Technische-Hochschule Nürnberg

Fakultät Elektrotechnik Feinwerktechnik Informationstechnik

Elektro- und Informationstechnik

Abschlussarbeit (Bachelor) von

Adam R e i f

A renderer for plane-filling curves

WS 2017/2018

BACHELOR THESIS

Design and implementation of software for the visualization of plane-filling CURVES

cross-platform, extensible, open source

by

ADAM REIF

for a degree in electrical engineering and computer science at

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

focus topic:	information technology
student number:	2742638
supervisor:	Prof. Dr. Jörg Arndt
second reader:	Prof. Dr. Thomas Mahr

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Reif	Vorname: Adam	Matrikel-Nr.: 2742638
Fakultät: Elektro-, Feinwerk-, Informationstechnik	Studiengang: Elektrotechnik und Informationstechnik	
Semester: Wintersemester	2017/2018	

Titel der Abschlussarbeit:
Design and implementation of software for the visualization of plane-filling CURVES

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit [✓ genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,]

☐ genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von 0 Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format auf einem Datenträger beigefügt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Ort, Datum, Unterschrift Studierende/Studierender

Contents

Glossary

API Application Programming Interface. 5, 6, 21, 23, 25, 26, 36

Bounding Box The smallest possible rectangle (in 2D graphics) that fully contains an arbitrary geometric structure. ii, II, 1, 35–37

CI Continuous Integration. ii, 3, 11–13, 15, 28–30

CLI Commandline Interface. 18, 19, 33, 34, 42, 47

cloud Remote storage or computing on the Internet. 10–12

code coverage The percentage of lines of code executed during a run of a program in relation to its total lines of code. 15, 30

DevOps Development Operations, field of Software Engineering. 9

FLOSS Free/Libre and Open Source Software. III, 13, 16

Git A Version Control System. 11, 29, 36, 45

GitHub Cloud-based Git-frontend with additional project/issue tracking and extensive 3rd-party tool integration. 11–13, 29, 30, 33

GNU The GNU Project is a mass-collaboration project in the Free/Libre and Open Source Software (FLOSS) community. 8, 39

Graphics Processing Unit A processing unit specialized for massively parallel execution of commands common in computer graphics. 6

GUI Graphical User Interface. ii, 6, 7, 18, 21, 23, 24, 26–28, 31, 33–36, 38, 40, 42, 45–47

HAL Hardware Abstraction Layer. 7

IDE Integrated Development Environment. 21

L-System Lindenmayer System. II, 1–3, 6, 17, 31, 33, 36, 40–42, 44, 46

Meta-Object System An exetension to the C++ programming language used by the Qt Framework. 7, 23, 34, 40

MOC Meta-Object Compiler. 8

MSDN Microsoft Developer Network. 23

OMG Object Management Group. 15

OOP Object-Oriented Programming. II, 9, 38, 40

OS Operating System. 5

PFC Plane-Filling Curve. 3, 31, 42, 47

POSIX Portable Operating System Interface. 5, 8

QML Qt Modeling Language. 23, 24, 27, 34, 35

RTE Round-trip Engineering. 16, 33

SCM Software Configuration Management. 10

SDL Simple DirectMedia Layer. 7, 21

SFML Simple and Fast Multimedia Library. 6

SVG Scalable Vector Graphics. ii, II, 38, 40–42

UML Unified Markup Language. 3, 15, 16

Unit Testing Testing a functional unit (of software) irrespective of interactions with the rest of the code. 3

VCS Version Control System. 10, 11, 13, 29

Version Control System A tool for tracking, and keeping history of modifications (of code). 10

1 Introduction

1.1 Field of Research

Fractal geometry is an area of mathematical research that concerns itself with mathematically describing n-dimensional geometries - limited here to geometries on a plane - that display *interesting* characteristics.

Figure 1.1 shows a rendering of one such geometry called GOSPER'S FLOWSNAKE. It is a single, uninterrupted curve on a triangular grid, which is

self-similar The macroscopic behavior of the curve is the same as its microscopic behavior

self-avoiding The curve is made from one continuous line that never intersects with itself.

edge-covering The curve travels across every cell of its grid, i.e. it fills up a given, arbitrary area completely

plane-filling The curve grows outward on the 2D plane without bounds. Infinitely iterated it covers the full plane.

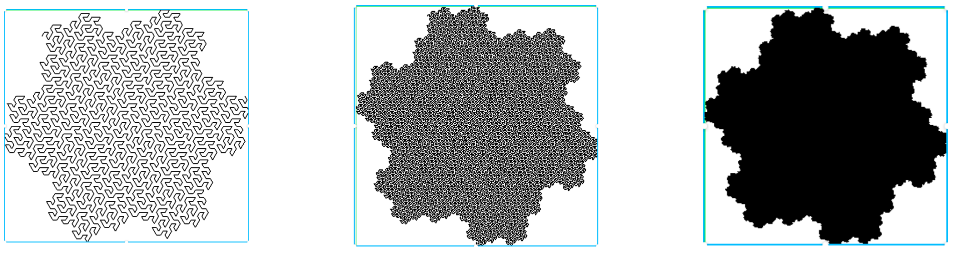

(a) 4 iterates (b) 5 iterates, rescaled (c) 8 iterates, rescaled

Figure 1.1: Rendering of GOSPER'S FLOWSNAKE, with Bounding Box

The curve is obtained by iterating a so-called L-System.

Originally introduced by biologist ARISTID LINDENMAYER in 1968 "as a foundation for an axiomatic theory of development"[Prusinkiewicz and Hanan 2013, Preface]. Originally intended to describe the growth of plants, it was found to be a useful language for a wide class of fractal geometries. An L-System consists of an initial string - the *axiom* - which is manipulated via a set of substitution *rules*, the result of which is a called the first *iterate* of the L-System. Subsequent *iterates* are obtained by applying the *rules* to the current *iterate*.

The L-System for figure 1.1 is given in Arndt [2016, p.7] to be

$$\text{L}(\textit{axiom}) \quad \text{L} \rightarrow \text{L+R++R-L- -LL-R+} \quad \text{and} \quad \text{R} \rightarrow \text{-L+RR++R+L- -L-R}$$

It yields the following *iterates*:

axiom	L
First iterate	L+R++R-L–LL-R+
Second iterate	L+R++R-L–LL-R++-L+RR++R+L–L-R++-L+ RR++R+L–L-R-L+R++R-L–LL-R+–L+R++R-L –LL-R+L+R++R-L–LL-R+–L+RR++R+L–L-R+

Table 1.1: GOSPER'S FLOWSNAKE iterates 0-2

The characters in this string are assigned special meaning with respect to the curve:

Alphabetic character Denotes drawing a line from the current position forward by a defined length. Forward being defined as the current direction

+ - Characters denoting changes in direction by a set amount, e.g. in figure 1.1 by $\pm 60°$, creating a triangular grid of movement

Those rules define a way to render a curve given by an *iterate* of a L-System, and can be extended by other, auxiliary characters, e.g. _ which changes color of subsequent segments.

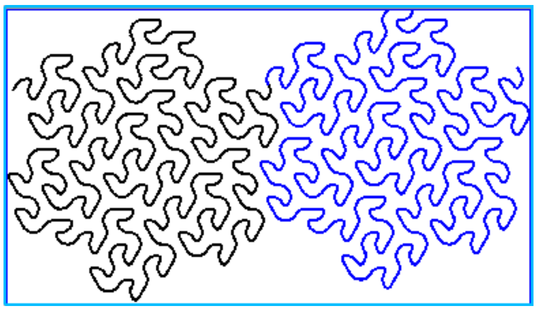


Figure 1.2: 3-iterate rendering of GOSPER'S FLOWSNAKE with axiom L_L and a rounding factor of 0.5

1.2 Problem Statement

In 2016, Prof. Jörg Arndt, the supervisor of this thesis, conducted research on finding plane-filling curves for L-System with one non-constant character [Arndt 2016] and presented 2D renderings of the curves found.

The tools used to get from a L-System description to a graphical, pdf-embeddable rendering of the iterated curve were assortments of chained commandline scripts, leaving much to be desired in flexibility and ease-of-use.

Thus, a request was issued for the creation of a *cross-platform, maintainable and extensible* software that is able to create, visualize and export a Plane-Filling Curve (PFC) from its L-System description.

1.3 Approach

Since the additional requirements introduce significant complexity on top of the "implement a renderer/exporter" core task, a structured approach to finding a solution is taken using decomposition methods from software engineering according to the following procedure.

1. Requirements to a solution are formulated and discussed.
2. Available technologies for satisfying given requirements are researched.
3. An architectural model is created using Unified Markup Language (UML) to define the system architecture.
4. The model is implemented and refined/amended to address issues arising during the implementation process.
5. The software is tested during development with Continuous Integration (CI) and Unit Testing techniques.
6. A short investigation of application performance is conducted.

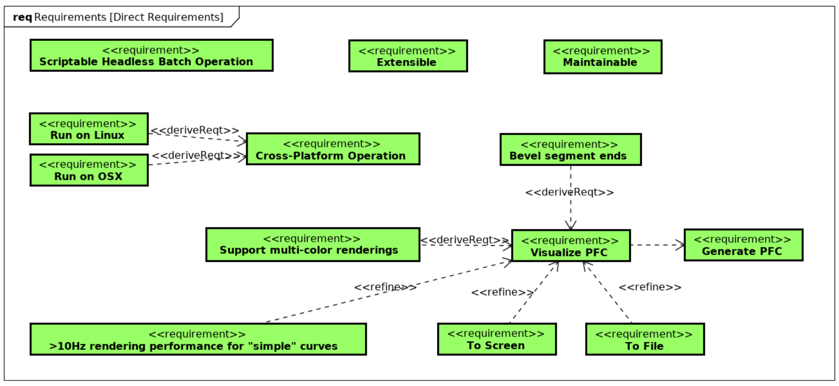

req Requirements [Direct Requirements]
<<requirement>> Scriptable Headless Batch Operation
<<requirement>> Extensible
<<requirement>> Maintainable
<<requirement>> Run on Linux
<<requirement>> Run on OSX
<<deriveReqt>>
<<deriveReqt>>
<<requirement>> Cross-Platform Operation
<<requirement>> Bevel segment ends
<<deriveReqt>>
<<requirement>> Support multi-color renderings
<<deriveReqt>>
<<requirement>> Visualize PFC
<<requirement>> Generate PFC
<<refine>>
<<refine>>
<<refine>>
<<requirement>> >10Hz rendering performance for "simple" curves
<<requirement>> To Screen
<<requirement>> To File

Figure 1.3: Direct user requirements to a solution

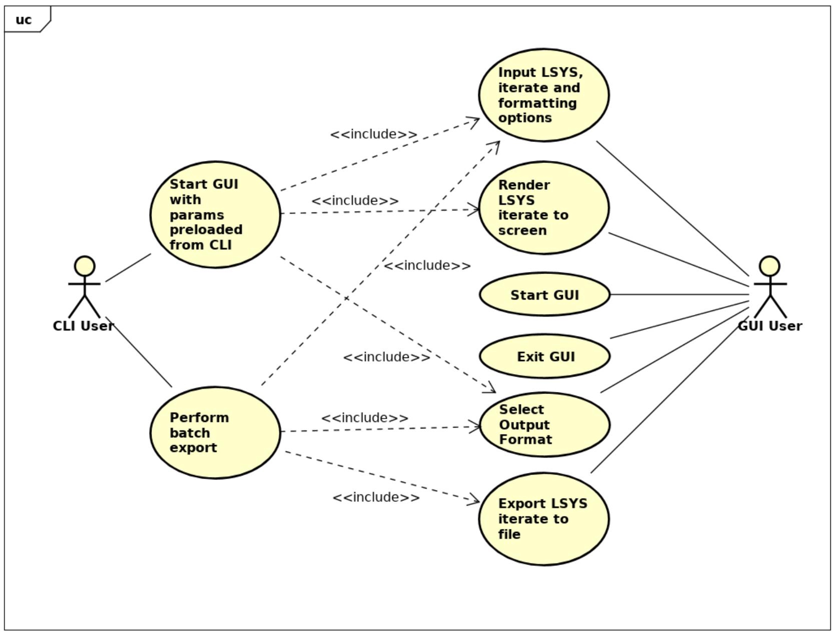


Figure 1.4: Workflow with the pfcrender tool

1.4 Requirements to a Solution

In order for the tool to be useful in research, requirements to a satisfactory solution were discussed with and agreed upon by Prof. Arndt, which are given in figure 1.3. Two possible workflows are shown in the use-case diagram 1.4.

These requirements are discussed in detail in the following sections and an architecture is formulated that satisfies them.

2 Research on Architecture Design Options

We discuss the given requirements, derive architecture design goals and research possibilities of implementing those goals.

2.1 Cross-OS Operation

Writing portable software has historically been difficult, as each Operating System (OS) has its own, not necessarily cross-compatible Application Programming Interface (API).

While Linux and MacOS comply with the Portable Operating System Interface (POSIX) standard - which defines an API for system calls and associated utility programs - Microsoft Windows has historically had a non POSIX-compliant proprietary system programming API.

Recently, Microsoft is making advances towards POSIX compatibility in Win10[1], and the Linux world offers a similar compatibility layer with WINE[2]. While the performance of the windows POSIX layer is promising[3], neither it, nor WINE[4] are on par with programs implementing each system's native interface directly at the time of writing.

As application performance is a requirement, writing the program in a cross-platform way becomes a necessity.

Notable API differences between POSIX systems and Windows are

Library Loading POSIX Systems use dlopen(), Windows uses LoadLibrary() and different runtime library layouts (.dll on Windows, .so on POSIX)

3D Graphics Backend Windows mainly uses Microsoft GDI or DirectX and - more recently - AMD Vulkan, while the POSIX world mostly uses OpenGL.

[1]`https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-overview/`
[2]`https://en.wikipedia.org/w/index.php?title=Wine_(software)&oldid=823689556`
[3]`https://www.phoronix.com/scan.php?page=article&item=windows-10-lxcore&num=2`
[4]`https://wiki.winehq.org/Performance`

Code depending on any of the above directly is therefore nonportable, which - as these are central concepts in any Graphical User Interface (GUI) app - makes implementing a cross-platform program from-scratch, i.e. without depending on OS-abstracting third-party libraries - an arduous task.

Thankfully, several of those libraries exist that provide a higher-level API, abstracting programmers from OS-specific code. They differ wildly in feature-scope, some being abstractions of the graphics layer, some being full-featured cross-platform GUI toolkits and will be investigated in the following section.

2.1.1 Rendering Frameworks Providing OS Abstraction

We define rendering as per Wikipedia[5] to mean

Definition 1 (Rendering)**.** The automatic process of generating a photorealistic or non-photorealistic image from a 2D or 3D model (or models in what collectively could be called a scene file) by means of computer programs.

It involves the decomposition of the 2D model (in our case - the L-System iterate) to graphical primitives (*polygons*) described by points in 2D space (*vertices*) and drawing information like color (*material*) for uploading to the Graphics Processing Unit which then rasterizes those primitives to colored pixels for display on screen.

An OS abstraction is defined to be an API that allows the same high-level source code to run on multiple low-level, platform-specific rendering backends, for example OpenGL and DirectX.

Cairo Cairo[6] is a C-based 2D vector graphics library which is widely used in open-source projects like GTK+, WebKit and the Poppler PDF rendering library.

SFML The C++ written Simple and Fast Multimedia Library (SFML)[7] is a relative newcomer to the multi-OS application landscape, with v1 released in 2007.

While its core is a 2D graphics engine, add-ons exist for providing GUI application functionality.

At the time of writing, it is mainly used in in free or indie game releases.

[5]`https://en.wikipedia.org/w/index.php?title=Rendering_(computer_graphics)&oldid=822283842`
[6]`https://www.cairographics.org/`
[7]`https://en.wikipedia.org/w/index.php?title=Simple_and_Fast_Multimedia_Library&oldid=820377932`

SDL The Simple DirectMedia Layer (SDL)[8] is a cross-platform Hardware Abstraction Layer (HAL) library for 3D graphics created in 1998, with extensive use in games industry[9] to provide 2D & 3D rendering.

GTK+ Unlike above solutions, GTK+[10], written in C - is a full GUI application toolkit, providing not only a HAL, but also graphical control elements, called widgets. Since 2005 it uses cairo (2.1.1) to provide its low-level rendering functionality.

Qt Framework The Qt Framework, like GTK+ is a full-featured GUI application development toolkit and offers extensive functionality.

Like GTK+ it has a widget-based API, but additionally brings a JavaFX-like declarative API called QtQuick, which provides scene based rendering.

In addition to providing graphics and GUI abstractions, it also provides a platform-independent library loading wrapper.

Asynchronous GUI programming is provided through an extension to the C++ language called the Meta-Object System and an event loop.

2.1.2 Build System Generators

Having cross-platform compilable sources is not the only requirement to getting a program developable and deployable cross-platform. In addition to sources and headers provided with the project, compiling needs information on

- which compiler/linker toolchain is available and where
- location of all dependencies/libraries the project links to
- if any platform specific preprocessor directives or linker flags have to be set
- Build type (Debug/Release) and architecture (arm, x86, amd64 etc.)

This information is typically stored in "project stores" like Visual Studio's solution, or makefiles. Since those are inherently platform-specific, and manually creating one for each project type and platform is cumbersome at best, so-called makefile generators were created, which parse build information from script files written by the programmer and then identify actual locations of dependencies and available

[8] https://www.libsdl.org/
[9] https://en.wikipedia.org/w/index.php?title=List_of_games_using_SDL&oldid=819037514
[10] https://www.gtk.org/

tools automatically. If the process is successful and all dependencies are found, they output a build definition (makefile, Visual Studio solution or similar) that runs the compiler with all necessary options. Often those tools can also create automated deployments for the software they are building, e.g. generate a make install target. The most widely used ones are discussed here

GNU Autotools A collection of the GNU utilities `autoconf`, `automake` and `libtool`[11]. It is the first widely used cross-platform build configurator, existing since the 1980s. It is used to generate a `./configure` script, that can be executed on any system that provides a bourne-shell (standard on POSIX systems) and generates a makefile.

Its reliance on the bourne-shell makes it unsuitable for use on Windows systems, which don't provide this POSIX utility.

QMake The native makefile generator of Qt[12]. Defines build definitions in .pro files and handles Qt specific tools like the Meta-Object Compiler (MOC) - necessary to translate the Qt-specific C++ language extensions into code parseable by a standard C++ compiler - automatically as needed, giving a straightforward experience when building Qt programs.

CMake A build generator with widespread usage in open source and commercial projects, CMake[13] gathers information on what/how to build from CMakeLists.txt files added to the project's sources. CMake can generate projects in an plethora of formats, including Visual Studio, GNU make, Apple Xcode and Eclipse. In addition to being a pure build system generator, CMake can be used to define commands that should be run pre-/post build and it comes with a unit testing framework called CTest.

2.2 Maintainability

Maintainability was declared a main goal in the requirements, and this is an issue with much broader reach than writing clean code.

The term describes the ease of performing maintenance tasks on software, the latter of which is defined in Bray et al. [1997, p.234] as

[11] `https://www.gnu.org/software/automake/manual/html_node/GNU-Build-System.html#GNU-Build-System`
[12] `http://doc.qt.io/qt-5/qmake-manual.html`
[13] `https://cmake.org/`

Definition 2 (maintenance)**.** The modification of a software product after delivery to correct faults, improve performance or other attributes, or to adapt the product to a changed environment

It is a highly subjective term, Bray et al. [1997, p.231] mention coefficients like average extended cyclomatic complexity per module, average number of lines of code per module and average lines of comments per module for a *maintainability index* - an attempt at quantizing maintainability.

Apart from the actual code, non-coding factors also impact the maintainability of a software project, e.g.

- Ease of collaboration with other developers
- Tracking of open issues and logging of modifications
- Early indication of breaking changes
- Incremental integration of changes with possibility to roll back to an earlier point in history

The importance of such "meta-programming" techniques is testified by the fact that a job field for software engineers called Development Operations, field of Software Engineering (DevOps) exists, which engages in finding ways to optimize coding workflow. A definition by Bass et al. [2015, p.23]:

Definition 3 (DevOps)**.** DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality.

While this field is too extensive to thoroughly investigate in the course of this thesis, some DevOps practices concerning automation of non-development work, while still guaranteeing that changes made don't degrade functionality of the software, will be implemented here.

Understandability, though distinct from maintainability in Bray et al. [1997], consists of factors like

- encapsulation of functionality in logical units - i.e. classes according to the implementation of OOP principles in C++
- common code formatting guidelines, i.e. making sure code from different contributors "looks the same"
- consistent code documentation

We will also consider matters of understandability in this work, as adhering to above rules increases productivity of developers and lowers the barrier of entry for new coders.

2.2.1 Version Control

Version Control Systems are used in Software Configuration Management (SCM) to keep track of changes (called *commits*) to a codebase. They provide features like

Commit history tracking Including author, timestamp and commit message

Reverting commits Returning to an earlier state of the codebase in case problems arise

Merging Automatic conflict resolutions in case two developers changed the same files

Branching Maintaining different states of the codebase in so-called branches

Using a Version Control System (VCS) is often mandatory in industry due to traceability requirements and but their use can be beneficial even for single-developer projects. A workflow that is enabled though VCSes is working on multiple separate features in parallel, only merging changes back into the main codebase after a feature has been completed. The main codebase can meanwhile keep evolving and the VCS will resolve the desynchronization automatically upon merge.

Most major VCSes have ecosystems that allow for hosting open-source repositories on the cloud[14] to facilitate contributions. Although a few conceptual differences exist, selection of a VCS is mostly a matter of personal preference. An overview over some popular VCSes available on both Windows and Linux follows.

Apache Subversion Subversion[15] is a VCS that follows a centralized approach. A SVN client can obtain a working copy by *checking out* a specific revision of the codebase from a central server that maintains the revisioning information. The client has no notion of versioning apart from its current working copy, all revisioning happens by interacting with the server, which thus must be available for anything but local changes.

It supports branching, tagging and commit messages.

[14]e.g. `https://sourceforge.net/create/`
[15]`https://subversion.apache.org/`

Mercurial Mercurial[16] is a decentralized VCS, getting a working copy does not fetch only the current status, but rather the complete repository including all versioning information, i.e. all interaction like committing changes, branching and checking out different code revisions can be done locally, while a central repository can be used to synchronize local changes with others.

Git Git[17], like Mercurial is a distributed VCS. It was conceived by Linus Torvalds, founder of the Linux kernel, to manage the large number of contributors on the project.

Like with Mercurial, key benefits are

Decentralization No reliance on availability of a central server for daily work

Smart merging Conflict resolution algorithms are fairly good at finding and automatically fixing trivial conflicts, keeping developer interaction on merges to a minimum

Central repository support Though Git operates locally, it supports pushing code to and pulling from remote repositories, enabling the use of a central "synchronization" repository, which forms the basis for the popular GitHub

Distinguishing factor of Git is its large existing userbase[18] that has resulted in a multitude of third-party tools integrating the VCS, mostly via its cloud-repository provider GitHub[19].

2.2.2 Continuous Integration

CI is a concept popularized in the rise of agile software development. In classical development structures, features are coded to perceived completion before they are introduced into the codebase of a product, their functionality maybe tested in unit-tests. Integration of completed features is performed in batches on a fixed schedule with integration tests following the procedure, leading to an eventual release.

A method of avoiding those time- and effort-consuming "big-bang" integrations is offered by CI. Instead of batching up features for integration, compilation and testing of code under development is done *continuously* e.g. every day. If changes to the new code break the build or existing functionality, the developer is notified

[16]`https://www.mercurial-scm.org/`
[17]`https://git-scm.com/`
[18]`https://www.openhub.net/repositories/compare`
[19]`http://github.com`

quickly. This limits the number of possible causes for the failure, reducing the time loss of bug hunting.

CI relies on having one or more systems in a consistent "ready to build/run" configuration for each type of target systems to be supported. Since running the application may change the target system's configuration, obtaining reproducible results without manual re-configuration of the build system is made possible through use of virtualization solutions like docker[20] and VMWare[21].

It is rarely feasible for small projects to maintain the necessary build systems themselves. Virtualization and CI are often the only possibility of testing the software on all to-be-supported platforms though, as the developer will not necessarily have access to machines in all configurations. In this particular case for example, development was done on Linux, with no physical access to MacOS devices to test the code on.

Thankfully, existing cloud providers offer virtualized build systems in various configurations as a service, often free for open-source projects.

Travis CI Travis CI[22] at the time of writing provides virtual machines running Linux (Ubuntu 12.04 and 14.04) and Mac OS X to provide builds on their cloud platform.

It integrates with GitHub for starting a build on each commit and reports completion status back to GitHub.

Builds are queued on commit, depending on the load of servers it can take some time for them to actually execute.

Configuration happens in a .travis.yml file that has to be provided with the sources and the service is free for open source projects.

Circle CI Circle CI[23] distinguishes itself from Travis through its larger selection of preconfigured docker containers, reducing configuration overhead of the build environment. Configuration is done through a .circleci file in the sources and one Linux-based docker container is free. MacOS X is paid only.

Like with Travis, they integrate with Github.

[20] `https://www.docker.com/`
[21] `https://www.vmware.com/`
[22] `https://travis-ci.org`
[23] `https://circleci.com`

Appveyor CI Appveyor[24] is a Windows-only CI provider, enabling builds on the various Microsoft Visual Studio compilers.

It provides GitHub integration, the build is configured through an appveyor.yml file in the sources, and is free for open source projects.

2.2.3 Code Formatting Guidelines

Creating a common code format can be as simple as writing down a set of rules in a code-of-conduct letter for coders to adhere to. What constitutes a "good" formatting guideline is a highly subjective matter however, which is the reason for "different looking code" in the first place, and differences in opinion are difficult to overcome.

It is unlikely that consensus will be reached on a coding style standard throughout the full C++ community, as the language is used in very different scenarios ranging from embedded computing to complex scientific computations, and "sensible" formatting differs by use-case. Several companies and larger FLOSS projects often define guidelines for their employees / contributors however, e.g. the WebKit project[25] and Google[26].

Though it is possible to manually review every contribution and deny it if it violates code syle guidelines, doing so occupies a developer's time unproductively. Since reformatting code (in the case of C++) simply consists of parsing and reorganizing characters in a text file, it can be automated.

Tools like clang-format[27] and astyle[28] automatically reformat source code when supplied with a formatting contract.

When used in conjunction with a VCS, formatting can be automatically applied before publishing changes (e.g. git pre-commit hook) to other parties, completely abstracting the developer from the process - and thus enabling him to freely disregard code style guidelines during development, without violating them on check-in.

[24] `https://ci.appveyor.com/login`
[25] `https://webkit.org/code-style-guidelines/`
[26] `https://github.com/google/styleguide`
[27] `https://clang.llvm.org/docs/ClangFormat.html`
[28] `http://astyle.sourceforge.net/`

2.2.4 Automated Unit Testing Frameworks

Research into this field was mostly motivated by a blog post[29], presenting compelling arguments for devoting time to creating an automated testing environment, which closes with the following words

> Nobody has the time to write tests, but true professionals make the time.

While manual testing of software is expensive for large corporations, it is strictly infeasible for small-scale software projects, which don't have the necessary manpower to still make progress while repeatedly testing changes.

Creating an automated testing suite amounts to a one-time investment of work (not counting the work needed to write tests for new code), at which point the developer can resume devoting his time to coding - until something breaks, at which point the developer benefits of reduced bug hunting time due to the localized nature of unit tests.

The main benefit of having an automated testing suite covering most of the software as given in above post is the fact that - while manual testing would only concern itself with parts of the software supposedly affected by changes - the full test suite can run on every check-in without causing significant cost to the project, increasing the chance to discover accidental cross-influences of changes to other parts of the software, which would normally remain undetected.

Having automated testing requires a software that can automatically run said tests. Many such frameworks exist, mostly differing in the way how tests are defined. Some of those available for the C++ language are:

QtTest [30] The native unit testing framework of Qt (described in 3.2.1), integrated into its QtCreator IDE.

CTest [31] Due to unit testing being closely related to building the actual software, CTest is distributed with the CMake build tool presented in 3.2.2. It shares syntax with CMake, i.e. test definitions can be integrated into the same files describing how to build the software.

Google Test [32] An open source standalone testing framework aiming to keep test definition overhead low while staying platform-neutral.

[29] `https://medium.com/@fatboyxpc/the-hard-truth-nobody-has-time-to-write-tests-68a122a1a0e3`
[30] `http://doc.qt.io/qt-5/qttest-index.html`
[31] `https://www.vtk.org/Wiki/index.php?title=CTest:FAQ&oldid=9429`
[32] `https://github.com/google/googletest/`

As the simple presence of tests does not confer any confidence in the functional integrity of a program by itself, it is possible to add instrumentation to a program to compile a report on which lines of code were actually executed during a run. These reports can be generated when running tests, and in sum constitute a code coverage report - i.e. give insight into which parts of the program were actually executed during testing and are thus proven to work as tested.

Generating coverage information during program execution is possible by compiling the program with gcc and including the following flags[33]

```
g++ -fprofile-arcs -ftest-coverage program.cpp
```

This adds instrumentation to the compiled program, which will dump `.gcno` and `.gcda` files to the filesystem, that can then be compiled to a report with other tools like gcov[34] and either viewed directly with lcov[35] or uploaded to a testing/coverage dashboard like cdash[36] or codecov.io[37] for visualization.

In combination with CI and a test suite close to 100% code coverage ran on every check-in, a solution like this is a great indicator that the program still works as intended after changes have been performed while costing close-to-0 ongoing time investment from developers.

2.2.5 Model-driven Design and Round-Trip Engineering

First step to writing a understandable program is understanding it yourself. Since most problems in programming today are fairly large in scope, it is often conducive to this process to decompose the main problem into several smaller subproblems. While this approach reduces the complexity of each individual problem and enables parallel work on each subproject, it introduces the overhead of keeping track of the relationships between all subproblems and guaranteeing interoperability between the individual solutions. A graphical method of mapping those relations is often conducive to understanding an architecture and maintaining synchronization between code and model. One such method is the UML standard by the Object Management Group (OMG), purpose of which - as taken from the specification[38] is

[33] https://cmake.org/Wiki/index.php?title=CTest/Coverage&oldid=57013
[34] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[35] http://ltp.sourceforge.net/coverage/lcov.php
[36] https://www.cdash.org/
[37] https://codecov.io/
[38] http://www.omg.org/spec/UML/2.5.1/PDF

> to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes

While certainly helpful in the initial phases of system architecture, models - like documentation - when manually created are cumbersome to maintain and keep synchronized with the running changes to the architecture.

In model-driven software design, the /glsuml model can be used to generate boilerplate parts of the code that map to elements of the model (e.g. class / namespace declarations), saving the implementer from writing boilerplate code. Architectural changes made in the code need to be manually introduced to the model though, which is at the discretion of the programmer and cannot be enforced, which often leads to the model desynchronizing from the codebase.

Round-trip Engineering (RTE) tooling helps alleviate this problem by automating the process of feeding changes from the code back into the UML model.

Depending on the target programming language, the task of reintroducing changes can be very difficult. C++, being a very syntactically complex language, is notoriously difficult to round-trip engineer, the effect of which is that the investment of time necessary to write an RTE tool is only feasible for companies, making FLOSS tools supporting RTE very rare.

During research, only a single open source / free-for-education UML modeling tool could be identified that offers RTE for C++: Astah[39]

Exporting a model to C++ code is a core functionality of Astah, import from C++ is provided through a plugin[40].

2.2.6 Automatically Generated Documentation

The necessity of documenting code is a topic of heated discussion among programmers. Some argue that today's high-level programming languages are mostly self-documenting, so why restate the obvious.

Others claim that stating the intent of a section of code clearly in plain language can make code much easier to understand, and is - for this very reason - a part of the *maintainability index* presented in section 2.2.

[39] `http://astah.net`
[40] `http://astah.net/features/cpp-reverse-plugin`

Documenting is most unintrusively done directly in the code and can - when using one of the available documentation generators like Doxygen[41] or Sphinx[42] - lead to great documentation for people not wanting to sift through source code, while not overly distracting a developer from his work.

2.3 Extensibility

A definition of extensibility in Johansson, p.3:

Definition 4 (Extensibility)**.** We define extensibility as the ability of a system to be extended with new functionality with minimal or no effects on its internal structure and data flow.

2.3.1 Plugin Architecture

It is apparent from the requirements in figure 1.3 that most components have strictly disjunct responsibilities. The main common element is the *iterate* of the L-System.

The L-System generator takes some input and outputs a *iterate* that can serve as a data model. GUI, SVG and PDF renderings all operate on this model, but don't influence each other's operation.

This characteristic is conducive to segregating those components of the app not just to different classes, but to completely separate compilation units, i.e. standalone libraries called plugins.

In combination with a dynamic loading system, this confers several benefits:

- Reduced compilation time of the main application
- "Pluggability" of new functionality without recompilation of the main app
- Lower startup times, as functionality can be loaded once it is needed, keeping the size of the core binary small

Drawbacks to this implementation are

- Updating the main app with changes breaking public APIs used by plugins will fail at runtime until all plugins have been updated/recompiled

[41] `https://github.com/doxygen/doxygen`
[42] `http://www.sphinx-doc.org/en/stable/`

- If plugins have dependencies on other plugins, maintenance complexity grows exponentially and loading sequence must be enforced

The additionally introduced complexity can - in sufficiently complex projects - significantly outweigh modularization benefits, resulting in what is called *plugin hell*[43]. The Eclipse IDE, in its core being a plugin loader and all functionality supplied by plugins is a prime example of the possible complexity.

2.3.2 Configuration

The solution is required to be scriptable, which means all configuration options necessary for automated operation must be selectable from the command line. This focus on a feature-rich Commandline Interface (CLI) leads to the necessity of good CLI documentation, i.e. a form of "help" message that documents the options available to the user. Providing and maintaining a help page manually is not only cumbersome, it runs the risk of going out-of-sync with the codebase.

This problem becomes more significant when using a plugin architecture, as available options depend on the presence or absence of plugins that use them. This in turn is runtime information that cannot be known at compile time, thus making any of the standard, static methods like "hardcoded" help message, manpage or README files impossible to use.

Though it was not a direct requirement, it is desirable to provide the GUI with a method of setting configuration options as well, to eventually enable users that don't need scriptability to operate the application fully from the GUI.

Thus, a method of configuration is needed that is easily extensible with new options, accessible from CLI and GUI and generates CLI documentation automatically at runtime.

2.3.2.1 Singleton Pattern

Whenever a piece of information needs to be known in global scope, i.e. across the whole application and exist only once - as is the case for a store of configuration information - the Singleton, described in Gamma et al. [1994, pp. 127ff] is a very useful design pattern.

[43]`https://queue.acm.org/detail.cfm?id=1053345`

Listing 2.1: Singleton Implementation in C++

```cpp
class Config_Registry {
private:
    static Config_Registry* instance;
    Config_Registry();

public:
    static Config_Registry* getInstance()
    {
      if (instance == nullptr)
          instance = new Config_Registry();
      return instance;
    }
    vector<config_opt_T> conf_opts;
    ~Config_Registry();
    Config_Registry(Config_Registry&) = delete;
    Config_Registry operator=() = delete;
}
```

Key element of this class is the private constructor, which can only be executed through the class-static `getInstance()` method, which returns an already existing instance, or creates a new one. Copying and moving are prevented by the deleted copy and assignment operators - presence of which prevents autogeneration of move constructors by the compiler. It follows that an object so-created can exist only once in the program and can be accessed from any file that can access the `getInstance()` method, i.e. has the header included. This class can then contain a list of config options and other elements of interest to multiple parts of the program.

Since the pointer variable storing the singleton has static duration, the lifetime of its object only ends when delete is called on the instance explicitly, meaning care has to be taken to ensure its proper destruction and prevent memory leaks.

2.3.3 Commandline Parsing

C++ does not provide a standard built-in facility to provide parsing capability for parsing parameters passed to the program as commandline options.

It is conducive to extensibility (and mandatory when implementing a plugin-based architecture as described in 2.3.1) if the program provides a CLI that is easily extended with additional commandline options.

The main commandline parsing solution available in any standard C++ environment is getopt(), which is inherited from C and has a procedural interface that is not easily extensible.

Some alternatives to getopt commandline parsing are:

Manually parse argv Build a parser for the system-provided commandline parameter array. While this is the most flexible solution w.r.t. customization, it is reinventing the wheel and also very inflexible regarding extensibility.

Boost.progam_options - A C++ native, highly configurable solution shipped with the open source Boost library[44]

QCommandLineParser [45] A Qt native parsing class. Batch-parses all parameters and returns data structures with positional arguments in correct order, and a (reordered) map of switches with their parameters

2.3.4 Option Persistency

Since it is often unnecessary and bothersome to set every config option for the program on the command line each time it is run, keeping config options persistently over program relaunches is useful. The typical implementation of this persistent store again varies by platform, ranging from putting keys into the Windows registry, over property list files on Mac to plain .ini or .cfg files on Linux.

The Qt Framework Qt comes with a wrapper around this platform disparity named `QSettings`, which allows for transparent and cross-platform settings and state storage to disk.

[44] `http://www.boost.org/doc/libs/1_58_0/doc/html/program_options.html`
[45] `http://doc.qt.io/qt-5/qcommandlineparser.html`

3 Chosen Application Architecture

As several alternatives were found in research for satisfying the requirements, we discuss the architecture chosen for implementation of the `pfcrender` tool. It is shown alongside the original requirements in the following graphs, split between functional- (figure 3.1) and non-functional (figure 3.2) requirements. As in figure 1.3, direct requirements are highlighted in green. Note that each direct requirement is satisfied or verified through an implementation requirement or test case.

3.1 Functional Requirements

The decision with the widest implications to the software architecture is selection of the appropriate graphics framework library from the ones discussed in section 2.1.1. It is driven by three main concerns, *extensibility, performance and ease-of-use*, leading to selection of the Qt Framework for the following reasons:

- Platform-independent dynamic library loading abstraction provided, which is useful for implementing a plugin architecture as described in section 2.3.1
- Classes provided for exporting e.g. to SVG and PDF
- The QtQuick API exposes direct OpenGL access, i.e. rendering performance is assumed to be comparable to the more light-weight frameworks like SDL
- GUI development support through the QtCreator Integrated Development Environment (IDE)
- Included QtTest Unit-Testing Framework

Qt provides most of the wanted features in one framework, reducing the amount of necessary dependencies to provide all required functionality. The only additional dependency to Qt introduced to the project is the drawing API QNanoPainter (described in section 3.1.2.2), which was introduced as manipulating OpenGL geometry directly proved unwieldy for rendering arcs (section 4.1.1.1)

A description of the Qt provided mechanisms used in this tool follows.

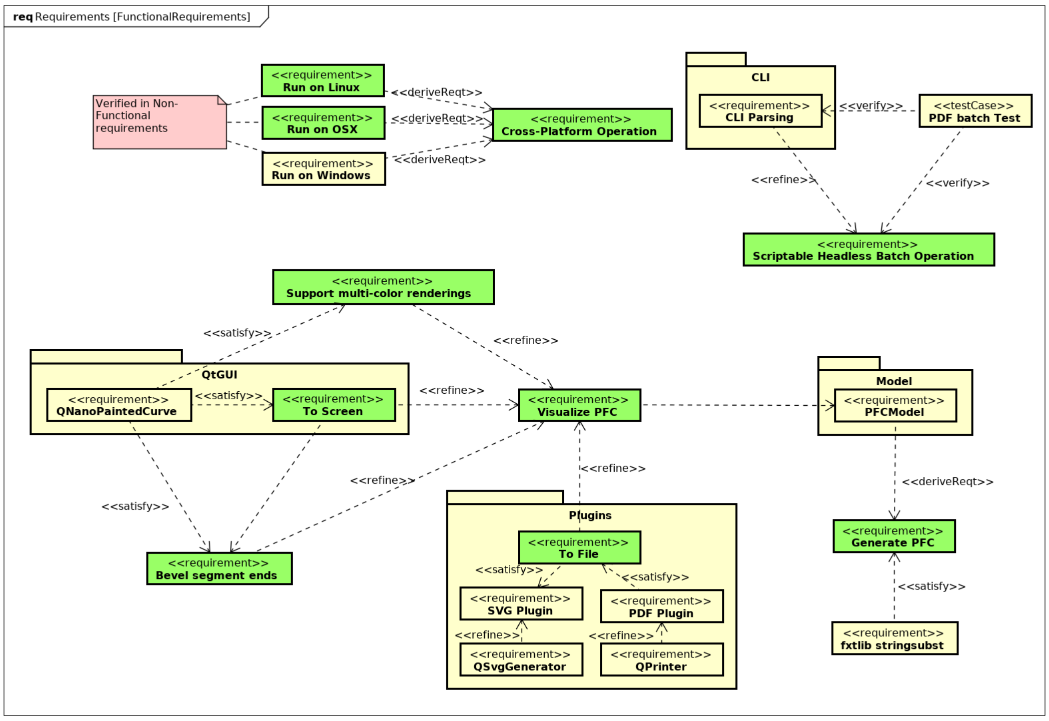

req Requirements [FunctionalRequirements]
Verified in Non-Functional requirements
<<requirement>> Run on Linux
<<requirement>> Run on OSX
<<requirement>> Run on Windows
<<deriveReqt>>
<<deriveReqt>>
<<deriveReqt>>
<<requirement>> Cross-Platform Operation
CLI
<<requirement>> CLI Parsing
<<verify>>
<<testCase>> PDF batch Test
<<refine>>
<<verify>>
<<requirement>> Scriptable Headless Batch Operation
<<requirement>> Support multi-color renderings
<<satisfy>>
<<refine>>
QtGUI
<<requirement>> QNanoPaintedCurve
<<satisfy>>
<<requirement>> To Screen
<<refine>>
<<requirement>> Visualize PFC
Model
<<requirement>> PFCModel
<<refine>>
<<deriveReqt>>
<<refine>>
<<satisfy>>
<<requirement>> Bevel segment ends
Plugins
<<requirement>> To File
<<satisfy>>
<<satisfy>>
<<requirement>> SVG Plugin
<<requirement>> PDF Plugin
<<refine>>
<<refine>>
<<requirement>> QSvgGenerator
<<requirement>> QPrinter
<<requirement>> Generate PFC
<<satisfy>>
<<requirement>> fxtlib stringsubst

Figure 3.1: Functional requirements

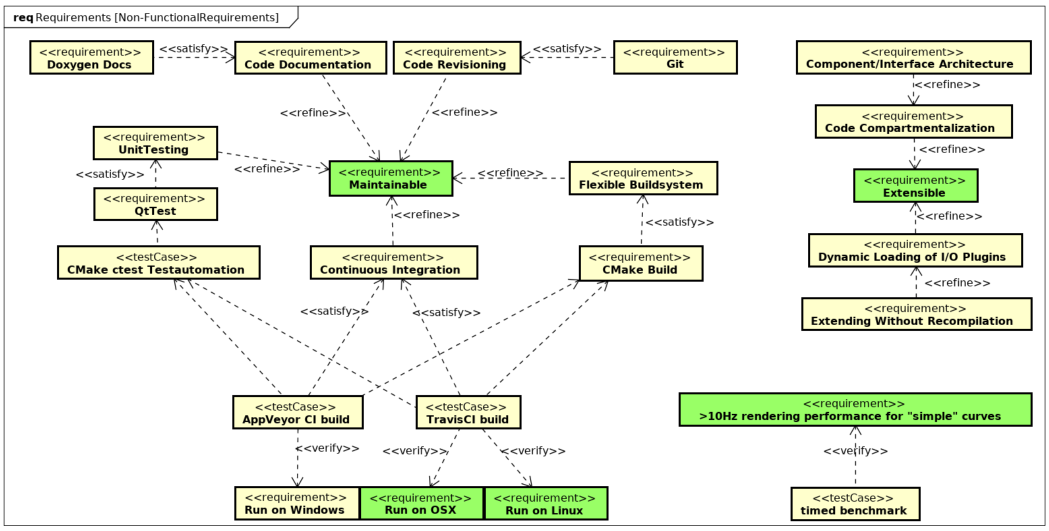

req Requirements [Non-FunctionalRequirements]
<<requirement>> Doxygen Docs
<<satisfy>>
<<requirement>> Code Documentation
<<requirement>> Code Revisioning
<<satisfy>>
<<requirement>> Git
<<requirement>> Component/Interface Architecture
<<refine>>
<<requirement>> Code Compartmentalization
<<refine>>
<<requirement>> Extensible
<<refine>>
<<requirement>> Dynamic Loading of I/O Plugins
<<refine>>
<<requirement>> Extending Without Recompilation
<<refine>>
<<refine>>
<<requirement>> UnitTesting
<<satisfy>>
<<requirement>> QtTest
<<refine>>
<<requirement>> Maintainable
<<refine>>
<<requirement>> Flexible Buildsystem
<<refine>>
<<satisfy>>
<<testCase>> CMake ctest Testautomation
<<requirement>> Continuous Integration
<<requirement>> CMake Build
<<satisfy>>
<<satisfy>>
<<testCase>> AppVeyor CI build
<<testCase>> TravisCI build
<<requirement>> >10Hz rendering performance for "simple" curves
<<verify>>
<<verify>>
<<verify>>
<<verify>>
<<requirement>> Run on Windows
<<requirement>> Run on OSX
<<requirement>> Run on Linux
<<testCase>> timed benchmark

Figure 3.2: Non-functional requirements

3.1.1 Dynamic Library Loading

Windows DLL API mandates symbols callable from outside a library to be prefixed with a macro on export and import. The Microsoft Developer Network (MSDN)[46] defines it the following way

```
__declspec( dllimport ) declarator
__declspec( dllexport ) declarator
```

Linux compilers like gcc will not be able to parse this keyword and fail compilation. While it is possible to work around this issue by wrapping the keyword in a preprocessor macro that expands to nothing on Linux, this mechanism is cumbersome to implement.

Qt provides a wrapper around this functionality with a class called `QPluginLoader`[47].

Making a library compatible to use with this mechanism involves some extra steps compared to loading a shared library directly, which pertain to making the Qt Meta-Object System aware of the plugin[48], but spares the effort of including platform specific adapter code/preprocessor instructions to each plugin.

3.1.2 Rendering in Qt

Qt offers two APIs, which differ in their rendering methods:

QWidgets The main API of Qt. It is similar to Java SWING and handles GUI design and functionality in C++ using Qt-provided or custom `QWidget`-derived classes. Each widget handles rendering itself through `QPainter` or direct OpenGL calls.

QtQuick A relatively new (since Qt 4.7) javascript-based declarative language called Qt Modeling Language (QML), similar to the XML-based JavaFX, is used to define the GUI frontend (*QML Types*), while C++ classes in the backend (`QQuickItem`) are used for performance intensive calculation and business logic. Rendering information from `QQuickItems` is composed to a *scene graph*, which is tasked with composing and executing relevant drawing calls on the backend. Though no formal study of performance differences between both APIs was found, documentation by the QtCompany[49] suggests performance between both models is comparable if some guidelines are followed.

[46]https://msdn.microsoft.com/en-us/library/3y1sfaz2.aspx
[47]http://doc.qt.io/qt-5/qpluginloader.html
[48]http://doc.qt.io/qt-5/plugins-howto.html
[49]http://doc.qt.io/qt-5/qtquick-performance.html

Since the additional model-/view abstraction is conducive to extensibility of the tool, as the declarative QML language makes GUI extensions simple, *QtQuick* is selected for the implementation.

3.1.2.1 QtQuick Scene Graph Rendering

Scene-based rendering as defined in section 2.1.1 describes the process of assembling the sum of drawing calls of all structures to be drawn and rendering them all in batch. The Qt Framework creates a rendering scene of its GUI elements in a structure called the Qt Scene Graph[50]. It represents a hierarchical parent-child relationship between objects in the GUI.

Every graphical item in QtQuick, represented by an object of the `QQuickItem` class running in the GUI thread has a corresponding `QSGNode` object in the scene graph, which contains the rendering instructions for this visual element.

The scene graph is separated from the application logic by running in a *rendering thread*, periodically redrawing the screen or when triggered through events. When rendering, the scene graph blocks the GUI thread to allow QSG* rendering classes to synchronize data with their `QQuickItems`, unblocks the GUI, establishes which elements are actually visible based on criteria like opacity and overlapping elements and then issues the necessary rendering calls on the backend.

The following methods of drawing to *QWidgets* or the scene graph API are offered:

QPainter Oldest drawing API of Qt used in *QWidgets*. Offers a high-level API (e.g. drawLine(), drawArc()), is not directly usable in *QtQuick* though. Offers highest amount of integration to other Qt classes (notably QPrinter and QSVGGenerator)

QtQuick/QML shape Placing line segments directly in a QML file and loading it to the SceneGraph (creating quadratic bezier curves added in Qt 5.10 (late 2017), too late to consider for this work[51])

QtQuick HTML5 canvas Draws on top of a HTML5 canvas embedded into the qt app

QtQuickItem with custom scene graph node Instantiates a QQuickItem with custom appearance by directly setting vertices and material for the underlying OpenGL to render

[50] `http://doc.qt.io/qt-5/qtquick-visualcanvas-scenegraph.html`
[51] `https://blog.qt.io/blog/2017/08/10/let-there-be-more-shapes/`

QQuickPaintedItem An adaptation of the QPainter API for the SceneGraph based *QtQuick*

QNanoPainter [52] A third party plugin, offering a mixture of QPainter and HTML5 canvas API to draw on an OpenGL framebuffer object or `QSGRenderNode` to be placed in the *QtQuick* scene graph. Offers Painter-like productivity with less performance overhead.

The QML and HTML5 APIs were disregarded due to their interpreted and thus slow nature.

This leaves direct QuickItem implementation, QQuickPaintedItem and QNanoPainter-based QuickItem as viable options.

Due to performance measurements[53], a direct implementation was chosen initially (see 4.1.1.1). The operations performed using this API were too inflexible to allow for generation of quadratic bézier curves necessary to implement rounded edges, so a higher level API was selected.

Further performance benchmarks[54] led to preferring QNanoPainter over the built-in QQuickPaintedItem, even though it meant introducing an additional dependency into the project.

It is worth noting, that QPainter API was still used for file outputs (see sections 4.2.2 and 4.2.3), where performance is not as critical, due to its integration with other Qt exporter classes.

3.1.2.2 QNanoPainter

Using QNanoPainter involves subclassing `QNanoQuickItem` and a corresponding `QNanoQuickItemPainter`, which are implementations of `QQuickItem` and `QSGNode` of the scene graph API.

It can draw on two different backends: `QSGRenderNode` and a `QOpenGLFramebufferObject`. The first one is an instance of `QSGNode` and gets rendered according to the scene graph logic, while the latter is not placed in the scene graph, but instead as an over- or underlay and rendered before or after scene graph rendering respectively.

[52] `https://github.com/QUItCoding/qnanopainter`
[53] `https://www.vikingsoftware.com/qtquick-custom-item-performance/`
[54] `http://kgronholm.blogspot.com/2017/12/qt-510-rendering-benchmarks.html`

3.1.3 Configuration

It was decided to introduce a central store of configuration that will be accessible to the entire application using the Singleton pattern discussed in section 2.3.2.1.

Each plugin is required to publish and read its configuration options from this central store.

We define the following possibilities of setting options to the central store, in order of execution and inverse order of precedence:

1. Options and state information are read from disk
2. Options can be given on the command line
3. Options can be set from within the GUI

3.1.3.1 Persistent Filesystem Storage - QSettings

The Qt framework comes with a provider for storing settings called QSettings. It abstracts the programmer from platform specific storage solutions of a configuration store and provides a simple `save()` and `restore()` API for the backing store on the filesystem it manages on its own. Configuration data is made available as a map of configuration identifier and configuration value.

3.1.3.2 Commandline Parsing - QCommandlineParser

Parsing of the commandline is one of the first steps in program execution. Since successfully parsing given arguments (and generating documentation on available options accessible by running `pfcrender --help`) requires all available options to be known, it follows that all plugins must be loaded on startup to query them for their config options.

While this is not computationally efficient, as not all plugins will necessarily be executed during program runtime, maintainability outweighs this performance issue. Loading all plugins is a once-per-execution process delaying application startup, not normal operation. Also, the number of plugins is fairly small. A benefit of loading all plugins at startup is that it allows for automatically gathering available functionality, omitting the need for a separate configuration file of plugins and their locations.

It is thus defined that plugin libraries will be located in a Plugins subfolder below the binary executable that contains no files other than plugins, so they can be crawled and loaded each time the program executes.

This design allows for a *self-documenting, truly runtime extensible* plugin system, where plugins can be added/removed from the program without additional configuration steps.

Due to it supporting our use-case, being easily extensible and not introducing any more dependencies into the project above the already used Qt Framework libraries, `QCommandLineParser` is chosen to provide the main CLI-based interface to the program.

A drawback of QCommandlineParser is batch parsing of the commandline, making it impossible to group configuration switches (denoted by - -switch syntax) by their position of occurrence in the argument string.

We thus define the following:

1. Actions taken by the program shall be specified on the commandline as positional arguments (with no preceding -), to guarantee their execution in order
2. configuration options are specified with a - -prefix and shall be unique throughout the application (including all plugins)

3.1.3.3 GUI configuration - QQuickItem

In order to make configuration possible from the GUI, a plugin shall expose a configuration screen described in QML as a `QQuickItem` through its plugin interface that contains all logic for setting configuration options on the plugin for the GUI to display.

3.2 Non-functional Requirements

3.2.1 Qt Unit Testing Framework

Qt comes with its own unit-testing framework integrated into QtCreator. The following code presents an example test for a fictional Counter class:

Listing 3.1: Unit Test for a Counter class

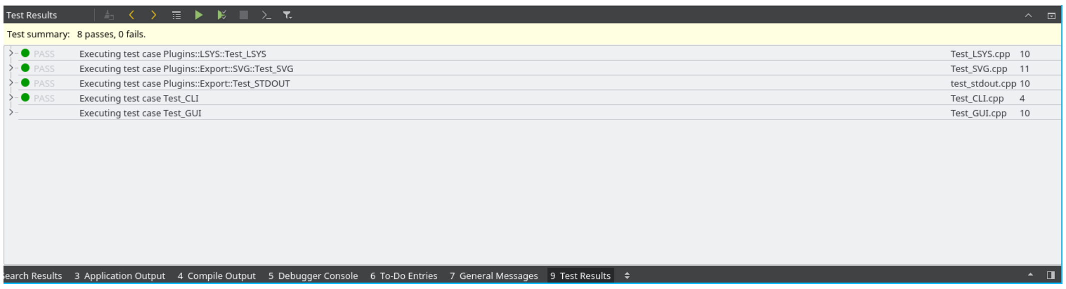
```cpp
#include <QtTest>
#include "Counter.h"

class MyTest : public QObject {
    Q_OBJECT

    Counter* ctr;

private slots:
    void initTestCase();
    void cleanupTestCase();

    void do_count();
  //void do_downcount(); // additional test routines
};

void MyTest::initTestCase()
{
  ctr = new Counter(0);
}

void MyTest::cleanupTestCase()
{
    delete ctr;
}

void MyTest::do_count()
{
  ctr->count();
  QVERIFY(ctr->getValue() == 1);
}
QTEST_MAIN(MyTest)
#include "MyTest.moc"
```

This code builds an executable (main routine provided by the `QTEST_MAIN` macro) that runs the given test routines. It is integrated with QtCreator, which gathers test executables created in this way automatically, can run them from the GUI and display PASS/FAIL status depending on `QVERIFY()` and return value as shown in figure 3.3.

Since it is a standalone executable, it can also be run by other testing environments like CMake's CTest, and thus be automated and integrated into CI builds as shown in figure 3.4

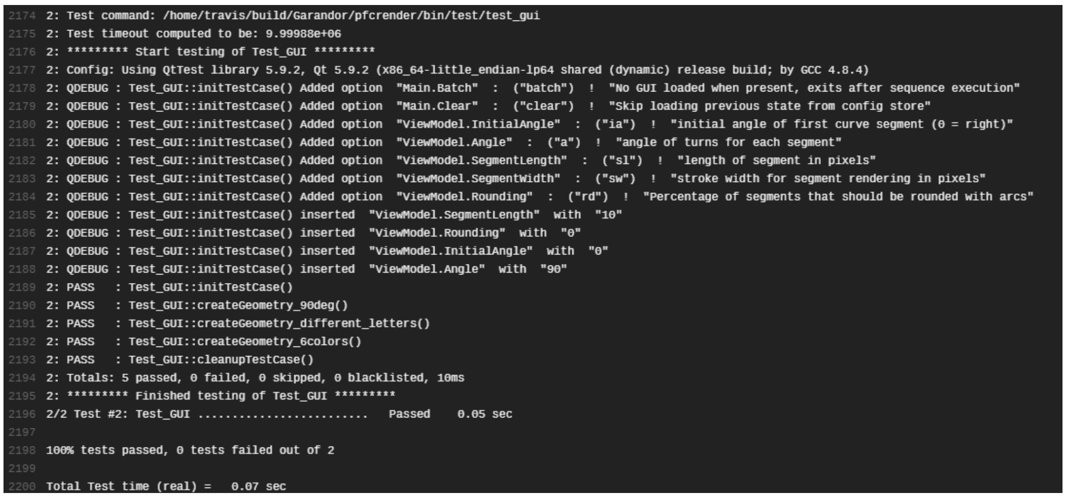
3.2.2 Makefile Generator - CMake

Though as described in section 2.1.2 Qt has its own makefile generator with qmake, CMake was selected for the following reasons



```
Test Results
Test summary:  8 passes, 0 fails.
● PASS   Executing test case Plugins::LSYS::Test_LSYS            Test_LSYS.cpp   10
● PASS   Executing test case Plugins::Export::SVG::Test_SVG      Test_SVG.cpp    11
● PASS   Executing test case Plugins::Export::Test_STDOUT        test_stdout.cpp 10
● PASS   Executing test case Test_CLI                            Test_CLI.cpp    4
         Executing test case Test_GUI                            Test_GUI.cpp    10
```

Figure 3.3: Unit Testing Output of QtCreator



```
2: Test command: /home/travis/build/Garandor/pfcrender/bin/test/test_gui
2: Test timeout computed to be: 9.99988e+06
2: ********* Start testing of Test_GUI *********
2: Config: Using QtTest library 5.9.2, Qt 5.9.2 (x86_64-little_endian-lp64 shared (dynamic) release build; by GCC 4.8.4)
2: QDEBUG : Test_GUI::initTestCase() Added option  "Main.Batch"  :  ("batch")  !  "No GUI loaded when present, exits after sequence execution"
2: QDEBUG : Test_GUI::initTestCase() Added option  "Main.Clear"  :  ("clear")  !  "Skip loading previous state from config store"
2: QDEBUG : Test_GUI::initTestCase() Added option  "ViewModel.InitialAngle"  :  ("ia")  !  "initial angle of first curve segment (0 = right)"
2: QDEBUG : Test_GUI::initTestCase() Added option  "ViewModel.Angle"  :  ("a")  !  "angle of turns for each segment"
2: QDEBUG : Test_GUI::initTestCase() Added option  "ViewModel.SegmentLength"  :  ("sl")  !  "length of segment in pixels"
2: QDEBUG : Test_GUI::initTestCase() Added option  "ViewModel.SegmentWidth"  :  ("sw")  !  "stroke width for segment rendering in pixels"
2: QDEBUG : Test_GUI::initTestCase() Added option  "ViewModel.Rounding"  :  ("rd")  !  "Percentage of segments that should be rounded with arcs"
2: QDEBUG : Test_GUI::initTestCase() inserted  "ViewModel.SegmentLength"  with  "10"
2: QDEBUG : Test_GUI::initTestCase() inserted  "ViewModel.Rounding"  with  "0"
2: QDEBUG : Test_GUI::initTestCase() inserted  "ViewModel.InitialAngle"  with  "0"
2: QDEBUG : Test_GUI::initTestCase() inserted  "ViewModel.Angle"  with  "90"
2: PASS   : Test_GUI::initTestCase()
2: PASS   : Test_GUI::createGeometry_90deg()
2: PASS   : Test_GUI::createGeometry_different_letters()
2: PASS   : Test_GUI::createGeometry_6colors()
2: PASS   : Test_GUI::cleanupTestCase()
2: Totals: 5 passed, 0 failed, 0 skipped, 0 blacklisted, 10ms
2: ********* Finished testing of Test_GUI *********
2/2 Test #2: Test_GUI .........................   Passed    0.05 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   0.07 sec
```

Figure 3.4: Unit Testing Output of a CI build using Travis CI

Widespread Use Though more complex than QMake, CMake is extensively documented

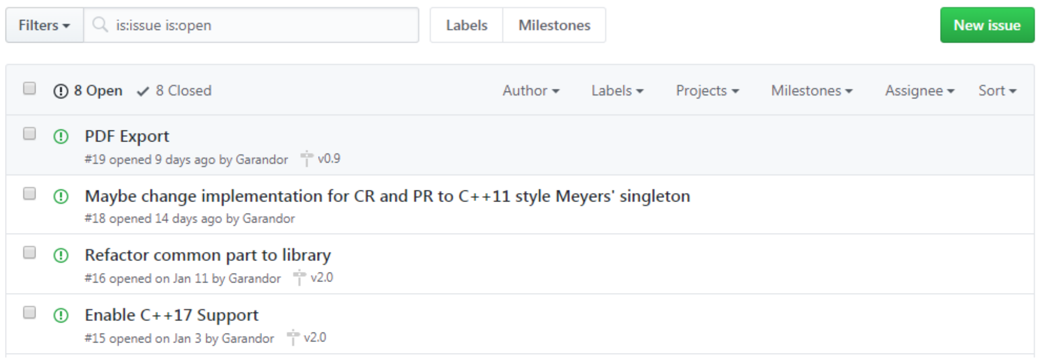
CTest Test-Runner Allows to define available unit tests in CMake configuration files and automatically run them after build completion

Especially the test runner is a large gain for maintainability when integrated with CI as shown in figure 3.4, as it makes automated builds more informative. A build will not only report failure on changes that break compilation, but also on changes that break runtime functionality for which tests exist.

3.2.3 Version Control System

Due to the useful issue tracking feature of GitHub and its extensive third party infrastructure integration with services like CI, Git is selected as VCS and GitHub as the central repository provider.

The project can be found at `https://github.com/Garandor/pfcrender`



PDF Export
#19 opened 9 days ago by Garandor v0.9

Maybe change implementation for CR and PR to C++11 style Meyers' singleton
#18 opened 14 days ago by Garandor

Refactor common part to library
#16 opened on Jan 11 by Garandor v2.0

Enable C++17 Support
#15 opened on Jan 3 by Garandor v2.0

Figure 3.5: GitHub issue tracking board

We leverage GitHub's integration by connecting it to the CI providers Appveyor for windows builds, and Travis CI for Linux and MacOS X, due to their free-for-open source service.

CI services for our repository are located at

- `https://travis-ci.org/Garandor/pfcrender`
- `https://ci.appveyor.com/project/Garandor/pfcrender`

Furthermore, CI runs are used to run a CTest suite of unit tests generating code coverage reports as described in section 2.2.4, which are then uploaded to cloud coverage dashboard codecov.io at `https://codecov.io/gh/Garandor/pfcrender`.

Documentation for this project is automatically generated from Doxygen structured comments in the code (section 2.2.6) on every CI build of the master branch on GitHub, which is then uploaded to github-pages, a website/documentation hosting service of GitHub[55].

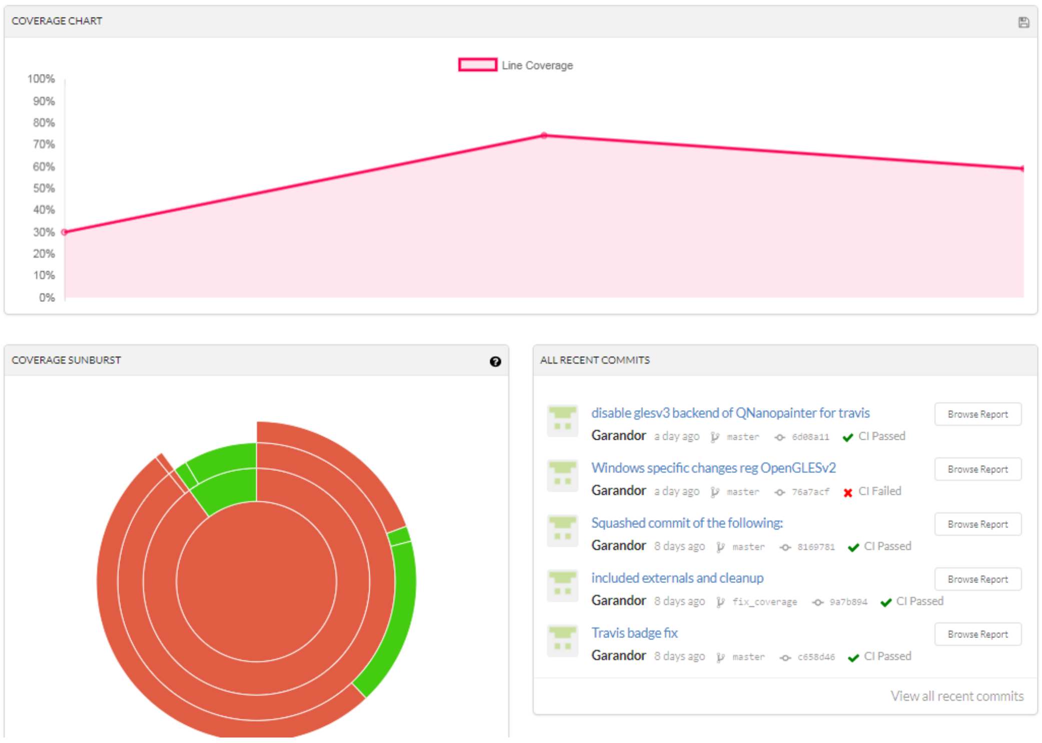
It is located at `https://garandor.github.io/pfcrender/`

All of the additional services are linked from the `REAMDME.md` displayed on the main page of the repository for easy access.

3.3 Architecture Hierarchy

A plugin architecture as described in section 2.3.1 was chosen for implementing `pfcrender`, as plugins will mainly consist of new Import/Export formats with limited intrinsic complexity and no dependencies on other plugins, avoiding the *plugin hell* scenario mentioned in section 2.3.1.

[55] `https://pages.github.com/`



disable glesv3 backend of QNanopainter for travis — Browse Report
Garandor a day ago master 6d08a11 ✓ CI Passed

Windows specific changes reg OpenGLESv2 — Browse Report
Garandor a day ago master 76a7acf ✗ CI Failed

Squashed commit of the following: — Browse Report
Garandor 8 days ago master 8169781 ✓ CI Passed

included externals and cleanup — Browse Report
Garandor 8 days ago fix_coverage 9a7b894 ✓ CI Passed

Travis badge fix — Browse Report
Garandor 8 days ago master c658d46 ✓ CI Passed

View all recent commits

Figure 3.6: Codecov.io coverage dashboard

The resulting component architecture is shown in figure 3.7. While the GUI could in theory be extracted to an export plugin as well, leaving the possibility to configure operation from the GUI was deemed useful. As this would depend on the GUI plugin receiving the `QQuickItem` config screen from the plugins, it would result in plugins having dependencies on other plugins, which is to be avoided due to the danger of leading to *plugin hell*.

A key benefit of not binding the application to model import from the L-System generator module is in extensibility. This architecture also supports a use-case where a library of pre-iterated strings describing PFCs is held in a different *PFC library tool* - for example categorized by their canonical names like the GOSPER'S FLOWSNAKE curve presented earlier - and can be integrated with the renderer by providing a new import plugin.

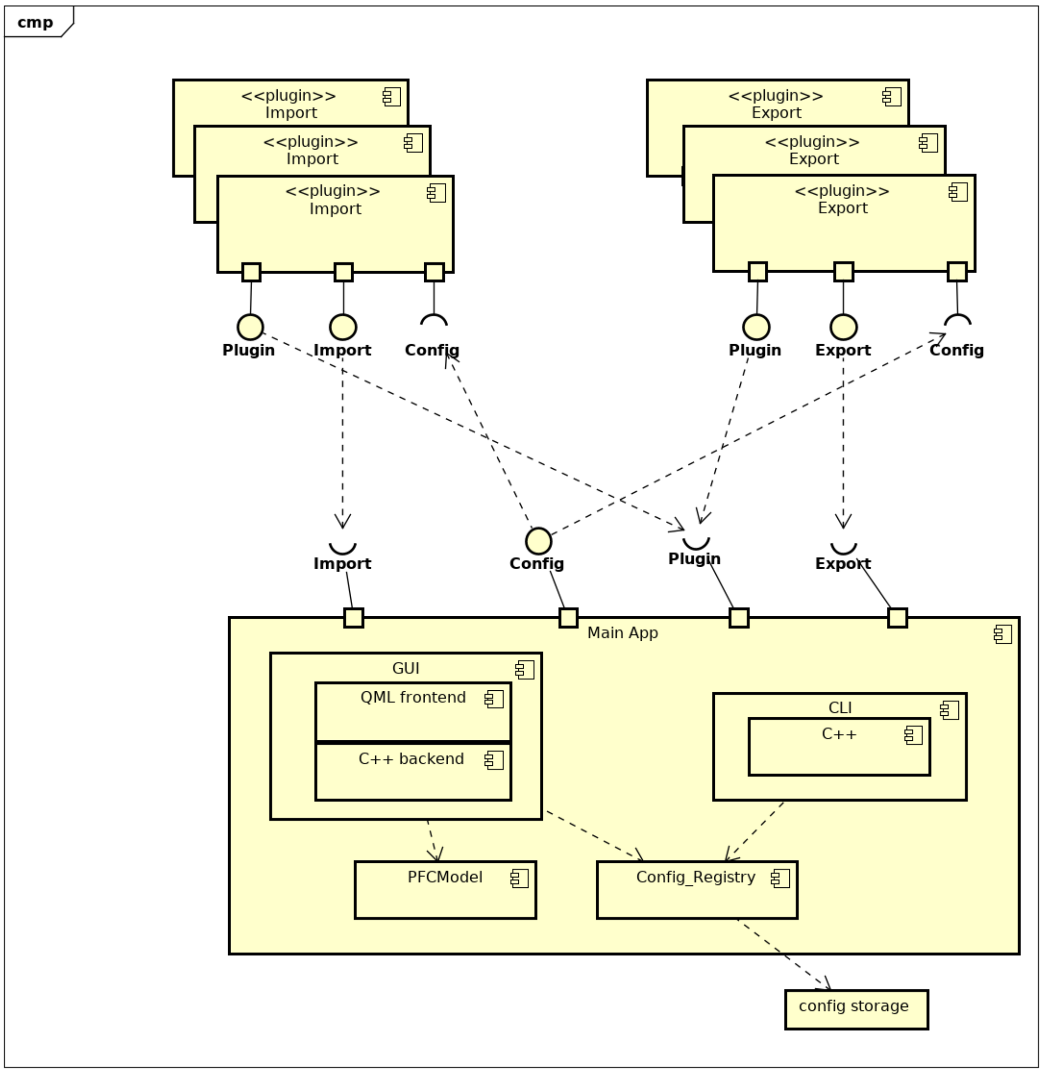


Figure 3.7: Main application and plugin separation

4 Architecture Implementation

In the following chapter we comment on the implementation of the individual decompositions as described in figure 3.7. Current source code can be found on the project GitHub presented in section 3.2.3 and a frozen state is attached to this thesis on compact disk.

It should be noted that although Astah claimed RTE support for C++, it was found to be unreliable due to poor support of the C++ namespace feature. The application was thus developed using a conventional model-driven design approach.

Code in this project is formatted in compliance to the WebKit formatting rules presented in section 2.2.3 automatically on file save using QtCreator integration of clang-format.

4.1 Main Application

The core functionality of the program was established in section 3.3 to be

- Configuration storage
- Plugin management
- Curve rendering to screen
- CLI parsing

These components lead to a further decomposition shown in figure 4.1.

Initialization and Operation The main entry point into the application is not shown in the static class diagram, but it involves creating a Qt `QGuiApplication` object and a `PFCRenderGUI` or `PFCRenderCLI` object.

In case of GUI operation, a `QQmlEngine` is instantiated to handle the declarative GUI part and the Qt event loop is executed through the `QGuiApplication`. Requested operations like L-System import or PDF export are handled by a `SequenceWalker`,

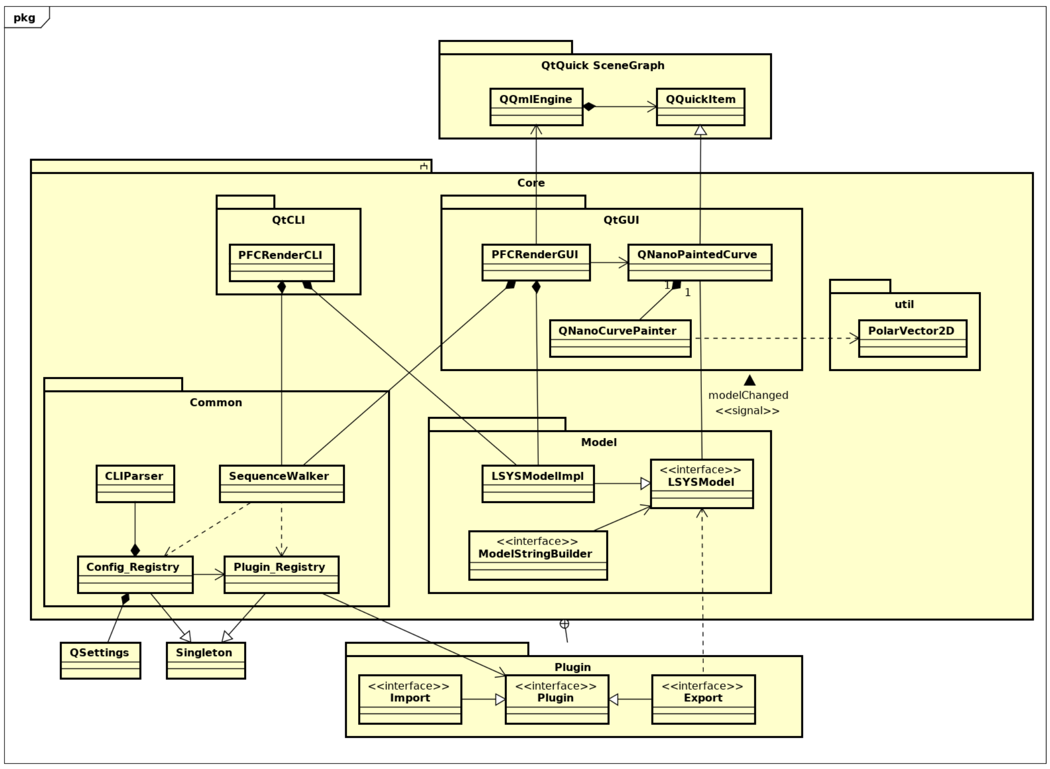


Figure 4.1: Class Structure of Core Application

upon completion of which the GUI starts handling all further interaction asynchronously through a Meta-Object System based messaging system called *Signals and Slots*.

In batch mode, the `PFCRenderCLI` simply calls the `SequenceWalker` which handles requested operations and then exits the program.

Configuration `Config_Registry` is the central store of configuration information as outlined in section 3.1.3 and - upon creation of its instance, which happens the first time some class wants to access configuration data - resolves the sequence of reading settings from file, creating the `Plugin_Registry` to query for options, and parsing the CLI. It is also available to the QML part of the application as an exported *QML Type*, allowing GUI screens written in QML to access config data.

Plugins Since Plugins need to be loaded to query for config information anyway, it makes sense to hold references to them active, so they do not have to be loaded a second time once accessed for execution. `Plugin_Registry` is a second Singleton tasked with resolving all available plugins and holds this reference associated with plugin name for simple lookup.

Model Creation The model is created once on startup of the application via the `PFCRender*` objects. Once the sequence walker executes an import plugin, it is given a `ModelStringBuilder` object to execute. Upon completion, it pushes an owning `std::unique_ptr<>` to that string to the model to avoid potentially costly string copying.

Benefit of returning a builder object instead of direct plugin invocation is supporting deferred / delegated execution, as it becomes simple for the sequence walker to hand the builder over to a thread. This keeps the GUI responsive while computation occurs in the background. This feature is planned for a subsequent release, as it was not a direct requirement.

GUI Creation The GUI is loaded from QML files on startup, the `QNanoPaintedCurve` displaying an empty rendering. It is connected to the `modelChanged` signal and - on notification of a change - copies the model string to its renderer - `QNanoCurvePainter`. It then notifies the *scene graph* it needs to be redrawn. When the rendering thread gets to the `QNanoCurvePainter` node upon re-rendering the scene graph, it invokes the `QNanoCurvePainter::paint()` function which parses the string into actual geometry. On completed rendering, the QNanoPaintedCurve notifies the QML part of the GUI if its Bounding Box has changed, in order to scale the result to the available window size.

Export The `SequenceWalker` hands the model string over to the plugin, which is then free to do its job, fetching configuration from the `Config_Registry`.

4.1.1 Challenges Faced in Rendering the Model Curve

Had performance not been a direct requirement, the QPainter API would have been sufficiently sophisticated to enable curve drawing. Since QPainter is too slow - as found in section 3.1.2 - an alternative approach is necessary.

The first studied alternative was direct insertion of geometry into a `QSSGeometryNode` scene graph node, which turned out to be too inflexible to satisfy all requirements.

An alternative was ultimately adopted with of using QNanoPainter.

We will now discuss challenges faced with both rendering methods.

4.1.1.1 QSGGeometryNode - Direct Geometry Insertion into the Scene Graph

First approach was direct creation of a `QSGGeometryNode`, which is a Qt SceneGraph frontend for direct specification of graphical primitives (*vertices* and *material*), providing a low-level abstraction on top of the rendering backends. The code used in this approach is available on the `rendermethod_QSGGeometry` branch of the Git repository.

Drawing a curve from straight lines (i.e. no rounding of corners) including on-the-fly Bounding Box creation (see section 4.1.1.3) and rescaling of the rendering to screen size through a `QSGTransformNode` was elegantly possible using this approach. As the `QSGGeometryNode` supports a LineStrip rendering style, coordinates of curve segments could be directly specified as vertices.

As for drawing arcs though, no API is provided. While a manual solution could be implemented by calculating coordinates of points along the arc and connecting them with straight lines, this "reinventing the wheel" approach was dropped in favor of the higher-level drawing API QNanoPainter, which provides a bezier curve/arc drawing command.

4.1.1.2 QNanoPainter - Constant Redrawing

During implementation, it was found that QNanoPainter forces the scene graph managed by Qt to call the `QNanoQuickItemPainter::paint()` function on every frame, with no regards whether the underlying geometry changed or not. In our case, this leads to needlessly reparsing the model string, wasting system resources and making the GUI unresponsive on higher L-System iterates.

When the scene graph collects data for rendering, it reuses cached data of a node if it has not changed for efficiency. To indicate changes that need redrawing, dirty bits are set on that node, which specify in which way a node's data has changed. Listing 4.1 shows the method called by the scene graph to query for new data as implemented in QNanoPainter.

This code calls `markDirty()` on every execution, forcing the scene graph to call the `QNanoCurvePainter::paint()` function to redraw the geometry.

This behavior is likely intended for user code compatibility in QNanoPainter, as the framework also offers drawing on `QQuickFramebufferObject` as a backend, which does not have a notion of dirtyness as it is rendered directly (as an overlay to the scene graph).

Listing 4.1: QNanoQuickItem as of commit de45f31e

```cpp
QSGNode *QNanoQuickItem::updatePaintNode(QSGNode *node, QQuickItem::UpdatePaintNodeData *
    nodeData)
{
    Q_UNUSED(nodeData)
    QNanoQuickItemPainter *n = static_cast<QNanoQuickItemPainter *>(node);
    if (!n) {
        n = createItemPainter();
    }
    n->synchronizePainter(this);
    n->markDirty(QSGNode::DirtyMaterial);
    return n;
}
```

We resolved this problem by forking QNanoPainter, removing the `markDirty()` call, adding it to the `QNanoCurvePainter::synchronize()` method instead, which is called by `synchronizePainter()` in listing 4.1. `QNanoCurvePainter` is a class to be implemented by the user of QNanoPainter, thus exposing control over whether to redraw or not to the user. A pull request to the official QNanoPainter repository, maintaining compatibility between the two backends by flipping the logic of `markDirty()` to `markUnchanged()` will be submitted shortly.

4.1.1.3 QNanoPainter - Calculating the Bounding Box

Drawing the geometry from the string is an iterative process, where the model string has to be parsed character by character. The final size and position of the curve is not known until after the full curve has been constructed. It then becomes necessary to relocate and rescale the drawing in order to fit it on the output device used (e.g. screen, printer page size). There are two approaches to achieving this:

- Go through the string once, just to gather min and max coordinates to create the bounding box. Apply a coordinate transformation calculated from the bounding box to the drawing tool and reparse the string, drawing the geometry at its final size and position.
- Pick up the bounding box while drawing the geometry. Apply a transformation on the already drawn geometry afterwards.

Regarding the performance impact of iterating through a potentially very long string a second time, the latter is obviously preferable. The iterative drawing frameworks QPainter and QNanoPainter do not provide any facility to apply transformations after drawing has been done, as they place actual rendering calls using the current rendering engine.

A workaround to this problem was found for the GUI, where rendering performance is critical. Using the `QSGRenderNode` backend, QNanoPainter does not issue drawing calls directly upon execution, but queues them up to be drawn when the scene graph reaches the current node. It is possible to apply the bounding box transformations to a parent node an the scene graph, which - as per the logic of the scene graph - applies to all child nodes and transforms the coordinate system of the executed drawing calls.

As no such deferred drawing is possible when using `QPrinter` to print to PDF or `QSVGGenerator` for SVG, the bounding box is precalculated and applied to the calls directly. Since file export is not a time critical operation, reparsing is considered tolerable in this case.

4.1.1.4 QNanoPainter - Copying of the Model String

A problem introduced by using the QtQuick scene graph to render the geometry is threaded execution. Due to Qt being a GUI framework, where application responsiveness is a main design goal, GUI operation is decoupled from rendering by running the latter in a separate thread. This results in the `QNanoCurvePainter::paint()` method, which parses the model string to be executed on the rendering thread. This method accesses a reference to the model string, while the GUI thread running in parallel could potentially modify the model while it is being parsed, which is unsafe. QNanoPainter provides a `QNanoCurvePaintedItem:::synchronize()` method, which can be used to synchronize variables from the GUI context to the Painter object while the GUI thread is blocked, avoiding this problem by copying the model string from the model to the QNanoCurvePainter every time it changes. Though this is suboptimal for performance, it cannot be avoided without implementing an addition mutual exclusion mechanism solution.

4.1.1.5 OOP - Encapsulation of Functionality vs. Method Call overhead

Switching execution context - i.e. calling a subroutine - is a multistep process[56], which can in some cases impede program performance.

If a method is simple but executes a billion times, the cycles spent saving and restoring stack frames will add up to become a significant percentage of the execution time of the program.

[56]`https://en.wikipedia.org/w/index.php?title=Call_stack&oldid=824753306`

As methods / subroutines are an essential abstraction of high-level programming languages and usage can rarely be avoided unless code duplication is tolerated, modern compilers help mitigate this problem by automatically *inlining* function calls where useful.

Inlining a function means copying the content of its body to its place of execution, avoiding the indirection of the function call. This skips stack frame saving and a jump/return instruction pair, but leads to the function body being duplicated on every execution, increasing executable size.

Since - contrary to embedded systems - program space and RAM are usually not a bottleneck on desktop systems, it often makes sense to ignore the larger executable size and optimize a program for runtime performance (e.g. for the GNU compiler: `g++ -O2 program.cpp`) leading to various optimizations like inlining being performed automatically.

While compilers do this type of optimization automatically, C++ syntax has a keyword *inline*, which - when prepended the function definition - hints to the compiler that this function *should* be inlined for performance. *Inlining* is extensively used in model string parsing to encapsulate character handling code in methods, without incurring the performance overhead.

4.1.1.6 Polymorphism and Performance

Although C++ strives to provide zero-performance-overhead abstractions, some functionalities come at a performance cost.

Polymorphic classes are a key abstraction of C++ and are distinct from simple inheritance by the presence of the *virtual* keyword on one or more of their members.

This keyword enables a derived class, referred to through a pointer to the base class to execute its own implementation (an *override*) of the *virtual* member instead of the one (possibly, but not necessarily) implemented by the base class.

While this mechanism is fundamental to enabling C++ to provide the so called *interface abstraction*, it comes at a performance cost.

Calls to a *virtual* function are not made by directly jumping to the address of the callee as in the above case after saving the stack frame. They access a so-called *vtable* first, which is tasked with resolving the address of the member function. This additional indirection can become a significant bottleneck for performance when the *virtual* function is called often, e.g. in our case, when parsing a character of the model string.

Parsing the model string was originally done in its own class to keep the code for parsing encapsulated and localized. Since this parsing is accessed by multiple other classes (`QNanoCurvePainter` and every export plugin), each have their own implementations for handling specific actions, e.g. `AddSegment()`. The standard OOP way to implement this functionality is in having the `ModelStringParser` be a class with *virtual* members. For long model strings, this - as explained above - would result in countless *vtable* resolutions, significantly hampering parsing performance as *vtable* calls are runtime dependent and *cannot be inlined* by the compiler.

As a solution to this encapsulation vs. performance problem, we opted to implement the common functionality as a class member in a declaration/definition preprocessor macro pair, which is then included in each class that uses the parser.

The macro is located in `src/Model/ModelStringParser.h` and basically amounts to copying the parsing code to each implementation class. It avoids code duplication and makes the parser function local to each implementing object at compile time, allowing inlining.

4.2 Plugins

A plugin is defined through the implementation of its interface class - currently import or export. Each plugin is supposed to publish its information in a common `PluginInfo` structure and is optionally allowed to expose a `ConfigScreen` item for configuration through the GUI.

In addition to the functional interfaces, a plugin must also inherit from the `QObject` class and expose some metadata to the Meta-Object System to be importable via the `QPluginLoader` mechanism.

Four plugins were implemented in the course of this thesis

- L-System Import
- Export to SVG
- Export to PDF
- Export to STDOUT

With the exception of the STDOUT exporter, which is trivial and will not be discussed here, we discuss the plugin implementations.

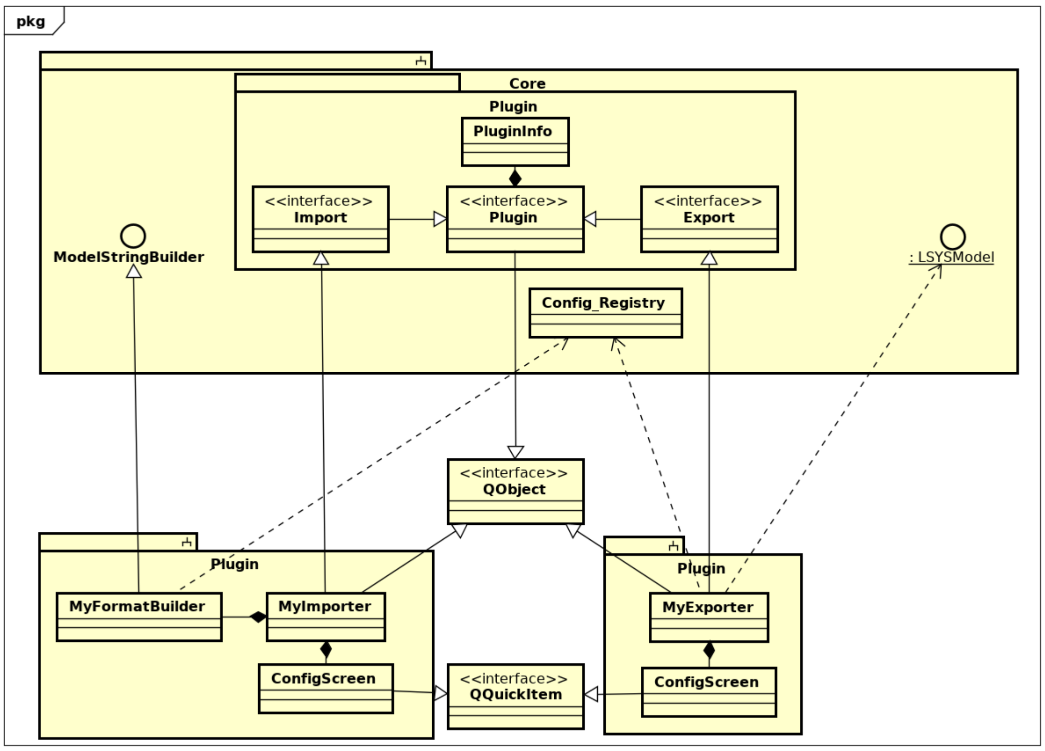


Figure 4.2: Plugin Interface

4.2.1 L-System Input

This plugin implements the iteration of a given L-System description as shown in table 1.1 to generate a model string. As this task is critical to overall application performance, an efficient algorithm for iteration has to be selected.

We decided to use the `stringsubst` algorithm from the *fxtlib*[57] as recommended by Prof. Arndt.

4.2.2 SVG Output

As stated in section 4.1.1.3, SVG and PDF output both precalculate the bounding box and apply the resulting transformation to `QPainter` before issuing drawing calls.

A SVG - unlike PDF - does not have a fixed maximum drawing or paper size, thus the painter need only be translated to avoid drawing at negative coordinates.

[57] `https://jjj.de/fxt/fxtpage.html`

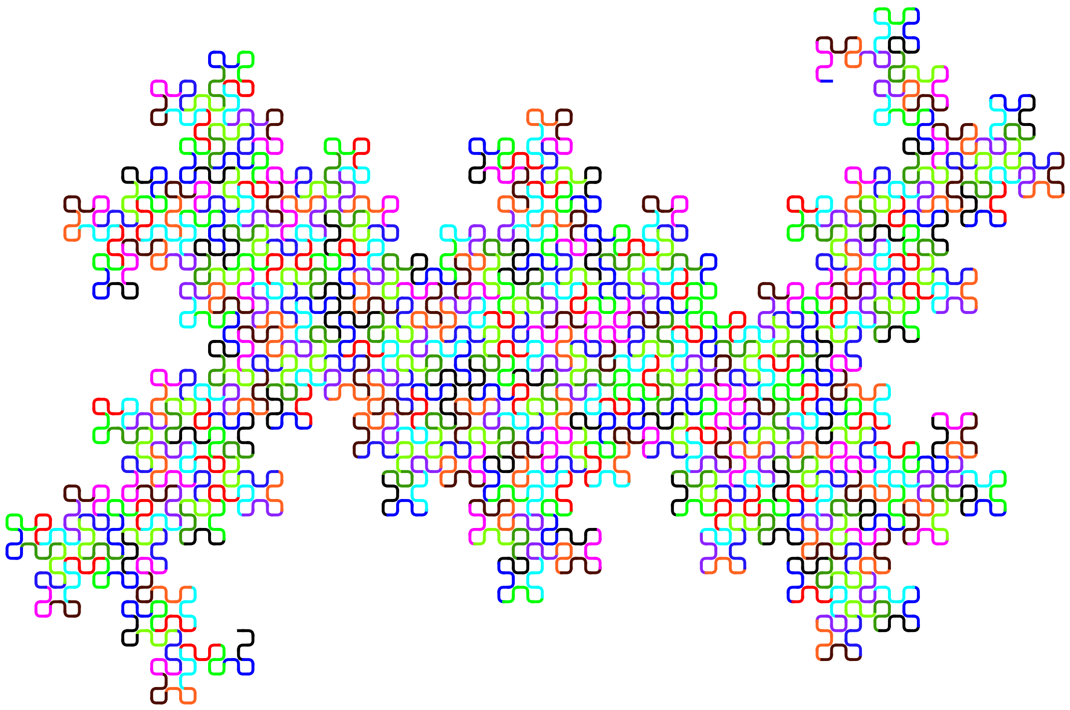


Figure 4.3: SVG Output of a 5th iterate R5-DRAGON

Figure 4.3 shows a SVG rendering of the R5-DRAGON converted to pdf for including using the LaTeX package svg. The R5-DRAGON is a PFC on the square grid, whose L-System is given in Arndt [2016][p.23] and was created using

```
$ ./pfcrender lsys --clear --it 5 --rules "F F F+F+F-F-F_ + + - - _ _ ~ ~" --sw 2 --sl 10 --
    rd 0.33 --batch svg --a 90
```

4.2.3 PDF Output

PDF Output is implemented to print to a DIN A4 landscape page and scale to fill it while keeping the aspect ratio of the original curve. In subsequent releases, configuration of the output parameters will be implemented as CLI and GUI configuration options.

Figure 4.4 shows a pdf rendering of GOSPER'S FLOWSNAKE obtained by running

```
$ ./pfcrender lsys --rules "L_L L L+R++R-L--LL-R+ R -L+RR++R+L--L-R + + - - _ _ ~ ~" --sw 2
    --sl 10 --it 4 --rd 0.5 --ia -60 --a 60 pdf --batch
```

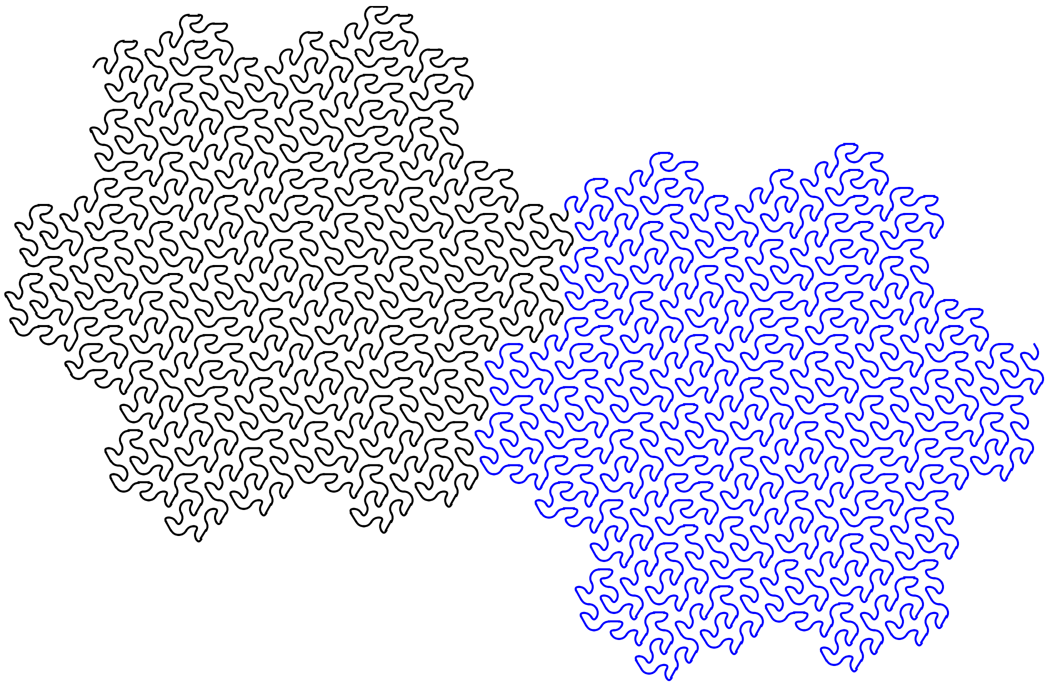
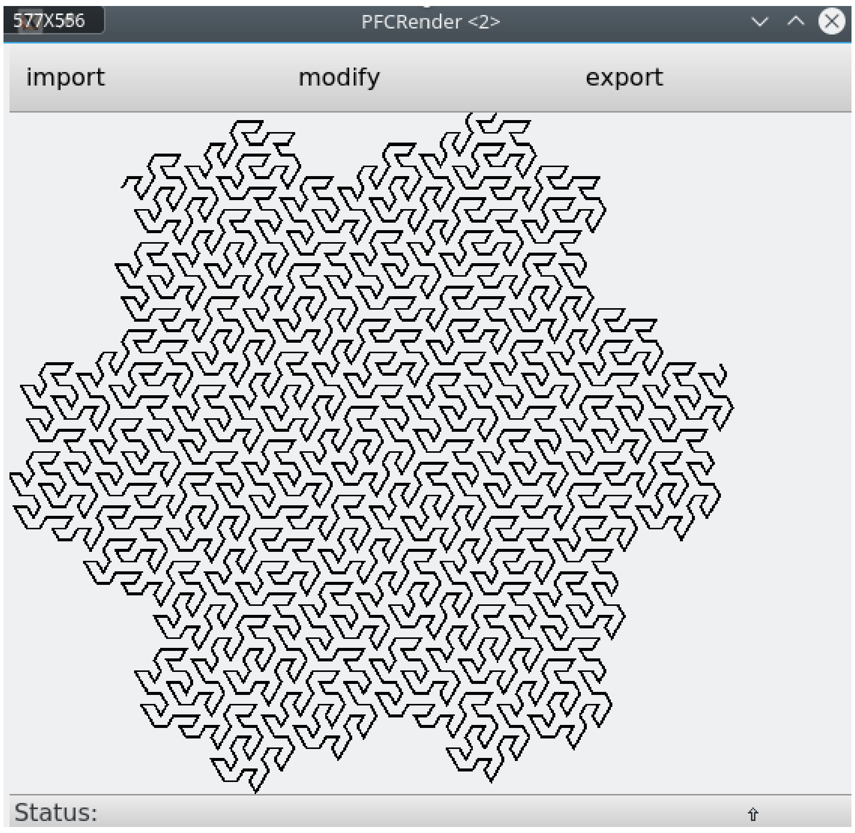


Figure 4.4: PDF Output of *Gosper's flowsnake*

5 Application Performance

A short study of the rendering performance of our solution is conducted to verify the targeted 10fps for a "simple image". We define the fourth iterate of GOSPER'S FLOWSNAKE, rendered with no rounding as a "simple image" and conduct the following benchmarks using its L-System, invoked as

```
$ ./pfcrender lsys --rules "L L L+R++R-L--LL-R+ R -L+RR++R+L--L-R + + - - _ _ ~ ~" --sw 2 --
    sl 10 --it 4 --rd 0 --ia -60 --a 60
```



Measurements are taken in an Archlinux virtual machine on an Intel Xeon E5 system at 3.5Ghz clock frequency and the program is compiled in release configuration with the following optimization flags

```
-O3 -DNDEBUG -flto=full
```

5.1 Batch Mode Execution Timing

The following measurements are taken using the following shell command based on the Unix `time` utility. It executes the application 100 times in batch mode and averages the reported execution times of each run.

```
for i in {1..100}; do { time ( ./pfcrender --clear --batch lsys --rules "L L L+R++R-L--LL-R+
    R -L+RR++R+L--L-R + + - - _ _ ~ ~" --sw 2 --sl 10 --it 4 --rd 0 --ia -60 --a 60 ) } 2>&1
    |tail -1| sed 's/.* //'|cut -d" " -f1|cut -c1-4 ;done | awk '{ total += $1; count++ } END
     { print total/count }'
```

We compare the runtime executing different plugins in batch mode.

Testcase	Application runtime[ms]
Import only	68.7
Import & SVG output	73.3
Import & PDF output	76.1

Table 5.1: Execution timings for the 4th iterate of GOSPER'S FLOWSNAKE

It is apparent from these results, that the actual execution of a plugin's routine has a low impact on application runtime when the model is simple, as timings are dominated by application startup overhead. Irrespective of this effect, the proposed 100ms rendering time target is reached, even though SVG and PDF use the slow QPainter drawing engine and access the filesystem.

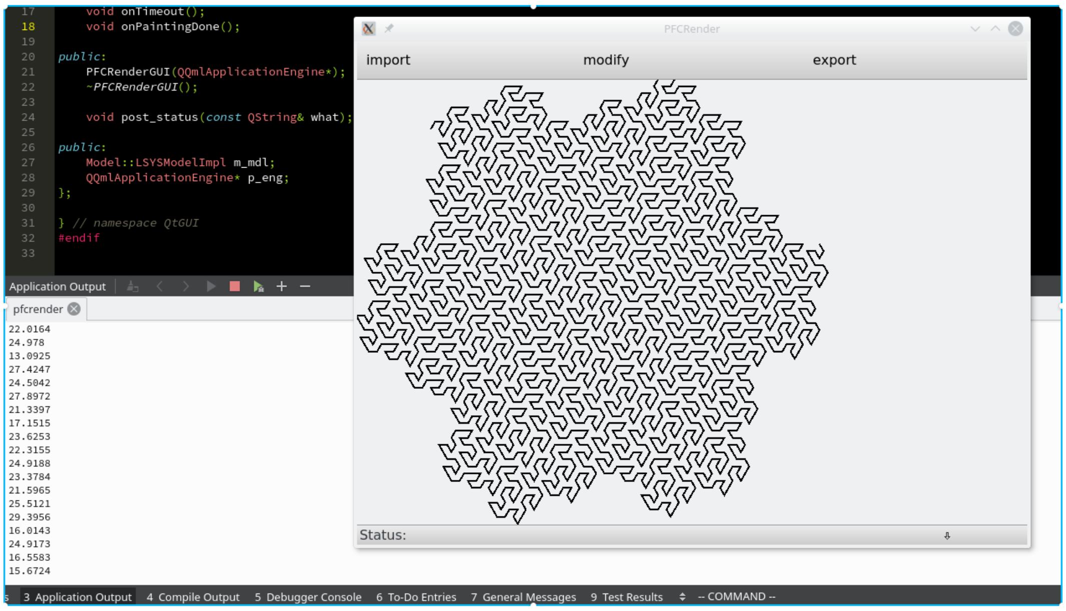
5.2 GUI Rendering Timing

As the timing of application startup and input plugin execution is known from the measurements in section 5.1, we benchmark the timing from triggering a model change to finishing of the rendering.

In order to obtain execution timings within the code, a test case was built into the sources, which can be found in Git as tagged commit `benchmark_17-2-12`.

It starts a timer, feeds a pre-iterated model string of GOSPER'S FLOWSNAKE to the model and waits for the `QNanoCurvePainter` to signal completion of its `paint()` method, after which the rendering is finished.

Reporting completion via the Signals/Slots mechanism is not a precise way of timing measurement as signal notifications are asynchronously handled by the GUI event loop. This can introduce additional delays to the timing, depending on when



```
    void onTimeout();
    void onPaintingDone();

public:
    PFCRenderGUI(QQmlApplicationEngine*);
    ~PFCRenderGUI();

    void post_status(const QString& what);

public:
    Model::LSYSModelImpl m_mdl;
    QQmlApplicationEngine* p_eng;
};

} // namespace QtGUI
#endif
```

```
22.0164
24.978
13.0925
27.4247
24.5042
27.8972
21.3397
17.1515
23.6253
22.3155
24.9188
23.3784
21.5965
25.5121
29.3956
16.0143
24.9173
16.5583
15.6724
```

Figure 5.1: GUI Rendering Cycle Measurement

application control flow returns to it. As we are interested in the lower bound of performance though, this is not an issue here.

Timings are printed to console and - as shown in figure 5.1 - consistently below 30ms. The total GUI rendering time, consisting of import from L-System as measured in table 5.1 (70ms) and this rendering time is thus confirmed to be just below the 100ms mark required in figure 1.3.

6 Summary

This work originated from the need for an *extensible, maintainable, cross-platform* solution for rendering PFCs to facilitate ongoing research in the field. These criteria span a wide range of software engineering techniques. The most accessible to small-scale open-source projects of these were selected for implementation of the architecture. The result is a *self-documenting, self-testing and self-deploying open-source project that encourages collaboration*.

The software architecture itself was designed specifically for extensibility by isolating non-core functionality to plugins and holding configuration information in a central, automatically filled repository and guarantees cross-platform operation by building on top of the Qt Framework.

The resulting implementation is a solution to most requirements given in fig 1.3 by design, the required rendering performance of 10 frames per second was verified via benchmarking.

At the time of writing, SVG (section 4.2.2) and PDF (section 4.2.3) plugins are in a proof-of-concept state and a possibility of configuration via GUI has not been implemented. Along with extending the application, e.g. by implementing a *PFC library tool* like proposed in section 3.3, refining GUI functionality and increasing the configurability of the export plugins are recommended improvements to the existing solution.

Rendering performance of the application currently is limited through the one-shot characteristic of model creation, as no in-program facility of generating a new PFC yet exists apart from specification on the CLI, resulting in rendering of multiple PFCs requires application restarts. This incurrs the startup time delay of plugin loading each time. A GUI embedded configuration screen for the import plugin or a facility for rerunning imports from within the application would thus be a desirable extension.

Bibliography

J. Arndt. Plane-filling curves on all uniform grids. 2016. 1.1, 1.2, 4.2.2

L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015. 2.2

M. Bray, K. Brune, D. A. Fisher, J. Foreman, and M. Gerken. C4 software technology reference guide-a prototype. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1997. 2.2, 2.2, 2.2

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 2.3.2.1

A. L. N. Johansson. Designing for extensibility: An action research study of maximizing extensibility by means of design principles. 2009. *URL: https://gupea. ub. gu. se/bitstream/2077/20561/1/gupea_2077_20561_1. pdf (hämtad 2017-05-07)*. 2.3

P. Prusinkiewicz and J. Hanan. *Lindenmayer systems, fractals, and plants*, volume 79. Springer Science & Business Media, 2013. 1.1

List of Figures

List of Tables

Listings