The Journal of Open Source Software

ACEngine: Augmented Reality UNIX C++ Engine for Enhanced Visual Guidance in Digital Fabrication

Andrea Settimi[1¶], **Hong-Bin Yang**[1], **Julien Gamerro**[2], and **Yves Weinand**[1]

1 IBOIS EPFL, Switzerland **2** Independent Researcher, Switzerland ¶ Corresponding author

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Arfon Smith

Reviewers:

- @preetishkakkar
- @ApocalyVec

Submitted: 30 December 2024
Published: unpublished

Summary

Augmented Carpentry Engine (ACEngine) is a lightweight and fast-developing UNIX C++ engine for prototyping AR applications leveraging bleeding-edge robotic vision research for digital fabrication. It features a modular layer-stack flow, a geometry framework for managing 3D objects, a computed feedback system for visual guidance, and an AR rendering system for synthesizing digital instructions into a simple monocular camera feed.

Statement of need

ACEngine (ACE) addresses critical limitations in existing augmented reality (AR) tools for digital fabrication. CompasXR (Kenny et al., 2024), the only open-source AR tool available in the digital fabrication field, provides a valuable common platform, particularly for assembly tasks. However, it currently lacks a streamlined integration pipeline for advanced robotic vision technologies due to its reliance on Unity (Unity Technologies, 2023) and the Windows operating system (OS). In the field of AR fabrication, developers from the current Incon.ai (Furrer et al., 2024) represent the peak of AR engine innovation with robotic vision algorithm integration for digital fabrication in research (Mitterberger et al., 2020; Sandy et al., 2016; Sandy & Buchli, 2018), nevertheless, its codebase remains unavailable to the public.

AC aims to fill this gap by providing a lightweight, open-source, and UNIX-compatible C++ engine for AR applications in digital fabrication. Its software architecture is similar to existing free software engines (T. ezEngine Contributors, 2024; T. T. 3D. Contributors, 2024; Linietsky et al., 2024), yet it prioritizes rapid prototyping, flexibility, and customization for extended reality (XR) manufacturing using accessible sensors and hardware. Unlike feature-rich game engines with excessive functionalities or proprietary constraints (Epic Games, 2019; Unity Technologies, 2023), ACE is lightweight, aided by the adoption of a bloat-free UI system (T. D. Contributors, 2024), with a render powered by OpenGL (Woo et al., 1999), and maintains full compatibility with Linux systems, crucial for integrating the latest open-source robotic vision technologies in AR manufacturing.

The following sections provide an overview of the key components of the ACEngine, including the layer-stack flow, geometry framework, computed feedback system, and AR rendering system.

Layer-stack flow

The main AR engine is managed by a layer-stack flow. Designed as a modular system, each layer encapsulates the code for a specific domain of the AR application, such as camera processing, sensor's self-localization, object tracking, UI, and rendering. The general order and expansion of these layers can be configured in the top-level main file `ACApp.cpp`. This architecture provides flexibility to customize key AR features as needed, including integrating new sensors, modifying the rendering pipeline, or adapting camera pose estimation methods. For instance, users can implement pose estimation based on tags (Muñoz-Salinas et al., 2019), features (Campos et al., 2021), or hybrid (Settimi et al., 2024) approaches as supported by the software out of the box.

Each layer in the stack inherits from a superclass interface defined in `Layer.h`, which includes event-like methods triggered at various points during frame processing (e.g., `OnFrameAwake()`, `OnFrameStart()`, etc). These methods are invoked by the main `Run()` function in the singleton application loop from `Application.h`. This design allows application tasks to be containerized and executed sequentially while facilitating data exchange between specific layers through the `AIAC_APP` macro, enabling the retrieval of any particular layer data. Exchange between layers can also take place in a more structured way with the integrated event system (`ApplicationEvent.h`), which is capable of queuing events from layers and trigger them in the next main loop.

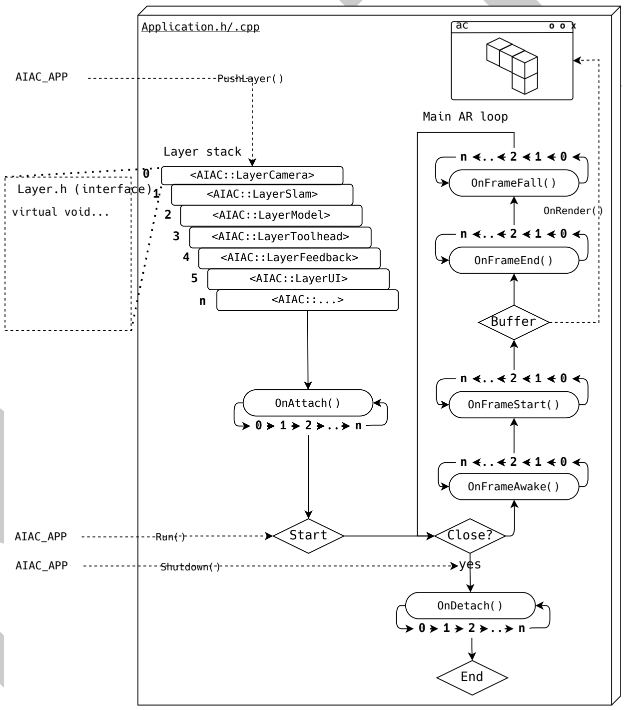

Figure 1: Illustration of the layer-stack design and the main loop for the AR engine.

Geometry framework

The geometry framework provides a unified infrastructure for handling all 3D objects in the scene, including CAD models, scanned models, and fabrication instructions. This framework enables easy interaction between application layers and 3D objects while being tightly integrated with the rendering system, which implicitly manages OpenGL resources, simplifying the workload for application layers.

The geometry is organized into the following primitive shapes: point, line, circle, cylinder, polyline, triangle, mesh, and text. Each of them is a class (e.g., `GOPoint`, `GOLine`, `GOCircle`, etc.) that inherits the base class `GOPrimitive`, where "GO" stands for Geometry Object. The base class manages general attributes and provides interfaces such as visibility and transformation, while the subclasses handle their specific data and functionality.

The geometric system is designed to maintain a global registry, called `GORegistry`, which tracks all geometrical objects (GOs). When a GO is created, it is added to the scene using a static function specific to its shape, such as `GOPoint::Add()` or `GOLine::Add()`. Upon initialization, each GO receives a unique UUID and registers itself in a global hash table. This table is accessible throughout the system, allowing application layers to retrieve specific objects by their UUIDs or to iterate through all objects in order to access or modify their properties. This design ensures that all layers interact with the same geometries present in the scene.

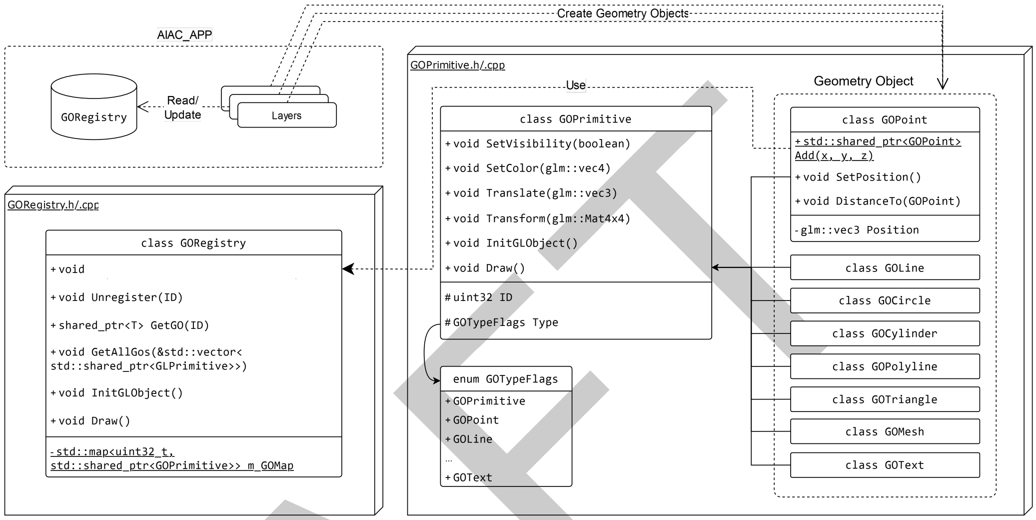

Figure 2: Structure of the Geometry Framework.

Computed Feedback System

The `LayerFeedback.h` module manages the computation of all essential data required to provide visual guidance to users during the fabrication process. Feedback computation primarily relies on data retrieved from two preceding layers:

1. `LayerModel.h`: contains the execution model and geometries associated with the currently active hole or cut.
2. `LayerToolhead.h`: provides similar information, but specific to the toolhead currently attached to the tool.

Feedback is categorized based on similar operations, such as drilling (`HoleFeedback.h`), circular cutting (`CutCircularSawFeedback.h`), and chainsaw cutting (`CutChainSawFeedback.h`). Each feedback category inherits from an interface class (`AIAC/Feedback/FabFeedback.h`), which defines high-level control functions like `Update()`, `Activate()`, and `Deactivate()`.

The visual guidance for each tool may consist of multiple visual cues, most of which are implemented using the template `FeedbackVisualizer.h`. These internal components (e.g., `CutBladeThicknessVisualizer.h` or `CutPlaneVisualizer.h`) handle their own geometric visual cue calculations and store representations as GO instances in a member vector of the corresponding superclass. Visualization of these GO elements, and thus the feedback itself, can be selectively enabled or entirely toggled on/off using the `Activate()` and `Deactivate()` functions.

The Journal of Open Source Software

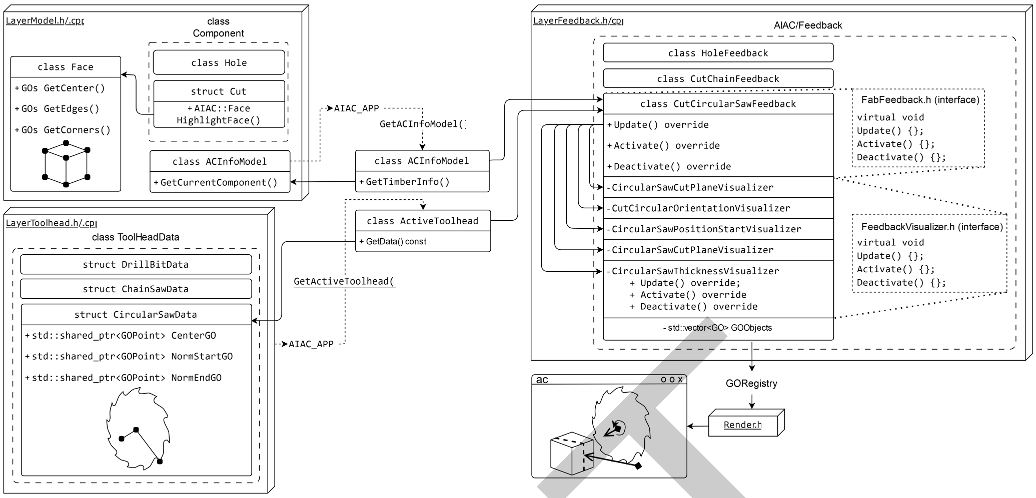

Figure 3: Dataflow for the functioning of the Augmented Carpentry's feedback system.

AR rendering

The rendering system manages two viewports: the main AR view and the 3D viewport. The AR view combines captured images with virtual objects, such as CAD models and feedback graphics, to provide clear and intuitive instructions. The 3D viewport serves as an interface for navigating the entire scene, enabling users to easily inspect different components or specific details. The system consists of the following key components:

1. `Renderer.h`: defines the core logic of the rendering pipeline and manages essential attributes.
2. `Viewport.h`: handles the sub-frame buffer. The renderer calls `Activate()` to switch the buffer for rendering.
3. `GLObject.h`: helps GO manage OpenGL resources, allocating memory and buffering data for rendering. Each GO may contain one or multiple GLObjects stored in a list. By invoking `Draw()`, the content is rendered to the currently active frame buffer.

The `Renderer::OnRender()` function triggers after all layers have finished processing. During this stage, `RenderMainView()` uses positional data from `LayerSLAM` to compute a projection matrix and overlay scene geometry from the `GORegistry` onto the captured image, producing an accurate AR view. Next, `RenderGlobalView()` directs output to a 3D viewport, adjusting the projection based on user navigation.

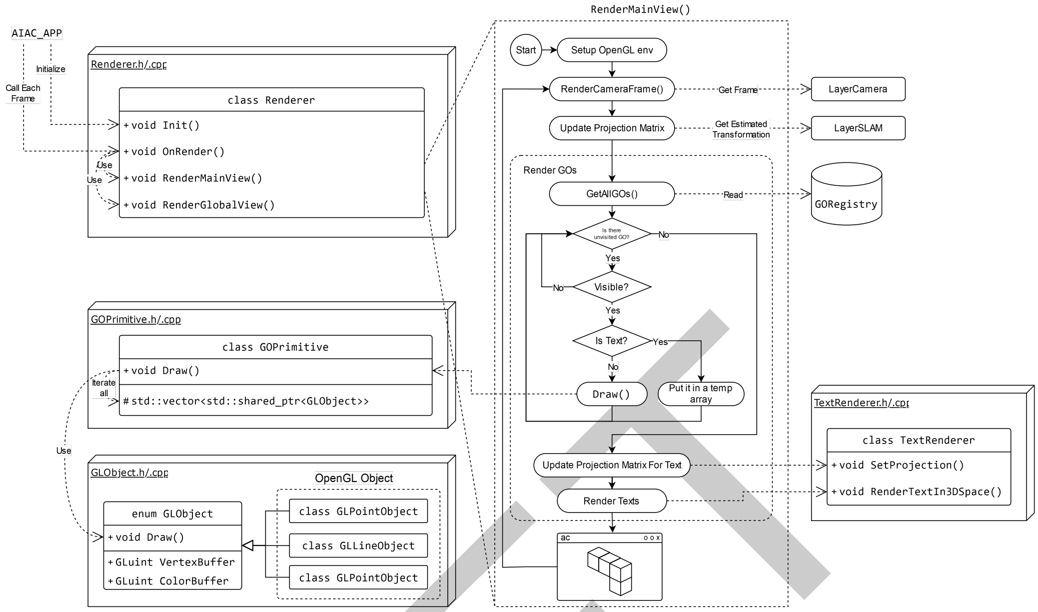

Figure 4: Dataflow of the rendering system and the pipeline for AR rendering.

Acknowledgments

We would like to thank all the contributors to the ACEngine project, including the developers, researchers, and users who have provided valuable feedback and suggestions. Special thanks to the GIS and the Center for Imaging EPFL groups, for their support throughout the development process.

References

Campos, C., Elvira, R., Rodriguez, J. J. G., Montiel, J. M. M., & Tardos, J. D. (2021). ORB-SLAM3: An accurate open-source library for visual, visual inertial, and multimap SLAM. *IEEE Transactions on Robotics*, *37*(6), 1874–1890. https://doi.org/10.1109/tro.2021.3075644

Contributors, T. D. (2024). *Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies* (Version 1.91.5). https://github.com/ocornut/imgui

Contributors, T. ezEngine. (2024). *EzEngine engine* (Version 4.0.3). https://ezengine.net/

Contributors, T. T. 3D. (2024). *Torque 3D engine* (Version 24.9). https://github.com/TorqueGameEngines/Torque3D

Epic Games. (2019). *Unreal engine* (Version 4.22.1). https://www.unrealengine.com

Furrer, F., Stich, M., Regenass, F., & Mansfield, B. (2024). *Instructive Construction*. https://incon.ai/.

Kenny, J., Mitterberger, D., Casas, G., Alexi, E., Gramazio, F., & Kohler, M. (2024). *COMPAS XR: Extended reality workflows for the COMPAS framework*. https://github.com/compas-dev/compas_xr/. https://doi.org/10.5281/zenodo.12514526

Linietsky, M., Manzur, A., Verschelde, R., & others, many. (2024). *Godot Engine – Multi-platform 2D and 3D engine* (Version 4.3). https://github.com/godotengine/godot?tab=coc-ov-file

Mitterberger, D., Dörfler, K., Sandy, T., Salveridou, F., Hutter, M., Gramazio, F., & Kohler, M. (2020). Augmented bricklaying. *Construction Robotics*, *4*(3-4), 151–161. https://doi.org/10.1007/s41693-020-00035-8

Muñoz-Salinas, R., Marín-Jimenez, M. J., & Medina-Carnicer, R. (2019). SPM-SLAM: Simultaneous localization and mapping with squared planar markers. *Pattern Recognition*, *86*, 156–171. https://doi.org/10.1016/j.patcog.2018.09.003

Sandy, T., & Buchli, J. (2018). Object-based visual-inertial tracking for additive fabrication. *IEEE Robotics and Automation Letters*, *3*(3), 1370–1377. https://doi.org/10.1109/lra.2018.2798700

Sandy, T., Giftthaler, M., Dorfler, K., Kohler, M., & Buchli, J. (2016, May). Autonomous repositioning and localization of an in situ fabricator. *2016 IEEE International Conference on Robotics and Automation (ICRA)*. https://doi.org/10.1109/icra.2016.7487449

Settimi, A., Yang, H.-B., Gamerro, J., & Weinand, Y. (2024). TSLAM: A tag-based object-centered monocular navigation system for augmented manual woodworking. *Construction Robotics*, *8*(1). https://doi.org/10.1007/s41693-024-00118-w

Unity Technologies. (2023). *Unity* (Version 2023.2.3). https://unity.com/

Woo, M., Neider, J., Davis, T., & Shreiner, D. (1999). *OpenGL programming guide: The official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc.