



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

BLENDER ADD-ON FOR CONVERSION OF NODES TO VECTOR GRAPHICS

BLENDER ADD-ON PRO PŘEVOD UZLŮ DO VEKTOROVÉ GRAFIKY

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR

AUTOR PRÁCE

Bc. FILIP DRÁBER

SUPERVISOR

VEDOUCÍ PRÁCE

Ing. TOMÁŠ CHLUBNA

BRNO 2024

Master's Thesis Assignment



153734

Institut: Department of Computer Graphics and Multimedia (DCGM)
Student: **Dráber Filip, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Computer Graphics and Interaction
Title: **Blender Add-on For Conversion of Nodes to Vector Graphics**
Category: Computer Graphics
Academic year: 2023/24

Assignment:

1. Learn about the basic workflow in 3D modelling program Blender (version 3.5 and above).
2. Learn about the scripting Blender Python API and the specification of SVG vector format.
3. Propose an add-on which will extend Blender by conversion of nodes to SVG format.
4. Study the possible and existing solutions.
5. Propose the user interface for the add-on.
6. Implement the add-on and demonstrate its functionality with various data.
7. Document and publish the add-on.
8. Create a video representing the results.

Literature:

- [Blender API documentation](#)
- Dengler, Patrick, et al. "Scalable vector graphics (SVG) 1.1." (2011).W3C

Requirements for the semestral defence:
Items 1-5, experiments leading to item 6.

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **Chlubna Tomáš, Ing.**
Head of Department: Černocký Jan, prof. Dr. Ing.
Beginning of work: 1.11.2023
Submission deadline: 17.5.2024
Approval date: 9.11.2023

Abstract

This project proposes and implements a Blender add-on which lets the user export on-demand vector representations of the Node graphs they have created within the editor. The technical report delves into the technical aspects of both Blender's internal representation as well as features of the Scalable Vector Graphics format (SVG) which are used to produce the vector representation. It showcases how the add-on describes various Node types and streamlines their representation. Finally, it evaluates the usability and performance of the add-on, which has already been used in an academic environment at time of writing.

Abstrakt

Tento projekt navrhuje a implementuje rozšíření pro aplikaci Blender, které umožňuje uživateli exportovat vektorovou reprezentaci uzlových grafů, které v aplikaci vytvořili. Tato technická zpráva se zabývá technickými aspekty interní reprezentace Blenderu a také různými vlastnostmi formátu SVG, který je použit pro vytvoření vektorové reprezentace. Zpráva ukazuje, jak rozšíření popisuje různé druhy uzlů a sjednocuje jejich reprezentaci. Nakonec posuzuje použitelnost a funkčnost tohoto rozšíření, které již bylo použito v akademickém prostředí.

Keywords

Blender, Node editor, add-on, vector graphics, SVG, Python, academic writing

Klíčová slova

Blender, editor uzlů, rozšíření, vektorová grafika, SVG, Python, akademické psaní

Reference

DRÁBER, Filip. *Blender Add-on For Conversion of Nodes to Vector Graphics*. Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Chlubna

Rozšířený abstrakt

Blender je mnohoúčelový open-source nástroj na téměř jakoukoli práci s 3D modely a vizualizací. Některé z jeho pracovních prostředí využívají tzv. uzlových grafů (Node graphs), které jsou příkladem vizuálního programování. Koncoví uživatelé tohoto programu mají možnost skrz ně vytvářet procedury, které upravují vzhled scény, na které zrovna pracují, a jejího obsahu.

Pokud chce uživatel dokumentovat takový uzlový graf, může buď vytvořit snímek obrázky, nebo ve svém oblíbeném programu na tvorbu vektorových obrázků vzhled výsledného grafu překreslit. První varianta není žádoucí, protože vytváří nepřesnou, rastrovou reprezentaci grafu, a ta druhá je nadmíru pracná. V ideálním případě by už v rámci samotného editoru měl možnost jednoduše vytvořit vektorový snímek svého grafu.

Blender však dává uživateli možnost si nainstalovat různá rozšíření, která vznikají v rámci jeho početné komunity. V rámci této práce je navržené a implementované právě takové rozšíření, které řeší problém s dokumentací uzlových grafů.

V rámci této technické zprávy je nejdřív popsána aplikace Blender, ale hlavně její aplikační rozhraní, se kterým interagují skripty v jazyce Python. Pozornost je věnovaná také samotným uzlům a uzlovým grafům a jejich reprezentaci. Dále je zde také popsán vektorový formát obrázku SVG, do kterého rozšíření exportuje uživatelův uzlový graf.

Ve fázi návrhu je předložena metoda, kterou rozšíření transformuje vnitřní reprezentaci dat Blenderu do vektorové reprezentace uživatelského uzlového grafu. Je také vytvořen mock-up uživatelského rozhraní takového rozšíření. V rámci této části je také popsáno, jak byl proveden výzkum mezi potenciálními uživateli takového rozšíření – uživateli aplikace, a autory technických zpráv. Tento výzkum se dotazoval na uživatelskou zkušenost s aplikací Blender a také na jeho případy využití této aplikace.

Kapitola implementace se věnuje zejména dvěma hlavním řešením, která byla v rámci práce implementována. Tím prvním je vytvoření tříd typu Widget, které nabízejí jednotnou formu reprezentace různých elementů uživatelského rozhraní aplikace Blender ve formátu SVG. Součástí tohoto řešení jsou i třídy, které do sebe vnořují jiné instance tříd typu Widget, a zvyšují tak formátovací a vyjadřovací schopnost tohoto řešení. Druhým stěžejním bodem implementace je vytvoření slovníku, který obsahuje záznam pro každý typ uzlu, který lze v dosavadní verzi aplikace Blender najít a použít. V rámci slovníku je definován vzhled uzlu a různá chování, která nelze získat z vnitřní reprezentace aplikace.

Následující kapitola obsahuje zhodnocení rozšíření, a to z hlediska vizuálního, uživatelského i technického. V rámci této kapitoly je vypsáno několik markantních rozdílů mezi vzhledem uzlů v rámci aplikace Blender a jejich reprezentací ve výsledném vektorovém souboru. Je také proveden a zdokumentován test použitelnosti rozšíření. V rámci něj je zkoumáno, jak uživatelé pracují s rozšířením a jestli rozumí, co jim rozšíření vlastně nabízí. Je navíc testováno, jestli jsou obrázky vygenerované tímto rozšířením dostatečně užitečné k tomu, aby mohli uživatelé zrekonstruovat uzlový graf, ze kterého byly pořízeny. Z technického hlediska je rozšíření posuzováno na základě velikosti samotných vektorových výstupů, ale také času, který rozšíření stráví produkováním těchto výstupů. Je zkoumán dopad complexity uzlů, velikostí grafů i uživatelských nastavení tohoto rozšíření na zmíněné veličiny.

Rozšíření je nakonec zdokumentováno videem, a nasazeno na on-line platformy ke sdílení podobných rozšíření pro aplikaci Blender pro další použití v akademické i profesionální sféře.

Blender Add-on For Conversion of Nodes to Vector Graphics

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Tomáš Chlubna. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Filip Dráber
May 16, 2024

Acknowledgements

I wish to thank my supervisor for his patience, willingness, optimism and reading speed shown off throughout the preparation of this thesis. I also wish to thank my friends who helped take my mind off the thesis during downtime even when they themselves were preparing their own theses at the time. Finally, I wish to thank all the willing testers of the add-on, be it other students, academics, or artists who shall remain unnamed as to not tarnish the professionalism of this thesis.

Contents

1	Introduction	2
2	Theoretical Background	3
2.1	Blender	3
2.2	Scalable Vector Graphics	11
3	Proposal	23
3.1	Add-on Workflow	23
3.2	User Interface	24
4	Implementation	28
4.1	Widgets	29
4.2	Node Specifications	34
4.3	Export Properties	37
4.4	Converter Initialization	38
4.5	SVG Output	42
5	Evaluation	49
5.1	Research Survey	49
5.2	Quality Evaluation	50
5.3	Measurements	55
5.4	Future Work	61
6	Conclusion	62
	Bibliography	63
A	Survey	65
B	User Testing	67
C	Poster	69
D	SD Card Content	70

Chapter 1

Introduction

Visual programming is a popular paradigm that is taking root in areas such as smart home technologies, game engines, and other various applications. End users of these technologies can design and extend scripts using visual elements such as blocks and wires that connect them [9]. They may wish to document their solutions, especially in an academic or professional context, but do not have any correct option as to how to depict them. The user may simply take a screenshot of the application, which is quick, but pollutes the image with unnecessary information from the editor’s user interface [10, p. 214]. The result image is also in the raster format, which is considered bad practice when depicting abstract content. Only one option is left for the author: open a vector graphics editor such as Inkscape and painstakingly remake the content, rectangle by rectangle.

This project solves the outlined problem for Blender, which is an open-source multipurpose 3D editing software. Blender utilizes visual programming in various areas contained within the application, in the form of Nodes. The openness of Blender allows members of the community to design their own extensions (called add-ons). This project implements an add-on which receives an internal representation of Blender’s runtime on input, and transforms these data into a vector representation of the currently-edited Node tree at the click of a button, saving the user’s time and dignity.

Chapter 2 of this technical report details the two principal technologies that this project abridges: the Blender application itself and the Scalable Vector Graphics (SVG) image format. For the former, it focuses on relevant features and aspects of the software’s Node editor that affect the visual appearance of the output. For the latter, it describes its structure and documents elements and technologies used to accurately depict the Node graphs.

Chapter 3 then describes the proposed design of the application, from its functionality to its user interface. It also describes the initial user research that led to expanding the proposed design by adding multiple requested features.

Chapter 4 delves into the implementation of the add-on itself. It documents various solutions devised to get around limitations of the provided data, such as a mechanism to account for Node-specific differences during the construction of the Nodes’ representation, or a class that streamlines defining the structure and appearance of various elements of the user interface.

Finally, Chapter 5 accounts for the differences between the Nodes’ appearance and the output and showcases various metrics measured to assess the add-on’s performance. It also includes a final round of user feedback focused on the usability of the add-on, and the informative quality of the vector image on output.

Chapter 2

Theoretical Background

The foundation of this thesis comprises of two major technologies: the Blender application and the SVG vector image format. In this chapter, elements of both that are relevant for this thesis are described separately. Combining both for the purposes of this thesis is described in further detail in Chapter 3.

2.1 Blender

Blender is a computer-aided design tool, primarily for 3D design and modeling, which started as proprietary software, but was eventually acquired by the community after a seven-week fundraising effort. It is used to render high-fidelity still images and it also supports animation, architecture, video game development, scientific visualization, and industrial design [6, p. 10-11]. Recently, Blender has also found use in the academic sphere as a tool to create professional 3D models and visualize data [7].

Blender is written in C++, and its persistent data are defined by structures referred to as *DNA*, which features a meta-description of the structure’s layout and thus enables serialization and a degree of backwards compatibility (as described in Blender’s own developer documentation¹). Compliant to the biological analogy, Blender also implements *RNA* – an interface to the DNA structures, accessible during runtime by the application, user interface, and Blender’s built-in Python interpreter alike.

The RNA technology is made up of Struct objects that represent data and which can contain any number of attached Properties. These Properties come in the form of multiple types, mostly reflecting typical data types present in programming such as integers, strings, or float values. However, Property types exist to represent more complex data structures such as enumerables or vectors. Most of the interface is not defined explicitly, but is generated during the preprocessing stage when the application is built. Relations between data blocks represented by Struct objects are managed by RNA Pointer Properties.

2.1.1 Add-ons

Part of what makes Blender appealing to general use is its openness to external add-ons. Blender’s add-ons can be categorized under two labels, Official and Community. A third type, Testing, was at some point used, but has since been deprecated.

¹<https://developer.blender.org/docs/features/core/dna/>

Anyone can share add-on scripts for others to install. For an add-on to be accepted under the Official label, it must pass verification which requires the author to properly document and, most importantly, maintain their script in the long term, vowing to respond to bug reports and keeping up with Blender’s own release cycle².

The add-ons are written in the Python 3 scripting language. Blender comes with its own Python interpreter to execute these scripts and perform other operations (such as rendering the user interface) [2, API Overview].

2.1.2 The bpy Module

Within its own interpreter, Blender exposes the `bpy` module. This module is an API which allows for various interactions with Blender’s runtime; namely, the RNA structures. The module consists of several submodules:

- Context (`context`): Properties related to the area currently accessed by the user.
- Data Access (`data`): Access to Blender’s internal data (the current blend-file).
- Message Bus (`msgbus`): Framework for notifications of changes to the data blocks.
- Operators (`ops`): Tools accessible through various UI elements, and also invoked from within the app’s runtime.
- Types (`types`): Definitions of all data types.
- Utilities (`utils`): Utility functions for Blender, including preview-related and unit-related functions.
- Path Utilities (`path`): Utility functions for handling paths, similar to Python’s own `os.path` module.
- Application Data (`app`): Application values unchanged during runtime.
- Property Definitions (`props`): Definitions of Properties that extend Blender’s internal data.

The Context Submodule

Arguably the most important submodule for this project is the `context` submodule. Its content is always related to the current active area that the user is focusing on and is also passed to Operators (functions in the user interface) by Blender itself.

Some of the `context` submodule properties are available all the time, such as the current rendering engine, Preferences, Scene object or a reference to the active Area. Others are exposed or hidden, depending on the type of editor that the focus is currently on. For example, the `selected_nodes` property can only be accessed if the user is currently working in a Node Editor (which would be the case when this project’s add-on’s main functionality is invoked).

User preferences are also accessible through this submodule. These allow for users to configure various aspects of their editor and, most importantly for this project, also let them adjust their Theme: a collection of appearance and color settings applied to the entire user interface, including the Nodes themselves. Accessing the `context.preferences` property during export enables the SVG colors to match those of their editor.

²<https://developer.blender.org/docs/handbook/addons/>

Installing an Add-on

To enable a custom add-on in Blender (for development, for example), the add-on source code needs to be included in Blender’s `scripts/addons` sub-directory (found in Blender’s application data directory³). Blender can be presented with the source code in two ways: either as a single, standalone Python script, or as a Python module – a directory containing an `__init__.py` file, as well as all the other source files.

When loading scripts, Blender runs every script in the `scripts/addons` subdirectory, as well as every `__init__.py` of each module. Importantly, when using Blender’s *Reload Scripts* operation, the application only reloads the same files it launches as described above. For this reason, it is sometimes necessary to manually add functionality that forces a reload of the other source files in the module, which may be necessary, for example, during an iterative development process.

Registering an Add-on

A Python script must satisfy several qualifiers to be recognized as an add-on. First, the script needs to begin by defining a variable named `bl_info`, which contains meta information about the add-on itself, including the author’s name, add-on name, description, location within the application, type, and support level, among others⁴. Additionally, the add-on must implement a `register` and `unregister` method. Within these, the `register_class` and `unregister_class` methods (from the `bpy.utils` module) must be called to actually enable elements⁵ of the add-on within the application.

User Interface

An add-on can be included in Blender’s user interface in various ways. In the case of Operators (described below), the designer simply needs to specify an attribute of the Operator for it to procedurally appear in Blender’s user interface. If more complex user interface layouts are required, the add-on can register classes of Blender’s `bpy.types.Panel` type. Panels are containers for user interface elements that allow the user to adjust the value of properties, and calling Operators using buttons. The `bl_space_type` and `bl_region_type` attributes of a Panel class specify where in the user interface of Blender this Panel should appear (which editor, and which part of the editor, respectively).

Each Panel class implements a `draw` method, which receives the current `context` as an argument. This method defines the content of a Panel by calling methods on the Panel’s `layout` attribute. For example, calling a `prop` or `operator` method adds a widget representing a property, or a button calling the Operator, respectively. The layout of a Panel can be further structured by calling methods such as `row` or `col`. These methods segment the layout and return objects which the aforementioned methods can be called on as well, organizing the Panel’s layout to the designer’s liking.

³https://docs.blender.org/manual/en/latest/advanced/blender_directory_layout.html#blender-directory-layout

⁴The full list is available at <https://wiki.blender.org/wiki/Process/Addons/Guidelines/metainfo>.

⁵Subclasses of Blender’s `Panel`, `UList`, `Menu`, `Header`, `Operator`, `KeyingSetInfo`, `RenderEngine`, `AssetShelf` classes

2.1.3 Operators

In Blender, Operators are methods that are triggered by Blender’s user interface. They are structures derived from the `bpy.types.Operator` type, and can implement multiple attributes and functions, mainly the `poll` and `execute` functions [8, ch. 7].

The Operator’s `poll` function serves to verify that the context is prepared to execute the Operator (for example, checking if any objects are selected). If the verification fails when the Operator is executed, an exception is raised. However, the `poll` function can be called on its own, making it possible to check if an Operator is executable without relying on Python’s `try-except` structure.

The Operator’s actual effect upon its execution is defined in the `execute` function, which is always called with `context` as an argument. Operators act as methods that alter the state of the application and are not used to obtain and return values from the application. Instead, each `execute` function (and other similar ones) must return one of predefined values [2, Operator Return Items]. The outcome of the Operator’s execution helps determine how to, for example, add them to the list of actions to Undo. Possible return values include the following:

- `{‘FINISHED’}` – The Operator has successfully exited, and can be undone.
- `{‘CANCELLED’}` – The Operator did not do anything, so it is not necessary to register it as an event to undo.

2.1.4 Nodes

Various areas of the editor use Node graphs, which are a form of visual programming. This paradigm allows users without a background in programming (typically the end user of an application) to create and customize software applications through two-dimensional visual elements [9]. Blender’s model (similar to, for example, Unreal Engine’s Blueprints⁶) encapsulates algorithms into visual blocks. These blocks, called Nodes, have specified inputs and outputs, as well as properties that the user can manually adjust. Their functions span from simple value extraction from a structure or minor mathematical operations to large-scale definitions of lighting models.

Blender currently contains three primary editors where users can encounter Nodes within their Blender workflows: Shading, Compositing, and Geometry. Shading Node trees affect the properties of materials, particularly in relation to their textures and light properties. Compositing Nodes dictate the composition of various scene elements and views into the desired image and works on a (rendered) image basis. Geometry Nodes are a procedural, nondestructive way of altering the scene geometry, for example by translating or duplicating meshes without requiring the user to make substantial changes to the contents of what they are working on.

Nodes expose and transfer their input and output values through sockets. These are named visual rows on the Node that offer an anchoring point for Links: representations of matches between Node outputs and inputs (an example is provided in Figure 2.2). Connecting one Node’s output to the input of another via a Link defines the execution of the Node tree’s computation. The output value is computed first before the computation which uses said value on an input. It is possible for the user to create a cycle of data flow, but the editor highlights and then ignores these erroneous connections. In this project, a Link refers

⁶<https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints>

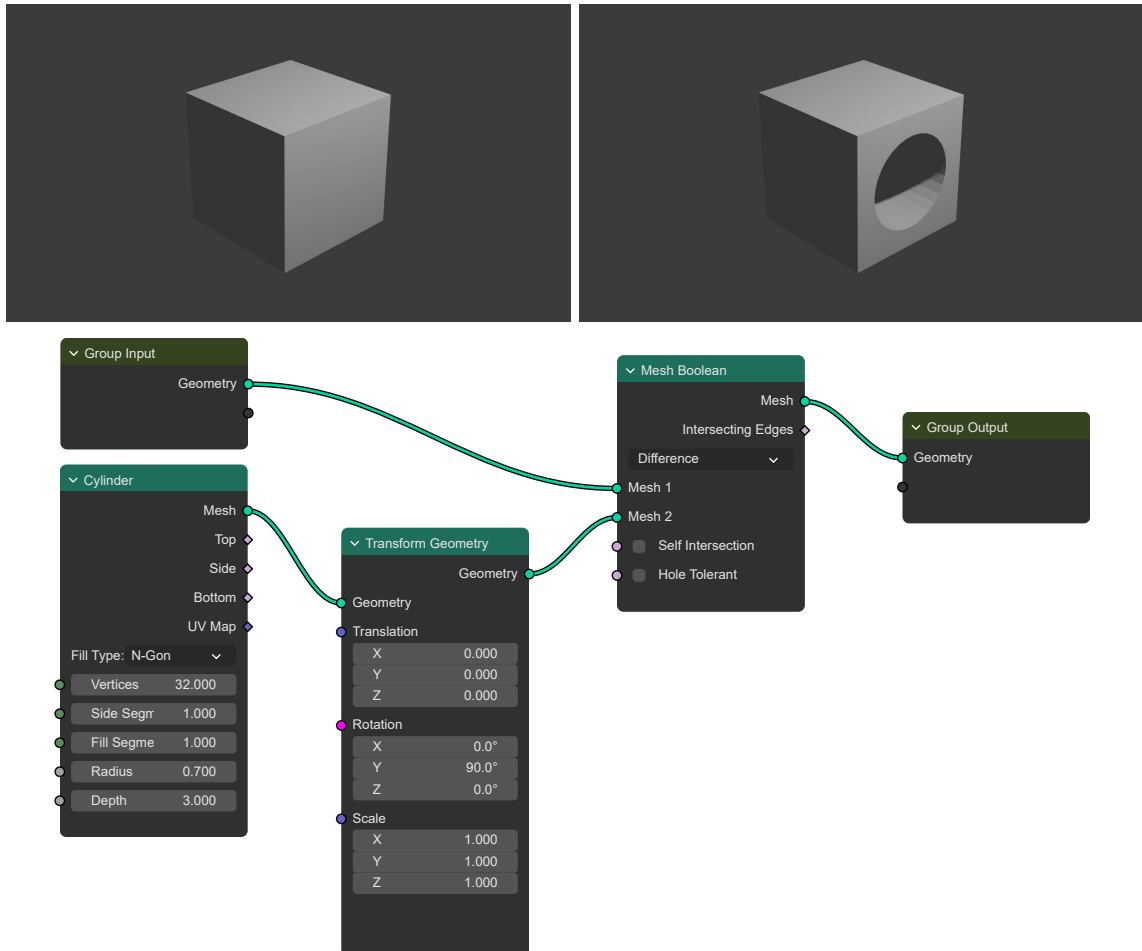


Figure 2.1: Example of using Geometry Nodes to create a hole in a cube without having to interact with the actual meshes included in the scene. A cylinder is created with the Cylinder Node, rotated with the Transform Geometry Node and then subtracted from the input mesh (cube) with the Mesh Boolean Node. The renders display the cube without (upper left) and with (upper right) the Geometry Nodes in effect.

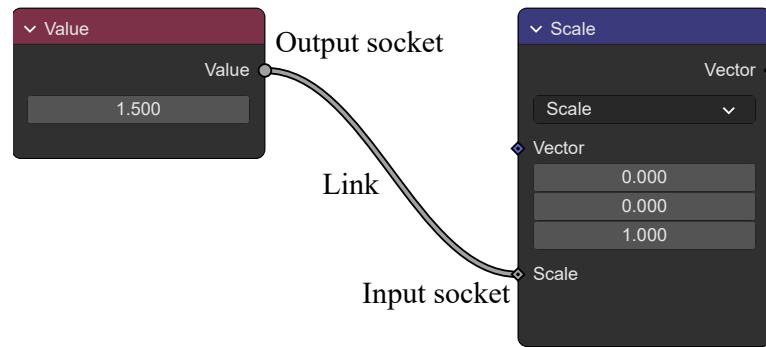


Figure 2.2: Example of two connected Nodes via a Link. The Value Node’s output socket exposes a value determined by the slider (1.5), which is connected to the Vector Math Node’s Scale input socket. This Node scales a vector determined by the Vector input socket by the value determined by the Value socket, and exposes it in the Vector output socket (which would now contain the vector $[0, 0, 1.5]$).

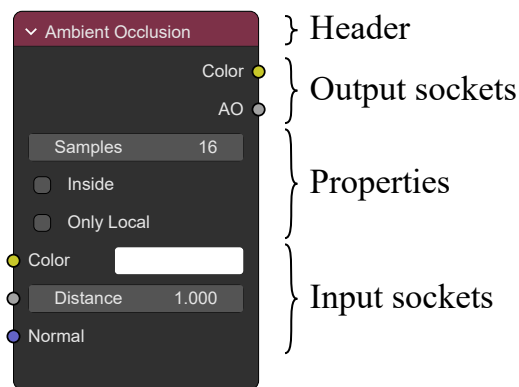


Figure 2.3: Example of an Ambient Occlusion Node, used in the Compositing editor. It has two outputs, Color and AO (the resulting shading from applying ambient occlusion, and the factor for each pixel, respectively), three properties (one value, two boolean-type), and three inputs.

to the connection between an output and an input, and the term *wire* refers to the visual representation of the Link. Since Blender refers to the property defining the curvature of wires as “noodliness”, it is technically possible to refer to the visualization of Links as *noodles*. Within Blender’s internal representation, Links are implemented as objects referred to as `bpy.types.NodeLink`. These objects have attributes that refer to the sockets to which they are connected but none that describe their appearance.

An output socket is only characterized by its displayed **name** property. Its anchor point is always present, placed at the right edge of the Node, but its shape and color may vary (by default, based on the type of data it represents). An input socket is similar to the output socket, but its anchor point is on the left edge (enforcing a left-to-right flow of information), and an additional element is present to let the user set a default value should the socket not be connected. In case of vectors, this may cause the socket to take more space up rather than just a single row.

Node category	Amount
Compositing	101
Function	27
Geometry	175
Shader	99
Texture	36
Total	438

Table 2.1: Total amount of Node types in the Blender 4.1 documentation, in the five most numerous categories.

Between the three main Node editors, more than 430 different Nodes are available (categorized in Table 2.1), with more being added with new releases. Each represents a different function and has its own set of properties (described below).

Node Contents

Visually, a Node consists of the following parts: a header, its output and input sockets, and properties, shown in Figure 2.3. Each Node has a header, and at least one socket, but it is possible for a Node to have no properties whatsoever, or none of either input or output sockets.

Header – The header of a node contains its name and a small arrow which toggles whether the Node is collapsed (hidden). By default, the Node’s name is taken from the Node object’s `bl_idname` attribute. However, it can be overridden by setting a custom `label` for the Node, and certain Nodes will change their displayed name based on a specific property of the Node (i. e. the Vector Math Node, based on the selected mathematical operation, seen in Figure 2.2). The color of the header is determined by the category to which the Node belongs, and this category-color pairing can be adjusted in the Preferences window. Like with the name, the color can also change depending on the Node’s properties (for example, the new Mix Node, which can pose as a Node from the Converter, Color or Vector category). The header may also feature a Preview button which toggles a small preview of the Node’s output above the Node itself.

Outputs – The Node’s output are individual sockets that expose the results of the Node’s internal calculations. Each socket has a name and a Link anchor icon whose shape and color reflects the type of the value on output.

Properties – Almost all Nodes feature properties: UI elements unique to the Node type that control the attributes of its inner computation. The internal representation of each Node features several properties, including those inherited by any Node object, as well as those controlled by these UI elements. However, there is no consistency between what unique properties are defined for a Node and what is actually displayed. Certain options are hidden under certain circumstances (the values of other properties, or the current rendering engine, for example).

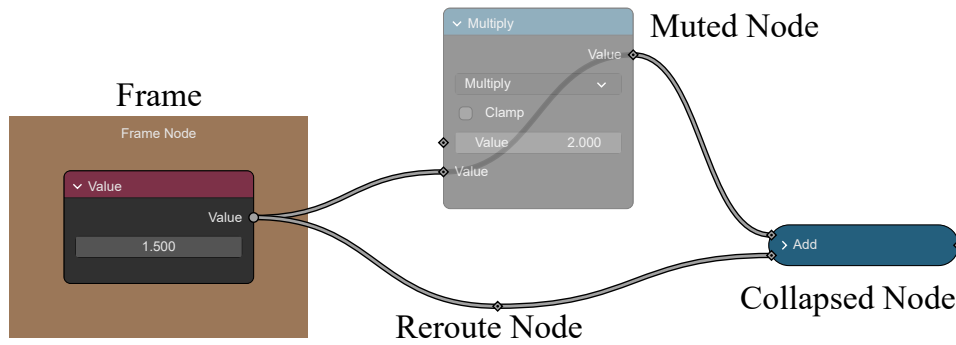


Figure 2.4: Examples of the advanced features of a Node editor: Node Frames, muting Nodes (including an inner Link), Reroute Nodes and collapsing/hiding Nodes.

Inputs – The Node inputs are sockets just like outputs. Unlike those, however, they do not allow for more than one Link per socket. If no Link is connected to an input, the socket provides a default value, and a UI element is present to let the user set this default value.

2.1.5 Advanced Node Operations

In the Node editor environment, the user has more options to construct the Node graph beyond just adding and removing Nodes. Special functions, seen in Figure 2.4, are included in Blender to allow users to organize their graph even as it grows and becomes more complex.

Node Hiding - The user can hide the contents of a Node by collapsing it, leaving only its header visible. This does not affect the functionality of the Node, and does not interfere with its input and output Links. Each socket's anchor remains visible on the header, and the dimensions of the header might change to ensure that all anchors' markers remain apart.

Node Muting - Muting a Node lets the user disable a Node's functionality without having to alter the structure of the graph. It turns the Node translucent to distinguish this status. To maintain the original data flow, a Node will automatically generate internal Links which connect its own inputs to outputs, showing how the unchanged data is transferred.

Reroute Nodes - Reroute Nodes are a special kind of Node represented by only one input and output, and rendered only as a socket marker along the wire of a Link. Their use is to control the placement of Links, which are otherwise automatically drawn between their originating output socket and target input socket.

Node Frames - Node Frames are a tool for organising a Node graph into visual sections to which the user can assign a label and color. They do not affect the functionality of a Node graph as a whole. At the same time, they are not an inert object, as manipulating Nodes in a graph will interact with the Node Frames present in the graph. For each Node, it is tracked whether or not it belongs to a Node Frame.

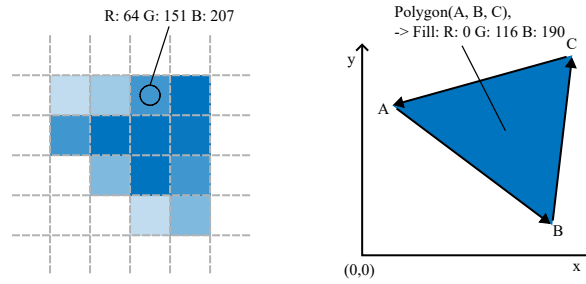


Figure 2.5: Comparison between the raster (left) and vector (right) graphical models. Raster utilizes small elements (pixels), vector works with discrete graphical elements such as polygons.

Node Groups - Grouping several Nodes together in a graph causes them to be replaced by a single Group Node in the original graph. The Nodes themselves are transferred to a wholly different graph, with a Group Input and Group Output Node added to control the data flow. Blender allows for manual control over the inputs and outputs of such Group Nodes, allowing the user to hide complex Node structures away as well as reuse such structures in different places in the graph.

Fortunately, the Group Node is defined the same way as any other regular Node, and the resulting group graphs are displayed in the editor as any other graph, so this feature does not interfere with the proposed design of the application.

2.2 Scalable Vector Graphics

Scalable Vector Graphics (SVG) is an XML-based format for creating, distributing, and displaying two-dimensional vector graphics. It is maintained by a work group of the World Wide Web Consortium (W3C), and the `image/svg+xml`⁷ MIME type is specifically registered for it. Displaying SVG files is now implemented by most web explorer applications, and conventional vector graphics editors can read and export to SVG [12].

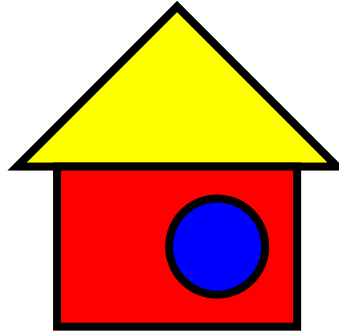
2.2.1 Vector Graphics

As opposed to raster graphics, which convey visual information through a lattice of different-colored elements (called pixels), vector graphics rely on a collection of elementary geometric shapes drawn with certain dimensions and coordinates. For a visual example, see Figure 2.5. The upside of vector graphics is their ability to scale them without producing the same artifacts that a pixelated raster image does. Therefore, they are the preferred option for simple images such as schematics and diagrams.

2.2.2 XML

The basis of the SVG format is XML (Extensible Markup Language), which is a document format which stores information in tagged, nested entities. Tags (`<tag>...</tag>`) wrap around data stored between them in a serial fashion, and may contain metadata in the form of element attributes (`<tag attr='val'>`).

⁷<https://www.w3.org/TR/SVGTiny12/mimereg.html>



```
<svg>
  <polygon points='30 10 10 30 50 30' fill='yellow'/>
  <rect x='15' y='30' width='30' height='20' fill='red'/>
  <circle cx='35' cy='40' r='6' fill='blue'/>
</svg>
```

Figure 2.6: A rendition of a simple house using vector graphics. Each element nested in the `svg` grouping element defines a shape using various attributes that specify its position, size, and appearance. Specifications regarding the shape outline have been left out. As some readers may know, the selection of colors is not arbitrary.

Additional meta information can be provided to XML documents, such as an XML schema, which defines the proper structure of an XML document as well as the necessary attributes of certain tags. Other meta information can pertain to the semantics of various XML tags.

As it builds upon XML, SVG is technically a namespace that adds semantics to a selection of tags (such as `circle` or `a`), and specifies how to process these documents to achieve consistent visual results [5]. An example of an SVG structure that defines visual content is provided in Figure 2.6.

In the add-on, building the SVG structure is handled using Python’s `xml.etree` module. In this module, individual elements are instantiated as `Element` objects, and the structure is defined by appending such objects to other objects, treating them like lists [3, ch. 6].

Well-formed XML

XML specification sets several rules, referred to as *well-formedness constraints*, which define the formal syntax of XML elements, as well as an XML document as a whole [5, ch. 1.2].

Such constraints can pertain to individual elements. Examples include:

- Should an element use a *start-tag* and *end-tag* (the two tags in the example above), then its names must match.
- An attribute name must not appear within a tag more than once.
- The `<` character may not appear within an attribute value.

A well-formed XML document conforms to all well-formedness constraints and has further requirements placed upon its structure (rather than content). Such document consists of a single (*root*) element (which may contain sub-elements), as well as *prolog* and *Misc.* (sic) sections before and after the element, respectively, containing additional meta information and comments [5, ch. 2.3].

For SVG, conformance to standards is via its Document-Object Model (a platform-neutral representation of trees), rather than a particular file format. A standalone SVG file must be a well-formed XML document (as described in Section 2.2.2). Its root element must be an `svg` element and it as well as any other elements within the tree must be a *conforming XML-compatible SVG markup fragment*, conforming to various XML and CSS specifications [16, ch. G.2].

2.2.3 Concepts

SVG represents visual content via graphical elements placed in its XML document. Applications capable of rendering SVG content, called *user agents*, display this content to a desired viewport. The rendering process follows the *painter's model* – elements defined further in the document are rendered over the elements defined earlier, potentially partially or completely occluding them. For example, in Figure 2.6, if the `circle` element was defined before the `rect` element, then the circle would be completely occluded by the rectangle.

In standard applications, SVG uses a Cartesian coordinate system with a rightward x axis and a downward y axis, with its origin positioned in the top-left corner of the viewport. Units of measurement are, by default, abstract. Grouping elements (such as `svg` and `g`) 'reset' the coordinate system within themselves, allowing to set the content's position only in relation to the bounding element instead of having to take the group's position on the canvas into account.

Standard 2D homogeneous transformations can be applied to any element positioned in the canvas by having a `transform` attribute. One or more individual transforms can be specified for this attribute, either in their shorthand forms of `translate(...)`, `rotate(...)` etc., or directly as a six-element transformation matrix: `matrix(a b c d e f)`. This can apply to elements that lack attributes to position them directly, such as the `g` element, for which one cannot simply define an x and y attribute.

The visual model of SVG is comprised of basic elements, mainly lines connecting pairs of points, and shapes, comprised of a series of lines. Each shape element (such as `rect` or `ellipse`) thus has an equivalent definition by a `path` element (described below).

The key distinction is that shapes are commonly regarded for their inner area, enclosed by the series of lines defining them. This opens their appearance to be altered through a different set of attributes to lines, detailed below.

Paint

The SVG format offers various ways of filling a shape with color information. The immediate option is a single-color fill, defined by setting a shape's `fill` property or styling attribute. Colors are defined in the SRGB color space [1] and can be symbolized in various ways: using one of SVG's recognized keywords⁸, or by providing its RGB definition. SVG supports multiple formats for providing an RGB definition.

⁸<https://www.w3.org/TR/SVG11/types.html#ColorKeywords>

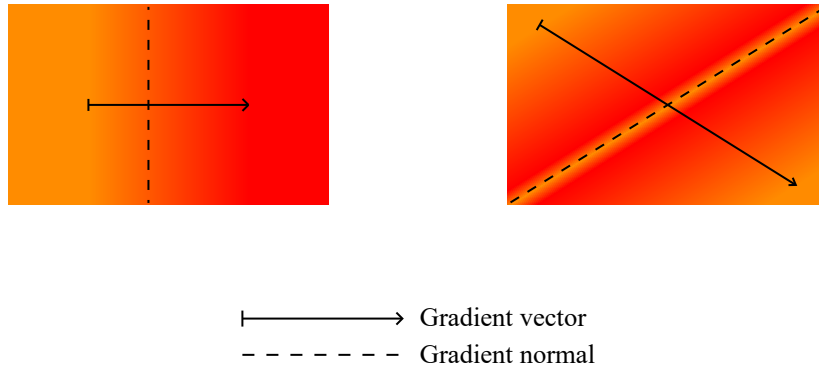


Figure 2.7: Examples of linear gradients, and visualisations of their gradient vectors as well as a highlight of a normal along which all colors are the same. The second gradient is more complex to showcase one such normal, and its vector is defined using the object’s bounding box, removing the normal’s orthogonality due to the transform in place.

An alternative to a single-color fill is using a gradient, which is a smooth color transition between multiple key colors. SVG implements two gradients as standalone elements: **linearGradient** and **radialGradient**, which are referenced by the **fill** property of shapes that the gradient is intended to be applied to. Each gradient element then contains several **stop** elements which define the key colors of a gradient with their **stop-color** and **stop-opacity** properties, and position in the gradient with the **offset** property.

The **offset** property is a number in the interval $\langle 0; 1 \rangle$ (which can be represented by an equivalent percentage). Although linear and radial gradients use different constructions to determine the color of any given point, the syntax of their respective **stop** elements remains the same. A shared attribute of both gradient elements is **gradientUnits**. It determines whether the coordinates are factors of the object’s bounding box, or coordinates in units matching the user space when the element is rendered (the **"objectBoundingBox"** option, or the **"userSpaceOnUse"** option, respectively).

Linear gradients are positioned using a *gradient vector*. Its start and end points are constructed during rendering based on the dimensions of the object that is painted in. Four coordinate attributes, **x1**, **y1**, **x2** and **y2** determine the position of these end points. The colors sampled from the gradient are rendered along *normals* to the gradient vector, which are usually perpendicular to the vector. However, certain transformations may affect this angle of coincidence (when using the **"objectBoundingBox"** option or applying a scale transform). Examples of linear gradients and how the angle can change are in Figure 2.7

Radial gradients are placed using a circle, defined by the gradient element’s **cx**, **cy** and **r** attributes, and a starting point defined by the **fx** and **fy** attributes. If the latter two attributes are not defined, they default to the center of the circle. The color of the 0% gradient stop is sampled at the starting point and progresses outward until the color of the 100% gradient stop is sampled along this circle. Examples of radial gradients with different starting points are in Figure 2.8.

Of additional relevance to this project is the fact that despite the documented expected behavior of SVG gradients, applications such as Inkscape treat certain gradient attributes differently (especially pertaining to default, unspecified attribute values). For a gradient

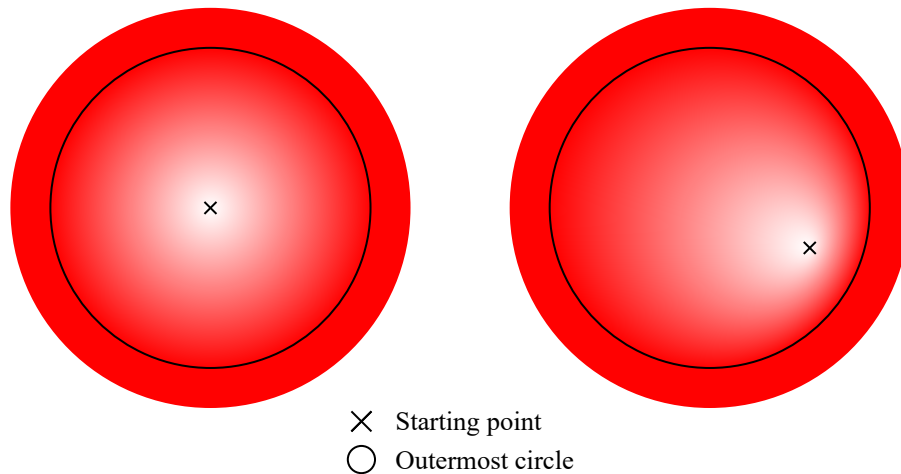


Figure 2.8: Examples of a radial gradient applied to a shape, with different starting points (marked by a $\times$ symbol) but identical outer circle (highlighted). The left gradient's starting point is kept as the center of the circle, while the right gradient's starting point is offset within the circle.

to function properly in Inkscape, it must be defined using the `"userSpaceOnUse"` option of the `gradientUnits` property. Fortunately, in this project, gradients are utilized ad hoc, so the expected dimensions of the filled-in shape are known and can be used to define the gradients.

Furthermore, Inkscape takes a different approach to radial gradients. Instead of the `radialGradient` element containing `stop` elements determining its appearance, it instead contains a reference to a `linearGradient` element which defines the key colors, even if it is not used anywhere on its own. Both of these issues are addressed in the project to allow the user to make adjustments to the exported file.

Styling

Visual appearance of individual elements can be defined using the same styling attributes offered by Cascading Style Sheets (CSS, described in Section 2.2.4). To render SVG, user agents need to support CSS styling (which most browsers already do, allowing them to render SVG files). If the agent further supports CSS 2.1 styling, measures can also be specified with pixels.

Clipping and Masking

Advanced cropping and transparency levels can be achieved through clipping and masking. Clipping selects a path to serve as a 'cutout', and only those parts of elements affected by a clipping path that overlap with it are rendered by the user agent. Effectively, a path is turned into a 1-bit visibility mask.

Masking is a more advanced form, as it allows the user agent to render elements affected by a mask with varying opacities, sampling the current element's opacity from an attribute of the mask: either its luminance, or alpha value [4]. Figure 2.9 shows examples of both.

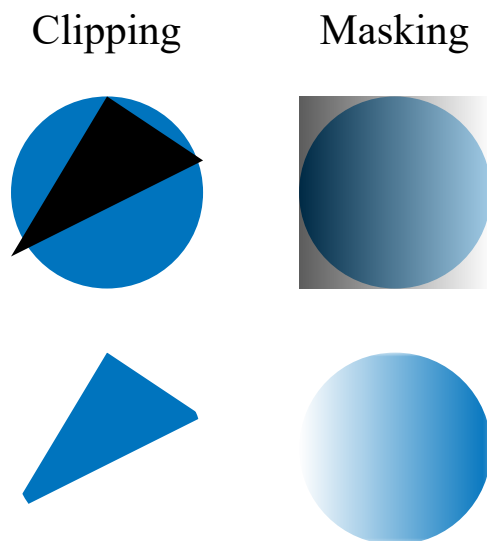


Figure 2.9: Examples of clipping and masking in SVG. Clipping is performed on a blue circle using a triangle (upper left), and only their overlap is rendered (lower left). Luminance-based masking is performed on the same circle using a black-to-white gradient (upper right, gradient has been made translucent for visibility), transferring the gradient onto the circle (lower right).

Organization

Although the final look of an SVG file depends primarily on the graphical elements included in the structure, it helps to organize the content with specific elements that control the structure rather than the appearance itself.

svg – The **svg** element is the necessary root element to all SVG documents, but also a standalone grouping element. If a given **svg** element is the root element, it is useful to define a **viewBox** attribute for it, which selects the rendered area of SVG content and hints at the expected size to display the image at. The value of the **viewBox** attribute is of the four-number format **X Y W H** (optionally comma-separated), which specifies the upper-left corner (**X**, **Y**) of the rendering area, and its width and height (**W**, **H**).

While it is commonly cited about vector graphics that scaling them does not incur any visual errors and that they are for it effectively dimensionless, SVG files rely on an application (*user agent*) to display them. These applications render an SVG image to a viewport of a specific size. In cases such as a browser or a $\text{\LaTeX}$ typesetting application that naively displays an SVG file, they scale this viewport to match the dimensions given by the file’s elements, or the root element’s **viewBox** attribute.

If the user agent has different, explicit viewport dimensions, then the rendered SVG content is appropriately transformed to fit this viewport, potentially affecting its aspect ratio. This can be further handled with the **preserveAspectRatio** attribute. This attribute specifies where to anchor the resized content within the viewport and whether it should be scaled down to fit entirely, or if it should cover the viewport entirely at the cost of cropping parts of the content [16, ch. 7.8]. Figure 2.10 shows different settings for this property.

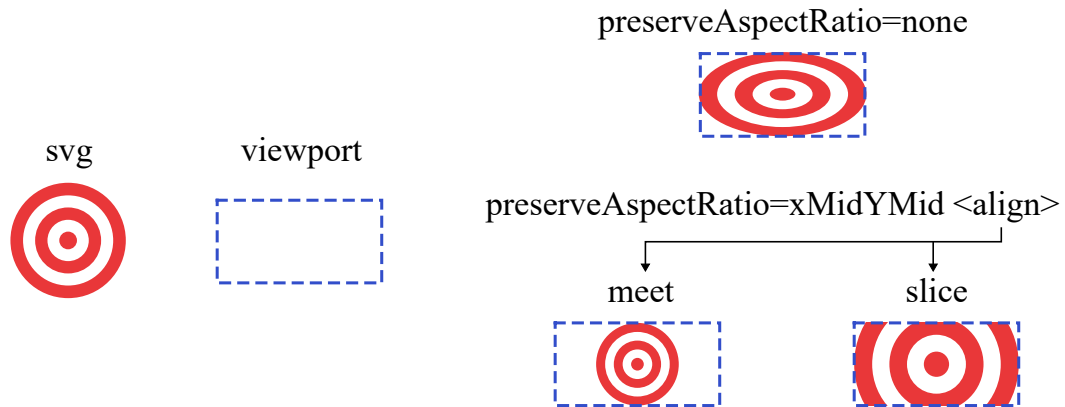


Figure 2.10: Examples of how an `svg` element (red target) can be specified to fit a provided viewport (blue rectangle) when the viewport’s aspect ratio doesn’t match that of the element.

Nesting `svg` elements allows for logical groupings. Like many other elements, a nested `svg` element should come with the attributes `x` and `y` which specify its positions within its parent coordinate space, and defining their own `viewBox` attributes is a shorthand for clipping their content to a rectangular area.

Unfortunately, it cannot be used for grouping in this project, as vector editors such as Inkscape treat nested `svg` elements differently to regular groups, making them immutable and jeopardizing any user attempts to adjust the vector image after the export.

g – The `g` element is a simple element that adds all of its children elements to a group. Unlike the `svg` element, it cannot be positioned simply by specifying a `x` and `y` attribute, and instead requires a `transform` attribute.

In addition to positioning, two other functions are utilized in this project;

- Setting a `clip-path` attribute to a `g` element causes all of elements in the group to be affected.
- Specifying a `class` attribute of a `g` element enables more intricate styling through the use of CSS and its selectors (described in Section 2.2.4).

defs – When defining elements for reuse throughout the document, it is useful to do so in a single `defs` element, which exposes the elements to be referenced within the document, while no child of the `defs` element is actually rendered on its own. It is recommended to include elements such as gradients in this group [12, ch. 5.3.2], even though these gradients are not rendered whatsoever and their location within the SVG structure does not matter.

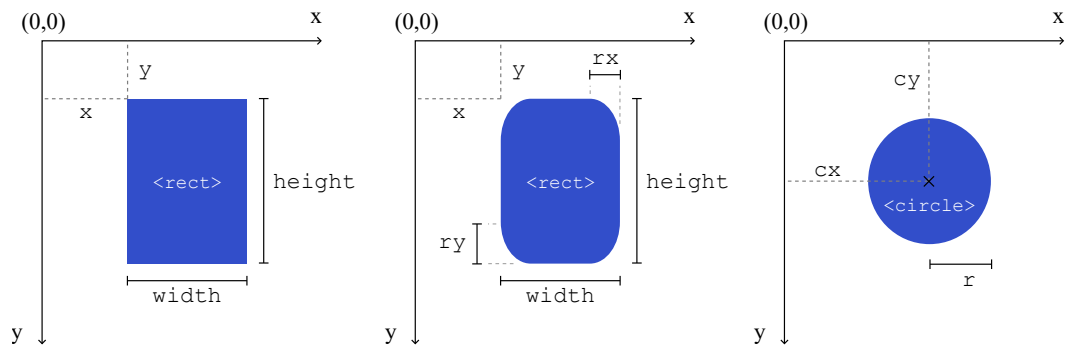


Figure 2.11: Examples of `rect` and `circle` (right) elements and how their various attributes affect their position and appearance. The former is showcased without any corner rounding (left) as well with specifications of both corner rounding dimensions (center).

symbol – Another example of an element that is not rendered on its own but only on reference is the `symbol` element. It effectively serves as a `g` element whose content defines a template, but can also make use of the `viewBox` and `preserveAspectRatio` properties described alongside the `svg` element. This is another type of an element that should be defined in the `defs` element.

To render the content defined by the template in a `symbol` element, the `use` element is used with a `href` attribute referencing the `id` attribute of the `symbol` element.

Content

This Section lists multiple elements which provide visual content for the SVG, as opposed to grouping or metadata only.

rect – One of the basic shapes that can be rendered using SVG is the rectangle. The `rect` element defines an axis-aligned rectangle of a specific width and height, and can have its corners rounded, as shown in Figure 2.11.

Again, the `rect` element has two attributes, `x` and `y` that determine its position within its parent coordinate space. Then, the attributes `width` and `height` specify its dimensions, and the attributes `rx` and `ry` determine the (in order) horizontal and vertical radii of arcs that replace the rectangle’s usual sharp corners.

The corner rounding attributes affect all corners uniformly. For different corners, the shape has to be drawn using the `path` element.

circle – Similar to the `rect` element, the `circle` element renders a circle to the canvas. It is defined by its center’s coordinates, given by the `cx` and `cy` attribute, and by its radius, given by the `r` attribute. Visualization of how these attributes define a circle is shown in Figure 2.11

path – While SVG also offers simple elements for straight lines and connected line segments (named `line` and `polyline`, intuitively), the most generic, flexible element to describe any type of line (or, as mentioned above, the outline of a shape in an equivalent fashion) is `path`. Its single proper attribute, `d` (for *path data*), contains a string representing a sequence of instructions that need to be processed to render the desired line.

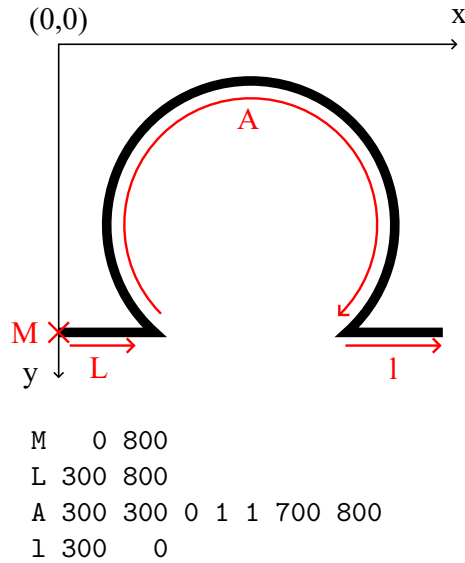


Figure 2.12: An Omega symbol drawn using the `path` element, with the series of instructions passed to the `d` attribute highlighted in the image and spelled out below. Of note is the last instruction, `l`, which differs from the earlier `L` instruction by being relative (its equivalent would have been `L 1000 800`).

The commands include the following:

- `M x y`: *moveto*, establish an initial point at (x, y) ,
- `L x y`: *lineto*, draw a straight line from current point to coordinates (x, y) ,
- `C x1 y1 x2 y2 x y`: *curveto*, draw a cubic Bézier curve with control points (x_1, y_1) and (x_2, y_2) to point (x, y) ,
- `A rx ry rot large-arc sweep x y`: *arcto*, draw an arc of an ellipse of axis dimensions r_x and r_y , with its horizontal axis rotated by rot degrees, ending in point (x, y) . Select one of the four generic solutions with flags `large-arc` and `sweep`⁹,
- `z`: *closepath*, draw a line to initial point and close path around resulting shape.

Generally, all of these instructions start with an established point. Multiple instructions of the same type can be repeated without having to specify the initial letter each time (provided that they use arguments). Such instructions also have relative variants (denoted by a lowercase letter instead of an uppercase letter) that define the endpoint by adding the provided coordinates to the starting point. For an example of a `path` element, see Figure 2.12.

As mentioned earlier, each shape-defining element has an equivalent *path data* instruction series that makes the `path` element produce a shape equal to said shape with those instructions, and they are featured in the SVG documentation [16, Chapter 9.1].

⁹This is a notoriously complex instruction, and an issue has been raised in 2011 to propose a simpler variant. The issue remains open to this day:

<https://www.w3.org/Graphics/SVG/WG/track/actions/3151?changelog>



Figure 2.13: The relative positions of a `rect`, `circle` and `text` element, with the `text` positioned to the same coordinate as the red circle, but with different `text-anchor` attribute value each time: `"start"` (left), `"middle"` (center) and `"end"` (right).

text – Instead of having to reconstruct each character with other graphical elements, SVG offers a handful of elements, such as `text`, which render the text content of the element through *glyphs* – units of rendered content, and effectively path-type elements that conform to the same attributes as any other path type element. Typically, there is a one-to-one ratio between characters and glyphs, although there can be exceptions where a single (for example accented) character requires multiple glyphs, or, inversely, multiple characters are rendered via a single glyphs (like ligatures¹⁰).

Text elements are one of the few cases in SVG where strings which are freely written between tags matter. It is this exact content that is rendered to the canvas. Several styling attributes exist that control the appearance of glyphs rendered in an SVG document, known as the font selection properties [16, Chapter 10.10]. Examples include the generic `font-family` attribute, `font-weight` to control boldness or `font-size` to control the size of the rendered glyphs.

The `text` elements also implement the `text-anchor` attribute. Each `text` element is positioned as a point and this attribute determines in which direction the text should be rendered from the point. Under default conditions, with Latin alphabets, this positions a line along which the glyphs are placed (but various text elements can be rendered along any path-type element, and languages such as Arabic or Han can be rendered from right to left, or top to bottom, respectively). The options for this attribute are `"start"`, `"middle"`, and `"end"`, meaning where in relation to the rendered text the control point should be. The difference is shown in Figure 2.13.

2.2.4 CSS

The styling language known as CSS (short for Cascading Style Sheets) was devised by the World Wide Web Consortium in 1994 [11, ch. 20]. It was a response to the growing need to control and improve the appearance of web pages defined in HTML, whose appearance attributes were quickly becoming unsustainable. Although its intent was to work with HTML documents, it is applicable to XML documents in general, including the SVG format. An example of using a CSS style sheet to style an SVG image is shown in Figure 2.14.

The goal of this styling language is to define the appearance of various HTML (or, in general, XML) elements by including a *style sheet* – a collection of rules (such as those in Figure 2.14 that target specific elements, or combinations thereof. They are either included in place using a `style` element, or imported from an external file using a `link` element. If

¹⁰[https://en.wikipedia.org/wiki/Ligature_\(writing\)](https://en.wikipedia.org/wiki/Ligature_(writing))

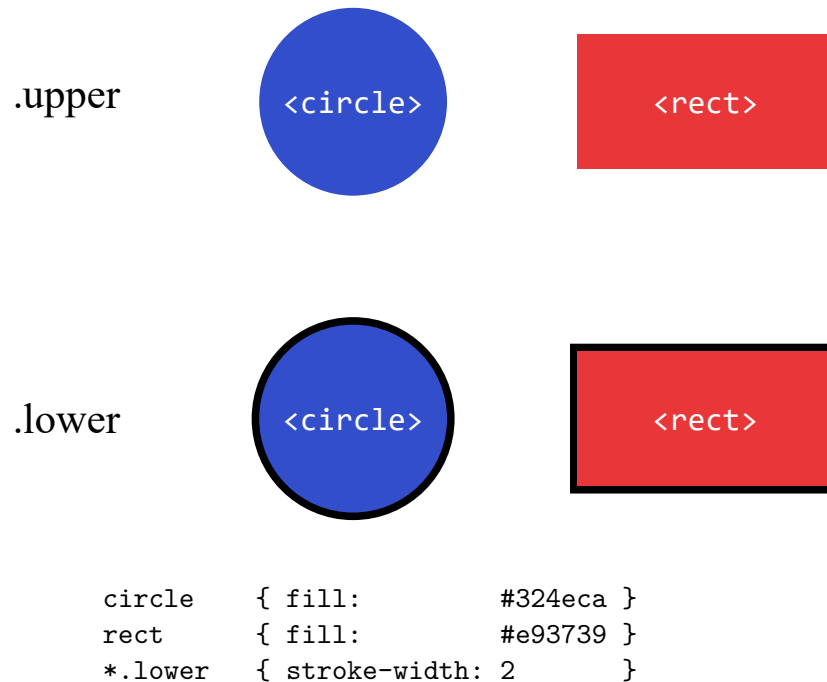


Figure 2.14: Example of applying CSS rules (bottom) onto an SVG document. The rules with selectors `circle` and `rect`, change the fill color of these elements. The last rule sets the stroke width of each element of the `lower` class to 2.

necessary, the appearance of a given element can be directly controlled by defining a `style` attribute for it [14, ch. 1].

CSS technology makes extensive use of assigning the `class` and `id` attributes to individual elements. The former is meant to assign multiple elements to a virtual group and allow them to be addressed and styled in a generic fashion. The latter is a form of naming a given element and should be unique across the entire document [14, ch. 2.1].

Stylesheet

Stylesheets in CSS are made up of rules which comprise a *selector* and a list of *declarations*. The selector part determines which elements will be affected, while the declarations define the individual styling attributes applied to the selected elements. Selectors are a composition of selection rules. Together, these rules provide strong expressive power to select various elements. The building blocks of a selector include the following:

- `*`: The universal selector. It selects any element, and is implicit wherever a part of a compound selector is missing.
- `elem`: Selector of elements of a given type (`elem` in this case).
- `.cls`: Selector of elements with a given `class` attribute (`cls` in this case), denoted by the period symbol.
- `#name`: Selector of element(s) with a given `id` attribute (`name` in this case), denoted by the `#` sign.

In case of the `class` and `id` selector, they technically immediately follow another selector (such as an element name, for example `rect.cls`). Using them on their own as presented is an example of the implicit `*` universal selector (for example `.cls` is interpreted as `*.cls`).

Selectors can be further combined, sometimes utilizing the XML structure in targeting specific elements:

- `s1, s2`: Selects elements targeted by selector `s1` or selector `s2`.
- `s1 s2`: Selects elements targeted by selector `s2` which are descendants of elements targeted by selector `s1`.
- `s1 > s2`: Selects elements targeted by selector `s2` which are direct children of elements targeted by `s1`.

As such, web designers have a powerful tool to control the appearance of individual elements in a generic fashion even in complex document structures [14, ch. 4].

Chapter 3

Proposal

The goal of this project is to develop a Blender add-on which lets the user create SVG representations of their Node graphs, for use in documentation and technical reports such as this one. To demonstrate that, this technical report depicts all Blender Nodes using the add-on (save for a handful of examples in Section 5.2.1, where screenshots are provided to illustrate differences between the source and the output). This chapter describes the proposed workflow for the add-on and initial plans for the project.

3.1 Add-on Workflow

The add-on implements a simple pipeline, as outlined in Figure 3.1. A tab is added to the user interface of the Node editor area, which includes an Export button. Pressing it invokes an Operator implemented in this project. As described in Section 2.1.3, an Operator has access to the `context` submodule, and is thus able to find the currently-displayed Node graph, extract the relevant information from the representation and construct an SVG document which mirrors the graph’s appearance.

Blender’s `bpy` module is, however, limited in that it does not expose the exact contents of what the application is currently displaying on screen. If it did, an algorithm could perform a 1:1 transform of the visual elements into an SVG file based on how the application renders them. A question¹ has been posted on the Blender Developer Forum asking about this specific issue, but has not garnered any significant traction.

Likewise, using a tool such as RenderDoc² shows how a single frame of the application is rendered. One can even see how the frame develops with each draw call, but Blender does not implement a call stack which RenderDoc could display to pinpoint where individual draw calls were made from in the C++ source code. Tracing these calls in the source code would give further insight on the exact details of how Node trees are rendered, and could improve the fidelity of the exported file.

Using SVG’s painter model (described in Section 2.2.3), the proposed output of the add-on can be described as follows: the document starts with the necessary metadata and namespace definitions, and the content is encompassed in a single `svg` element. This element contains a single `defs` element, which contains the CSS style sheet, as well as the definitions of symbols reused throughout the document.

¹<https://devtalk.blender.org/t/nodes-editor-ui-specification/32224>

²<https://renderdoc.org>

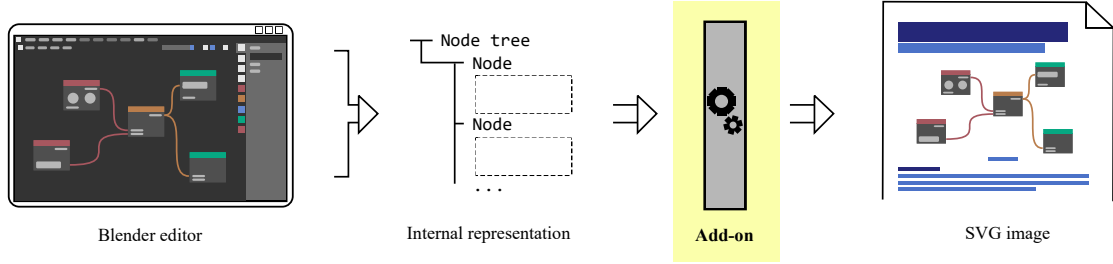


Figure 3.1: Schematic of the purpose of the add-on. The Node tree open in the Blender editor is processed by the add-on implemented in this thesis to produce an SVG image representing the Node tree, for uses such as writing technical reports.

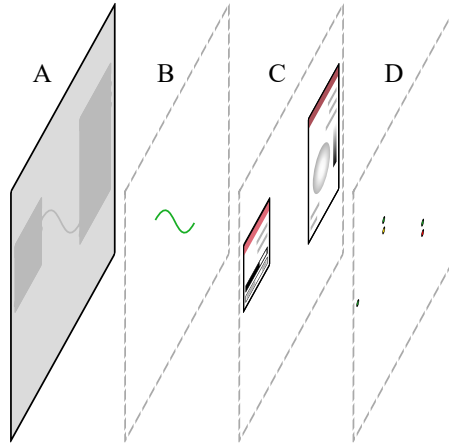


Figure 3.2: The proposed order of element definitions in the SVG output. First, the Node Frames are rendered (A), and the Links over them (B), which are always rendered behind Nodes themselves. Then, the Nodes are added (C) and, finally, the markers are added (D), which are not clipped by the same paths as the Nodes themselves.

Then, the representations of each Node are featured in a specific order, described in Figure 3.2:

1. Node Frames
2. Links
3. Nodes
4. Socket markers (which are not affected by the same clipping path as the Nodes themselves)

3.2 User Interface

As part of the thesis itself and potential user research, a vector mock-up (seen in Figure 3.3) of the add-on's potential user interface has been designed. In addition to exposing the *Export* button to run the implemented exporting operator, the panel allows the user to alter various options of the export process itself.

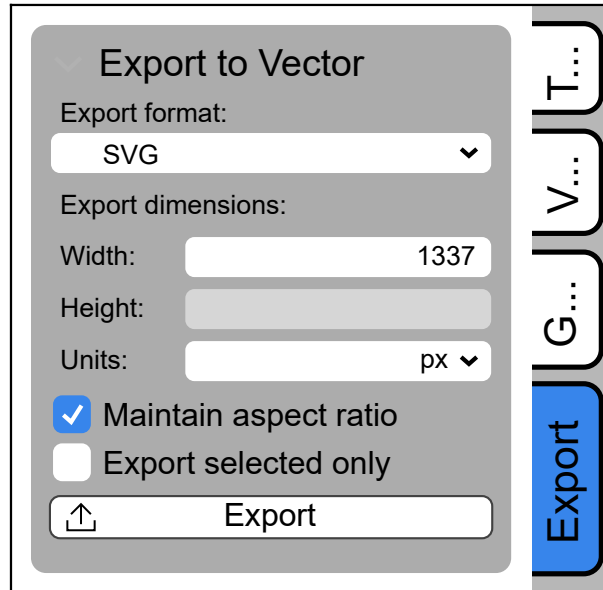


Figure 3.3: Mock-up of the proposed add-on user interface, including various export settings. It is modeled after Blender’s Toolbar UI area. The image’s color scheme and cropping has been altered from that in the form, but its content remains the same.

Mainly, the user can choose the target format of the export, as well as the dimensions of the resulting image. Although intuitively one should not have to care about exact dimensions with vector images, it has been detailed in Section 2.2.3 that it could be beneficial to define the image’s expected dimensions using the `viewBox` attribute. Finally, the add-on lets user limit the exporting process to the selected Nodes only. One notable element is missing from this proposal, which is the exported file’s target name and directory. It has since been implemented in the project.

Early into the development cycle, after initial explorations of the two main technologies detailed in Chapter 2, a survey was conducted to collect feedback from the potential user base. The survey was distributed as a Google Forms questionnaire. It contained two main segments: questions on experience with the Blender application and questions on the user’s experience with writing technical reports (especially regarding including vector images). The content of the questionnaire is described in Section 5.1. The results are then included in Appendix A.

Part of the user feedback to the initial concept was a request for multiple different features, which mandated an update to the user interface. The connecting factor to all these features is control over the output’s appearance, be it conforming to the user’s current Preferences setting or allowing the user to define aspects of the output themselves.

3.2.1 Element Quality

Certain elements³ of the content of various Node types will not transfer to vector graphics, and their appearance can only be imitated (for example, using multiple thin rectangles to imitate a nonlinear gradient). However, the quality of this imitation scales with the amount of elements used, and users may want to limit the size of the output file at the cost of the quality of the imitation.

For that reason, two options are added to the user interface: *Element Quality* and *Use Gradients*. The former is a numerical property that represents the partitioning of complex UI elements. Widgets such as Color Picker or Ramp (described in Section 4.1) are broken down into triangles and rectangles, respectively. The higher the number of partitions, the smaller their individual differences. Similarly, graph curves are reconstructed by sampling across their domain, with the number of samples matching the *Quality* property.

The second option, *Use Gradients*, is a boolean type property which determines if ad hoc linear gradients should be generated for each partition of these widgets. Specifically, this means that for every affected SVG element, a `linearGradient` element is created which contains at least two `stop` elements, quadrupling the amount of elements required to represent said widget.

3.2.2 Colors and Shapes

The most requested export option was the ability to control the color scheme of the output. Originally, this only extended to the colors of the Nodes' headers (defined by their category as described in Section 2.1.4).

However, more color options had to be added over time, as users would want complete control over the appearance of the export. Even those who would not, Blender comes with multiple preset color Themes which affect the appearance of their Node editors, and it would be reasonable for their exports to match their selected color scheme. Examples of Nodes rendered in different Blender Theme presets are shown in Figure 3.4.

The color adjustments let the user change the color of all rendered text. Since the background over which a text can be rendered is variable, the user has the option to define the color of text in multiple contexts, such as when rendered over the Node's base color or over a Value slider. This distinction copies Blender's own multiple definitions of text color based on what context they appear in. If the user does not want to set each individual text color, they can instead check the *Use generic color* option and set a generic color for all text.

The colors of each non-text, non-header element of the Node's appearance can also be set by the user. This ranges from the background of each Node, through multiple text fields and value bars, to the graph axes that certain Nodes contain. No generic option is provided for these elements.

Then, the user can also set the color of each socket type. These colors are not accessible in the Preferences or anywhere in the `bpy` module. Their default values have been explicitly included in the project. Finally, the output can be adjusted by setting the outline of all rectangle elements and whether the corners are rounded, mirroring Blender's appearance.

The user may not want to alter the appearance of the exported image at all. They then have the ability for all the colors to be directly imported from their Theme settings by checking the *Use Theme colors* option.

³Specifically: Color Picker, Curve, and Ramp, all described in Section 4.1.

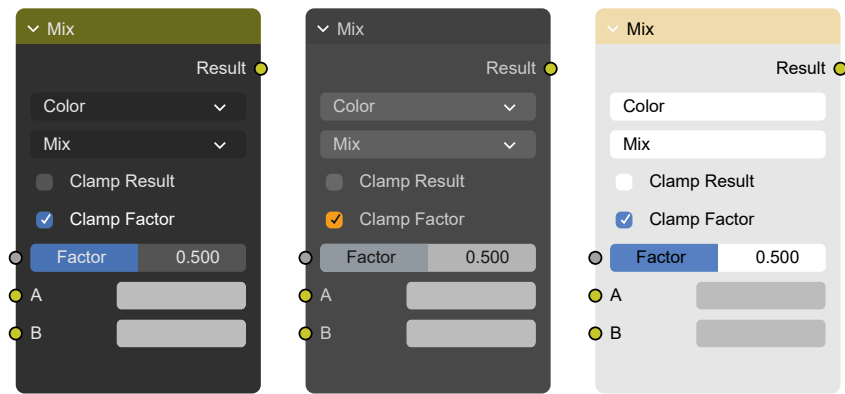


Figure 3.4: The Mix Node in three different Blender Theme presets, available in the editor by default: *Blender Dark*, *Modo*, and *Print Friendly*. All were exported using the Use Theme colors option after the editor’s Theme was changed accordingly.

3.2.3 Presets

Due to the large number of options in the add-on, users may want a quick way to set the export properties up to their preference. For that reason, the add-on’s user interface includes a *Save/Load Configuration* panel. In this panel, the user can decide between saving all their settings to a configuration file, or importing their settings from a previously exported file. The two operations are handled by separate UI elements instead of a single one that switches between modes to ensure that the user does not accidentally overwrite their settings.

Chapter 4

Implementation

The add-on is implemented as a Python module composed of multiple files as opposed to a single script (as described in Section 2.1.2). It consists of a `__init__.py` script, featuring all the prerequisites described in Section 2.1.1, and a `src` folder containing all the other scripts:

- `constants.py`: Variables that do not change throughout the editor's runtime and execution of the export operator.
- `methods.py`: Methods and functions called from different places in the add-on.
- `widgets.py`: Classes representing the graphical widgets of Blender's user interface, including methods to represent themselves as SVG objects.
- `categories.py`: Dictionary defining the meta properties of each Node type.
- `header.py`: The `UIHeader` class which represents the Header of a regular Node.
- `shape.py`: The `UIShape` class which represents the markers placed over socket's anchors.
- `node.py`: Various classes which represent different kinds of Nodes.
- `converter.py`: The `Converter` class which encapsulates the export process.
- `property_group.py`: The Property Group which encapsulates all of the properties used by the add-on.
- `operators.py`: Operators implemented as derivatives of the `bpy.types.Operator` class to invoke from the user interface.
- `panels.py`: The user interface of the add-on itself, implemented as derivatives of the `bpy.types.Panel` class.

To perform the conversion itself, the user must invoke the `UIExportOperator` Operator from the user interface, which creates an instance of the `Converter` class, and calls its `convert` method which returns the SVG representation. The operator then prints said representation to the desired file alongside relevant XML headers.

4.1 Widgets

To represent the set of UI elements that make up the Node’s socket and properties’ visual appearance, this project implements Widgets – objects that define the SVG representations of certain UI elements. Widgets make up the entirety of the Nodes’ contents, and some are defined in such a way that they incorporate other Widgets as part of themselves, or serve as a container for multiple different Widgets when formatting is required.

Each Widget is implemented as a class that inherits from a parent `Widget` class. Each Widget subclass fully inherits one specific function from this parent class, the `svg` function. This function handles the generic SVG structure of representing Widgets: an overarching `g` element that contains the relevant `transform` attribute, and a `clipPath-g` pair that holds the actual Widget content and crops it to have rounded corners if necessary. This inner `g` element also has a `class` attribute set to the `css_classname` defined for each different Widget subclass, ensuring that the overarching CSS styling properly targets elements of the same type based on what Widget they belong to. Building the SVG structure of the output is handled using Python’s `xml.etree` module. The `svg` function returns the representation as instances of `ET.Element`.

The `svg` function signature allows for the Widget’s position to be dictated by the `x` and `y` argument. It is always called with a `width` argument that determines the width of the group element framing the contents. It offers an option to add padding along the vertical edges of the group for readability purposes, manipulated through the `resize` argument.

However, the content of the inner-most `g` element is handled outside the `svg` function: instead, each subclass implements its own `fill_svg` function which actually populates the inner-most element with the representation of the Widget.

Each Widget also implements its own `height` function, which returns the expected height of the SVG representation of an initialized Widget. It is not represented as a simple class attribute, as the height of certain Widgets may change depending on their content. Examples of all listed Widgets are shown in Figure 4.5.

Label – The Label Widget is the primary way to add strings of text to the SVG output, and is commonly invoked by other Widgets. It is initialized with two arguments, `text` (defaults to an empty string), which defines the displayed string, and `alignment`, which determines how the text is positioned within the Widget’s boundaries.

The core of the Widget’s SVG representation is a single `text` element. Its contents are set to the desired string, while the required alignment is achieved through its `x` attribute and `text-anchor` attribute (described in Section 2.2.3). The use of the label widget is shown in Figure 4.1.

Value – This is the main widget for displaying any sort of numerical values. It takes two primary arguments: `name`, which is displayed over the bar and `value` itself.

The Widget’s contents consist of a `rect` element and one or two Label Widgets reflecting the two initialization arguments. However, an additional argument can be passed during initialization, `minmax`, which, if not `None`, is a tuple of two numbers. A bar then fills up a portion of the Widget’s frame, proportional to where the `value` is along the interval defined by this argument. The use of the Value widget is shown in Figure 4.5 (at the end of the list).

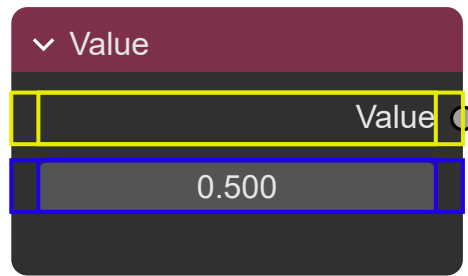


Figure 4.1: Examples of the Label (yellow) and Value (blue) Widgets in the Value Node. The smaller inner rectangle marks the dimensions of the Widget’s clipping rectangle. The Widgets’ `svg` functions are called with the width of the Node, but are shrunk to the displayed width.

The Float Widget class is a special case of the Value Widget, where the value is appropriately formatted to avoid long strings of decimal places caused by floating point errors. The Angle Widget class is, in turn, a case of the Float Widget, which appends a degree mark to the displayed text.

Boolean – This Widget represents boolean properties exposed to the user via a checkbox. It is initialized with two arguments, `name`, which displays next to the checkbox, and `value`, which determines whether the checkbox is actually checked or not.

The Boolean Widget defines the elements that make up the checkbox; displaying the property name is relegated to a Label Widget created within the invocation of the `fill_svg` function. An example of a checked Boolean Widget is shown in Figure 4.2.

Columns – This Widget has no rendered content of its own. Instead, it is a tool to break the space reserved for it down into columns which are filled with other Widgets. These are passed as `Widget` objects at initialization through the `wids` argument.

The columns are, by default, of uniform width, but a number array of equal length to the `wids` argument can be passed through the `ratios` argument, which scales each column to the same proportion as the corresponding number in the array to the sum of this array. Finally, the `resize_override` argument determines whether the Widgets contained within the Columns Widgets should be resized on their own. An example of a Columns Widget containing a Label and an RGBA Widget is shown in Figure 4.2.

The `FortySixty` widget class is implemented as a shorthand for an instance of `Columns` with the ratios implicitly set to 2:3.

RGBA – Display of a simple color value, taking only a single argument: `color`, which is a string representation of the desired color. It is then used to set the `fill` attribute of a `rect` element. An RGBA Widget is shown as part of a Columns Widget in Figure 4.2.

Vector – Many values are passed as Vectors consisting of three separate values. This Widget displays them as a column of three nameless `Value` Widgets, whose values are defined in the `values` argument. The name of the attribute, given by the `name` argument, is displayed as a Label above this column.

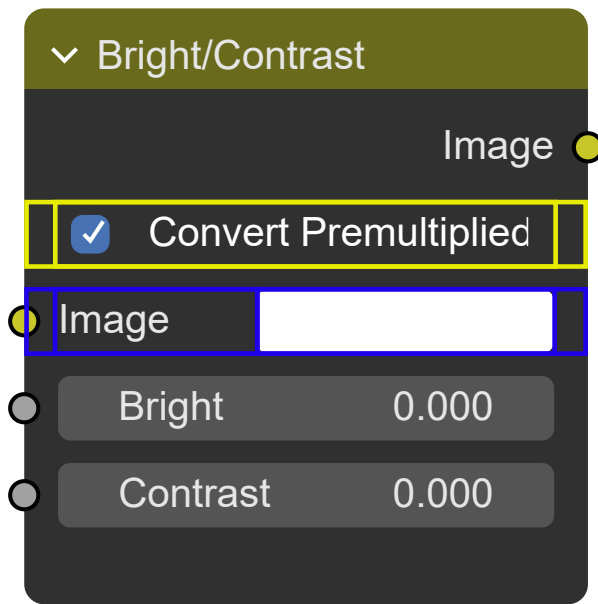


Figure 4.2: Examples of multiple Widget types on the Brightness/Contrast Node. The Boolean Widget (yellow) is presented with a checkbox, and represents a boolean property whose value is set to True. A color socket (blue) is represented by the Columns Widget, which contains two different Widgets: Label (left), and RGBA (right).

Dropdown – This is the primary way to display enumerable properties. As a Widget, it only takes one argument – `value` – which is displayed in a box alongside a downwards arrow just like they are in the Blender user interface. A `LabeledDropdown` variant exists which adds a Label next to the Dropdown, with the ratio of the two widths dependent on the dimensions of the Node.

However, the enumerable properties are initialized with tuples representing each option. Simply passing the immediate value of the property does not accurately reflect the string displayed in the user interface. For this reason, the `dropdown` function is implemented, which is called in place of the Widget constructor itself. This function reads the RNA structure of the property, where it accesses the individual options and is able to retrieve the display name of the selected option.

Color Picker – User color selection is not handled via an RGBA Widget alone, but rather through the Color Picker, which consists of a color wheel and a brightness slider. The color wheel is imported and scaled from a `symbol` element defined in the SVG file’s `defs` element (described in Section 4.5). The slider is generated ad hoc as a `rect` element filled with a linear gradient.

The user selects the color on the Color Picker by sliding two cursors over the color wheel and the slider. The tricky aspect of reconstructing the Color Picker is finding the proper coordinates to render these cursors at based on the color alone.

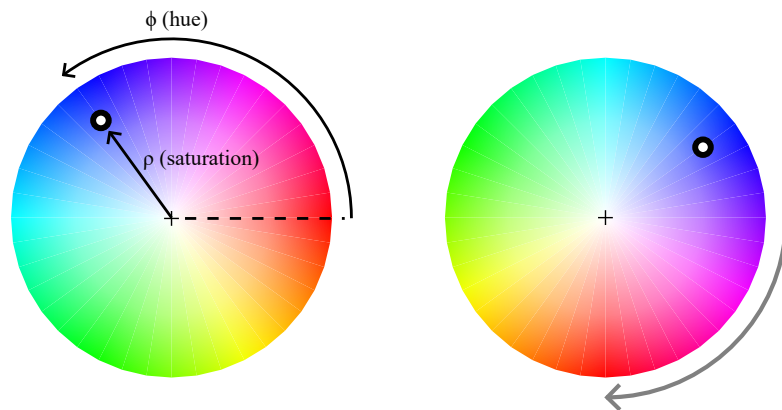


Figure 4.3: Visualization of how the position of the Color Picker cursor is calculated from the color’s hue and saturation values, and how the desired position is achieved by also adjusting for the position of red color ($hue = 0$).

The position of the first cursor in relation to the color wheel is calculated by converting the Color Picker’s color value to the HSV space. The coordinates of the cursor are first found in polar coordinates (ρ, ϕ) , where the radius ρ is equal to the converted color’s saturation, and angle ϕ as a proportion of the color’s hue:

$$\phi = (hue - 0.75) \cdot 2\pi$$

The subtraction of 0.75 is performed to account for the fact that the color red, which is represented by a hue value of 0, is found at the bottom of the color wheel, but an angle of 0 points to the right edge of the color wheel. A visualization of this calculation is in Figure 4.3

The Cartesian coordinates are finally found through the `methods.polarToCartesian` function, which utilizes Blender’s built-in `cmath` module that implements complex numbers. The position of the other cursor is simply the value element of the HSV color.

With all that said, this is only one of the possible appearances of the Color Picker widget in Blender. The user can choose a different type in the Preferences setting, and even the wheel’s appearance alone changes based on the color display settings defined under the Scene > Color Management property group. For that reason, the Color Picker widget SVG is only present illustratively, and the informational value of a Color Picker is always present in the RGBA Widget right underneath it.

SelectBar – An alternate mean of showing an enumerable property, this widget displays all the options, provided as a list through the `options` argument, and highlights the selected option, identified by the `select_index` argument. The Widget represents the options as `rect` elements with Label Widgets rendered over them.

Similarly to the Dropdown Widget, this Widget subclass is referenced by a `selectBar` function rather than its constructor. This function retrieves the name of each option as well as the index of the selected option required to call the constructor.

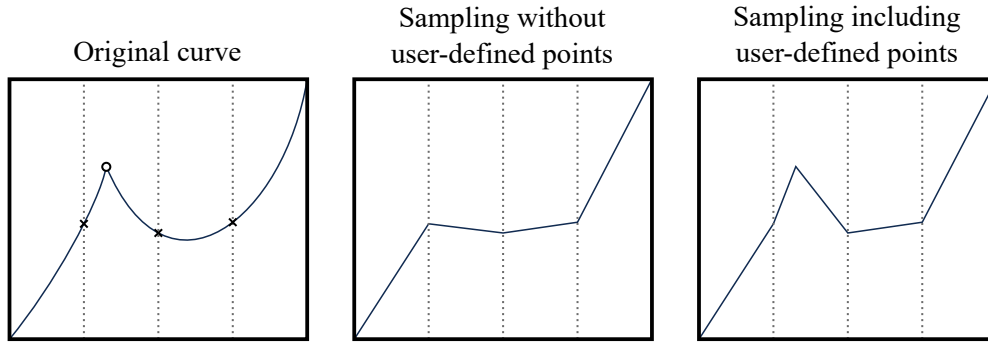


Figure 4.4: Visualization of an extreme case of how sampling at fixed intervals induces information loss, as the single user-defined point is completely ignored in the result. Adding the point to the curve’s path leads to it much closely resembling the original curve.

Curves – In some Node types, Blender offers a small graph editor, referred to as Curves. The user can define functions by dragging control points in a graph and selecting the smoothness of the curve. The Curves Widget takes the `curves` argument, which is a list of all the possible curves it should display. Each element of this list is a triple of:

- the color of the curve,
- coordinates of various points of the curve presented as (x, y) pairs,
- a boolean value of whether the space under the curve should be filled in.

The widget displays each curve as either a `polyline` or `polygon` element, depending on whether the area under the curves should be shaded.

To extract the relevant curves from a `CurveMapping` object, a wrapper function called `curve` is implemented. It is called with the property of the Node which features such a graph. It iterates over all the `CurveMap` objects defined in the given mapping, and uses its `evaluate` function to sample the value of the curve at intervals determined by the *Quality* property, described in Section 3.2.1. It then builds the required list of triples to pass to the `Curves` constructor depending on a `type` argument passed to the function.

However, the shape of a curve is defined by the points that the user places. The aforementioned process completely ignores these points, which would cause a significant error in the visualization, demonstrated in Figure 4.4. For that reason, after the list of (x, y) coordinates is obtained, the coordinates of these user-defined points are also added to it.

Unfortunately, the elements used to render these curves – `polyline` and `polygon` have their path defined in the same order as the coordinates in this list. Therefore, simply appending the user-defined point coordinates to the list would lead to an erroneous behavior of the line. For that reason, a `insertIntoSortedByKey` utility method is implemented. This method performs a binary search on a sorted array to find the right position for the user-defined point coordinates, sorting by their x coordinate.

Ramp – Certain Nodes use the Ramp element to map a value on the $\langle 0; 1 \rangle$ interval onto a color, similarly to how gradients are defined in editors such as Inkscape. Two arguments of this Widget, `color_mode` and `interpolation` only display the settings of the Ramp widget. The informational value is contained in the `evals` and `stops` arguments.

The `stops` argument contains the positions and colors of the manual stops that the user can place. However, the appearance of the color bar cannot be reconstructed simply by rendering gradients between two adjacent stops, as the interpolation of colors can be adjusted by the user. For that reason, the `evals` argument instead samples the color of the bar at intervals defined by the *Quality* export property. The color bar is then rendered either as rectangles of solid color for each sample (except the last one). If the *Use Gradients* export property is set, these rectangles instead have a gradient defined to smoothen the transition between adjacent samples.

This Widget is another example of having a wrapper function (`ramp`) called instead of the constructor itself to retrieve the relevant information from the `ColorRamp` type property of the Node.

String – The display of string data (as opposed to generic names) is handled by the String Widget, which renders a string (passed as the `value` argument) as a Label over a `rect` element. If a `name` argument is provided, the Widget is instead rendered as a FortySixty Widget.

Two more wrapper functions are implemented, `object` and `image` which construct String Widgets. The `object` function tries to retrieve the `name` property of a referenced object if one is defined, while the `image` function also adds a SelectBar Widget which represents the interface present if no image is selected in the property it models.

Plenty of other Widget types are implemented as subclasses of the String Widget without having any overriding definition of their own, as they are rendered in a similar fashion. If the differences (such as icons, or any major differences that Blender may introduce to the user interface) are to be implemented, it can be done here without having to replace constructors elsewhere in the code.

4.2 Node Specifications

As mentioned earlier, the `bpy.types.Node` object does not include nearly all the information required to recreate the appearance of the Node in SVG. For this reason, the add-on includes a large `NODE_SPECIFICATION` dictionary, implemented in the `categories.py` script.

Each entry in this dictionary is indexed by the Node's `bl_idname` property, which is consistent even if multiple Nodes of the same type are present in the graph. The entries are dictionaries themselves, and each contains at least one item: `class`. The value of this item is a string indicating what class the Node belongs to, which in turn informs the color of the Node header. Classes are distinguished by color in the Preferences menu, and match the categorization in menus such as the Add Node menu.

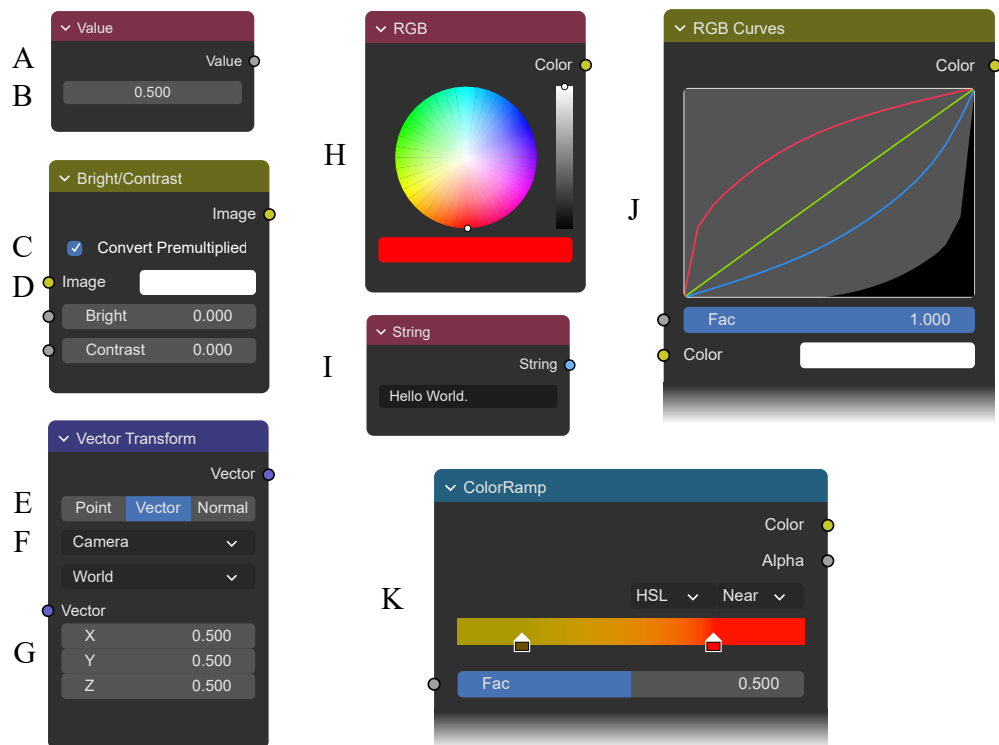


Figure 4.5: Examples of all listed Widgets across multiple Node types. A – Label, B – Value, C – Boolean, D – Columns with RGBA, E – SelectBar, F – Dropdown, G – Vector, H – Color Picker, I – String, J – Curves, K – Ramp.

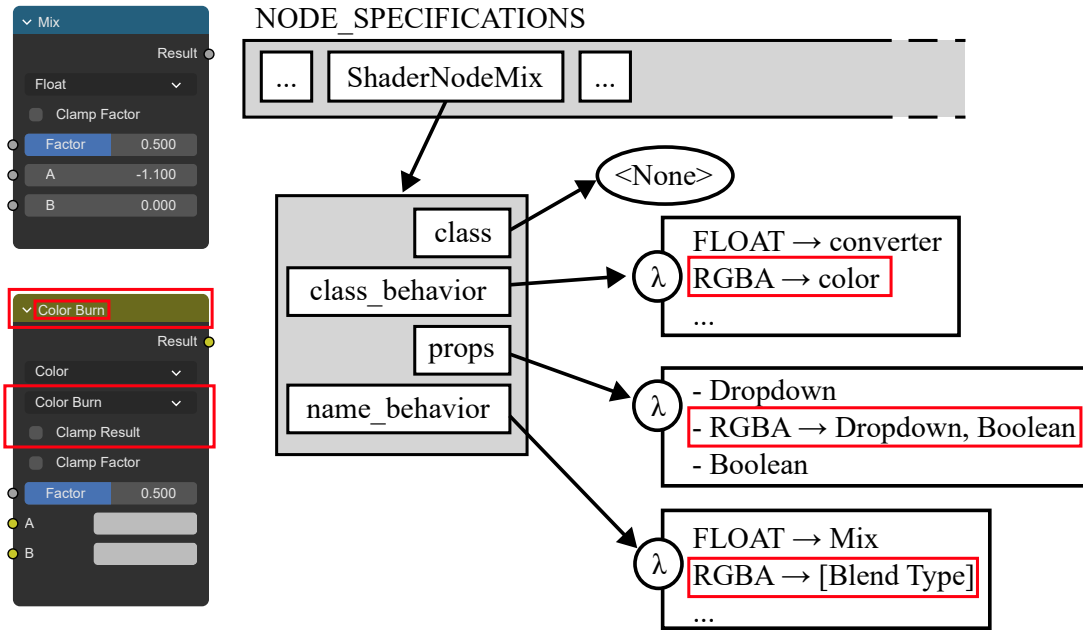


Figure 4.6: Examples of the Mix Node with different property settings, alongside a visualization of its entry in the `NODE_SPECIFICATIONS` dictionary (right). The entry is a dictionary itself, and specifies what properties should be displayed in its `props` entry. It also specifies that both the Node’s name and color can change (by having both a `class_behavior` and a `name_behavior` entry). The respective differences are highlighted in red. Differences in output and input sockets are accounted for in Blender’s internal representation.

For Nodes that contain any properties (that is, elements of the user interface that are neither outputs nor inputs), the `props` entry is also present in their dictionary entry. This entry contains a two-argument lambda function, designed to be called with one argument being the instance of the processed Node in question, and the other a dictionary of supplementing arguments regarding aspects such as element quality.

This lambda function always returns a list of initialized Widget objects (described in Section 4.1). While the body of a lambda function is an expression, and therefore the list cannot be constructed using the regular sequential code execution, Python’s ternary operators and the `*` list operator is used to achieve the variability of what properties are displayed based on certain configurations. An example is provided in Figure 4.6

Sometimes, property configuration can affect more than just what other properties are displayed. Certain Nodes change their name and/or category depending on their setting. For this reason, some entries in the `NODE_SPECIFICATIONS` dictionary include either or both of these items: `name_behavior`, and `class_behavior`, which contain a lambda function similar to the one under `props`, which returns either the name or the class based on the Node representation passed to it as an argument. An example of a Node whose entry in the dictionary has both of these attributes is `ShaderNodeMix` (commonly referred to as Mix in the editor). Its entry is shown in Figure 4.6.

4.3 Export Properties

To define various properties of the export process and the SVG output, this project utilizes Blender’s property structures, since Properties are persistent and can be easily added to and manipulated from the user interface. They are implemented in the `property_group.py` script.

For ease of access, the export’s Properties are all defined as class attributes of the `ExportPropertyGroup` class, which inherits from Blender’s `bpy.types.AddonPreferences` class. The `bl_idname` attribute of this class causes Blender to register these preferences into the `context` submodule, accessible specifically with this structure:

```
context.preferences.addons[<bl_idname>].preferences
```

Setting the `bl_idname` attribute to Python’s inherent `__package__` variable¹ simplifies the access to this property group, as the variable is consistent for all scripts within the add-on module.

The choice of which class represents what property partially comes down to how they are represented in the user interface. For example, properties implemented as the `BoolProperty` class are represented by a checkbox.

Sometimes, the appearance of the UI widget is affected by not just the Property class, but also its `subtype` attribute. This is the case of the `header_opacity` property, which has the ‘PERCENTAGE’ subtype, and is thus rendered as a slider rather than a regular numerical value. Likewise, for file location properties (such as the export target), the subtype of their `StringProperty` is set to ‘FILE_PATH’. This adds a button to the string field which opens the file selection modal window, and users then do not have to manually type in the desired location’s full path.

The colors are represented as `FloatVectorProperty`. This type has two subtypes that cause it to appear as a color widget: ‘COLOR’, and ‘COLOR_GAMMA’, which accounts for any gamma correction performed during display. However, simply setting the subtype does not ensure proper behavior of the widget. The size of the vector (given by the `size` attribute) has to be set to 3 or 4, with the latter adding the color’s alpha channel to the property. In addition to that, the values of the property must be limited by setting its `min` and `max` attributes to 0 and 1, respectively. Otherwise, it is almost impossible to select colors other than white and black in the color picker invoked when the user wants to change the color.

4.3.1 User Interface

Based on user feedback (from the survey described in Section 5.1), the location for the add-on’s user interface is in the Toolbar: a set of panels collapsed on the right side of a given editor.

The Toolbar is populated by registering classes which inherit from the `bpy.types.Panel` class, and whose `bl_region_type` property is set to ‘UI’. To avoid repeating the setting in every Panel implemented by this add-on, a superclass of Panels, `UIPanel` is implemented. It derives from the `bpy.types.Panel` class and contains the aforementioned property, as well as a `bl_space_type` property set to ‘NODE_EDITOR’, which ensures that it only appears in Node contexts.

¹As suggested by the example in the Blender API documentation:
<https://docs.blender.org/api/current/bpy.types.AddonPreferences.html>

Further abstraction is accomplished by defining a second generic class, `UIColorPanel`. This class inherits from `UIPanel`, but also contains a `bl_parent_id` property which designates it as a sub-panel of another Panel. Furthermore, it also implements a `poll` class method which returns whether the Panel should be drawn or not, depending on whether the user wants to specify custom colors of the output. All the color properties – text colors, header colors, element colors, and socket colors – are displayed in such Panels, and will disappear from the user interface if the *Use theme colors* option is checked.

4.4 Converter Initialization

All processing and creating an internal representation of the Node tree as class instances which return their SVG representation is contained within the `Converter` class. This class is implemented in the `converter.py` script. The class constructor is called with the `context` passed as an argument and prepares all the relevant intermediate `UINode` objects (described in Section 4.4.1). Then, calling the Converter’s `convert` method returns the SVG tree containing the representations of the converted classes.

When the Converter is constructed, it is given a reference to Blender’s `context` property, which is described in Section 2.1.2. Blender’s internal structure contains all of the Node graphs, but finding the one specific Node graph to export is done by referencing the `space_data` property that points to the current active editor. If the editor is a Node editor, which is guaranteed by exposing the operator in user interface defined to only appear in the Node editor, then the `space_data` property contains a `node_tree` pointer property referring to the relevant Node graph.

Furthermore, the Converter accesses the user-defined export properties. Their location within Blender is described further in Section 4.3. The Links are stored in the `links` attribute of the `nodetree` variable, and the Nodes in `nodes`.

4.4.1 Intermediate Classes

The interface between each Node’s internal Blender representation in the `bpy` module and their SVG representation in the output is the `UINode` class. Four classes are derived from this class: `UINodeRegular`, `UIHiddenNode`, `UIFrameNode`, and `UIRerouteNode`.

All of these classes are initialized with a reference to a `bpy.types.Node` instance representing the given Node in Blender. With the exception of `UIRerouteNode`, they all have their height and width defined in the `dimensions` property. However, this property has to be multiplied by a factor 0.8, as Blender overstates the dimensions of these Nodes compared to how they appear in the editor. They all implement an `svg` method which returns their vector representation to include in the add-on’s output.

The `UINodeRegular` and `UIHiddenNode` both inherit from `UINodeSpecified`, which is a subclass of the `UINode` class. This subclass implements value initialization pertaining to the Node specifications (described in Section 4.2), its display name, and header color. An overview of how the classes inherit from each other is depicted in Figure 4.7.

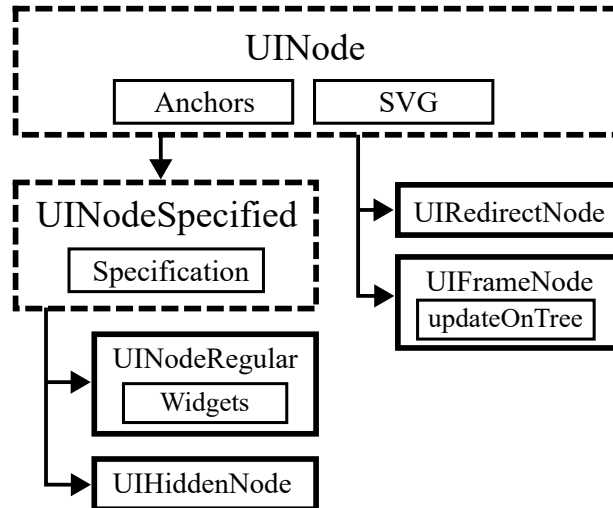


Figure 4.7: Overview of the inheritance structure of `UINode` and its subclasses, which act as an interface between Blender’s inner representation and the SVG output. The parent classes (with dashed outline) are virtual, and are not instantiated on their own within the add-on. Each class implements the ability to provide anchors (socket markers), and its own SVG representation. The `UINodeSpecified` subclasses utilize the `NODE_SPECIFICATION` dictionary from Section 4.2, and the `UINodeRegular` class works with Widgets. The `UIFrameNode` implements the `updateOnTree` method required to process nested Nodes, described in Section 4.5.3.

4.4.2 Registering Widgets

Since there is no difference between the appearance of sockets and individual Node properties, it only makes sense to represent them uniformly in the SVG output. During the initialization of the `UINodeRegular` object, the outputs, properties (from the `props` entry in the `NODE_SPECIFICATIONS` dictionary) and the inputs are iterated over to retrieve their Widget representations (as described in Section 4.1). The properties have their Widget representation stored in the Node’s library entry. For output and input sockets, a dictionary of lambda functions that return Widget constructors indexed by the socket’s `type` property is provided. A `register_widget` function is called on each Widget representation. This function adds a pair of the current `height` attribute of the parent object and the Widget representation to an array, and increases the `height` attribute by the Widget’s estimated height. If the total height of all Widgets exceeds the original Node height, the latter is adjusted to fit the content. In the opposite case, the Node height is left unchanged, as empty space within Nodes is not an uncommon sight in Blender.

When the `svg` function of a `UINodeRegular` object is called, the pairs are iterated over, and the SVG element representing the Node is extended by an `svg` call of each Widget, which includes the paired `height` value used for the `y` attribute of positioning the Widget.

4.4.3 Node Frames Initialization

The presence of Node Frames in the Node graph introduces additional complexity into the process. For one, the `location` property of Nodes nested within a Node Frame no longer gives its position within the whole area, but only within the Frame itself. Nodes

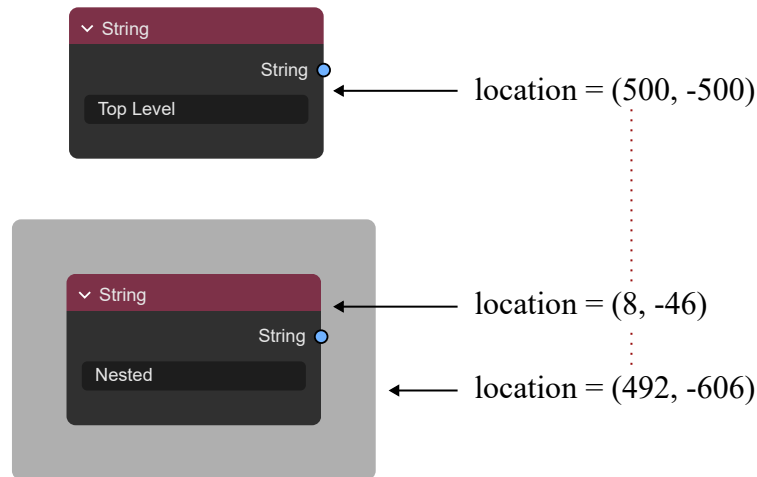


Figure 4.8: Example of how a Node's `location` property changes when it is added to a Frame. Both String Nodes should have the same x coordinate as they are above each other in the canvas, but the nested Node's `location` property becomes its relative position to the Node Frame. It is highlighted that the nested Node's absolute position can still be reconstructed by adding its relative position to the Frame's.

maintain a reference to Frames they are nested in through their `parent` property, and so their absolute position can be recovered by adding the absolute position of the Frame to their `location` property. This phenomenon is illustrated in Figure 4.8.

Unfortunately, Node Frames are coded as Nodes themselves, and can be nested in other Frames as well, meaning that finding the absolute position of a Node within multiple Node Frames requires recursively finding the absolute position of the parent and propagating the calculation back to Node. However, a Frame does not contain a reference to the Nodes within itself, complicating the task.

Furthermore, the dimensions of a Frame can be adjusted by moving a Node contained within it. This has the poorly-documented side-effect of removing any consistency between the value of the Frame's `location` property, and where it actually is positioned in the editor. Fortunately, this disparity is covered by the nested Nodes' own `location` properties, which make up for the difference and can still be positioned by taking their `location` property and adding it to the parent Node's absolute position. However, it makes reconstructing the Frame itself in the SVG output virtually impossible.

A Frame is subject to several constraints when it contains any Node. Its shape always encompasses the Nodes themselves with extra padding. For that reason, the minimal Frame can always be reconstructed from the contained Nodes themselves by finding their bounding box. However, if a Frame does not contain any Nodes, it is omitted from the exported SVG.

Two problems need to be solved at once: each nested Node needs to have its position updated based on the parent Frame, and each Frame needs to update its dimensions to the bounding box of the Nodes contained within. The solution is that during the first iteration over all to-be-exported Nodes, then Nodes without a `parent` property value are put in a `top_level` list, and those with a `parent` property are added to a `frame_children` dictionary, indexed by the pointer to the parent Frame.

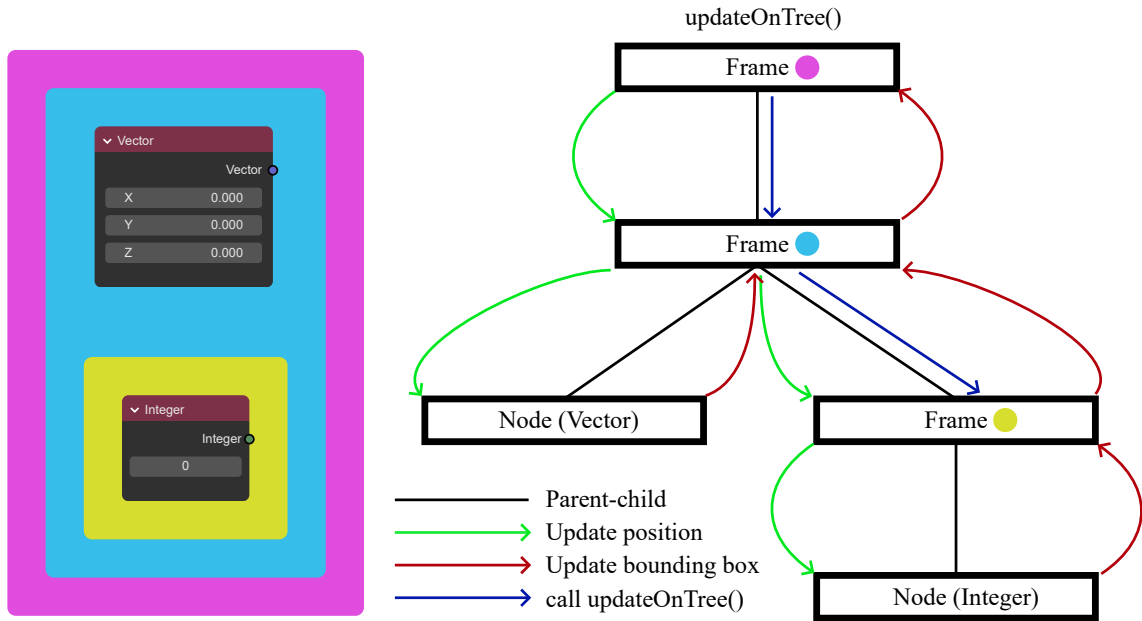


Figure 4.9: Diagram of how the properties of nested Nodes and Frames (left) are processed with the `updateOnTree` method. Each Frame iterates over its children (black), first updating their relative position with its own (green), call the `updateOnTree` method if the child is a Frame (blue), and finally update their own bounding box once the recursion is stepped out of (red).

Regular Nodes in the `top_level` property are passed to the Converter's `nodes` attribute, which contains all prepared representations of the regular Nodes for the export process. Representations Frames in the `top_level` property have an instance method called `updateOnTree` which references the `frame_children` dictionary. This method iterates over the Frame's children (described by the `frame_children` dictionary), and, for each, updates their position, then, if they are a Frame, it calls the `updateOnTree` method. Finally, the Frame updates its dimensions based on the bounding box of the child. This has to happen after the function call, since the dimensions of a Frame may change during its execution. Figure 4.9 is a diagram of the `updateOnTree` function executed on a more complex tree structure of nested Nodes and Frames.

4.4.4 Preparing Anchors

To position the small markers present with every socket, the `UINode` class has an `anchors` property. During initialization, each instance of this class populates this dictionary with the markers' positions with regards to the Node, and an instance of `UISHape` – an interface between Blender's internal representation of sockets and the markers in the SVG output. Entries in the dictionary are indexed by the `as_pointer()` value of each socket, which is used to render Links. The `anchors` property is retrieved from each `UINode` instance at the end of the `Converter` object's initialization using the `UINode` instance's `getAnchors` method, which returns the markers' absolute positions and `UISHape` representations.

The coordinates of an anchor in the `anchors` dictionary are relative to the Node's position, as the Node's absolute position may not be known during initialization due to



Figure 4.10: Example of two Nodes – Combine XYZ and Brick Texture – which have different amount of sockets. Blender automatically increases the size of hidden Nodes to fit all of their socket markers along the Node’s round edge.

Node Frames. Instead, if the positions of the anchors need to be known, they are accessed through the `getAnchor` method that returns the dictionary but with absolute positions.

For instances of the `UINodeRegular` class, the anchors are initialized during the Widget registration process, using the current cumulative height of all registered Widgets so far. For instances of the `UIHiddenNode` class, calculating the anchors’ position is different. Hidden Nodes render socket markers along their edges, and the Node’s dimensions themselves adapt to the amount of displayed markers². Figure 4.10 showcases this phenomenon.

The markers are spaced along either semicircle that makes up the Node’s shape in regular intervals, splitting the half-circle into $N + 1$ equal arcs, where N is the number of markers. Thus, the marker’s position in relation to the center of the semicircle that is the respective edge can be calculated using the polar coordinates (ρ, ϕ) . This is similar to finding the coordinates of the cursor on a Color Picker Widget, as described in Section 4.1.

The radius ρ is consistent and equal to the radius of the semicircle. The angle ϕ of the n -th input socket marker out of N is be found as follows:

$$\phi = \frac{\pi}{2} + n \cdot \frac{\pi}{N+1} \quad (4.1)$$

Analogously, since they appear on opposite sides, the polar coordinates of the output socket markers would be found by subtracting the $n \cdot \frac{\pi}{N+1}$ element instead of adding it. An example of this calculation for input sockets is shown in Figure 4.11. After the polar coordinates are found, they are converted to Cartesian coordinates just like with the Color Picker widget.

Instances of the `UIRedirectNode` class (which represents Reroute Nodes) define only a single marker, with the coordinates $(0,0)$. This marker is, in fact, the only visual representation of the Reroute Node. Instances of the `UIFrameNode` do not define any markers, as Frame Nodes do not feature any sockets.

4.5 SVG Output

Once the `Converter` object is initialized, its `convert` method is called. This method returns an `svg` element which contains the SVG representation of all the Nodes and Links. The output SVG file can be broken down into separate elements. This section details their individual implementations. The overall structure of the output as already outlined in Section 3.1.

The Node Frames are defined first, simply as `rect` elements, optionally with a `text` element on top. Then, the wires are rendered as pairs of `path` elements (one as outline, another as the wire itself, bearing the relevant color). The Nodes themselves are rendered

²Fortunately, the adaptive size is reflected in the Node’s `dimensions` attribute, thus there is no need to calculate the Node size manually.

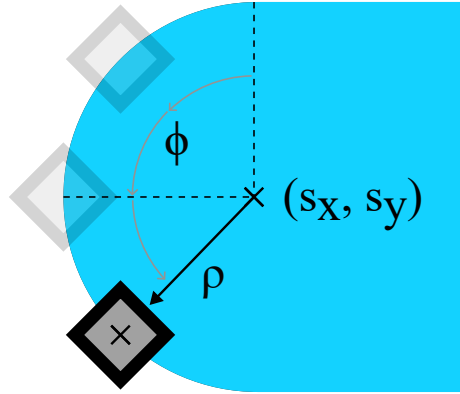


Figure 4.11: Showcase of how the coordinates of a Node’s third input socket marker are found. The radius ρ is consistent and equal to half the hidden Node’s height. The angle ϕ is found as the sum of gaps between the markers (added to the starting angle of $\frac{\pi}{2}$, not pictured). After converting the polar coordinates to Cartesian coordinates, they still need to be added to the point that is (s_x, s_y) to find their relative position to the Node itself. Both coordinates of this point are, in fact, half the Node’s height.

over the wires. Nodes are represented as groups of elements that include the rectangle making up the Node’s shape, its header, and the individual sockets and properties (in case of a regular Node). Finally, the markers are added on top, as they are not affected by the clipping frames of the Nodes to which they belong.

4.5.1 Parent element

The vector representation is contained within a parent `svg` element. One of said element’s attributes is the `viewBox` attribute detailed in Section 2.2.3 which defines the canvas area rendered into the user space.

Since Blender and SVG use mutually inverse vertical coordinates, the individual objects’ `location` attribute is multiplied by -1 to get their position in the SVG output. Besides that, no adjustment is made to the location of individual objects within the SVG. Instead, the objects’ position within the canvas is guaranteed by setting the aforementioned `viewBox` attribute to transform the objects into the canvas. The boundaries of this box are determined by the upper left and lower right corners of the most extreme Nodes as well as calculated extrema of wires (described in Section 4.5.4) that connect an output to an input positioned to the left of the output (in essence twisting the wire backwards). Finally, the dimensions are adjusted by a fixed padding value.

4.5.2 The `defs` element

The single `defs` element of the `svg` element contains a CSS style definition of the final document, as well as definitions of symbols that are reused throughout the document.

The text content of the `style` element is a block of CSS styling rules, described in Section 2.2.4. These rules are written using all the different user-defined properties of the

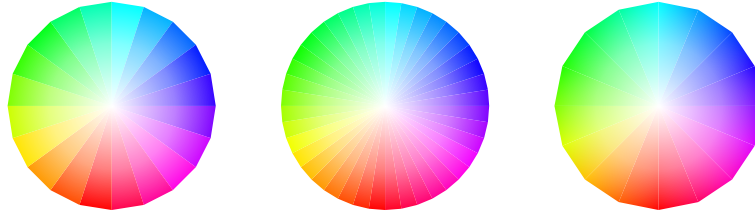


Figure 4.12: Examples of the Color Wheel symbol rendered at different quality settings: *Quality* of 10 with no gradients (left), *Quality* of 20 with no gradients (center) and *Quality* of 10 using gradients (right).

export process and describe the appearance of individual elements. The elements themselves thus only need a `class` attribute which is independent from the export properties, meaning that the styling attributes do not need to be passed down through function calls, and are instead handled by the rendering engine.

The contents of the `style` element extend to setting the `fill`, `stroke` and `stroke-width` attributes, and uses the cascading property of CSS to address `text` elements differently if they are children of specific widgets. The `symbol` elements defined in the `defs` element consist of definitions of the anchor markers, arrows, the color wheel and the gradient used for the Hue Correct Node.

The Color Wheel is one of the widgets affected by the *Quality* property. Due to its complexity and the limitations of the SVG format, its appearance is only approximated. The widget is made up of wedges: triangles (defined by the `polygon` element), each with one vertex in the widget's center. The positions of outside vertices is calculated by iterating over the full angle ($2 \cdot \pi$) in $2 \cdot \text{quality}$ steps, and finding the position of a point that far along a circle with a fixed radius. Examples of how the *Quality* property affects the appearance of the Color Wheel is provided in Figure 4.12. The angle also helps determine the color of the wedge, as the proportion of how far along the circumference the point is corresponds to the Hue property of the HSV color model.

If the *Use Gradients* option is enabled, a custom `linearGradient` element is added before each wedge, defined by the colors in both outside vertices, and adjusted so that it ('its guiding vector') spans the edge between said vertices. An example of using gradients is provided in Figure 4.12

Finally, a circle filled with a radial gradient of solid white color fading out towards the edges is added on top of the wedges, to simulate the colors losing saturation towards the center of the color picker. This also helps hide some of the more apparent differences between wedges' colors if the *Use Gradients* option is enabled.

As mentioned in Section 2.2.3, editors such as Inkscape struggle with certain gradient definitions. The radial gradient will not function if simply defined as a `radialGradient` element with only stops. If one remakes a similar circle in Inkscape, they find that the gradient is actually defined by a `linearGradient`, and the `radialGradient` element referenced by the `fill` attribute of the overlaying circle instead references the `linearGradient` element without containing any `stop` elements of its own.

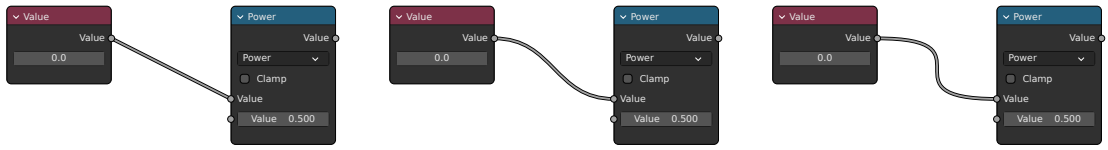


Figure 4.13: Examples of exported wire shapes with the 'Noodle Curving' property set to 0 (left), 5 (center) and 10 (right).

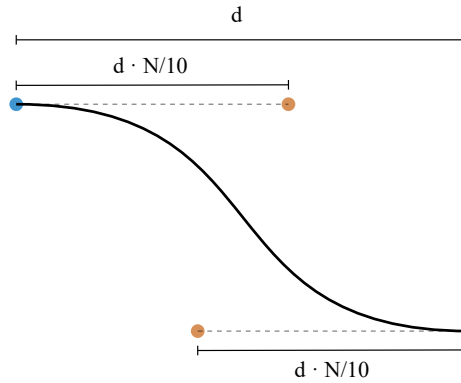


Figure 4.14: Example of how the control points' (orange) position is calculated based on the start and end point (blue) with a Noodle Curving factor of $N = 6$.

4.5.3 Node Frames

All of the problems with Node Frames are solved during the initialization phase. When the output is generated, the representation of Node Frames is generated from the respective `UIFrameNode` objects as plain rectangles, potentially adorned with a `text` element if they have a label set.

Since Node Frames are always rendered behind regular Nodes regardless of whether the Node belongs to the Frame or not, they are defined separately in the SVG output. Otherwise, elements of the graph could be logically grouped in `g` elements where each would represent one Node Frame and its content.

4.5.4 Wires

Wires are rendered as Cubic Bézier curves. Bézier curves are a mainstay of computer graphics because of their ease of comprehension and stability. Cubic Bézier curves define a curve using four control points, and describe the curve as a weighted average of these four points, with weights defined for each value of variable t on the interval $\langle 0; 1 \rangle$ [13, ch. 15].

In this case, the curve's first and fourth control points (start and end point, respectively) are set to the anchors' coordinates. The positions of the second and third control points are calculated by offsetting the first and fourth control point positions, respectively. The offset is a product of the horizontal distance between them and the `noodle_curving` property in the current Theme, divided by the maximum value of 10. Different values of the `noodles_curving` property are shown in Figure 4.13, and the calculation of the position of control points for the value of 6 is shown in Figure 4.14.

Since wires can connect outputs to inputs that are further to the left on the canvas, they will extrude out beyond the bounding box of the Nodes they are connecting, and

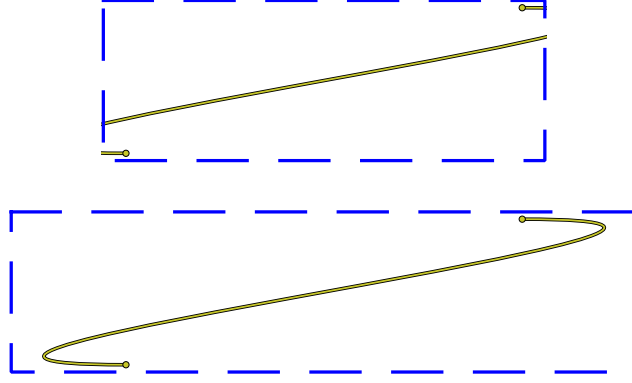


Figure 4.15: Demonstration of the `svg` element's `viewBox` property (blue rectangle) when not expanding it by finding the Bézier extrema (upper) and when accounting for the curving (lower).

potentially beyond the bounding box of all Nodes. To prevent the parent `viewBox` property from clipping such wires, the box is adjusted by the wires' extrema, which are calculated analytically during the conversion. Figure 4.15 illustrates this problem and its solution.

Since the wire never goes outside the vertical range delimited by its input and output, only the horizontal coordinate needs to be considered. For that reason, the Cubic Bézier curve equation (adapted from [13, ch. 15]) can be adjusted to only yield the horizontal coordinate x , like in Equation 4.2.

$$x(t) = (1 - t)^3 x_0 + 3 \cdot (1 - t)^2 t x_1 + 3 \cdot (1 - t) t^2 x_2 + t^3 x_3 \quad (4.2)$$

The extrema are clearly points where the rate of change of the x coordinate goes from positive to negative, or vice versa. Therefore, the desired points are those where the derivative of the Equation 4.2 is equal to zero. The derivative of a Cubic Bézier curve, shown in Equation 4.3, is a quadratic equation with unknown t , shown in Equation 4.4. Quadratic equations can have zero, one, or two solutions. In this case, the presence of two solutions is controlled for by only performing the calculations on wires where it can be proven that they bend backward.

$$x'(t) = 3 \cdot (1 - t)^2 (x_1 - x_0) + 6 \cdot (1 - t) t (x_2 - x_1) + 3 \cdot t^2 (x_3 - x_2) \quad (4.3)$$

$$(-3x_0 + 9x_1 - 9x_2 + 3x_3)t^2 + (6x_0 - 12x_1 + 6x_2)t - 3x_0 + 3x_1 = 0 \quad (4.4)$$

The implementation features a `solveQuadratic` function, which takes in arguments `a`, `b` and `c` representing the multipliers of t^2 , t^1 and t^0 (omitted as it equals 1) in Equation 4.4. This function is part of a greater function, `getBezierExtrema`, which takes a single coordinate of each control point (`x0` through `x3`) and returns the coordinate at which the curve reaches its local extrema. These two values can then immediately be compared to the dimensions of the parent element's `viewBox` property, and pad it to make it fit the wire.

Of course, if the `noodle_curving` property is set to 0, no such calculations have to be performed.

4.5.5 Nodes

Each Node in the currently displayed Node Graph (with its `select` attribute set to True if exporting selected Nodes only is enabled) is represented as a `g` element with a unique `id` attribute based on the unique identifier that Blender assigns to individual objects. The unique identifier is required for the contents of the element to properly reference its unique clipping path. The position of the Node is also defined by a `transform` attribute (described in Section 2.2.3). The values of this attribute are taken directly from the Node object's `location` attribute, or, if the Node is inside a Node Frame, adapted based on its location (as described in Section 4.5.3).

The immediate children of this group element are a `clipPath` element, consisting of a single `rect` element, and another `g` element which holds the Node's visual representation and references the aforementioned clipping path through its `clip-path` attribute.

Headers

The Header of each Node is defined as three elements; a `rect` element that covers the top stripe of the rendered Node, an arrow, and a `text` element which carries the Node's name. The Node's name is determined by the Node object's `name` property, a custom `label` property, or a specification in the `NODE_SPECIFICATIONS` dictionary.

The Header transparency is defined in the Export Property Group (described in Section 4.3). The color is either retrieved from the Property Group as well, or the application Preferences if the `use_theme_colors` property is enabled.

The process of building a Node's SVG representation is implemented in the `node.py` script. A regular (neither Reroute nor Frame) Node that has not been collapsed is represented by an instance of the `UINode` class, and is initialized with a reference to the `Node` instance in Blender's runtime.

Based on the Node's `bl_idname` attribute, it is assigned a `specification` object from the `NODE_SPECIFICATIONS` dictionary. Some of the output object's properties are defined during initialization, such as the Node's display name. Blender will append a three-digit number³ to the `name` of multiple Nodes with the same display name even if their display name is the same. This is fixed by a simple regular expression match, which detects if the name ends on three digits, and removes those last three characters if that is the case.

The `UINode` object is also assigned an `id` attribute to distinguish its SVG elements in the output file. This is taken from the original `name` attribute of the Node, including the unique numbers.

4.5.6 Markers

The markers representing sockets are added after all Nodes, and are therefore rendered above them. They are not part of the Node groups themselves as they would be subject to the clipping path that trim the Nodes down and optionally round their corners off.

As described in Section 4.4.4, the `Converter` object compiles all the marker information at the end of initialization. As part of the `convert` method, it extends the parent group by calling the `svg` method of each marker's `UIShape` representation.

The canonical marker shapes (circle, diamond, square) are all defined as symbols, as are the dots that may appear in their center. The marker's representation in the SVG output

³In case of having more than a thousand instances of the same Node type, Blender will append a four-digit number. However, this case is so extreme, it is not addressed in the script.

is thus guaranteed using a `use` element. Each socket has a `display_shape` attribute that defines the marker's appearance, which is now used to determine which symbol to refer to with the `use` element's `href` attribute.

The `use` element also has a `class` attribute that ensures that the marker will appear in the appropriate color thanks to the CSS style sheet included in the output. Furthermore, the `use` element is also wrapped in a `g` element, whose `transform` attribute determines the position of the marker in the final SVG.

Chapter 5

Evaluation

The scope of this project goes beyond design and implementation. This chapter goes into the details about testing the implemented add-on, and mentions how the add-on is distributed to users. In this chapter, the add-on’s output is described and evaluated against its source. Differences, intentional or not, are explored. It also contains measurements related to file size and export time. Furthermore, the chapter describes the distribution and usage of the add-on.

5.1 Research Survey

As mentioned in Section 3.2, a research survey was conducted to poll the potential user base about their experience with Blender and academic writing. A link to the survey, as well as its results, is listed in Appendix A.

First, the respondents were asked what they use Blender for and in what context they use it. Furthermore, they were asked what types of add-ons they use (of the three described in Section 2.1.1), and encouraged to name any add-ons they use on a regular basis.

They were then shown an edited screenshot of Blender’s 3D Viewport editor, seen in Figure 5.1. Four areas of the user interface were highlighted and respondents were asked to rate how much they would expect to find the controls of an add-on in the proposed areas.

The questionnaire then focused on Blender’s Node editors. The respondents were asked to select which editors (of the three editors described in Section 2.1.4) they had experience with and which Node functionalities they used. These functionalities included common examples of hiding Nodes, but also more advanced features such as Reroute Nodes, Node muting, Node Frames and Node groups. To collect data on commonly used Node types, the user could also submit a screenshot of a Node graph they used in one of their projects.

The last part pertaining to the Nodes showed the respondent the proposed user interface mock-up (seen earlier in Figure 3.3), and asked the respondents how they would interact with it with specific goals in mind: changing the output format or size, or explicitly exporting a specific Node to a given paper format.

The section of the questionnaire asking about writing technical reports was shorter and focused on the respondent’s experience with importing and setting vector images. In particular, it asked about their preferred formats, what technologies they use to import them (such as $\text{\LaTeX}$ packages), and if they ran into any trouble in the process.

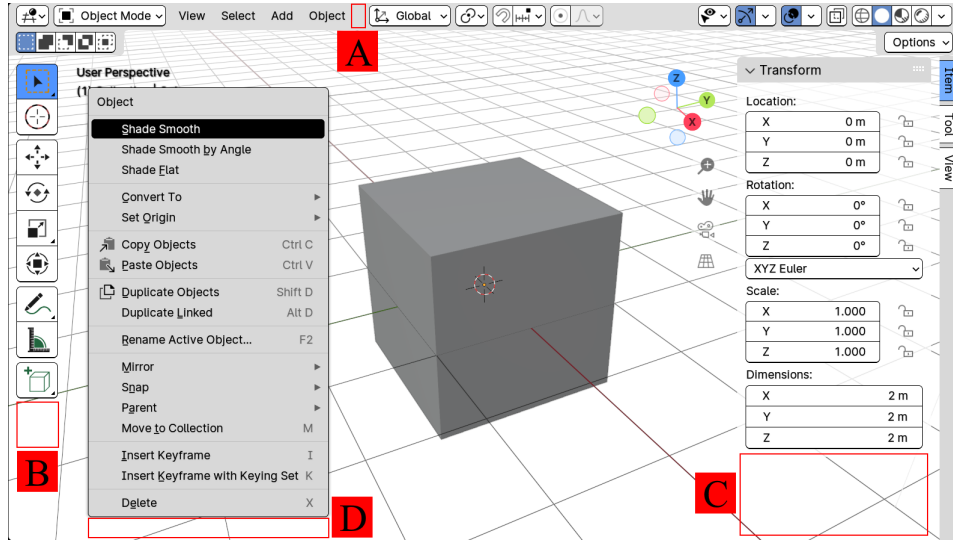


Figure 5.1: An edited Blender screenshot to poll the user base as to where they would expect to find a given add-on. The editor’s color scheme and aspect ratio has been altered from how the screenshot appears in the form, but its content remains the same.

5.2 Quality Evaluation

Due to missing detailed data and similar inconsistencies in the `bpy` module, there is little point in performing an exact pixel comparison between the Blender depiction of Nodes and their SVG representation. Instead, individual differences between certain elements and their representation are taken into account in this section.

5.2.1 Differences

As discussed throughout this thesis, imperfections exist in the representation of the Node graph. These can be traced to various sources. It is mainly the limitations of the `bpy` module in terms of providing all the required information. Other times, it is due to the complexity of certain UI elements, especially when there’s little need to perfectly copy their appearance. Finally, certain differences may appear as a result of minor (perhaps even undocumented) changes between different editions of Blender. Below is a list of the few most notable ones:

Normal Widget – Sometimes, a vector value will not be displayed through the usual triplet of value bars, but instead through a sphere reflecting incoming light. In the SVG output, this is still represented as the three values. This is due to the lack of indication that a socket actually uses this display method, the fact that the socket would be difficult to recreate using SVG’s elements, and the low frequency of occurrence of this phenomenon. The difference is showcased in Figure 5.2.

Socket Dropdowns – In more modern versions of Blender, the Principled BSDF Node uses dropdown elements to hide groups of sockets from the user interface. However, this change is not reflected in the available internal representation (neither the presence or dropdowns, nor hiding the affected sockets). Since little information loss is caused by displaying more data than the user interface and this feature is present in

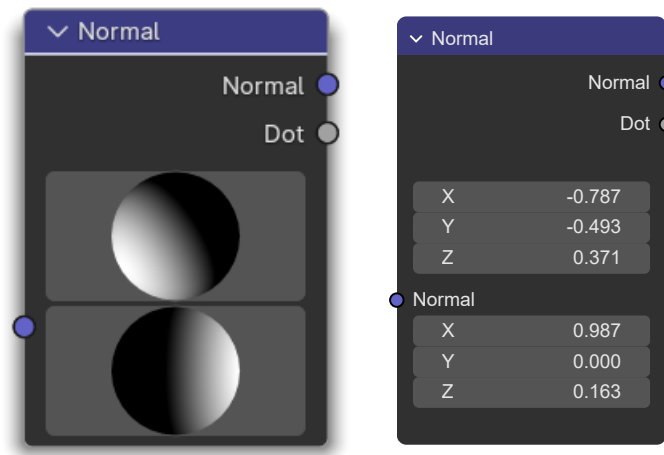


Figure 5.2: Comparison between the Normal Node as showcased in Blender (left, screenshot) and how the add-on represents it in SVG (right).

only one Node type so far, the add-on does not account for this feature whatsoever and displays the sockets that are made available. The comparison is showcased in Figure 5.3.

Object Widgets – Many Node properties refer to an existing data block in Blender (such as an object or an image). Blender’s own UI elements tend to differ in small ways (such as adding a button) based on the type of this object. However, at minimized loss of information value, the need to design special Widgets for each object type is circumvented by rendering these properties using String Widgets, defaulting to displaying the object name (if any is even provided). An example of this behavior for the Image widget is shown in Figure 5.4.

Graphs – Certain Nodes let the user design value mapping through the use of graphs. The curves of these graphs are represented in the output by sampling the curve value at predetermined intervals (described in Section 4.1).

As an example where Blender’s immediate representation of the Nodes contains less information value than the SVG output, Blender only displays one of multiple curves at a time if the Node supports multiple curves (such as RGB Curves). In the Output, the curves are rendered atop each other, starting with any that has the `infill` attribute set to True. Each curve is rendered by a different color, defined by the *Axis X*, *Axis Y* and *Axis Z* colors in either the add-on preferences, or in Blender’s Theme menu. Example of how the SVG output displays all curves at once when Blender would only display one is provided in Figure 5.5.

Node Frames – As mentioned in Section 4.5.3, Node Frames behave inconsistently with regards to their location in the editor. Since the add-on cannot reconstruct the Frame’s size and position in the output, it instead defaults to rendering Frames as padded bounding boxes of their content, and not rendering empty Frames whatsoever. An example of both phenomena is shown in Figure 5.6.

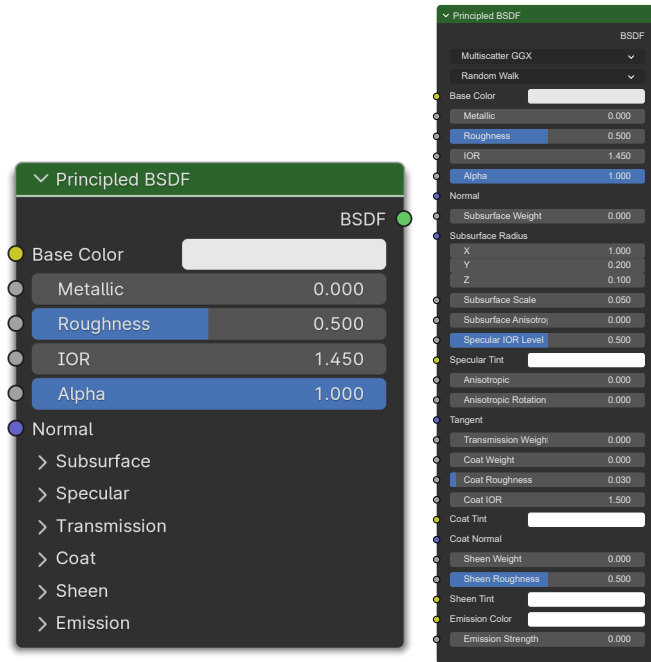


Figure 5.3: Comparison between the modern depiction of the Principled BSDF Node (left, screenshot) and how the add-on represents it in SVG (right).

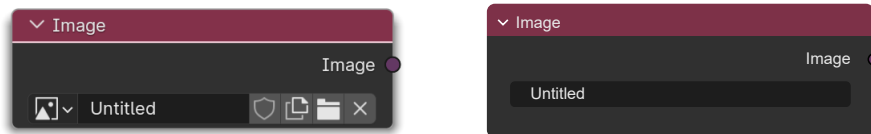


Figure 5.4: Comparison between the full Image widget in Blender's user interface (left, screenshot) and how the add-on simplifies it (right).

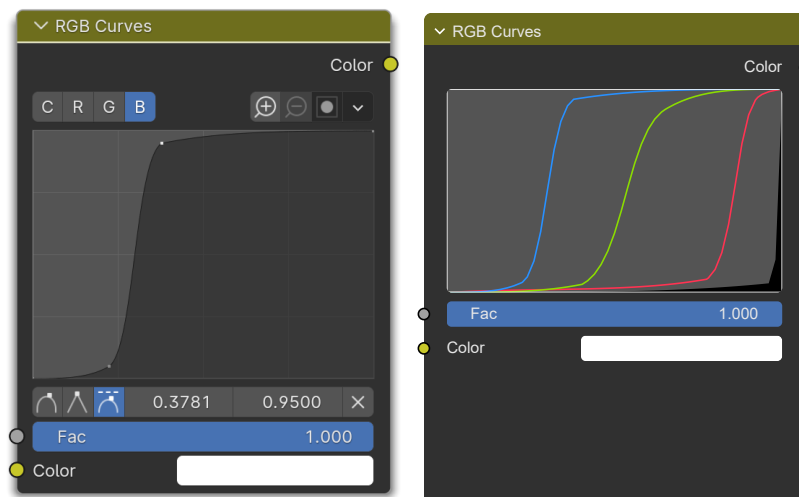


Figure 5.5: Comparison between the Curves Widget in Blender's user interface (left, screenshot) and how the add-on displays all curves at once instead of just one (right).



Figure 5.6: Showcase of how the add-on handles Frames. Originally, the largest Frame contained the Node and an empty Frame (left, screenshot). However, the add-on removes empty frames entirely, and determines Frames’ position and size based on their content to produce the output (right).

5.2.2 User Testing

One of the stated goals of the add-on is conveying enough information to let users re-create Node graphs just from seeing the SVG output. A user test is conducted to verify this through renders of a 3D scene after the user has re-created the graph in their application.

The test is a two-part process. First, the tester is provided a Blender project which contains a Node graph (seen in Figure 5.7). They are tasked with exporting two versions of it. First, only the three central Nodes, excluding the Principled BSDF and Material Output Nodes. The tester is thus required to find not only the Export button and target file setting (as a name is specified for each export), but also recognizing the use of the Export Selected Only setting.

The second part of the test lets the tester choose any of the three primary Node editors. For each they want to test, they can download an image of a Node graph created in said editor, and exported by the add-on. Their goal is then to re-create the graph in Blender.

After creating the respective Node graph, the tester is asked to create a render of their scene (without manipulating with anything else), and saving the render. The result of this test is therefore an image, which can be processed and compared against the reference image to obtain quantitative result scores. The pairs of reference Node graphs and their resulting renders is included in Appendix B.

In the end, four users tested the protocol and provided their feedback. The users generally had no problem with the first part. The most common negative feedback in this section was the lack of tooltips they could use to verify their assumptions (which they were making correctly). In fact, the part that most testers got stuck on was installing and locating the add-on, which was addressed in the documentation supporting the published add-on.

The second part also yielded positive results because, at a glance, all the resulting renders showed that the testers successfully remade the Node graphs. However, it was quickly discovered that small differences in testers’ Preferences (such as Color Management) could incur errors despite the graph being an exact reconstruction of the sample the tester was given.

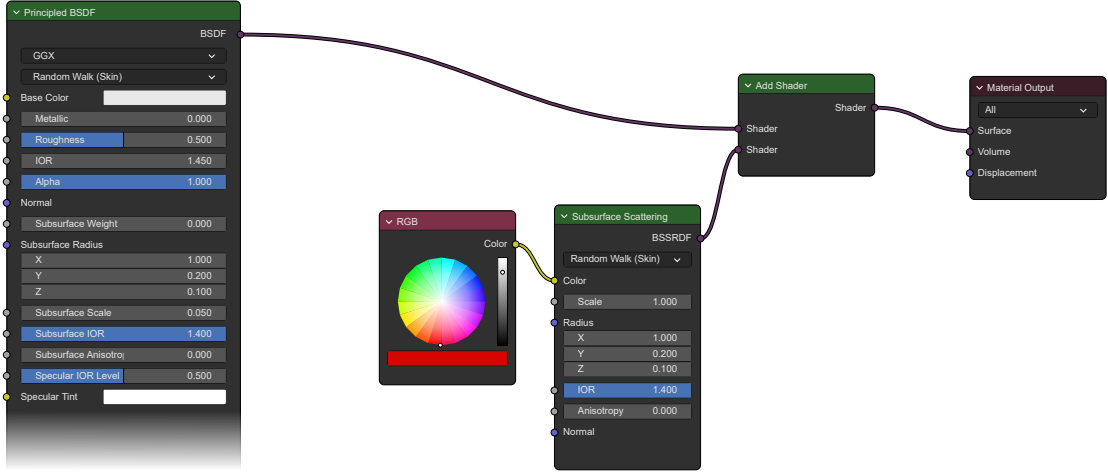


Figure 5.7: The Node graph in an example Blender project that users work with during the first part of the test. The Principled BSDF Node has been cropped, its full appearance is shown in Figure 5.3.

	Shading	Compositing	Geometry
00	0.1%	0.0%	0.0%
01	1.8%	1.8%	1.8%
02	0.6%	-	-
03	1.8%	-	-

Table 5.1: The SSIM comparison results for each tester-submitted image. Since the mean SSIM for two identical images is 1, the error is presented as the percentage left after subtracting the *MSSIM* result from 1. Notable are the results of Testers 01 and 03, who attained a higher error.

For that reason, the error of user results when compared to the reference images is measured using the Structural Similarity Index Measure (SSIM). This metric reduces error incurred by details such as every pixel being a little brighter than in the reference image, and instead proposes a human perception-centric model of gauging the difference between two images [15]. The comparison is implemented in the `compare.py` script included in the directory, and uses the `scikit-image` implementation of the comparison method¹. Specifically, the mean SSIM is found for each tester image.

After manual review, it was found that the highest-error images differ from the reference images by being brighter. Even if every tester works on the same source file, this is an issue that can show up and introduce errors, lessening the ability to quantify the tester’s performance based on the output alone.

Besides that, the only recorded issue amidst testers was found in the Shader Node graph. Specifically, users did not know which exact color is displayed in the image. Most assumed it was the RGB value of (255, 0, 0). Even if the user uses a color picker tool on the resulting image, Blender and various applications perform color correction operations (similarly to what might have caused the brightness error). This prevents the user from getting the same

¹as showcased by Daniel Yoga here:
<https://medium.com/@danielyogatama.dy/ssim-on-python-eb1a76a2799b>

Node Type	Lines	Characters
<i>Shading</i>		
Principled BSDF	814	45245
Principled Volume	393	22983
Brick Texture	388	20409
<i>Compositing</i>		
Corner Pin	426	23959
Keying	425	22603
Color Spill	312	18102
<i>Geometry</i>		
Bézier Segment	441	26744
Quadratic Bézier	326	19943
Volume Cube	345	18687

Table 5.2: Largest Nodes by representation of each Node editor. Duplicates were removed (Brick Texture also reappears in Geometry editor), and Nodes affected by the *Quality* property were also removed from the table as their size is variable.

color which was present in the Blender Node editor when the vector image was exported. A new way to display colors could be a topic for future development of this add-on.

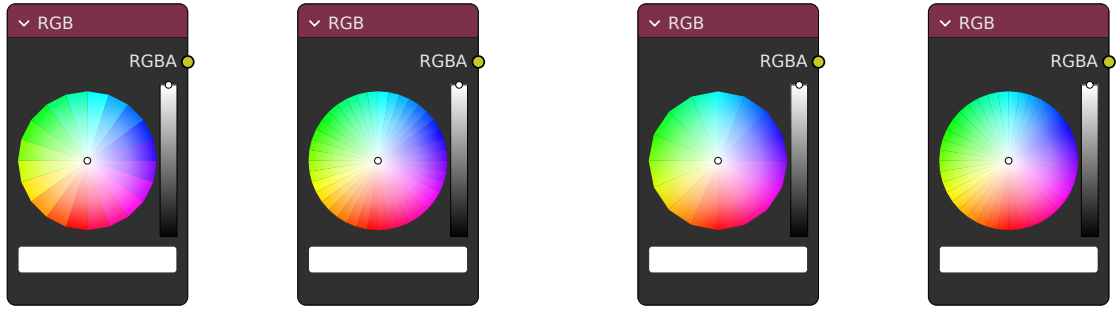
5.3 Measurements

Two more metrics that can be evaluated for the application are the following: the size of the output depending on the content of the graph, and the time it takes to produce it.

5.3.1 Output Size

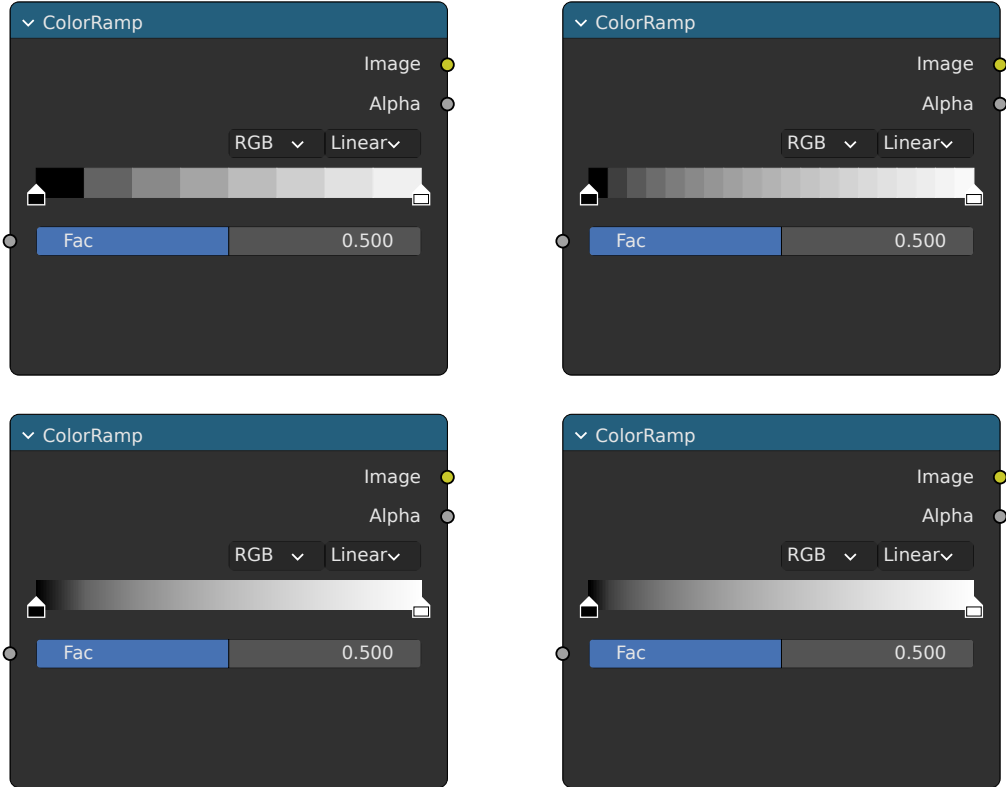
To gauge the sizes of the SVG representations of different Node types, a script was created to create a standalone representation of each selected Node. Instead of exporting the representation to a file, instead, the `defs` element is removed, and the remaining SVG is indented and converted to a string. The character and line count of this string is chosen to represent these Node types. The largest Nodes of each editor (save for Nodes affected by the *Quality* property) are listed in Table 5.2.

Adjusting the *Quality* property also affects the size of an affected Node’s representation. Additionally, turning the *Use Gradients* option on has different effects on different widgets. For the Color Picker Widget, it adds a `linearGradient` element to each polygon that makes up the color circle, effectively doubling the amount of elements whose count is already affected by the *Quality* property. Conversely, with Ramp elements, it replaces the partitioning of the color bar into smaller triangles, and only applies a single `linearGradient` element to a rectangle which spans the entire bar. The differences in appearance and sizes for the RGB Node are given in Figure 5.8, and for the Color Ramp Node in Figure 5.9. In both cases, the `defs` element is included in the count.



	$Q = 10$, no grad.	$Q = 20$, no grad.	$Q = 10$, grad.	$Q = 20$, grad.
Characters	8552	12140	13762	25354

Figure 5.8: Visual and size differences between different quality settings for the RGB Node. The *Quality* property (represented by Q) is set to 10 and 20, and Nodes are shown both without (left) and with (right) the *Use Gradients* option enabled. Table values are given in character count, this time including the `defs` element as it changes with the properties.



	$Quality = 8$	$Quality = 20$
No Gradient	14801	20355
Gradient	19514	31871

Figure 5.9: Visual and size differences between different quality settings for the Color Ramp Node. The *Quality* property values are 8 (left) and 20 (right), and bottom two Nodes have gradients enabled. Table values are given in character count, this time including the `defs` element as it changes with the properties.

5.3.2 Export Time

During regular usage of the add-on, the export process is almost instant, so there is no pressing need to optimize the add-on performance. For testing purposes, however, the duration of the export is measured regardless to see if there are any lurking threats of high computational complexity. The duration measurement as well as add-on profiling is done using the `cProfile`² module.

For the purposes of the first round of measurements, the Node graph provided for the Compositing part of the user testing (described in Section 5.2.2) was used as reference. Measurements done for Node Frames are performed on their own, special Node graphs.

Profiling

To see how much time each part of the add-on takes to execute, profiling is performed on the add-on. Statistics found using the `cProfile` module include the number of calls and the total and average time spent in execution of a code block. The latter two are also found both for the time spent in execution of the code proper, and including time spent in execution of methods called from within said block.

Finding the proportion of execution time between various methods was done on two examples: exporting a single Node graph comprising of six Nodes, and exporting a graph of 80 copies of those six Nodes.

The charts in Figure 5.10 show the proportion of times taken to execute the 5 most time-consuming methods in each case. Notably, while the top three methods (the `svg` method for Widgets, indentation for formatting the SVG output, and the `convert` method which iterates over all Nodes during execution. Notably, however the fourth and fifth most time-consuming methods change between the two examples. One is the `makeDefs` method which is called exactly once during each export process, the other is a list comprehension method for converting colors, which is also disproportionately called from the `makeDefs` method. In the second chart, they are replaced with two `fill_svg` methods which belong to the Value and Boolean widget.

Regular Graphs

The charts in Figure 5.10 suggests that there is no method whose complexity would cause its execution time to greatly increase with raising the number of Nodes. To gauge the execution times based on the Node graph sizes, it was measured across copies of the same six-Node graph.

Charts in Figure 5.11 showcase the execution times over Node graphs of different sizes. It appears that there is a linear relationship between the Node count and execution time. This does not prove that the execution times do not suddenly begin exceeding these bounds at higher Node counts. However, the measurements were done up to 960 Nodes, which is an extreme number of Nodes for a graph to have, especially when Blender offers so many features for the user to keep their graphs tidy (described in Section 2.1.4).

Finally, the time to export N copies of the Node graph was tested on logarithmically increasing numbers of 5, 10, 20, 40, 80 and 160 copies. For these measurements, the process was only timed over 100 iterations.

²<https://docs.python.org/3/library/profile.html>

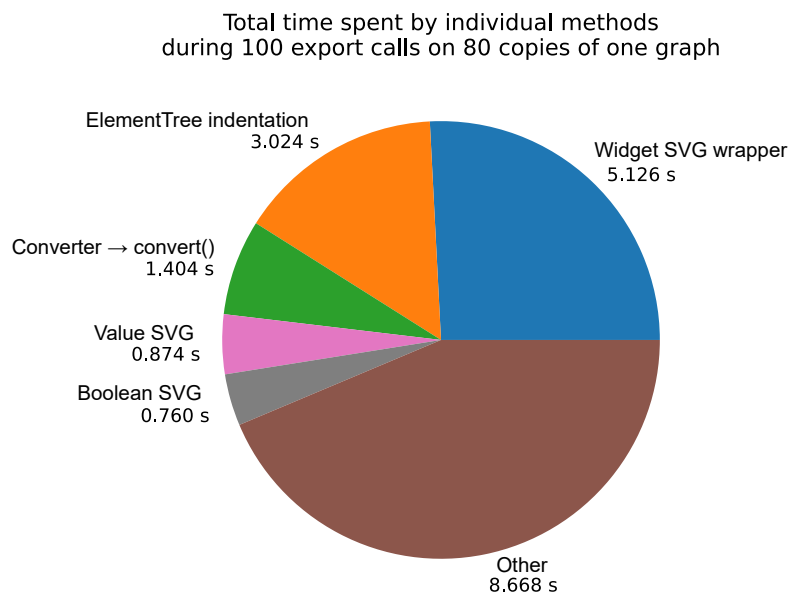
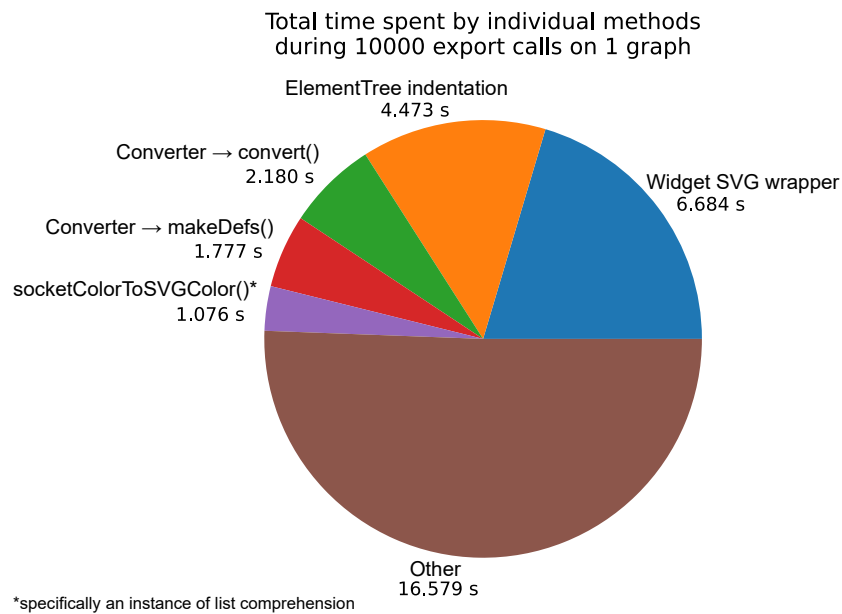


Figure 5.10: Charts of the proportional time spent by proper execution of methods during the export process. Five methods with the longest execution times are highlighted for each case. The times are measured over multiple iterations on a single Node tree (10000 iterations on a six-Node graph above, 100 iterations on 80 copies of the same graph below). The colors of the fourth and fifth most time-consuming method in each chart are different to highlight the different methods.

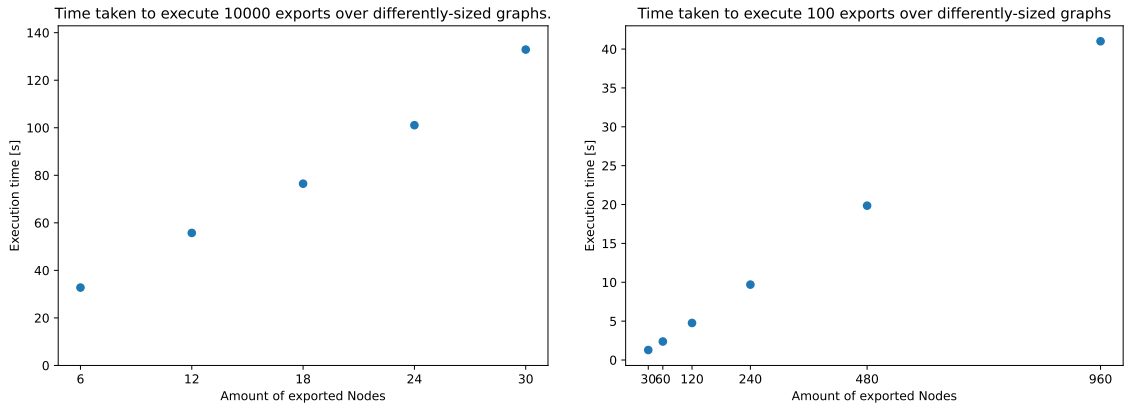


Figure 5.11: Execution times of Node graphs of varying sizes being exported, including average Node counts (left), and more extreme cases (right). While both charts use different iteration counts, the relationship between Node count and execution time remains the same.

Node Frames

While the regular export process does not entail functions which risk breaking the linear relationship between Node count and execution time, the presence of Node Frames (as described in Section 4.4.3) complicates the initialization part of the export process.

Theoretically, not even the Node Frame processing should introduce excess complexity to the process. All Nodes in the graph are iterated over twice: once to determine whether any of them have a parent, and the second time in either a loop over those that don't or over those that do. If a Frame is present, then its `updateOnTree` method is called, and it iterates over its children (which are, by then, known). While this technically could iterate over every other Node in the graph, the method only iterates over direct descendants, and each Node only has one parent, meaning that each Node will be iterated over within the `updateOnTree` method at most once, adding only at most one more loop over all Nodes to the execution.

To test how the presence of Node Frames affects computation time, a special Node graph was devised which features eight regular Geometry Nodes, and four types of nested Node Frames, depicted in Figure 5.12. Then, the execution time was measured while exporting this graph a thousand times with only certain Frames in place, to see if there is any relationship not just between the presence and amount of Node Frames, but the size of their content, too. Removing a Frame for this experiment leaves its content intact, though, meaning that each variant of the graph always contains the eight Geometry Nodes.

The results are presented in Figure 5.13. Even though the most complex graph that includes all frames A to D has a higher execution time than almost every other case, it is still trumped by the graph which only includes Frames A and D. This suggests that the presence of Frames does not affect the execution time as much as other random factors. In fact, if the linear relationship hinted at with the previous experiments is to be believed, the high value for the ABCD graph may just be the result of the fact that each Frame is a Node itself. Therefore, it contributes to the execution time just by adding to the volume of exported content alone.

Finally, to see if using Node Frames does not affect the performance of the add-on at higher numbers, one last test was performed. Exporting a large number (256) of Geometry

Graph content	Execution time [s]
256 Nodes	13.746
128 Nodes in 128 nested Frames	6.532
256 Nodes in 256 nested Frames	14.324

Table 5.3: Time taken to perform a thousand export operations over large graphs. The first

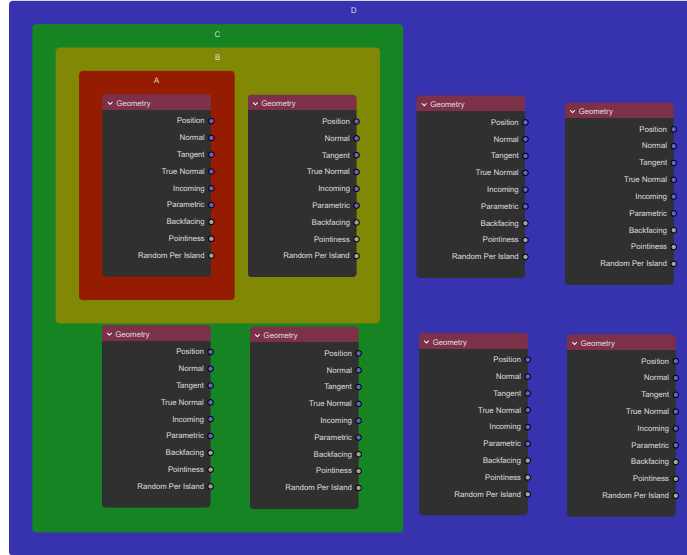


Figure 5.12: The Node graph used to measure the impact of Frames on the execution time. Four Node Frames are present, labeled A to D, with A being the smallest, containing only one Node, and D being the largest, containing all other Nodes and Frames.

Nodes was measured on its own. Then, half of these Nodes were put into Frames first, and these Frames were recursively nested so that each Frame (except for the last) contained one Geometry Node, and one other Frame. Finally, this amount was doubled for the last measurement. The results are provided in Table 5.3. Even though the total number of Nodes is the same for the cases with 256 Geometry Nodes, and 128 Geometry Nodes and 128 Frames, the latter is exported in roughly half the time. This indicates that the content of the Nodes plays a bigger role than the Node count alone, as each Geometry Node contains several UI elements, while Frames are, on their own, empty.

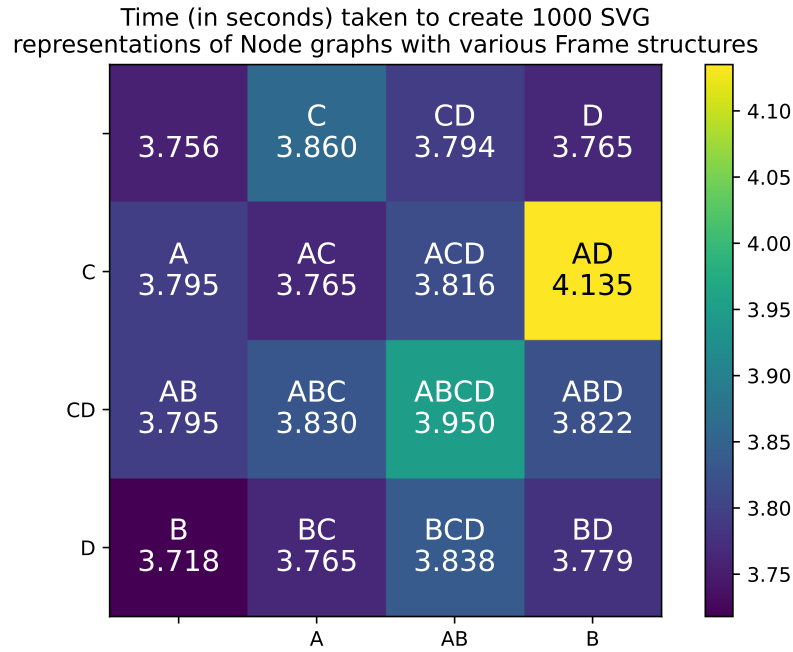


Figure 5.13: A chart of seconds taken to perform a thousand export operations over the graph in Figure 5.12. The chart shows times for each combination of frames A to D.

5.4 Future Work

While the add-on has been implemented to fulfill the assignment of this thesis, there is still room for improvement. The Blender editor is changing rapidly and dynamically. The most recent long-term support version, 4.1, has officially been released in March 2024, and has introduced changes which some users already requested to be addressed when providing feedback to the add-on. At the moment, the add-on lacks any form of 'future-proofing', or the logistics to rapidly update it alongside the editor itself. However, it appears that most changes related to Node editors could be addressed just by updating the `NODE_SPECIFICATIONS` dictionary. The `categories.py` script (where the dictionary is defined) could be extended by a function that updates the dictionary using Blender's built-in (aptly named) `update` method based on the editor's version.

Certain user requests have remained unaddressed at times of publishing. Namely, the proposed option to define the expected width of the viewport to which the SVG output will be rendered to is still not present in the add-on. Fortunately, this is not a game-breaker for the user as a vector image can still be easily scaled in the document in which it is included. Likewise, there has been a request to let the add-on read an exported SVG file and import its contents back into a Node editor. This has not been attempted yet as the add-on only reads from Blender's internal representation and does not actually alter it. However, it is considered, since it would greatly expand the applicability of the add-on. After all, Blender itself does not provide a direct way of copying and saving Node graphs on their own.

Finally, from a performance point of view, no function has been found during profiling that severely hamper the export process. That said, there were no optimizations done on the output itself, and its size could probably be cut down.

Chapter 6

Conclusion

The goal of this thesis was to implement an add-on that would give users an easy method to represent their Node editor creations in documents such as technical reports. Such add-on has been proposed, implemented, tested, and shared to public marketplaces. It has reportedly found use with some students that were working on their thesis this past year.

Arguably the most important parts of the project’s realizations were designing Widgets: a language of visual elements used to represent the content of Blender’s Nodes; and Node Specification: a simple method to streamline the inevitable details of appearance and behavior of individual Node types.

There is always potential for future development, and the thesis mentions ways in which the add-on can still be improved and maintained throughout Blender’s own release cycle. It is, after all, one of the conditions of pushing an add-on into the canonical application release itself.

The add-on was showcased at the Excel@FIT 2024 conference. The promotional poster is included in Appendix C. The add-on has also been published not only on GitHub¹, but also on Gumroad², and promoted at the Blender Artist forum³. With that, and a video that showcases the add-on’s functionality⁴, all of the steps of the assignments have been completing, save perhaps for the very first step about learning the basic Blender workflow: The author has, throughout communicating with other Blender users for testing purposes, found themselves to still be a rookie in terms of Blender’s capabilities.

¹<https://github.com/draberf/blender-node-export>

²<https://draberf.gumroad.com/l/blender-node-export>

³<https://blenderartists.org/t/export-nodes-to-svg/1530058>

⁴<https://www.youtube.com/watch?v=KYiybW2WdeE>

Bibliography

- [1] *Amendment 1 - Multimedia systems and equipment - Colour measurement and management - Part 2-1: Colour management - Default RGB colour space - sRGB*. 1st ed. TC 100/TA 2 - Colour measurement and management, january 2003.
- [2] *Blender 4.1 Python API Documentation*. Blender Foundation, 2024. Available at: <https://docs.blender.org/api/current/index.html>.
- [3] BEAZLEY, D. M. *Python cookbook* .: 3nd ed.th ed. Beijing: O'Reilly, 2013. ISBN 978-1-449-34037-7.
- [4] BIRTLES, B., SCHULZE, D. and JR., T. A. *CSS Masking Module Level 1*. Candidate Recommendation. W3C, august 2021. Available at: <https://www.w3.org/TR/2021/CRD-css-masking-1-20210805/>.
- [5] BRAY, T., SPERBERG MCQUEEN, M., MALER, E., YERGEAU, F. and PAOLI, J. *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [online]. W3C Recommendation. W3C, november 2008. Available at: <https://www.w3.org/TR/2008/REC-xml-20081126/>.
- [6] GUMSTER, J. van. *Blender for dummies*. 4th ed. Hoboken, 2020. ISBN 978-1-119-61696-2.
- [7] HAN, D. Four tips to create better figures for scientific papers. *MRS Bulletin*. 2020, vol. 45, p. 1061. DOI: 10.1557/mrs.2020.324.
- [8] HOLLISTER, B. E. *Core Blender Development: Understanding the Essential Source Code*. 1st ed. Apress, 2020. ISBN 978-1484264140.
- [9] KUHAİL, M. A., FAROOQ, S., HAMMAD, R. and BAHJA, M. Characterizing Visual Programming Approaches for End-User Developers: A Systematic Review. *IEEE Access*. 2021, vol. 9, p. 14181–14202. DOI: 10.1109/ACCESS.2021.3051043.
- [10] LEBRUN, J.-L. *Scientific Writing 2.0: a reader and writer's guide*. 1st ed. World Scientific Publishing, 2011. ISBN 9814350605.
- [11] LIE, H. W. and BOS, B. *Cascading Style Sheets: Designing for the Web*. 2nd ed. Addison Wesley, 1999. ISBN 0-201-59625-3.
- [12] LILLEY, C., BRINZA, B., WILLIGERS, E., STOREY, D., BELLAMY ROYDS, A. et al. *Scalable Vector Graphics (SVG) 2* [online]. Candidate Recommendation. W3C, october 2018. Available at: <https://www.w3.org/TR/2018/CR-SVG2-20181004/>.

- [13] MORTENSON, M. *Mathematics for Computer Graphics Applications*. 2nd ed. Industrial Press, 1999. ISBN 9780831131111.
- [14] SNÍŽEK, M. *CSS pro zelenáče*. Praha: Neocortex, 2003. ISBN 80-86330-14-1.
- [15] WANG, Z., BOVIK, A., SHEIKH, H. and SIMONCELLI, E. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*. 2004, vol. 13, no. 4, p. 600–612. DOI: 10.1109/TIP.2003.819861.
- [16] WATT, J., FUJISAWA, J., DAHLSTRÖM, E., LILLEY, C., SCHEPERS, D. et al. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C Recommendation. W3C, august 2011. Available at: <https://www.w3.org/TR/2011/REC-SVG11-20110816/>.

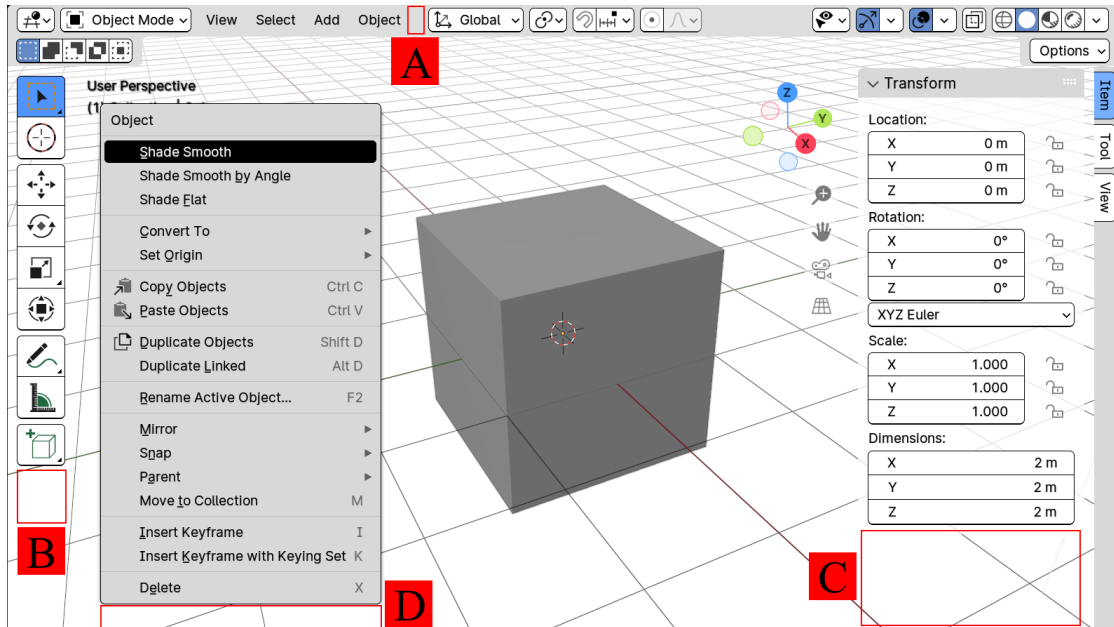
Appendix A

Survey

The initial research survey is available [here](#). At time of writing, 7 people have responded to the survey. Some of the interesting answers are provided here:

- *Please list any add-ons you knowingly use on a regular basis* – Respondents have cited the Node Wrangler add-on, as well as an add-on which lets the user export the current scene into SVG¹.
- *If you have an add-on that provides an operation (e. g. taking a screenshot of your current 3D viewport), would you expect to find it in the following areas?* – The results are shown in Figure [A.1](#). Respondents would most likely search for the add-on in the Toolbar.

¹<https://github.com/Craszh/BlenderModelToSVG>



If you have an add-on that provides an operation (e. g. taking a screenshot of your current 3D viewport), would you expect to find it in the following areas?

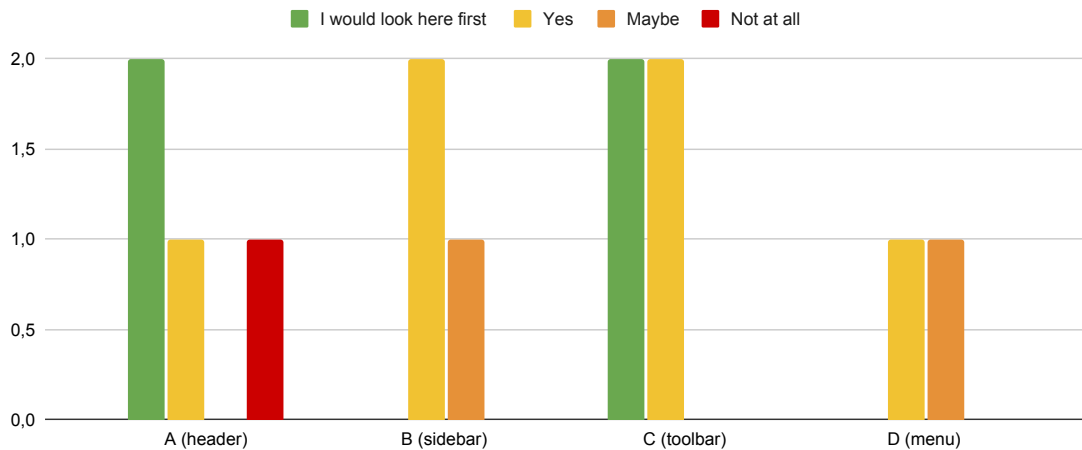


Figure A.1: User responses when asked what area of the editor (above) they would expect to find the given add-on in. The example add-on was selected as it is similar in application to what the project's add-on aims to do.

Appendix B

User Testing

This Appendix contains all three reference graphs, presented in Figures B.1, B.2, and B.3 that testers received for the testing process outlined in Section 5.2.2. The test protocol is available [here](#).

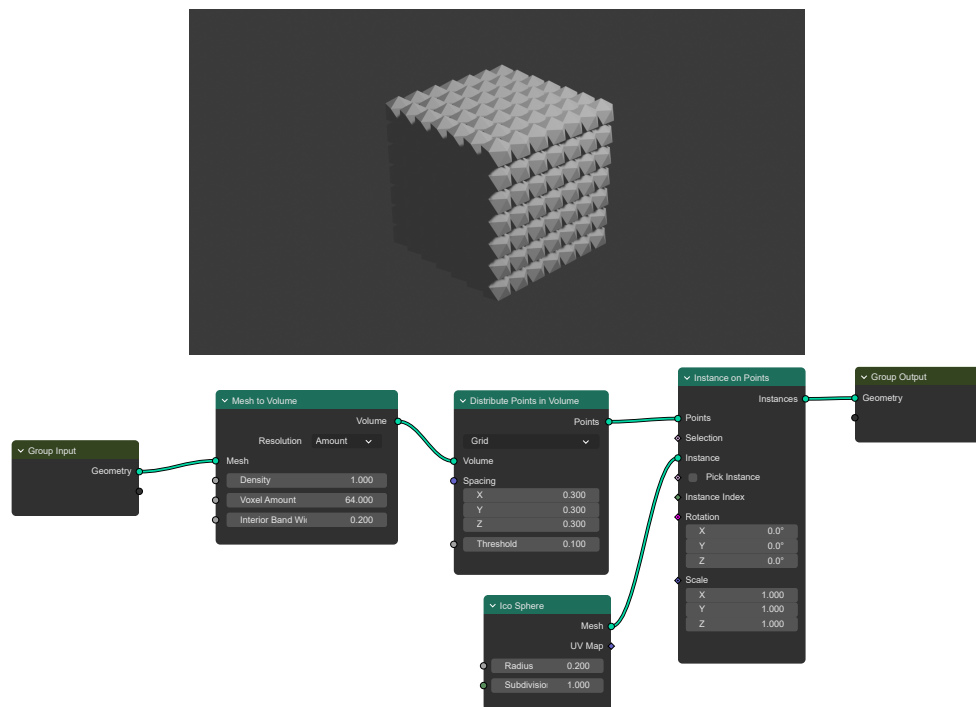


Figure B.1: Geometry Nodes example graph (below) and the resulting render after applying it to the default Scene.

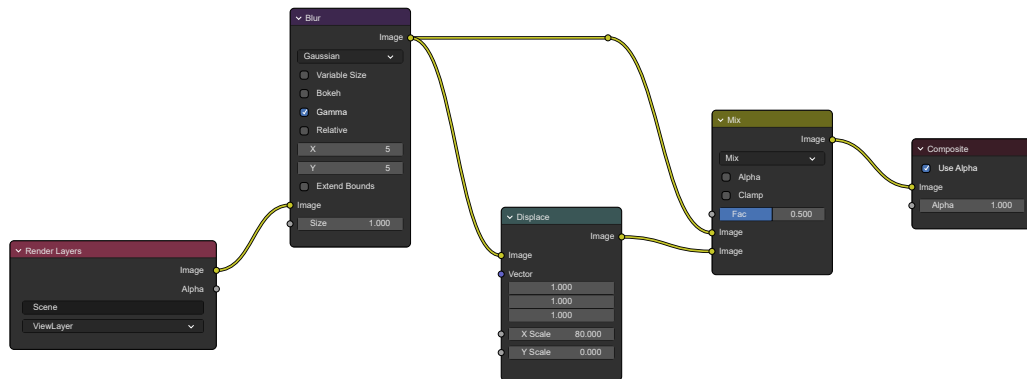
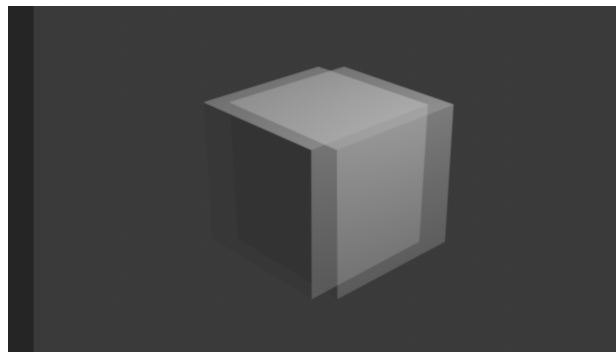


Figure B.2: Compositing Nodes example graph (below) and the resulting render after applying it to the default Scene.

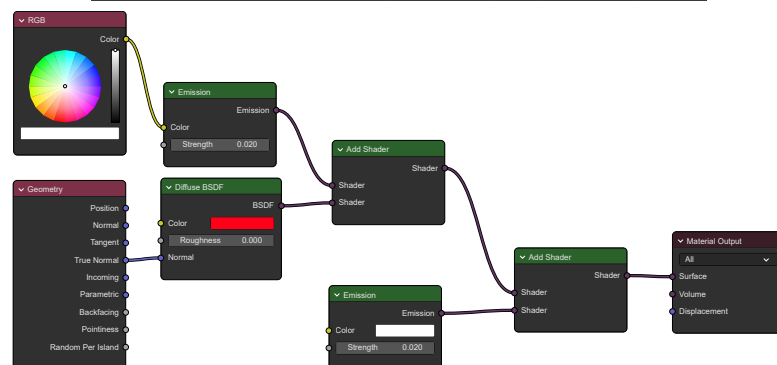
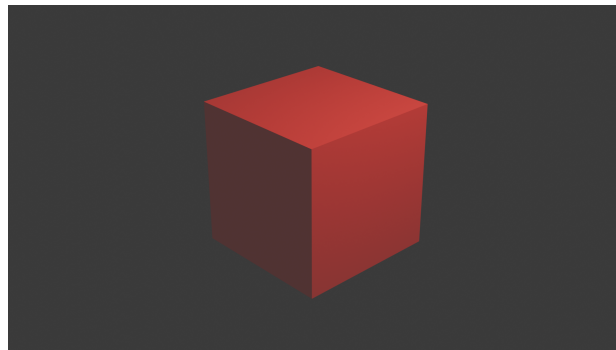
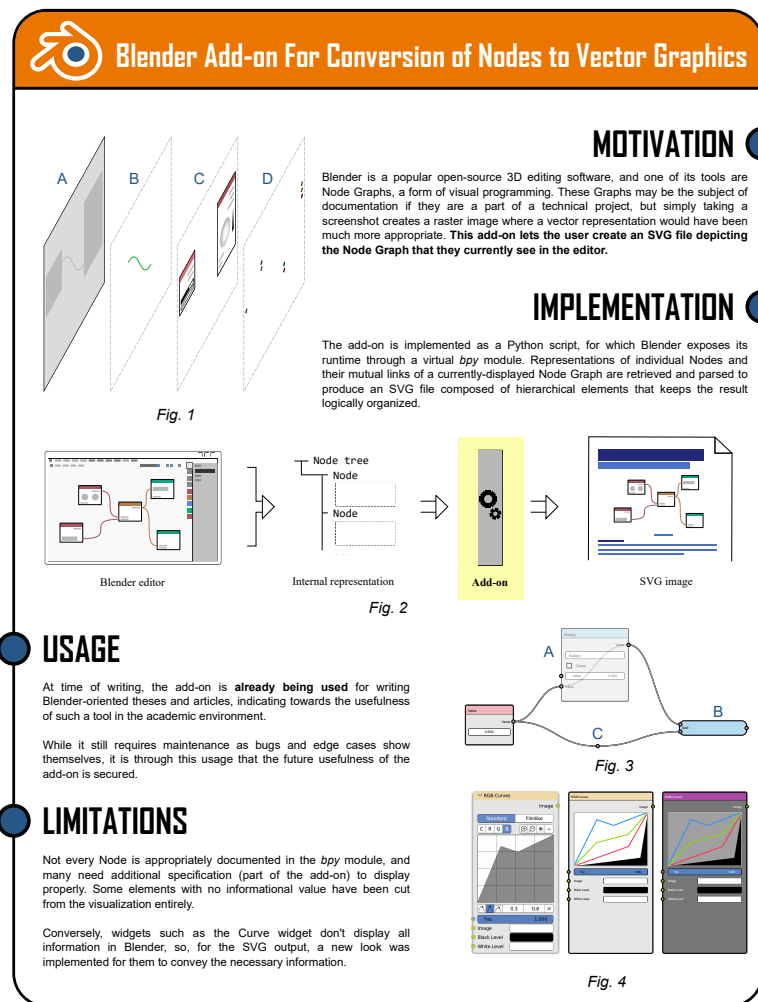


Figure B.3: Geometry Nodes example graph (below) and the resulting render after applying it to the default Scene.

Appendix C

Poster

This project was showcased at the Excel@FIT 2024 conference¹ with the following poster:



Author: Bc. Filip Dráber

2024

Supervisor: Ing. Tomáš Chlubna



¹<https://excel.fit.vutbr.cz>, possibly <https://excel.fit.vutbr.cz/2024> in the future.

Appendix D

SD Card Content

The following files and folders can be found on the medium included with this technical report:

- `MASTERS_THESIS_2024_draber.pdf` – this document
- `src/` – source code of the add-on
- `doc/` – source files for this document
- `README.md` – documentation of the add-on
- `LICENSE` – text of the GPL 3.0 license which the add-on is compliant with
- `Makefile` – script to generate an archive for installation with using ‘make’
- `blender-node-export.zip` – archive with the add-on (can be installed within Blender)