Loe Feijs
Department of Industrial Design, Eindhoven University of Technology, The Netherlands
l.m.g.feijs@tue.nl

Multi-tasking and Arduino: Why and How?

Abstract
In this article I argue that it is important to develop experiential prototypes which have multi-tasking capabilities. At the same time I show that for embedded prototype software based on the popular Arduino platform this is not too difficult. The approach is explained and illustrated using technical examples – practical and hands-on, down to the code level. At the same time a few helpful notations for designing and documenting the software are introduced and illustrated by the same examples. Finally a few case studies of the technical approach are listed.

Keywords
Prototyping, specification languages, parallelism, multitasking, experiential, models

1 Introduction

In the word, in which we live, there is a lot of parallelism. Around us many things happen and they happen more or less simultaneously. Zooming in to the detailed behavior and the internal working of the objects around us we see more parallelism. If we zoom out to our environment and consider what happens at some distance even more parallelism becomes visible. As human beings, we cope with this aspect of the world's complexity remarkably well. We are able to perform multitasking (Fig. 1).

Fig. 1. Humans are good in multitasking: seeing, navigating, communicating, feeling temperature etc.

So if we are designing artifacts which display some form of intelligence, it is natural to assume that they can perform multitasking as well. For ambient intelligent environments, this is usually the case indeed, which is no surprise when they are equipped with powerful sensors such as cameras and data from mobile devices; these are all connected to networks of servers which run in the background with lots of parallelism anyhow. Modern computers, including PCs, smart phones and Internet servers are equipped with Operating Systems which allow for many parallel processes on a single computer. Even small stand-alone embedded processors can have operating systems, usually marketed as RTOS (real time operating system). Technically these are appropriate, but somehow they are considered high-threshold for programming quick experiential prototypes. For this reason interaction designers and industrial designers

prefer systems such as MAX, Processing (for PC and MAC platforms) and Arduino (for embedded applications). Arduino does not come with an operating system. It works with the same theatre metaphor as the Processing environment which is meant for creative work on PC. There are two functions: `setup()` prepares the stage and `loop()` defines the actions.

Arduinos are used as control devices, in combination with sensors in of a huge variety of experimental art installations, experiential prototypes and experimental research vehicles. See [1] and [2]. Some of these systems sense light conditions, others detect sounds, temperature differences, bodily parameters, movements and so on. Their actuators include motors, loudspeakers, lamps, lasers, fluid valves and servo motors, voice coils, memory metals, just to name a few of many options.

For many of these sensors and actuators there is a software library and examples how to connect, read and write these sensors. Usually `setup()` defines which device is connected to which input or output pins. And usually `loop()` contains a few read, write, and delay actions. The actions depend on the inputs received so a few if-else or switch-case statements are thrown in and a few variables to keep track of what goes on in the external world.

All this is well, but when combining a few sensors and actuators the programs suddenly become much harder to design. During the delay needed for one task, the Arduino is unreceptive for the signals of other sensors. All the nested if-else and switch case statements soon blow up the `loop()` to a multi-page debugging nightmare. Lowering the project's ambition level and adjusting the planning to allow for more programming and debugging time are typical decisions I witnessed in student projects.

As a way out, some of the designers work with interrupts, allowing external events to launch interrupt service routines which briefly suspend the ongoing task. No doubt this is a powerful technique, but it requires a much deeper knowledge of the processor and its hardware. Often the service routines are written in assembly language and manipulate the processor's timer registers.

In this article I describe an approach to avoid these difficulties - the only price being a bit of discipline to adhere to an orderly structure. At the same time the approach offers very nice opportunities to document the software at a higher level than the code itself. In particular, I propose State Transition Diagrams (STD, sometimes called Finite State Machines) as a helpful way of explaining and documenting the logic of decision making (normally implicitly coded in if-else and switch case). And I propose Data Flow Diagrams (DFD) as a desirable way of giving a high-level overview of the parallel tasks going on and how they communicate via shared variables.

The notations of STD and DFD are not new: they were developed in software engineering and telecommunication industry more than two decades ago. They are among the simplest and yet most powerful notations of software engineering. Yet they appear not to have been handed down with the other nice things which were adopted in Arduino culture.

The next sections contain examples illustrating the proposed principles. The nice property of the approach is that it is compositional: the `loop()` does not become exponentially more complex. A `loop()` with two parallel tasks contains 2+1 statements. A `loop()` with twenty parallel tasks contains 20+1 statements.

2 State Transition Diagrams

The example of this section serves mainly to introduce the STD notation. The system is the type of light encountered in the corridors of hotels: the guest pushes a button and the light goes on. After ten seconds, the light goes off automatically. The system therefore has two states: ON and OFF, represented by the circles in the diagram of Fig. 2.

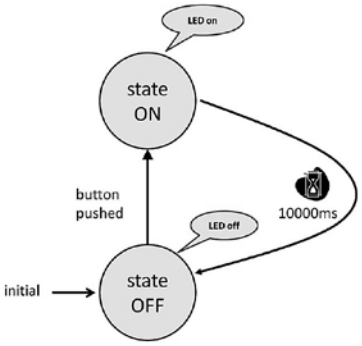

Fig. 2. State Transition Diagram with two states.

One of these states is labeled to be the initial state. The states themselves are an abstraction and the text balloons tell about the resulting actions in the real world. The transitions from one state to the next are represented by arrows. Each arrow is labeled by a condition or an external action which tells when the transition happens. The transition from state OFF to state ON happens when the hotel guest has pushed the button. The transition from state ON to state OFF happens when a timer set to 10,000ms goes off. Typical experiential prototypes cope with experiences ranging from a few milliseconds to many seconds or minutes, which is why time is specified in ms throughout this article.

The STD is a high-level description of the behavior which is helpful for communication and documentation purposes. The same behavior will be coded in a programming language, for example the C++ dialect of Arduino, but this tends to be harder for stakeholders who are not familiar with such language. This is the code:

```
void LEDstep()
      int button;
      button = digitalRead(buttonPin);
      switch (state){
      case OFF:
            if (button == HIGH){
                state = ON;
                timer = 1000; //10000ms
                analogWrite(ledPin,255);
             }  break;
      case ON:
            if (timer <= 0){
                state = OFF;
                analogWrite(ledPin,0);
            } else timer--;
      }
}
```

This LEDstep can be viewed as a process which runs forever. When it writes 255 to the ledPin, the external light will be switched on (assuming LED light). The step is to be called every 10ms which guarantees that the timing works right. The rest of the code now follows the theatre metaphor.

```
void setup() {
      pinMode(ledPin, OUTPUT);
      pinMode(buttonPin, INPUT);
}

void loop()  {
      LEDstep();
      delay(10);//ms
}
```

The state transition diagrams are not restricted to the on and off states, but can be used for a wide variety of configurations up to tens or even hundreds of states (although I would not recommend hundreds because then it would probably be difficult to understand). As an example, consider a more sophisticated version of the hotel corridor light in which the light does not simply shut off at once, but dims smoothly during two seconds (so the guest has a few seconds to find the light button again). See Fig. 3. The implementation is done along the same lines as before.

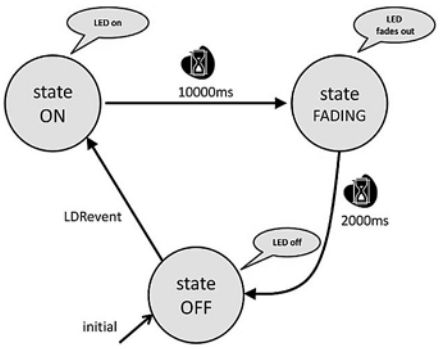

Fig. 3. State Transition Diagram with three states.

STDs have been used in engineering for a long time, see for example [3]. Harel extended them with hierarchy into a formalism called Statecharts [4], which now is also part of UML [5].

3 Data Flow Diagrams

Now the example is extended by introducing a second process. Consider a system where the light must go on whenever there is a sudden change in the external light condition. This could be used in an exhibition where the visitor approaching an object on display would create a shadow which would be detected to initiate a

demonstration sequence of the object on display. In this case the demonstration sequence is just a LED going on. A practical sensor for doing this is an LDR, a light-dependent resistor.

Of course it could be tried to include the statements reading the LDR directly into the State Transition Diagram, but I argue that it is much better to have a clear separation of concerns: there are two processes now:

- The LDR process which reads the LDR and finds out whether there is a sudden change in light intensitity;
- The LED process which drives the external output, for example the same LED as before.

Parallel processes is a powerful concept. It is essentially multitasking. But when designing the software of the prototype some communication between these processes has to be provided. I like to use shared variables for that.

In the Data Flow Diagram of Fig. 4 there are two processes, each represented by a circle. So depending on the type of diagram, a circle may be either a state (in an STD) or a process (in an DFD). Here in Fig. 4, it is a process. The variables are indicated by the box-like structures. A variable is like a small storage bin to memorize a number, a text string, or a truth value, or even a whole sequence of those. Usually there is information flowing back and forth between processes and variables. The information flows are represented by arrows. For example LEDstep writes its state into a variable called state. And LEDstep also reads and writes its timer. These could be called private variables (of LEDstep). LDRstep signals the occurrence of an event by putting the truthvalue true in a shared variable called LDRevent. This event in turn is read by the LED process.

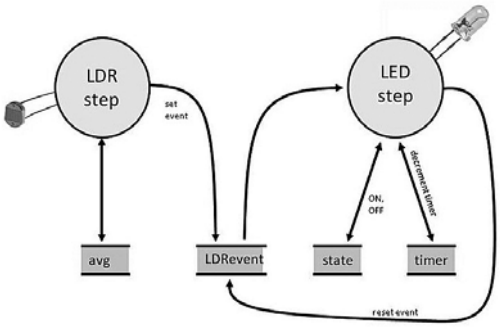

Fig. 4. Data Flow Diagram with two processes and four variables.

Note that also the LDR process has one private variable, called avg, for average. That will be explained later. Data Flow Diagrams have been made popular by Yourdon [6]

Remark: computer scientists have invented powerful alternatives for inter-process communication, such as message queues, sockets and semaphores. Although these are valuable and even indispensible for certain complex engineering problems, the simplicity of shared variables is a great asset during the development of experiential prototypes and 4D sketching [7].

4 How to implement multitasking

Continuing the example I shall demonstrate how these parallel processes are implemented effortlessly. The initialization of the variables determines how the multitasking system will begin its active life. Already when the variable is declared in Arduino language it can be given a begin value:

```
int state = OFF;
int timer = 0;
boolean LDRevent = false;
```

The LED process is still almost the same as the one of Section 2. The only additions are about resetting the event flag (i.e. the variable LDRevent). The implementation follows the Arduino theatre metaphor again:

```
void setup()  {
   pinMode(ledPin, OUTPUT);
   pinMode(ldrPin, INPUT);
   avg = (float)analogRead(ldrPin);
}

void loop()  {
      LDRstep();
      LEDstep();
      delay(10);//ms
}
```

For all practical purposes, this gives us perfect multitasking. To be honest, it is not really true parallelism: technically it is so-called interleaving parallelism. Considering the consecutive calls of delay as an "idle process", the three processes alternate quickly as follows: LDRstep is active during about 0.01

milliseconds (or perhaps even less), then LEDstep is active during about 0.01 milliseconds (or perhaps even less) and then the "idle process" takes 9.98 milliseconds. And this goes on forever in an infinite loop. So why do I call this "perfect multitasking for all practical purposes?" This is because for the human observers of this type of experiential prototype we cannot perceive such quick switching between processes anyhow (like in a movie theatre where 24 images per second give an impression of continuous motion from individual images). Moreover, our bodies are not fast enough to make movements which would go undetected by the sensor process. The light condition is checked every 10ms, which is fast enough.

Some readers may be worried by the delay statements and the time wasted by the idle process, so I ought to clarify my position about it: typical embedded processors are fast enough to drive our experiential prototypes and still have sufficient time left. For an Arduino Uno, running at 16Mhz, the processor can spend most of its time in an idle process. The repeated calls of `delay`(10) constitute this idle process, which runs in the background in parallel with the `LEDstep()` process. In the early days of computing, and sometimes even still today, idle processor time is considered to be a valuable resource not to be wasted. Mathematicians may use it to calculate digits of π, for example [8]. In industrial design I am quite happy to waste the processor's time in the repeated application of a `delay`(10) statement. Speed of 4D sketching and peace of mind regarding programming problems are my first priority. Things only become (seriously) more difficult when an Arduino has to do audio or video processing.

5 Adaptivity

One of the most important reasons for wanting experiential prototypes to do multitasking is that they have to know what is going on in their environment. Just like we, humans, a smart system, even a not so-very-smart artificial system needs to continuously monitor its environment. This need becomes more urgent when multiple modalities are involved: sensing light, sensing movement, and so on. I shall illustrate this by an example: detecting the presence of humans by an LDR. This is a cheap alternative for camera-image analysis, capacitive sensors, ultrasound distance radars etc. The obvious idea is to choose a suitable threshold value and test whether the voltage coming out of the LDR goes below or above that threshold.

We humans have an amazing adaptivity with respect to changing illumination levels: we can already see a bit at 0.002 lux on a moonless clear night sky with airglow and already much more at 0.27–1.0 lux when there is a full moon on a clear night. We work in 320–500 lux office lighting and amazingly we are not completely overloaded and saturated at 32,000–130,000 lux in direct sunlight (source: [9]). Precisely because we are so good in adapting, we hardly notice the huge differences around us consciously. That why it is tempting to assume that a fixed threshold in a technical prototype, though not perfect, may be not such a bad idea after all. In practice however, a fixed threshold works miserably and unreliably unless we exert full control over the external light conditions. But wouldn't it be great to equip the experiential prototype with adaptivity?

In this section the earlier example of the LDR-based event detection is worked out in more detail and equipped with an elementary form of adaptivity. The idea is simple: if I would know the average LDR voltage level, then I could use this average as a threshold. The LDR can be connected from +5V to pin A0 with a pull-down resistor of 22k to GND. The incoming voltages should be averaged: that is why our process LDRstep has a private variable called avg. It is a floating point number, so it can be calculated with some precision (not rounding to the nearest integer). This is how it works: if we have 2000 values, V0, V1, V2, … , V1998, V1999, say then we all know how to calculate the average: AVG = (V0 + V1 + V2 + … + V1998 + V1999)/2000. Now it is a bad idea trying to store 2000 sensor values inside an embedded processor. An Arduino Uno has only 2k RAM. But I can calculate it in steps keeping track of a "current approximation" of the average.

```
void LDRstep(){
      int v = analogRead(ldrPin);
      avg = 0.9995 * avg
          + 0.0005 * (float)v;
      float delta = 20.0;
      float dif = (float)v - avg;
      if (abs(dif) > delta)
          LDRevent = true;
}
```

So in each step, the new average is mostly equal to the old average, and adjusted a little bit by taking the most recently read value into account. The ideas of "mostly equal" and "a little bit" are made precise by weighing factors of 0.9995 and 0.0005, which are found as 1999/2000 and 1/2000, respectively. The effect is that the more recent values contribute most and that very old values are faded out. Technically, it is a so-called exponentially weighted moving average. In this case the time constant is 20s because this is 2000 times 10ms (the time between sensor samples). It has some resemblance to the idea of a moving time window, but formally it is more like a low-pass filter. And it is very memory-friendly: this one floating point number average occupies only 4 bytes.

Although the software is adaptive now, the hardware is still rigid: one LDR can only give a Voltage range sampled and analog-digital converted to a range of 0-1023 values, so in darkness the Arduino reads 0 or 1 and in full sunlight the LDR is saturated and the Arduino reads a value near 1023. An alternative would be to have a kind of shutter, like the pupil of our human eyes. After all, for us humans, it is not only the brain which adapts to varying lighting conditions, we do have special sensor "hardware" (the pupil). The hardware for my Arduino example consists of two voltage dividers with one LDR each, but with different resistors.

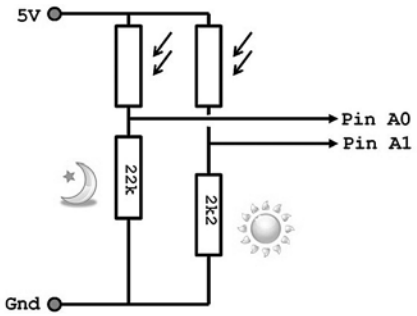

Fig. 5. Sensor circuitry for two light dependent resistors.

Now the multitasking capability developed in the previous section comes in handy. Another copy of the LDR process is all that is needed. Each process can have its own average calculation and event detection. So, when the first LDR is saturated, the other is still sensitive. Or the other way around: when the second LDR does not sense anything yet, the first is already acting. Rather than giving more program code, I summarize the architecture of this system by another DFD in Fig. 6.

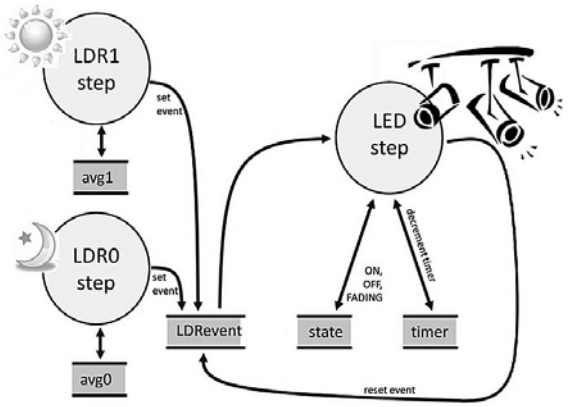

Fig. 6. Data Flow Diagram with three processes and five variables.

The loop function of the system is still extremely simple: just one more step added. The system now has three processes and one idle process.

```
void loop()  {
     LDR0step();
     LDR1step();
     LEDstep();
     delay(10);//ms
}
```

The two LDR processes are almost identical, except for their input pins and private variables. There is no extra complexity involved in adding this new parallel task. Even if I would add ten more light sensors, ten temperature sensors and ten moisture sensors, the software would keep its simplicity (I would run out of analog hardware pins long before running out of software resources).

Similar background tasks can make a system adaptive with respect to temperature, fluid levels, a user's heart rate and so on. Also matters of movement in 3D space give rise to background tasks such as calibration and navigation. A system's current position and orientation can be obtained by calculation from sensor data by methods like dead reckoning (developed in ship navigation when sailors had sextants and ship logs only – before John Harrison invented the chronometer).

6 Fine grained timing control

It is tempting to assume that the proposed approach only works with processes and timers which are rather slow, such as the 10 and 20 seconds of the examples so far. Fortunately the approach can work with fine-grained time control as well, without any problem. I shall demonstrate how to control a servo motor, as task which requires subtle timing in the sub-millisecond range.

As an intermezzo, let me explain how a servo motor must be handled. A servo motor needs commands which are repeated 50 times per second. Each "command" has to be in the form of a pulse whose width is around 1.5ms. A pulse of 0.8ms means "turn completely left" and 2.2ms means "go to the 180 degrees right position". Obviously 1.5ms tells the servo to go to the central position. In this way any position between 0 and 180 degrees is possible.

I describe the software for a system with two servos. It is a kind of medical-inspired artistic installation where one servo moves as a simulated heart beat and the other a simulated breathing. I show the DFD first in Fig. 7.

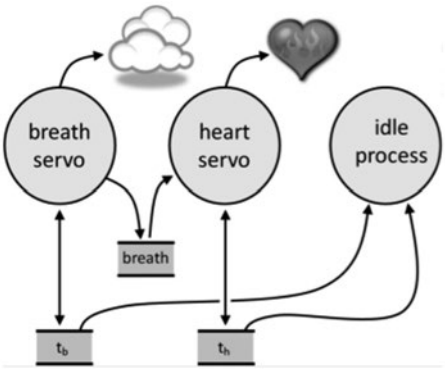

Fig. 7. Data Flow Diagram with three processes and three variables.

It is well known that the breath modulates the heart rate, a phenomenon known as respiratory sinus arrhythmia (RSA). This is modelled by the breath variable between the breath servo process and the heart servo process. Moreover the idle process has access to the fine grained time information t_b and t_h of the servo commands. The code fragment shows how this boils down to the implementation level is:

```
void breathServoStep(){
      th = … ;//between 800 and 2200
      digitalWrite(servoPin,HIGH);
      delayMicroseconds(th);
      digitalWrite(servoPin,LOW);
}

void idleStep(){
      delayMicroseconds(10000 - th);
      delayMicroseconds(10000 - tb);
}

void loop(){
      breathServoStep();
      heartServoStep();
      idleStep();
}
```

In this way I can even control the servo with an accuracy of 0.1 degree (vs. 1.0 degree for standard servo.h).

7 Outlook

My claim is that the proposed way of organizing the embedded software in experiential prototypes facilitates highly multitasking systems. Moreover these systems are much more robust and reliable than traditional naive approaches. For example the adaptivity added in Section 4 eliminates the need for hidden calibration potentiometers. It happened that prototypes which worked fine in the design studio fail at the crucial moment of the demo. *"I am sorry, but I swear, yesterday it did work"* is a story I heard often. Too often. The main reason (apart from unexcusable issues like loose wires and empty batteries) is that the prototype is not adaptive.

Nobody has the evil intention to lead unexperienced design students into the dead-end street of programs which are complicated, mono-tasking and unreliable. Typical Arduino code examples are attractive and low threshold. Yet there is danger. Let me clarify the paradox by showing an example about a servo, which comes with the usual Arduino environment (without the comment):

```
void loop()
{
  for(pos = 0; pos < 180; pos += 1)
  {
    myservo.write(pos);
    delay(15);
  }
  for(pos = 180; pos>=1; pos-=1)
  {
    myservo.write(pos);
    delay(15);
  }
}
```

In principle this is a good program. Yet there is an opportunity for misery to begin when this loop is taken as a template for an extension, include more sensors and actuators. Soon the main loop may contain all the delays, and a cocktail of nested if-else and switch-case statements for a variety of multiplied state transition diagrams. Despite a lot of apparent code complexity, multitasking is still difficult: the Arduino does one thing after the other. The approach outlined in this article prevents the paradox from happening in the first place. At the same time abstract thinking, precision and documentation get better by the proposed STD and DFD.

In TU/e ID I have seen (and sometimes contributed to) succesful examples of good multitasking in areas ranging from fashion to medical simulation. A few examples:

- The "Close-to-you" concept demonstrator and research test tool by Sibrecht Bouwstra of TU/e ID where a visual and haptic information display re-animates a (prematurely born) baby's heart rate and breathing. Multitasking, STD and DFD were all used. The example of section 4 is inspired by Sibrecht's work.
- Perceptive objects by Eva Deckers such as PeP, PeP+ and PeR, capable of sensing their environment, even perceiving presence of a person and his or her activity. The objects have a moving light body which emerges from a multitude of LEDs and are capable of perceptive interaction or even perceptual crossing. The objects are highly adaptive. DFD notation was used succesfully to manage complexity. Eva was awarded a Cum Laude for her PhD [10].
- Drapely-o-lightment, an innovative skirt in high-tech fashion design by Marina Toeters and me. Drapely-o-lightment is about OLEDs and their embedding in soft fabrics. An Arduino senses the presence of other persons and feeds several (state-dependent) light patterns into the OLEDs. Drapely-o-lightenment was shown at Architextiles in Tilburg, Place-it in Berlin, Gouden Geesten in Utrecht, Pretty Smart Textiles Ronse, Belgium, the pop-up fashion show of Amsterdam Fashion week, Smart Textiles Salon 2013 Gent Belgium, TEDx Brainport, and the Bridges Mathematical Art Galleries Enschede [11].

The multitasking approach explained here is not new, and has been used succesfully in traditional engineering projects outside the design community [12], where it is called "synchronous multitasking". It is popular in industrial automation because the programs tend to be more deterministic than programs using the interrupt-based multitasking of Windows and Linux. The determinism allows for better testing and debugging. The main contribution of this article to make such lost knowledge available to the design community again and translate it to our Arduino environment. In the multitasking approach, tasks can be put together in a multitasking way effortlessly: the adpative LDRs, the heart and breath servos etc. could be blended into a single program as simple as:

```
void loop(){
      LDR0step();
      LDR1step();
      LEDstep();
      navigationStep();
      heartServoStep();
      breathServoStep();
      // more
      idleStep();
}
```

If the reader thinks that the examples are small and simple-minded: yes, *that is the big advantage*.

References

1. Banzi, M.: Getting Started with Arduino, O'Reilly Media, 2008.
2. Arduino, http://www.arduino.cc/
3. Booth, T.: (1967) Sequential Machines and Automata Theory, John Wiley and Sons, New York
4. Harel, D.: Statecharts, a formalism for complex systems, Science of computer programming, 8 (1987) 231-274.

5. UML ® Resource Page, http://www.uml.org/
6. Edward Yourdon, http://yourdon.com/
7. Kyffin, S., Feijs, L., Djajadiningrat, T.: Exploring expression of form, action and interaction. In: Sloane, A. (Ed.): Home-Oriented Informatics and Telematics, pp. 171-192, IFIP International Federation for Information Processing Volume 178 (2005)
8. Tsz-Wo Sze: The Two Quadrillionth Bit of Pi is 0! Distributed Computation of Pi with Apache Hadoop, Computing Research Repository, Vol abs/1008.3171, 2010,http://arxiv.org/abs/1008.3171
9. Wikipedia: Lux. en.wikipedia.org/wiki/Lux.
10. Deckers, E.: Perceptive qualities in systems of interactive products. PhD thesis TU/e, May 2013.
11. Feijs, L., Toeters, M.: Drapely-o-lightment, Bridges Mathematical Art Galleries, gallery.bridgesmathart.org/exhibitions/2013-bridges-conference/feijs
12. Marchand, E., Rutten, E., Chaumette, F.: From Data-Flow Task to Multitasking: Applying the Synchronous Approach to Active Vision in Robotics, IEEE Trans. On Control Systems Technology, Vol. 5, No. 2, pp 200 – 216. (1997).