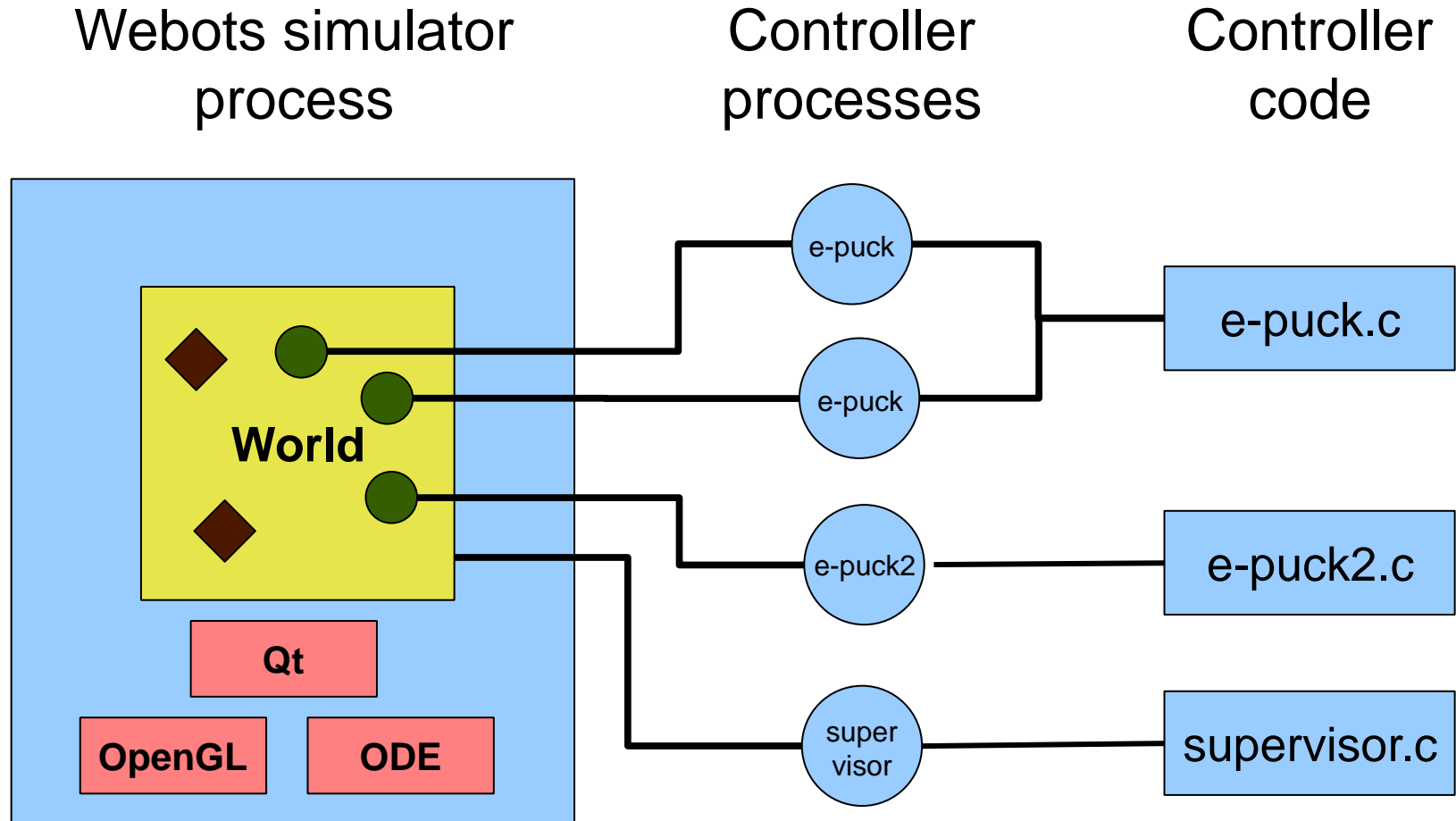
- Capture **non-linearities** and **noise** of sensors.
- However, **calibration** is often approximative.
- Most often, sensor response is defined by a lookup table (here a proximity sensor):



```
lookupTable [
    0
    0.1
    0.2
    0.3
    0.37
    1000
    1000
    400
    50
    30
    0,
    0.1,
    0.1,
    0.1,
    0 ]
distance value noise
```



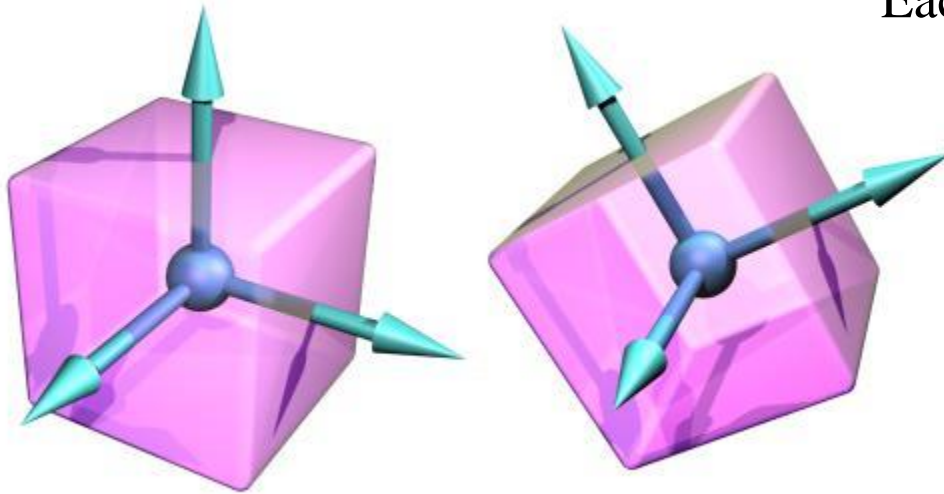
Webots principles



The more robots, the slower the simulation!

(Newtonian) physics-based simulation (ODE)

Rigid bodies simulation:

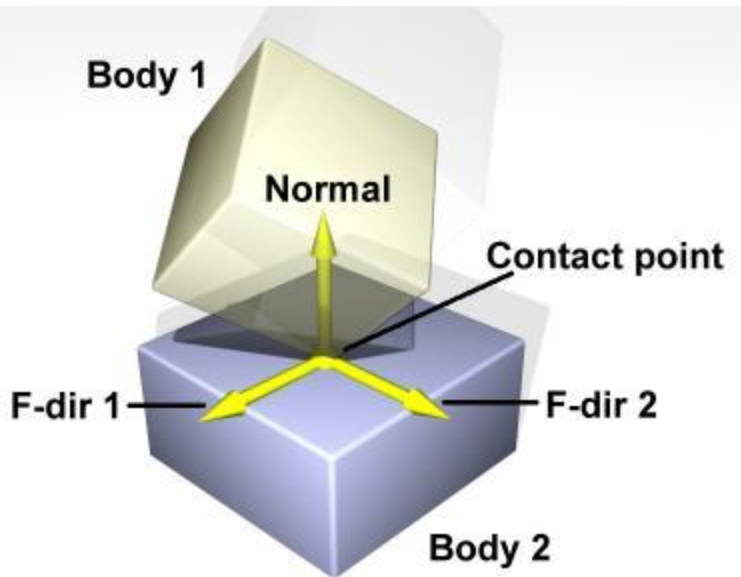


Each body is defined by its:

- Shape
- Mass (or density)
- Center of mass and Inertia Matrix
- Velocity (angular and linear)

(Newtonian) physics-based simulation (ODE)

Collision detection:



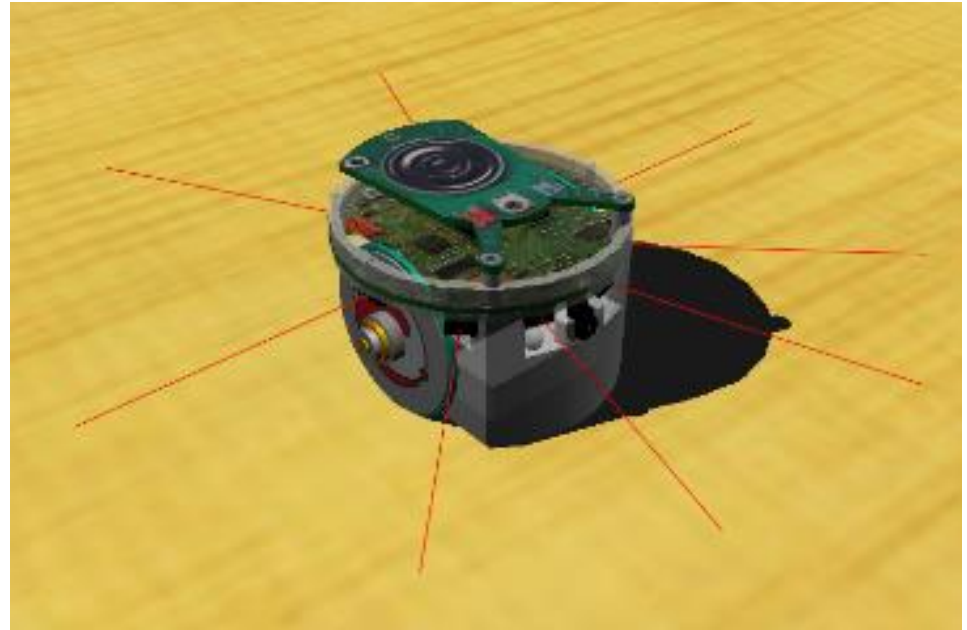
Configurable friction:



Real and simulated e-puck



Real e-puck



Simulated e-puck (Webots)

- sensor- and actuator-based
- noise, nonlinearities of sensors and actuators reproduced
- kinematic (e.g., speed, position) and dynamic (e.g., mass, forces, friction)

Supervisor

- A **supervisor** is a program that controls the world and its robots.
- Similarly as a robot, supervisor is a node driven by a controller with **extended capabilities** that supervises the whole world.
- The supervisor can read/write any field of any node in the scene tree in order to:
 - move or rotate any object in the scene
 - simulate changing environmental conditions
 - track the position of a robot
- The supervisor can also take a snapshot of the scene or create movies.

Webots controllers

- A Webots controller is a **C program** (Webots also supports C++, Java, Python, and even Matlab).
- Therefore, everything you can do in conventional C programs, you can do in a Webots controller.
- You must just keep in mind that your controller **will eventually run on a robot** (only simulated one on this semester).
- Webots **does not** simulate the microcontroller of your robot:
 - Your controller will run **much slower** on the real robot than it does in Webots .
 - Your memory will be **much more limited** on the real robot than in Webots.
- In the first case, the behavior of the real robot will be very different from that of the simulated robot. In the second case, you will not be able to compile your controller!

A typical Webots controller

```
#include <webots/robot.h>
#include <webots/differential_wheels.h>
#include <webots/distance_sensor.h>
```

Includes for accessing Webots API

```
#define TIME_STEP 32
```

Define simulation step in milliseconds

```
int main(int argc, const char *argv[]) {
    double speed[2] = {200.0, 200.0};
    double sensor_value;
    WbDeviceTag sensor;
```

Define data structures

```
// initialize webots
wb_robot_init();
```

Initialize Webots

```
// find distance sensors
sensor = wb_robot_get_device("ps0");
wb_distance_sensor_enable(sensor, TIME_STEP);
```

Get and enable different devices
(sensors and actuators)

```
// main loop
while (wb_robot_step(TIME_STEP) != -1) {
    sensor_value = wb_distance_sensor_get_value(sensor);
    if (sensor_value > 500.0) {
        speed[0] = 0.0; speed[1] = 0.0;
    }

    // set the motors speed
    wb_differential_wheels_set_speed(speed[0], speed[1]);
}
```

Perform one simulation step

Read sensor values

Controller lifetime loop (robot
behavior must be coded here)

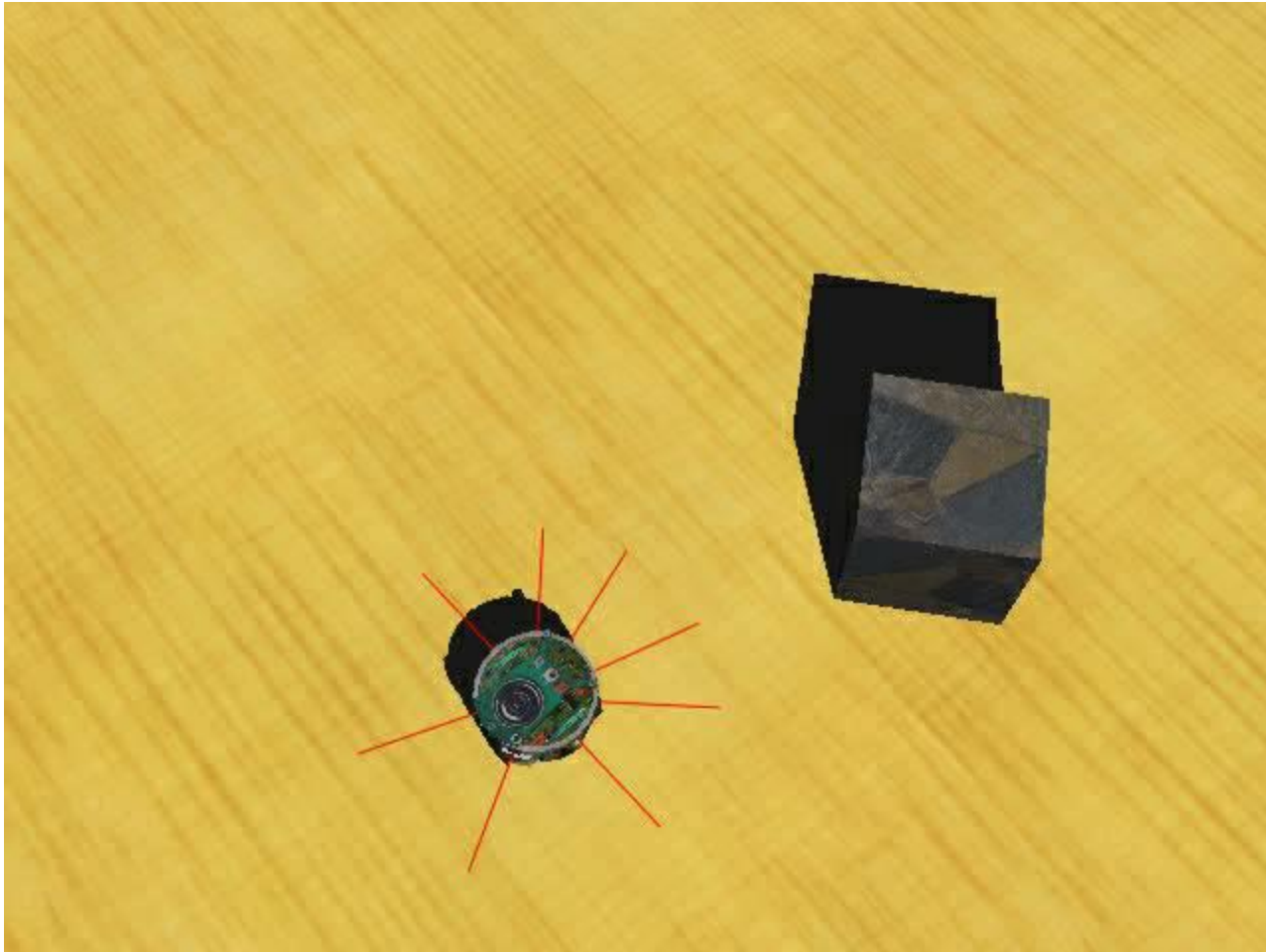
Update actuators

```
// cleanup webots resources
wb_robot_cleanup();
```

Cleanup Webots resources

```
}
```

How does it look like?



- **Definition:** an application programming interface (API) is a set of functions, procedures, methods or classes that an operating system, library or service provides to support requests made by computer programs.
- Webots provides a lot of such functions that allow you to interact with the different devices of your robot:

```
#include <webots/distance_sensor.h>
```

```
void wb_distance_sensor_enable(WbDeviceTag tag, int ms);  
void wb_distance_sensor_disable(WbDeviceTag tag);  
double wb_distance_sensor_get_value(WbDeviceTag tag);
```

- You can find all of them in the Webots Reference Manual available at <https://www.cyberbotics.com/doc/reference/index> !
- The principle of the API is that you must always enable a sensor before using them (pretty much like on a real robot)!
- **The Webots API is NOT available on the real robot.** This means that you will need to modify your controller before transferring it to the real robot. In the **specific** case of the e-puck, cross-compilation is available, though.



- If you want to use a camera, you need to include the following header file:

```
#include <webots/camera.h>
```

- The first thing to do next is to get the device by using the standard function:

```
WbDeviceTag camera = wb_robot_get_device("cam");
```

device tag

name of the device
(see scene tree)

- Then, you need to enable it before using it (and possibly disable it if you no longer need it):

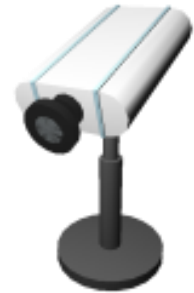
```
void wb_camera_enable(WbDeviceTag tag, int ms);
void wb_camera_disable(WbDeviceTag tag);
```

device tag

1/refresh rate in kHz

- You also have a number of auxiliary functions that might come in handy at some point:

```
int wb_camera_get_width(WbDeviceTag tag);
int wb_camera_get_height(WbDeviceTag tag);
```



- Then, you can of course get an image using the following functions:

```
unsigned char *wb_camera_get_image(WbDeviceTag tag);
unsigned char wb_camera_image_get_red(const unsigned char *image,
int width, int x, int y);
unsigned char wb_camera_image_get_green(const unsigned char *image,
int width, int x, int y);
unsigned char wb_camera_image_get_blue(const unsigned char *image,
int width, int x, int y);
unsigned char wb_camera_image_get_grey(const unsigned char *image,
int width, int x, int y);
```

- Here is an example of usage of these functions:

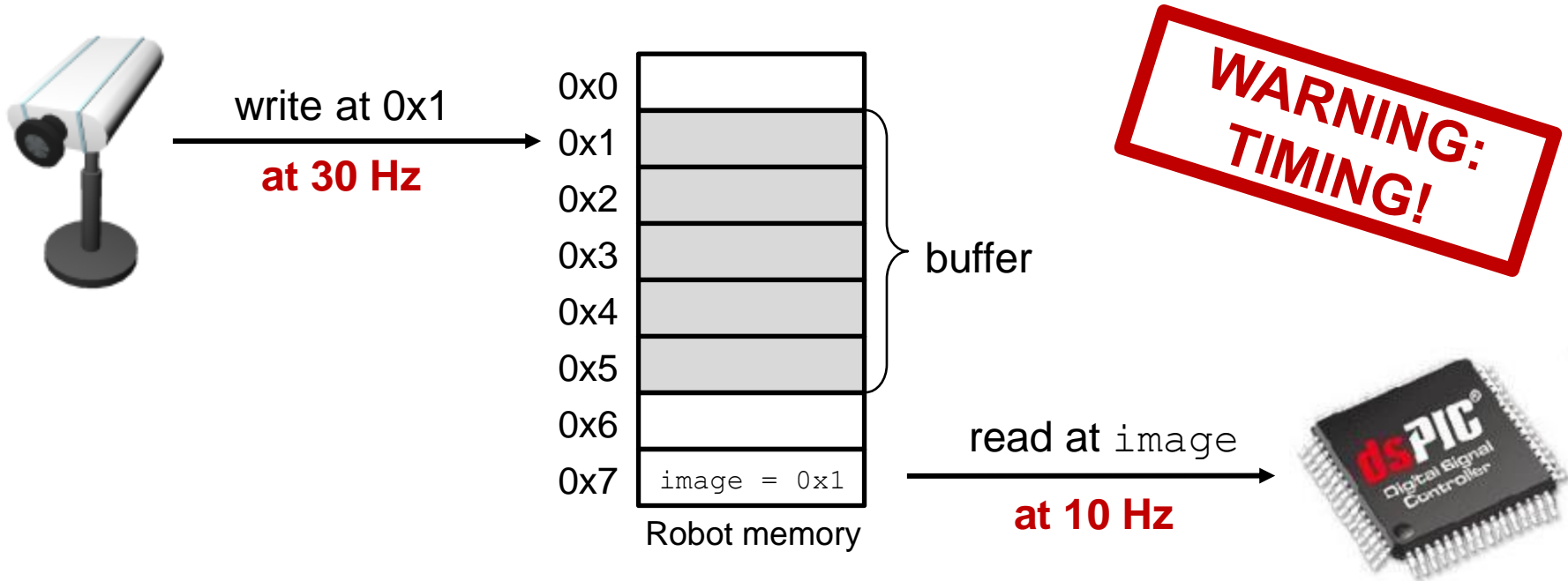
```
const unsigned char *image = wb_camera_get_image(camera);
for (int x = 0; x < image_width; x++) {
    for (int y = 0; y < image_height; y++) {
        int r = wb_camera_image_get_red(image, image_width, x, y);
        int g = wb_camera_image_get_green(image, image_width, x, y);
        int b = wb_camera_image_get_blue(image, image_width, x, y);
        printf("red=%d, green=%d, blue=%d", r, g, b);
    }
}
```

The concept of buffer

- A **buffer** is a region of memory used to temporarily hold data while it is being moved from one place to another.
- Typically, the data is stored in a buffer as it is retrieved from an input device (such as a sensor) or just before it is sent to an output device (such as an actuator).
- The device writes/reads in the buffer independently of the controller. Therefore, to read the device, you just need to read the buffer, using **a pointer**:

```
const unsigned char *image = wb_camera_get_image(camera);      image buffer
for (int x = 0; x < image_width; x++) {
    for (int y = 0; y < image_height; y++) {
        int r = wb_camera_image_get_red(image, image_width, x, y);
        int g = wb_camera_image_get_green(image, image_width, x, y);
        int b = wb_camera_image_get_blue(image, image_width, x, y);
        printf("red=%d, green=%d, blue=%d", r, g, b);
    }
}
```

Buffer mechanism



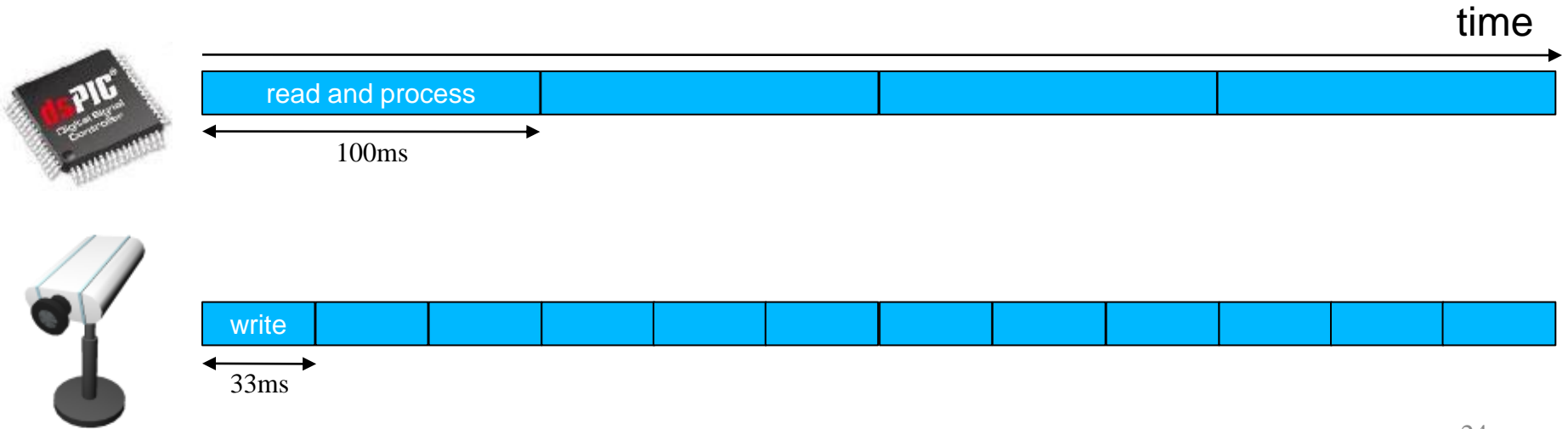
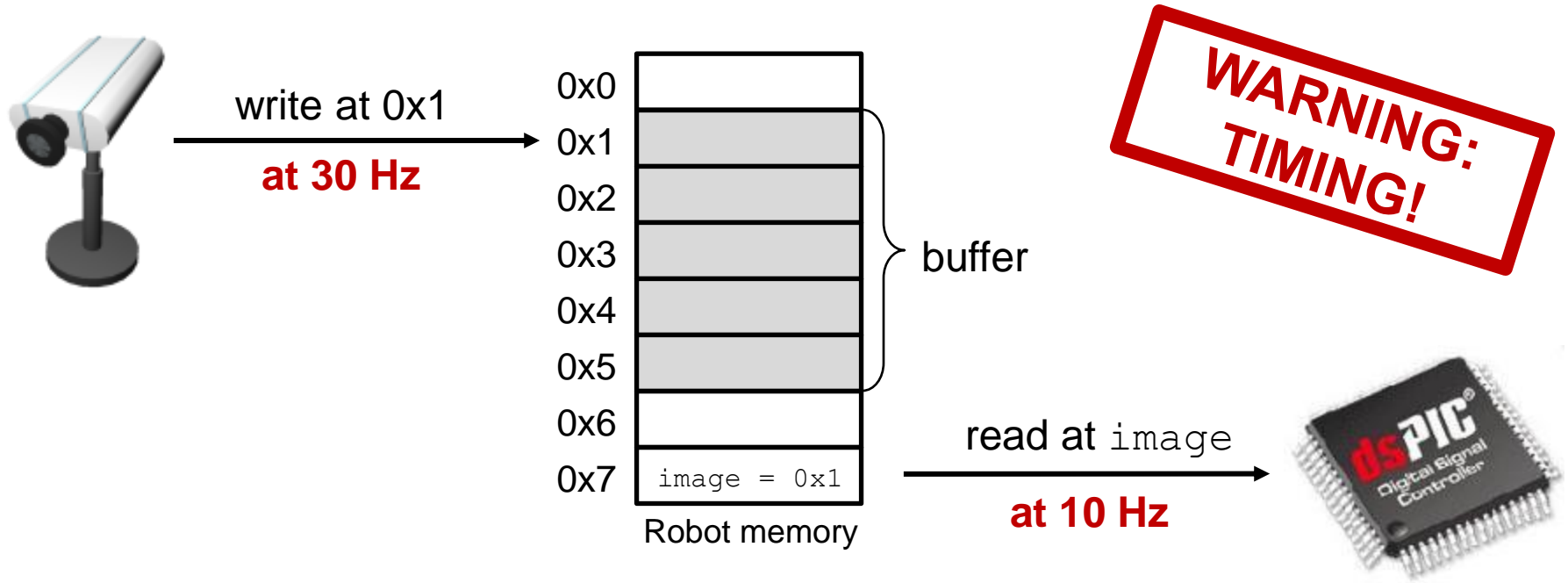
```

const unsigned char *image = wb_camera_get_image(camera);
for (int x = 0; x < image_width; x++) {
  for (int y = 0; y < image_height; y++) {
    int r = wb_camera_image_get_red(image, image_width, x, y);
    int g = wb_camera_image_get_green(image, image_width, x, y);
    int b = wb_camera_image_get_blue(image, image_width, x, y);
    printf("red=%d, green=%d, blue=%d", r, g, b);
  }
}

```

Data can be lost! Only one frame out of three is actually processed here!

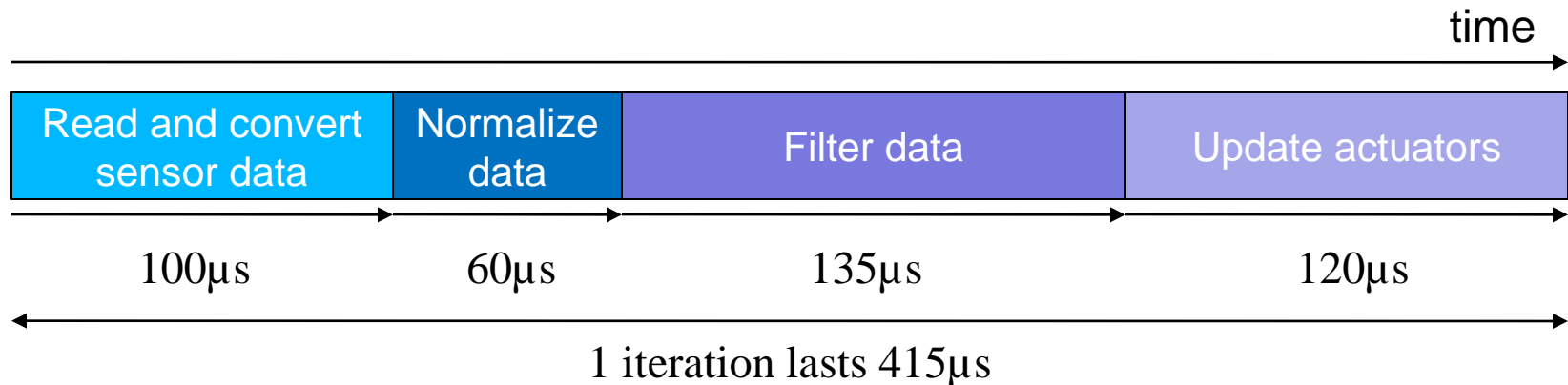
Buffer mechanism



- Each type of robot (DifferentialWheels, Robot or Supervisor) controller may be **synchronous** or **asynchronous**.
- Webots waits for the requests of **synchronous** robots before performing the next simulation step.
- It does not wait for **asynchronous** robots.
- Hence, an **asynchronous** robot may be **late** (if the controller is computationally expensive, or runs on a remote computer with a slow network connection).
- Obviously, in reality, all robots are **asynchronous** (with respect to real time).
- In practice, we use synchronous robots in simulation because Webots (like most simulation packages) **does not** simulate the microcontroller anyway.
- Details: <https://cyberbotics.com/doc/reference/robot#synchronous-versus-asynchronous-controllers>

Real time?

- A robot (like any computing entity) has **limited computational resources**. Therefore, any controller has a **computational cost**, which can be expressed as the time required to perform one iteration of the main loop.



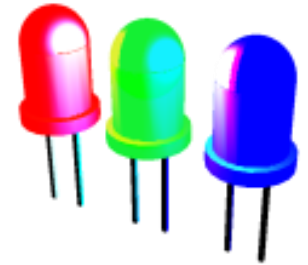
- For instance, the total duration of one single iteration of the controller depicted above is $415\mu\text{s}$ (~ 0.5 ms).
- Therefore, the maximal execution speed of the loop (which is also the update rate of the actuators) is 2 kHz.

Perception-to-Action Loop (closed-loop system)

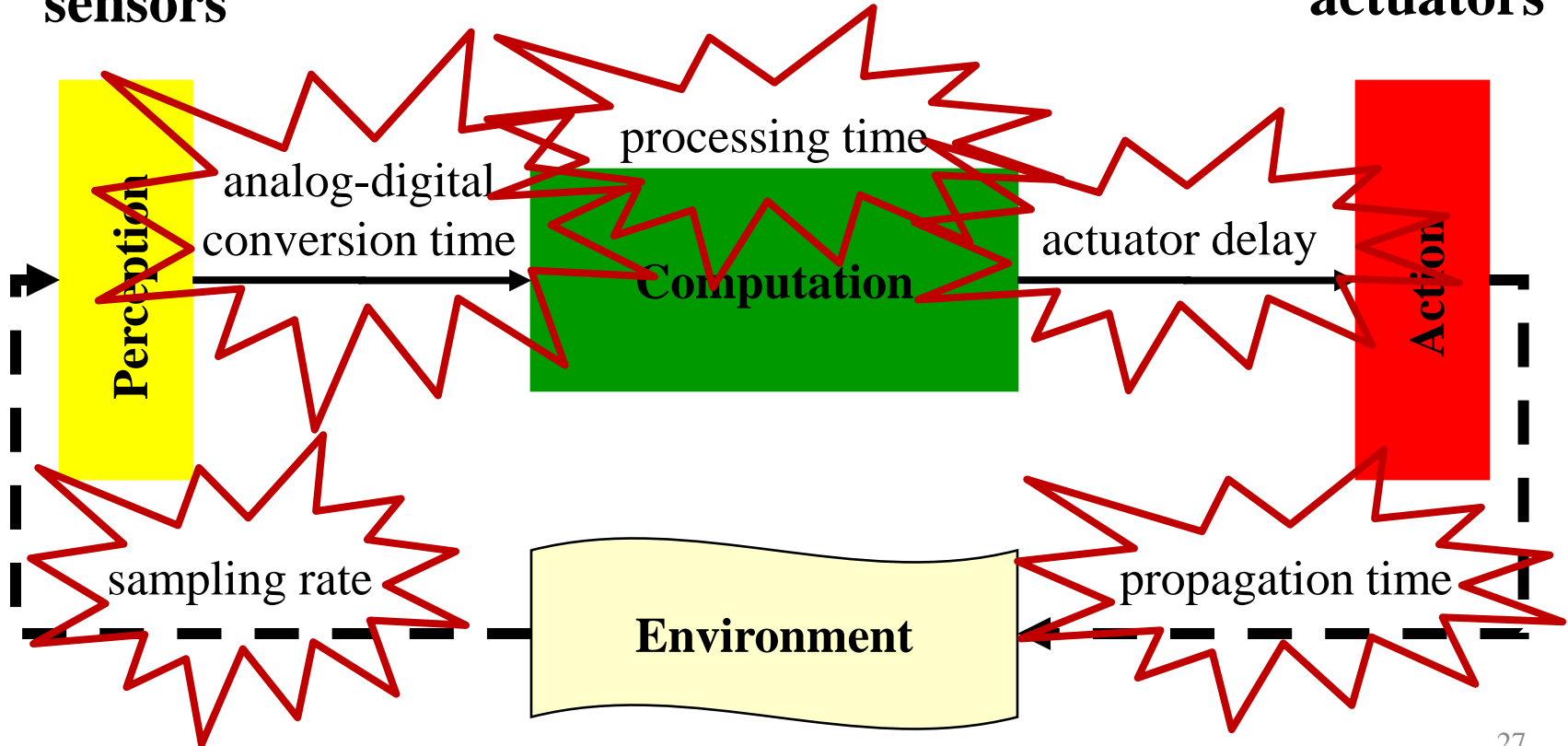


sensors

Delays are everywhere!



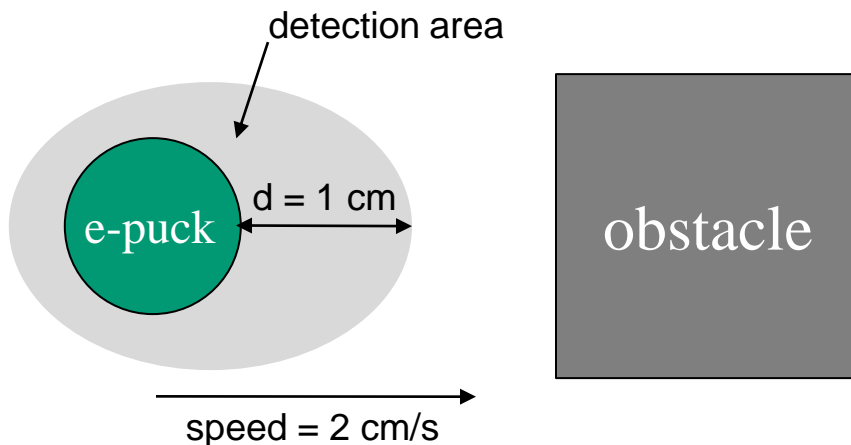
actuators



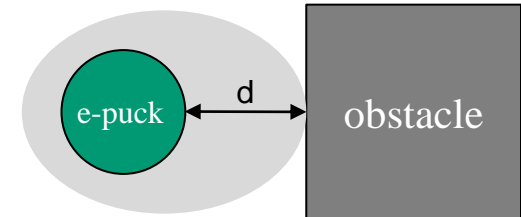
Real time!

- The perception-to-action delay defines the responsiveness of the robot.
- If the perception-to-action loop is too slow, the robot (and, in general, the embedded system) might miss important events!
- **Obstacle avoidance example:**

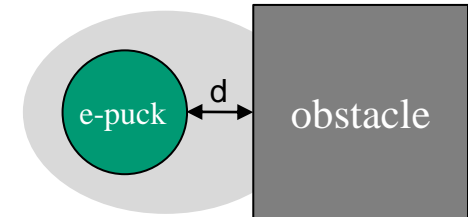
- What is the maximal perception-to-action loop delay (in ms) required to prevent collisions?



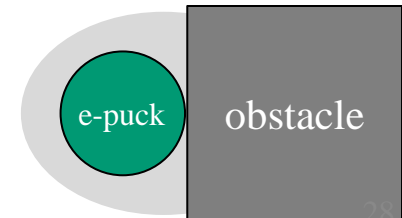
$t = 0$ ms (no detection at $d = 1$ cm)



$t = 250$ ms (1st detection at $d = 0.5$ cm)



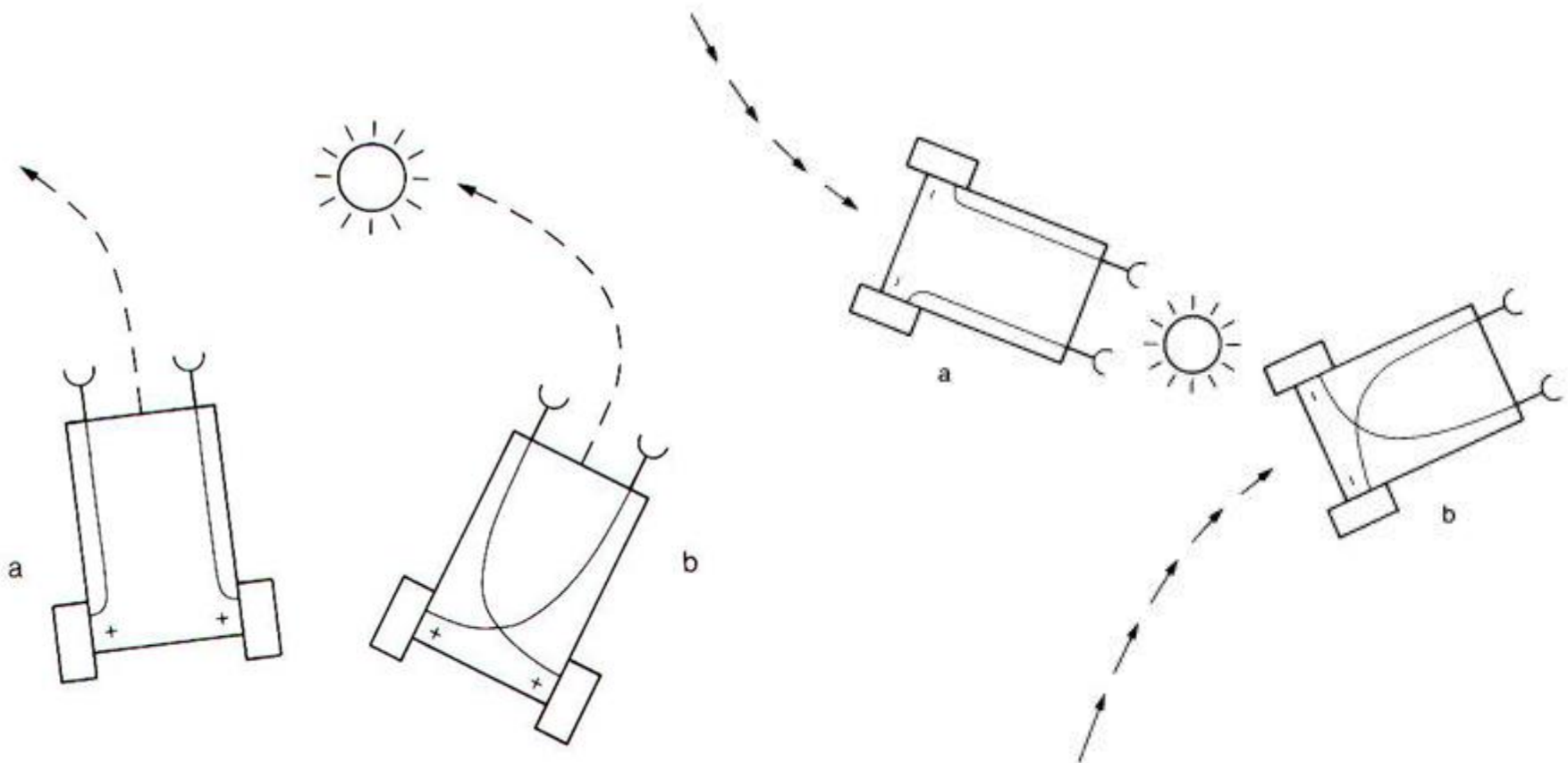
$t = 500$ ms (stop at $d = 0$ cm)



1 perception-to-action loop

Theoretical answer: at most 250ms (at least 4 Hz)!
In practice, we need much faster responses!

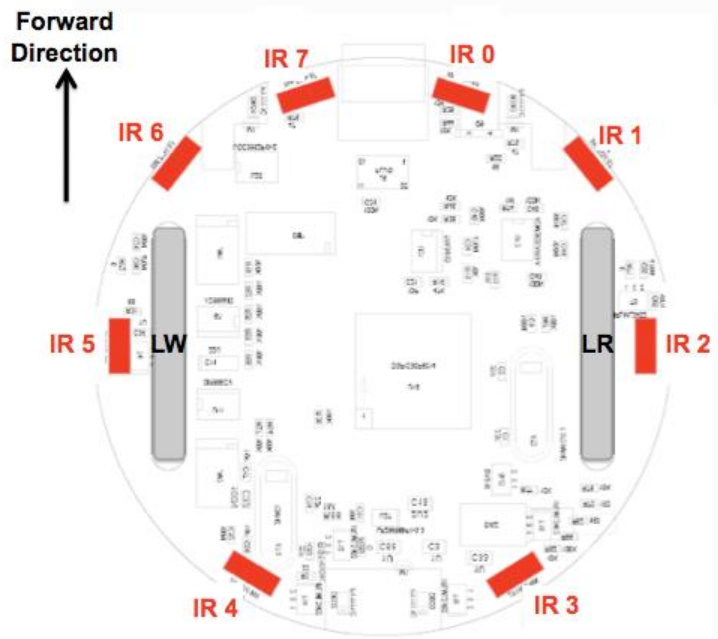
Robot control – Braitenberg



Excitatory connections

Inhibitory connections

Braitenberg applied to e-puck

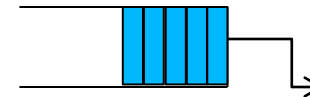


- 2 actuators
- 8 proximity sensors
- Motor speed is a linear combination:

$$\begin{bmatrix} v_L \\ v_R \end{bmatrix} = \begin{bmatrix} \alpha_{L0} & \alpha_{L1} & \cdots & \alpha_{L7} \\ \alpha_{R0} & \alpha_{R1} & \cdots & \alpha_{R7} \end{bmatrix} \cdot \begin{bmatrix} d_{IR0} \\ \vdots \\ d_{IR7} \end{bmatrix}$$

- Webots **does not** simulate the communication delays:
 - Communication packets have always a delay of a time-step.
 - Which may be longer or shorter than reality.

```
static WbDeviceTag emitter = wb_robot_get_device("emitter");
...
for (i = 0; i < 5; i++)
    wb_emitter_send(emitter, "Hello!", 7);
```



Emitter

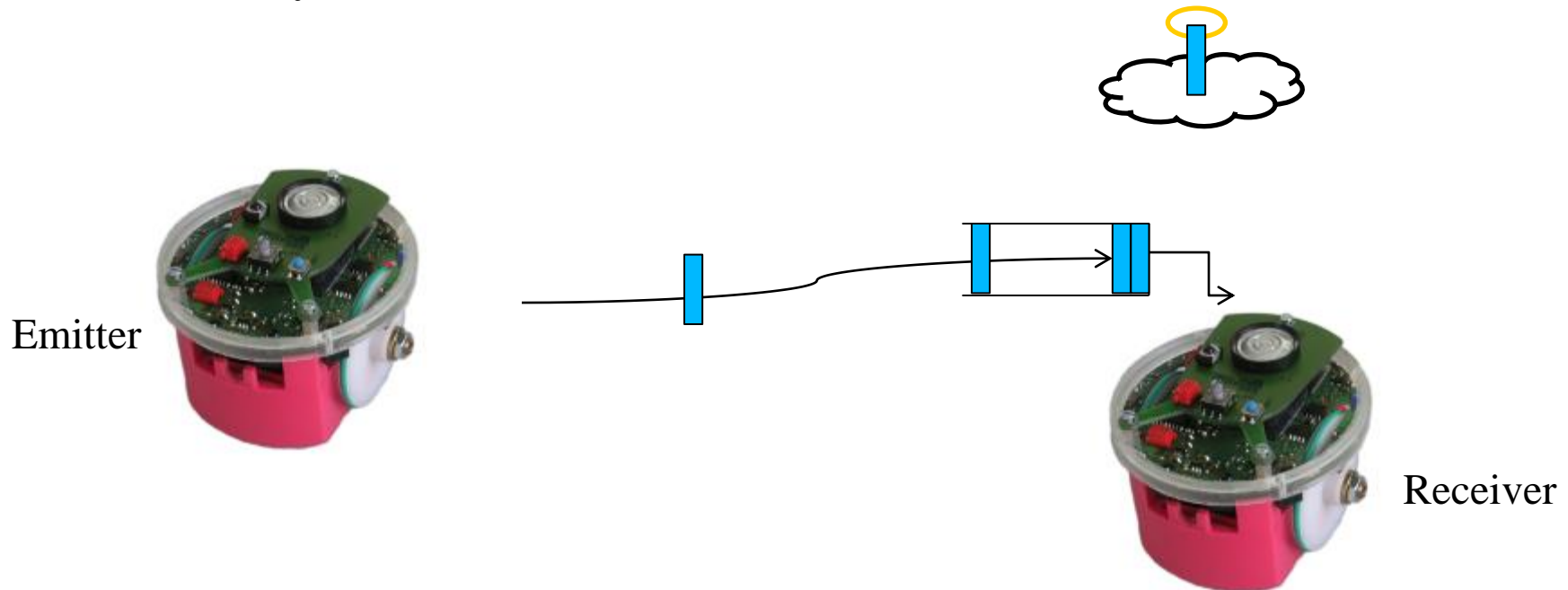
**After a
time-step**



Receiver

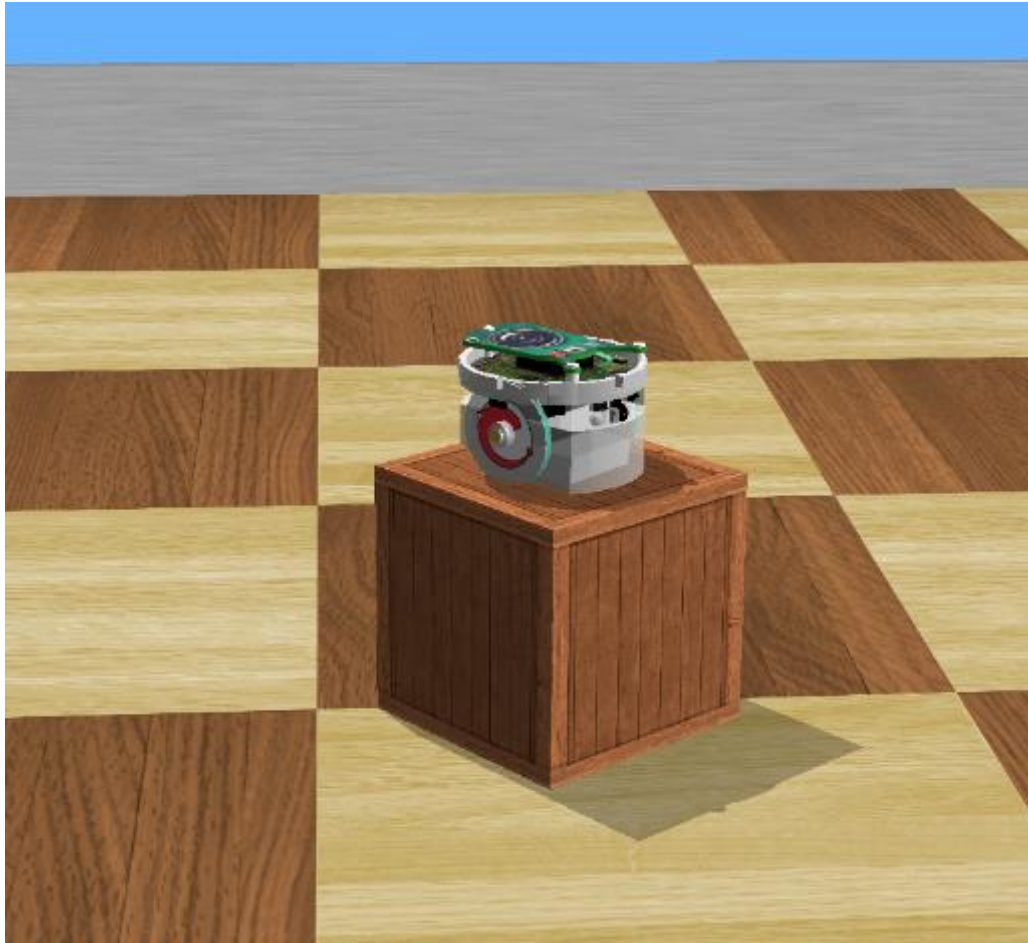
```
static WbDeviceTag receiver = wb_robot_get_device("receiver");
wb_receiver_enable(receiver, TIME_STEP);
...
while (wb_receiver_get_queue_length(receiver) > 0) {
    const char *message = wb_receiver_get_data(receiver);
    wb_receiver_next_packet(receiver);
}
```

- Everything is asynchronous and subject to variable delays.
- Packets may be lost.



- One cannot always expect that packets will arrive.
- Hence, the controller needs to account for this.
- Wireless communication is an **advanced physical phenomena**.

How to detect that e-puck is falling using accelerometer



The controller

```
#include <webots/robot.h>
#include <webots/differential_wheels.h>
#include <webots/accelerometer.h>

#include <stdio.h>

#define TIME_STEP 64

int main(int argc, char *argv[]) {

    /* initialize Webots */
    wb_robot_init();

    /* get and enable devices */
    WbDeviceTag accelerometer = wb_robot_get_device("accelerometer");
    wb_accelerometer_enable(accelerometer, TIME_STEP);

    /* move the robot */
    wb_differential_wheels_set_speed(100, 100);

    /* main loop */
    while (wb_robot_step(TIME_STEP) != -1) {
        const double *accelerations = wb_accelerometer_get_values(accelerometer);

        if(accelerations[2] < 5.0) {
            printf("HELP!!!\n");
            wb_differential_wheels_set_speed(0, 0);
        }
    }
    wb_robot_cleanup();
    return 0;
}
```

- Default empty Webots controller
- Move the e-puck
- Get and enable the accelerometer
- Read accelerometer
- Check if falling

Conclusion

Take-home messages

- Programming embedded systems can be a difficult task: simulation is here to help!
- Simulation allows one to achieve rapid prototyping and experimentation **without using actual hardware**.
- However, simulation is **an abstraction** of reality and does not account for all the details of the target system. Webots, as a **high-fidelity system-level simulator**, does not take into account all sub-aspects (e.g. voltage drop due to motor current of e-puck)
- Be especially careful about **timing issues** (i.e., perception-to-action delay): Webots only approximates these aspects.
- Real-time aspects are a key ingredient of embedded systems: more about that later

Reading and acknowledgements

- Have a look at the Webots User Guide:
<https://www.cyberbotics.com/doc/guide/index>
- If you need help about a specific function and/or device, refer to the Webots Reference Manual:
<https://www.cyberbotics.com/doc/reference/index>
- In case of problems with the e-puck robot, refer to the official website
<https://cyberbotics.com/doc/guide/epuck>