A rigging Script Editor

Maya integrated IDE to support riggers

Robin van den Eerenbeemd

August 19, 2025

MSc Computer Animation and Visual Effects
NCCA - Bournemouth University

Acknowledgments

Thank You

Jon Macey

Jian Chang

NCCA Bournemouth

My fellow CAVE students

Special Thanks to:
My friends and family

1 Abstract

Auto-rigging or automatic rigging is an important technique in any pipeline that involves animation to keep rigs streamlined and modular. This paper explores auto-rigging through a specialized script editor, it provides a simple way to access files and organize them inside of Autodesk Maya, most riggers software of choice, which eliminates the need for an external IDE.

While this paper is aimed at rigging, the editor is versatile in usage and could be adapted or used without adaptation for any part of the pipeline that requires easy access to scripts, *e.g. modular modeling.*

Contents

Acknowledgments **2**

1 Abstract **3**

2 Introduction **6**
2.1 Objective 6

3 Related Work **7**
3.1 Modular Rigging 7
3.2 Maya Script Editor 8

4 Technical background **9**
4.1 Maya API 9
4.2 QT5 9
4.3 User interface guidelines 9

5 Implementation **10**
5.1 Overview 10
5.2 Interface design 11
5.3 Plugin design 13
5.3.1 Commands 13
5.3.2 Template 14
5.3.3 Script Editor 15
5.3.4 Navigation 17
5.3.5 Testing 18
5.3.6 Installation 19

6 Conclusion **20**
6.1 Appendix A 22

List of Figures

1 MGear in Maya [1] . . . 7
2 The Charcoal Editor 2 [2] . . . 8
3 The final UI of the *Bes Editor* . . . 10
4 The UI design for the *Bes Editor* . . . 11
5 Selected Command Produces Imports and Executable . . . 13
6 Hand Before . . . 14
7 Template for Hand . . . 14
8 Hand After . . . 14
9 Splitscreen . . . 15
10 Console Colours . . . 16
11 Dockable Script Editor . . . 16
12 Button Bar Explained . . . 17
13 The Mock Process [3] . . . 18
14 Newly created Bar after installation . . . 19
15 Installation Feedback . . . 19
16 New Command Window . . . 22
17 Edit Command . . . 22
18 Rename Command . . . 23
19 Save Main Script . . . 23
20 Load Template . . . 24
21 New Template . . . 24
22 Edit Template . . . 25
23 ButtonBar . . . 25
24 Layout with everything turned off . . . 25
25 Linux Bes Editor . . . 26
26 Windows Bes Editor . . . 26

2 Introduction

Rigging is an important step within any commercial, film or game studio that uses 3D animation. It is the bridge between model and animation, a model is provided with a set of bones, which drives the skin and muscle deformation. Control shapes or controllers, are used by an animator, to control the movement of the bones.

To lessen production time often auto-rigging tools, libraries or proprietary scripts are used for repetitive tasks. Alongside improved timelines, scripting allows for multiple fast iterations, which doesn't only benefit animators but modelers equally since often a model goes through multiple versions before the final one is approved. If a rig wasn't created with reiteration in mind, it would require a rigger to redo large parts of the set-up with each model variation.

This thesis describes a plugin for Autodesk Maya designed to make this process easier. It allows the user to create a library of functions and templates that are easily accessible and usable. The tool is flexible and would be simple to expand upon later by any user familiar with C++. By creating rigs fully through templates and/or functions the rig can be easily rebuild if a model is updated or reused for similar characters.

The plugin was developed using C++ and supports Python. It was developed for Maya 2023 but should work in any Maya version that supports QT5, which includes Maya 2017 to 2024.

This paper discusses related work, the technical background, and the set-up of the tool, with a detailed examination of its code structure and design choices, and concludes with a discussion of possible future developments.

2.1 Objective

The main objective of this tool is to provide an easy-to-use script editor that is tailored to modular pipelines, particularly for rigging, for beginner scriptwriters or riggers who are looking for an extra development tool. *Bes Editor* is designed with simplicity in mind, offering a straight-forward interface that includes features such as syntax highlighting and automatic module imports.

When a user selects a command to add to their script, the editor not only imports the corresponding module but also inserts the function name along with its variables and any descriptive comments saved in the user's custom library. In addition, users can save and reuse full templates such as biped or quadruped rig setups and leverage utility features like search-and-replace to help script writing.

These features make *Bes Editor* beginner-friendly while also serving as a valuable tool for experienced riggers. It supports fast prototyping and testing of functions or templates before integrating them into a larger production pipeline.

3 Related Work

3.1 Modular Rigging

Most studios, both large and small, often choose a modular approach to rigging due to the flexibility and re-usability. As discussed earlier, this methodology enables faster iterations and better adaptation to production changes. Consequently, the demand for modular rigging tools has increased, resulting in the development of numerous solutions some widely adopted, others proprietary.

While auto-rigging tools are available for various 3D software packages, this paper focuses exclusively on solutions developed for Autodesk Maya. The reason for this focus is that, despite growing diversity in software usage for rigging, Maya remains the industry standard across most studios in both games and film.

One widely used framework is mGear, which is an open-source rigging and animation solution. It was designed by mcsGear but by various artists and technical developers have contributed. Its goal is to offer a modular rigging system and automation tools [1].

Another popular tool that is geared more towards use with motion-capture is Mixamo by Adobe. It automatically creates a rigged full human skeleton that is compatible with their animation library. [4].

Softwares often have their own tools aswell, Maya for instance has the Quick Rig Tool which can be interesting for beginners or small productions but doesn't work well enough for advanced rigs like MGear does.

Often riggers, despite the wealth of tools available, make their own framework, both for the exercise it provides but mostly because it allows for a lot of control over the system and lets the creator integrate any new parts directly into the framework instead of having to built on top.

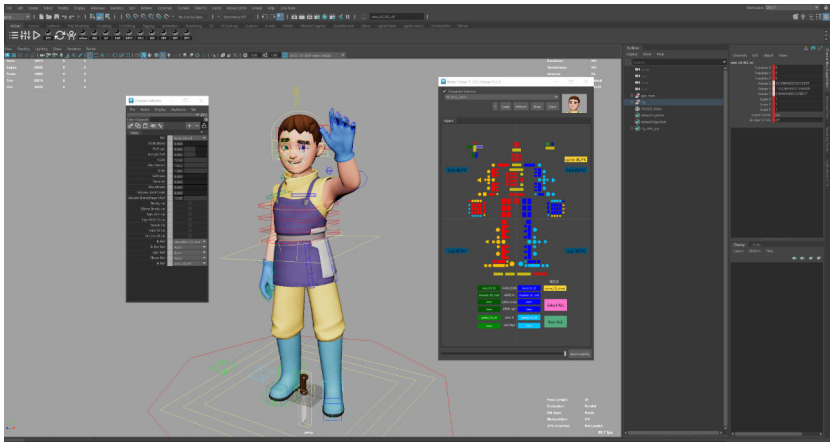

Figure 1: MGear in Maya [1]

3.2 Maya Script Editor

When scripting in Maya there are three options when it comes to the IDE (integrated development environment). There is Mayas internal script editor, an external IDE or a plugin. Using Mayas own script editor is most convenient since it doesn't require a user to set up anything before starting to script but it lacks a lot of features people look for in an IDE such as split-screen, easy file management etc.

External IDEs like Visual Studio Code or PyCharm offer such features, but they require additional setup to connect with Maya's environment, like configuring command ports. This setup process can be a barrier for less technical artists or teams working under tight deadlines.

As a middle ground, some developers go for plugins within Maya itself. These plugins aim to offer enhanced functionality directly inside the Maya interface, combining the convenience of the Script Editor with advanced features typically found in external IDEs.

The most prominent editor plugin on the market is Charcoal Editor 2 by Zurbrigg, which was created to meet the production needs of Maya technical directors and tool developers. It offers many features and removes the need for an external IDE [2].

This thesis describes an editor plugin that allows for a more user friendly experience than the Maya script editor, doesn't require complex external IDE setups and is open-source.

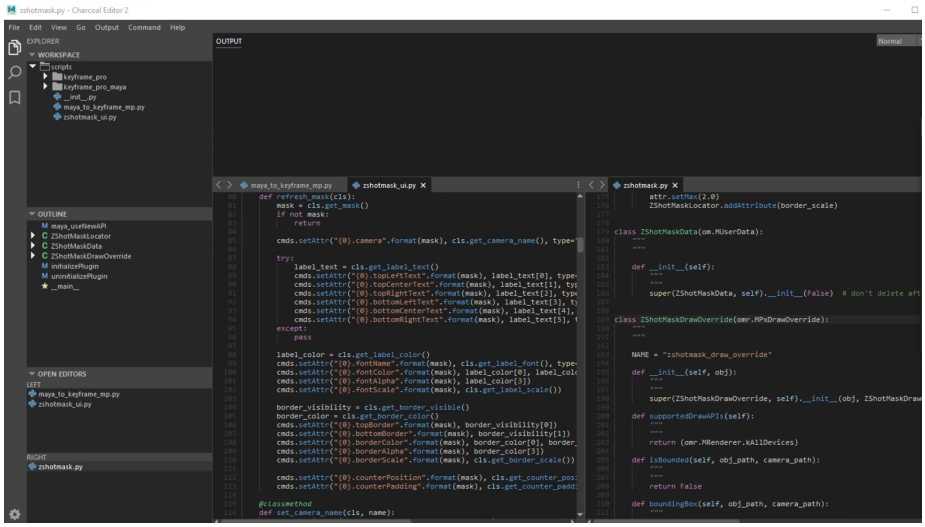

Figure 2: The Charcoal Editor 2 [2]

4 Technical background

The *Bes Editor* plugin was developed using the Maya API with a Qt5-based interface and supports scripting in Python.

4.1 Maya API

Maya API allows developers to create custom tools, nodes or commands that integrate directly into the Maya environment by using the internal libraries of Maya. The best way to utilize the Maya API is through the Maya devkit which contains C++, python and .NET APIs [5] The Maya API defines four types of C++ objects: *wrappers, objects, function sets, and proxies.* Wrappers are convenience classes, including iterators. Objects serve as the base class for all Maya entities, such as curves, DAG nodes, and IK solvers. Functions sets contain functions that operate a specific type of node and proxies are abstract classes that can be used to implement news types of Maya objects *e.g. custom nodes.* [6]

4.2 QT5

QT is a cross-platform application development framework to simplify UI development in multiply languages, but the framework itself is written in C++. A lot of applications use QT for its UI including Autodesk Maya, Developers can use Mayas cross-platform toolkit to customize Maya's user interface with QT. QT comes with an IDE called QT Creator for easy development, even though it's not per se necessary [7].

The user interface for this thesis was built using QT creator because it allows a simple way to create dock-able panels, custom widgets and event-driven UI elements as a stand-alone application before integration it into a Maya environment.

4.3 User interface guidelines

User interface design should be simple and goal-driven, minimizing complexity and reducing the time users spend searching for functions or information. Consistency in layout, language, and iconography helps establish reliable mental models; therefore, interfaces should prioritise clarity in these elements and provide contextual help when needed. The interface should feel intuitive to its target audience. Flexibility and efficiency are also important, as different users may employ the same tools for distinct purposes and may require adaptable layouts to optimise their workflow. When errors occur, the program should offer clear, actionable feedback to guide the user forward and minimise frustration [8].

The user interface developed for this thesis was designed with these principles in mind. Throughout the following chapters, design choices will be described and, where appropriate, justified in relation to these guidelines.

5 Implementation

5.1 Overview

Bes Editor, composes of two primary components, a Qt user interface and a Maya integration layer that handles interaction with the Maya environment. The UI manages all user-facing elements, such as layout, text editing, and control buttons. It is implemented using Qt5 in C++, with C++ functions directly connected to the interface components via Qt's signal and slot mechanism.
The Maya integration layer provides back-end functionality, including the execution of Python scripts and plugin management. This layer communicates with the Maya environment through the Maya API, allowing *Bes Editor* to interact with Maya through for instance feedback in the terminal and running commands.
Together these components form a cohesive, user-friendly plugin that gives users easy access to their scripts.

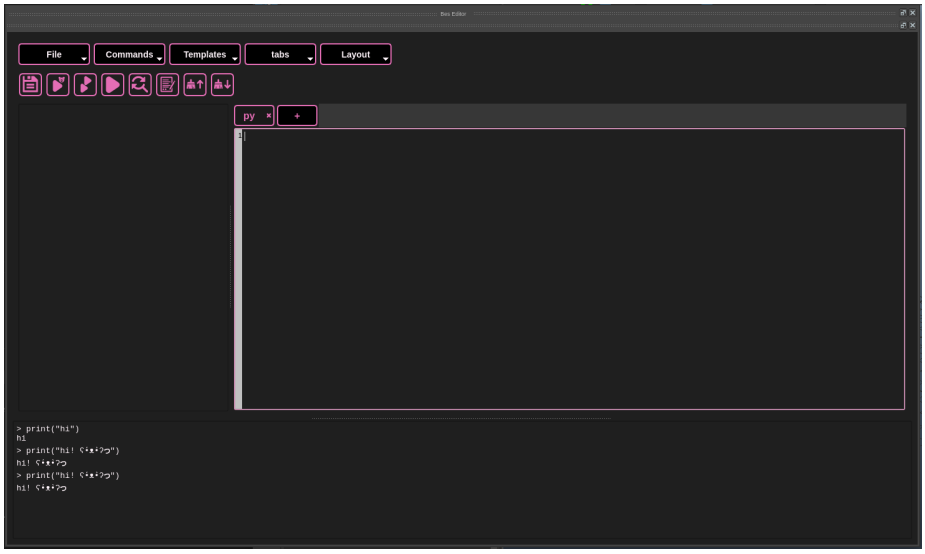

Figure 3: The final UI of the *Bes Editor*

5.2 Interface design

The user interface was created using Qt and mirrors elements from the Maya Script Editor that are user-friendly, such as the tabbed layout and pictograph-style button bar. Other components such as the overloaded terminal and the non-intuitive search-and-replace functionality were redesigned for improved usability. In addition to replacing these less accessible features, a custom file management system was introduced to help file functions and templates, see Figure 4.

Certain elements of the interface deviate from the conventions typically found in script IDEs, such as the placement of the file editing menus and the button bar. In this design, the button bar is positioned directly above the script editor rather than above the console (code output). This decision was made to reduce the distance the user must move their mouse between editing actions. Since the button bar and menus are primarily used during code editing, positioning them closer to the editor is expected to improve efficiency and reduce unnecessary cursor movement.

The UI can be adjusted through the layout menu, where several UI elements can be hidden and the console and editor can be switched. When they are switched the buttonbar icons of both will swap so the arrows keep pointing in the right direction as to not confuse the user.

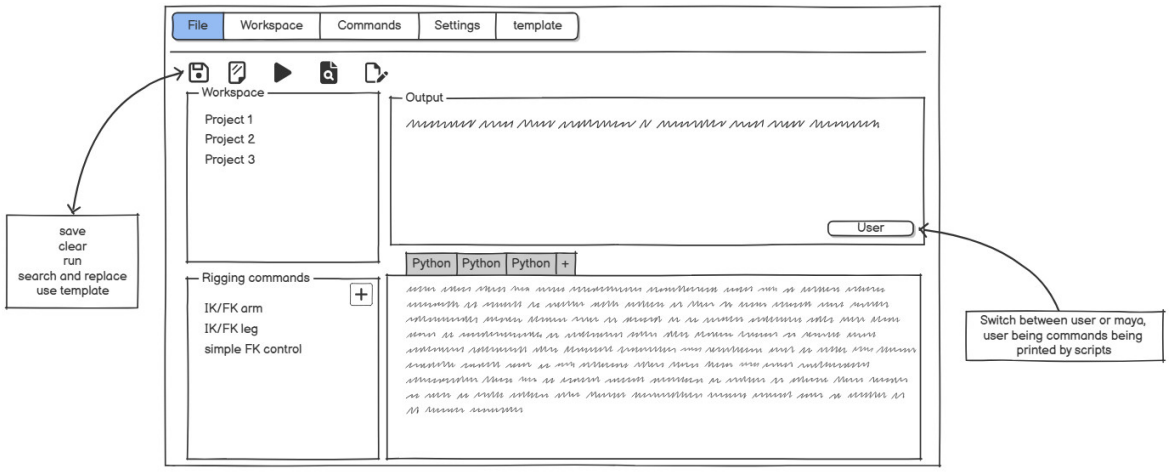

Figure 4: The UI design for the *Bes Editor*

To ensure modularity and ease of maintenance, the UI components are organized into separate classes. This structure promotes a clean and extensible code-base. All components are assembled in the `ScriptEditorPanel` class, which manages the overall layout and connects key menu elements to their functions. Multiple of the components are laid out using a .UI file, which use an XML format to represent form elements/ the widget tree and are exclusively used in QT. The Bes editor mostly uses a single inheritance approach where standard QT widgets are subclassed and used to set up the user interface. A standard system is then used for making signal and slot connections [9]. The visuals are adjusted with Style.h which contains all the style-sheets.

The workspace shown in the original design (Figure 4) was not implemented due to time constraints. It was considered the least critical of the main widgets, as the command and template systems already provide accessible means for the user to manage their files.

5.3 Plugin design

The plugin was created using the Maya API, it's a simple design where extensibility was the most important.

5.3.1 Commands

The commands system is built up out of 3 classes: CommandList, EditCommand and NewCommand, each class manages a separate widget with a distinct purpose, collectively maintaining the QList which contains all the commands the user added.

CommandList plays the most central role, as it owns the QList widget and is responsible for populating it. The population process involves looking through the user-added files and adding any python files identified to the list. When the user interacts with a list item, a signal is emitted. The corresponding slot compiles the necessary information to execute the selected function; *the required import statements, the function with associated variables and user-defined information*, and writes this content to the editor. As seen in Figure 5. In addition, CommandList provides functions that don't require a separate interface such as rename and remove.

EditCommand and NewCommand operate in a similar way, each open a QDialog containing a QPlainTextEdit. NewCommand enables the user to save a new command to the previously established list, it safeguards that the user isn't overwriting any existing files and has entered a valid name through dialog.

Commands can consist of either individual functions or classes, if a class is being saved, the user is given a place too specify the function which they would like to be inserted in the editor when selected from the list. EditCommand re-opens an existing file of choice, allowing the user to modify and save the contents, overwriting the previous file.

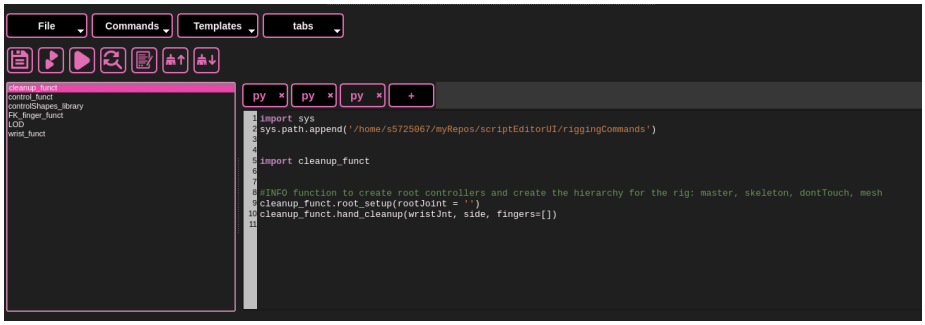

Figure 5: Selected Command Produces Imports and Executable

5.3.2 Template

The template system programmatically works in a very similar way as the command system, the main exception is that the list to call templates from has a separate UI. When templates are loaded, the entire text contained in the template file will be pasted into the editor. There are two applications for this, first being what it is intended for, the user can use the command list or other functions to create a template for any type of rig *e.g. a biped rig*. Secondly it can be used as a simple way to safely import and update work in progress on functions or classes before adding them to the main pipeline.

To illustrate a practical use case of a template, Figure 6 presents a hand model with its underlying skeleton prior to the application of a hand template. Applying the template, as shown in Figure 7, produces the rig depicted in Figure 8. The template itself, also displayed in Figure 7, is composed of functions listed in the saved command list of the same figure. Once executed, the template generates the complete rig in approximately two seconds, requiring no additional user input.

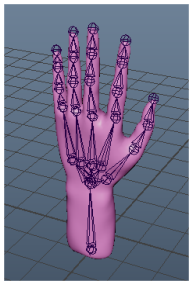

Figure 6: Hand Before

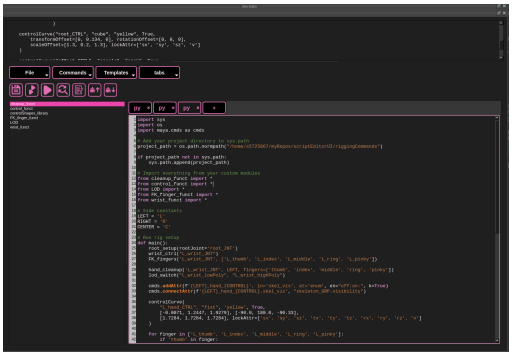

Figure 7: Template for Hand

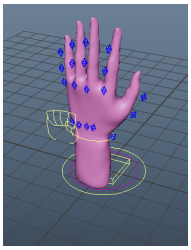

Figure 8: Hand After

5.3.3 Script Editor

The script editor is composed of 4 elements: the editor, the console, the highlighter and the line-numbers. The parent widget to the editor is a QPlainTextEdit, the highlighter and line-numbers are tied into this widget to elevate it from a regular text editor to one that can comfortably be used to program.

The line-number display is created using a QPainter object and is updated through the code editor class. Whenever a new line is added to the editor, the display is refreshed to include the corresponding line number. By visualizing line numbers it is easier for the user to keep track of their code and traceback errors, improving quality of life.

The syntax highlighter applies a predefined set of rules to enhance code readability through color-based syntax highlighting. Each rule associates a specific color to their function, following a scheme inspired by Visual Studio. For example, Python keywords *e.g., for, if, class* are displayed in pink, user-defined commands are displayed in green, and Maya internal commands (cmds) are displayed in orange.

The script editor integrates syntax highlighting and line-numbers functionality into each editor instance. Since the user can create multiple tabs and splitscreens, multiple instances can exist simultaneously. The management of tabs and split-screen layouts is handled by the TabScriptEditor class, while the ScriptEditor class keeps track of the currently active editor. This distinction ensures that other functions, such as opening a script or clearing the editor, operate on the correct code editor instance.

Figure 9: Splitscreen

The console constitutes the final component of the script editor. In addition to creating the console interface, the corresponding class is responsible for executing code. The console is implemented as a QPlainTextEdit widget, which displays feedback generated by the executed code. Error messages are shown in red, execution results in green, and other output in gray. Code execution is performed through a Python snippet that captures Maya's output by redirecting its stdout and stderr streams, which store the application's standard results and error messages.
The original intention was to implement this functionality in C++ using the Maya API, specifically the stdOutStream and stdErrorStream from the MStreamUtils class. However, as these streams are only available in Maya 2024 and later, this approach was not feasible for the current implementation. Instead, the functionality was rerouted through the Python snippet to achieve equivalent output capture. In future iterations, the project would benefit from being rewritten to target Maya 2024 and later.
Code can be run three different ways, the user can run full script, partial script or main script. Full script works in the way previously described for the whole script, partial script only runs on the selected code. Main script which is saved through the command menu will run a previously saved script that the user does not have to have open in their current editor.

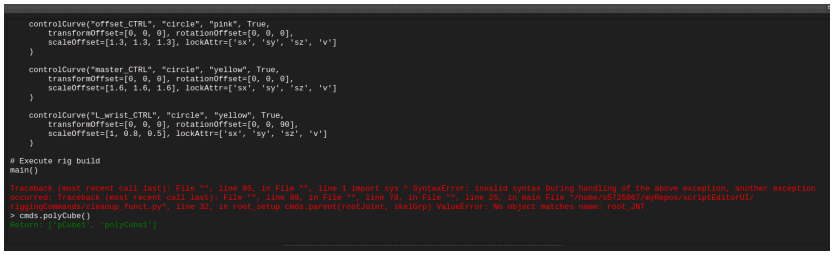

Figure 10: Console Colours

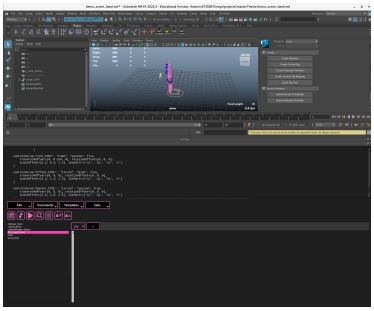

Figure 11: Dockable Script Editor

5.3.4 Navigation

The text within the editor can be navigated and managed through both the file menu and the button bar. The file menu provides access to operations such as *New Script, Open Script, Save Script As*, and *Exit*. These options allow the user to employ the operating system's file management system to load and save .py files into the active script editor. Selecting Exit closes the entire Bes Editor application.

The button bar offers quick-access links to several functions also available in the menus, see Figure 12, such as Load Template, as well as to features exclusive to the button bar. These include Search and Search and Replace, which enable the user to locate specific words within the active editor, with options for case-sensitive or case-insensitive matching. The button bar additionally provides controls to clear either the terminal or the editor contents.

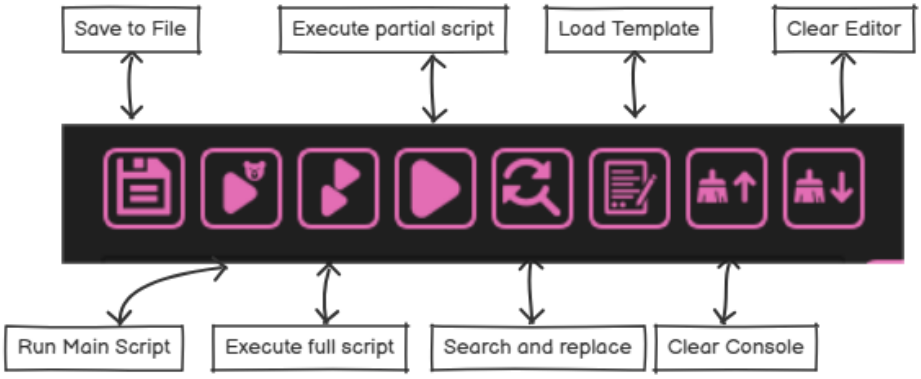

Figure 12: Button Bar Explained

5.3.5 Testing

Testing is an important part of software development, it helps catch bugs early and ensures code continues to run smoothly after any changes are made without requiring manual verification. For Maya Plugins testing is certainly possible but comes with added difficulty. Different kinds of testing could be applied, such as *unit tests, UI tests and system tests.*

In this project, only a minimal set of unit tests were implemented using GoogleTest. These tests check that the plugin can be loaded and unloaded within Maya, and that its user interface can be invoked without errors. This provides a basic safety net: if the plugin fails to initialize or register correctly, the tests will immediately catch it.

Extending testing further is challenging for several reasons, the classes would need to have their functions exposed or use friend classes [10] that can access the private members which is a practice heavily debated by software engineers [11] as a bad approach.

UI testing introduces another layer of difficulty. While Qt offers a testing framework to simulate user interaction, this would either have to run the UI in a headless Maya environment or require implementing a new main window class so the code could be executed standalone. After attempting to integrate both approaches, the decision was made that, at the time, manually testing after important code changes was more time-efficient than pursuing this method for fuller test coverage.

Another effective approach for unit testing a UI-heavy project like this is mocking. Mocking involves creating simplified, stand-alone versions of units. Any external dependencies, such as APIs or backend components, are replaced by mock objects, allowing the UI to be tested independently. This creates an isolated environment where the behavior of individual components can be tested more reliably [3].

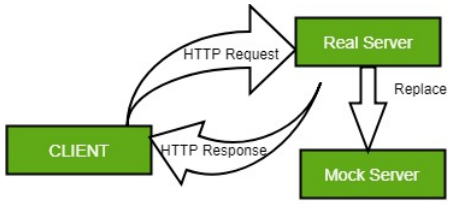

Figure 13: The Mock Process [3]

5.3.6 Installation

Installation is performed via a drag-and-drop Python script, allowing the user to install the plugin by simply dragging the script file into the Maya script execution field. Upon execution, the script automatically installs all components required for the plugin to function in their correct locations. Specifically, it places the .so file containing the compiled plugin into Maya's plugin directory, which unless modified by the user's environment configuration, is the default location Maya searches for plugins. In addition, the script creates the necessary folder structure for the plugin and adds a new menu bar entry within Maya containing a button to launch the editor. There is a compiled version for both Linux and Windows so it will install on either of those operating systems.

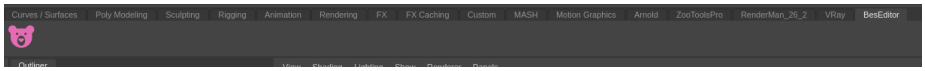

Figure 14: Newly created Bar after installation

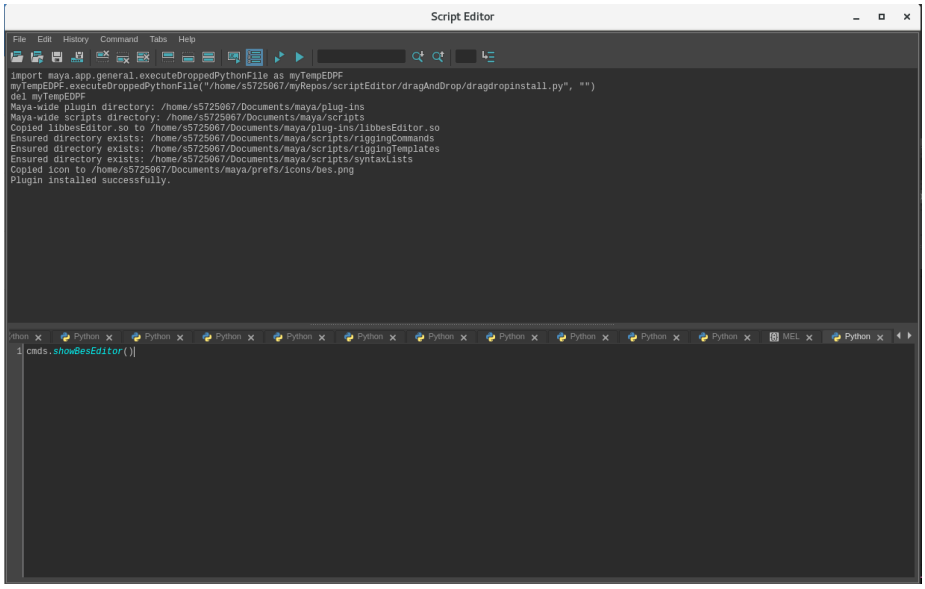

Figure 15: Installation Feedback

6 Conclusion

The implementation of a fully functional UI for the *Bes Editor* establishes a solid and extensible foundation that can be expanded with additional functionality in the future. The current implementation includes all core features; however, several enhancements could further improve its capabilities, such as:

- **Workspace window** , see Figure 4
- **Smart indenting**
- **Auto completion**
- **Version control** , either internal or a link with git
- **Testing** , better testing coverage

The system was tested with Maya 2023 and Qt5. Future development could benefit from migrating to Qt6, which would ensure compatibility with Maya 2024 and later versions, providing access to new Maya API functionality introduced in Maya 2024.

Furthermore, the scope of the project could be expanded beyond Maya. As most of the core functionality relies on Qt and C++ without requiring the Maya API, it could be refactored for integration into other software applications that would benefit from a script editor following this approach, such as Nuke or Blender.

To use my first "I" in this paper, I would like to emphasize that alongside the practical advantages of C++ and its genuine benefits as a language, a major reason for choosing it was my personal desire to improve my skills. I believe I achieved that goal, even though there is still much more to learn, which I will keep working on in the future.

References

[1] J. Passerin, "Mgear framework." https://mgear-framework.com/, 2025. Accessed: 2025-08-14.

[2] C. Zurbrigg, "Charcoal editor 2, maya scripting evolved." https://zurbrigg.com/charcoal-editor-2, 2025. Accessed: 2025-08-14.

[3] GeeksForGeeks, "Mock (introduction) - software engineering." https://www.geeksforgeeks.org/software-engineering/software-engineering-mock-introduction/, 2024. Accessed: 2025-08-19.

[4] Mixamo, "Adobe mixamo." https://www.mixamo.com/#/, 2025. Accessed: 2025-08-14.

[5] M. Autodesk, "The maya api." https://help.autodesk.com/view/MAYAUL/2024/ENU/?guid=Maya_SDK_Maya_API_introduction_API_Basics_html, 2025. Accessed: 2025-08-14.

[6] C. Vernon, "Maya api programming." https://www.chadvernon.com/maya-api-programming/#introduction, 2012. Accessed: 2025-08-14.

[7] Q. Documentation, "Code editor example." https://stuff.mit.edu/afs/athena/software/texmaker_v5.0.2/qt57/doc/qtwidgets/qtwidgets-widgets-codeeditor-example.html, 2016. Accessed: 2025-08-14.

[8] E. Wong, "User interface design guidelines: 10 rules of thumb." https://www.interaction-design.org/literature/article/user-interface-design-guidelines-10-rules-of-thumb?srsltid=AfmBOoqFhng-WGJ_m7-b23EHcs2zeZdBSVkX1aDqLTKM6CL9kWMc2WkN, 2025. Accessed: 2025-08-14.

[9] A. Margaryan, "Signals and slots in qt." https://medium.com/@arsenmargaryan38/signals-and-slots-in-qt-4496792d409b, 2024. Accessed: 2025-08-14.

[10] GeeksForGeeks, "Friend class and function in c++." https://www.geeksforgeeks.org/cpp/friend-class-function-cpp/, 2025. Accessed: 2025-08-19.

[11] Akavall, "Unit test private method in c++ using a friend class." https://softwareengineering.stackexchange.com/questions/257705/unit-test-private-method-in-c-using-a-friend-class, 2015. Accessed: 2025-08-19.

6.1 Appendix A

Interface elements

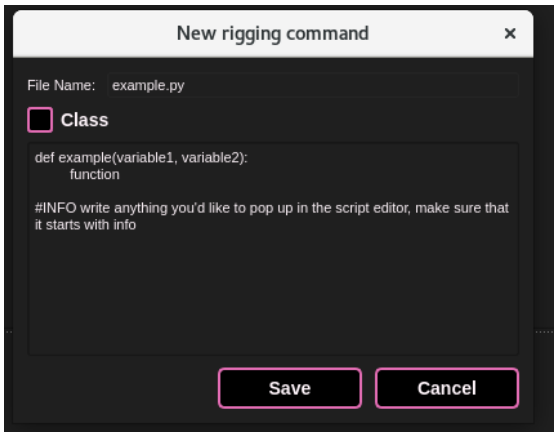

Figure 16: New Command Window

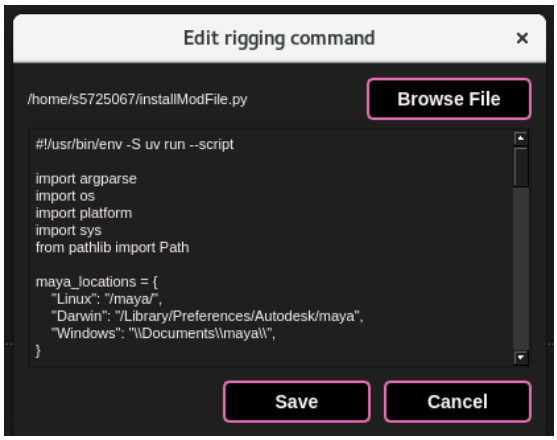

Figure 17: Edit Command

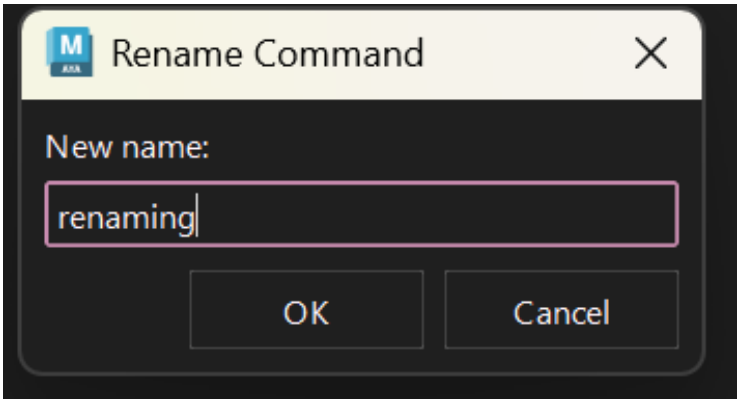

Figure 18: Rename Command

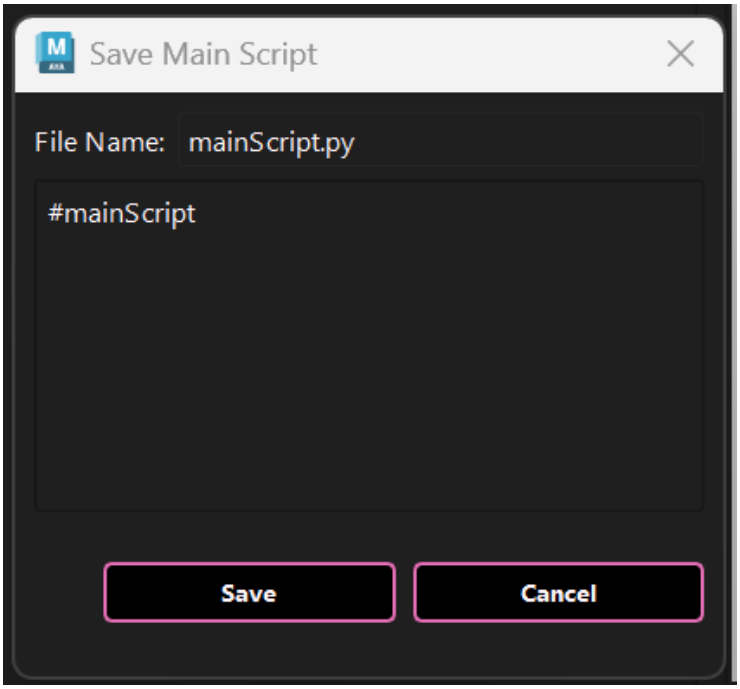

Figure 19: Save Main Script

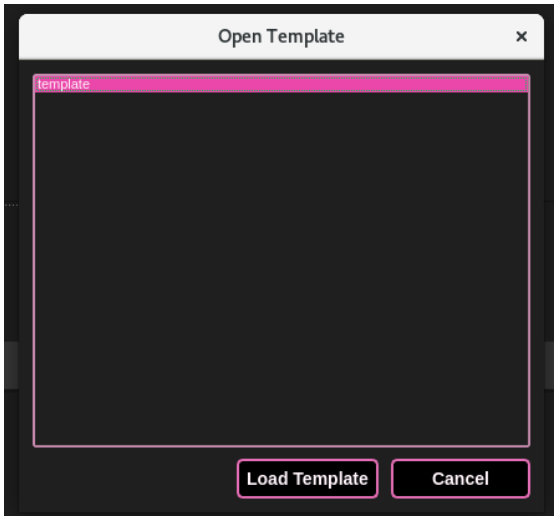

Figure 20: Load Template

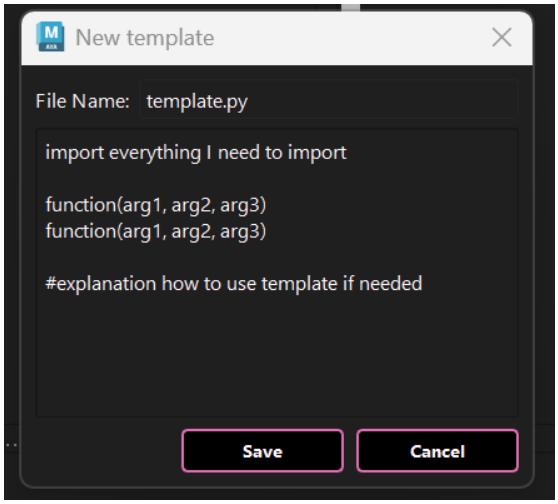

Figure 21: New Template

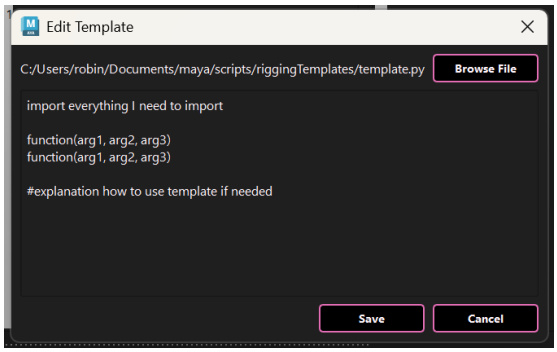

Figure 22: Edit Template

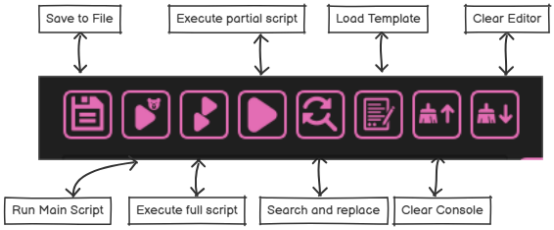

Figure 23: ButtonBar

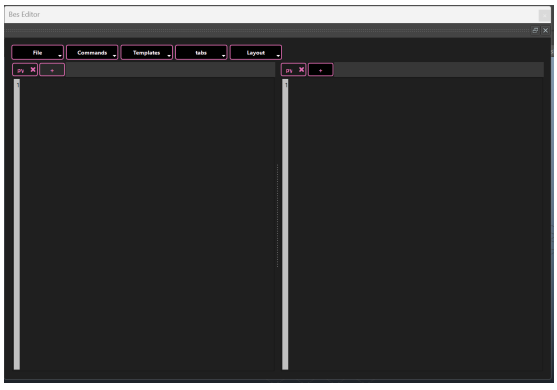

Figure 24: Layout with everything turned off

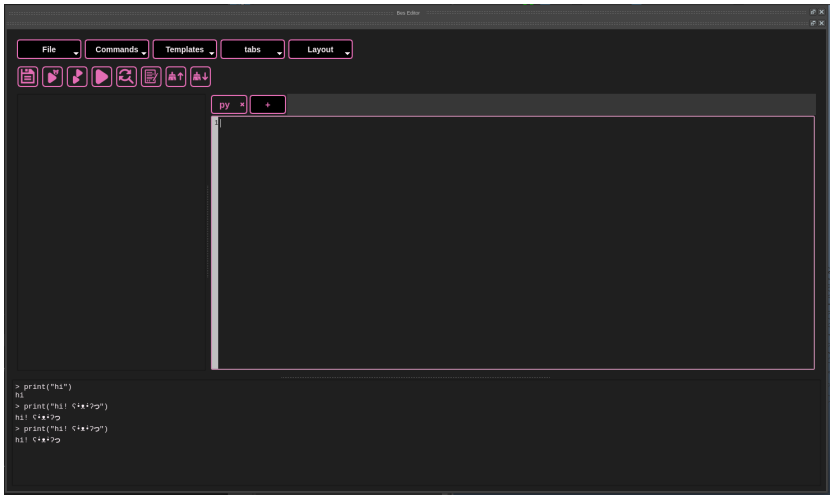

Figure 25: Linux Bes Editor

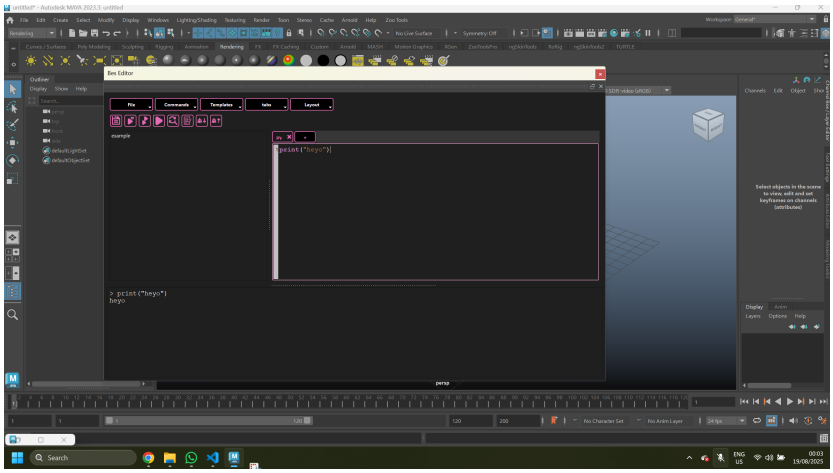

Figure 26: Windows Bes Editor