Allegheny College
Computer And Information Science Department

Senior Thesis

Suzanne

by

Keven Michel Duverglas

ALLEGHENY COLLEGE

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Project Supervisor: **Janyl Jumadinova**
Co-Supervisor: **Douglas Luman**

Abstract

A well-researched student project. Like very.

Table of contents

List of Figures

List of Tables

1 Template description

This repository contains the starter materials for your thesis in Computer Science 600 and 610. The main directory of this repository contains the Markdown template for a project designed for use with GitHub Classroom. To learn more about the course in which these assignments were completed, please refer to the `README.md` file.

The template specifies various settings in the `_metadata.yml` file included in the repository. Change the appropriate values under the `Project-specific values` heading. Changing other values outside of that section may cause the project to fail to build. **Modify these values at your own risk.**

Author your thesis in the `index.md` document using appropriate Markdown hierarchy and syntax; GitHub Actions will automatically create a PDF from the relevant files.. Consult the `README` of the proposal repository to learn how to properly build and release this PDFs.

1.1 Citations and references

Including references throughout requires a specific pseudo-Markdown tag, demonstrated in the following blockquote. (Inspect the `thesis.md` file to see the format.)

> A citation, when included correctly, will appear as it does at the end of this sentence. [**?**]

1.2 Labeling figures

To label a figure (i.e. an image), referencing the image using correct Markdown will automatically caption the figure:

```
![Label](images/IMAGE_NAME.png)
```

1.3 Labeling tables

To provide a label for a table, write a short caption for the table and prefix the caption with `Table:` as in the example below:

```
Table: A two-row table demonstrating tables

|Row number | Description |
|:----------|:------------|
|1          |Row 1        |
|2          |Row 2        |
```

1.4 Other template information

Two things specific to this template to also keep in mind:

1. It is your responsibility to remove this description section before building the PDF version you plan to defend.
2. References *will only appear if cited correctly* in the text

1.5 Note on LaTeX commands

Documents may include specific LaTeX commands *in Markdown*. To render these, surround the commands with markup denoting LaTeX. For example:

```
Checkmark character:   $\checkmark$
Superscript character: $^{\dag}$
```

If using a special package not included in the template, add the desired LaTeX package or command/macro to the `header-includes` property in `config.yaml`.

Should this package not be included in the environment shipped with this template, you may also need to add the package to the GitHub Actions Workflow.

Direct any questions about issues to your first reader.

2 Introduction

2.1 Project overview

Blender is a free, open-source 3D creation suite used for modeling, animation, effects, simulation, and rendering [3, 12]. It powers professional production pipelines and is increasingly used in research, engineering, and higher education. Studies show that Blender's Python architecture, open-source licensing, and advanced rendering capabilities make it suitable even for scientific and engineering workflows, such as generating synthetic digital image correlation images for computational experiments [11]. Because Blender is both powerful and freely available, it is widely adopted by students, independent artists, and early-career creators building portfolios on limited budgets.

However, Blender's strength comes with a cost: a steep learning curve. Prior analyses of Blender's interface highlight how beginners struggle with its dense layout, multi-editor environment, and mode-dependent tool system [12]. New users must manage concepts such as object versus mesh data, operator conventions, modifier order, shading settings, and Python-driven tools long before they can produce high-quality work. This can slow progress, reduce confidence, and limit the number of completed portfolio pieces.

This project introduces **Suzanne**, a Blender add-on that lives in the right-hand **N-panel** and provides short, numbered, in-viewport steps for common tasks. Instead of searching externally for guidance while working, users receive instructions directly beside the 3D Viewport. The goal is to reduce context switching, help users execute reliably, and increase the number of polished artifacts that students and independent artists can publish.

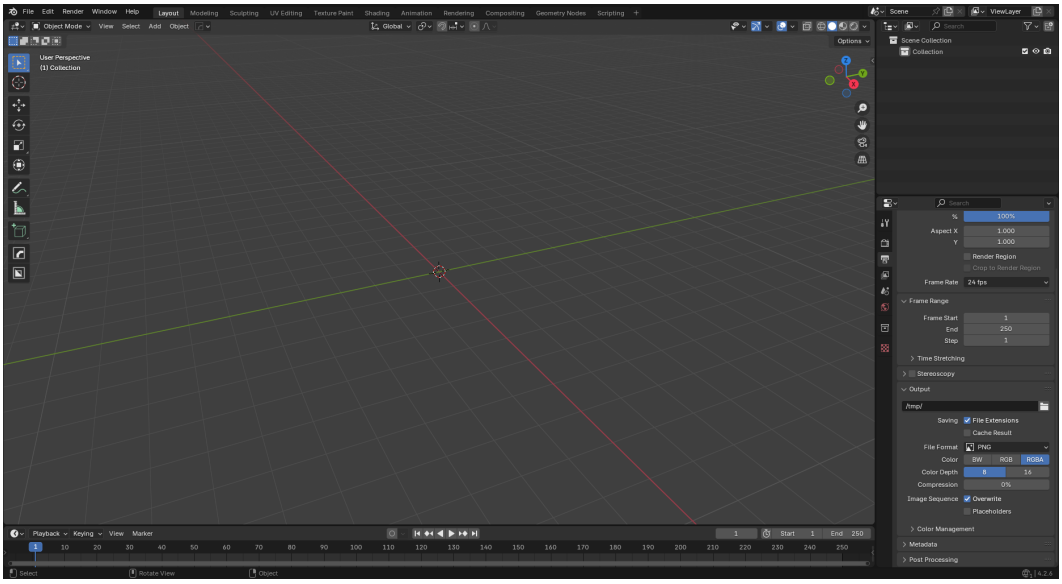

Figure 1: Blender interface with key areas visible: 3D Viewport (center), Outliner (right), Properties (bottom-right), and the N-panel (right sidebar).

2.2 Key terms and concepts (grounding the reader)

- **N-panel.** A vertical sidebar toggled with **N** in the 3D Viewport hosting add-ons and tools. Suzanne is located here to keep guidance directly inside the creative workspace [3].
- **Mode.** Blender tools are mode-specific (e.g., Object Mode vs. Edit Mode). Many operators behave differently or are unavailable depending on the mode, making explicit mode requirements essential in step-by-step guidance [3, 12].
- **Operator.** Any action invoked by menus, buttons, shortcuts, or the **F3** search. Naming operators (e.g., *Mesh > Normals > Recalculate Outside*) is key for reproducibility and structured documentation [3].
- **Modifier stack.** A series of non-destructive operations whose **order** changes results. For example, applying Bevel before Subdivision Surface produces a different silhouette and shading than the reverse [3].
- **Grounding.** Retrieval-Augmented Generation (RAG) combines large language models with authoritative sources. Integrating RAG ensures instructional steps match Blender's official terminology and correct behavior [6].

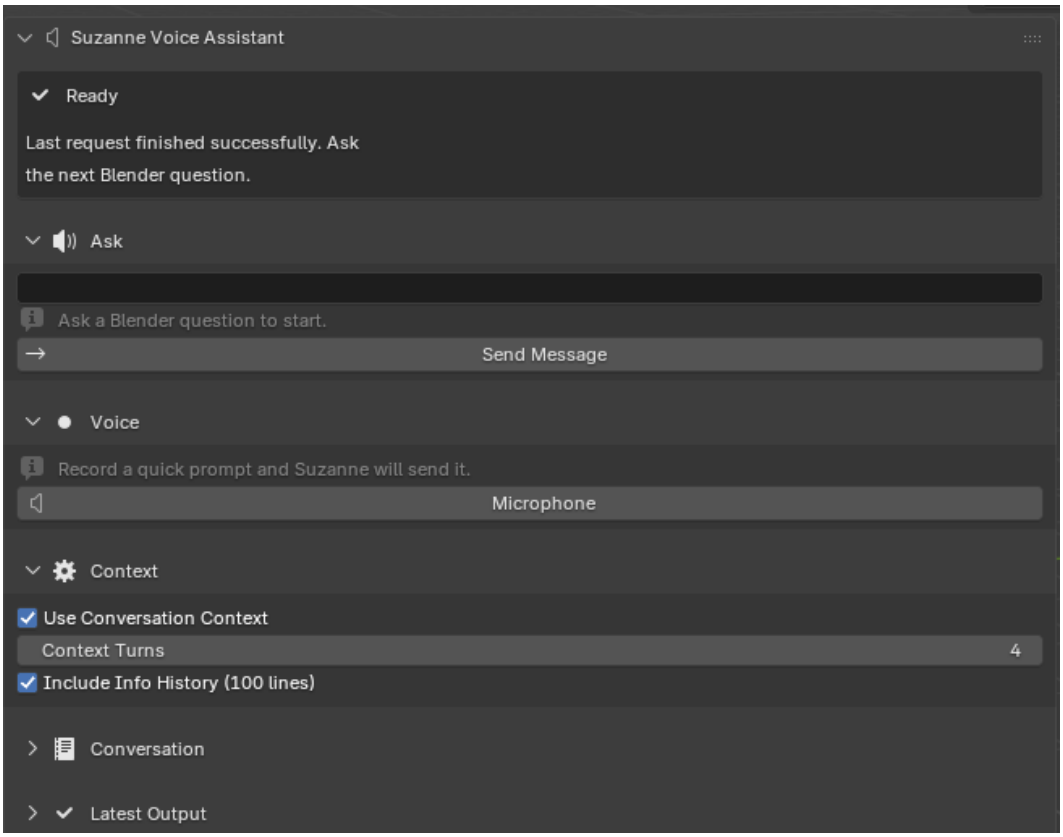

Figure 2: Suzanne add-on in Blender's N-panel, showing the current interface with status, text prompt, voice input, context controls, conversation tools, and latest output sections beside the 3D Viewport.

2.3 Motivation (why this matters for students and portfolios)

In my personal experience learning Blender over the past five years, most challenges involved "micro-execution"—not understanding the concept of what to do, but figuring out *which* operator to call, *what* mode to be in, and *what* order to perform actions. Research has shown that Blender's growing scope and tool density can overwhelm newcomers and slow their learning process [12]. Even small tasks such as beveling edges without shading artifacts or setting up lighting often require watching multiple tutorials or searching through forums.

Meanwhile, early-career 3D creators—students, hobbyists, and emerging artists—grow primarily through their **portfolios**. Recruiters and instructors evaluate:

- Clean topology visible in wireframe or clay renders
- High-quality lighting and presentation
- Turntables and breakdowns

- UV layouts and readable materials
- Clear process documentation

Students typically post these artifacts to ArtStation, Behance, GitHub Pages, or social media. However, producing consistent, high-quality work requires fluid execution, and execution is often slowed by searching for instructions outside the application.

Suzanne aims to address this gap: **deliver high-clarity, minimal-step instructions inside Blender**, grounded on authoritative documentation and consistent terminology.

This approach is supported by educational research showing that AI-based learning tools improve cognitive outcomes when instructional content is concise, context-specific, and directly actionable [9]. Suzanne follows these principles by placing guidance beside the active viewport and presenting it as small, verifiable steps that can be performed immediately.

2.4 Problem statement (what gap this work addresses)

Blender learners lose time, motivation, and project momentum because most guidance lives **outside** the application, spread across long YouTube videos, scattered forum posts, or generic documentation. These sources are rarely tailored to the user's current mode, object selection, or workflow context. As a result, beginners struggle with:

- Mode confusion
- Misordered modifiers
- Shading artifacts
- Inconsistent steps from mixed-version tutorials
- Difficulty reproducing actions from memory

The core gap is **micro-execution**: mode, operator, panel path, and modifier order. This project addresses that gap by building an in-viewport assistant that:

1. Returns verifiable steps inside the N-panel
2. Grounds instructions on the official Blender Manual
3. Uses retrieval techniques aligned with RAG best practices to maintain correctness [6]
4. Supports small troubleshooting branches to handle common issues

Because Blender is increasingly used in research and engineering [11], reliable execution and clear documentation also support academic reproducibility.

2.5 Project goals (what this thesis will deliver)

1. **Step-based teaching.** Provide concise, numbered instructions that reflect Blender's operator names and panel paths, always stating prerequisites (mode, selection, object type).

2. **Documentation grounding.** Integrate retrieval over the Blender Manual to maintain terminology accuracy and consistency with Blender's UI labels [3, 6].

3. **Learning iteration.** Support follow-ups like *next step*, *repeat this*, and corrective branches (e.g., faceting → Shade Smooth → Auto Smooth → Recalculate Normals).

4. **Optional code execution with guardrails.** Present small Python snippets that can be safely executed after user confirmation, leveraging Blender's scripting environment [11].

5. **Portfolio-driven defaults.** Provide guidance for modeling, lighting, and rendering workflows commonly used in portfolio pieces.

6. **Evaluation.** Assess Suzanne through software verification and task-based Blender experiments focused on reliability, instructional clarity, and workflow support.

2.6 Assumptions, limitations, and delimitations

2.6.1 Assumptions

This project rests on several practical assumptions about the environment in which Suzanne is used and the people who choose to install it:

- **Installed Blender and access to the N-panel.**
  It is assumed that users already have a working installation of Blender and understand how to access the right-hand N-panel in the 3D Viewport [3]. Suzanne is implemented as an N-panel add-on rather than a standalone application, so users must be able to enable add-ons, save .blend files, and navigate basic interface regions. The project does not attempt to teach operating-system installation, GPU drivers, or basic Blender navigation from scratch.

- **Informed consent for API usage.**
  When users submit text or audio to Suzanne's AI features, those requests are processed through a third-party API. The system assumes that users are aware of this and consent to it at the point of configuration—specifically, when they paste an API key into the add-on preferences and enable AI-driven features. The design presumes that users (or instructors, in a classroom setting) have reviewed relevant institutional or personal policies regarding the use of external AI services.

- **English-language interface as the baseline.**
  The initial release assumes that Blender's UI is set to English and that users can work with English operator names, menu labels, and instructions. Suzanne's retrieval pipeline targets the English Blender Manual [3], and the language model is prompted to mirror that vocabulary. Multilingual support and localization are identified as important directions for future work but are not treated as requirements for the first version.

- **Intermediate technical comfort.**
  Although Suzanne targets novices in terms of Blender skill, it assumes a basic level of technical comfort: users can install add-ons, manage API keys, and understand the idea of "saving before running code." The system does not, for example, guide users through package manager installation, shell configuration, or advanced debugging.

2.6.2 Limitations

Despite careful design choices, Suzanne has several important limitations that shape how its results should be interpreted:

- **Residual inaccuracy in LLM-generated steps.**
  Grounding on the Blender Manual and using retrieval-augmented generation improves factuality and terminology alignment, but it does not eliminate mistakes [6]. The model may still suggest slightly outdated menu paths, omit necessary steps, or assume a different initial selection than the user has. Users are therefore advised to treat Suzanne's instructions as **high-quality suggestions**, not guaranteed truths, and to cross-check against the Manual or their own experience when something appears wrong.

- **Restricted code execution scope.**
  For safety reasons, code execution is intentionally limited to a safe subset of Blender's Python API, focusing on object creation, transforms, lights, cameras, and shader nodes. While this reduces the risk of destructive or security-sensitive operations, it also means that Suzanne cannot assist with every possible automation scenario. Advanced scripting tasks—such as complex rigging tools, file I/O, or integration with external render farms—are explicitly out of scope for the current version.

- **Version- and hardware-dependent behavior.**
  Blender evolves rapidly, and operator locations, defaults, or UI layouts can change between releases [3]. Suzanne targets a specific range of Blender versions during development; outside that range, some instructions may no longer match the interface exactly. Similarly, behavior can vary by platform (Windows, macOS, Linux) and hardware configuration (GPU vs CPU, different input devices). The project cannot guarantee identical behavior across all environments, and some user-reported issues may stem from these external differences.

- **Limited awareness of scene context.**
  Suzanne has only partial visibility into the user's scene state. While future versions might integrate deeper inspection tools, the current implementation often infers context from user prompts and a small set of requested scene details. This can lead to misalignment when the scene contains unusual setups (e.g., non-standard hierarchies, heavily customized keymaps, or add-on-specific data structures).

- **Evaluation scope.**
  The evaluation described later in this thesis is constrained to a limited number of tasks, users, and time. Results about time-on-task, perceived usefulness, or error reduction should be interpreted as **initial evidence**, not definitive proof of general effectiveness across all Blender workflows or learner populations.

2.6.3 Delimitations

In addition to inherent limitations, the project also includes deliberate **design choices** that narrow its scope. These delimitations are not weaknesses, but boundaries set so the work remains feasible and coherent:

- **Focus on learnability, modeling, shading, and presentation.**
  Suzanne is explicitly aimed at core workflows that support beginner and intermediate portfolio pieces: modeling (especially hard-surface and simple organic forms), shading, lighting, and basic presentation (turntables, still renders). Advanced areas such as character rigging, complex simulation (fluids, cloth, smoke), geometry nodes systems, and compositing are intentionally left out of the initial feature set. This allows the project to concentrate on the "bread-and-butter" tasks that most early-career artists must master to build a credible portfolio.

- **No live microphone-based transcription in v0.x.**
  Although speech-to-text could further reduce friction for some users, the current version does not implement live microphone capture or continuous audio transcription. All prompts are entered as text, and any audio-based features are limited to explicitly uploaded or recorded snippets. This avoids additional privacy and consent complexity and keeps the interaction model simpler for evaluation.

- **No end-user analytics or behavioral tracking.**
  Suzanne does not collect telemetry or analytics about how users interact with the add-on. There are no built-in dashboards tracking which prompts are most common, which steps cause difficulty, or how often scripts are executed. While such data could be valuable for iterative design and research, it would also introduce significant privacy and governance concerns. Instead, this thesis relies on local software verification and authored task demonstrations rather than behavioral tracking or human-subject data collection.

- **No human-subject dataset in this thesis.** The thesis does not report survey responses, timing logs, or other human-subject study data. Claims are therefore limited to implemented system behavior, automated reliability checks, and the documented task-based experiments presented later. Formal user studies remain future work.

- **Single primary documentation source.**
  For grounding, Suzanne relies primarily on the Blender Manual and does not, in this version, integrate other textual sources such as third-party books, course notes, or forum archives [3]. This delimitation keeps the retrieval pipeline manageable and the terminology consistent, but it also means that insights from community best practices (for example, from popular tutorial series or studio pipelines) are only reflected indirectly through prompt design, not through direct retrieval.

By making these assumptions, limitations, and delimitations explicit, the thesis clarifies the conditions under which Suzanne is expected to work well and the boundaries beyond which its claims should be treated cautiously. Later chapters return to these points when interpreting evaluation results and outlining directions for future work.

2.7 Ethical considerations (narrative)

2.7.1 Privacy and consent

Submitted prompts and audio files are processed through a third-party API. Research shows that large language models (LLMs) face serious risks related to privacy, data leakage, and unintended memorization of sensitive content, even when providers claim to filter or anonymize

data [5]. In the context of student work and personal projects, leaked prompts could reveal identifying details, coursework, or unpublished research, including descriptions of in-progress thesis ideas, screenshots of original models, or references to real people. Since Blender is often used to create highly personal or autobiographical work, these risks are not abstract; a prompt describing a "self-portrait scene in my dorm room at Allegheny" can easily become identifying if mishandled.

To reduce these risks, Suzanne adopts a **local-first** design wherever possible. API keys are stored only on the user's machine, inside Blender's add-on preferences, and are never transmitted to any external server controlled by the add-on developer. Suzanne does not implement its own logging of prompts or responses; once a session ends, there is no add-on-level history of user queries. The only data sent to the third-party provider is the text and/or audio that the user explicitly submits as part of a request. The interface includes clear warnings about avoiding sensitive material (e.g., real names, proprietary data, or confidential assets), and the documentation encourages users to consult institutional policies on AI tool usage before integrating Suzanne into graded coursework or research workflows.

In any future formal study of Suzanne, volunteers should be informed about what data leaves their machine, which provider processes it, and how long it may be retained according to that provider's terms [5]. They should also be able to disable API-based features entirely and still use the add-on as a structured reminder of manual workflows grounded in the Blender Manual [3]. In classroom settings, instructors are encouraged to provide alternative, non-AI pathways to complete assignments so that students who are uncomfortable with third-party processing are not penalized.

2.7.2 Reliability and user control

While grounding on the Blender Manual and retrieval-augmented generation reduces some errors, LLMs remain fallible and can still propose incorrect or incomplete sequences of steps [6, 5]. For example, a generated workflow might reference an operator that moved in a newer Blender version, assume the wrong selection mode, or omit a crucial modifier step. Survey research on AI systems emphasizes that tools used in high-stakes or educational contexts must be designed around human oversight, transparency, and reversible actions to maintain user trust [5]. An assistant that silently edits the scene or hides its reasoning would be misaligned with these principles.

Suzanne therefore treats the model as an *advisor*, not an authority. It always displays instructional steps before any changes are applied and explicitly labels them as suggestions that should be verified by the user. When code snippets are generated, they appear in a dedicated panel where users can inspect the Python before deciding whether to run it. Code execution is strictly opt-in: Suzanne never executes code automatically in response to a prompt. Users must press a separate confirmation button, reinforcing the mental separation between "seeing advice" and "changing the scene."

Blender's own undo stack is highlighted as the primary recovery mechanism if something behaves unexpectedly. The add-on's documentation recommends that users save incremental versions of their .blend file (for example, `scene_v03.blend`, `scene_v04.blend`) before experimenting with code-driven changes. The interface also encourages users to cross-check instructions against the Blender Manual when results look suspicious or differ from expectations [3]. In effect, the design continually nudges users to maintain **interpretive control**: Suzanne can suggest the next move, but the user decides whether it is appropriate for their current scene and learning goals.

2.7.3 Bias and inclusivity

Educational research on AI-based learning tools highlights the importance of clear, accessible feedback that supports diverse learners, rather than favoring only those with high prior knowledge or specific linguistic backgrounds [9]. Blender itself already presents a high barrier to entry: the interface is dense, the terminology is specialized, and much community documentation assumes familiarity with English technical jargon and gaming culture. Without care, an AI assistant could easily amplify these barriers—by using slang, skipping explicit prerequisites, or tailoring examples to a narrow subset of users.

Suzanne's instruction style is therefore intentionally **plain and procedural**. Each step names the relevant mode, operator, and UI path instead of assuming tacit knowledge or relying on vague phrases like "clean up the mesh." For instance, instead of saying "fix the shading," Suzanne might say "Switch to *Object Mode*, select the object, then choose *Object > Shade Smooth* and enable *Auto Smooth* in the *Object Data Properties > Normals* panel." This benefits students, self-taught artists, and non-native English speakers who may be less familiar with community slang or informal tutorial styles. It also supports learners who prefer to map instructions carefully to the interface rather than following along with a video at the instructor's pace.

At the same time, the project acknowledges that underlying language models can encode societal biases in examples, metaphors, or suggested asset names [5]. To mitigate this, Suzanne intentionally scopes its responses toward **technical actions** (operators, modes, and parameters) and away from content that labels or describes people. The documentation discourages prompts that rely on demographic stereotypes (e.g., asking for "typical" appearances of certain groups) or that seek value judgments about whose work "looks better." When portfolio examples are mentioned, they are framed in terms of topology cleanliness, lighting clarity, and presentation conventions, not in terms of personal attributes. The long-term goal is to support skill-building and confidence, particularly for learners who may not see themselves represented in mainstream 3D education spaces.

2.7.4 Cost transparency and outages

LLM providers can change pricing, rate limits, and model availability with little notice. This volatility is especially relevant for students and independent artists working with limited budgets, who may not be able to absorb unexpected charges or interruptions. A tool that quietly consumes API credits in the background or fails without explanation would undermine both trust and accessibility.

Suzanne addresses this by making **API usage explicit and interruptible**. The add-on requires users to paste their own API key rather than bundling any shared or hidden key, which makes the cost relationship clear: any charges are between the user and the provider. When an API request fails—because of quota exhaustion, authentication errors, or network issues—Suzanne surfaces explicit error messages rather than silently falling back to an empty response. Users are pointed toward their provider dashboard to check usage and are encouraged to set their own spending limits.

Whenever API-based features are unavailable, Suzanne falls back on workflows grounded in Blender's official practices—for instance, pointing users directly to relevant sections of the Blender Manual or suggesting manual operator paths [3]. In classroom scenarios, instructors can choose to disable the API-dependent features entirely and still use the add-on as a structured, manual recipe panel. This ensures that the tool remains a useful learning aid even when AI services are unavailable or unaffordable, and it reinforces that the

core knowledge lives in Blender's open documentation rather than in any single commercial model.

2.7.5 Security and scope

Because unrestricted Python execution in Blender can cause serious harm—from deleting or corrupting scenes to interacting with the file system or network—Suzanne deliberately limits what kind of code it can propose. Security surveys of LLMs warn that generated code can be manipulated or misused to escalate privileges, exfiltrate data, or perform other unintended actions, especially when execution is automated or opaque [5]. Blender add-ons that expose "run arbitrary code" endpoints without constraints effectively grant the model the same power as an expert user with full access to the scene and environment.

In response, Suzanne restricts script generation to a narrow slice of Blender's API: object creation, transforms, lights, cameras, and shader nodes. Operations such as deleting objects, applying all modifiers, or resetting entire scenes are either excluded or heavily discouraged. The add-on explicitly avoids file operations (opening, saving, or deleting files), external network calls, or direct system-level access, thereby reducing the potential attack surface. Any script that appears in the UI is kept short enough for a motivated user to skim, and it is formatted clearly so that parameter values and operator names are visible.

Suzanne also leverages Blender's existing safety mechanisms. Scripts run within Blender's Python environment, which already exposes undo and redo for most scene operations. Users are encouraged to save .blend files frequently and to experiment on copies rather than production scenes. The documentation includes a "safety checklist" that recommends: (1) saving before executing any script, (2) inspecting code for obviously destructive calls, and (3) using undo immediately if an unexpected change occurs. These guardrails do not eliminate all risk, but they align Suzanne with best-practice recommendations for LLM-driven code execution: minimize permissions, maximize visibility, and keep humans firmly in control [5]. In combination with the privacy and consent measures above, this scoped design aims to make Suzanne a responsible, student-friendly integration of AI into Blender rather than a source of hidden technical or ethical debt.

2.8 What does *not* belong in the Introduction

Practical modeling recipes, shading fixes, or operator sequences should not be included here. These belong in **Methods** or an **Appendix**, where Suzanne's generated steps can be presented clearly. The Introduction is focused on background, motivation, problem definition, goals, scope, and ethics.

2.9 Chapter roadmap

The remainder of this thesis proceeds as follows:

- **Related Work** reviews Blender learning challenges, prior add-ons, AI tutoring systems, RAG-based grounding, and safe model usage.
- **Methods** details the N-panel UI, retrieval pipeline, grounding strategy, and safe code execution model.
- **Evaluation** presents the study design, tasks, metrics, and analysis.

- **Discussion** interprets results and limitations.
- **Future Work** explores multilingual support, richer scene graph awareness, and expanded tool-calling capabilities.

3 Related Work

3.1 Overview and Scope

This chapter reviews prior work in six areas that together frame Suzanne's design and research questions: (i) how people currently learn Blender through manuals, books, and community pedagogy; (ii) in-app guidance and "copilot"-style assistance in creative and coding tools; (iii) AI teaching assistants and step-oriented learning tools; (iv) retrieval-augmented generation (RAG) grounded in trusted documentation; (v) code generation and safety in end-user environments; and (vi) portfolio-based assessment and learning outcomes.

Across these strands, a common pattern appears: powerful tools—whether 3D suites, IDEs, or AI models—offer enormous expressive range but place a high burden on **micro-execution**: knowing which operation to call, in which context, and in what order. Existing resources for Blender tend to live *outside* the viewport (books, PDFs, long videos, forum threads) or, in the case of recent AI assistants, focus on generating code rather than teaching reproducible steps. Suzanne positions itself at the intersection of these literatures as an **in-viewport, step-by-step tutor** that (a) grounds its responses in the official Blender Manual [3], (b) returns numbered, operator-named steps inside the N-panel, and (c) optionally surfaces small Python snippets behind explicit guardrails.

3.2 Learning Blender: Manuals, Tutorials, and Community Help

Blender's official Manual is the primary authoritative source for terminology, operator behavior, and UI labels [3]. It defines core concepts such as modes, the modifier stack, and data-block organization, and it documents the precise menu paths and shortcuts associated with each operator. Because of its breadth and authoritative status, this work treats the Manual as the ground truth for operator names, panel labels, and expected behavior: Suzanne's retrieval pipeline targets these pages, and its UI mirrors the Manual's language where possible.

Research on Blender's interface and evolution reinforces how challenging the software can be for new users. Soni et al. review multiple versions of Blender and highlight the complexity added by its multi-editor layout, dense menus, and mode-dependent tools [12]. They note that changes across versions can also create friction when users follow older tutorials, contributing to confusion about where options are located or why certain operators behave differently. These findings align with anecdotal reports from the Blender community that beginners frequently struggle with mode switching, modifier order, and shading artifacts.

Beyond official documentation, Blender learners rely heavily on **project-based resources** such as Arijan Belec's *Blender 3D Incredible Models*, which walks readers through hard-surface modeling, procedural texturing, and rendering workflows [2]. Books like this provide rich, end-to-end examples but still require learners to juggle a separate text or PDF alongside the 3D viewport, translating prose descriptions and screenshots into local actions.

Video tutorials represent another major learning ecosystem. The Blender Guru "donut" series, for instance, has become a de facto introduction to modeling, shading, and rendering for many beginners [10]. Long-form videos offer detailed demonstrations and a sense of narrative progression, but they also introduce substantial **context-switching costs**: learners pause, rewind, or scrub to find the relevant moment; they struggle to map instructions onto different Blender versions; and they often have to adapt the tutorial to their own scene.

Taken together, prior work and practitioner resources show that Blender learners have access to deep, high-quality material, but it is fragmented across manuals, books, and videos.

Very little of this guidance appears as **short, verifiable steps inside the viewport** itself, which is the gap Suzanne aims to explore.

3.3 In-App Guidance and "Copilots" in Creative and Coding Tools

In the broader HCI and software-engineering literature, there is strong interest in **in-context assistance** that appears directly within the tools people use. Chilana et al. study MarmalAid, a web-based 3D modeling environment that embeds real-time expert chat within the 3D scene [4]. Their observational work with novice–expert pairs shows that in-context chat leads novices to ask more task-focused questions and reduces the friction of switching to external help channels. This supports the idea that assistance is more effective when it is tightly coupled to the workspace rather than offloaded to separate windows or devices.

Although most large-scale empirical work on copilot-style assistance has focused on **coding** rather than 3D art, the underlying themes are similar. Inline suggestions in IDEs reduce the need to consult external documentation and can speed up repetitive tasks. Luo et al.'s systematic review of AI-based learning tools in higher education notes that tools integrated into students' existing workflows—such as writing or coding environments—tend to show stronger effects on engagement and perceived usefulness than standalone web portals [9].

For Blender specifically, community tools such as custom pie menus and quick-access panels provide a form of "micro-guidance" by surfacing commonly used operators, but they rarely explain *why* a given operator is appropriate or what prerequisites (mode, selection, object type) must hold. Suzanne builds on the general insight from MarmalAid and AI-enhanced IDEs: assistance should appear **where the work happens**. It extends this idea by focusing not on chat alone, but on **small, numbered recipes** that explicitly list prerequisites and operator names.

3.4 Blender Add-ons and Prior Attempts at Guidance

Recent years have seen the emergence of Blender add-ons that integrate large language models directly into the application. BlenderGPT is an early example: a plug-in that exposes a GPT-4/3.5-backed panel where users can type natural-language commands like "create a cube at the origin," which the system then translates into Python scripts executed inside Blender [7]. BlenderGPT demonstrates that LLMs can control Blender's Python API and support natural-language scene manipulation, but it primarily targets **code generation and automation**, not pedagogy. Generated scripts run immediately, and while users can inspect them via the system console, the main interaction is "describe the goal, then let the model act."

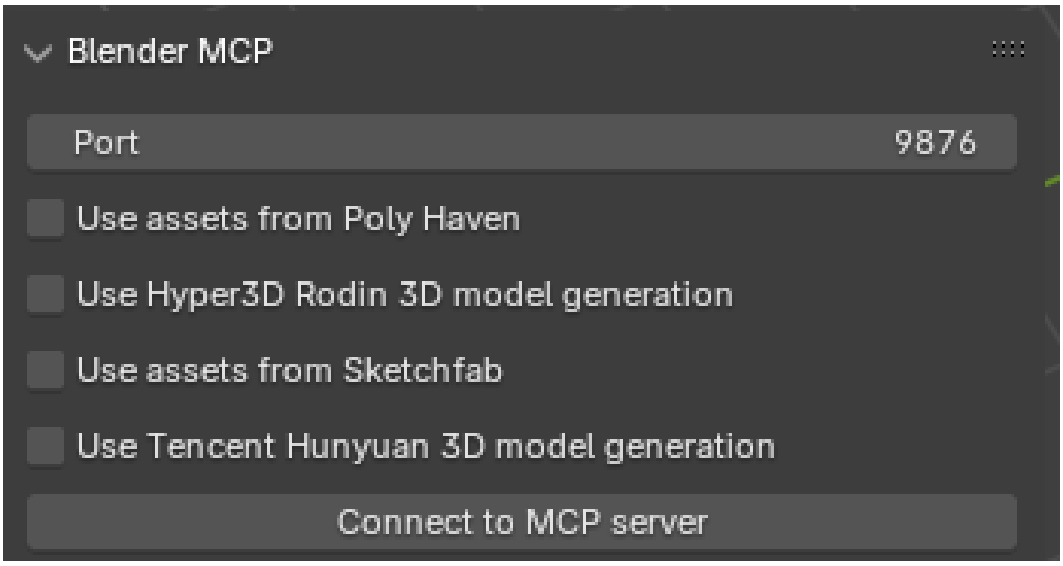

Figure 3: Panels for existing AI-powered Blender add-ons, such as BlenderGPT and BlenderMCP, shown inside Blender's sidebar. These tools accept natural-language prompts and execute generated Python directly, emphasizing automation rather than step-by-step instruction.

BlenderMCP extends this pattern by connecting Blender to Claude through the Model Context Protocol [1]. It exposes tools for scene inspection, object manipulation, material adjustments, asset retrieval from Poly Haven and Sketchfab, and arbitrary Python code execution. BlenderMCP explicitly positions itself as an AI-assisted 3D modeling companion, allowing high-level prompts like "create a low poly dungeon scene" or "make this car red and metallic." Similar to BlenderGPT, however, its focus is on **delegating actions** to the model rather than teaching users how to perform those actions themselves.

These systems show that:

1. LLMs can successfully interact with Blender's Python API; and
2. there is community demand for conversational, in-Blender assistants.

At the same time, they surface design and safety challenges discussed later in this chapter. For the purposes of this thesis, they serve as **baseline examples** of AI-powered Blender add-ons that emphasize automation. Suzanne diverges by centering **instruction**: instead of simply executing a scene change, it returns a small set of numbered steps (e.g., required mode, menu path, parameter values) and only optionally offers Python snippets behind explicit confirmations. In that sense, Suzanne is closer to MarmalAid's in-context help [4] than to an autonomous copilot.

3.5 AI for Teaching and Tutoring

Beyond 3D modeling, there is a growing body of work on AI-based learning tools in higher education. Luo et al. synthesize studies of AI-enabled systems—ranging from chatbots and recommendation engines to adaptive tutors—and report generally positive effects on engagement, perceived usefulness, and, in some cases, learning outcomes [9]. They highlight that tools are most effective when they provide **specific, actionable feedback** aligned

with course goals, and when they are integrated into students' regular workflows rather than offered as optional extras.

The review also notes risks that are directly relevant to Suzanne: over-reliance on AI advice, inconsistent accuracy, and limited transparency about how suggestions are produced [9]. These concerns echo broader security and privacy issues raised by Das et al., who catalog a range of vulnerabilities in large language models, including data leakage, jailbreaking, and misuse across domains such as education and healthcare [5].

From the perspective of procedural skill learning, prior work on intelligent tutoring systems and worked examples (summarized in reviews like Luo et al.'s) suggests that **step-by-step guidance with gradual fading** can help novices internalize complex procedures rather than simply copying answers. While Suzanne does not implement a full student model or adaptive fading, its design is inspired by this literature: it presents short, numbered steps with explicit prerequisites, and encourages learners to repeat and modify these sequences in their own scenes.

3.6 Retrieval over Trusted Docs: Grounding on the Blender Manual

Large language models are known to hallucinate or deviate from domain terminology when prompted without external grounding. Retrieval-augmented generation (RAG) is a widely discussed pattern for mitigating these issues by retrieving relevant documents before or during generation [6]. Gao et al. survey RAG architectures and show that conditioning models on retrieved passages can improve factual accuracy and alignment with specialized vocabularies, especially in technical domains [6].

Suzanne adopts this pattern by retrieving passages from the Blender Manual [3] for each user query and using them as context when generating instructions. In practice, this means that operator names, panel paths, and option labels are drawn from the same text that users would encounter if they opened the Manual directly. The system also surfaces links or section references back to the Manual, encouraging users to verify instructions and explore deeper explanations.

Grounding on the Manual further supports *version transparency*: when Blender's UI or operator behavior changes, the documentation is typically updated first. By aligning Suzanne's outputs with the Manual rather than with arbitrary web pages or forum posts, the system aims to remain closer to Blender's official semantics, while still benefiting from the flexibility of language models.

3.7 Code Generation in End-User Tools: Safety and UX

As BlenderGPT and BlenderMCP illustrate, LLM-driven code generation in end-user tools offers powerful benefits but also introduces significant risks [7, 1]. On the positive side, generated Python can automate repetitive scene setup, create lighting and camera rigs, or scaffold shader node networks that would be tedious to construct manually. In scientific workflows, scripted pipelines built on Blender's Python architecture have been used to generate synthetic imagery for digital image correlation experiments, demonstrating that Blender can support reproducible, research-grade pipelines [11].

However, arbitrary code execution inside a rich 3D environment is inherently risky. A single destructive operator (e.g., applying modifiers, deleting objects, or clearing transforms)

can irreversibly alter a scene if the user does not immediately notice. BlenderMCP's documentation explicitly warns users about the dangers of its `execute_blender_code` tool and recommends saving work before issuing commands [1]. Das et al.'s survey of LLM security concerns generalizes this problem, noting that models can be prompted—intentionally or accidentally—to produce harmful code, access sensitive data, or trigger unintended side effects [5].

From a UX standpoint, these findings suggest several design norms for mixed-initiative code generation:

* **Transparency** – users should see the generated script before it runs.
* **Explicit confirmation** – execution should be opt-in, not automatic.
* **Scoped capabilities** – tools should avoid file I/O, networking, and other high-risk operations in default modes.
* **Reversibility** – environments should support undo or rollback.

Suzanne adopts these norms by showing the generated Python snippet in a dedicated area, requiring explicit confirmation before execution, restricting code generation to a safe subset of Blender's API (object creation, transforms, lights, cameras, shader nodes), and relying on Blender's undo stack as a primary recovery mechanism. In contrast to BlenderGPT and BlenderMCP, which often treat code as the main output, Suzanne treats code as **optional scaffolding** that complements its primary product: human-readable, reproducible steps.

3.8 Portfolios, Artifacts, and Evaluation in 3D Learning

Although this thesis focuses on Blender rather than formal writing instruction, ideas from portfolio-based assessment transfer directly to 3D art education. Lam argues that "assessment as learning" reframes portfolios from static evidence of achievement to part of an ongoing cycle where students generate work, receive feedback, and reflect on their progress [8]. In portfolio-based classrooms, artifacts are valued not only for their final quality but also for the metacognitive skills students develop by revisiting and revising their work.

In 3D modeling and digital art, instructors and reviewers similarly evaluate students through **bodies of work** rather than isolated assignments. Hard-surface projects like those in Belec's *Blender 3D Incredible Models* emphasize clean topology, consistent shading, and thoughtful presentation—qualities that are visible across turntables, wireframe renders, and breakdown images [2]. Scientific uses of Blender, such as generating synthetic image-correlation datasets, also rely on reproducible pipelines and documented workflows [11].

Suzanne is designed with these portfolio dynamics in mind. By helping learners execute common modeling, shading, and lighting tasks more reliably, it aims to increase the number of **finished, portfolio-ready artifacts** they can produce within a given time frame. Step-level guidance—explicitly naming modes, operators, and modifier order—supports repeatability: once a learner has successfully completed a workflow with Suzanne, they can apply the same pattern to new projects or adapt it to more complex scenes.

3.9 Ethical and Governance Considerations

The use of LLMs inside educational and creative tools raises ethical questions around privacy, security, and equity. Das et al. document a wide range of attack vectors and privacy risks for large language models, including data poisoning, prompt injection, and leakage of personally

identifiable information [5]. When such models are integrated into classroom or portfolio workflows, these risks intersect with institutional responsibilities to protect student data.

Luo et al. note that many AI-based learning tools do not clearly communicate what data they collect or how it is used, which can erode trust among students and instructors [9]. They argue for transparent governance and explicit consent when deploying AI in higher-education settings. For open-source, locally installed tools like Suzanne, this translates into design choices such as keeping API keys on the user's machine, avoiding server-side logging of prompts, and clearly warning users against sending sensitive content.

Equity and access are also central. Blender itself is free and open source [3], which has made it a critical tool for students and independent artists who cannot afford commercial 3D suites. By building Suzanne as a free add-on that runs inside Blender and by grounding its responses in publicly available documentation, the project seeks to support self-taught learners and students at resource-limited institutions. At the same time, reliance on third-party LLM APIs introduces cost and connectivity constraints; this thesis acknowledges those limitations and points forward to potential offline or on-device models in future work.

3.10 Synthesis: Remaining Gaps

Existing literature and tools establish several important points. First, Blender's power and open-source ecosystem make it attractive for education, portfolios, and even scientific workflows, but its interface and mode system pose significant challenges for novices [3, 12, 2, 11]. Second, in-context help and embedded communication channels—such as MarmalAid's real-time expert chat—can make it easier for learners to seek and apply guidance without leaving their workspace [4]. Third, AI-based learning tools and LLM-powered assistants show promise for providing personalized feedback and automation, but they raise concerns about accuracy, over-reliance, privacy, and security [9, 5]. Finally, portfolio-based assessment emphasizes cycles of production and reflection, placing value on tools that help students consistently produce and refine artifacts [8].

Within this landscape, there is still a clear gap: **no existing system provides an in-viewport, step-by-step tutor for Blender that is explicitly grounded in the official Manual, designed around portfolio-oriented workflows, and equipped with safe, optional code execution.** Current AI add-ons such as BlenderGPT and BlenderMCP prioritize automation and code generation [7, 1]; community tutorials and books provide rich instruction but remain external to the viewport [2, 10].

Suzanne is designed to address this gap by:

1. Delivering short, numbered, operator-named steps inside the N-panel;

2. Grounding those steps on retrieved passages from the Blender Manual [3, 6];

3. Providing optional Python snippets under explicit guardrails informed by LLM security literature [5]; and

4. Targeting modeling, shading, and presentation workflows that contribute directly to students' and independent artists' portfolios.

The next chapter details how these ideas are realized in the system's architecture, including the N-panel UI, retrieval and grounding pipeline, safe code-execution model, and the

evaluation framework used to assess Suzanne through software verification and task-based experiments.

4 Methods

4.1 Chapter purpose and methodological stance

This chapter explains how Suzanne was designed, implemented, and prepared for evaluation as an in-viewport Blender assistant. The Introduction established the core problem as micro-execution friction (mode, operator, panel path, and action order), while Related Work positioned Suzanne against two common alternatives: external learning resources and automation-first AI add-ons [3, 12, 7, 1]. Methods therefore focuses on *how* the system operationalizes those insights in software, interface behavior, and safety controls.

The project follows a design-and-build methodology common in applied HCI and educational tooling:

1. Define requirements from literature and practice (Blender learning pain points, in-context tutoring needs, and safety constraints).
2. Build an executable prototype inside Blender's N-panel.
3. Iterate on reliability and usability through repeated local testing in authentic modeling workflows.
4. Prepare measurable outputs for a later experimental chapter (task time, completion quality, and perceived usefulness).

Rather than treating model responses as opaque outputs, the implementation treats each interaction as a reproducible pipeline: user input -> validated request -> model response -> formatted procedural output in the viewport. This pipeline orientation is central to the methodological goal of reducing context switching and improving repeatable task execution.

4.2 Development process

4.2.1 Phase 1: Requirement extraction

Requirements were extracted from three sources: (a) Blender documentation and interface behavior [3], (b) prior studies on beginner friction in Blender [12], and (c) AI-learning-tool findings emphasizing in-context and actionable guidance [9].

The resulting requirement set emphasized:

- Locality: assistance must appear in the active workspace, not in a separate website.
- Procedural clarity: responses should be short, ordered, and immediately actionable.
- Safety: no silent scene modifications and no hidden execution of generated code.
- Practical deployment: installation and operation should fit student hardware and software constraints.

4.2.2 Phase 2: Prototype architecture and implementation

The first implementation target was a Blender add-on loaded through the standard add-on registration system (`register()` / `unregister()`), with persistent interaction state stored in `Scene` properties, user-level configuration stored in add-on preferences, and local conversation history persisted to disk with a temporary-directory fallback. This choice aligns with Blender's architecture and keeps the workflow entirely in-app [3].

Core interaction paths were implemented as operators:

- Text path: submit prompt -> optionally attach recent conversation turns and Blender Info history -> receive response.
- Voice path: start/stop microphone capture -> transcribe -> optionally attach recent context -> submit transcript -> receive response.
- Conversation path: create, select, rename, delete, and preview local conversations.
- Utility path: API-key validation, model refresh, microphone/transcription diagnostics, and recordings-folder access.

4.2.3 Phase 3: reliability hardening

After the initial feature set worked end-to-end, iteration prioritized failure behavior rather than feature expansion. The main hardening tasks were:

- Clear status signaling (`Ready`, `Recording...`, `Sending...`, `Error`).
- Explicit handling for missing keys, missing files, empty transcripts, empty outputs, and HTTP failures.
- Local fallback logic for recordings and conversation storage when add-on directories are not writable.
- UI formatting logic for long responses, output previews, and empty states so multi-step instructions remain readable in the panel.

These hardening steps were chosen because the dominant user risk in educational contexts is not only wrong answers, but interrupted or confusing workflows that break learner momentum.

4.2.4 Phase 4: evaluation readiness

The final development phase prepared the system for controlled comparison in later chapters by stabilizing the feature surface and defining what is considered in-scope behavior for experiments. At this stage, Suzanne is treated as a mixed-initiative assistant: it recommends, the user decides, and all scene edits remain user-mediated.

4.3 System requirements and traceability

To keep claims testable, each major thesis goal was mapped to an implementation responsibility and observable system behavior.

Table 1: Requirement-to-implementation traceability for Suzanne

Requirement	Design decision	Observable behavior
In-viewport assistance	N-panel integration in `VIEW_3D`	User never leaves Blender to ask for help
Procedural responses	Prompt shaping + UI formatting for numbered steps	Output appears as short action sequence
Input flexibility	Text prompt plus microphone-driven flow	Both typed and spoken intents are supported

Requirement	Design decision	Observable behavior
Context-aware help	Optional conversation memory and Blender Info-history attachment	Responses can incorporate recent workflow context when enabled
API transparency	Explicit key entry in preferences and key-test operator	User can verify connectivity before tasks
Fault tolerance	Guard checks and HTTP/IO error handling	Failures are visible and recoverable
Safety-first behavior	No automatic scene mutation from generated text	User remains the final actor
Responsible deployment	Local storage of settings, conversations, and recordings with no telemetry path	Lower privacy exposure for student use

This traceability table shaped coding priorities and chapter-level evaluation planning.

4.4 System architecture

Suzanne is implemented as a Blender-resident, event-driven assistant with three layers:

1. Interface layer: collapsible `Status`, `Ask`, `Voice`, `Context`, `Conversation`, and `Latest Output` cards in the N-panel.
2. Orchestration layer: operators that manage validation, request sequencing, recording toggles, conversation management, and state transitions.
3. Service layer: network calls for transcription and response generation, local audio capture utilities, and local JSON-backed conversation storage.

The architecture is intentionally simple because reliability and transparency were prioritized over autonomous behavior. Instead of hidden background orchestration, each major transition is user-triggered and surfaced in the UI.

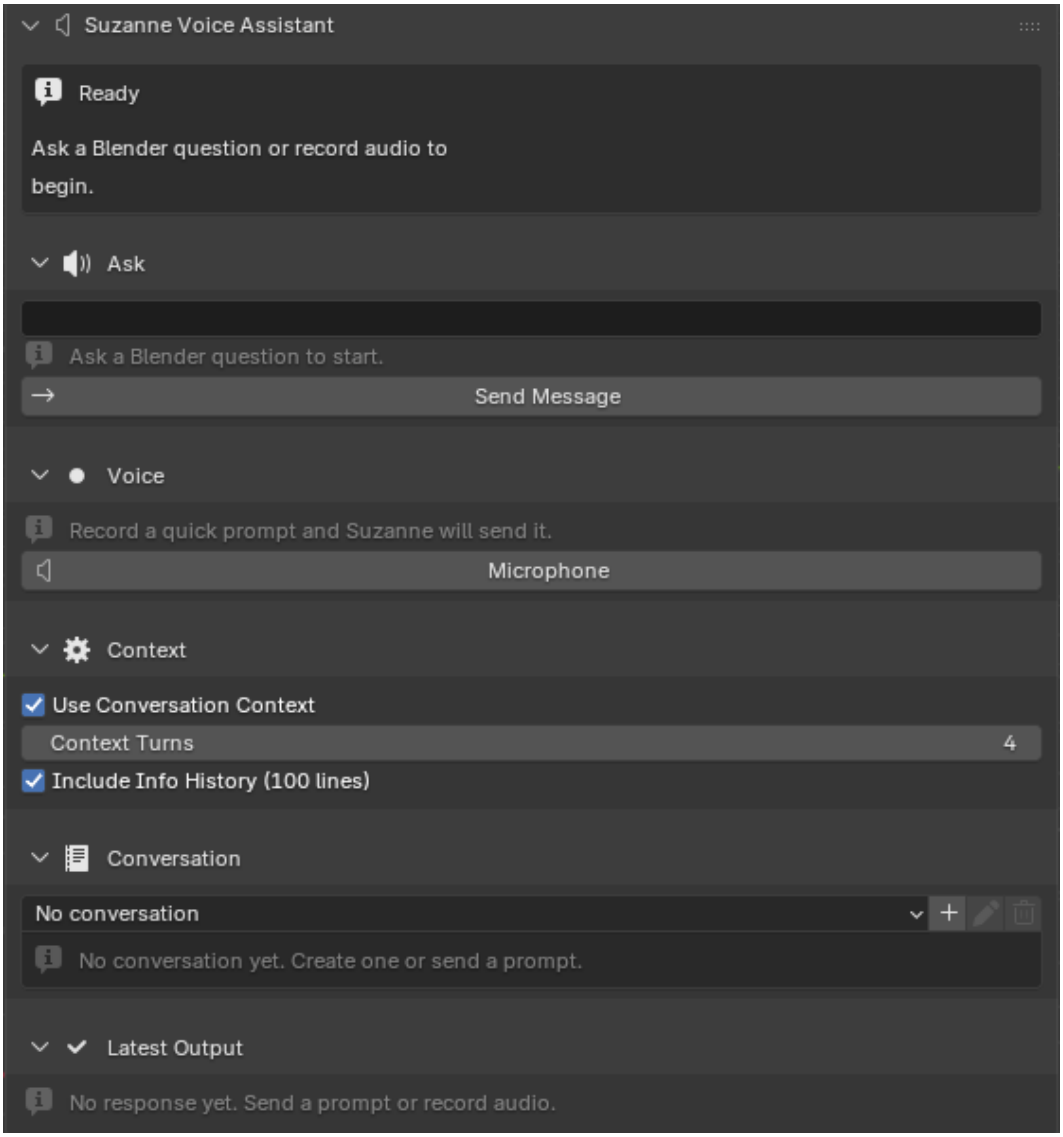

Figure 4: Current Suzanne interface in Blender's N-panel, showing the status card, text prompt area, voice control, context settings, conversation controls, and latest-output preview in the same workspace as the active scene.

In the implemented pipeline, the assistant does not introspect the full scene graph automatically. Scene awareness is inferred primarily from user prompts plus optional attached context: recent local conversation turns and the last 100 lines of Blender's Info history. This keeps integration lightweight while still allowing limited context-sensitive assistance, though it constrains precision for unusual scenes or workflows not visible in the recent interaction history.

4.5 Blender integration details

4.5.1 Registration and state model

Suzanne follows Blender add-on conventions for class registration and property initialization [3]. Runtime interaction data is maintained in `Scene` properties, including:

- Current status string.
- Current message prompt.
- Last transcript text and last model response.
- Last recorded audio file path.
- Active conversation selection and conversation-context settings.
- Info-history attachment toggle and the last captured Info-history block.
- Latest-output view and transcript/response expansion state.

Configuration values (API key, response/transcription model selections, file prefix, conversation auto-save behavior, and diagnostics feedback) are stored in add-on preferences. This separation was chosen so task state and configuration state remain distinct and easier to reason about during testing.

4.5.2 Panel design and interaction constraints

The panel is now structured around a set of Blender-native collapsible cards: `Status`, `Ask`, `Voice`, `Context`, `Conversation`, and `Latest Output`. This layout keeps the primary interaction loop visible while allowing supporting controls to remain compact until needed.

The interaction loop is:

1. Enter (or dictate) intent.
2. Optionally attach recent conversation turns or Blender Info history.
3. Send request.
4. Read the latest transcript or step-oriented response.
5. Apply steps manually in the scene.

A single microphone button toggles recording on/off to reduce control-surface complexity for beginners. The status card updates on each transition, functioning as lightweight feedback for asynchronous operations (recording, network request, response rendering). Empty states in the `Conversation` and `Latest Output` cards make it clear when no history or response is available yet, and long transcripts or responses can be expanded in place when necessary.

4.5.3 Cross-platform audio capture strategy

Because Blender is cross-platform and student devices vary, audio capture was implemented with OS-specific command paths:

- Linux: `ffmpeg` with ALSA input.
- Windows: `ffmpeg` with DirectShow input.
- macOS: bundled `atunc` utility for capture.

Recorded files are normalized to mono 16 kHz WAV for consistent transcription behavior. If the add-on directory cannot store recordings, the system falls back to a temporary directory. This avoids hard failures in locked-down lab environments.

4.6 End-to-end interaction workflows

4.6.1 Text workflow

The text path was designed for direct, low-latency interaction from the viewport.

```
Algorithm 1: Text request handling
Input: user_prompt
Output: formatted procedural response in N-panel

1: if user_prompt is empty then
2:     show validation error in panel
3:     return
4: end if
5: read API key from add-on preferences
6: if API key missing then
7:     show key error and return
8: end if
9: collect optional conversation context and Blender Info history if enabled
10: apply Blender-only prompt prefix and build request payload
11: send request to response model endpoint
12: parse output_text (or structured fallback content)
13: store response in scene state and append local conversation exchange
14: render wrapped lines in response box
```

This workflow is intentionally explicit and synchronous from the user's perspective. There are no hidden retries or silent fallbacks that could obscure what happened during a request.

4.6.2 Voice workflow

The voice path extends the text workflow by inserting capture and transcription stages.

```
Algorithm 2: Voice request handling
Input: microphone toggle events
Output: transcript + procedural response in N-panel

1: on first press, start recording process and set status=Recording
2: on second press, stop process and wait for output file
3: if file missing then show error and abort
4: send audio file to transcription endpoint
5: if transcript empty then show error and abort
6: collect optional conversation context and Blender Info history if enabled
7: apply Blender-only prompt prefix
8: send transcript to response endpoint
9: store transcript, file path, response, and local conversation exchange
10: render transcript and response in panel
```

This two-press model was selected over push-to-talk hold behavior because it lowers

motor-demand complexity for novices and allows longer utterances without continuous key holding.

4.7 Network and model interaction layer

4.7.1 API endpoints and payload flow

The implementation uses HTTPS requests to model APIs for two tasks:

- Audio transcription (`/v1/audio/transcriptions`) with multipart file payloads.
- Text response generation (`/v1/responses`) with JSON payloads.

A lightweight key-test operation (`/v1/models`) is provided in preferences to reduce setup uncertainty before first use. This small affordance significantly reduced setup friction during internal testing because users can distinguish key issues from prompt-quality issues.

4.7.2 Error handling and response robustness

The system treats network interaction as failure-prone and therefore includes guarded parsing and user-facing error messages for:

- Missing or malformed API keys.
- HTTP transport failures.
- Non-JSON or unexpected response structures.
- Empty transcripts or empty model outputs.

When primary response fields are absent, the parser attempts structured fallback extraction from nested output content. This improves resilience across model-response format differences while keeping the UI contract stable.

4.8 Grounding and response-formation strategy

A major methodological goal is grounding outputs in authoritative Blender language so instructions remain reproducible and verifiable [3, 6]. The full grounding strategy is defined in three layers:

1. **Domain constraint layer.** An always-applied Blender-only prefix prevents off-domain drift and keeps responses task-focused.
2. **Terminology alignment layer.** Prompting style favors explicit mode names, operator names, and panel paths.
3. **Retrieval layer.** A retrieval-augmented extension is specified to inject relevant Manual passages before generation.

The current evaluated build implements layers (1) and (2) directly and is architected to accept layer (3) as a modular extension. This allows transparent reporting of what is already operational versus what is specified for the full thesis target.

For the retrieval extension, passage ranking follows standard vector-similarity scoring [6]:

$$\text{score}(q, d_i) = \cos(\mathbf{e}_q, \mathbf{e}_{d_i}) = \frac{\mathbf{e}_q \cdot \mathbf{e}_{d_i}}{\|\mathbf{e}_q\| \, \|\mathbf{e}_{d_i}\|}$$

where $\mathbf{e}_q$ is the query embedding and $\mathbf{e}_{d_i}$ is the embedding of document chunk d_i. Top-ranked chunks are then inserted into the generation context to reduce terminology drift and menu-path hallucination.

4.8.1 Response schema for procedural clarity

Regardless of input modality, the response format is designed to preserve instructional structure:

- Short, ordered steps.
- Explicit prerequisite states (mode, selection assumptions).
- Concrete operator/menu naming where possible.
- Troubleshooting branches when likely failure points are detected.

This schema reflects findings from AI-learning literature that actionable, context-proximate feedback is more useful than generic prose [9].

4.9 Safe code-assistance model

Related work showed that automation-first Blender copilots often execute generated code quickly, which improves speed but can increase risk [7, 1, 5]. Suzanne's method is deliberately conservative:

- Primary output is human-readable procedure, not autonomous execution.
- Any code-like content is treated as optional scaffolding for user inspection.
- Scene changes remain user-initiated in Blender.

For the planned guarded execution extension, the policy model includes:

1. Explicit confirmation before any run action.
2. Restricted operation classes (object creation/transforms/lights/cameras/shader nodes).
3. Blocked operations for high-risk file/network/system effects.
4. Immediate rollback guidance using Blender's undo stack.

By separating *advice* from *execution authority*, the method keeps user agency central and aligns with security guidance on minimizing model-side permissions [5].

4.10 Responsible-computing controls in implementation

Ethical concerns were translated into concrete implementation controls rather than left as abstract policy.

4.10.1 Privacy and data minimization

- API keys are user-supplied in local add-on preferences.
- No separate telemetry service is embedded in the add-on.
- Conversation history and recordings are stored locally on the user's machine.
- Data sent externally is limited to explicit user inputs plus any context blocks the user chooses to attach (recent conversation turns and/or recent Blender Info history).

This local-first approach reduces unnecessary data propagation while acknowledging that third-party API processing remains part of the architecture [5].

4.10.2 Transparency and cost visibility

The system surfaces failures directly (e.g., key, quota, network, decode) instead of silently degrading output quality. Making failure modes visible helps users manage API budgets and prevents misattributing infrastructure issues to user competence.

4.10.3 Inclusivity by instruction style

UI output is cleaned and line-wrapped for readability, and markdown-heavy formatting is normalized before display. The intent is to improve clarity for novices and non-native readers by emphasizing operational language over stylistic flair.

4.11 Implementation environment and reproducibility

4.11.1 Software stack

The prototype runs as a Blender add-on for Blender 5.0.0+ [3], using Python within Blender's runtime and standard libraries for process and HTTP orchestration. External tooling dependencies are intentionally minimal:

- `ffmpeg` (Linux/Windows) for microphone capture.
- bundled `atunc` utility (macOS) for microphone capture.
- Network access to model APIs for transcription and response generation.

4.11.2 Reproducibility protocol

To support repeatable demonstrations and evaluation setup, the following run protocol was used:

1. Install/enable add-on in Blender.
2. Configure API key and response/transcription model settings in preferences.
3. Run built-in diagnostics such as `Test API Key`, `Test Microphone`, and `Test Transcription` as needed.
4. Set context options (`Use Conversation Context`, `Context Turns`, `Include Info History (100 lines)`) for the trial.
5. Run fixed benchmark prompts/tasks and record completion observations.

Because Blender versions and OS audio stacks vary, environment metadata (OS, Blender version, selected models, and whether conversation or Info-history context was enabled) is logged as part of experiment setup documentation.

4.11.3 Versioned capability statement

To avoid overclaiming, methods reporting distinguishes current implementation from scoped extension work.

Table 2: Implemented capabilities versus scoped extensions

Capability area	Implemented in current build	Scoped extension
In-viewport text assistant	Yes	N/A
Voice capture and transcription	Yes	N/A
Blender-only domain gating	Yes (always-on prompt prefix)	Richer intent classification
Conversation/context support	Yes (local conversation memory + optional Info-history attachment)	Deeper scene introspection and richer grounding
Retrieval grounding from Manual	Partial (prompt-level alignment)	Full chunk retrieval + citation injection
Procedural step formatting	Yes	Adaptive difficulty/fading
Code execution inside add-on	No autonomous execution	Guarded, opt-in constrained runner
User safety controls	Yes (validation/status/errors)	Formal policy engine and audit trails

This separation supports methodological integrity: the chapter captures both delivered engineering work and the explicit next-step architecture required to fully realize the thesis design goals.

4.12 Methods-level limitations

Several methodological constraints influence interpretation of later results:

- Scene-context inference is still indirect: it relies on prompts, recent conversation turns, and Info-history snapshots rather than full scene introspection.
- Grounding is currently strongest at terminology/prompt levels, with full RAG integration staged as extension work.
- API-dependent behavior introduces latency and availability variability outside Blender control.
- Microphone quality and device configuration can affect transcription quality and therefore downstream instruction quality.

These limits are not hidden defects; they are declared boundaries that shape valid claims in evaluation and discussion chapters.

4.13 Transition to evaluation

This Methods chapter established the system design and implementation pipeline used to operationalize Suzanne as an in-viewport instructional assistant. The next chapter evaluates this method through software verification and task-based experiments aligned with portfolio-relevant Blender tasks.

5 Experiments

This chapter evaluates Suzanne at two levels. First, it reports a completed software-verification pass over the Blender add-on implementation. Second, it presents three task-based experiments that demonstrate how Suzanne supports representative portfolio-oriented Blender workflows inside the viewport. This structure keeps the claims matched to the evidence: the automated suite evaluates deterministic software behavior, while the task experiments evaluate whether Suzanne can deliver usable instructional support for simple, complex, and context-aware workflows.

5.1 Experimental Design

5.1.1 Evaluation goals and research questions

The evaluation is organized around three practical research questions:

1. **RQ1: Reliability.** Do Suzanne's core interaction paths behave consistently enough to support repeated use inside Blender?
2. **RQ2: Task support.** Can Suzanne provide usable in-viewport guidance for representative Blender tasks without forcing the workflow out into external search?
3. **RQ3: Instructional clarity and context.** Do Suzanne's responses remain clear, actionable, and context-sensitive enough to support learning-oriented Blender work?

These questions follow directly from the claims made in the Introduction and Methods chapters. Suzanne is not framed as a fully autonomous copilot; it is framed as an in-viewport instructional tool whose value depends on stable interface behavior, actionable task guidance, and user trust [4, 9].

5.1.2 Staged evaluation structure

The evaluation is staged across deterministic verification and task-based demonstrations.

Table 3: Evaluation layers used in this chapter

Evaluation layer	Purpose	Evidence presented here
Software verification	Confirm deterministic behavior of operators, panel rendering, preferences, and state transitions	Automated Python test suite with `65` passing checks
Task-based evaluation	Demonstrate usable in-viewport guidance across representative Blender workflows	Three authored experiments covering basic question answering, complex procedure generation, and context-aware action reconstruction

This design fits the realities of Blender add-on development. Some claims are best tested through automation, such as whether blank prompts are rejected or whether an error state is rendered correctly. Other claims are best illustrated through concrete task runs in the Blender interface, where the usefulness of the resulting guidance can be seen directly.

5.1.3 Evaluation scope

This chapter reports completed software-verification results and task demonstrations executed inside Blender. Accordingly, the claims supported here are about system stability, procedural usefulness, and task coverage rather than population-level usability outcomes.

5.2 Evaluation

5.2.1 Completed software verification

Before the task demonstrations, the build was tested as a software artifact. The Suzanne repository includes a Python test suite covering operators, panel behavior, preferences, state registration, and shared utility functions. These tests use a mocked `bpy` environment so that Blender-specific logic can be exercised repeatably without manual clicking inside the UI. This is especially useful for edge cases that are tedious to reproduce by hand, such as missing API keys, offline requests, empty output panes, or repeated property registration.

The first set of checks verifies that Suzanne rejects invalid input early and with a readable message:

```
def test_send_message_execute_rejects_blank_prompt():
    modules = load_suzanne_modules()
    context = make_context(
        modules.common.ADDON_MODULE,
        scene=make_scene(suzanne_va_prompt="   "),
    )
    operator = modules.operators.SUZANNEVA_OT_send_message()

    result = operator.execute(context)

    assert result == {"CANCELLED"}
    assert operator._reports[-1][1] == "Please type a message first."
```

This test matters because a tutoring tool that fails opaquely can interrupt learner momentum faster than one that simply refuses an invalid request. The desired behavior is not only cancellation, but cancellation with a concrete, beginner-readable explanation.

The next example verifies that network failure surfaces a visible error state instead of silently failing:

```
with mock.patch.object(
    modules.operators,
    "_call_chatgpt",
    side_effect=modules.operators.URLError("offline"),
):
    result = operator.execute(context)

assert result == {"CANCELLED"}
assert scene.suzanne_va_status == "Idle (error)"
assert "Send failed" in operator._reports[-1][1]
```

This supports one of the main design goals established in Methods: failure should be explicit and recoverable. In a classroom or portfolio workflow, a user needs to know whether a poor outcome came from their Blender steps, their prompt, or the external service connection.

A third group of checks verifies that the panel communicates state clearly:

```
scene.suzanne_va_status = "Idle (error)"
assert sidebar._status_presentation(scene, False) == (
    "Error",
    "Idle (error)",
    "ERROR",
    True,
)
```

Even though this is a small UI test, it is directly relevant to the thesis argument. Suzanne is meant to reduce micro-execution confusion, so its own status language must stay simple and legible. The suite also checks other presentation branches, including `Ready`, `Sending...`, `Recording...`, conversation empty states, and the hiding of API-key details in preferences.

The `65` passing checks map directly onto the verification areas summarized in the following table. The table is therefore not a separate high-level abstraction; it is a grouped explanation of what those `65` checks were actually checking for across the add-on.

Table 4: Summary of current automated verification outcomes

Test family	What the checks verified	Why it matters
Prompt validation	Blank or whitespace-only prompts are rejected and reported in readable language	Prevents confusing empty requests and keeps feedback beginner-friendly
Send pipeline	Successful requests update transcript, latest response, conversation state, and status fields consistently	Confirms the core interaction loop works end to end
Error handling	Offline or API failures surface explicit error states and readable diagnostics	Makes failures recoverable instead of silent
Panel usability	Status cards, empty states, previews, and UI presentation branches render expected labels and messages	Keeps the interface legible during real task work
Preference safety	API-key controls and diagnostics behave correctly without exposing sensitive setup details	Supports trust and safer configuration

Test family	What the checks verified	Why it matters
State lifecycle	Scene properties register idempotently, persist correctly, and clear without corruption	Prevents instability across repeated runs

Together, these categories account for the breadth of the `65` checks. The significance of the test suite is therefore not only the number itself, but the spread of coverage across validation, interaction flow, failure behavior, interface clarity, and add-on state management.

When the local test suite was executed, all `65` checks passed in `0.30` seconds. This is not the same as proving that every model-generated instruction is correct. What it does show is that the non-model scaffolding around Suzanne is stable: validation works, user-facing failures are surfaced, panel states are coherent, and repeated runs do not corrupt add-on state. For a system intended to support learners, this reliability layer is a necessary prerequisite for broader usability claims.

5.2.2 Task-based evaluation

The three task-based experiments below move from a simple instructional query to a longer procedural workflow and then to a context-aware reconstruction task. Together, they show how Suzanne behaves when used as an in-viewport guide for increasingly demanding kinds of support.

5.2.2.1 Experiment 1: Basic question answering in the Blender viewport The first task-based experiment tests Suzanne's most basic instructional path: whether a user can ask a simple Blender question inside the add-on and receive a correct, readable, and immediately actionable response without leaving the interface. For this trial, the prompt entered into Suzanne was, "How do I add a light to my scene in Blender?" This is a suitable first test because lighting is a common beginner workflow, the expected procedure is easy to verify, and the task exercises the core text-query pipeline from prompt entry to visible response.

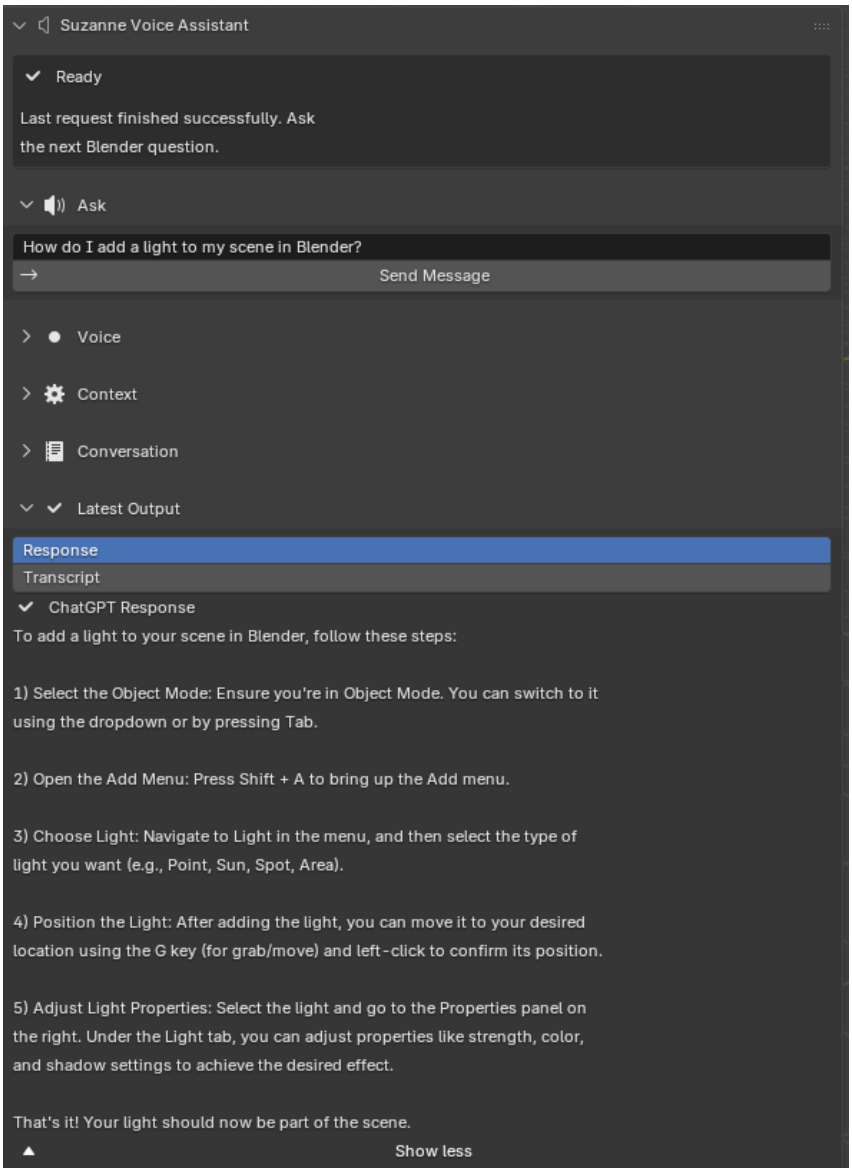

Figure 5: Experiment 1 screenshot showing Suzanne answering a simple lighting question inside Blender's N-panel.

Suzanne returned a numbered, step-by-step answer that instructed the user to remain in Object Mode, open the Add menu with `Shift + A`, choose a light type, position the light, and then adjust its properties in the right-side panels. This response is functionally correct for standard Blender interaction and uses interface terms that match what a beginner would actually see on screen. Just as importantly, the answer is procedural rather than vague. Instead of describing lighting conceptually, Suzanne gives the user a sequence they can follow immediately in the same workspace.

Table 5: Summary of Experiment 1

Aspect	Observation
Goal	Verify that Suzanne can answer a simple Blender question correctly inside the N-panel
Prompt	"How do I add a light to my scene in Blender?"
Observed output	Suzanne produced a short numbered procedure for adding, placing, and adjusting a light
Outcome	Successful
Interpretation	Suzanne's baseline question-answering workflow functioned correctly and provided usable in-viewport guidance

This experiment establishes that Suzanne's simplest interaction loop is already useful at the point of use. Before the system can be trusted with longer workflows, it must first show that it can handle ordinary interface questions correctly and clearly.

5.2.2.2 Experiment 2: Complex procedural guidance for fire simulation The second task-based experiment tests whether Suzanne can support a more advanced Blender workflow that requires multiple ordered setup steps rather than a short, single-action answer. For this trial, the prompt entered into Suzanne was, "How do I create a basic fire simulation in Blender?" Fire simulation is a stronger stress test than the first experiment because it involves several connected systems, including a simulation domain, a flow emitter, physics settings, material setup, and baking or caching behavior. In other words, the task is complex enough that an incomplete or poorly ordered answer would be much harder for a user to apply successfully.

Because Suzanne's response exceeded the visible height of the N-panel, the result was captured in two screenshots.

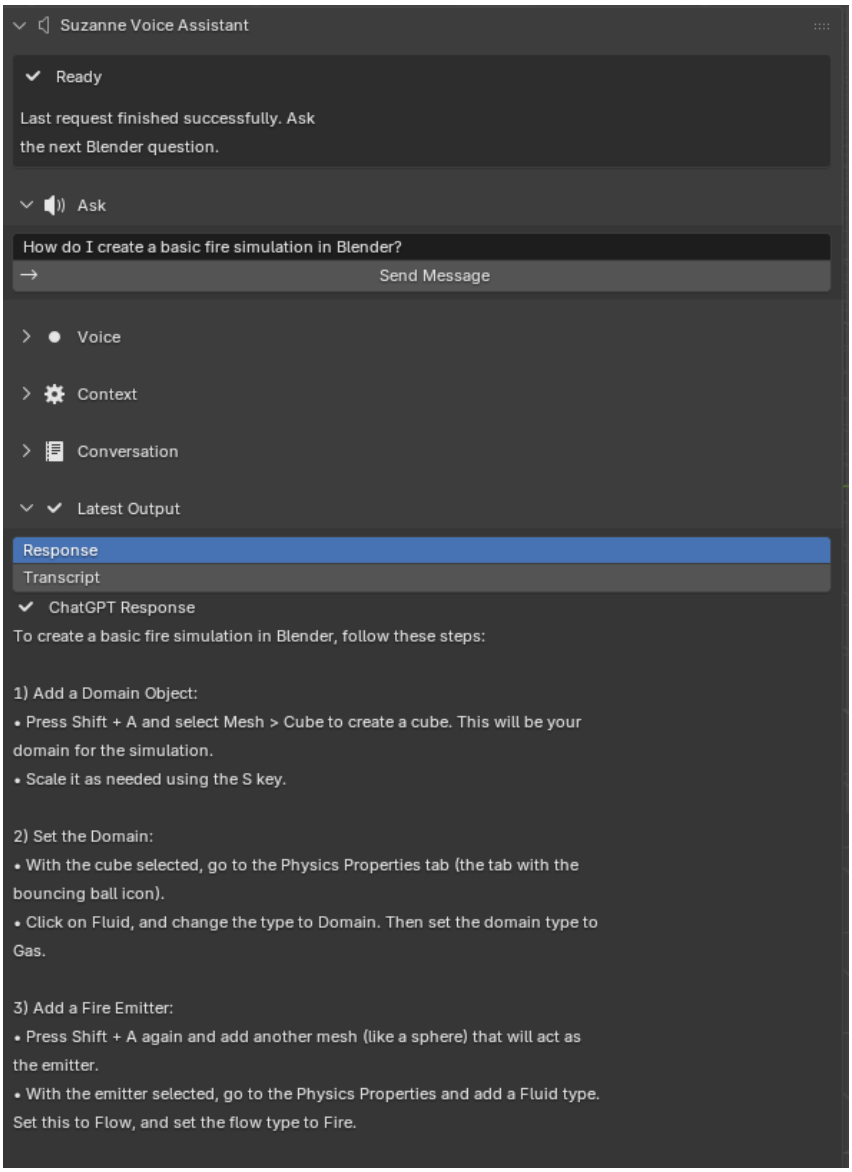

Figure 6: Experiment 2 screenshot showing the first part of Suzanne's fire-simulation response in Blender's N-panel.

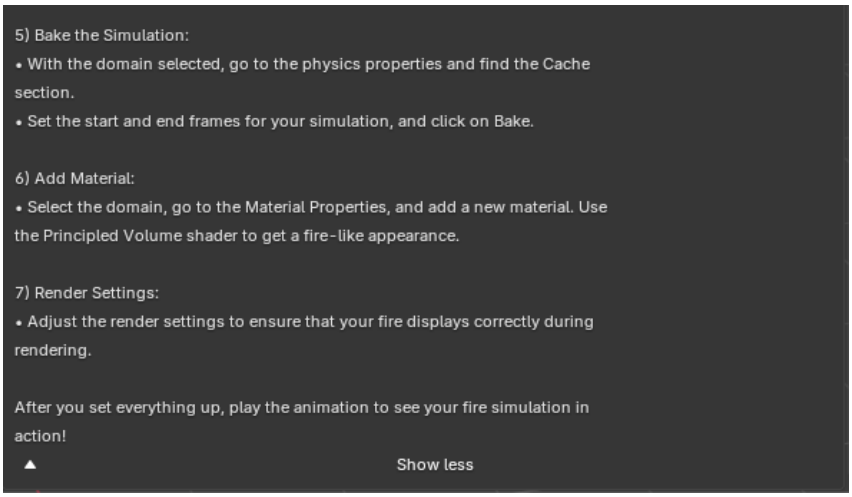

Figure 7: Experiment 2 screenshot showing the continuation of Suzanne's fire-simulation response in Blender's N-panel.

Suzanne returned a structured, ordered procedure that included creating a domain object, configuring the domain as a gas simulation, adding a separate emitter, setting the emitter as a fire flow source, baking the simulation, assigning a material, and adjusting render settings. This is the kind of longer procedural answer that Suzanne is intended to support: it keeps the user inside Blender while still presenting a workflow that would otherwise require searching across multiple external references.

The generated response is also broadly consistent with the standard workflow described in the Blender Manual, which explains that gas simulations require at least a domain object and a flow object, followed by material assignment and cache baking [3]. Suzanne's answer therefore appears substantively correct at the workflow level, even though this experiment documents procedural completeness rather than a timed benchmark.

Table 6: Summary of Experiment 2

Aspect	Observation
Goal	Evaluate whether Suzanne can provide usable guidance for a more complex Blender simulation task
Prompt	"How do I create a basic fire simulation in Blender?"
Observed output	Suzanne produced a multi-step workflow covering domain setup, emitter setup, bake steps, material assignment, and render considerations
Outcome	Successful as a complex-response test

Aspect	Observation
Interpretation	Suzanne handled a longer, more technically demanding query and returned guidance that broadly matches documented Blender workflow

This second experiment strengthens the evaluation by showing that Suzanne is not limited to very short beginner questions. It can also generate longer instructional sequences for tasks that involve several dependent setup stages, which is central to the thesis claim that in-viewport guidance can reduce friction for practical Blender work.

5.2.2.3 Experiment 3: Context-aware reconstruction of recent Blender actions The third task-based experiment evaluates Suzanne's context feature rather than its general question-answering ability alone. In this trial, the `Include Info History (100 lines)` option was enabled in the Context panel, and the user asked, "what actions did I just perform in Blender? Please summarize them in order and explain what I was trying to do." This is an important test because it asks Suzanne to infer recent activity from Blender session history instead of answering a generic procedural question from prior knowledge alone.

As with the previous experiment, the response extended beyond the visible panel height and was captured in two screenshots.

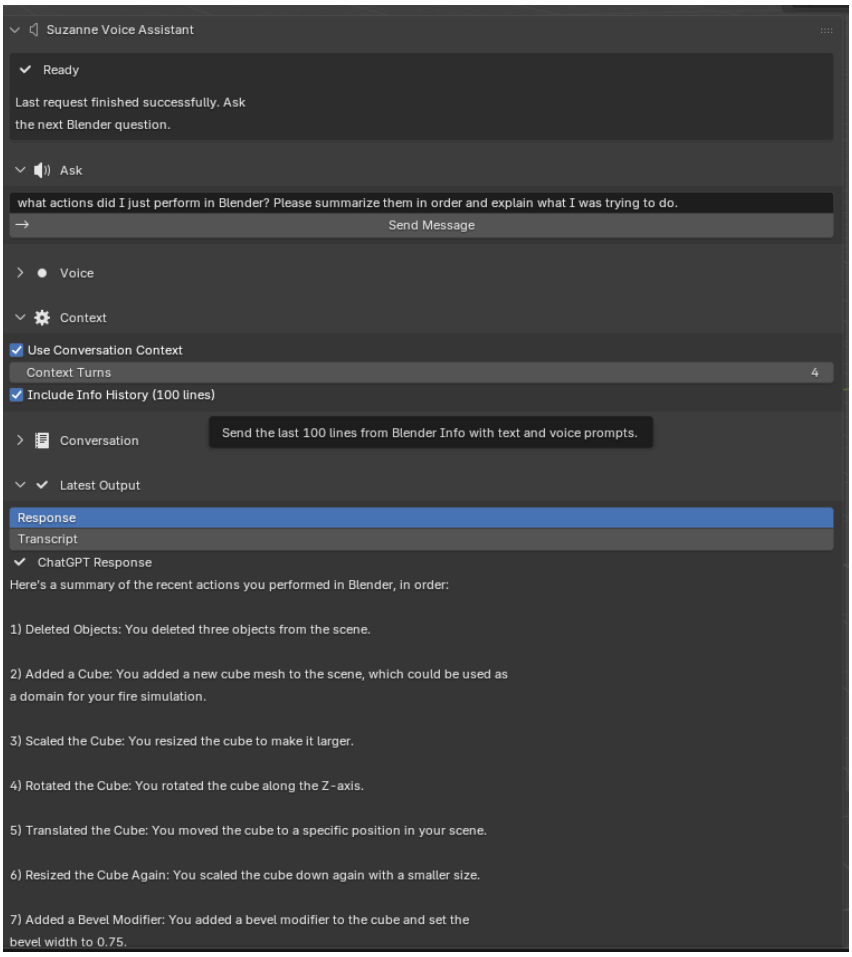

Figure 8: Experiment 3 screenshot showing the prompt and the first part of Suzanne's context-aware response in Blender's N-panel.

Figure 9: Experiment 3 screenshot showing the continuation of Suzanne's context-aware response in Blender's N-panel.

Suzanne responded with an ordered summary of recent scene operations, including deleting objects, adding a cube, scaling it, rotating it, translating it, resizing it again, adding a bevel modifier, and applying smooth shading. It then interpreted those actions as part of a likely modeling workflow, specifically suggesting that the cube may have been prepared as a domain object for a fire simulation. This is a meaningful result because it shows Suzanne using recent session context to produce a situationally grounded explanation rather than only returning generic help text.

Methodologically, this experiment is especially relevant to the thesis because it addresses

one of the core limitations of many external help sources: they do not know what the user has just done. By contrast, Suzanne can incorporate Blender's recent Info history and reflect it back into the conversation. In the captured session, conversation context was also enabled, so this example should be interpreted as evidence of context-aware assistance rather than as an isolated benchmark of Info-history retrieval alone. Even with that caveat, the response clearly tracks recent viewport activity in a way that ordinary static documentation cannot.

Table 7: Summary of Experiment 3

Aspect	Observation
Goal	Evaluate whether Suzanne can use recent Blender session context to infer and summarize user actions
Prompt	"what actions did I just perform in Blender? Please summarize them in order and explain what I was trying to do."
Observed output	Suzanne reconstructed an ordered sequence of recent modeling operations and inferred the likely purpose of the workflow
Outcome	Successful as a context-aware assistance test
Interpretation	Suzanne used recent session context to generate a grounded, workflow-specific explanation rather than only generic Blender advice

Together, the three experiments illustrate a clear progression of capability: basic question answering, longer procedural guidance for a complex task, and context-aware interpretation of recent user actions. That progression supports the thesis claim that Suzanne is not merely a generic chatbot embedded in Blender, but a more situated instructional assistant designed to reduce micro-execution friction inside the viewport.

5.2.3 Answers to the research questions

5.2.3.1 RQ1: Reliability RQ1 is answered positively. The software-verification layer shows that Suzanne's core interaction paths behave predictably across input validation, request handling, failure recovery, panel presentation, preference controls, and state registration. The `65` checks matter because they cover the full support structure around the add-on rather than only a single happy-path request.

5.2.3.2 RQ2: Task support RQ2 is also supported by the task-based evaluation. Across the three experiments, Suzanne kept guidance inside Blender for a beginner lighting question, a longer fire-simulation workflow, and a context-aware explanation of recent user activity. In each case, the output was specific enough to support continued work in the viewport rather than forcing the user out to search for the next step elsewhere.

5.2.3.3 RQ3: Instructional clarity and context RQ3 is supported by the observed response quality. Suzanne's outputs were readable, step-oriented, and aligned with Blender

terminology, and the third experiment showed that the system can incorporate recent Info-history context to produce situationally grounded guidance. Taken together, the experiments show that Suzanne is not only present in the interface, but capable of producing assistance that is concrete enough to be useful for learning-oriented work.

5.3 Threats to Validity

5.3.1 Internal validity

The task-based experiments use author-selected prompts and scene setups, so task choice can influence how strong Suzanne appears. Familiar workflows may naturally produce stronger outputs than unusual scenes or ambiguous prompts. External API availability and network latency can also affect response timing and wording across runs.

5.3.2 Construct validity

Time-on-task and perceived trust are not directly measured in this chapter, so the evaluation should not be read as a complete learning study. The automated test suite measures software reliability rather than pedagogical truthfulness, and the task demonstrations show procedural usefulness for selected workflows rather than long-term retention or transfer. Passing checks show that Suzanne behaves consistently as an add-on; they do not guarantee that every generated instruction sequence is correct in every Blender scene.

5.3.3 External validity

The selected tasks emphasize beginner and intermediate portfolio workflows. That is appropriate for Suzanne's scope, but it limits generalization. Results from lighting setup, fire simulation, and modeling-context reconstruction should not be overstated as evidence for advanced rigging, simulation pipelines, compositing, or studio-scale production work. Generalization is also constrained by Blender version, English-language UI assumptions, prompt phrasing, and local hardware differences.

5.3.4 Conclusion validity

The evidence in this chapter is descriptive and qualitative rather than inferential. It supports claims about reliability and demonstrated task support, but it does not justify broad quantitative claims about efficiency gains or superiority over alternative tools. The defensible conclusion is that Suzanne is technically stable and demonstrably useful for the representative workflows evaluated here.

Overall, these validity threats do not negate the value of the evaluation. They clarify the kind of claim this chapter can support: initial evidence that a carefully scoped, in-viewport Blender assistant is technically reliable and practically useful for reducing micro-execution friction on common portfolio-building tasks.

6 Conclusion

This thesis began from a practical problem: Blender is powerful, but beginners often lose momentum not because they lack creative ideas, but because they get stuck on micro-execution. They need to know which operator to call, which mode to use, which panel to open, and what order to follow. Existing help systems often answer these questions outside Blender through manuals, videos, forums, or automation-heavy AI tools. Suzanne was developed as a different response to that problem: an in-viewport instructional assistant that lives in Blender's N-panel, keeps help close to the active scene, and returns short, actionable guidance rather than opaque automation.

Across the thesis, Suzanne was framed not as a fully autonomous copilot, but as a mixed-initiative learning tool shaped by three design priorities. First, it keeps assistance local to the workspace, reducing the need to switch away from Blender [4]. Second, it emphasizes procedural clarity through step-based responses aligned with Blender terminology and interface structure [3, 9]. Third, it preserves user control through explicit feedback, bounded scope, and the refusal to silently modify scenes or execute arbitrary actions without oversight [5].

6.1 Summary of Results

The implemented system shows that this design is technically viable. Suzanne now supports typed prompts, microphone-based prompt capture, local conversation memory, optional inclusion of recent Blender Info history, status-aware interface feedback, and a structured `Latest Output` review workflow. The Methods chapter demonstrated that these features are not just interface decoration; they are part of a reproducible interaction pipeline with clear validation, state transitions, and recovery behavior. In architectural terms, Suzanne successfully operationalizes the central thesis idea that AI guidance can be embedded directly into Blender without collapsing into hidden automation.

The strongest completed evidence in the thesis is the software-verification layer. The automated test suite passed all `65` checks, covering prompt validation, send behavior, failure handling, panel rendering, preference safety, and state lifecycle behavior. This does not prove that every generated instruction is always correct, but it does show that the non-model scaffolding around Suzanne is stable enough to support repeated use and task-based evaluation. That reliability matters because instructional tools fail pedagogically when their own interface behavior is confusing or inconsistent.

The three use-case experiments also support the thesis argument at a practical level. The first experiment showed that Suzanne can answer a straightforward Blender question with a clear, usable procedure inside the N-panel. The second showed that Suzanne can generate a longer workflow for a more complex task, in this case a basic fire simulation, without collapsing into vague or purely conceptual advice. The third showed that Suzanne can use recent Blender session context to summarize what the user has just done, suggesting a path toward more situationally grounded assistance. Together, these results support the claim that Suzanne is more than a generic chatbot embedded in Blender. It functions as a scoped, context-sensitive instructional add-on aimed at reducing friction in portfolio-oriented Blender work.

At the same time, the thesis remains methodologically careful about what has and has not yet been proven. The current evidence is strongest on implementation quality, reliability, and demonstrated task support. The most defensible conclusion is that Suzanne is a credible

and well-scoped prototype with clear evidence of technical stability, practical usefulness in representative Blender workflows, and a design that aligns with both prior literature and practical Blender learning needs.

6.2 Future Work

The most immediate next step is to complete the planned human-facing evaluation. A within-subject comparison between Suzanne and external-search workflows would make it possible to measure time-on-task, completion quality, context switching, recovery burden, and perceived usefulness under controlled conditions. That study would be especially valuable for testing whether the instructional advantages suggested by the current experiments translate into measurable gains for students and early-career creators working on realistic Blender tasks.

Beyond evaluation, Suzanne itself has several natural expansion paths. One major direction is deeper grounding. The current build already aligns strongly with Blender terminology and can attach recent interaction context, but a fuller retrieval layer over the Blender Manual could improve menu-path accuracy, reduce hallucinated steps, and make it possible to cite or surface specific documentation passages alongside responses [6]. Another direction is richer scene awareness. Right now Suzanne infers context indirectly through prompts, conversation memory, and Info-history snippets. Future versions could inspect selected objects, active modes, visible modifiers, material slots, or render settings directly, allowing the assistant to tailor guidance more precisely to the user's actual scene state.

Another important future direction is adaptive pedagogy. Suzanne currently returns concise procedural steps, but later versions could adjust explanation depth for different learners, provide optional troubleshooting branches automatically, or gradually fade support as users become more confident. For example, a beginner-facing mode might emphasize every prerequisite and panel path, while a more advanced mode could focus on only the critical actions or likely failure points. This would move Suzanne closer to a true tutoring system while preserving its in-viewport usability.

The future I find most exciting, however, is broader than Suzanne alone. This project suggests the beginning of a family of Blender add-ons built around the same philosophy: small, safe, in-context tools that reduce friction for creators without taking control away from them. Suzanne could grow into a wider ecosystem that includes add-ons for portfolio review, lighting critique, material setup guidance, scene-preparation checklists, topology analysis, or pipeline documentation. One add-on might help users prepare clean presentation renders; another might explain shader nodes in plain language; another might assist with organizing assets or documenting reproducible scene workflows. In that sense, Suzanne is not only a single tool but also a proof of concept for a broader design pattern in Blender add-on development.

That larger add-on ecosystem could also support my own future work as a developer and researcher. Rather than building one increasingly monolithic assistant, I can imagine creating a set of specialized tools that share a common design language: clear status feedback, grounded terminology, strong safety guardrails, and support for learning through doing. Suzanne can remain the general in-viewport tutor, while other add-ons explore adjacent needs in 3D creation. This would allow future projects to remain focused and maintainable while still contributing to the same overall goal: making Blender more approachable, more teachable, and more productive for students and independent artists.

6.3 Future Ethical Implications and Recommendations

If Suzanne or future related add-ons are released more broadly, ethical considerations will remain central. The first major issue is privacy. Because prompts, audio, and optional context may be sent to a third-party provider, users must understand what leaves their machine and what does not. Future public versions should continue to make API usage explicit, avoid hidden telemetry, and clearly label any contextual data attached to a request. In educational settings, users should always have a non-AI alternative so that students are not forced into third-party processing to complete coursework [5, 9].

The second issue is reliability and over-trust. Even when grounded on documentation, language-model outputs can still be incomplete, outdated, or slightly misaligned with a given scene. Future versions should therefore keep Suzanne's advisor role visible. The system should continue to present suggestions as suggestions, preserve opt-in behavior for any code execution, and encourage verification against Blender's official documentation when appropriate [3]. As richer scene awareness or automation features are added, the design should remain conservative: visibility before action, confirmation before execution, and undo-friendly workflows wherever possible.

The third issue is equity and access. Tools like Suzanne can help lower the barrier to Blender, but they can also create new inequalities if they depend on paid APIs, stable internet access, or English-only documentation. Future development should therefore prioritize cost transparency, graceful degradation when AI services are unavailable, and eventual exploration of cheaper or local model options. If I expand into additional add-ons, I should carry forward the same principle: the tool should help learners build skill and confidence, not create new hidden dependencies that only some users can afford.

Overall, this project argues that AI in creative software is most promising when it is narrow enough to be trustworthy, visible enough to be inspectable, and supportive enough to help users keep ownership of their work. Suzanne does not solve every Blender learning problem, and it is not yet the final form of an intelligent in-viewport tutor. What it does show is that there is real value in designing AI assistance around instructional clarity, user agency, and workflow locality. That combination offers a practical foundation not only for Suzanne's continued growth, but also for a wider future of Blender add-ons that teach, assist, and empower rather than simply automate.

References

[1] Siddharth Ahuja. 2025. BlenderMCP: Blender Model Context Protocol Integration. https://github.com/ahujasid/blender-mcp. GitHub repository. Accessed 2025-12-04.

[2] Arijan Belec. 2022. *Blender 3D Incredible Models: A Comprehensive Guide to Hard-Surface Modeling, Procedural Texturing, and Rendering*. Packt Publishing, Birmingham, UK.

[3] Blender Foundation. 2025. *Blender Manual*. https://docs.blender.org/manual/en/latest/ Accessed 2025-11-11.

[4] Parmit K. Chilana, Nathaniel Hudson, Srinjita Bhaduri, Prashant Shashikumar, and Shaun Kane. 2018. Supporting Remote Real-Time Expert Help: Opportunities and Challenges for Novice 3D Modelers. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 157–166. https://doi.org/10.1109/VLHCC.2018.8506568

[5] Badhan Chandra Das, M. Hadi Amini, and Yanzhao Wu. 2025. Security and Privacy Challenges of Large Language Models: A Survey. *ACM Comput. Surv.* 57, 6, Article 152 (Feb. 2025), 39 pages. https://doi.org/10.1145/3712001

[6] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2, 1 (2023).

[7] gd3kr. 2023. BlenderGPT. https://github.com/gd3kr/BlenderGPT. GitHub repository, MIT license. Accessed 2025-12-04.

[8] Ricky Lam. 2015. Assessment as Learning: Examining a Cycle of Teaching, Learning, and Assessment of Writing in the Portfolio-Based Classroom. *Studies in Higher Education* 40, 11 (2015), 1900–1917. https://doi.org/10.1080/03075079.2014.999317

[9] Jihao Luo, Chenxu Zheng, Jiamin Yin, and Hock Hai Teo. 2025. Design and assessment of AI-based learning tools in higher education: a systematic review. *International Journal of Educational Technology in Higher Education* 22, 42 (2025). https://doi.org/10.1186/s41239-025-00532-7

[10] Andrew Price. 2026. Blender Donut Tutorial. https://www.youtube.com/watch?v=-tbSCMbJA6o. YouTube video, Blender Guru channel. Accessed 2025-12-04.

[11] D. P. Rohe and E. M. C. Jones. 2022. Generation of Synthetic Digital Image Correlation Images Using the Open-Source Blender Software. *Experimental Techniques* 46, 5 (2022), 615–631. https://doi.org/10.1007/s40799-021-00537-6

[12] Lav Soni, Amanpreet Kaur, and Avinash Sharma. 2023. A Review on Different Versions and Interfaces of Blender Software. In *2023 7th International Conference on Trends in Electronics and Informatics (ICOEI)*. 882–887. https://doi.org/10.1109/ICOEI56765.2023.10125672