

Forge: Extending Anvil for Visual Evaluation of Rendering Pipelines

Kevin Napoli^a, Keith Bugeja^b and Sandro Spina^c

CGVG, Department of Computer Science, Faculty of ICT, University of Malta, Msida, Malta
{kevin.napoli.10, keith.bugeja, sandro.spina}@um.edu.mt

Keywords: Distributed Rendering Evaluation, Anvil, Graphical Applications, Rendering Pipeline Visualisation.

Abstract: This paper introduces Forge, an extension of Anvil, aimed at enhancing evaluation processes in computer graphics pipelines. Forge addresses critical challenges in rendering systems, such as ensuring consistent configurations, minimising human error, and increasing reproducibility of experimental results. By decoupling evaluation logic from rendering engines, Forge facilitates seamless comparisons across different systems without manual configuration. The framework's architecture supports decentralised evaluations, enabling operations across diverse environments and platforms. This flexibility allows for both local and remote evaluations, making Forge adaptable for a broad range of research applications, from small-scale experiments to comprehensive distributed rendering evaluations. Through case studies, this paper demonstrates Forge's effectiveness in verifying rendering techniques, comparing performance, and aiding development of new algorithms, thereby providing a robust solution for accurate and reliable comparative studies in the field of computer graphics.

1 INTRODUCTION

Software evaluation is crucial for assessing the quality of computer graphics (CG) software, especially in rendering systems. Unlike basic software evaluations, where metrics can be measured in direct terms, graphics software evaluation is far more complex. It often involves subjective factors like image quality, visual fidelity, and rendering realism, which are challenging to quantify. In Monte Carlo rendering, for instance, both image quality and rendering performance are key aspects of evaluation, with metrics such as Structural Similarity Index (SSIM) frequently used to assess how accurately a renderer simulates real-world lighting, textures, and shading.

A major challenge is maintaining consistent configurations across rendering systems. Small differences in scene setup, camera positioning, or lighting can lead to invalid comparisons, especially when generating reference images for evaluation. Standardisation is crucial for reliable, reproducible results. Manual configuration is susceptible to human error through incorrect settings or misinterpretation. The problem is worsened by evaluation logic being tightly coupled with rendering engines, making it difficult to separate evaluation from the tools themselves.

To address these challenges, we introduce Forge, an extension of Anvil, originally designed for visual debugging of physically based rendering (PBR) (Napoli et al., 2022). Anvil operates on higher-level abstractions, working with structured data types such as atoms and molecules that represent common 3D elements like vectors and rays. In this paper, we broaden Anvil's scope to evaluate CG pipelines, offering a solution that automates, standardises, and streamlines the evaluation process.

Forge builds on Anvil's modular architecture, providing a centralised framework that decouples evaluation logic from rendering engines. This allows researchers to compare and measure rendering algorithms across different systems without manual configuration or tool-specific constraints. Forge also introduces a WebSocket System for decentralised, distributed evaluations, allowing systems to run locally or remotely across various languages and platforms. This approach enhances scalability and flexibility, making it suitable for diverse research and development scenarios, from small-scale experiments to distributed render farms. By addressing common issues in graphics software evaluation such as human error, configuration mismatches, and fragmentation across tools, Forge improves the reproducibility of results.

The case studies presented demonstrate Forge's capabilities in verifying rendering techniques, comparing their performance, and enabling new algorithm

^a  <https://orcid.org/0000-0001-9749-0509>

^b  <https://orcid.org/0000-0002-3111-1251>

^c  <https://orcid.org/0000-0001-7197-410X>

development. By integrating evaluation workflows, we provide a tool for research and development in CG, streamlining the process of testing and validating rendering software across applications.

2 RELATED WORK

Anvil (Napoli et al., 2022) is a visual debugging tool for PBR applications that simplifies the identification of rendering issues. Unlike traditional low-level debuggers, Anvil visualises and analyses higher-level data structures called *atoms* and *molecules*, which represent fundamental 3D graphics components like vectors and rays, mapped to user data through reflection. By monitoring these structures, Anvil provides insights into the rendering process through visualisations and analysis tools (*systems*), allowing users to identify anomalies and set breakpoints to pinpoint problems during runtime. Built on Entity Component System (ECS), Anvil’s modular design enables users to create and share debugging tools for their needs. ECS is an architectural pattern widely used in game development (Unity Technologies, 2024) and performance-critical software systems. ECS separates data (components) from behaviour (systems), using entities as unique identifiers. Evolving from component-based architectures (Martin, 2007; Bilas, 2002), it offers improved performance through cache-friendly data organisation and enhanced code reusability. Its data-oriented design principles make it well-suited for scenarios with large numbers of objects, leading to its adoption in modern game engines and software frameworks. Although Forge builds upon Anvil, its architectural design is focused on evaluation workflows rather than serving solely as a debugging framework.

RealXtend (Alatalo, 2011) introduced a modular approach to virtual world architectures based on the Entity-Component-Attribute model. It emphasised flexibility and network synchronisation, combining ECS architecture with efficient real-time communication. It allowed for dynamic component addition and real-time updates, facilitating the creation of interactive and extensible virtual environments. Dahl et al. (Dahl et al., 2013) extended this work, using WebSocket – a protocol enabling full-duplex communication over a single TCP connection widely adopted across major web browsers and servers (Wang et al., 2013) – to communicate with web-based clients, illustrating the potential of ECS-based systems for distributed environments. While realXtend focuses on flexible and extensible virtual world architectures, its goals differ from those of Forge, which is primarily

designed to standardise and streamline the evaluation of rendering systems rather than supporting virtual world simulations.

Elements (Papagiannakis et al., 2023) provides another example of an ECS-based framework, but with a focus on education. It is a lightweight, open-source Python tool designed to teach CG concepts by implementing an ECS architecture within a scene graph. Elements allows students to explore modern CG pipelines, bridging theory and practice. While it shares a similar architectural foundation with Forge, its primary focus is pedagogical, whereas Forge aims to evaluate, debug and standardise rendering techniques, with support for distributed computation and performance assessments.

NVIDIA’s Falcor (Kallweit et al., 2022) provides a platform for real-time rendering research by offering a high-level abstraction over modern graphics APIs like DirectX 12 and Vulkan. Falcor enables rapid prototyping of advanced rendering techniques, featuring a flexible render graph system, built-in profiling tools, and support for shader hot reloading. However, while Falcor accelerates research through quick composition of rendering pipelines, it does not address the need for standardised evaluations across different rendering engines and lacks built-in network capabilities, limiting its ability to facilitate distributed or remote evaluations. Furthermore, it only provides access to FLIP (Andersson et al., 2021; Andersson et al., 2020), which specialises in detecting perceptual differences in rendered images and Mean Squared Error (MSE), which measures pixel-level differences between images.

More broadly, there has been a growing focus on replicability and reproducibility. Bonneel et al. (Bonneel et al., 2020) emphasise the importance of making research code available to ensure that results can be replicated. Despite these efforts, most available research code remains specific to individual studies or tools, rather than forming part of a reusable, modular framework. While current initiatives focus on improving replicability within specific projects, they do not provide a generalised, system-agnostic platform for evaluating and comparing rendering techniques across different engines and configurations.

To the best of our knowledge, no existing tool fully addresses the niche that Forge aims to fill: a standardised, modular framework for evaluating rendering techniques across diverse pipelines. This may be because most graphics engines are typically evaluated using custom-built tools tailored to specific engines and rarely shared publicly. This fragmentation has led to a lack of standardised tools for cross-engine comparisons, which motivates Forge’s development.

3 FORGE: AN ANVIL EVALUATION FRAMEWORK

Anvil is a C++ library to which applications can link, allowing them to submit entities along with their relevant components. These components, termed *molecules*, consist of *atoms* such as position and direction, and are well-defined and documented. For instance, a path debug entity requires a *path* component, and systems become active when all necessary components are present. Forge¹ extends Anvil’s functionality while preserving its core design principles, providing powerful analytical tools with minimal setup and disruption. These tools are implemented as systems within the ECS design pattern since Anvil supports extensibility through the addition of systems which can be registered as plugins. While Anvil’s library-based approach is effective for debugging, Forge operates as a standalone application to better facilitate rendering pipeline evaluation. This architectural shift enables centralised control and permits the implementation of standardised evaluation workflows, requiring renderers to expose compatible interfaces with Forge systems.

3.1 Design

Forge is an application framework that incorporates all of Anvil’s functionality but manages execution internally. It maintains its own component repository, system registry, and entities. The `tick` method, which executes systems sequentially in their order of registration, is responsible for system execution by passing entities as parameters. Throughout this paper, sequential `tick` calls are denoted as `tick0`, `tick1`, etc. Before evaluation begins, Forge requires a startup configuration. Figure 1 demonstrates a general configuration file that specifies an evaluation and loads the Evaluation System, with optional configuration parameters available for each evaluation.

```
{
  "evaluations": [{
    "name": "EvaluationSystem",
    "config": { "key": "value" }
  }]
}
```

Figure 1: JSON configuration for a general evaluation system.

Figure 2 illustrates the evaluation process which is typically divided into three phases. Following startup

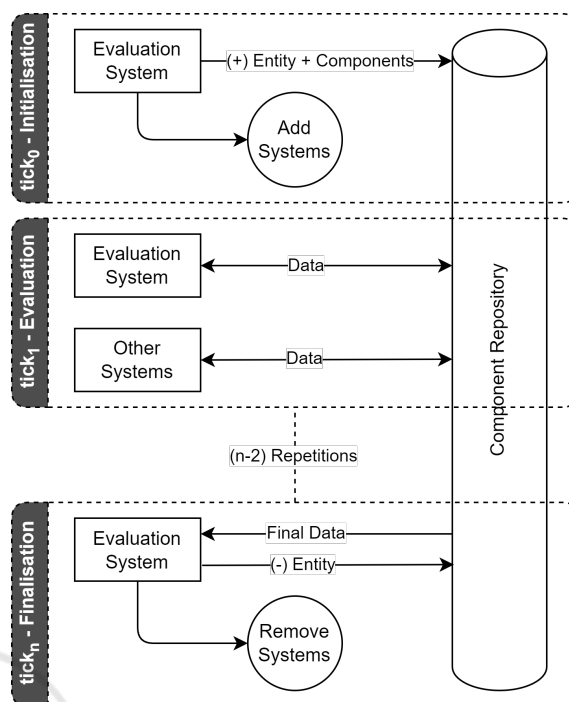


Figure 2: General process diagram for one evaluation. (+) is add, (-) is remove.

configuration, the *initialisation* phase (`tick0`) loads and initialises required systems. The evaluation system typically adds necessary entities and components during `tick0` or `tick1`. Between `tick1` and `tickn-1`, in the *evaluation* phase, systems perform their evaluation tasks where results are iteratively collected and processed - for instance, retrieving rendering images and then computing similarity scores. At `tickn`, in the *finalisation* phase, where the evaluation result is presented and where cleanup occurs: the evaluation system removes all systems added during `tick0` and any entities with their components created during evaluation. If specified in the startup configuration, additional evaluations may follow. Evaluation systems are thus responsible for system configuration, bootstrapping, and data collection throughout this process.

3.1.1 Synchronisation

In Forge, systems execute sequentially within each `tick`. However, systems without data dependencies (those not accessing the same component types) can run in parallel to improve throughput. To maintain component consistency across multiple Forge instances, Forge provides a synchronisation specification. Any Synchronisation System implementing this specification acts as a fence, ensuring all preceding dependent systems complete their execution before

¹<https://gitlab.com/cgvg/feanor/forge>

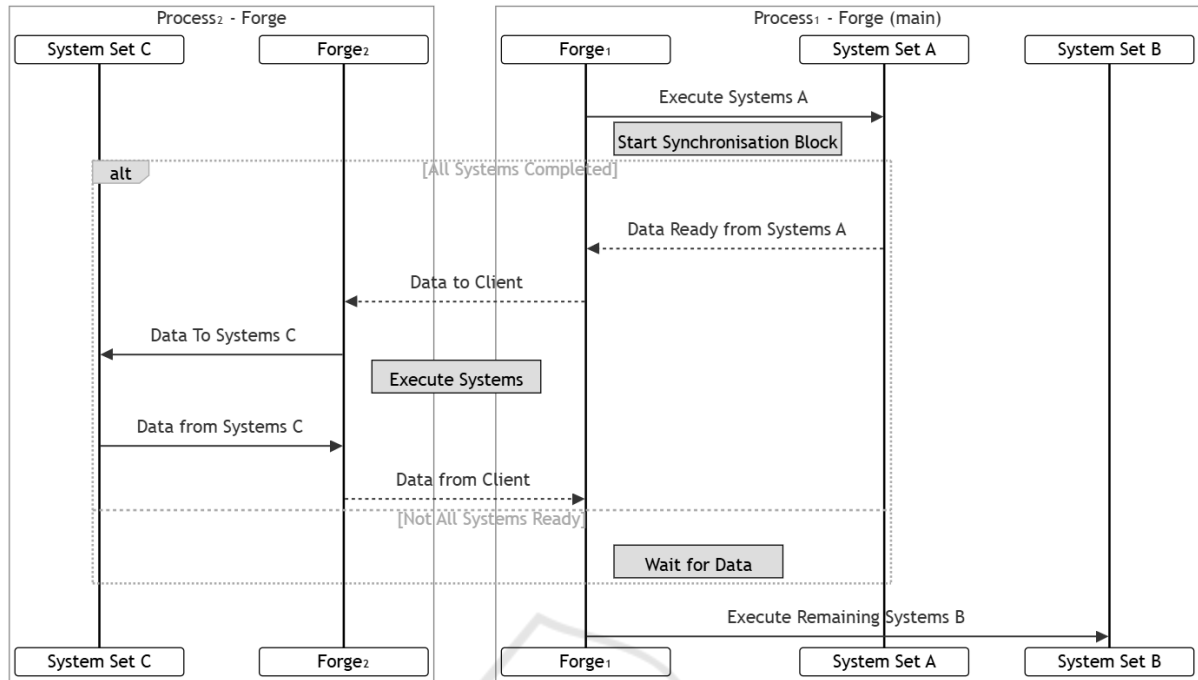


Figure 3: Multiple Forge instances communicating through the synchronisation specification. The synchronisation block (dashed rectangle) depicts the synchronisation process; the synchronisation systems are implicitly represented between System sets A and B, and encompassed in System set C.

proceeding.

Figure 3 shows Forge in a multi-process setup. The main instance contains: System set A (pre-synchronisation), the Synchronisation System, and System set B (post-synchronisation). System set C runs in a separate process with its own memory space. The synchronisation system, enhanced for decentralised operation across multiple machines, ensures System set A completes before exchanging data with System set C on its separate Forge instance. After synchronisation completes, System set B proceeds with execution.

3.1.2 WebSocket Synchronisation

The WebSocket System implements the synchronisation specification to enable interprocess or remote system integration. Forge can operate across different language runtimes like Python, using their native libraries. WebSockets provide broad language compatibility and browser support, enabling rapid development of web-based evaluation systems.

The WebSocket System works in two modes: *server* and *client*. In server mode, it captures snapshots of all entities at a given moment and broadcasts them to connected client systems. In client mode, it encapsulates systems requiring external data, operating in an isolated context that depends solely on entity state from its corresponding instance.

Whenever `tick` runs, the server sends entities to connected clients. Client systems can modify these entities as needed, and after all systems finish execution, the modified entities are sent back to the server. The server then updates its original entities, creating the effect that changes occurred locally. Entities with reflection components (Napoli et al., 2022) are read-only - any attempts to modify them are silently ignored.

3.2 Usage

A Forge system requires the implementation of two methods: `execute()` and `required_components()`. `execute()` runs automatically when Forge calls `tick`. While optional, `required_components()` should return a list of components needed by the system, allowing Forge to build a dependency graph for concurrent system execution.

A Proxy System is needed when integrating components or systems not natively supported by Forge that run in separate processes. This includes external tools, custom library components, and third-party systems outside of Forge’s built-in capabilities. For instance, integrating an external renderer with Forge for evaluation would require the following approach:

1. **Proxy System Development:** An Anvil system should be developed to function as an intermedi-

ary between Forge and the renderer.

2. **Interface Requirement:** The renderer must expose an interface that the Proxy System can utilise for configuration purposes.
3. **Configuration Handling:** The Proxy System should be designed to accept standard renderer configuration molecules, which it will use to properly initialise the renderer.
4. **Result Collection:** The Proxy System should also collect the output from the renderer and organise it into molecules that Anvil/Forge can process.
5. **Direct Molecule Option:** When working with renderers previously integrated with Anvil for debugging, the Proxy System can be simplified. These renderers can be modified to output result molecules in a Forge-compatible format directly, streamlining the integration process.

When a required Evaluation System is not available, users must implement a custom one following the three stages described in Section 3.1. An Evaluation System operates as a state machine, with each stage triggered by `tick` events and synchronised across all systems. This multi-stage approach is a consequence of the ECS pattern, ensuring that all systems perform the necessary work and remain synchronised at each stage of the evaluation.

Finally, after implementation is complete, Forge instances are initialised using a configuration format like that shown in Figure 1. For multiprocess or distributed setups, each Forge instance requires its own separate JSON configuration file.

4 USE CASES

Forge was evaluated using two CG test cases: comparing outputs between two different rendering algorithms on the same scene, and assessing spectral denoising with pre-computed coefficients in animation. In addition to being typical tasks in CG evaluation workflows, such as rendering buffers and comparing raw versus denoised images, the use cases also highlight Forge’s reusability and synchronisation features.

4.1 Light and Path Tracing Verification

Verification is a critical aspect of CG, particularly for ensuring the accuracy of rendering algorithms. When comparing different techniques, such as light tracing and path tracing, verification is essential to confirm the correctness of their implementations and the consistency of their outputs. Both results must be either

Table 1: Cornell Box Shiny comparing light and path traced scores. TM = ACES tone mapping.

Metric	Score	Score TM
FLIP	4.53e-02	8.12e-03
HDRVDP3	9.90	8.84
MSE	9.38e-05	1.77e-05
PSNR	69.94	47.53
SSIM	1.00	0.99

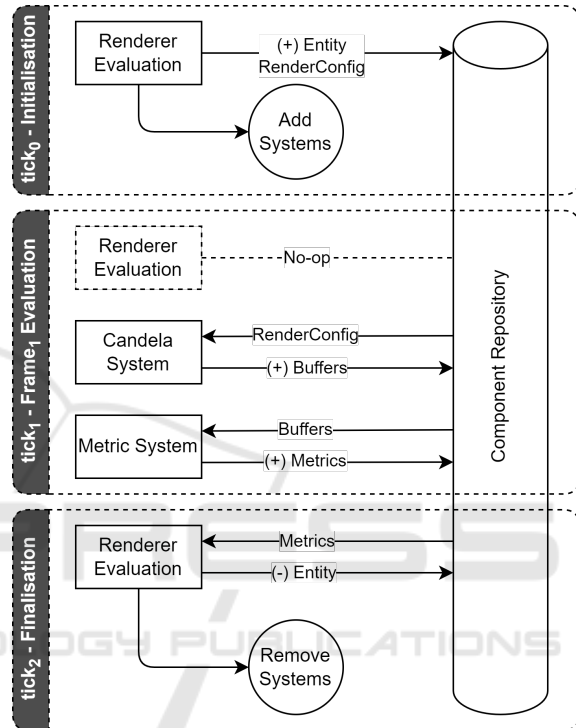


Figure 4: Light and path tracing flow diagram.

identical or closely aligned within acceptable error margins, determined using image similarity metrics. This comparison confirms the theoretical equivalence of the algorithms, validates their implementations, and helps identify discrepancies that may arise from numerical precision, sampling strategies, or other implementation details.

In this experiment, we use our in-house renderer, Candela, which operates via JSON configuration files through the command line. To integrate Candela with Forge, we developed a Proxy System using one of two approaches: exposing Candela as a shared library (risking new bugs), or writing a Proxy System that launches Candela as a separate process. While both methods require sufficient renderer configurability, we chose the latter approach and implemented it in Python, as simplicity was prioritised over performance.

When encountering an Entity with a

`RenderConfig` molecule, this System initiates a Candela renderer process using a generated JSON configuration that specifies the rendering technique (light tracer or path tracer) and scene setup. After the frame is processed, Candela saves the output buffers to the file system. The System then creates a `Buffer` molecule containing the radiance buffer data and metadata, including buffer type, pixel format, and image characteristics (reference or noisy).

The image comparison is managed by a separate Forge instance running a dedicated Python System. This System processes entities with `Buffer` molecules, comparing noisy buffers against reference images or other noisy buffers. After executing the comparison using five different metrics, it attaches the results to the entity as a new `Metrics` molecule.

The Renderer Evaluation system orchestrates the entire evaluation process. At startup, it loads its configuration and, during `tick0`, creates a `RenderConfig` molecule containing settings for evaluating both path tracing and light tracing algorithms on a modified Cornell Box scene. The WebSocket System runs in *server* mode, awaiting connections from both the Candela and Metric systems. The Renderer Evaluation System monitors for the appearance of the `Metrics` molecule, which indicates the completion of the evaluation. Once detected, it records the results and performs cleanup operations.

In the other processes, the WebSocket System operates in *client* mode and wraps both the Candela and Metric systems. During `tick1`, the Candela System processes the received `RenderConfig` molecule to generate both path tracing and light tracing buffers, which it attaches to the Entity. Subsequently, the Metric System analyses these buffers and adds comparison results via a `Metrics` molecule to the same Entity. The WebSocket System manages entity state synchronisation and transfer between all processes. The complete workflow, including all processes and data flow, is illustrated in Figure 4.

4.2 Spectral Denoising

Spectral denoising (Napoli et al., 2024) involves decomposing images typically into a frequency-related domain where noise can be more easily distinguished from the true signal. A multi-dimensional thresholding function is applied in this domain, followed by an inverse transformation to produce a denoised image. A search algorithm identifies effective thresholding coefficients, which we evaluate for their ability to reduce noise in animated caustics.

The image denoising is performed by a Python application called Spectral Image Denoising (SID),

which applies thresholding coefficients and configuration properties. A corresponding Proxy System was developed in Python to integrate SID with Forge, allowing direct import of the denoising logic. The system activates when an Entity contains both a `CoefficientConfig` molecule and a `Buffer` molecule (containing noisy and reference caustic buffers). Upon detection, denoising is performed and the denoised caustics buffer is added to the `Buffer` molecule.

The Spectral Evaluation System conducts evaluations using configuration data that includes coefficient values and renderer settings for a 16-frame animation. Figure 5 shows the data flow between systems, the component repository, and the System's operations at each `tick`. Configuration data for both Candela and SID systems is initially attached to a new Entity. Similar to the Renderer Evaluation System in Section 4.1, the WebSocket System operates in *server* mode, while Candela and SID systems run as separate processes in *client* mode through the WebSocket System.

During `tick1`, the Evaluation System waits for results. The Candela System (reused) renders the scene using the `RenderConfig` molecule, producing frame buffers with a noisy image (16 spp) and a reference image (65,536 spp). The SID System then denoises the 16 spp image using parameters from the `CoefficientConfig` molecule. The Metric System compares the noisy, reference, and denoised buffers, generating five metrics for both Noisy-Ref and Denoised-Ref comparisons in the `Metrics` molecule. This process repeats for all frames, with the SID System collecting metric results at each subsequent `tick`.

Once all frames have been evaluated, `tickn` initiates finalisation, cleans up by removing the Entity containing the attached molecules, and generates the result plot shown in Figure 6. The graph shows how Curvelet coefficients perform in denoising animations using MSE on the same scene it was trained with, but viewed from different angles. The Denoised-Ref line consistently remains below the Noisy-Ref line, indicating that the denoising process was successful for this animation. Figure 7 displays the noisy and denoised buffers generated for Frame 8.

5 DISCUSSION

Anvil organises data into atoms and molecules - uniquely labelled data structures with semantic meaning. This ensures systems work with well-defined data formats. For instance, a `Metrics` component au-

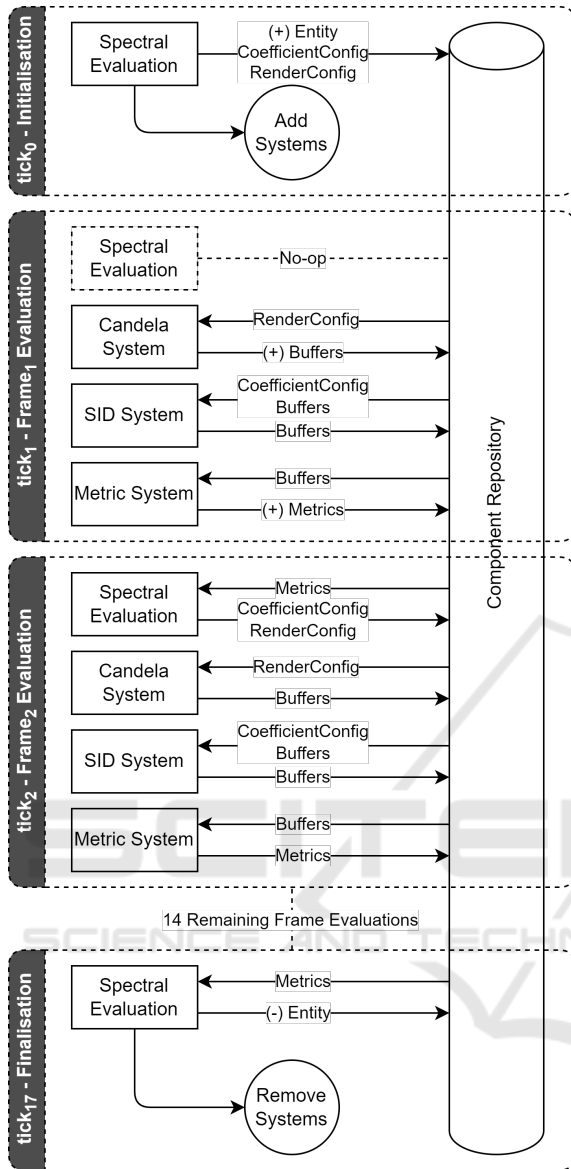


Figure 5: Spectral flow diagram.

tomatically provides five specific image quality metrics: MSE, SSIM, FLIP, PSNR, and HDRVDP3. Anvil’s registry of components and systems enables easy reuse of existing molecules when developing new systems.

The effectiveness of Forge depends heavily on its available components, systems, and evaluation systems. For example, the spectral denoising case study in Section 4.2 benefited from existing Candela and Metric systems, significantly reducing development time. Forge’s plug-in architecture, built on Anvil, makes it easy for developers to create and share new systems. The platform’s extensibility is evident in how the light and path tracing verification could be

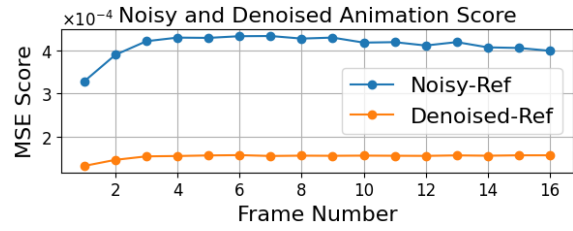


Figure 6: Spectral animation MSE score (lower is better).

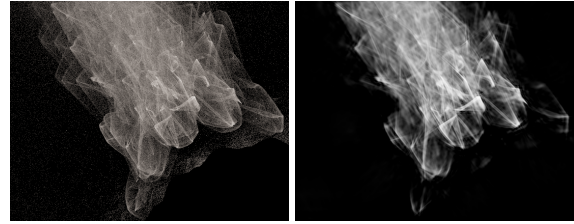


Figure 7: Frame 8. Left: noisy, right: denoised.

enhanced by adding other renderer systems that use `RenderConfig` molecules. For instance, implementing a Proxy System for Mitsuba would enable verification against this renderer by simply registering it during Forge’s initialisation.

Forge enables renderer evaluation without modifying the renderer’s source code, separating evaluation from application logic. Any renderer that can be configured via files or an API can interface with Forge. The platform’s synchronisation features, like the WebSocket System, allow systems to operate in isolated processes, including remote execution. This architecture supports scalability, making Forge suitable for large-scale evaluations such as those in render farms.

Forge supports real-time evaluation, allowing analysis of renderers as they respond to user input like camera movements or material adjustments. Other systems can process these frames to assess temporal coherence, motion blur quality, or sampling strategies. This immediate feedback helps artists and developers make informed decisions about scene composition, lighting, and algorithms during content creation. Real-time analysis can also reveal bugs that appear under specific scene conditions, making Forge valuable for both post-render analysis and interactive development in CG pipelines.

Forge’s architectural design not only makes evaluation more efficient but also minimises the potential for human error and configuration inconsistencies, which are frequent sources of inaccuracy in comparative CG studies. While Forge was originally designed for use in the context of CG, it can also be applied in other domains that have similar workflow characteristics.

6 CONCLUSIONS

This paper presents Forge, which builds upon Anvil to standardise evaluation processes in computer graphics pipelines. The system addresses major challenges in the field, including maintaining consistent configurations across renderers, reducing human error, and improving experimental reproducibility. Through these features, Forge enables researchers and developers to conduct reliable and accurate comparative studies of rendering techniques.

Forge's modular architecture offers flexibility and adaptability for diverse evaluation needs, allowing users to integrate new systems and tools without disrupting existing workflows. Its synchronisation interface supports decentralised operations across instances, as demonstrated by its WebSocket System for remote evaluations, coordinating systems in separate processes or machines. This design enhances scalability and versatility, making Forge ideal for various research contexts, from simple experiments to complex distributed rendering evaluations in environments like render farms.

The paper's case studies demonstrate Forge's effectiveness across various applications, such as validating rendering techniques by verifying algorithm consistency and correctness, and measuring performance and visual differences. The framework is a valuable research tool, offering a standardised environment to compare metrics like image quality, computational efficiency, and noise reduction. By automating evaluation processes, Forge reduces configuration inconsistencies and human error, leading to more reliable results.

A key limitation of Forge is its steep learning curve for setup and usage, especially for new users. Until wider adoption leads to more community-contributed tools, users may struggle to integrate it smoothly into their workflows and need to invest time developing custom systems. Furthermore, the WebSocket System for remote evaluations, while enabling distributed operations, may introduce network latency that could affect timing-sensitive measurements. This is particularly relevant for real-time rendering scenarios where precise performance analysis is crucial.

Future development of Forge should prioritise three key areas: improving accessibility through comprehensive documentation, tutorials, and example projects to ease adoption; optimising the WebSocket System for time-sensitive evaluations; and expanding evaluation capabilities through new metrics and machine learning-based analysis techniques. Active community participation will be crucial for contributing additional evaluation systems and metrics, ultimately

enhancing the framework's versatility across different research applications. Furthermore, a comparative analysis should be performed to assess CG workflows with and without the use of Forge, employing alternative tools for comparison.

REFERENCES

- Alatalo, T. (2011). An entity-component model for extensible virtual worlds. *IEEE Internet Computing*, 15(5):30–37.
- Andersson, P., Nilsson, J., Akenine-Möller, T., Oskarsson, M., Åström, K., and Fairchild, M. D. (2020). Flip: A difference evaluator for alternating images. *Proc. ACM Comput. Graph. Interact. Tech.*, 3(2):15–1.
- Andersson, P., Nilsson, J., Shirley, P., and Akenine-Möller, T. (2021). Visualizing Errors in Rendered High Dynamic Range Images. In Theisel, H. and Wimmer, M., editors, *Eurographics 2021 - Short Papers*. The Eurographics Association.
- Bilas, S. (2002). A data-driven game object system. In *Game Developers Conference Proceedings*, volume 2.
- Bonneel, N., Coeurjolly, D., Digne, J., and Mellado, N. (2020). Code replicability in computer graphics. *ACM Trans. Graph.*, 39(4).
- Dahl, T., Koskela, T., Hickey, S., and Valtjus-Anttila, J. (2013). A virtual world web client utilizing an entity-component model. In *2013 seventh international conference on next generation mobile apps, services and technologies*, pages 7–12. IEEE.
- Kallweit, S., Clarberg, P., Kolb, C., Davidovič, T., Yao, K.-H., Foley, T., He, Y., Wu, L., Chen, L., Akenine-Möller, T., Wyman, C., Crassin, C., and Benty, N. (2022). The Falcor rendering framework.
- Martin, A. (2007). Entity Systems are the future of MMOG development.
- Napoli, K., Bugeja, K., Spina, S., and Magro, M. (2024). Spectral transforms for caustic denoising: A comparative analysis for monte carlo rendering. In *Advances in Computer Graphics: Proceedings of the 41st Computer Graphics International Conference, CGI 2024, July 1–5, LNCS*. Springer. In press.
- Napoli, K., Bugeja, K., Spina, S., Magro, M., and De Barro, A. (2022). Anvil: A tool for visual debugging of rendering pipelines. In *VISIGRAPP (I: GRAPP)*, pages 196–203.
- Papagiannakis, G., Kamarianakis, M., Protopsaltis, A., Angelis, D., and Zikas, P. (2023). Project elements: A computational entity-component-system in a scene-graph pythonic framework, for a neural, geometric computer graphics curriculum. *arXiv preprint arXiv:2302.07691*.
- Unity Technologies (2024). Introduction to the data-oriented technology stack for advanced unity developers.
- Wang, V., Salim, F., and Moskovits, P. (2013). *The Definitive Guide to HTML5 WebSocket*. Apress.