Implementation is Illustration: Zero-Overhead On-Demand Visualization for High-Performance Linear Algebra

Peter Rautek

King Abdullah University of Science and Technology (KAUST), Saudi Arabia

Abstract

Complex mathematical algorithms are often difficult to debug and explain. Traditional approaches to visualization usually require writing separate code, which often leads to divergence between the actual implementation and the visualization. We present a prototype implementation of "Implementation is Illustration" (`iii`), an experimental concept designed to bridge the gap between high-performance mathematical code and visual explanation. It explores a design where developers write standard mathematical code that can double as its own visualization, with zero runtime overhead when visualization is disabled. By augmenting standard types with optional hook capabilities, the library provides a "Visual Mode" for recording operations and a "Fast Mode" that completely compiles away the overhead. This allows developers to rapidly iterate between visualization and high performance execution. We validate this concept through a set of examples and performance benchmarks.

1. Introduction

In computer graphics and physical simulation, mathematical correctness is paramount. However, verifying and understanding complex geometric queries or algebraic manipulations remains a significant challenge. Developers often resort to writing auxiliary visualization code. This *debug visualization* is typically decoupled from the production code, leading to two implementations that must be kept in sync. As the project evolves, the visualization code often diverges, becoming misleading rather than helpful.

We explore this problem space with *Implementation is Illustration* (`iii`), an experimental library design built on the principle that the production source code serves as the single, authoritative specification for both execution and visual explanation. Through compile-time metaprogramming, instrumentation for visualization can be added or removed with zero runtime cost, so that the very same code that runs in production can also produce illustrations of its own behavior. Our design goals are as follows:

Visual Fidelity: The visualization must accurately reflect the mathematical operations performed by the code. Semantic tags allow developers to distinguish between conceptual primitives (e.g., points vs. directions) to produce the intended visualization.

Zero Overhead: When visualization is not needed, the library must impose no runtime cost. By using template specialization, all instrumentation compiles away in *Fast Mode*, matching the performance of optimized libraries such as Eigen.

Single Source of Truth: The production code itself must serve as the specification of the visualization, eliminating the need to maintain a separate visualizer that must be kept in sync.

Operation-Level Granularity: The system must capture individual operations (additions, dot products, etc.) and their history automatically. Instrumentation is part of the type system and records these details without developer intervention.

Our current prototype demonstrates how all design goals can be addressed. It targets linear algebra operations in a geometric context, such as points, directions, angles, lines, meshes, and cells, with plans to extend coverage to discrete scalar and vector fields. It is well-suited for visual computing algorithms that operate on 3D geometric primitives and where inspecting the meaning of intermediate results is important. The system is not designed for general linear algebra tasks such as dense matrix computations, GPU code visualization, or non-geometric data structures such as graphs.

2. Related Work

Algorithm Visualization. The history of algorithm visualization has shifted from pedagogical tools (e.g., illustration of sorting algorithms [Mar11]) and basic verification tools for production software, often termed *Visual Debugging* or *Data Flow Tracing* [DFIN02] to more sophisticated visualization frameworks. Our work is influenced by *Learnable Programming* [Vic12], which advocates for environments where data flow and program state are visible by default. While modern notebook environments [GP21] offer interactivity, they often struggle with the complexity of compiled, performance-critical geometry processing pipelines.

Visualization in Geometry Processing. The C++ geometry ecosystem typically bifurcates into computation and visualization. High-performance libraries like Eigen [GJ*10] utilize expression

DOI: 10.2312/egs.20261026

templates for speed but lack built-in visualization. Libraries like libigl [JP*18] and the widely adopted Polyscope [S*19] bridge this gap by acting as efficient *data sinks*. Polyscope excels at visualizing static quantities (meshes, vector fields) but operates primarily as a post-process inspection tool.

Unifying Notation and Code. Li et al. [LKJG21, LKJG22, LKC*24] present a family of domain-specific languages that compile Markdown-like mathematical notation into both typeset documents and executable code. These works share our design goal of a single source of truth that eliminates redundant encodings of the same mathematical knowledge. Their focus is document-centric (i.e., bridging the gap between written notation and code), whereas `iii` is code- and performance-centric (i.e., deriving visualization directly from a C++ implementation). In the future, a document processor that compiles mathematical notation to an `iii` backend could unite the benefits of both, providing high-quality typeset documents, executable code, and visualization from a single source.

Low Level Program Visualization. Industrial graphics debuggers like NVIDIA Nsight [NVI23] and RenderDoc [Kar23] provide deep inspection of GPU pipelines and shaders for debugging and performance optimization. Klein et al. [KBG*16] propose a framework for the interactive visualization of massively parallel programs. While we share the goal of making program behavior visible through non-invasive code augmentation, they target the visualization of GPU kernel runtime behavior (memory access patterns, thread divergence, etc.).

Tracing and Metaprogramming. Technically, our approach parallels architectures found in Differentiable Programming and Just-In-Time (JIT) compilers. Frameworks like Dr.Jit [JSRV22] and Taichi [HLA*19] employ operator overloading to build computational graphs for automatic differentiation and GPU kernel generation. We adapt these meta-programming techniques to the domain of explanation. By leveraging C++17 `if constexpr`, we ensure that our *Visual Mode* instrumentation compiles away completely for production in *Fast Mode*.

3. Design

The design of `iii` enables developers to selectively visualize parts of their code without impacting the rest of the application.

3.1. Selective Visualization

A key feature of the library is the ability to mix different vector types within the same codebase. The library supports three distinct categories of types that can interoperate:

Raw: Standard *Eigen library* types (e.g., Eigen::Vector3d) for legacy code or external interfaces. **Fast:** `iii` wrappers in *Fast Mode* (iii::Vector3<double, false>) which are compatible with raw types but provide the interface for potential visualization. **Visual:** `iii` wrappers in *Visual Mode* (iii::Vector3<double, true>) that carry instrumentation and are used for visualization.

This design allows developers to easily "opt-in" to visualize a subsystems. For example, a physics solver might be running in *Fast Mode* for performance, while collision detection is temporarily switched to *Visual Mode* to debug an intersection test. The compile-time switching allows for very fast iteration cycles between high-performance execution and detailed visual inspection.

3.2. Non-Invasive Augmentation

The `iii` library offers a way to augment variables in a way that is non-invasive for the performance when compiled in *Fast Mode*.

Semantic Augmentation. Mathematical types often carry implicit semantics that are lost using pure linear algebra. In `iii`, this information can be explicitly attached to the variables.

A 3-vector might represent a generic direction, a point in space, a color, or a normal vector. `iii` allows variables to be augmented with semantic tags that guide the visualization (e.g., drawing arrows vs. spheres) without affecting the mathematical behavior.

```
void visualize_ray_hit(Vec3 p, Vec3 d) {
    p.set_semantic("Point");     // a point
    d.set_semantic("Direction"); // a direction
    auto endpoint = p + d; // a point
}
```

Visualization Augmentations. Beyond type semantics, `iii` exposes direct control over visual attributes. Developers can use methods like `set_color()`, `set_visible()`, or `set_origin()` to fine-tune the output. For instance, `set_color` allows visually grouping related vectors by color, while `set_origin)` explicitly anchors a vector to a specific point in space for correct rendering.

Crucially, all these augmentations utilize 'if constexpr' blocks within the library. In *Fast Mode*, the entire body of these setter functions degenerate to no-ops and are removed by the optimizer.

4. Implementation

The core of `iii` relies on C++ templates and compile-time optimization to switch between *Fast* and *Visual* modes.

4.1. Dual-Mode Types

The library provides mathematical types (e.g., iii::Vector3) that act as wrappers around standard types (like Eigen::Vector3d). These wrappers differ based on a boolean template parameter, 'Record'.

In the **Visual Mode (Record = true)**, every mathematical operation (addition, dot product, etc.) is intercepted. The operation logs its inputs, the operation type, and the result to a centralized 'Recorder'. This builds a graph of operations for traversal and rendering.

In the **Fast Mode (Record = false)** mode, the wrapper functions are marked for inlining. Modern C++ compilers are able to optimize away these wrapper layers.

The following code snippet demonstrates the compile-time switching mechanism using constexpr that enables the zero-overhead performance of our library in *Fast Mode*:

```
template <typename Scalar, bool Record>
class Vector3 // { [...]
    // operations are conditionally instrumented
    template <bool R = Record>
    auto dot(const Vector3& other) const {
        if constexpr (R) { // code for Visual Mode }
        return data.dot(other.data);
    } // [...] };
```

4.2. Usage

Developers can easily switch between modes by changing a template parameter, while the core algorithm remains unchanged.

```
// Algorithm implementation (agnostic to mode)
template <typename Vec3>
Vec3 comp_physics(Vec3& pos, Vec3& vel) {
    return pos + vel * 0.1; // math syntax
}
// 1. production use (Fast Mode)
// compiles to raw arithmetic instructions
using FastVec = iii::Vector3<double, false>;
auto res_fast = comp_physics(FastVec(0,0,0), FastVec(1,1,1));
// 2. illustration use (Visual Mode)
// records all operations for visualization.
using VisualVec = iii::Vector3<double, true>;
auto res_vis = comp_physics(VisualVec(0,0,0), VisualVec(1,1,1));
```

The key difference is that `VisualVec` instantiates the instrumentation hooks, while `FastVec` compiles them away, ensuring that the final executable for production has no overhead.

4.3. Function Wrappers

To capture the context of operations, `iii` provides a Function wrapper. This utility logs function entry and exit, along with argument values. This hierarchical logging allows the visualization to reconstruct the call stack and group geometric operations.

```
// function generating output for visualization
auto math_func = iii::Function("Math Helpers", [](auto a, auto b){
        auto result = a.cross(b);
        return result.normalized();
    });
```

4.4. Interactive Parameters

To enable real-time exploration, `iii` provides a Parameter<T> class that links C++ variables to GUI controls in the visualizer. In *Visual Mode*, the constructor registers the variable with a global map, allowing the external UI to override its value at runtime. In *Fast Mode*, the registration logic is compiled out, and the class collapses to a simple wrapper around the primitive type, often optimized into a standard register allocation.

```
// Define a tunable parameter
iii::Parameter<double> gravity("Gravity", 9.81);

void step(double dt) {
    // gravity behaves like a double
    // in GUI: "Gravity" slider appears
    velocity -= gravity * dt;
}
```

4.5. Visualization Backend

To validate the `iii` concept, we implemented a prototype visualization backend that is completely decoupled from the tracing stage. The `iii` library acts purely as an event producer, emitting a stream of lightweight, data-oriented events (e.g., ‘EventCreate‘, ‘EventMove‘, ‘EventBinaryOp‘) to registered listeners.

The backend implementation consumes these events to reconstruct the scene state. It maintains a mapping from `iii` object IDs to graphical instances (meshes, lines, UI elements). This architecture ensures that the core mathematical library has no dependency on graphics APIs like OpenGL or Vulkan. The trace can be streamed over a network socket to a remote viewer, saved to disk for replay, or visualized in real-time within the same process.

5. Case Studies and Results

We demonstrate the utility of `iii` with two case studies and verify their performance characteristics. Figures 1 and 2 show the visualizer that generates the trace of the algorithms of the two case studies when run in *Visual Mode*. The proof of concept implementation of the visualizer serves as a visual proof that our implementation works. It is decoupled from the `iii` library, which means that the `iii` library has no notion of a visualizer. On the other hand our visualizer includes the `iii` library which enables us to interactively change simulation parameters. The GUI on the left of Figures 1 and 2 is automatically generated by the visualizer. It contains a timeline that allows the user to step through the recorded trace as well as pause and resume playback. A *Code* area shows the currently executed code or a compile time defined caption of the recorded step. The *Parameters* GUI shows the parameters that can be modified by the user which will re-run and re-record the simulation. The *Legend* area in the GUI displays the currently instantiated `iii` objects. A recording of the visualizer can be seen in the supplementary video.

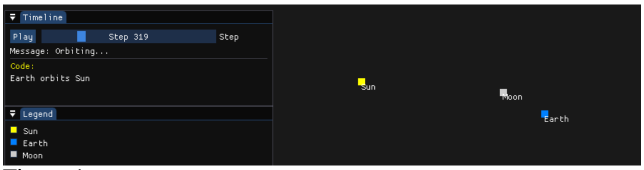

Figure 1: *Solar system simulation visualized. The orbits are traced automatically by the position updates.*

5.1. Solar System Simulation

In this example, we simulate a simple hierarchical planetary system. The `iii` library allows us to attach semantic properties like labels and colors directly to the mathematical entities. The source code demonstrates how visual properties are set once, and the position is updated in the simulation loop.

```
Vector3 sun, earth, moon;
sun.set_label("Sun");
sun.set_color(1.0, 1.0, 0.0);
earth.set_label("Earth");
earth.set_color(0.0, 0.5, 1.0);
// [...] more initialization
while (true) { // [...] Simulation Loop
  double earth_x = std::cos(t) * earth_r;
  double earth_z = std::sin(t) * earth_r;
  // updates are recorded for visualization
  earth[0] = earth_x;
  earth[1] = 0.0;
  earth[2] = earth_z;
}
```

5.2. Inverse Kinematics

We implemented a Cyclic Coordinate Descent solver. This example demonstrates mixing raw linear algebra for internal calculations and visual types for the state.

```
// iii types for optional visualization
std::vector<Vector3> joints;
// [...] joints initialized with labels/colors
using VecMath = Vector3::Base; // pure Eigen
// [...] Solver Loop
for (int j = i + 1; j <= num_segments; ++j) {
  VecMath p_current = joints[j]; // read as raw
  // perform rotation using raw Eigen math
  VecMath v_rot = rotation * (p_current - r_vec);
  // assign back to Vector3:
  joints[j] = r_vec + v_rot;
}
```


Proceedings published by Eurographics - The European Association for Computer Graphics.

Crucially, the inner loop uses VecMath (the raw Eigen base class) for calculations that do not need to be visualized, and only assigns back to the joints variable (visual types) at the end of the step.

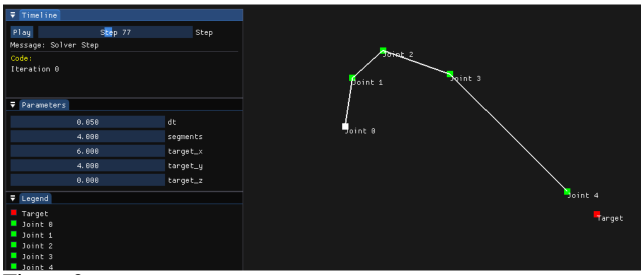

Figure 2: *Inverse Kinematics solver. The green joints adjust iteratively to reach the red target.*

5.3. Performance

Validating the claim that the library can run with zero overhead is critical for adoption and the major result of our experiments. We benchmarked native Eigen types against our *Fast Mode* as well as the *Visual Mode*. The benchmarks include a mixed workload of vector operations, scalar multiplications, parameter access, as well as the two case studies and were run over 50 million iterations.

Operation	Baseline	iii (Fast)	iii (Visual)
Mixed Vec Arith.	2.8 ns	2.8 ns	1,400 ns
Param Access	2.2 ns	2.2 ns	2.2 ns
Solar Sys. (per iter)	16 ns	16 ns	3,500 ns
Inv. Kinematics (per iter)	85 ns	85 ns	33,000 ns

Table 1: Performance comparison. Baseline refers to Eigen::Vector3d for arithmetic, and native double for parameters. Visual includes overhead.

As shown in Table 1, the performance of `iii` in *Fast Mode* is the same as with Eigen directly, confirming that the abstraction layer compiles away completely. When *Visual Mode* is enabled via the template parameter the performance drops dramatically as expected. The benchmarks range in complexity from approximately 20 LoC (Mixed Vec Arithmetic) to 80 LoC (Solar System) and 140 LoC (Inverse Kinematics). The overhead measured in *Visual Mode* stems entirely from the tracing stage (event creation and storage); visualization rendering is fully decoupled and asynchronous and is not included in these timings. The benchmarks demonstrate that `iii` *Fast Mode* results in identical performance to native Eigen, which validates the implementation and shows that achieving the design goals of *Implementation is Illustration* is possible, when using modern C++.

6. Conclusion and Future Work

Our prototype implementation of `iii` demonstrates that it is reasonable to unify implementation and visualization in performance-critical C++ code. By leveraging modern metaprogramming and compile-time optimization, developers can gain deep visual insights into their algorithms without sacrificing the performance required for production systems. It is important to emphasize that the current iteration of `iii` is a prototype intended to validate the feasibility of this approach. The long term goal is to make a drop-in replacement for the Eigen library. This paper is limited to a proof of concept confirming that the abstraction penalties typically associated with visualization wrappers can be effectively compiled away.

The `iii` library is strictly decoupled from the visualization frontend. The library acts solely as an event producer, emitting a stream of operations without dependency on any specific graphics API. The current visualization backend utilizes OpenGL, serving as a proof of concept of the consumption of the event stream.

In the future we want to implement a high-fidelity illustration renderer based on the Vulkan API. While the current OpenGL implementation validates the data flow, a Vulkan-based backend will allow for significantly higher draw-call counts and more sophisticated shading models, enabling the visualization of large-scale physics simulations and complex geometry processing pipelines in real-time. We also plan to expand the wrapper coverage to support the full breadth of the Eigen API, moving `iii` from a prototype to a production-ready tool for the graphics community.

References

[DFIN02] Demetrescu C., Finocchi I., Italiano G. F., Näher S.: *Visualization in Algorithm Engineering: Tools and Techniques*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 24–50.

[GJ*10] Guennebaud G., Jacob B., et al.: Eigen. https://libeigen.gitlab.io, 2010.

[GP21] Granger B. E., Pérez F.: Jupyter: Thinking and storytelling with code and data. *Computing in Sci. and Eng. 23*, 2 (2021), 7–14.

[HLA*19] Hu Y., Li T.-M., Anderson L., Ragan-Kelley J., Durand F.: Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph. 38*, 6 (Nov. 2019).

[JP*18] Jacobson A., Panozzo D., et al.: libigl: A simple C++ geometry processing library, 2018. Accessed: 2026-01-15. URL: https://libigl.github.io/.

[JSRV22] Jakob W., Speierer S., Roussel N., Vicini D.: DR.JIT: A Just-In-Time Compiler for Differentiable Rendering. *Transactions on Graphics (Proceedings of SIGGRAPH) 41*, 4 (2022).

[Kar23] Karlsson B.: Renderdoc, 2023. Accessed: 2026-01-15. URL: https://renderdoc.org/.

[KBG*16] Klein T., Bruckner S., Gröller M. E., Hadwiger M., Rautek P.: Towards interactive visual exploration of parallel programs using a domain-specific language. In *Proceedings of the 4th International Workshop on OpenCL* (2016), IWOCL '16, ACM.

[LKC*24] Li Y., Kamil S., Crane K., Jacobson A., Gingold Y.: IMESH: A DSL for mesh processing. *ACM Trans. Graph. 43*, 5 (2024).

[LKJG21] Li Y., Kamil S., Jacobson A., Gingold Y.: I♡LA: Compilable markdown for linear algebra. *ACM Trans. Graph. 40*, 6 (2021).

[LKJG22] Li Y., Kamil S., Jacobson A., Gingold Y.: H♡rtDown: Document processor for executable linear algebra papers. In *SIGGRAPH Asia 2022 Conference Papers* (2022), SA '22, Association for Computing Machinery.

[Mar11] Martin D. R.: Sorting algorithms animations, 2011. URL: https://www.toptal.com/developers/sorting-algorithms.

[NVI23] NVIDIA: Nvidia nsight graphics, 2023. Accessed: 2026-01-15. URL: https://developer.nvidia.com/nsight-graphics.

[S*19] Sharp N., et al.: Polyscope, 2019. Accessed: 2026-01-15. URL: https://polyscope.run.

[Vic12] Victor B.: Learnable programming: Designing a programming system for understanding programs. http://worrydream.com/LearnableProgramming/, 2012. Accessed: 2026-01-15.