Efficient simulation of distributed sensing and control environments

Dominique Dhoutaut[1,2], Benoît Piranda[1] and Julien Bourgeois[1,2]
Université de Franche-Comté[1] - FEMTO-ST Institute[2], UMR CNRS 6174
1 Cours Leprince-Ringuet - 25200 Montbéliard, FRANCE
Email : {dominique.dhoutaut, julien.bourgeois}@femto-st.fr, benoit.piranda@univ-fcomte.fr

***Abstract*—The rise of the Internet of Things raises many challenges among which is the ability to efficiently simulate a real 3D environment with intelligent objects able to sense and act. Furthermore, the apparition of micro-objects able to communicate forces such as a simulator to scale up in the number of simulated nodes. In this paper, we report the progresses made in the design of a new kind of simulator named VisibleSim. VisibleSim mixes a discrete-event core simulator with discrete-time functionnalities in the most efficient way so that simulations can scale up in numbers. Experiments show that VisibleSim can accurately and smoothly simulate 2 millions of nodes at a rate of 650k events/sec on a simple laptop.**

I. Introduction

The history of Internet begun in the 70s and it is now a mature network widely used for various uses. Internet offers a reliable connectivity, high-bandwidth and a low latency to its users so that there is no debate on another way of connecting people and computers all around the world. However, a new way to communicate is growing outside as well as inside of Internet. The Internet of Things (IoT) federates the things that need to communicate and their requirements are different form normal computers or humans. Some things will still need high-bandwidth and low latency but most of them need low power as well as low bandwidth and latency is not an issue. New service providers are therefore appearing and deploying separate networks. One of the most promising IoT providers is SIGFox [1] who has designed a radically new network, based on its own networking cards and on its own base transmitting stations. This network allows therefore objects to be remotely monitored at a very low price.

On the application side, a new development environment is also needed to take into account the specificities of IoT: roughly, things can sense and/or act on real-world, they can be mobile and they are numerous. To the best of our knowledge no current simulator can scale up to several millions nodes, while offering integration in a real 3D environment which means sensing, control and mobility. We address this problem by proposing a new simulator called VisibleSim that is able to simulate intelligent communicating objects placed in a real 3D world.

The two main ideas of VisibleSim are the following. The first one is to mix a discrete-event core simulator with discrete-time functionnalities in the most efficient way so that simulations can scale up in numbers. The second one is to be as generic as possible so that any kind of distributed program could be easily plugged in VisibleSim. This article presents its architecture and reports preliminary experiments with distributed intelligent MEMS [2] examples.

This work has been funded by the Labex ACTION program (contract ANR-11-LABX-01-01) and ANR/RGC (contracts ANR-12-IS02-0004-01 and 3-ZG1F) and ANR (contract ANR-2011-BS03-005)

II. Contribution

A. Architecture

The design choices of the proposed simulator come from the requirements detailed here:

- **Deterministic and versatile.** As the execution path of large scale distributed applications is often very tedious to understand and debug, the simulator should be able to reproduce exactly any given scenario, and thus being deterministic. Moreover, we do not want our simulator to be limited to one particular target platform, but instead we would like it to be expandable to meet various needs. Following a very common practice in the network simulation field, our simulator is event-driven. This allows a full reproducibility of the experiments. Because it has been carefully designed this way from the start, it also intrinsically proposes an expandable model. Many basic events and object types are provided, from which programmers can easily derive their own.
- **Scalable.** It should be able to handle large numbers of elements, because new domains of application like IoT or distributed intelligent MEMS show their interest when thousands or even millions of units act together. Scalability is handled through the possibility to use more or less detailed event modeling of objects and components. This tradeoff is necessary, albeit it has to be handled carefully. This is particularly true for the networking model where a complete implementation of the protocol stack is possible but would prevent the simulator from going over a few thousands simulated components. In fact, many phenomenon can be simulated in an efficient and still correct enough way, as long as one is aware of the impact on other layers.
- **Simulate the environment.** As the envisioned applications interact with their physical environment, the relevant elements in the environment should also be simulated. The sensing and actuating simulated code should be tricked to think it is actually sensing and actuating real things, and the reactions to the simulated actuation should be realistic. This means we need a physical simulation that can be handled at a reasonable cost and with a sufficient precision. We decided

for techniques coming partly from the video game industry. Here again, a necessary tradeoff has to be made between the precision and the scalability. Our event-driven simulator core still enables various degrees of precision depending on the need of a particular application.

- **Ways to understand and analyze.** Because of the complex nature of large scale distributed applications, ways to display and better grasp their general behavior are required. Two complementary approaches are provided: a powerful 3D visualization interface and a tracing and filtering tool allowing to get to the very detail of any particular point of interest (be it interaction with the environment, code execution, networking, etc.).
- **Transparency and choice of programming model.** It should allow the use of simulated code as similar as possible to the one which would be deployed on a real implementation, ideally fully identical. Also, because different programming paradigms can be chosen by application implementers, it should ofter at least support for polling and event based models. Because of its internals, the event model is the natural behavior of the simulator as it sends relevant events to the application code which then chooses to react to it or not. An event driven application thus requires almost no additional code to run on the simulator. However a ”translation” framework is also provided for applications preferring to using polling. Using polling may be a little less straightforward, as the application implementer has to be careful of a few details. Continuously polling a sensor for a state change may be acceptable on an independent hardware implementation but is obviously not in a simulator running thousands of elements !
- **Able to cooperate.** Because upper layer softwares and applications already exist, it should be able to drive and interact with separate codes, and to make use of external dedicated codes should the need arise. Two aspects are covered here. The first one is the ability to delegate details to external software. This can be a link with a dedicated network simulator, should the need for a fully implemented network stack arise. In that case the simulator only manages the network interfaces, but the actual encapsulation, collision detection and other computations are done in a community recognized simulator such as NS2 or NS3. The second point is the ability to interact with real hardware, be it real sensors / actuators exposed to the simulated code (verifying that the code is able to handle the real thing), or real code exposed to a simulated environment (because the sensors / actuators are not available yet).

B. Simulated environments and languages

VisibleSim can already simulate four different environments working with three different languages. The design of VisibleSim presented in this section, allows plugging different languages either compiled or interpreted and the change to the physical environment has been kept as simple as possible.

1) Smart Blocks: Smart Blocks [3] is an effort to build a self-reconfigurable modular conveyor based on a contact-free technology. This conveyor is composed of centimeter-size blocks (2 cm) which are linked together to form the conveying surface (see Figure 1). Each block includes a MEMS actuator array in the upper face in order to move

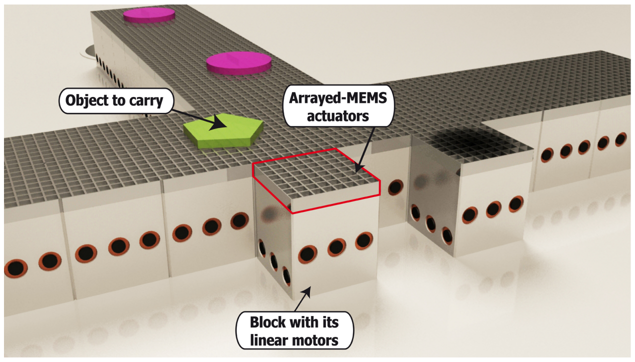


Fig. 1. Smart Blocks modular conveyor

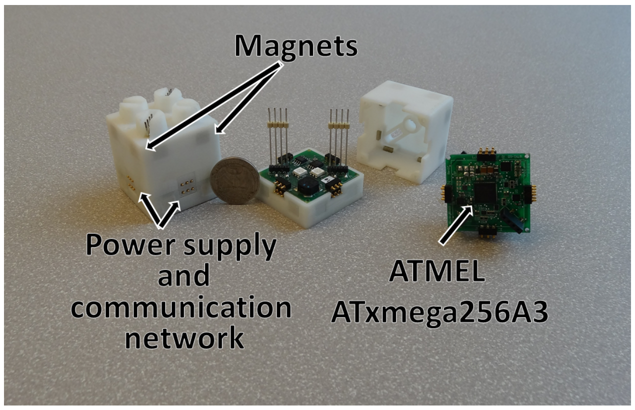


Fig. 2. Blinky blocks detailed

the objects. These dense actuator arrays can move either sub-centimeter size objects or even bigger ones. Sensors able to detect the object positions are also integrated in the upper face thanks to innovating MEMS/CMOS integration. Each block has its own processing unit as a micro-controller, and some communication ports will link it with its neighbors in order to plan global transport policies or to decide to reconfigure the shape of the conveyor in case of faulty blocks or of series change. The model of a block includes sensors, actuator array (upper side), reconfiguration actuators for mobility and 4-way communications to its neighbors. Within VisibleSim, objects are simulated and can be detected by sensors and carried out by the actuators. Each block can slide on its neighbor and the reality checker can detect wrong configurations like equilibrium of the ensemble. Each block is programmed in C using a special API.

2) Blinky blocks: A Blinky Blocks system [4] is a modular distributed execution environment composed of centimeter-size blocks (4cm each) that are attached to each other using magnets. A Blinky Block is built around an ATMEL ATxmega256A3-AU microcontroller which has 6 UARTS. Each block can therefore be serial-linked to up to 6 other blocks. The connection between the blocks also provides power supply (see figure 2). Each block has a sensor (inertial measurement unit, IMU) and it can play sounds and change its color. Two languages can be used to program Blinky Blocks, either C or Meld which follows the logic programming paradigm [5]. A Meld program is transformed into bytecode which is interpreted by a virtual machine written in C++.

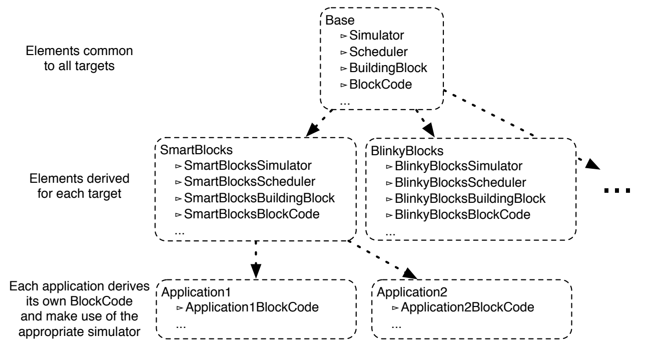


Fig. 3. Simulator architecture : multiple targets and code re-use through inheritance

3) Claytronics: The principal objective of the Claytronics project [6] is to realize the concept of programmable matter by aggregating Claytronics atoms (Catoms). This matter is envisioned to be composed of millions of Catoms, silicon millimeter-scale balls, which can turn around each other so that they can change the global shape of the ensemble. Like Blinky Blocks, Catoms can be programmed using Meld. But, despite recent advances in the manufacturing of Catoms [7], [8], they will not be mass-produced in the near future. Therefore, Meld programs cannot be executed on Catoms yet but simulations can be used. DPRSim, developed by Intel Research, has successfully supported the execution of Meld programs on millions of simulated Catoms [9] but the need for a higher precision pushed to the use of VisibleSim.

4) Multicores: The newest version of the Meld language can also be used to program multi-cores architectures. Although, this version does not need sensing and actuation, visualization of the performances have proven to be useful.

C. Simulation core

1) Multiple targets: To more easily cope with the design choices, the simulator is written in C++ and make an extensive use of the object programming paradigm. To ease the implementation of new target platforms, as summerized on figure 3, the simulator core is divided into two layers. The first layer provides a number of base objects representing components independently from any particular target. This include among others the "BuildingBlock" as the base robotic unit, network interfaces, sensors, actuators, the messages and an abstraction of the code running on a block. The second layer provides specialization for a particular target through classes derived from those of the first layer. If you consider the "BlinkyBlocks" platform for example, you then get dedicated Block (cubic shape), sensors (BlinkyBlocks can be tapped on), actuators (they have colored lights) and network interfaces (they connect through port on each of their face). Someone wanting to develop and test a new application for BlinkyBlocks thus only has to derive its own code from the provided abstraction of code running on this platform (BlinkyBlocksCode class), having to specify only a few required methods.

2) Event model: Our simulator is event driven, this means that everything is modeled as en event and is scheduled for processing. The simulator maintains an ordered list of all the events waiting to be processed. It always consumes the one at the head of this list. Consuming an event means calling its associated callback function. Depending on the event type, this function can be internal or user defined, although the application programmer is only required to override two functions : one that is automatically called when a BlockCode is initialized, and the other called when any event concerning this BlockCode occur afterwards. The user get a reference to the event occuring and then decide what to do (it could be a data packet received, data coming from a sensor, a timer, etc.)

Motion of the blocks themselves is managed through events. Basically "startMoving" event marks the beginning of the motion, ended by a "stopMoving" event. Whenever the exact position of a block is required, the simulator simply interpolates it. This approach is both very fast and precise.

During the processing of an event, it is common to schedule one or more new events, which will be processed later. It is outside of the topic of this paper to describe completely the event model. Still, the simulator core is able to ensure the correct chronology of all the events, provided the user follows the guidelines and never calls the callback functions himself.

3) Networking model: Because of the necessity to be fast enough to scale into large number of blocks, tradeoffs have been made regarding the level of detail. The networking makes use of the event model and proposes outgoing and incoming queues where the user can put and get its messages. The simulator automatically tracks the state of the provided network interfaces and discards or delays messages according to disconnections or bandwidth limitations. Messages themselves are implemented as class derived from a provided base class. An application programmer can thus easily implement any message type he may need.

4) Physical engine: Each block, through its sensors and actuators, produces interactions with the outside world. For example, a smart block may activate an electromagnetic system in order to move in relation to another block, or may open a valve enabling an air jet of the conveying system. These actions towards the physical world must be followed by a physical effect: the block has to slide along its neighbor, or a conveyed object has to move.
To standardize these actions of blocks towards the real world, we consider that the actuators will produce forces on external objects. Then, a special part of the simulator, called physical engine, deals with the effects of these forces on all the other objects. For the simulator, all objects can be considered as physical objects, according to their capabilities of mobility, they will be considered as mobiles or obstacles.

The physics engine must simulate all the motions of objects in the environment that are subjected to forces by applying rules of Newtonian physics. We consider that forces are applied at the center of gravity of objects, the physics engine then infers accelerations, velocities and inertia, simulating real movements of objects.
Depending on the simulated blocks, the physical engine is used to determine the dynamic effects of the application of forces on moving objects (see dynamics paragraph) or to determine the stability of the mechanical system maintained by contact forces. Moreover, in the case of smart blocks, blocks are provided with pneumatic conveyors for moving objects on their surface. In this configuration, conveyed objects are managed

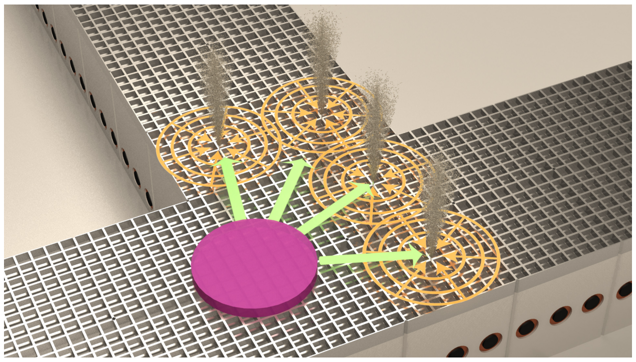


Fig. 4. Forces produced by air jet on a mobile object.

by the physical engine.

In the case of 3D blocks, all blocks are considered as mobile by the physical engine. A work in progress aims at developing algorithms able to verify the stability of a set of blocks using two verifications. One on the stability of links between blocks and another one on the balance of the overall structure. In this problem, the physical engine calculates the forces between each pair of blocks. In the case of Blinky Blocks, the physical engine is not currently used, as real Blinky Blocks are arranged manually and have no mechanical actuators.

a) Simulation of the dynamics: The simulation of the physical dynamic consists in simulating as precisely as possible the actual movement of simple objects (called mobile object) subjected to forces produced by the pneumatic conveying system. In the Smart Blocks project, mobiles are conveyed on the upper surface of the blocks by a pneumatic system. This system offers the dual capabilities of carrying and moving the mobile objects by applying vertical and tangential forces using two kinds of air jets. Vertical forces are used to prevent friction on moving conveyed objects, like on an air-hockey table. Tangential forces are produced by high speed air-jets, that create a local suction effect to slide mobile objects. The generated force is oriented in the direction of the air-jet. Delettre et al presents the model of this pneumatic actuator in [10].

In the Smart Blocks project, each block is able to activate an air-jet, which will apply a force to each mobile object nearby. As shown in figure 4, attraction forces $\vec{F}_{air}$ produced by air-jets are directed towards the mobile object (green arrows) and its norm depends on the distance between the mobile and the air-jet source. The motion of the mobile object is obtained in combining many air-jet of nearby blocks.

The physical simulator engine performs calculations to determine mobile objects movements applying the second Newton law: $\sum \vec{F_i} = m \times a$

Attractions forces produced by air jets are calculated using the formula proposed by [10], moreover additivity properties have been verified by the authors.

$$\vec{F}_{air} = \iint -b\vec{U_P}ds \tag{1}$$

In the case of a thin parallelepipedic shape, the previous equation may by simplified in:

$$\left\{ \begin{array}{lcl} \vec{F}_{air,x} & = & \sum_{i=1}^{N} -\frac{\Lambda_i \times b}{2\pi}\left[f_1 \cos\left(\theta+\phi\right) + f_2 \sin\left(\theta-\phi\right)\right] \\ \vec{F}_{air,y} & = & \sum_{i=1}^{N} -\frac{\Lambda_i \times b}{2\pi}\left[f_1 \sin\left(\theta+\phi\right) - f_2 \cos\left(\theta-\phi\right)\right] \end{array} \right. \tag{2}$$

where $\Lambda \times b$ is considered as a physical constant and f_1 and f_2 are two functions that depend on the dimensions of the mobile object and its relative position with the jet.

Then, we use the Euler integration method to solve this system of differential equations:

$$\left\{ \begin{array}{lcl} m\ddot{x} & = & \vec{F}_{air,x} - bl_x l_y \dot{x} \\ m\ddot{y} & = & \vec{F}_{air,y} - bl_x l_y \dot{y} \\ I\ddot{\alpha} & = & \Gamma_{air} - \frac{b\left(l_x^3 l_y + l_x l_y^3\right)}{12}\dot{\alpha} \end{array} \right. \tag{3}$$

In order to validate our physical engine, we have compared simulated and real behaviors. The reference motion of a thin parallelepiped object was captured in the air table of the Smart Surface project. We obtain very similar results with the simulation as shown in the figure 5.

b) Reality checker: The reality checker applies a static physical simulation in order to verify the mechanical stability of a set of 3D blocks. Considering a configuration of 3D blocks, this checking is in two parts :

- A block may have to support others as it can be seen in figure 6a. Linkage forces between the blocks must be compared to the weight forces to ensure that the configuration is physically possible. In the case shown in figure 6a, is the yellow block in the center able to hold the three blocks on the left?
- Before moving a block, we have to verify the stability of the system. It is even more important when planning the

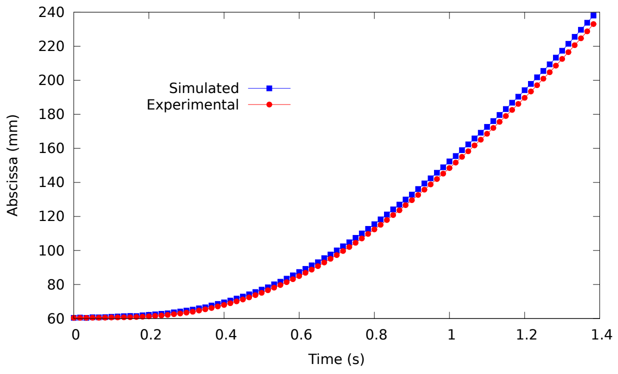


Fig. 5. Comparison of the simulated and the experimental position of a parallelepiped object moving on the Smart surface.

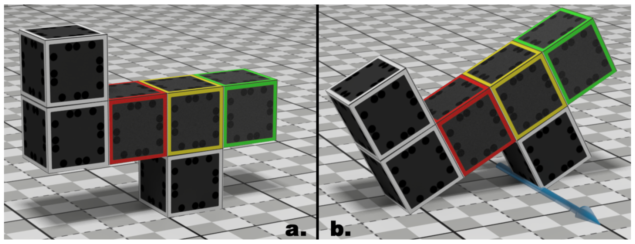


Fig. 6. Tasks of the reality checker: a. linkage forces; b. global stability.

motion for the set of blocks to reach a new configuration. The reality checker has to calculate the torque for many pivot axes and eventually deduces a rotation of the set of blocks as shown in figure 6b. In our case, pivot axes are in contact areas between blocks and the floor.

The implementation of the reality checker is currently under development.

D. User interface

The simulator interface allows both the observation of the simulated scene under many points of view and the interrogation of the status of blocks and system informations in order to debug the BlockCode during development.

For the SmartBlocks, the simulator provides the opportunity to know the internal memory state of each micro-robot in real time during the simulation, along with an history of all messages exchanged between blocks and all outputs written in the BlockCodes. The interface allows the selection of one block by clicking on it, then displays a filtered list of events for this block. The user interface comprises two parts, as shown in figure 7:

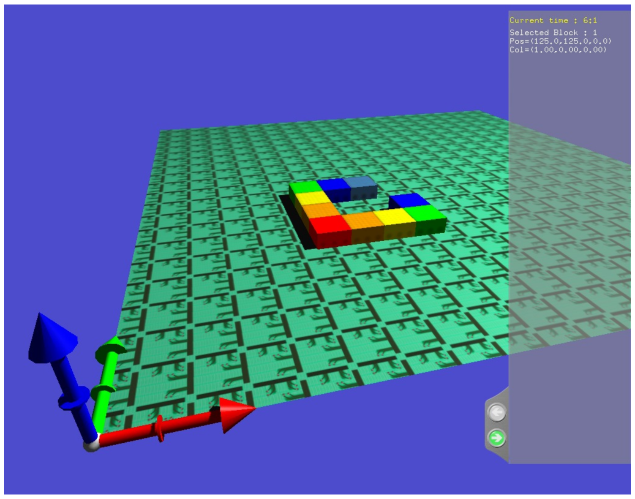
- The main area contains the 3D representation of the set of blocks, this area is as big as possible in order to show the biggest possible part of the scene.
- The text window presents traces of the simulation. It contains information about the selected block. This window can be hidden by clicking on an icon.

The scene can be shown under free point a view, turning the camera, and zooming with the mouse. If a block is selected, the focus of the camera may be automatically placed on this block, it's also possible to freely change the camera target position with a mouse action.

c) Interactive actions on blocks: The implementation of a large number of blocks causes a significant clutter of the 3D interface. To easily check the behavior of the program on all blocks, we must be able to distinguish information related to a particular block. VisibleSim interface offers different tools to do that. First, when you hover the mouse cursor over a block, its number appears in a popup window. Then when a block is selected, it flashes and you can read in the sliding window on the right all the user specified information from the $GetInfo()$ function overrided in the BlockCode, along with all the messages that have been sent and received by this block.

In the BlockCode, creating a debugging trace is performed by calling a $trace$ function. To ease debugging, the format and content written in the sliding window are simply defined by overloading a class method in the BlockCode. Another way to debug BlockCode consists in changing the color of the blocks. BlockCode functions $color(R, G, B)$ or $color(id)$ allow to change the color of the associated block by specifying the RGB components or the id of the color in a pre-defined color list.

d) XML configuration file: We have previously seen how VisibleSim defines the behavior of the code running within each block. In order to describe the 'world context' of the experiment, we create a XML configuration file that



```
Current time : 6:1
Selected Block : 1
Pos=(125.0,125.0,0.0)
Col=(1.00,0.00,0.00)
```

Fig. 7. Screenshot of the interface.

enumerates its constituent elements: blocks, mobiles, obstacles, camera, spotlight etc. For each of these elements we precise their position and settings. The following example shows a XML configuration file specifying the position of the camera, the spot light and the list of all blocks with their initial color and position (it corresponds to the configuration shown in figure 7).

```
<?xml version="1.0" standalone="no" ?>
<world gridsize="20,20">
 <camera target="125,200,0" directionSpherical="0,70,500" angle="45"/>
 <spotlight target="250,250,0" directionSpherical="45,60,500" angle="40"/>

 <blockList color="0,255,0" blocksize="25.0,25.0,11.0" obj="hdmodel.obj">
  <block position="5,5" color="0,128,128"/>
  <block position="5,6"/> <block position="6,5"/>
  <block position="5,7"/> <block position="7,5"/>
  <block position="5,8"/> <block position="6,8"/>
  <block position="7,8"/> <block position="8,5"/>
  <block position="8,6"/>
 </blockList>
</world>
```

e) 3D representation: In order to help us to validate the behaviors of the blocks according to their relative position, color and inter-connections, the simulator has to geometrically represent all of them.
Each kind of blocks is associated to a geometrical description. Such a geometry can easily be modeled using a classical 3D modeler (3DS Max or Blender) and exported as an 'OBJ' description file. Depending on the application to simulate, we can use different 3D versions of blocks, as shown in figure 8. This allows for example to use blocks with a more precise geometry when they are few. On the other hand, we use much simpler textured blocks when the representation of a large number of blocks may slow the display speed. In this second case, to simplify and optimize the rendering, the 3D model is reduced to some triangles that have to be textured by a single image that describes the details of the models. For each 3D model of block, we have to define which polygons may change their color. We have defined a material called 'lighted', that is applied to these particular faces.

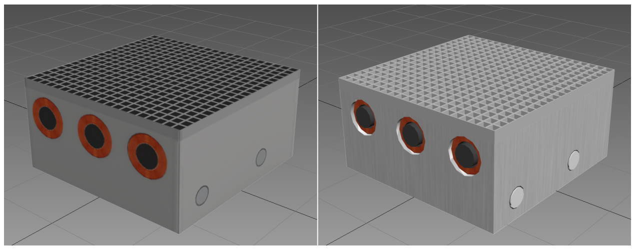


Fig. 8. The smart-block 3D model in two different levels of details.

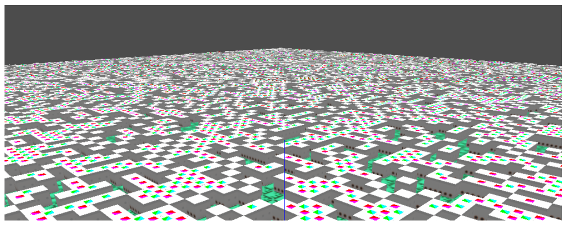


Fig. 9. Benchmarking the simulation core : thousands of blocks moving and communicating

Particular care has been taken to render the blocks with the illumination model processing the shadows of the blocks on the rest of the decor. The shadows not only provide a visual effect of realism, they are helpful to visualize the relative position of blocks in 3D. Shadows are calculated in real-time using a shadow mapping algorithm [11] implemented in a per pixel lighting shaders programs. In the shadow mapping algorithm introduced in [12], shadows are created by testing whether a pixel is visible from the light source or not, by comparing it to a depth image of the light source's view, stored in the form of a texture. The per pixel lighting method generates more regular lighting effects than the classical OpenGL per vertex lighting. It produces more precise lighting areas due to a pixel dependent lighting calculation [13].

f) Immersion in augmented reality: A work in progress aims to reconstruct the relative position of a set of real blocks using a commercial 3D sensor (Microsoft Kinect or Asus Xtion). The recording or measurement of the actual configuration is made by presenting the real Blinky Blocks to the sensor and progressively turning them so that they are seen from all angles by the 3D camera. This processing aims to automatically generate the XML configuration file for VisibleSim, listing all the blocks with their coordinates in space relatively to the first detected block.

In a second step, the real object is again presented to the sensor, so it can be quickly detected. This real-time video is supplemented by an overlay of synthesized images providing additional information on the blocks, it is called augmented reality. In the case of Blinky Blocks, this technology will allow us to provide debugging information (eg the state of a variable) in real time and directly on the image of the real Blinky Blocks captured by the camera.

g) Multimedia output: The simulator proposes many kinds of outputs. A single snap-shot capture of the scene without the interface or the video of the running of a simulation. An other output solution is given by a Collada file export that allows to generate a high quality synthesis image or video using a classical rendering software like 3DS Max or Blender.

III. Related works

VisibleSim is related to three types of simulators. The first one comprises the network simulators which are either commercial like OPNET [14] and QualNet [15] or freely available and used by the research community like NS2 [16], NS3 [17], OMNeT++ [18], SSFNet [19] or J-Sim [20]. All of these simulators have pros and cons but they can't be used directly for simulating a real environment and they don't scale in numbers of simulated nodes. Furthermore, integrating real programs is not easy and limits even more the scalability.
Modular robots or simply robots simulators belong to the second category. While certain simulators are dedicated to a specific hardware like Molecubes[21] or CrossCube [22] and therefore not usable in another context, others are more generic. For almost all the latter like Player/Stage [23], gazebo [24], USSR [25] or MORSE [26], network access is not properly modeled and they offer limited scalability. One exception to scalability is DPRSim [27], [9] which scales up to millions of simulated nodes but sacrify precision.
The third category belongs to networked control systems with simulators like RTSIM [28], Syndex [29] or TrueTime [30] which are good candidates for modeling real-time distributed systems. TrueTime is by far the most advanced project in this area. The network modeling is not as detailed as in the network simulators but it offers a better integration of latency-related aspect through the use of co-simulation between the network and the computation nodes. However, as TrueTime is based on MatLab, its expressivity is limited and integrating real programs is difficult.

IV. Experiments

A. Validation and Performance benchmark

To validate our simulator and verify that we meet the design requirements, we first build a basic but scalable scenario. This scenario sees an increasing number of blocks (from a few hundreds to a few millions) engaging in communications and movements. The movement pattern does not seek to mimic a real world application, but rather to stress the event processing capabilities of the simulator. We thus decide for a Brownian motion: Each block tries to randomly move one step in a randomly chosen direction. After reaching its destination, it waits for a random duration and then moves again. A 1-step movement is configured to last one second, and the waiting time is in average 0.5 second. At the same time, a block hand-picked as an initiator will broadcast information to its neighbors. This message is then retransmitted hop by hop, triggering a network flooding. Of course this simple algorithm is implemented as a BlockCode and is executed independently for each of the many blocks simulated. These combined movements and communications translate in a very large number of events as we increase the number of blocks. Figure 9 shows the visual output of this scenario, where the user can freely move around and know the state of the network links through graphic indicators.

We run this basic scenario multiple times, with a block number ranging from 7,000 to 2,000,000. Each time, we simulate 10 minutes of execution on a pretty standard computer

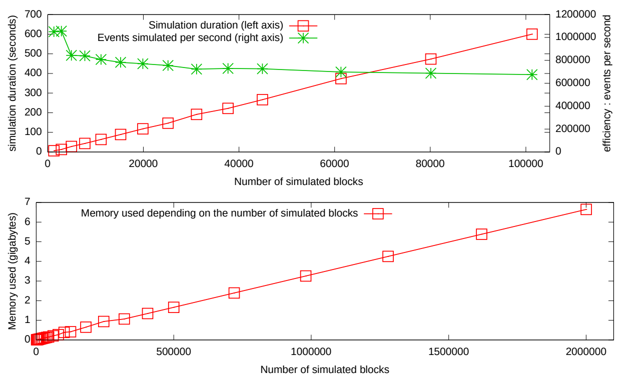


Fig. 10. Simulation time and memory usage depending on the number of blocks and events

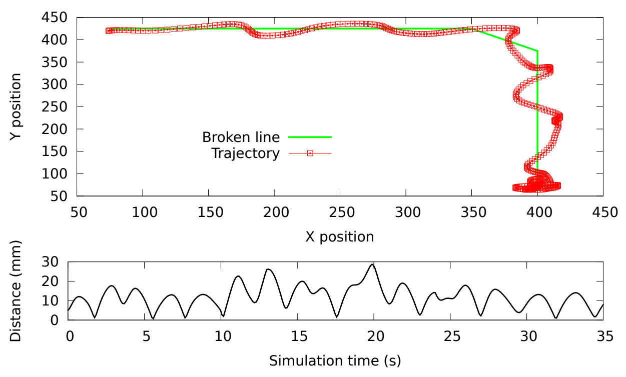


Fig. 11. Upper: Goal path and simulated trajectory. Lower: distance between the mobile object and the desired position during the simulation.

(2.56 GHz xeon processor, with 12 GB of ram). Figure 10 shows on its upper part how the simulation time increases almost linearly with the number of events to process (in this scenario, the number of events is directly proportional to the number of blocks). When the number of events in the waiting queue increases, we observe an initial drop of the performances (from around 1 million to a little above 600,000 events per second), which then stabilizes. Please note that the 100,000 blocks scenario triggered a little over 400 millions events and was processed in 10 real minutes on our computer, still making it an almost real-time simulation. The lower part of figure 10 shows the memory usage, which almost linearly scales with the number of blocks and the number of events in the queue at a given time. We conducted simulation with up to 2 millions of blocks, using a little over 6.5 GB of memory. Please note that for the largest scale simulations, our entry level 3D video card was not able to keep up with the number of blocks to display and we had to desactivate the display. The 3D interactive visualization is anyway much more useful with smaller number of blocks where individual behaviors need to be better understood. For very large scale simulations, the software is still able to produce a frame by frame animation showing the global behavior of the system.

B. *Physical simulation*

The main algorithmic problem associated with smart blocks is the conveying of parts. We consider a simple program embedded in the Smart Blocks which aim is to move a parallelepiped along a pre-determined path. For the studied mobile we define a simple path (green line in figure 11) built as a broken line passing through the following points :

t (s)	x (mm)	y (mm)
0	75	425
10	350	425
13	400	375
24	400	75

For each of these points we specify the time (in seconds) at which they must be reached.

The proposed BlockCode algorithm is divided into two steps. It first sends a message to each block containing the path to follow and the start time of the simulation ($animationTime$). Then each block waits for a message or event. If its sensors detect that the mobile is over, it compares the position of the mobile with the position it should occupy in the path at this time. According to this distance, it sends a message to some of its neighbors in order to activate their air jets and move the mobile in the direction of the required position. The following code shows this second part of the algorithm.

```
// t = animation time
t = currentTime-animationTime;
if (t>0) {
  if (detectMobile(1,x,y)) {
// calculate the current target position in (gx;gy)
    trajectory->getInterpolatedPos(t,gx,gy);
// calculate the direction from the mobile to the
    target position (vx;vy)
    vx = gx-x;    vy = gy-y;
    d = sqrt(vx*vx+vy*vy);
    vx/=d;    vy/=d;
    if (vx>0 && block->neighborEast != NULL) {
// send a Message to East neighbor
      Scheduler::schedule(new SendEventMessage(
          currentTime, new MessageJet(vx*1000),
          block->networkInterfaceEst));
    }
    if (vx<0 && bloc->neighborWest != NULL) {
// send a Message to West neighbor
      Scheduler::schedule(new SendEventMessage(
          currentTime, new MessageJet(-vx*1000),
          block->networkInterfaceWest));
    }
    if (vy>0 && bloc->neighborNorth != NULL) {
// send a Message to North neighbor
      Scheduler::schedule(new SendEventMessage(
          currentTime, new MessageJet(vy*1000),
          block->networkInterfaceNorth));
    }
    if (vy<0 && bloc->neighborSouth != NULL) {
// send a Message to South neighbor
      Scheduler::schedule(new SendEventMessage(
          currentTime, new MessageJet(-vy*1000),
          block->networkInterfaceSouth));
    }
  }
}
```

The graph presented in figure 11 represents the distance between the block and the desired position during the first 35 seconds of the simulation. The oscillations are mainly due to the inertia of the parallelepiped. But these oscillations remain at a low amplitude, and may be reduced by introducing the management of the current speed of the parallelepiped in the algorithm.

V. Conclusion

We presented a simulator called VisibleSim which targets intelligent objects and/or robots. From its core, VisibleSim is built to handle various target platforms, such as Smart Blocks, Blinky Blocks and many others. The aim is to share a maximum of codes between platforms in order to reduce the cost of new developements. VisbleSim is a discrete-events simulator which offers a great precision and the possibility to get high speed of simulation. This is illustrated with benchmarks showing that millions of independent moving and communicating blocks have been handled in almost real time by a normal computer. The paper also demonstrated the ease of extension through realistic air jets and physical constraints along with dedicated control applications. Last but not least, we emphasized the benefit of the interactive 3D interface easing the development of new applications. We now intend to extend the communications models proposed, especially adding a generic wireless communication core.

References

[1] "Sigfox: one network, a billion dreams," http://www.sigfox.com/, 2013.

[2] J. Bourgeois and S. Goldstein, "Distributed intelligent mems: Progresses and perspectives," in *ICT Innovations 2011*, ser. Advances in Intelligent and Soft Computing, L. Kocarev, Ed. Springer Berlin / Heidelberg, 2012, vol. 150, pp. 15–25.

[3] S. Mobes, G. J. Laurent, C. Clevy, N. L. Fort-Piat, B. Piranda, and J. Bourgeois, "Toward a 2d modular and self-reconfigurable robot for conveying microparts," in *Proceedings of the 2012 Second Workshop on Design, Control and Software Implementation for Distributed MEMS*, ser. DMEMS '12. IEEE Computer Society, 2012, pp. 7–13.

[4] B. T. Kirby, M. Ashley-Rollman, and S. C. Goldstein, "Blinky blocks: a physical ensemble programming platform," in *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*, ser. CHI EA '11. ACM, 2011, pp. 1111–1116.

[5] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell, "A language for large ensembles of independently executing nodes," in *Proceedings of the International Conference on Logic Programming (ICLP '09)*, July 2009.

[6] S. C. Goldstein, T. C. Mowry, J. D. Campbell, M. P. Ashley-Rollman, M. De Rosa, S. Funiak, J. F. Hoburg, M. E. Karagozler, B. Kirby, P. Lee, P. Pillai, J. R. Reid, D. D. Stancil, and M. P. Weller, "Beyond audio and video: Using claytronics to enable pario," *AI Magazine*, vol. 30, no. 2, July 2009.

[7] M. E. Karagozler, A. Thaker, S. C. Goldstein, and D. S. Ricketts, "Electrostatic actuation and control of micro robots using a post-processed high-voltage soi cmos chip," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2011.

[8] M. E. Karagozler, "Design, fabrication and characterization of an autonomous, sub-millimeter scale modular robot," Ph.D. dissertation, Carnegie Mellon University, 2012.

[9] M. P. Ashley-Rollman, P. Pillai, and M. L. Goodstein, "Simulating multi-million-robot ensembles," in *ICRA*, 2011, pp. 1006–1013.

[10] A. Delettre, G. J. Laurent, N. L. Fort-Piat, and C. Varnier, "3-dof potential air flow manipulation by inverse modeling control," in *CASE*, 2012, pp. 930–935.

[11] R. Fernando, S. Fernandez, K. Bala, and D. P. Greenberg, "Adaptive shadow maps," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '01. ACM, 2001, pp. 387–390.

[12] L. Williams, "Casting curved shadows on curved surfaces," in *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '78. ACM, 1978, pp. 270–274.

[13] D. Shreiner *et al.*, *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Addison-Wesley Professional, 2009.

[14] "http://www.opnet.com/products/modeler/."

[15] "Qualnet simulator," http://web.scalable-networks.com/content/qualnet.

[16] "The network simulator - *NS2*," http://www.isi.edu/nsnam/ns/.

[17] T. Henderson, S. Roy, S. Floyd, and G. Riley, "ns-3 project goals," in *Proceeding from the 2006 workshop on ns-2: the IP network simulator*. ACM, 2006, p. 13.

[18] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, ser. Simutools '08. ICST, 2008, pp. 60:1–60:10.

[19] D. M. Nicol, B. Premore, and A. Ogielski, "Using simulation to understand dynamic connectivity at the core of the internet," in *Proceedings of UKSim 2003*, Cambridge University, England, April 2003.

[20] J. Kačer, "J-Sim – a Java-based tool for discrete simulations," in *Proceedings of the 23rd International Autumn Colloquium ASIS-2001: Advanced Simulation of Systems*. Náměstí Msgre Šrámka 6, 70200 Ostrava, Czech Republic: MARQ, September 2001, pp. 135–141.

[21] V. Zykov, P. William, N. Lassabe, and H. Lipson, "Molecubes extended: Diversifying capabilities of open-source modular robotics," in *IROS-2008 Self-Reconfigurable Robotics Workshop*, 2008.

[22] Y. Meng, Y. Zhang, and Y. Jin, "Autonomous self-reconfiguration of modular robots by evolving a hierarchical mechanochemical model," *Computational Intelligence Magazine, IEEE*, vol. 6, no. 1, pp. 43 –54, feb. 2011.

[23] B. Gerkey, R. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th international conference on advanced robotics*, vol. 1. Portugal, 2003, pp. 317–323.

[24] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, vol. 3. IEEE, 2004, pp. 2149–2154.

[25] D. Christensen, D. Brandt, K. Stoy, and U. Schultz, "A unified simulator for self-reconfigurable robots," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. IEEE, 2008, pp. 870–876.

[26] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan, "Modular open robots simulation engine: Morse," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, may 2011, pp. 46 –51.

[27] B. D. Rister, J. Campbell, P. Pillai, and T. C. Mowry, "Integrated debugging of large modular robot ensembles," in *ICRA*, 2007, pp. 2227–2234.

[28] L. Palopoli, L. Abeni, G. Buttazzo, F. Conticelli, and M. Di Natale, "Real-time control system analysis: An integrated approach," in *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*. IEEE, 2000, pp. 131–140.

[29] Y. Sorel, "From modeling/simulation with scilab/scicos to optimized distributed embedded real-time implementation with syndex," in *Proceedings of the International Workshop On Scilab and Open Source Software Engineering, SOSSE'05*, Wuhan, China, Oct. 2005.

[30] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén, "How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime," *IEEE Control Systems Magazine*, vol. 23, no. 3, pp. 16–30, Jun. 2003.