A Visualization Module for Transparent Parallel Execution of Cellular Automata

Grzegorz Bazior[1], Andrea Giordano[2], Alessio De Rango[3], Davide Macrì[2], Giuseppe Mendicino[3], Luigi Rizzo[2], Rocco Rongo[4], Jarosław Wąs[1], and William Spataro[4]

[1] AGH University of Krakow, Poland
[2] ICAR-CNR, Rende, Italy
[3] University of Calabria, Department of Environmental Engineering, Italy
[4] University of Calabria, Department of Mathematics and Computer Science, Italy

Abstract. Transparent parallel execution of Cellular Automata (CA), or other grid-based numerical models, is an important topic enabling feasibility and scalability of large models used to simulate complex physical phenomena such as debris flows, hydrological, biological, and neuroscience systems. The need for transparency arises from the availability of heterogeneous parallel execution contexts, including CPU-based shared and distributed memory systems, GPU, and multi-GPU architectures. While several studies address the transparent execution of CA models across different platforms and optimization strategies, limited attention has been given to extending this transparency to the visualization module, which is essential for output analysis and debugging.
This paper presents an integration layer that bridges the OpenCAL++ cellular automata framework with a Qt/VTK visualization tool. The proposed solution enables model-independent offline visualization by collecting OpenCAL++ output files and providing a stable interface for model-aware data interpretation. In addition to implementation details, we present examples of its application to different CA models.

Keywords: Cellular automata · Qt/VTK visualization · HPC transparency

1 Introduction

Parallel executions of Cellular Automata (CA) handle multidimensional lattice states that are regularly modified during execution. Such lattice-state values can be profitably visualized to study the dynamics of the specific modeled system. Although the platform considered in this paper, i.e., the OpenCal++ cellular automata framework [8], targets transparency in parallel execution across heterogeneous execution contexts, the analysis and visualization phase is often still fragmented. For instance, the frequent cell inspection typically relies on ad hoc scripts and backend-dependent assumptions, and crucial model semantics may be lost once only raw per-step outputs are retained.

This paper focuses on the *integration boundary* between OpenCal++ execution and an external, model-aware visualization environment. A generic Qt/VTK

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

visualizer can efficiently render interactive 2D/3D views, but to remain consistent across models it must also know *how* to interpret substates (names, layout, valid ranges, no-value markers), how to retrieve default visual attributes (e.g., colors), and how to map fields to the coloring and the shape of displayed scenes. Such information is typically implicit in model code and configuration and cannot be conveyed solely by numerical dumps.

We introduce a methodology that bridges OpenCal++ runs output to the visualizer through two coupled artifacts: (i) an *execution descriptor* collecting run metadata such us grid dimensions, number of timestep, cell sub-state names, output file names and so on, and (ii) a compiled *model specific binary* exposing a stable interface for serialization and model-aware sub-state access. The visualizer loads these artifacts to uniformly reconstruct and inspect simulations, regardless of the execution context and the specific CA model.

The remainder of the paper is organized as follows. Section 2 recalls the main CA execution contexts and motivates the need for portable integration artifacts. Section 3 details the proposed OpenCal++–Visualizer integration layer and the associated packaging workflow. Section 4 presents the Qt/VTK visualization environment and the supported interaction workflows. Finally, Section 6 concludes the paper and outlines future work.

2 Background: Parallel Cellular Automata Execution and Integration Artifacts

Cellular Automata (CA) model complex systems as collections of simple entities evolving on regular lattices via local transition rules. This locality makes CA simulations naturally suitable for parallel execution, since each cell update depends on a bounded neighbourhood and can be computed concurrently across the grid [5,13]. In large-scale scenarios, the execution context determines where the simulation state physically resides and how it is produced and exposed to downstream analysis and visualization.

In distributed-memory settings, the cellular space is partitioned into subdomains assigned to different processes (Fig. 1). Because neighbourhoods may cross partition boundaries, processes exchange halo/ghost-cell data at each time step to preserve correctness (e.g., [4,11,6]). As a consequence, the global state is fragmented across processes and must be gathered to reconstruct a consistent view of the CA domain. In shared-memory systems, domain decomposition can still be applied, but subdomains are assigned to threads sharing the same address space; thus, explicit halo exchanges are not required, and synchronization mainly coordinates read/write buffers across timesteps. This makes the full simulation state directly accessible in memory, simplifying coupling with analysis and visualization components. Finally, GPGPU execution maps cell updates to massively parallel GPU threads [15]. State is typically evolved in device memory for many timesteps to reduce costly host-device transfers, with performance improvements achievable through shared-memory usage and memory-aware kernels [16]. Multi-GPU settings mirror the distributed-memory case, replacing inter-node communication with inter-device exchanges. Overall, heterogeneous contexts

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

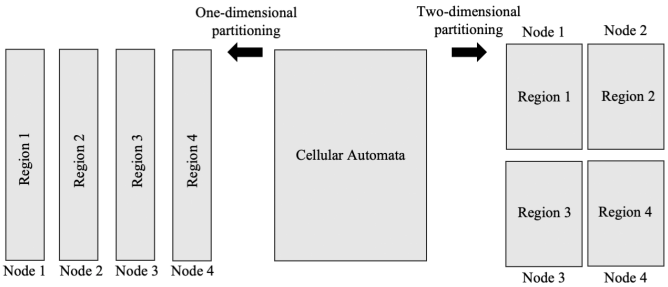


Fig. 1: Distributed-memory CA execution: the cellular space is partitioned into regions assigned to parallel nodes (examples of uni-dimensional and two-dimensional partitioning).

motivate a visualization workflow that abstracts the physical data location while exposing a consistent representation of the simulated fields.

The execution time on N processing elements can be expressed as an ideal speedup plus parallel overheads (e.g., communication and synchronization) [12]. In CA parallel execution, these overheads may significantly affect scalability in distributed and multi-device settings. OpenCal++ can mitigate such effects through strategies such as communication/computation interleaving, multi-border exchanges, look-ahead relaxation, and dynamic load balancing for uneven activity patterns [9,7,17,10]. While these techniques primarily target performance, they also influence how and when consistent state snapshots can be made available for inspection. Please refer to [8] for additional details.

In this paper, we focus on the integration boundary between OpenCal++ execution and the external visualization tool. Rather than requiring the visualizer to depend on backend-specific memory layouts, the execution stage produces a compact set of *integration artifacts* (see next section) that provide a stable contract between the simulation platform and the visualization module, enabling portable, model-aware inspection without coupling the GUI/rendering layer to the underlying execution context.

Related work. Visualization of HPC simulation data has been studied in both in-situ and post-hoc settings. In-situ frameworks such as ParaView Catalyst [2] and Ascent [14] execute analysis concurrently with the simulation, keeping data in memory and avoiding costly I/O. While these approaches minimize data movement, they require the simulation code to be instrumented with framework-specific adapters and introduce runtime coupling between the execution backend and the visualization layer, increasing the implementation burden for model developers. Our approach is deliberately *offline*: OpenCal++ produces self-contained execution packages (descriptor, per-timestep dumps, model binary) that are consumed by the visualizer independently of the simulation run, with no additional scripts or wrappers required on the visualization side. This decoupling offers two practical advantages. First, computationally expensive simulations can be executed on remote HPC systems (e.g., a supercomputer such as Leonardo) while results are inspected locally on a standard workstation, distributing only the output package

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

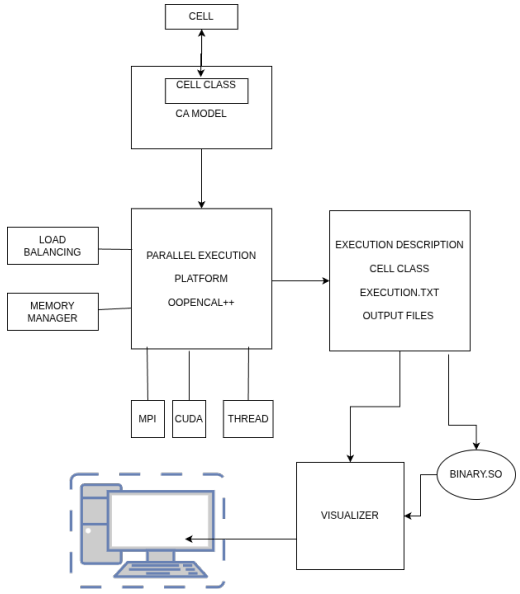


Fig. 2: Visualizer integration architecture.

and the visualizer binary without requiring access to the OpenCal++ runtime or source code. Second, the same execution package supports unrestricted post-hoc navigation across timesteps and multiple re-inspections without re-running the simulation. The key distinguishing feature with respect to generic post-processing pipelines is the *model-semantic contract*: rather than exposing raw numerical arrays that require user-side interpretation, our plug-in interface preserves substate naming, valid ranges, no-value conventions, and default rendering attributes, enabling consistent model-aware visualization without modifying the visualizer itself.

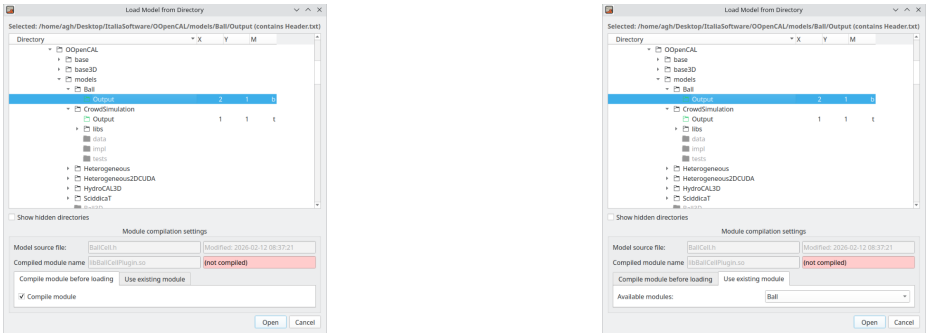
3 OpenCal++ Visualizer Integration Layer

This section formalizes the integration boundary between OpenCal++ executions and an external Qt/VTK visualizer. The objective is twofold: (i) keep the visualizer independent of OpenCal++ internal memory layouts and execution backends, and (ii) preserve *model semantics* needed for a consistent interpretation of simulation states, including substate naming and formatting, value constraints (e.g., valid ranges and no-value markers), and default visual attributes.

Our proposal is based to three artifacts produced at execution time:

1. *Execution descriptor*: a compact metadata file describing the run (grid dimensions, number of timesteps, substate names, output file names).
2. *Per-timestep outputs*: the simulation dumps (one or more fields per step, optionally in binary form).
3. *Model-specific binary*: a compiled shared library exposing a stable interface to retrieve model semantics required at visualization time.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25



(a) Execution package with plug-in compilation.

(b) Execution package with an already available plug-in.

Fig. 3: Model loading workflow. The dialog detects valid execution packages and either compiles a compatible plug-in or reuses an existing compiled module.

Figure 2 summarizes the proposed architecture and highlights the integration boundary. The *execution descriptor* allows the visualizer to reconstruct the spatial domain and locate the per-step outputs. However, numeric dumps alone are not sufficient to interpret sub-states consistently across models. To bridge this semantic gap, a model-related binary, i.e., `binary.so`, encapsulates model-specific code, specifically the `Cell` class, that contains the default rendering behavior. The `binary.so` is dynamically loaded by the visualizer at the beginning of its execution. In this way, we avoid recompiling the visualizer for each CA model.

To facilitate deployment across heterogeneous environments, the visualizer supports two complementary loading workflows, illustrated in Figure 3. When a compatible model plug-in is already present in the execution package, it is reused directly (Fig. 3b). Otherwise, the visualizer automatically triggers a lightweight rebuild step using the provided headers and sources to generate a platform-compatible shared library (Fig. 3a). This mechanism preserves portability while maintaining a stable visualization interface. In particular, each model provides a minimal `Cell`-level contract, see the Cell abstract interface in Listing 1.1, that supports serialization/deserialization of cell states, through serialize/deserialize methods, and model-specific rendering behavior, through `getSubStateValue` method. This contract enables the visualizer to (i) decode cell substates from the dumps and (ii) obtain consistent display attributes across models.

```
class Cell {
public:
  virtual int getSerializedSize() const = 0;
  virtual void deserialize(void* ptr, Cell* pCell) = 0;
  virtual void serialize(void*& ptr, const Cell* pCell) const = 0;
  virtual std::string getSubStateValue(const char* substate) const = 0;
  virtual Color getDefaultColor() const = 0;
};
```

Listing 1.1: Abstract `Cell` interface used at the OpenCal++–Visualizer integration boundary.

The integration contract deliberately separates *portable* artifacts (execution descriptor and per-timestep outputs) from *model-dependent* ones (the model binary). While metadata and dumps can be moved unchanged across machines, binary compatibility of the shared library depends on the operating system,

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

compiler toolchain, and ABI conventions. In our approach, a lightweight rebuild, including the model `Cell` class, is performed on the target machine, while keeping the execution descriptor and output dumps unchanged.

The current prototype assumes local filesystem access to the execution descriptor, outputs, and model binary. However, the same contract naturally supports future extensions where the visualizer fetches artifacts remotely (e.g., via a network share, object storage, or a thin client/server bridge deployed close to the HPC allocation), enabling interactive inspection of simulations produced on external systems without manual data staging. As a preliminary validation, we tested the visualizer under remote access conditions using `sshfs`, which exposes a directory located on a remote server as a locally mounted filesystem via the SSH protocol. The visualizer operated correctly on such mounts without requiring any modifications; performance, however, was visibly lower than with local storage, reflecting the additional latency and bandwidth limitations of remote file access. This motivates future work on remote-aware ingestion (e.g., caching and batched reads) and streaming-oriented access to execution artifacts.

4 Qt/VTK Visualizer and Interaction Workflows

This section describes the Qt/VTK visualization environment that *consumes* the integration artifacts produced by OpenCal++ runs (Section 3). The visualizer is designed to remain independent of OpenCal++ internal data layouts and execution backends, while still enabling *model-aware* inspection.

At startup, the visualizer parses the execution descriptor to reconstruct the lattice geometry (2D or 3D), enumerate available fields/substates, and resolve the location and encoding of per-timestep outputs. Model-specific semantics (e.g., default colors, textual formatting, and substate access policies) are provided by dynamically loading the model binary and querying the `Cell`-level contract (Listing 1.1). This separation allows the same GUI and rendering pipeline to be reused across models and across heterogeneous execution contexts. The VTK runtime is embedded into the Qt GUI through a native OpenGL widget (e.g., `QVTKOpenGLNativeWidget`). The GUI exposes controls for: (i) timestep navigation and playback, (ii) selection of active substates/fields, (iii) value range normalization and no-value handling, and (iv) view-mode configuration. In **2D mode**, the lattice is rendered as a map-like top-down view where a selected scalar field drives the colormap. In **3D mode**, a selected field can additionally drive surface elevation (extrusion), while coloring can be mapped from the same or from a different field, enabling combined analysis of morphology and dynamics (e.g., elevation from terrain and color from transported material). Interactivity is implemented through an incremental refresh procedure: when the timestep changes (via slider, playback, or direct selection), the application only considers data of the specific time step, updates VTK scalar arrays, optionally updates geometry when elevation mapping is enabled, and triggers rendering. This strategy limits overhead for long runs and multi-field states and avoids rebuilding the pipeline at each interaction.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

Algorithm 1: Timestep-driven refresh procedure used by the visualizer.

Input: timestep s, active field set $\mathcal{F}$, view mode m
Update internal parameters (grid size, s, active fields/ranges)
Read per-step field values for timestep s (only $\mathcal{F}$)
if $m = 3D$ ***and*** *elevation field is enabled* **then**
  Update geometry (height/extrusion driven by selected field)
Update scalar arrays and normalization ranges (colormap LUT)
Request render and synchronize GUI state (camera and controls)

During execution, OpenCal++ generates an output directory that the visualizer subsequently consumes. The directory includes the execution descriptor, the per-timestep dumps, and auxiliary files that support efficient random access and model-aware interpretation.

The generated files include:

1. `Header.txt` - the execution descriptor providing metadata required by the visualizer, such as grid dimensions, timestep range, output naming scheme, data encoding mode (text or binary), available sub-states, and optional reduction operators (see Listing 4).
2. Node data files (text or binary) - storing the simulation state for each timestep (potentially partitioned by node, depending on the backend and decomposition strategy).
3. Node index files - for each computational node, an index maps timesteps to byte offsets in the corresponding node data file, enabling efficient random access to selected timesteps without loading the entire trace into memory. In addition, index files may store local grid extents along x and y, which can vary over time when dynamic load balancing is enabled.
4. Model binary and public interface - a shared library implementing the visualization-facing contract (Section 3) and, when needed, its public header(s) to support consistent dynamic loading and integration on the visualization workstation.

Header file. The execution descriptor (`Header.txt`) contains a wide range of configuration parameters describing the simulation run. The excerpt below shows the entries that are directly relevant for visualization.

```
GENERAL:
    number_of_columns=500
    number_of_rows=500
    number_steps=500
    output_file_name=ball
DISTRIBUTED:
    number_node_x=4
    number_node_y=4
VISUALIZATION:
    mode=binary
```

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

For comparison, the `VISUALIZATION` section used in the CrowdSimulation model [3] may include additional metadata:

```
VISUALIZATION:
    mode=text
    substates=dynamicFloorField,staticFloorField
    reduction=pedestrian_count,max_dynamic_field
```

The following entries are particularly relevant for the visualizer:

1. `number_steps` - total number of simulation timesteps. Steps may be sparsely exported (e.g., every k-th step); gaps are handled transparently.
2. `number_of_columns` and `number_of_rows` - global domain dimensions.
3. `output_file_name` - base name used to generate all output files (e.g., `ballN_index.txt` for node N).
4. `number_node_x` and `number_node_y` - decomposition layout along x and y.
5. `VISUALIZATION` - visualization-specific metadata:
   - `mode` - dump encoding (`text` or `binary`),
   - `substates` (optional) - comma-separated list of exposed substates,
   - `reduction` (optional) - declared reductions synchronized per timestep.

4.1 Developing models compatible with the Visualizer in OpenCal++

OpenCal++ is designed to support a wide range of CA models; therefore, a primary objective is to make model integration straightforward while preserving performance and portability across execution backends. To introduce a new model, the developer creates a dedicated directory under the `models` subtree and provides the minimal set of source and configuration assets required by both the simulation runtime and the visualization toolchain.

At a minimum, a visualizer-compatible model provides:

1. a model configuration file (e.g., `Configure.txt`) that specifies simulation parameters and enables the generation of visualization-oriented metadata (notably the `VISUALIZATION` section discussed in before);
2. a C++ implementation of the `Cell` class conforming to the contract in Listing 1.1, encapsulating the per-cell state together with serialization hooks and visualization-facing accessors;
3. a model implementation derived from `Model2D` or `Model3D`, depending on spatial dimensionality, defining initialization, the per-step transition rule, and optional step-level methods.

Model base class. Listing 1.2 sketches the core methods expected by a typical 2D model implementation. In the actual framework, these methods are resolved through template-based static binding (to avoid unnecessary runtime dispatch and to enable backend specialization). Still, for clarity, we report only the methods a developer typically implements or customizes.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

```
template <class CellType, class SpaceType>
class Model2D {
public:
    // One-time setup and teardown
    void init();        // model-specific initialization
    void finalize();    // cleanup and finalization
    // Per-cell transition rule executed at each timestep
    void transitionFunction(int x, int y);
    // Optional step-level methods
    void reducerUpdate(int x, int y, const CellType& c);
    void endStep(int step);
    // Return true if this step should be exported for visualization
    bool visualizeStep(int step);
};
```

Listing 1.2: Sketch of the methods expected by a typical `Model2D` implementation in OpenCal++.

Space interface. Model implementations interact with the simulation domain through the `Space2D` abstraction, which provides access to cell storage, neighborhood queries, and step management. Internally, OpenCal++ specializes the space implementation to the selected backend (sequential, OpenMP/threads, MPI, CUDA). Listing 1.3 summarizes the essential methods commonly used by model developers.

```
template <class CellType>
class Space2D {
public:
    CellType& cell(int x, int y);
    void setCell(int x, int y, const CellType& value);
    // Initialization helpers
    void initCell(int x, int y, const CellType& value);
    // Commits newly computed state (buffer swap)
    void commitStep();
    // Local/global coordinate ranges (backend-dependent decomposition)
    RangeCoord range(); // local extent owned by this execution unit
    RangeCoord dimension(); // global domain dimensions
    void startStep(int step); // Step lifecycle
    // Output export for visualization/post-processing
    void writeOutput(int step, const char* outputFileName,
        int startX, int startY, int endX, int endY, int nodeId, CellType* regionCurr);
};
```

Listing 1.3: Simplified `Space2D` interface used by model implementations.

Reduction mechanism. OpenCal++ provides a reduction mechanism to aggregate global statistics across the cellular domain during simulation. Reductions are defined by the model developer and updated locally per cell. At the end of each timestep, partial contributions are synchronized and combined, yielding scalar values such as sums, extrema, or conditional counters. Reduced values can be queried inside model logic and exported as per-step metadata when enabled in the configuration (`VISUALIZATION: reduction=...`).

Portability considerations for model binaries. Since visualizer integration relies on a dynamically loadable model binary, developers should avoid assumptions about platform-dependent properties such as the size/layout of built-in C++ types and ABI-sensitive constructs. Locale-dependent behaviors should be avoided in

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

serialization and parsing: numeric formatting should enforce a locale-independent representation (e.g., decimal dot) to guarantee reproducible execution artifacts across workstations.

(a) Name only.

(b) Name + explicit min/max.

(c) Name + min/max + no-value.

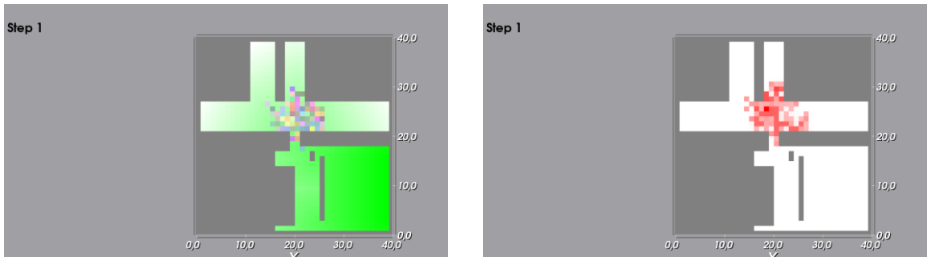
(d) Full metadata (format, range, no-value, gradient).

Fig. 4: Substate configuration levels supported by the visualizer. Explicit bounds stabilize colormap normalization and enable elevation mapping in 3D mode; no-value markers are excluded from rendering and range estimation.

In CA models, a cell state is typically composed of multiple variables (*substates*), representing physical, environmental, or behavioral quantities. Because substate meaning, numeric type, valid range, and missing-value conventions are model-dependent, the visualizer supports a metadata-driven configuration: substate semantics can be declared in the execution descriptor and complemented by model-side defaults (via the dynamically loaded binary), while still allowing user refinements at the GUI level (see Figure 4).

The configuration ranges from minimal (name only) to fully specified (name, numeric format, admissible range, no-value marker, and custom color gradient). When explicit bounds are not provided, the visualizer can estimate ranges on demand by scanning selected timestep(s); when a no-value marker is defined, such values are excluded from range estimation and rendering. Explicit min/max bounds are also beneficial to stabilize colormap normalization across timesteps and to enable the use of a substate as elevation in 3D mode.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25



(a) Default coloring. **(b)** Dynamic floor field.

Fig. 5: Crowd dynamics examples. The visualizer supports both model-defined default coloring and explicit substate-based colormaps.

5 Case studies: crowd dynamics and debris-flow simulation

To illustrate how sub-state semantics and model defaults translate into interactive analysis, we report two representative case studies: a crowd dynamics model and the SciddicaT debris-flow simulator.

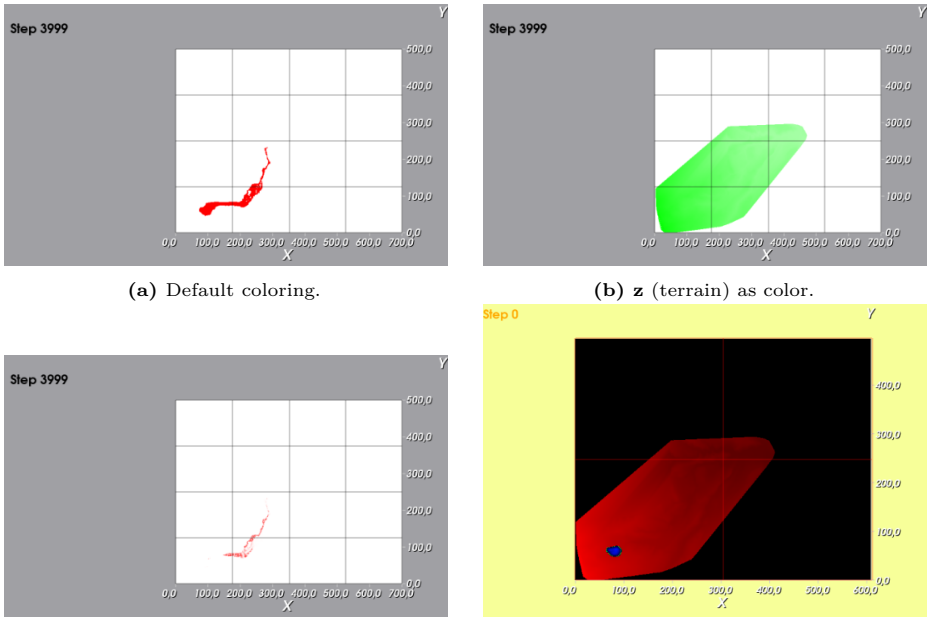
Crowd dynamics. In crowd simulations, the viewer can either rely on model-defined default rendering (categorical coloring driven by the model semantics) or explicitly map a numerical substate to a colormap. Figure 5 shows both modalities: default coloring provides an immediate separation between obstacles and walkable areas, whereas the dynamic floor field highlights transient motion traces. In this model, when a pedestrian leaves a cell it deposits a virtual trace that diffuses to neighboring cells and decays over time; the resulting field influences subsequent motion by promoting collective patterns and herding effects [3].

SciddicaT (debris/mud flow). In SciddicaT Figure 6 exposes substates such as terrain elevation (**z**) in figure 6b and transported material thickness (**h**) [1] in Figure 6c. In 2D mode, individual substates can drive coloring (e.g., **z** or **h**), and multiple layers can be combined through compositing to correlate morphology and transported mass in Figure 6d. In 3D mode, a selected substate (typically **z**) drives altitude while another substate drives surface color, supporting a combined representation of topography and evolving thickness Figures 7.

A key requirement of the proposed workflow is that the Qt/VTK visualizer remains stable while new CA models are introduced. Model-specific semantics (substates, formatting, default colors, missing-value policies, and serialization details) are provided as a dynamically loadable binary (shared library) generated alongside the execution descriptor. At load time, the visualizer resolves the exported entry points and instantiates a model adapter that exposes the `Cell`-level contract (Listing 1.1), enabling model-aware inspection without recompiling the application.

Why a compiled plug-in. A scripting layer (e.g., Python/Lua) would reduce ABI constraints, but it would also increase the implementation burden for model developers and may become a bottleneck when decoding large per-timestep dumps. For this reason, model semantics are provided as a native shared library dynamically loaded by the visualizer.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25



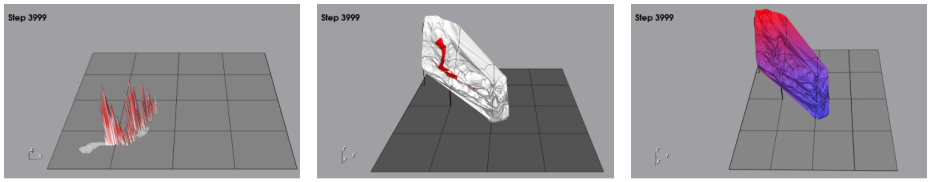
(a) Default coloring.

(b) **z** (terrain) as color.

(c) **h** (thickness) as color.

(d) Layered **h** over **z**.

Fig. 6: SciddicaT examples in 2D mode: default visualization, single-substate coloring, and layered visualization for combined inspection.



(a) **z** as altitude, **h** as color.

(b) **z** as altitude, default color.

(c) **z** as altitude, custom **h** gradient.

Fig. 7: SciddicaT examples in 3D mode: terrain elevation (**z**) drives altitude, while transported material thickness (**h**) can drive surface coloring.

Deployment modes. Two deployment modes are supported:

1. *Prebuilt*: the execution package ships a shared library already compiled for the target workstation.
2. *Rebuild*: the execution descriptor and per-timestep outputs are reused as-is, while the model module is rebuilt on the visualization machine from the exported headers/sources.

Portability is constrained by OS and ABI/toolchain compatibility (compiler version, standard library, calling conventions). When the visualization workstation differs from the execution platform, rebuilding the model module locally is recommended.

Finally, the same artifact-based design is compatible with future extensions that retrieve descriptors, outputs, and model binaries remotely (e.g., network shares, object storage, or thin client/server services close to the HPC allocation),

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

Table 1: Visualizer loading and rendering performance for two CA models (Ball and CrowdSimulation), three grid sizes, 100 timesteps, binary and text encoding. Grid size N means $N \times N$ cells; total cells $= N^2 \times 100$ steps (plus header rows). Throughput is computed as total cells divided by loading time. Rendering used a CPU-side Mesa software renderer.

Model	Format	Grid	Load (ms)	Render (ms)	cells/ms
Ball	binary	100×100	177	179	5 706
Ball	binary	500×500	2 334	2 310	10 819
Ball	binary	1000×1000	8 333	8 198	12 121
Ball	text	100×100	266	270	3 790
Ball	text	500×500	2 965	2 933	8 517
Ball	text	1000×1000	8 306	8 156	12 160
Crowd	binary	100×100	183	179	5 521
Crowd	binary	500×500	3 590	3 527	7 033
Crowd	binary	1000×1000	10 106	9 830	9 994
Crowd	text	100×100	341	335	2 961
Crowd	text	500×500	6 324	6 136	3 992
Crowd	text	1000×1000	19 264	18 839	5 243

enabling interactive inspection without manual staging on the visualization workstation.

Performance evaluation. To complement the qualitative case studies, Table 1 reports loading and rendering times measured on two representative models (Ball and CrowdSimulation, abbreviated CS) across three grid sizes (100×100, 500×500, 1000×1000) with 100 exported timesteps, in both binary and text encoding. Loading time covers the full data-ingestion phase (parsing the execution descriptor, reading per-step dumps via the index files, and populating VTK scalar arrays); rendering time covers the subsequent pipeline execution and display. Measurements were collected on a workstation equipped with an Intel Core i7-8750H CPU and 32 GB RAM; rendering relied on the CPU-side Mesa software renderer without discrete GPU acceleration, so the reported times represent a conservative baseline — hardware GPU rendering is expected to reduce rendering times further, particularly for large grids. Data loading was performed in single-threaded mode; parallel I/O is left as future work. Binary encoding consistently outperforms text, with loading times up to 1.9× faster for the CS model at the largest grid size. Throughput scales super-linearly with grid size in binary mode (from 5 706 to 12 121 cells/ms for Ball), reflecting improved I/O efficiency for larger sequential reads. Text mode exhibits higher variability between models due to per-character parsing overhead that grows with substate count (CS exposes more substates than Ball).

6 Conclusions

This paper introduced a portable integration layer that makes the OpenCal++ Visualizer handoff explicit and reproducible. By packaging an execution descriptor

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

together with per-timestep outputs and a dynamically loadable model binary, the proposed workflow decouples interactive inspection from the original execution backend while preserving model semantics required at visualization time (substate interpretation, formatting, and default visual attributes).

The Qt/VTK visualizer consumes these artifacts to provide interactive timestep navigation, multi-field inspection through configurable colormaps, optional 3D elevation mapping, and local picking for quantitative queries. The plug-in design avoids rebuilding the visualization application when new models are introduced and supports both prebuilt and rebuild deployment modes under ABI/toolchain constraints.

Future work will extend the integration toward remote and streaming access to execution packages, enabling interactive visualization of simulations produced on external HPC systems without manual data staging. Additional directions include in-situ coupling, volumetric rendering, vector field visualization, and scalable I/O backends for very large runs.

OpenCal++ is available at `https://github.com/alessioderango/OOpenCAL`.

Acknowledgments. This work was supported by the European Union – NextGenerationEU and by the Italian Ministry of Research (MUR) under the PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and by the Italian Ministry of Enterprises and Made in Italy under the PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP B29J24000200005). This research was partially funded by the Italian "ICSC National Center for HPC, Big Data and Quantum Computing" Project, CN00000013, and partially supported by the SERICS project (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Avolio, M., Di Gregorio, S., Mantovani, F., Pasuto, A., Rongo, R., Silvano, S., Spataro, W.: Simulation of the 1992 tessina landslide by a cellular automata model and future hazard scenarios. International Journal of Applied Earth Observation and Geoinformation **2**(1), 41–50 (2000). `https://doi.org/https://doi.org/10.1016/S0303-2434(00)85025-4`, `https://www.sciencedirect.com/science/article/pii/S0303243400850254`
2. Ayachit, U., et al.: ParaView Catalyst: Enabling in situ data analysis and visualization. In: Proc. First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV) (2015). `https://doi.org/10.1145/2828612.2828624`
3. Bazior, G., Wąs, J., Pałka, D.: Pedestrian dynamics model for high densities. Expert Systems with Applications **272**, 126775 (2025). `https://doi.org/https://doi.org/10.1016/j.eswa.2025.126775`, `https://www.sciencedirect.com/science/article/pii/S0957417425003975`
4. Cicirelli, F., Forestiero, A., Giordano, A., Mastroianni, C.: Parallelization of space-aware applications: Modeling and performance analysis. Journal of Network and Computer Applications **122**, 115–127 (2018)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25

5. De Rango, A., Furnari, L., Giordano, A., Senatore, A., D'Ambrosio, D., Spataro, W., Straface, S., Mendicino, G.: Opencal system extension and application to the three-dimensional richards equation for unsaturated flow. Computers and Mathematics with Applications **81**, 133–158 (2021). https://doi.org/10.1016/j.camwa.2020.05.017
6. De Rango, A., Spataro, D., Spataro, W., D'Ambrosio, D.: A first multi-gpu/multi-node implementation of the open computing abstraction layer. Journal of Computational Science **32**, 115–124 (2019). https://doi.org/https://doi.org/10.1016/j.jocs.2018.09.012, https://www.sciencedirect.com/science/article/pii/S1877750318303922
7. Giordano, A., D'Ambrosio, D., De Rango, A., Portaro, A., Spataro, W., Rongo, R.: Exploiting distributed discrete-event simulation techniques for parallel execution of cellular automata. In: Cicirelli, F., Guerrieri, A., Pizzuti, C., Socievole, A., Spezzano, G., Vinci, A. (eds.) Artificial Life and Evolutionary Computation. pp. 66–77. Springer International Publishing, Cham (2020)
8. Giordano, A., D'Ambrosio, D., Macrì, D., Rongo, R., Utrera, G., Gil, M., Spataro, W.: Opencal++: An object-oriented architecture for transparent parallel execution of cellular automata models. In: 2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). pp. 244–251 (2023). https://doi.org/10.1109/PDP59025.2023.00045
9. Giordano, A., De Rango, A., D'Ambrosio, D., Rongo, R., Spataro, W.: Strategies for parallel execution of cellular automata in distributed memory architectures. In: 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). pp. 406–413. IEEE (2019)
10. Giordano, A., De Rango, A., Rongo, R., D'Ambrosio, D., Spataro, W.: Dynamic load balancing in parallel execution of cellular automata. IEEE Transactions on Parallel and Distributed Systems **32**(2), 470–484 (2020)
11. Giordano, A., De Rango, A., Spataro, D., D'Ambrosio, D., Mastroianni, C., Folino, G., Spataro, W.: Parallel execution of cellular automata through space partitioning: the landslide simulation sciddicas3-hex case study. In: 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). pp. 505–510. IEEE (2017)
12. Grama, A., Gupta, A., Kumar, V.: Isoefficiency: measuring the scalability of parallel algorithms and architectures. IEEE Parallel & Distributed Technology: Systems & Applications **1**(3), 12–21 (1993). https://doi.org/10.1109/88.242438
13. Kumar, V.: Introduction to parallel computing, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., (2002)
14. Larsen, M., et al.: The ALPINE in situ infrastructure: Ascending from the ashes of strawman. In: Proc. In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization (ISAV) (2017). https://doi.org/10.1145/3144769.3144778
15. Renc, P., Pęcak, T., De Rango, A., Spataro, W., Mendicino, G., Wąs, J.: Towards efficient gpgpu cellular automata model implementation using persistent active cells. Journal of Computational Science **59**, 101538 (2022)
16. Spataro, D., D'Ambrosio, D., Filippone, G., Rongo, R., Spataro, W., Marocco, D.: The new sciara-fv3 numerical model and acceleration by gpgpu strategies. The International Journal of High Performance Computing Applications **31**(2), 163–176 (2017)
17. Zeigler, B., Moon, Y., Kim, D., Ball, G.: The devs environment for high-performance modeling and simulation. IEEE Computational Science and Engineering **4**(3), 61–71 (1997). https://doi.org/10.1109/99.615432

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29912-3_25