Extensible 3D Simulation of Aggregated Systems with FCPP

Giorgio Audrito[1][0000−0002−2319−0375], Luigi Rapetta[1], and Gianluca Torta[1][0000−0002−4276−7213]

Università degli Studi di Torino, Torino, Italy
{giorgio.audrito,gianluca.torta}@unito.it,
luigirapetta@libero.it

Abstract. Programming massively distributed systems in unreliable environments poses several non-trivial challenges. Such systems need to be able to adapt and self-organise, and special algorithms need to be developed for this purpose. In particular, simulators provide an irreplaceable tool for the development process.
Among other tools for programming self-organizing systems, the FCPP library stands out for its efficiency, portability and extensibility, and its support for aggregate programs. On the other hand, the simulator's output was limited up to now to numeric statistical information, reducing the user's ability to understand and interact with the system under simulation.
In this paper, we present a novel graphical user interface for FCPP, allowing for a real-time, interactive and three-dimensional visualization of the simulated system. Through this interface, the user can control the simulation flow, visualize summary information of the network at a single glance, and inspect detailed information via auxiliary windows. The interface is designed to require minimal effort from the end user for its setup, and can be further extended for increased interaction.

Keywords: Distributed computing · Aggregate computing · Toolchains.

1 Introduction

Human environments are increasingly populated by situated and mobile computing devices (phones, watches, vehicles, sensors, smart home appliances), and the problem of their coordination into meaningful distributed systems is therefore growing in importance as well as in complexity.

Implementing such distributed systems by individual programming of every single node has become prohibitively costly and error prone, driving the search for solutions increasing the autonomy of computing systems while reducing their overall complexity. In particular, the class of *Collective Adaptive Systems* (CAS), which are able to autonomously adapt their internal structure and function in response to external events, has the potential to meet the coordination challenges outlined above.

However, programming CAS in an effective way is in itself a non-trivial task. *Aggregate Programming* (AP) [5] is drawing an increasing attention as an effective approach to program distributed, situated CAS, based on the functional composition of reusable blocks of collective behaviour, with the goal of effectively achieving resilient complex behaviours in dynamic networks. Functional composition is key in enabling the study of properties like self-stabilisation [22] and density independence [6] on complex CAS starting from simple building blocks (e.g., broadcast, collective distance estimation and data aggregation).

Few different implementations of the AP paradigm exist: the Java external DSL Protelis [19], the Scala library Scafi [24] and the C++ library FCPP [2]. Among them, the latter stands out for its portability and high efficiency, obtained without sacrificing ease of use and extensibility. The main strenghts of FCPP are its component-based architecture, which enables extensibility, the widespread support for C++ on target architectures, which allows deployments on most systems (including microcontrollers), and its speed and memory requirements compared to the other JVM-based implementations.

A shortcoming of the simulation support of FCPP has been up to now the lack of a graphical interface. While it was possible to configure scenarios with hundreds of nodes, and run an AP program on each of them while keeping track of aggregate statistics on resulting node attributes, the dynamic evolution of such systems could not be visualized. However, such a visualization is often very important to get an intuitive idea of the system behavior, especially for debugging and spotting unexpected global emergent behaviors.

In this paper, we present an extension to FCPP defining new components for the 3D visualization of simulations, also providing a basic support to the simulation of environments with obstacles. Through the new interface, the user can control the simulation flow, visualize summary information of the network at a single glance, and inspect detailed information via auxiliary windows. Interestingly, as we shall demonstrate below, the new components perfectly fit within the existing architecture, showing that the component-based approach adopted for FCPP does indeed make it easier to extend it with new functionalities. A video demo of the FCPP GUI has also been made available.[1]

The paper is structured as follows. Section 2 presents the background, including the FCPP and OpenGL libraries. Section 3 discusses the extension of FCPP for 3D simulations. Section 4 shows four case studies demonstrating the new features. Finally, Section 5 concludes with plans of future development.

2 Background

2.1 Collective Adaptive Systems

Collective Adaptive Systems (CAS) are collections of intelligent agents able to automatically change their internal structure (i.e., the connections between their components) and/or their function in response to external inputs.

[1] `https://www.youtube.com/watch?v=zWsNqJMVxKs`

Due to such characteristics, programming CAS presents peculiar challenges, that must be addressed in ways that depend on the given context and goals. At a sufficient level of abstraction, the various approaches can be classified in four categories [4]:

- methods that abstract devices and/or networks, such as TOTA [16] and MapReduce [11];
- methods that provide geometrical or topological pattern languages, such as the *self-healing geometries* in [14];
- methods that provide languages for information retrieval and routing, such as TinyLime [10];
- general space-time computing models, such as StarLisp [15] and Aggregate Programming [5].

2.2 Aggregate Programming

Among the approaches previously mentioned, in this paper we focus on *Aggregate Programming* (AP). AP is characterized by the definition of a single program that is executed asynchronously in each node of a whole network. The networked system is thus modelled as a single aggregate machine, which manipulates collections of distributed data called *computational fields*.

Communication between devices is realised at low level through proximity-based broadcasts, which at a higher level generate *neighbouring fields*, i.e., maps from neighbour device identifiers to their relative values. Neighbouring fields cannot be accessed directly; instead, they are manipulated through "map" operations that produce a new field, and "fold" operations that synthesize a single value from a field.

Such aggregate computations can be expressed using the *Field Calculus* (FC) [3], which is a minimal universal language based on the functional paradigm. FC provides the necessary mechanism to express and functionally compose distributed computations neglecting low-level aspects such as synchronisation, delivery of messages between devices, and even the number and physical positions of the devices in the network. An FC program `P` running on every device δ of the network, executes the following steps periodically:

- the device perceives contextual information, i.e.:
  - data provided by local sensors,
  - local (state) information stored in the previous round,
  - messages received from neighbours after the previous round.

  As said above, the latter are made available to the program `P` as a *neighbouring field* ϕ;
- the device evaluates the program `P` considering as input the contextual information gathered as described above. Note, therefore, that `P` is not only executed by each device, but also at each round: when needed, different behaviors are obtained by branching statements in `P` based on the input context;

– the result of the local computation is stored locally (as local state), sent to neighbours and may produce outputs fed to local actuators.

The above steps, executed across space by different devices, and across time at different rounds, give rise to a global behaviour at the overall network-level [23] that can thus be viewed as a single aggregate machine.

While the neighbouring relation is usually based on physical spatial proximity, it is perfectly possible to define it as a logical relationship, for example as a master-slave relationship among devices independently of their position.

2.3 FCPP

FCPP (FieldCalc++) is a library written in the C++ language that implements the Field Calculus (FC). Given the goal of being able to deploy FCPP on as many platforms as possible, C++ has been chosen as the implementation language not only for its power and efficiency, but also because it can target most platforms, form microcontrollers to GPUs. Beside providing an internal DSL for expressing FC programs within C++, the library provides several features:

– a component-based software architecture, suitable to be extended and customised for different application scenarios, such as deployments on IoT devices, simulation, and HPC;
– an efficient implementation exploiting compile-time optimisations through advanced template programming [1];
– support for parallel execution of a simulated system or self-organising cloud application;
– tools for executing FC programs on simulations of distributed systems.

The only scenario that is currently fully supported by the implemented components of FCPP is the simulation of distributed systems. Compared to the alternative implementations of FC (Protelis [19] and Scafi [24]), it features additional simulation capabilities (3D environments, basic physics, probabilistic wireless connection models), with a significant reduction of the simulation cost, and a corresponding speedup of the development and test of new distributed algorithms. Moreover, thanks to the extensible architecture, it is much easier to address additional scenarios than with previous FC implementations. Two such scenarios are of particular practical interest:

– deployments on microcontroller-based systems typically used in IoT applications, which have limited computing power and memory;
– deployments as self-organising cloud applications, which require fine-grained parallelism in order to be able to scale with the resources allocated in the cloud.

The *graph connector* and *spawner* components for self-organising cloud applications are currently in a preliminary testing phase, as well as the *hardware connector*, *identifier* and *logger* components for physical deployments, together with drivers for the Contiki [12] and Miosix [21] real-time OSes.

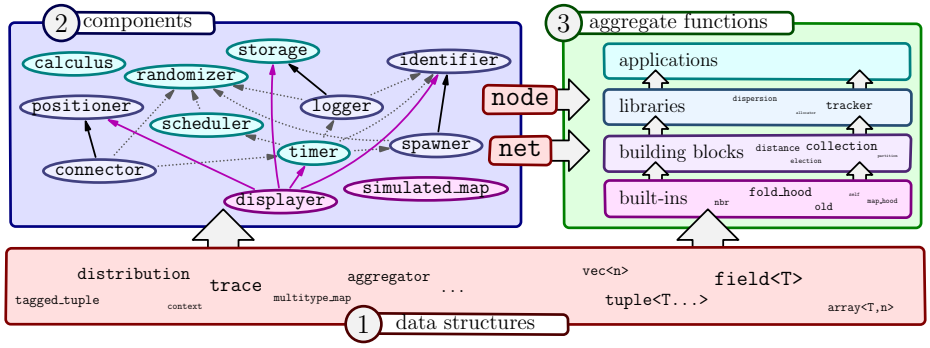

Fig. 1. Representation of the software architecture of FCPP as the combination of three main layers: *data structures* for both other layers, and *components* which provide node and network abstractions to *aggregate functions*. Components are categorized as general purpose (cyan), used across different domains, and simulation-specific (violet), with variations for different domains. The new *displayer* and *simulated map* (simulation-specific) components are highlighted in magenta. Dependencies between them can be either *hard* (solid), for which the pointed component is required as an ancestor of the other; or *soft* (dotted), for which the pointed component is not required, but if present, it should be an ancestor of the other component.

Figure 1 shows the architecture of the FCPP library, partitioned in three main conceptual layers:

1. *C++ data structures of general use*. Some are needed by the components of the second layer either for internal implementation or for the external specification of their options; other data structures are designed for implementing the aggregate functions of the third layer.
2. *components*. They define the abstractions for representing single devices (*node*s) and the overall network (*net*), which is fundamental in scenarios where there is no physical network, such as simulations and cloud-oriented applications. It is worth noting that, in an application based on FCPP, the two types `node` and `net` are obtained through template programming by combining a chosen sequence of components [17], each of them providing a needed functionality, in a *mixin*-like fashion [7,8]. The role of the component system is thus that of enabling the reuse of specific functionalities across different application scenarios.
3. *aggregate functions*. Actual implementations of FC programs, as templated functions with a `node` parameter; note that also these functions are partitioned in several layers, starting from the built-ins that implement the core of FC, up to the applications written by the users of the FCPP library.

Figure 1 shows the dependencies between components, i.e., whether a component needs another component as its ancestor in the mixin composition. The number of such dependencies has been kept as low as possible, and it is always

possible to substitute a "required" component for another offering an analogous interface in the composition.

2.4 OpenGL

OpenGL is an API standard for the development and maintenance of real-time graphical applications. As a standard, it defines the high-level output of a set of functions with a given signature, related to graphical rendering and processing of the relevant data structures. The implementations reside into the graphical drivers, developed for a specific set of GPUs.

Internally, OpenGL is structured as a state machine whose state is called *context*, and consists of a set of variables and settings influencing the rendering output. The standard offers both functions for modifying the context, and others relying on it. OpenGL applications can define and maintain multiple contexts, possibly assigning them to different threads: one thread can be bound to only one context at a time. Allocated resources can be bound to a context.

The rendering process in OpenGL is defined by a pipeline, with the main goals of *(i)* transforming a set of vertex coordinates from a 3D space to a 2D one; and *(ii)* transforming such 2D coordinates into actual pixels displayed on the screen. The steps of the pipeline consist of small programs called *shaders*, which can be executed in parallel SIMD[2] steps on the GPU. The programmer can define the behaviour of three shaders of the pipeline: *(i)* vertex shader; *(ii)* geometry shader; *(iii)* fragment shader. The first and the third are crucial for defining the behaviour of an OpenGL application, while the second one is optional. The vertex shader transforms 3D vertex coordinates through several coordinate spaces applying transformation matrices. The geometry shader processes a geometric primitive to be rendered (e.g., a triangle) by injecting additional vertices into it, generating additional primitives. Then, the *rasterization* process maps the transformed vertices into pixels to display: such process generates *fragments*, data collections representing the output of (part of) the associated pixels. The fragment shader processes these generated fragments by calculating their output colors, inducing the final colors of the corresponding pixels.

3 Extending FCPP with a Graphical User Interface

3.1 Features

FCPP as presented in [2] already supported the execution of simulated networks with 3D physics and running aggregate programs. However, the simulation results were only made available through statistical summaries of numerical values across the network, in the form of text data files, that could then be turned into plots. This limited the possible user interactions with the system, making tasks such as algorithm design and bug detection harder.

[2] Acronym for *Same Instruction, Multiple Data*.

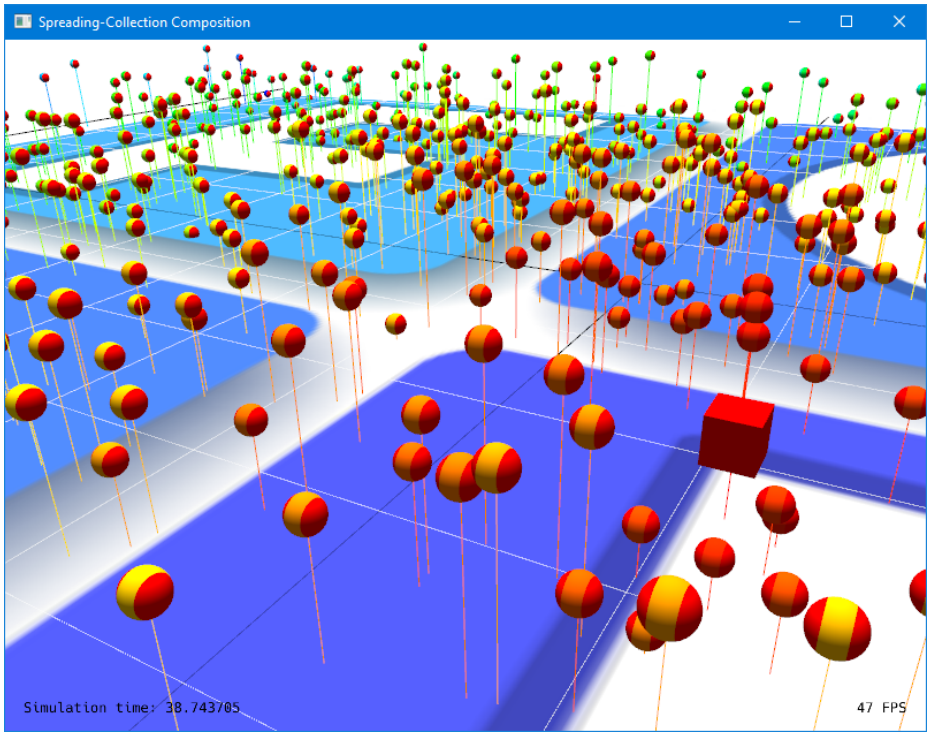

Fig. 2. Screenshot of the "spreading-collection" case study.

The introduction of the three-dimensional GUI drastically improves the interaction between the final user and FCPP's simulation toolset. The interface allows the user to visualize the simulated state of the network as a 3D scene (Fig. 2 and 3), that the user can navigate, zoom and rotate through. The 3D scene updates in real-time with the progress of the simulation, always displaying its current state. The GUI also allows for the tuning of the simulation speed, which can be paused, sped up or slowed down. Additional information is displayed within the main rendering window, such as the elapsed simulation time and the current *FPS*[3] value. The three-dimensional scene displays two main actors: *(i)* the grid plane; *(ii)* the nodes.

The grid plane is included to provide a reliable spatial reference, and consists of a plane placed along the x and y axes with a grid pattern of lines, possibly customized with a texture providing an additional spatial reference for the nodes to be compared with.[4] The texture information can also be accessed within the simulated program, to model interaction with obstacles thereby depicted. The

[3] Acronym for *Frames Per Second*.

[4] For instance, while simulating the flying patterns of a group of drones, a top-view of a city could be applied on the grid plane in order to compare the positions of the drones with the city blocks.

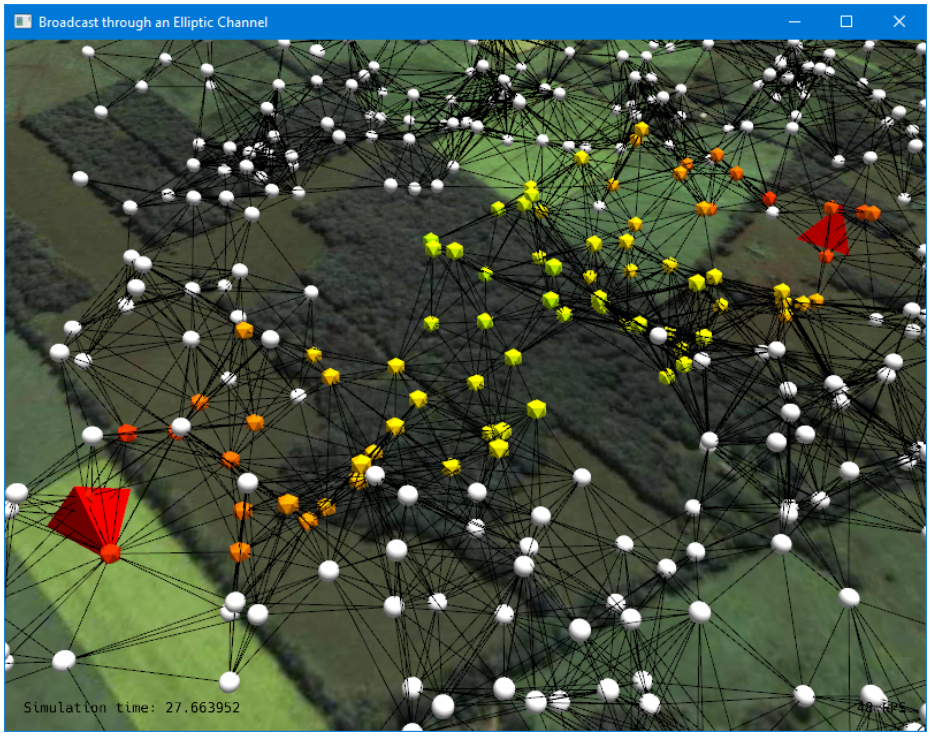

Fig. 3. Screenshot of the "channel-broadcast" case study. The grid and the pins are toggled off, while links to neighbours are displayed.

grid is automatically calibrated so that the step represents a power of 10 and a reasonable amount of lines are drawn, and can be toggled off (Fig. 3) in order to suppress visual noise if needed.

The nodes are a graphical representation of the devices forming the distributed network, with an appearance that represents summary information on the distributed computation, allowing to get a sense of the status of the whole network in a single glance. Nodes are characterized by a position along the three orthogonal axes, colored sections, a shape and a size. The position along the z axis can be emphasized by a pin starting from the center of the node and ending on the grid plane (Fig. 2). A node can support up to three colored sections (Fig. 2), each displaying a custom color defined by the programmer, which can change during the simulation. The node's shape is taken from a predefined set of three-dimensional models, which are scaled by the node's size value, and also both the shape and size can be set by the programmer and can change in real-time. Additionally, the GUI allows to toggle the visualization of the links of a node with its own neighbours (Fig. 3), explicating the network topology induced by message exchanges among devices.

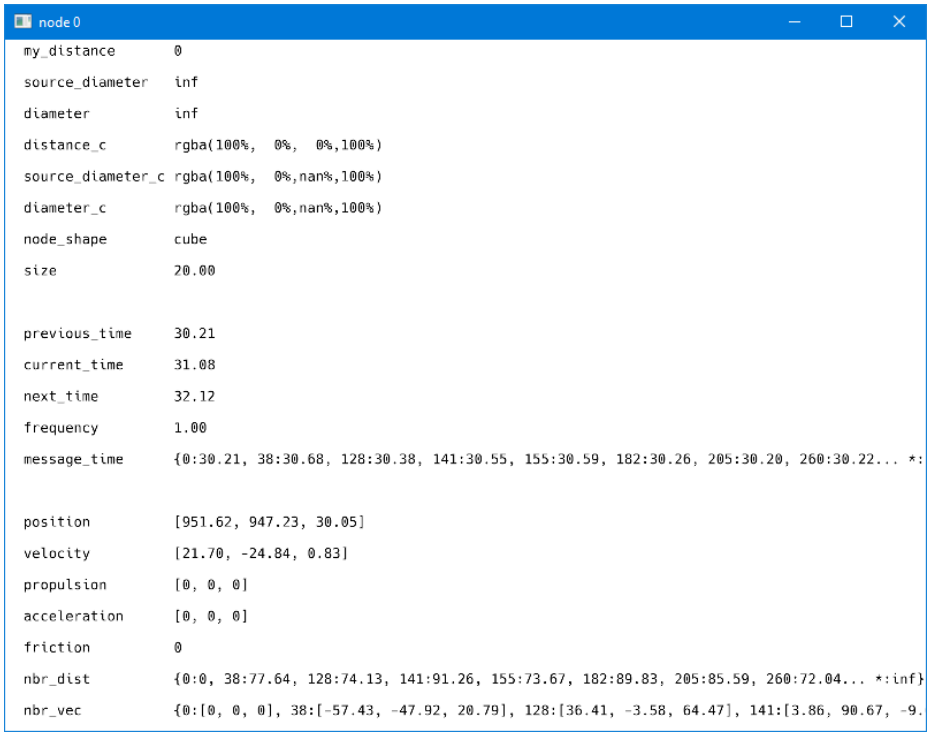

Fig. 4. Screenshot of a monitoring window.

The summary information given by the grid and nodes is complemented by the possibility of accessing detailed information for any particular node. By clicking on a node of choice, the user opens a new monitoring window (Fig. 4) displaying a customizable list of values for the selected node (defined through the *storage* component of FCPP). The list updates in real-time, allowing the user to monitor the evolution of such values together with the overall graphical representation. Hovered or selected nodes are emphasized, and any number of nodes can be selected at once. Selection of nodes can be toggled off, informing the user of the deactivation by changing the cursor's shape.

Overall, the presented features allow the programmer to better test and debug distributed applications, therefore promoting a better design process through an enhanced human-computer interaction. Furthermore, as we shall see in the next section, the introduction of the GUI into FCPP projects comes with a minimal cost for the end-user.

3.2 Architecture

The 3D GUI is managed by the *displayer* component, which is dedicated to update the rendering window, to regulate simulation speed and to manage user's

input and monitoring windows. As shown in Fig. 1, the displayer component (violet) has hard dependencies with other simulation-specific components: *(i)* positioner (to access positions to be represented); *(ii)* storage (to access detailed values for nodes); *(iii)* timer (to access time information); *(iv)* identifier (to access the nodes themselves). Just by adding a displayer component into an FCPP project (a one-word edit), the GUI is immediately available and already able to show the nodes' positions, connections and detailed information. With minimal effort, the representation can then be enhanced by adding a few lines to regulate the colors, shape and sizes of nodes into the program run by the nodes.

In case interaction with obstacles depicted in the texture is needed, a further *simulated map* component is provided, which provides nodes with information on their surroundings (in terms of obstacle presence, closest obstacles and empty spaces). This information can be used by library aggregate functions to mimic repulsive elastic forces, preventing nodes from crossing obstacles.

During an FCPP simulation, simulated actions are processed as *events* issued by the various components at given simulated times. The displayer follows the general execution path, by defining periodic update events which first initialize necessary attributes, and then handle: *(i)* finding the node under the cursor through auxiliary raycast operations; *(ii)* rendering the nodes, the plane and 2D information text; *(iii)* managing input events; *(iv)* managing monitoring windows; *(v)* swapping framebuffers.[5] The displayer component makes use of a few auxiliary classes: *(i) renderer* (abstraction of a rendering window which can be invoked to render on the relative window); *(ii) info_window* (abstraction of a monitoring window); *(iii) camera* (needed to define and manipulate the point of view from which the scene is rendered).

A renderer object has at its disposal both shared and window-specific resources. Among the latter there are a camera object and the context related to the window. The displayer owns an instance of the renderer and, since such renderer is the first one to get created, it allocates all the shared resources. Thus, such renderer is considered *master*, while other *slave* renderers will need to access the shared resources through the master. When clicking on a node, the displayer creates a new *info_window* instance which is handled by a secondary thread. An *info_window* object owns a (slave) renderer as well, which accesses the shared resources through the master renderer.

3.3 Technical Challenges

While realising the GUI for FCPP, several challenges were identified that needed to be addressed:

1. The graphic libraries used should be as *portable* as possible, in order not to reduce the overall portability of the FCPP tool;

[5] The *framebuffer* is a buffer storing frames to be rendered, and swapping such buffer basically means to display its content on screen.

2. The graphical representation should be sufficiently *performing*, lightweight and optimised to be able to process in real-time even large-scale simulated networks, minimising the overhead on the simulation itself;
3. The GUI code should *integrate* into the FCPP component structure, so that the mere addition and removal of a dedicated displayer component should enable and disable the user interface, without further actions needed on user code;
4. The *representation* should allow the recognition of high-level patterns in the whole network, as well as detailed data on individual nodes;
5. The interface should allow *interaction* with the simulation flow;
6. The GUI should be compatible with 2D as well as 3D spaces, and be compatible with spaces with obstacles as well.

In order to address portability (1), efficiency (2) and 3D readiness (6), OpenGL was chosen as reference graphical library, due to its low-level and portable nature (through the CMake build tool). Significant effort was spent on optimising the performances, with a strong impact, using strategies such as adaptive framerates (for the main window based on CPU availability, and for info windows based on the updates of the corresponding node), multi-threading (between the main window and info windows) and indexing/buffering of shapes of general use.

In order to address integration (3), a displayer component was developed, with a minimal node structure (caching node positions before draws, updating some display information after rounds, managing node highlight, and offering a draw method), and a more complex net structure (managing frame refreshes, keyboard and mouse inputs, and auxiliary windows). The data to be displayed was obtained through the existing *positioner* (for node positions) and *storage* (for node data) components, in order to further the integration with existing code. In order to allow the representation (4) requirements on whole networks, while preserving integration (3), options were provided to specify storage fields where node color, shape and size information could be written by aggregate programs and then read by the displayer component. Similarly, for individual nodes a fully template-based introspection mechanism was written to automatically extract a readable representation of all storage contents in info windows.

Finally, in order to address interaction (5), keyboard shortcuts were defined to regulate the simulation flow, inspired from video playback applications, along further shortcuts to regulate space navigation (also assisted by mouse input). In order to address the presence of obstacles (6), we included the possibility of loading a custom texture for the environment, while also providing an additional *simulated map* component to access texture information and detect nearby obstacles.

3.4 Comparison with Protelis and Scafi

The other two languages for the development of AP systems, Protelis [19] and Scafi [24], both rely on the external simulator Alchemist [18] for their simulation capabilities. Alchemist supports a number of features similar to the ones

developed into the new graphical interface for FCPP: the control of the simulation flow, a graphical visualization of the status of the overall network through colors and shapes, and access to detailed information on nodes through separate windows. However, Alchemist is only restricted to 2D simulations, and no support for physics is available. Furthermore, the performance increase of FCPP can be seen in the graphical interface as well: larger networks can be visualized in real-time, given the performance limits of a machine at hand.

It is worth noting that several multi-robot simulators exist, that support features such as the ones described here, and possibly more, e.g. [13,20]. Such tools, however, lack the integration with the FCPP library that allows the simulation of Aggregate Programming systems.

4 Case Studies

The FCPP distribution comes with a sample project,[6] as a reference to ease the setup of new projects. With the introduction of the GUI, the build system had to be moved from Bazel to CMake, and two additional graphical simulations were added to the sample project (which previously consisted of a single batch simulation), described in the following. In both simulations, devices move through waypoints randomly chosen in a parallelepiped area, and the connection topology is driven by the physical distance between the devices.

4.1 Composition of Spreading and Collection Blocks

The first example is an implementation of a typical monitoring application in aggregate computing. Such an application is designed by functionally combining few basic steps, which are provided by the FCPP coordination library: *(i)* distance computation from a selected source device; *(ii)* aggregation of the data distributed across the network, through paths of communication descending the distances previously computed; *(iii)* broadcasting the overall result computed in the source device to the whole network. In this specific example, the aggregation was set to approximate the network diameter.

Fig. 2 presents a screenshot of this application. The source node is highlighted by representing it as a larger cube. Other nodes are displayed as spheres with bands of color: the hue of the central band displays the distance approximated at step *(i)*, while the hue of the lateral bands displays the estimated diameter resulting from step *(iii)*. Note that central bands of the nodes closer to the source are more reddish, becoming more yellowish (then greenish, etc.) as they are farther.

4.2 Broadcast through a Self-organising Channel

The second example presents another typical aggregate routine: selection of a communication channel between a source and a destination, and broadcast of

[6] `https://github.com/fcpp/sample-project`

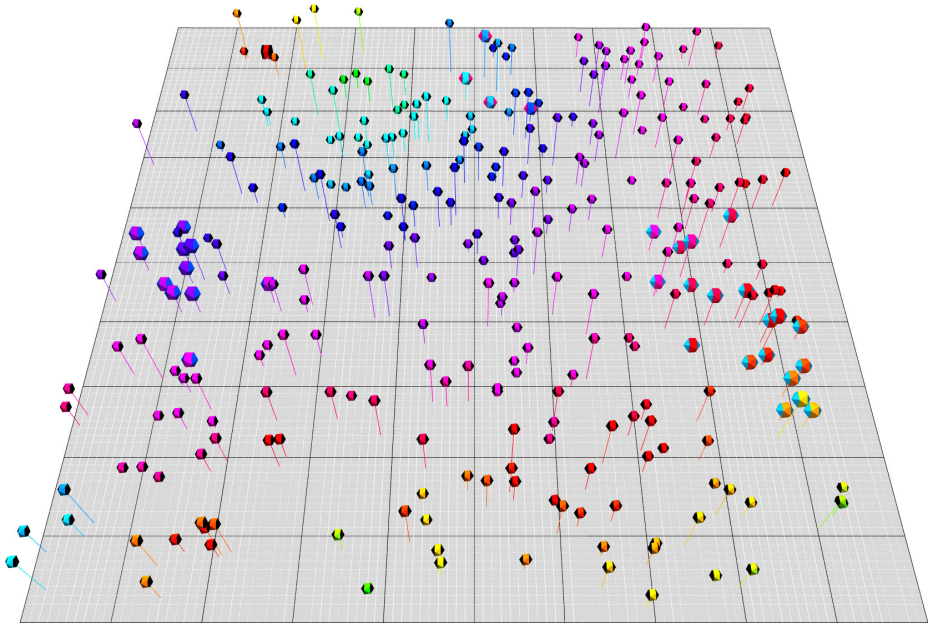

Fig. 5. Screenshot of the "message-dispatch" case study.

data through it. The channel is selected using the definition of ellipse as a locus of points, so that: *(i)* distances d_{is}, d_{id} from the source and destination devices are computed in every device i; *(ii)* the distance d_{ds} between source and destination is made available in the network through a broadcast; *(iii)* the channel area is then defined as those devices i such that $d_{is} + d_{id} \leq d_{ds} + w$, where w is a width parameter tuning the minor axis of the ellipse; *(iv)* data can then be broadcast in this restricted selected area. As before, all of these basic steps are available in the FCPP coordination library.

Fig. 3 presents a screenshot of this application. The source and destination devices are highlighted as larger tetrahedra. Devices outside of the communication channel are represented as white spheres, and the channel itself is visualised as colored icosahedra, with the color hue tuned to represent their distance from the source and destination.

4.3 Peer-to-peer Message Dispatch

The third example presents a further archetypal task in distributed systems: peer-to-peer dispatch of messages. This is accomplished through *aggregate processes* [9], that expand guided by an adaptive spanning tree structure. Figure 5 presents a screenshot of this application. The root of the spanning tree is represented as a larger cube (top-left). Nodes are colored according to their distance from the root in their central band. Nodes involved in message exchanges are larger, and their sides are colored with a color generated from the message. Three

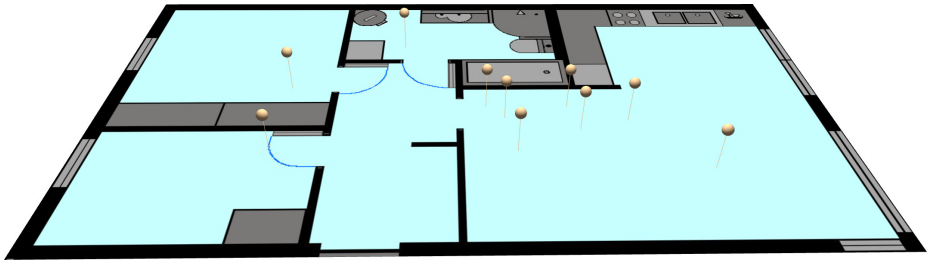

Fig. 6. Screenshot of the "apartment-walk" case study.

messages are currently being dispatched: one in light blue (right), another in red (top), and a last one in dark blue (left).

4.4 Random Walk in an Apartment

The fourth and last example presents a minimal scenario with obstacle avoidance. Figure 6 presents a screenshot of this application. In this scenario, 10 people (tan pins) are randomly moving through rooms in an apartment, while bouncing off walls (black areas), furniture (dark grey areas), and other people; being restricted to the free floor area (light blue).

5 Conclusion

In this paper, we presented a graphical user interface for the FCPP simulator. The GUI allows to control the simulation flow and to visualize both summary information of the network, through colors and shapes, and detailed information of individual devices, through auxiliary windows. The interface is implemented as a component that can be added to the others already available in FCPP, requiring minimal effort from the end user for its setup.

Future work may enhance the GUI with additional customisation options, support for loading general 3D models for nodes, and the support to multiple simultaneous visualizations of the same network. The possibilities for the user to affect the system under simulation may also be extended, by allowing to modify the detailed information of a node, or to drag-and-drop nodes to different locations. Also the possibility of filtering the displayed nodes based on their properties could be useful for the exploration of the system behavior, especially in 3D scenarios.

References

1. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional (2004)

2. Audrito, G.: FCPP: an efficient and extensible field calculus framework. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2020, Washington, DC, USA, August 17-21, 2020. pp. 153–159. IEEE (2020). https://doi.org/10.1109/ACSOS49614.2020.00037
3. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Trans. Comput. Log. **20**(1), 5:1–5:55 (2019). https://doi.org/10.1145/3285956
4. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. pp. 436–501. IGI Global (2013)
5. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the internet of things. IEEE Computer **48**(9), 22–30 (2015). https://doi.org/10.1109/MC.2015.261
6. Beal, J., Viroli, M., Pianini, D., Damiani, F.: Self-adaptation to device distribution in the internet of things. ACM Transactions on Autonomous and Adaptive Systems **12**(3), 12:1–12:29 (2017). https://doi.org/10.1145/3105758
7. Bracha, G., Cook, W.R.: Mixin-based inheritance. In: International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) / European Conference on Object-Oriented Programming (ECOOP). pp. 303–311. ACM (1990). https://doi.org/10.1145/97945.97982
8. Cannon, H.I.: Flavors: A non-hierarchical approach to object-oriented programming. Tech. rep., Artificial Intelligence Laboratory, MIT, USA (1979)
9. Casadei, R., Viroli, M., Audrito, G., Pianini, D., Damiani, F.: Engineering collective intelligence at the edge with aggregate processes. Eng. Appl. Artif. Intell. **97**, 104081 (2021). https://doi.org/10.1016/j.engappai.2020.104081
10. Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P.: Mobile data collection in sensor networks: The TinyLime. Pervasive and Mobile Computing **1**(4), 446–469 (2005). https://doi.org/10.1016/j.pmcj.2005.08.003
11. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008). https://doi.org/10.1145/1327452.1327492
12. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks. p. 455–462. LCN '04, IEEE Computer Society, USA (2004). https://doi.org/10.1109/LCN.2004.38, https://doi.org/10.1109/LCN.2004.38
13. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 2149–2154. Sendai, Japan (Sep 2004)
14. Kondacs, A.: Biologically-inspired self-assembly of two-dimensional shapes using global-to-local compilation. In: 18th International Joint Conference on Artificial Intelligence (IJCAI). pp. 633–638. Morgan Kaufmann (2003)
15. Lasser, C., Massar, J., Miney, J., Dayton, L.: Starlisp Reference Manual. Thinking Machines Corporation (1988)
16. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The TOTA approach. ACM Transactions on Software Engineering Methodologies **18**(4), 15:1–15:56 (2009). https://doi.org/10.1145/1538942.1538945
17. McIlroy, M.D., Buxton, J., Naur, P., Randell, B.: Mass-produced software components. In: 1st international conference on software engineering. pp. 88–98 (1968)
18. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. Journal of Simulation **7**(3), 202–215 (2013). https://doi.org/10.1057/jos.2012.27

19. Pianini, D., Viroli, M., Beal, J.: Protelis: practical aggregate programming. In: 30th ACM Symposium on Applied Computing (SAC). pp. 1846–1853. ACM (2015). https://doi.org/10.1145/2695664.2695913
20. Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., Mathews, N., Ferrante, E., Di Caro, G., Ducatelle, F., Birattari, M., Gambardella, L.M., Dorigo, M.: ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. Swarm Intelligence **6**(4), 271–295 (2012)
21. Terraneo, F., et al.: Miosix embedded os (2021), http://www.miosix.org/, Last accessed on 2021-7-8
22. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Transactions on Modeling and Computer Simulation **28**(2), 16:1–16:28 (2018). https://doi.org/10.1145/3177774
23. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Transactions on Modelling and Computer Simulation **28**(2), 16:1–16:28 (2018). https://doi.org/10.1145/3177774
24. Viroli, M., Casadei, R., Pianini, D.: Simulating large-scale aggregate mass with alchemist and scala. In: Federated Conference on Computer Science and Information Systems (FedCSIS). Annals of Computer Science and Information Systems, vol. 8, pp. 1495–1504. IEEE (2016). https://doi.org/10.15439/2016F407