GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Master's Thesis

submitted in partial fulfillment of the
requirements for the course "Applied Computer Science"

GroLink: implementing and testing a general application programming interface for the plant-modelling platform GroIMP

Tim Oberländer

Institute of Computer Science

Bachelor's and Master's Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

22. December 2023

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

First Supervisor: Prof. Dr. Kurth
Second Supervisor: Prof. Dr. Buck-Sorlin

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 22. December 2023

In its present state, the graph-grammar based simulation software GroIMP is mostly designed as a stand-alone modelling environment, where the start and the end of the workflow are located in GroIMP itself. This is one advantage of the software since it makes it easy to install and use for modeling and teaching. Yet over time, several projects appeared that were trying to integrate GroIMP as one part of a bigger pipeline. Although most of these projects found individual ways, these solutions are often laborious and only suited for the specific project.

GroLink is approaching this issue with a new general application programming interface (API). APIs are designed to enable the interaction between a server, in this case GroIMP and a client application, without the need for human interaction. In this work, the communication is based on HTTP and therefore a client can be implemented in almost any software environment. Since HTTP requests can be transferred between systems, it will also be possible to run GroIMP on a remote server or in a virtual environment. The API calls are intended to cover all basic uses of GroIMP and allow the client to create or open multiple models, manage and edit source files, execute RGG and XL commands and functions, import objects and receive information and results of the simulation including 3D exports.

One intermediate step of this project was the creation of a new generalized foundation for applications within GroIMP to improve its ability to work with different user interfaces (applications) like Grolink and the command line interface (CLI).

To simplify the usage of the GroLink server, an object-oriented client library concept is designed that is supposed to offer all connections, information and functionalities to client software. This library concept is fully implemented in Python (GroPy) and simplified in R (GroR).

To test the abilities of GroLink different use cases were designed and implemented, all of them inspired by existing projects or requests made to the developers.

Firstly, inspired by the work of Qinqin Long and Christophe Pradal on the co-simulation of OpenAlea and GroIMP, a GroPy-based automated exchange graph (XEG) processing pipeline was created that follows similar steps as the GroIMP side of the original connection.

The second use case is inspired by the work of Benjamin Spehle presented at the FSPM2023 on a GroIMP web interface in RShiny. In this use case, two interactive web applications were created using the GroR library, a generalized viewer for GSZ models including RGG/XL interaction and an interface to manipulate the environment of a specific plant model.

The third use case was discussed by different teams in the past and is mostly focused on the interaction with the 3D scene: The life integration of a GroIMP model into a game engine. To demonstrate that, different forest models were added to a game engine where a user can walk through and cut or plant trees and trigger growth steps which are then simulated in GroIMP.

In addition to the use cases, parallelized sensitivity analysis and digital twinning are also briefly introduced as possible use cases with small examples.

Contents

Chapter 1

Introduction

1.1 Motivation

The plant modelling platform GroIMP was first released 20 years ago and used in countless projects by students and researchers ever since. These projects as well as GroIMP and the general usage of computers evolved over time. In some cases, the focus of these projects shifted more and more towards the integration of GroIMP into bigger pipelines or other software.

Due to the current nature of GroIMP as a mainly graphical development environment, this kind of integration is not readily possible, so the various projects found individual solutions, as I did with my last project. During this project: "A Framework for a Digital Twin of a semi-closed greenhouse using the FSPM platform GroIMP", I experimented with the auto-generation of model reports and found this much more difficult than expected. Furthermore, the nature of the digital twin calls for an integrated approach.

After learning about other projects struggling with similar problems, I decided to develop a more general approach to tackle this problem with a solution that is integrated into GroIMP and can be used by different projects. It also seems to be a good addition to the software, opening up new possibilities for future projects.

1.2 Background

1.2.1 GroIMP

GroIMP (Growth Grammar-related Interactive Modelling Platform) is a software project initially designed and implemented by Ole Kniemeyer as part of his dissertation [1]. One of the main features is its own programming language XL which combines the high-level programming language Java with syntax for RGG. RGG (Relational Growth Grammars) is a generalization of

parallel graph grammars extending L-system [2] and provides rules and queries for working with the simulation graph. Besides that, several other features were implemented or added over time, including GPU-based raytracing, 3D modelling and several import and export interfaces.

GroIMP comes with a full graphical environment, including 3D and 2D views, a code editor, graph viewers, a console and explorers for model elements such as files, datasets, lights and shaders. This environment also enables the interactive application of rules, defined in the XL-Code. Besides this graphical user interface (GUI), GroIMP can also be used headless or as a simplified webserver to automatically run a predefined simulation without interaction.

Several models can be executed simultaneously in so-called workbenches and functions from different plugins can be applied, this leads to a quite complex software architecture which has even grown over time.

The focus of the following introduction to GroIMP is on the program structure to give an overview of the concepts and the architecture in which the new API will be embedded.

Registry

In GroIMP, the main structure that holds all information is the registry, which itself is a tree, similar to a computer file system. There can be several registries all extending their parent registries up to the root registry, which is created in the boot process. For instance, every workbench has its registry. A registry holds the majority of information on GroIMP and the models as leaf elements called items. This information includes the commands that can be used, the plugins that are loaded, the file interfaces for import and export, the available RGG functions and references to the model elements such as SourceFiles and datasets. To access this information each registry item can be queried from a GroIMP or an RGG function. It is also possible to resolve a registryitem, by querying the registry itself and then going stepwise upwards to the parent registry until the item is found or the root is reached.

The root registry is initialized based on the combination of the plugin.xml files of all added GroIMP-plugins. Therefore any new plugin can add elements to GroIMP. Moreover, the different registries can be manipulated during the runtime by Java code or by the execution of a hook, which are sets of predefined registry operations that can be triggered by events. This manipulation also includes the ability to override existing registryitems.

Commands

Most interaction with GroIMP is supposed to be a command pushed to a JobManager. A command is a wrapper around Java code that is applied on a context (for instance a workbench) and a second customizable object that can be adapted to different needs, such as transferring information or objects. The most common implementation of a command is the CommandItem, which is used to

link registry items to static Java functions without return values. These items can be linked to for instance the menu of the GUI representation of workbenches.

Factories

Similar to CommandItems factories are registry items and can be queried or resolved and then executed. In opposite to CommandItems factories create Java objects, such as panels or filters, depending on the properties given in the registry. GroIMP comes with a variety of factories for different elements. They are used for instance to get the parsers of the different files or to open explorers and other panels.

Plugin structure

The GroIMP software structure is plugin-based, meaning that almost all functionalities are in plugins. Some of these plugins are necessary for the basic usage of GroIMP while others only provide additional features such as video generation or handling of a specific file format.

The core part of a plugin is the plugin.xml file, where, as described above, the registry items, that are supposed to be added, are stored. Additionally, in the plugin.xml file, the dependencies and the namespace of the plugin are defined. Besides that, the plugin is likely to contain either Java files, other assets like images or GSZ files, or both. The content of those is connected to GroIMP with registry items such as CommandItems or factories. Moreover, it is possible to link a Java class, which extends the class Plugin, to the plugin.xml. This Java class then will be initialized directly when the plugin is loaded.

Finally, a plugin contains the plugin.properties file which should define all human-readable content of the plugin such as the names of functions or the text of messages. This file can be created in different languages to translate GroIMP.

Application

By starting, GroIMP executes two basic phases, first the plugins are loaded and the rootregistry is created, then afterwards, an application is started.

An application in GroIMP is registered as an item that references a Java class that includes a function called `public static void run (Application app)`, which is executed when the application is started. At the current state, the only existing application is the one that starts the GUI. Nevertheless, the design clearly shows that other applications were considered. Contrary to what could be expected the headless mode is not an individual application but a different mode within the one existing application.

Toolkits

As already mentioned the GroIMP GUI includes several panels such as explorers or viewers for files, graphs and other content. These panels should not be part of the GUI application but should be defined by panel factories and displayed with the help of toolkit commands. These toolkits are designed to visualize the panels, meaning that the SwingToolkit (the one for the GUI) would interpret the commands of for instance an explorer factory to visualize it to a panel with a tree view including the object to explore. While another toolkit could create a text-based representation of the same panel.

This design was intended to enable the creation of other visual representations without editing the individual panels, sadly some panels, especially most of the ones added later, do not follow this design. Besides the SwingToolkit currently only a headless toolkit, which displays nothing, is implemented.

Workbenches

As mentioned GroIMP can open multiple projects at the same time. The organizational unit for that is the workbench. Each newly created or opened project is loaded into an individual workbench, which includes a registry, a project graph, the main window and the JobManager.

For each workbench, a new child registry is created to which resources such as files or shaders and later compiled classes are added. The registry, as described above, is individual for each workbench. The project graph holds the actual simulation meaning the graph created by the RGG commands or the interactive 3D modelling tools. For the main window, the implementation depends on the toolkit, but in general, it is the place that holds all opened panels and manages their actions and disposal.

The last main part of each workbench is the JobManager, which acts in two functions. First, the JobManager is the runnable class that creates a thread for its workbench which gives GroIMP the ability to run multiple workbenches on multiple threads. Secondly, it takes care of the incoming commands. As mentioned above commands represent most interactions with GroIMP and therefore with the workbenches, the JobManager organizes these interactions, to ensure the right order of execution and the proper synchronization of resources. In a simplified way it puts the commands in a queue and executes them one by one.

1.2.2 HTTP

The hypertext transfer protocol (HTTP) is a stateless messaging protocol that is used to communicate between computers, namely between servers and clients. This protocol can be described as the base of the Internet [3].

URL

A URL (Uniform Resource Locator) is a specific type of URI (Uniform Resource Identifier) and is used to describe the location of a resource that can be reached using HTTP. A URL can be separated into 5 parts, starting with the host and the port. The host is the general address of the server that provides the resource. It is defined by either an IP address or a domain name. The port is used to address the specific process that is listening on this port, such as a web server or an SSH client.

Besides the host and the port, the URL is defined by the path to the resource and an optional query, which is defined by sets of keys and parameters. This query can be used for instance to describe what part of the resource should be returned or in which format. Additionally, a flag can be set to define further preferences [3].

Message

The Message contains all needed information and flows between the client and the server. The message sent from the client to the server is called a request message and the answer sent from the server is named response message. The request message describes the task the client wants the server to perform. This includes the method, the URL, if needed more header fields and in some cases a body that for example contains a file to upload to the server.

The response message contains a status code, the requested content and if needed more header fields [3].

Methods

The HTTP standard contains different methods, that can be defined in the request message, such as GET to get information, POST to add information or DELETE to remove information. The exact handling of these different methods fully depends on the application that processes the request. Only some requests such as POST contain a request body [3].

Status codes

To generalize the status of a response a numeric code is used, to describe if a response was successful, is still ongoing or in case of an error if the error comes from the client or the server. These codes are defined by 3 digits where the first digit describes the category of response and the last two describe the exact response [3], [4].

1.2.3 API

Robillard et al. define an API (Application Programming Interface) as: “the interface to a reusable software entity used by multiple clients outside the developing organization, and that can be distributed separately from environment code”[5]. The general term API is defined very broadly

and can describe multiple software scenarios from operation systems and libraries to web services for data science and mobile apps. For this work, the term is used to describe language-independent remote APIs such as REST or SOAP. These APIs follow an architecture that can be separated into server and client which are communicating over a standardized protocol, in most cases HTTP or something similar.
The communication is initialized by a request from the client and a response from the server. Similar to a webpage, this communication is not limited to one server and one client.

SOAP

SOAP (Simple Object Access Protocol) is an older and more regulated approach for API interaction. The communication between the server and the client is defined in XML-based envelopes and follows a quite complex structure. This leads on one hand to a more complex implementation and communication, but on the other hand increases clearance of the server's functionalities, since the function, the parameters and the potential results are defined [6], [7].

REST

In contrast to SOAP, REST (Representational State Transfer) is not a defined protocol but an API style that is partly oriented on the concepts of HTTP. Therefore an API is not implementing a protocol but can be considered as RESTfull. To do so the API must follow a server-client model and each request is supposed to work independently from the other and without knowledge of the server (stateless). Moreover, all resources and information are presented in a standardized way (mostly JSON) and can be found by a unique URL/URI. In comparison to SOAP, a RESTful API is faster and more flexible, yet less secure and reliable [7], [8].

1.2.4 Software Integration

Software integration aims to link different tools with different strengths to a combined software system that can fulfill a specific task. Furthermore, the usage of this software system should be similar to the usage of a single tool, meaning that the additional user interaction with this pipeline should be as minimal as possible.[9], [10]

From the 5 types of tool integration described by Wasserman [10], data and control integration are interesting for the communication between GroIMP and other software.

Data integration describes the ability of different tools to communicate through shared formatted files or datasets [10]. Due to the plugin design and past development, GroIMP can handle quite a range of formats for importing and exporting and is therefore quite equipped for this kind of integration. The second type, control integration, focuses on the direct communication between tools through events or notifications [10]. This type is not yet well included in GroIMP.

1.3 Related Work

1.3.1 Funktions- / Strukturorientierte Pflanzenmodellierung in E-Learning-Szenarien

In the work of Dirk Lanwert: "Funktions- / Strukturorientierte Pflanzenmodellierung in E-Learning-Szenarien" [11] the usage of the HTTP server of GroIMP [1] is described.

The HTTP server is used to open a specific GroIMP project. This project uses the RGG startup function, which is executed every time a module is loaded, to start the simulation, similar to the usage of the headless mode . In this newly created workbench, the HTTP connection can be accessed and further information can be extracted or response data can be added. Yet there is no way to reconnect to an open workbench with a new HTTP request and in his work Dirk Lanwert suggests a more interactive access to the GroIMP model.

1.3.2 The integration of different functional and structural plant models

In his dissertation [12] and in surrounding work [13], [14] Qinqin Long worked on the co-simulation of different FSPM platforms, mainly OpenAlea [15] and GroIMP, using the XEG (EXchange Graph) format.

While this work mainly focused on the structural integrity of the transferred graphs, a server-client model called middleware was implemented and tested. To enable this connection a specific GroIMP plugin is used as a specifically for this purpose created HTTP server that can be accessed from an OpenAlea plugin. With this connection, OpenAlea was able to turn a Multiscale Tree Graph (MTG) [16] into an XEG and transfer this to GroIMP. Then GroIMP was able to turn the XEG into an RGG graph, manipulate the plant, export it to XEG and transfer it back to OpenAlea, where it could be turned into an MTG that included the changes done by GroIMP. Additionally, it is possible to send XL-code with the XEG form OpenAlea to define the manipulation of the RGG graph in GroIMP.

This project provides important proof of the concept of co-simulation and the usage of XEG, yet the developed GroIMP plugin is highly specific to this exact use case.

1.3.3 Conveying the Effects of Climate Change - Interfacing Virtual Riesling with an interactive R shiny application

At the FSPM2023 Benjamin Spehle presented his work [17] on a new web-based user interface for a specific GroIMP model.
The used workflow is very straightforward, the web application creates a configuration file based on the user input and starts a headless simulation of the GroIMP model which then reads the configuration and creates output data that can be shown in the web application.

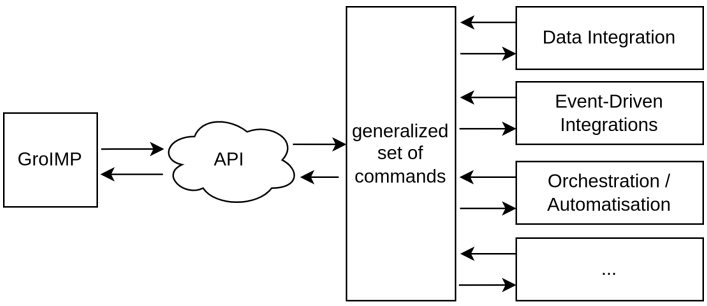


Figure 1.1: A structure of the planned approach showing the role of the API and the set of generalized commands.

1.3.4 A Framework for a Digital Twin of a semi-closed greenhouse using the FSPM platform GroIMP

This project [18] is the result of my internship at the University of Göttingen and the Institut Agro Rennes-Angers at Angers. The model created visualizes an existing semi-closed greenhouse in Angers. This project aimed to use this model as a digital twin capable of receiving measurements from the physical twin in real-time to calibrate the simulation. During the project, it was pointed out that an API could be an important tool to achieve this goal. As a first step towards an integrated model, an automated pipeline was created to run a headless simulation and generate a PDF report of the results using R and Rmarkdown [19]. To do this, a set of calibration files must be provided to the model in a specific folder, the model then creates log files that are used to generate the report when the simulation is complete.

1.4 Outline

To give this project a bit more shape and to clarify the goals, the areas of application and their needed functionalities will be explained in the following. Based on these, the generalized requirements of the project will be defined. This includes the requirements on the API -Server itself as well as on the set of commands that need to be implemented to fulfill the demands of the different applications (Fig. 1.1).

1.4.1 Areas of application

The areas of application are the result of the combination of known projects (Related Work 1.3) and the different fields of automatization and integration. The classification of projects and fields to the areas can not be done clearly, because most projects cover several areas of application and the areas of application mostly execute different types of automatization and integration at the same time. Additionally, it is important to clarify that in most cases the areas of application could already be covered with GroIMP and the API is only aiming to provide a better and more general way.

GroIMP as the source of 3D models

The creation of 3D models is one key feature of GroIMP as a rule-based 3D modelling platform, additionally, the further usage of these 3D models is enabled by the different possible export formats. Nevertheless, even with the improvements in headless 3D export made in GroIMP 2 [20], it is not easily possible to manipulate models from and export the 3D result to another software.

With an API the solution would be to run GroIMP in the background, and have another software *open or create a simulation* and then either directly *request the 3D model* or *run a simulation step* and then request the model. This field of application would contain approaches similar to the work of Dirk Lanwert or different types of data integration.

GroIMP as a file converter

This area is relatively small but still might be of interest due to the large range of different file formats that GroIMP can handle. Once again this is already possible in GroIMP by creating an empty project, importing the given file and exporting the result to the needed format. While this is quite feasible for one or two files it will be very time-consuming for larger quantities. Therefore the API could be used to create an automated pipeline, that: *create an empty project*, *import* the given file, *export* the needed file and *clean* the project for further usage or just *close* it.

GroIMP as an interpreter of FSPM models/commands

While GroIMP comes with a fully functional graphical user interface, there might be use cases where a simpler or more specific user interface is needed. An example of that can be seen in the project of Benjamin Spehle on R-Shiny web applications 1.3.3. Yet one key feature of GroIMP, the ability to interact with a running simulation through RGG functions or XL queries, is very hard to enable for this kind of interface since this interaction is currently limited to the GUI.

The API aims to change that by creating the possibility to *compile* loaded models and *execute given RGG commands* as well as *using XL queries*. Additionally, this approach could be combined with the usage of the API as a 3D model provider to enable the graphical representation. Moreover, the *project graph* and possibly *collected data* should be accessible and the *manipulation of attributes of the nodes* would be interesting. Finally, the *receiving of the manipulated model* similar to saving it would be a key feature.

GroIMP as an external service for manipulating models

The integration of GroIMP as a background service to manipulate plant models was already introduced as a part of the work of Qinqin Long which was mentioned above in section 1.3.2. Yet that integration was specified relying upon a certain workflow and the XEG format. Moreover, the use case needed its own specific GroIMP plugin.

Nevertheless, the generalized steps performed by the API would be very similar: First *defining the needed RGG commands* and *adding the plant model*, then *executing the RGG command* and *exporting the changed plant model*. Besides the model as a result it would also be important to be able to *receive the RGG output* and the *project graph*. A similar approach could be to use *XL queries to manipulate the model* instead of RGG commands.

Orchestration of multiple simulations from an external script

There are several scenarios where the automated execution of several models or multiple versions of the same model is useful, for instance, the fitting of a large number of model parameters using a sensitivity analysis. This orchestration of simulations can for now only be done by running the models on multiple instances of the headless mode. This approach is fully missing the ability to manipulate the simulation during the execution as well as a proper way to insert and receive data.

With the usage of the API a client should be able to *open multiple models*, *address them individually* and *execute them in parallel*. To make this a useful feature it also must be possible to *configure the models/edit config files* and *receive collected data*.

1.4.2 Project requirements & needed functionality

The different possible fields of application already highlight some requirements of the project, yet it is important to mention that due to the idea of a generalized API , these requirements are only guiding remarks for the first set of needed functionalities. The design must at any point be open for adding new functionalities following the GroIMP design of plugins and registry items. In the following, the requirements will be categorized into commands and needed API functionality similar to the schema in figure 1.1.

Categorised commands

The actions (written in italics) needed for the different fields of application can be split into four categories: project, model and file-handling as well as processing RGG and XL functions. The commands representing these actions can be seen in table 1.1. Yet as mentioned before this will not be an exclusive list, this is only the collection of commands planned for the initial version.

Table 1.1: List of the API commands that would be required for the diffrenent fields of application.

Command	Description
Project Handling	
open	Open a project and returning a reference for further usage.
create	Create a project of the given type and return a reference for further usage.

Command	Description
close	Close the workbench.
save	Save the project and and ensure the access to the saved file for the client.
examples	List and load the examples provided by GroIMP.
Model Handling	
add object	Read files such as XEG, RSML or OBJ and add the result to the project graph, similar to the GUI method Objects/insert file.
remove node	Remove a given node from the project graph.
get node attributes	Get the list of a given node's attributes, similar to the attribute editor.
set node attributes	Update the list of a given node's attributes, similar to the attribute editor.
export 3D	Export the 3d model to the given format and ensure the access to it, similar to the Export function of the view panel.
export graph	Export the project graph in a machine-readable format.
reset	Reset the model to the initial condition, similar to the reset button of the RGG toolbar.
SourceFile Handling (Similar to the usage of the GUI file exporer and JEdit)	
list files	Get a list of all existing files of a project.
read file	Get the content of a file.
update file	Set the new content of a file, similar to saving it.
add file	Add a new file to a project.
rename file	Change the name of a file.
remove file	Remove a file from a project.
RGG /XL management	
list commands	List all (RGG commands, like the buttons in the RGG toolbar.
execute command	Run an RGG command and receive printed results, similar to pressing a button in the toolbar and reading the result in the XL console.

Command	Description
execute query	Run an XL query and receive the results, similar to handling to the XL console.
compile	(Re-)compile the model, in the GUI this is connected to saving in JEdit.
get data	Read data generated during the execution, similar to the dataset explorer and dataset viewer.
General	
get logs	Get all eventual issues that are thrown by GroIMP as they are shown in the message panel.

Requirements on the API server

The main requirements on the API server besides the commands can be separated into the way the server is running and the way the server is handling projects.

The first part starts with the simple requirement that the server can run headless, meaning it can run without the GroIMP GUI, to improve its usage as a background process and enable the execution on systems without graphical environments. Additionally, the possibility of executing all commands without a common file system would be very advantageous. Thus, it would be possible to transfer all requests from one operating system (on which the client runs) to another operating system (on which the server runs). This would enable the API to run in a virtual container such as docker as well as on a remote server.

For the second part, the API must be able to handle multiple projects at the same time and forward the commands to the right projects in a nonblocking way to enable access to all running simulations at the same time. Additionally, the server must be able to handle commands that will be added later by other plugins. Finally, it should be considered to create the ability to open one project with the GUI and the API at the same time.

Chapter 2

GroIMP as API server

In the first part of the following chapter, the design and implementation of the API server (the GroLink plugin) and the needed foundation for this application are described. Afterwards, the basic usage will be shown and a small introduction to the development of extensions to GroLink will be given.

2.1 GroIMP Design

Based on the existing design of GroIMP and the requirements shown in the outlining of the project 1.4, it seems best that GroLink becomes a new GroIMP application. Therefore a new generalized foundation for GroIMP applications was necessary. This foundation mainly describes how GroIMP applications should handle their workbenches and how workbenches interact with the loaded projects. The current way has grown over the years and is focusing on the usage of the GUI and can hardly be separated from that. Therefore a new generalized way was created, to be useable for different applications including GroLink and the CLI. If the new foundation works as intended it could also be used as the basis for the currently implemented application (GUI).

The concept of this new foundation is described in figure 2.1 and the different components will be explained below.

2.1.1 Application design

As mentioned above GroLink is supposed to be a new GroIMP application. Besides a clearer code structure, this should also allow shipping a lighter version of GroIMP only containing the plugins required for GroLink Moreover, this would simplify the usage of it on computer systems without a graphical operation system.

With the new foundation, the base of an application like GroLink would be the abstract class UIApplication (Fig. 2.1). Each application based on the new foundation requires a workbench

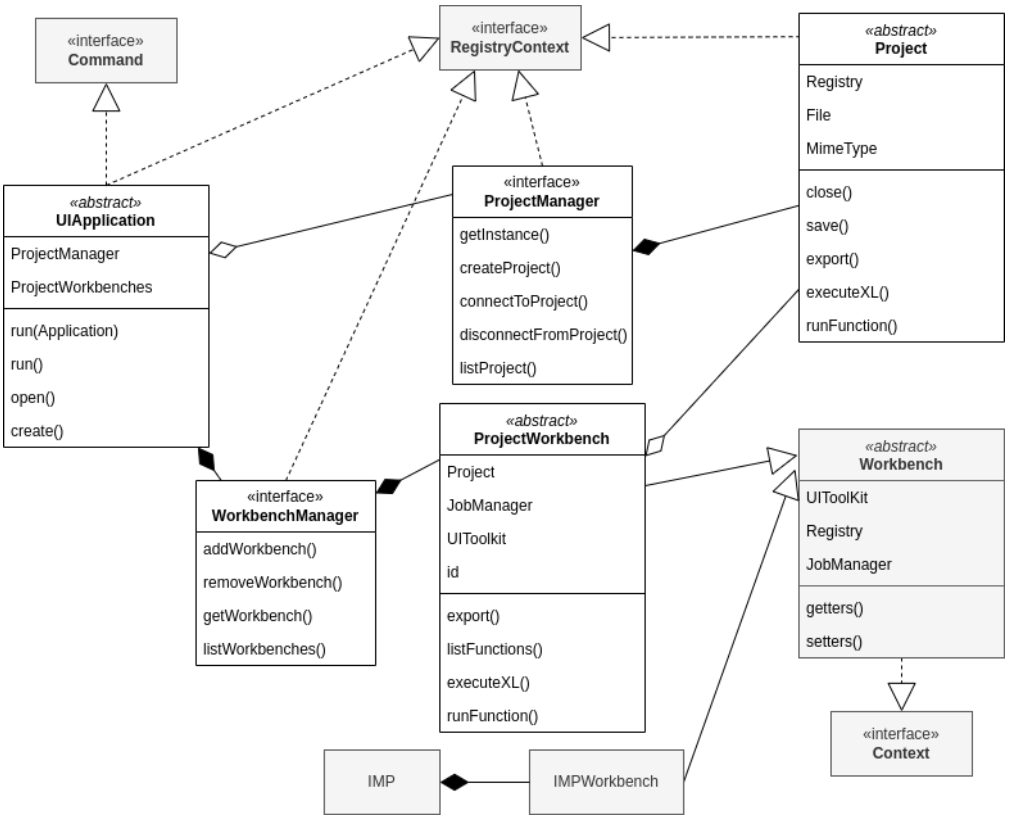


Figure 2.1: A schema of the generalized foundation for GroIMP applications such as GroLink and the CLI. The gray parts are already existing GroIMP components that will keep the same functionality. The new structure can be separated into project, workbench and application even though they are interconnected. The project structure consists of the ProjectManager which holds and creates all projects. The Project itself handles the content of the opened model and the direct interaction with it. The ProjectWorkbench then is a wrapper around the project for the interaction with the UIApplication. The WorkbenchManager of the UIApplication manages its ProjectWorkbenches. The UIApplication itself is the starting point of the execution and of the interaction with GroIMP.

manager and a suitable toolkit as well as a connection to the central project manager.

In all UIApplications, a main workbench should be created, added to the workbench manager and its JobManager initialized on a new thread. The main workbench will be used to create other workbenches and projects, by executing Java functions, which are defined in the application class, as commands. These functions then interact with the Workbench Manager.

Afterward, the thread of a UIApplication needs to "stay alive" because if this thread ends GroIMP is closed. Therefore it would be reasonable to use this thread for an action that is required during the whole execution. In the case of the APIApplication, this is the API (HTTP)server that listens to incoming requests.

2.1.2 Workbenches and Projects

The second important part of the new concept is that each application needs its own kind of workbench, for example, an API workbench. This is necessary because the different applications use different toolkits to return data and need to process different types of user input. Yet it results in difficulty regarding the aim to access the same simulation from two applications at the same time since the different applications expect different workbenches. Additionally, some of the information stored in a Workbench, like the currently opened panels, is only interesting for specific applications. Therefore this design includes the separation of the workbench and the real simulation (the project), resulting in two classes, one to handle the interaction and one to hold the simulation (Fig. 2.1). This way it will be possible to connect two different workbenches to the same project.

To do so the abstract class ProjectWorkbench is introduced as the base implementation for workbenches of applications that are based on the new foundation. The ProjectWorkbench will combine the project with a JobManager for handling commands and a UIToolkit for user interaction. The project itself is a class that holds the project graph and the registry and manages the interaction with them, which was previously done by the workbench as well. With references to the project class, the ProjectWorkbench can implement all functionalities originally implemented in a GroIMP workbench (Fig. 2.1).

2.1.3 Workbench management

Since an application can open multiple workbenches it needs a way to handle them. This is quite simple with the GUI since they are opened in new windows and the window manager of the operation system distinguishes them and organizes access to all of them. This is not possible for the CLI or the API, for them, it is necessary to have clear identifiers for each workbench and a way to get these identifiers.

Therefore the WorkbenchManager (Fig 2.1) is introduced as an interface to be implemented by new applications. This interface holds a collection of the workbenches opened by the application, a

way to access them and functions to add or remove workbenches from this collection.

2.1.4 Project management

In contrast to the application depending WorkbenchManagers, only one ProjectManager is supposed to be active and all running applications should register their projects there. To guarantee the independence of the ProjectManager it will be initialized during the boot process and not by any application. Besides that the ProjectManager is quite similar to the WorkbenchManager, it provides a list of projects and manages the access, the creation and the removal of them.

2.1.5 Command management

Even though most commands are general and can be used in all workbenches with the right abstraction layers and toolkits, there may be a need for some commands that are specific to a particular application. To solve this, hooks can be used to copy these specific commands to the command directory in the registry. These hooks should be executed on initialisation of the specific workbenches to manipulate their newly created workbench.

2.2 API Design

Regarding the design of the API itself, the outline provides some guiding points: The API should be able to handle commands that were added by other plugins. The commands have to be executed non-blocking so that several commands can be processed at the same time. Moreover, the application should be able to run without the graphical user interface and without a shared file system.

2.2.1 Call lifecycle

To get a first understanding of how the API should work, the life cycle of an API call (Fig. 2.2) has been described. This lifecycle describes the API-“cloud” in figure 1.1 in more detail. The client shown in the lifecycle can be replaced by the right part of the other figure and GroIMP as described there is located at the bottom of the lifecycle.

Going through the cycle should clarify what requirements the different components of the API must fulfill.

The life cycle begins and ends with the client, and this is also the point that can be defined most easily on the basis of the requirements. The client sends requests (Fig. 2.2 :1) to GroIMP and receives the results (Fig. 2.2 :5). With the request, the client demands the execution of a command on either a specific workbench or, for the management of workbenches, on the main workbench. The client needs an address to send the requests to, therefore all requests enter GroIMP at the “Server”. Yet if the server handles the requests itself, it is blocked for the duration of the execution.

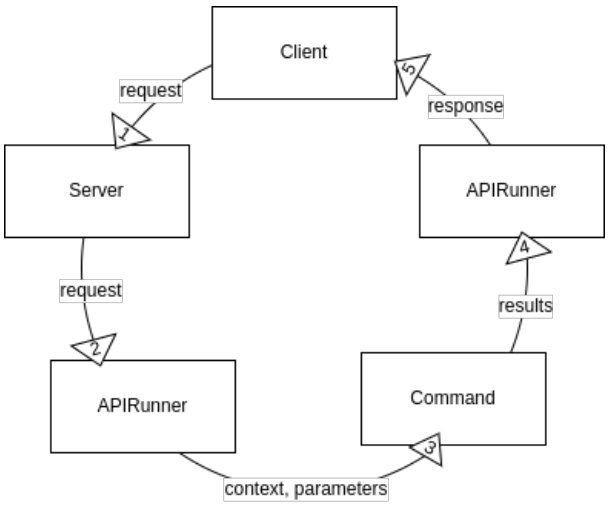


Figure 2.2: The simplified lifecycle of an API call starts when the client sends an HTTP request to the server. The server forwards the request to an APIRunner created for this purpose. The APIRunner executes the given command and collects the results. Finally, the APIRunner responds to the client without any further interaction with the HTTP server.

To avoid that the server creates a new object, a new APIRunner, and forwards the request to the runner (Fig. 2.2 :2). The runner is an implementation of Command and can therefore be pushed to the JobManager of the workbench of the request. The JobManagers then can queue as many runners as necessary, and execute them one by one. When the APIRunner is executed, the command the client requested will be resolved and then started (Fig. 2.2 :3). The command then can get the needed parameters from the APIRunner and add data to it (Fig. 2.2 :4). Finally, the APIRunner packs all the information and sends it back to the client (Fig. 2.2 :5).

2.2.2 APIRunner

As described above, the APIRunner is a central part of the lifecycle. It must frame the requested command in an environment that holds all the information needed to execute this command and return the result to the client. Moreover, this must be done independently from the server and the returning of the results must take place after the execution of the command. Therefore the best solution seems to be that the APIRunner class itself is a command that is executed on the JobManager and while being executed runs the "real" command as a subfunction. In this way, the runner can add itself as the info variable to the command, which enables the command to access the parameters and write results. Afterward, the runner is still in action and can return the results added by the command to the client. The workflow can be seen in figure 2.3. This design is inspired by the HTTPResponse class in de.grogra.imp.net.

In summary, the APIRunner must hold all the parameters of the API call and the command, and must provide a dataset that the command can use to add return values.

2.2.3 Server

The API server can simply be based on the existing server of GroIMP, because it does not need any specific functionalities, it only initializes a new APIRunner with the request and the client address as parameters. Additionally, this project can benefit from the fact that the existing server already handles all requests in new threads. Therefore even the parsing of the requests and the forwarding to the right JobManager is nonblocking.

As mentioned above the API server will be the thread that holds the APIApplication alive since it defines the only way to access the API and interact with GroIMP.

2.2.4 Calls

A call on an HTTP API is basically an HTTP request and follows the same structure on the URL as well as on the request body. Due to the aim of a stateless API, all information needed during the API lifecycle must be included in the request.

Besides the technical information regarding the location of the server (the host and the port), this information must include a reference to the workbench that is used and to the command that should be executed. The workbench reference can simply be based on the identification of the workbenches in the workbench manager extended by a static reference called app which leads to the main workbench and is used for managing the other workbenches. For the command the simplest way is to use the absolute path to the command item in the registry, in this way, GroIMP can just resolve this information and receive the command. Moreover, this enables the client to reach any CommandItem registered in GroIMP.

Since most commands need some kind of input information such as the name of a RGG function or the path to a file, this information must be included in the call as well. Additionally, it must be completely clear which input parameter represents which information. This information can be added as the query part of an HTTP request, consisting of key-value pairs.

Structure

The structure chosen in this project is inspired by the common structure of REST API calls, starting with the address of the server, then the object of interest (the workbench), then the action that should be executed (the command) and then additional parameters.

That leads to the following URL structure:

```
<host>:<port>/api/<ref>/<command>?<key>=<value>
```

- **host**: The IP of the server (or localhost)
- **port**: Where the API is listening
- **ref**: The reference to the workbench (‘app’ or ‘wb/<id>’)

- **command**: Registry path to the command item
- **key**: Name of the parameter
- **value**: Of the parameter

Request method

In contrast to a REST API , this project does not have to distinguish calls based on the request method, since each command is only representing one action. Therefore it should not make a difference how the command was called.

Only the fact that some request methods (e.g. PUT, POST) contain bodies with more data is of interest for this project because it gives the ability to send whole files to the API , which will enable the transfer of projects (as GSZ files) from the client to the server without the need for a shared file system. Nevertheless, the command triggered will be the same and the different abilities to access data must be defined in the command.

2.3 Preparatory work

Before starting with the implementation of the API the creation of the generalized foundation was necessary, including the project- and workbench management and the interfaces for UIApplications, projects and workbenches. This new foundation was developed in parallel with the CLI (Command Line Interface) application which enabled testing and improving on the structure.

The code of the foundation is divided into additions to the platform and the newly created ProjectManager plugin. This division is mostly between the definition in the platform and the implementation in the ProjectManager plugin.

2.3.1 Additions to platform

The additions to the platform plugin are mostly descriptive and therefore only consist of interfaces and abstract classes. The idea is to define a structure for GroIMP applications that holds all needed functionalities in a generalized way. To do so the UIApplication class is connected to the ProjectManager and the WorkbenchManager (Fig. 2.4).

Project

Following the separation of project and workbench, described in the design, the abstract class Project is implementing the interaction with the ProjectGraph and source objects, similar to the original workbench. This includes adding and removing nodes and managing source files and their content. This project has a registry and therefore implements the RegistryContext interface. Moreover, the project class handles the connection to the source of the loaded project and also the saving of it. Additionally, there are abstract functions that need to be implemented for any class extending Project, these functions cover the handling of (RGG)-functions, local file synchronization and the initializing and disposing of a project.

ProjectContext

In the style of the Context, RegistryContext and ThreadContext interfaces, the ProjectContext interface describes that a class contains a reference to a project. To implement the interface a class must contain the functions getProject() and getProjectRegistry(). This interface is implemented by the ProjectWorkbench and therefore by all workbenches extending the ProjectWorkbench (Fig. 2.4).

ProjectManager

This interface is the base definition of a ProjectManager as the one implemented in the Project-Manager plugin. To be able to store the added project the project manager requires a registry and therefore this interface extends the RegistryContext interface.

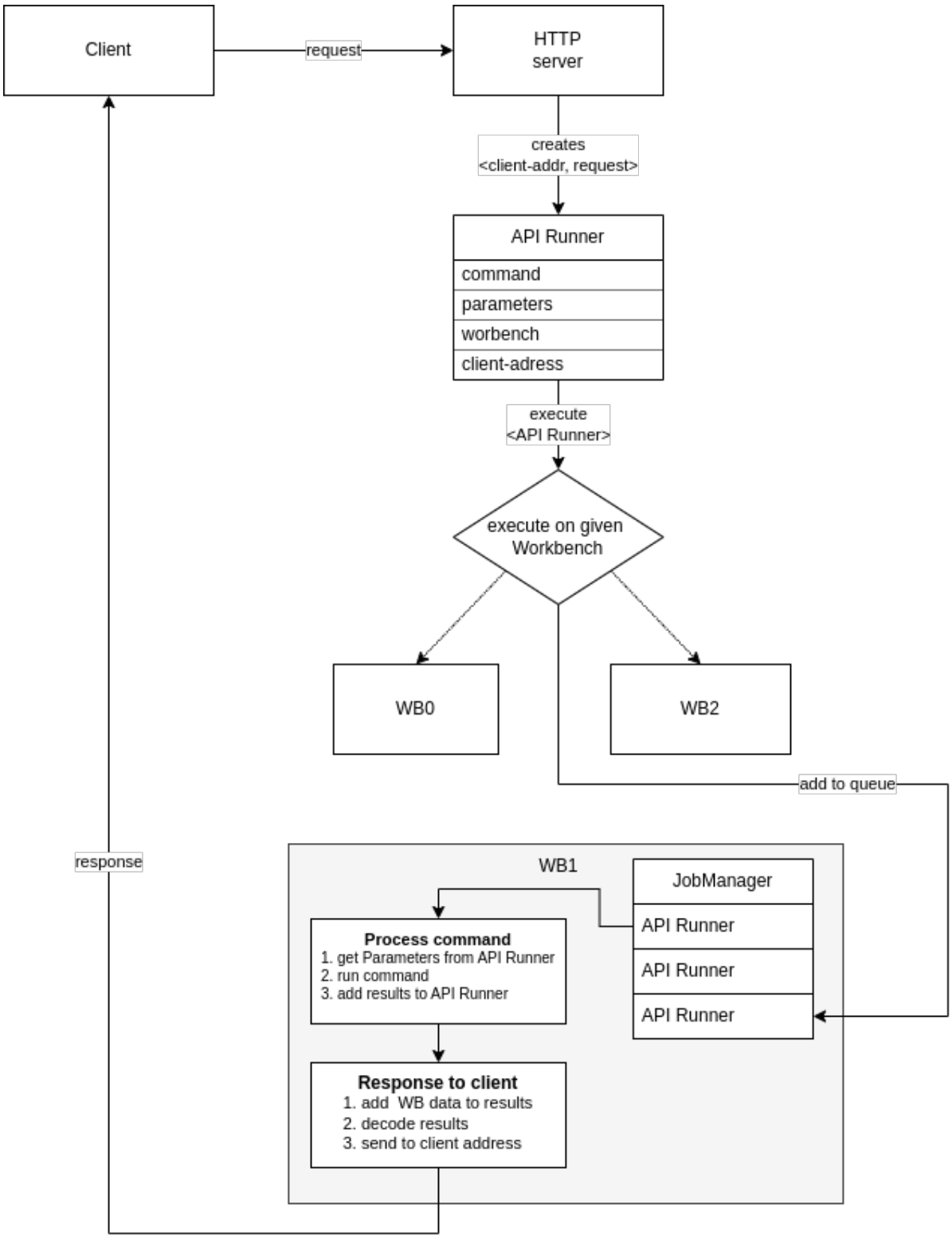


Figure 2.3: This schema of the APIRunner follows similar steps as the API lifecycle. It starts with the server that creates the runner based on the client address and the request. The request must contain all information needed to provide the information listed in the body of the APIRunner class (command, parameter, workbench). Using the workbench information the runner will be added to the queue of the JobManager manager of a specific workbench. The JobManager then executes the runners one by one in the same way as any other command. If a runner is executed it will start the command which then accesses the information of the runner to complete its task and add the results back to the runner. Finally, in the last step of the APIRunner command, the result is returned to the client.

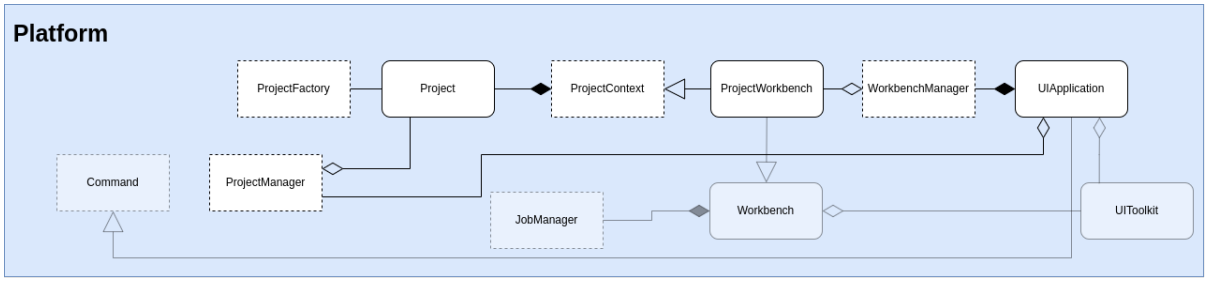


Figure 2.4: The relations of the parts of the new generalized foundation for GroIMP applications that are implemented in the platform with some closely related already existing elements for context (grayed out). Rounded boxes represent abstract classes and dotted boxes represent interfaces. This shows how the UIApplication is connected to the Project and the ProjectWorkbench through the managers.

The interface defines functions for opening and creating projects, managing them and connecting and deconnecting Workbenches to them. To clarify what implementation of the abstract class Project should be created the ProjectFactory is added as an additional parameter.

The ProjectManager creates the connection between the UIApplication and the Project (Fig. 2.4).

ProjectWorkbench

This abstract class represents the other half of the original workbench while referencing the first half by implementing ProjectContext (Fig. 2.4). In this way, the ProjectWorkbench holds the same functionality as the original workbench but forwards many functions to the project linked to it. The ProjectWorkbench extends the original workbench which enables it on the one hand to use the same functions designed in GroIMP such as `Context.getWorkbench()` and on the other hand can use some implementations from the original workbench.

Several GroIMP commands are defined in the ProjectWorkbench on a clear structure as shown in the code below (Lst. 2.1) for the execute command. First, the static function is called by the CommandItem, this function then resolves the current workbench and calls the abstract function with the info variable. That function must be defined in all Workbenches that extend the ProjectWorkbench and must translate the content of info into the needed parameters so it can call the non-static function of the ProjectWorkbench which then can perform the requested task.

WorkbenchManager

As defined in the design, each application will need some kind of workbench management. To clarify this management the WorkbenchManager interface is used. This interface holds the functions to create, list and close workbenches.

```
1 public static void execute(Item item, Object info, Context ctx) throws Exception {
2         ((ProjectWorkbench)ctx.getWorkbench()).execute(info);
3     }
4
5     public abstract void execute(Object info) throws Exception;
6
7     public void execute(String commandName) {
8         project.execute(commandName, this);
9     }
```

Listing 2.1: The static function in line 1 is linked to a CommandItem in the registry. This function runs the abstract execute(Object info) function on the ProjectWorkbench provided by the context. If this provided Workbench would be an APIWorkbench its execute(Object info) function would be called and extract the command name out of the AIPRunner instance provided by info. This function can then execute the "real" function in line 7. The powerful part of this structure is that for a CLIWorkbench only the implementation of the abstract function has to be different.

UIApplication

All GroIMP applications that follow the new structure should extend the abstract class UIApplication. It handles the connection to the ProjectManager, the WorkbenchManager, the UIToolkit and the MainWorkbench which is used to handle application-based requests such as opening a new project (Fig. 2.4). These application-based requests are defined as command items and follow the same structure as the commands in ProjectWorkbench (Lst. 2.1). Additionally, UIApplication extends plugin which enables the initialization of the application during the boot process and gaining access to the registry. Finally, UIApplication defines the static function `run (Application app)` which must be defined to add a Java class as a GroIMP application, and it extends Command, therefore each application can also be started during the execution of GroIMP.

ProjectFactory

This interface is used as a placeholder so the ProjectManager can work with all extensions of the Project class without specific knowledge. This works by creating a ProjectFactory of the type of project needed and passing the factory as a parameter to the ProjectManager, which then executes the factory function to create a project of the needed type. In the current implementation, the naming of the interface is misleading since it does not work like other factorys in GroIMP.

FilterSourceFactory

Through the long focus on the graphical user interface, some functionalities of GroIMP are implemented in a way that only works with graphical representation. One of these is the display of the examples as an HTML page. To fix this the FilterSourceFactory was implemented, which can add existing GroIMP projects to the registry in a way that resolves to FilterSources which then can be

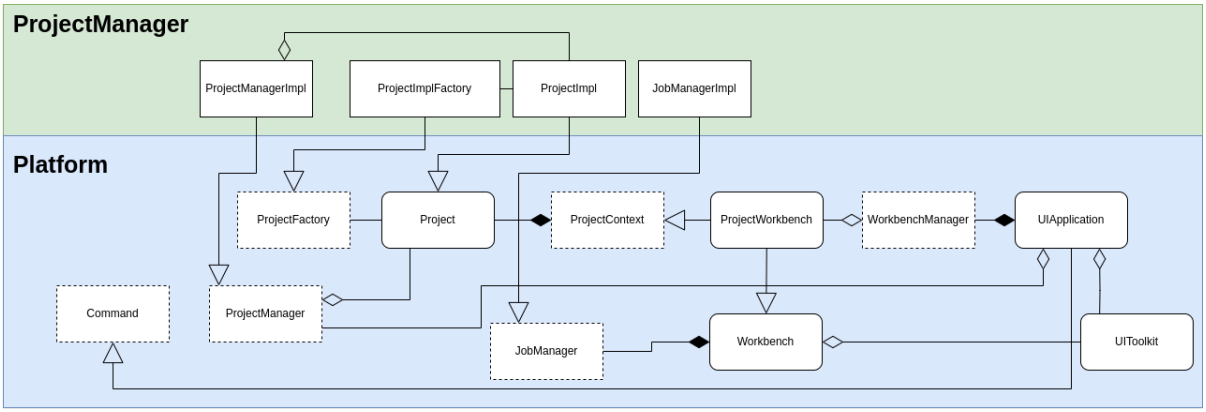


Figure 2.5: A UML diagram showing the Java classes implemented in the ProjectManager plugin in connection to the interfaces and subclasses they implement/extend.

opened in Workbenches. This factory is not only used for the examples but also for the templates for new projects. An additional benefit of this structure is that new examples and templates can be added in additional plugins. This will be explained below in the developing section of this chapter.

2.3.2 Project management

This plugin holds the generalizable parts of the implementations of the interfaces defined in the platform as shown in figure 2.5. Due to the separation of these implementations from the platform, it would be possible to replace all of the following classes in the future to either improve them or add functionality for a specific application. Additionally, the plugin is used to link the ProjectManagerImpl as a Plugin-class to the registry.

ProjectImpl

This implementation of the abstract class Project implements the execution of RGG functions and compiling through the visiting of source files. This implementation is used in GroLink and the CLI.

ProjectManagerImpl

This is the currently used implementation of the Project Manager. Due to the design choice that the ProjectManager should be independent of the application and be initialized directly by the boot process this class extends the abstract class Plugin and is referenced in the plugin.xml file of the plugin. The class stores the reference to the projects and the linked workbenches as registry items in the directory “ProjectManager/projects” with a directory for each project.

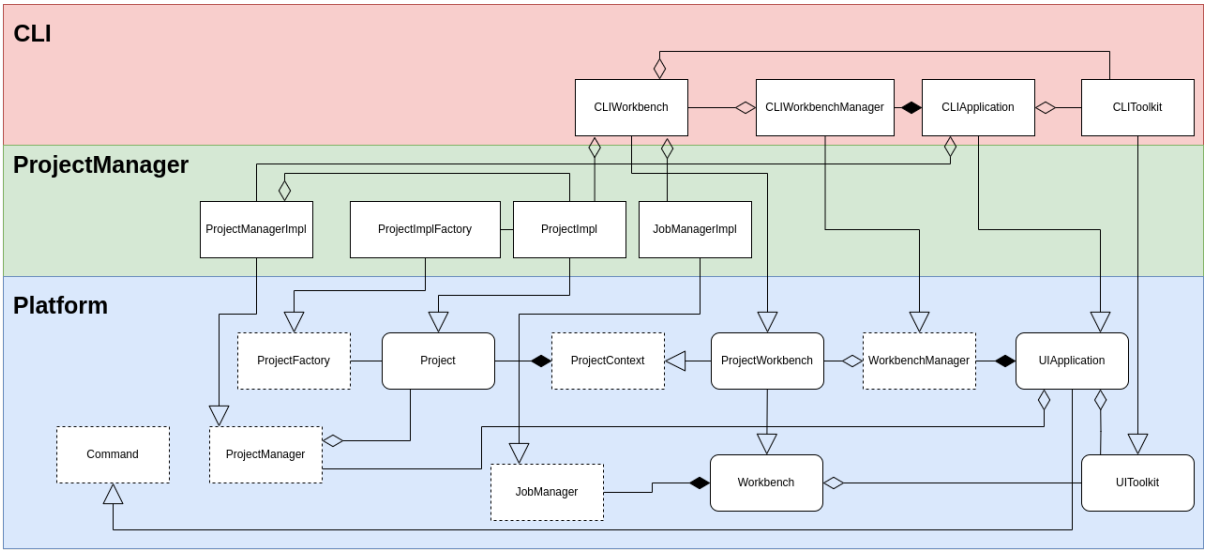


Figure 2.6: A UML diagram showing the basic classes of the GroIMP CLI and their connection to the platform and the ProjectManager. For simplification, this diagram is missing large parts of the CLI plugin structure including the implementation of the toolkit-elements and the autocompletion.

JobManagerImpl

This implementation of the JobManager interface is a slightly modified copy of the IMPJobManager. This JobManager is used by the CLI and the API.

ProjectFactoryImpl

A ProjectFactory for the ProjectImpl class.

2.3.3 CLI

The command line interface of GroIMP [21] is a newly created application that uses a text-based input system to navigate between different workbenches and edit and execute simulations.

The CLI application was in large parts developed by Gaëtan Heidsieck and therefore the description will be very brief as it is not really part of this project. In the following, only the structure that is either directly based on the subsystem described above or used later in the API will be described. The use of the CLI will also only be explained in its basic features.

Structure

The core of the CLI is the CLIApplication which extends UIApplication (Fig. 2.6) and is mainly used to receive the input from the command line using the JLine3 [22] library. This input contains beside the command line an implementation of the Nano text editor [23] which can be used to

edit RGG files. To properly interpret the input from the CLI to the project the CLIWorkbench was created which reads the parameters added behind the commands and parses them to execute the commands. This works as described above in the section about ProjectWorkbenches. The CLI application comes with a CLIWorkbenchManager, which is a hashmap-based implementation of the WorkbenchManager interface (Fig. 2.6) that creates CLIWorkbenches.

A last part important to mention here is the CLIToolkit, which describes how GroIMP is supposed to create the panels for the CLI. The CLIToolkit extends the UIToolkit and is needed to initialise the application and from there is added to each CLIWorkbench (Fig. 2.6).

While this approach works well for some panels, a subset of panels that can be used in the GUI are not implemented based on the toolkit approach and can therefore not be viewed in the CLI.

Usage

The CLI can be started in GroIMP using the application parameter '-a':

```
java -Xverify:none -jar core.jar -a cli
```

Listing 2.2: The system command to start the GroIMP CLI application. In some cases core.jar can be named differently

This will start the CLI and enable the user to execute application commands. Commands on the CLI are separated into three groups, application commands, window commands and workbench commands. Application commands are used for the management of the workbenches, including opening, listing or selecting workbenches. These commands always start with a $ sign. The window commands that start with % are used to interact with the panels generated by the toolkit, including listing the files and scene graph, or opening the XLConsole. The Workbench commands are used to interact with the Workbench by executing RGG functions or editing source files. Workbench commands do not have a defined start character.

An example use of the CLI can be seen in listing 2.3, which creates a new RGG Workbench, lists the source files, executes the run functions and displays the simulation graph.

```
1  [base]>>$create
2  [newRGG]>>ls
3
4  [0] : Files
5    [1] : Model.rgg
6    [2] : param
7      [3] : parameters.rgg
8
9  [newRGG]>>run
10 [newRGG]>>%graph/hierarchicobjectinspector
11 [0] : Node [0]
12   [1] : RGGRoot [21]
13     [2] : F [24]
14       [3] : RU [25]
15         [4] : RH [26]
16           [5] : A [27]
17       [6] : RU [28]
18         [7] : RH [29]
19           [8] : A [30]
20 [newRGG]>>
```

Listing 2.3: An example usage of the CLI application that starts with the creation of a new workbench based on a template in line 1. This new workbench is automatically selected as shown by the change of the prefix from line 1 "base" to line 2 "newRGG". The ls command in line 2 is a shortcut that starts the CLIToolkit representation of the file explorer. The original way to open a panel, starting with the percent sign, is shown in line 10 to open the project graph. The run command in line 9 is calling the RGG function run which includes the rewriting rule from the default new RGG model. The results can be seen in the graph below line 11.

2.4 Implementation

The implementation of the GroLink plugin is based on the new generalized foundation and partly extends classes from the CLI (Fig. 2.7). In the following the structure of the new plugin and the function of the different classes will be described.

2.4.1 Plugin structure

The structure of the GroLink plugin benefits greatly from the preparatory work and the CLI application. It follows the newly created structure for GroIMP applications consisting of extensions of UIApplication, CLIWorkbenchManager, CLIToolkit and ProjectWorkbench (Fig. 2.7). Moreover, the ProjectManagerImpl is used which follows the design concept for project management.

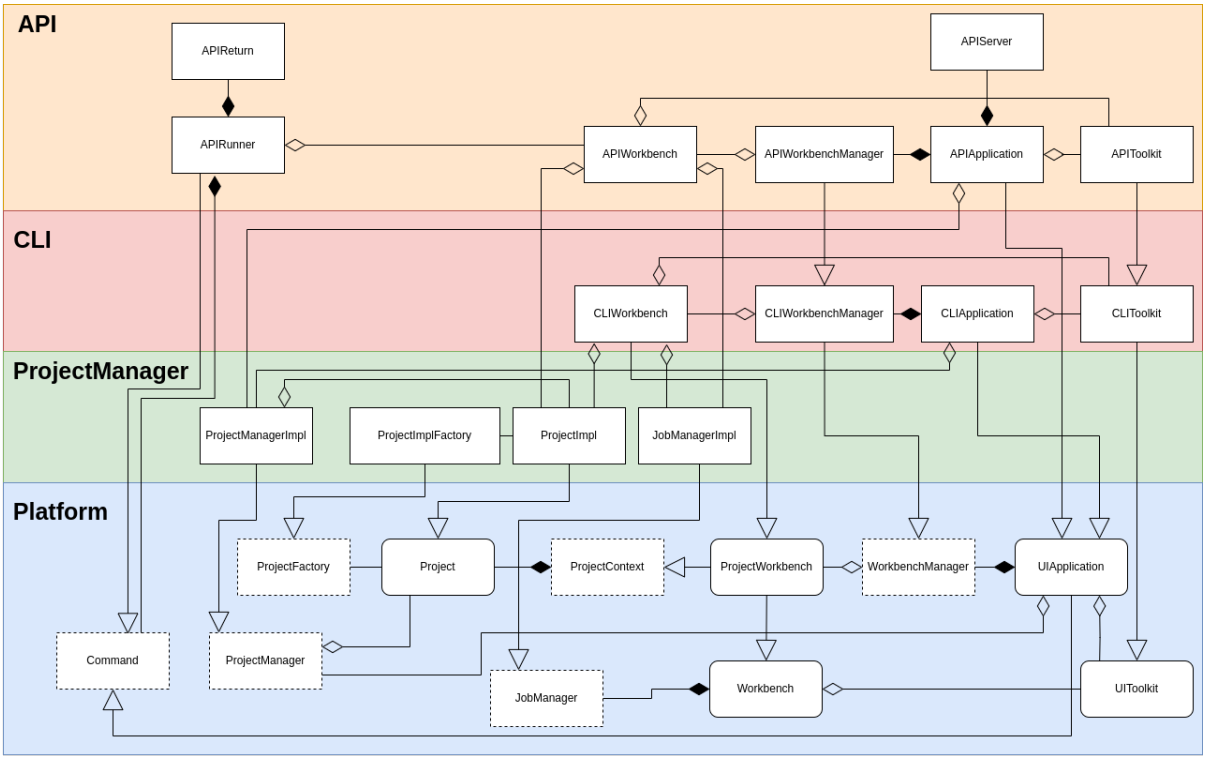


Figure 2.7: A UML class diagram showing the connections of the API/GroLink to the different layers that were described in the section about preparatory work.

2.4.2 APIApplication

As defined in the abstract UIApplication the APIApplication can be started with two functions, `run(Application app)` and `run(Object arg, Context context)` which enables it to be started as a GroIMP application and as a GroIMP command. These two static functions first define the port based on the GroIMP preferences or the command line parameter, if available. Then, it receives the instance of the running ProjectManagerImpl and creates a new APIWorkbenchManager and APIToolkit. Based on these 4 objects the instance of the application created during the boot process is properly initialized. This initialization sets the class variables and creates an instance of APIServer based on the port.

Finally, both run functions to execute the start of the now-initialized instance of the APIApplication, which creates the MainWorbench and starts the APIServer.

Besides this start process the APIApplication holds the implementation of the app commands following the structure of executing commands explained above in the section of the UIApplication. The detailed way in which the API queries parameters and adds results will be explained below in the APIRunner and the APIReturn section.

2.4.3 APIServer

The APIServer extends the abstract class Server implemented in Utils(de.grogra.http). The APIServer is initialized with a reference to the APIApplication which is later needed to initialize the APIRunner instances. The abstract class server is designed in a way that each request is pushed to a new thread and on that thread the abstract function handleRequest is executed. In the case of the APIServer this abstract function checks if the request starts with "/api" and if so creates a new APIRunner based on the request, the socket of the client who did it and the reference to the API application. This APIRunner is then handling the rest of this request including the response.

2.4.4 APIRunner

The APIRunner is as already shown in figure 2.2 a key part of the design and the concept of GroLink It carries the API call from the request through the execution to the response (Fig. 2.3). The class itself implements the interface command so it can be executed on the JobManager of the requested workbench (Fig. 2.7).

The general workflow of the APIRunner starts with its initialization by the API server. The first step is to parse the HTTP query to write the parameters into a StringMap and the HTTP path to resolve the workbench and the command.

In the second part, the APIRunner itself is executed as a command on the JobManager of the before-resolved workbench. As a command, the first thing the APIRunner does is to execute the "real" command, which was resolved in the first step, by just using the run function of the command

interface with itself as the second parameter (info) and the resolved workbench as the context. This gives the executed command the ability to access the APIRunner during its execution, including the getParameter function, the content (body) of the request and the addResponseContent functions. The getParameter function tries to read data from the StringMap that was generated out of the HTTP query, therefore the String used in getParameter must be identical to the key of the key-value set in the APICall. Additionally, it is possible to define if a parameter is allowed to be null, if not and the parameter does not exist it throws an APIParameterNotDefinedException exception. The content of the request is just the binary stream of data added by a POST or PUT command, it can be decoded to the different types of files.

With the addResponseContent functions, the command can add a key-value pair to the APIReturn instance of the APIRunner and therefore send data back to the client. If the response data is supposed to be a binary file this is added as a response content too, if so it is the only content returned.

Finally, after the run function of the command is completed the APIRunner, still executed by the JobManager, starts the collectAndSend function that adds the content of the XLConsole and the logger to the APIReturn data and sends it all as a response to the client.

Additionally, there is the ability to forward the collectAndSend function to other commands and let the APIRunner-execution end without responding to the client. This is needed for all commands that themself add commands to the JobManager, such as compiling. Because the JobManager executes commands in a queue, the results of a command added after the APIRunner can never be included in a response sent by the APIRunner. In that case, the newly added command is instructed to collect the data and send it. To do so it gets the APIRunner as a parameter.

The workflow of an APIRunner can fail at different points, which returns different exceptions. On the one hand, there are direct API problems such as a wrong command path (APICommand-NotFoundException) or a missing parameter (APIParameterNotDefinedException). On the other hand, there can be errors related to GroIMP and the execution of the command. In all cases, the API returns the exceptions to the client with the appropriate HTTP status code.

2.4.5 APIReturn

This class is a simple wrapper around a JSON array, that is used in the APIRunner to create the response message during the execution.

2.4.6 APIToolkit

At the current state of the development, the APIToolkit is a full extension of the CLIToolkit except for one function that is used to log application-wide errors.

2.4.7 APIWorkbench

The APIWorkbench is extending the ProjectWorkbench (Fig. 2.7) and implementing its abstract functions to work with the APIRunner. Moreover, the APIWorkbench includes an APILogHandler that is used to collect LogRecords as a JSON array and makes them accessible for the APIRunner which then adds them to the response as the log.

Besides the functions defined in the ProjectWorkbench the APIWorkbench holds additional functionalities that might in later improvements be moved to the APIToolkit. This includes the handling of SourceFiles and datasets and the ProjectGraph as well as the execution of XLQueries. For the last one, the XLinLineRunner class is used.

The structure of the JSON-based representation of the project graph shown in figure 2.8 can be seen in listing 2.4.

```
{
    "projectgraphNodes":[
        {"id":0,"type":"de.grogra.graph.impl.Node"},
        {"id":21,"type":"de.grogra.rgg.RGGRoot"},
        {"id":23,"type":"Model.A"},
        {"id":24,"type":"Model.A"}
    ],
    "projectgraphEdges":[
        [0,21,512],
        [21,23,512],
        [21,24,256]

    ],
    "projectgraphRoot":"0"
}
```

Listing 2.4: The JSON representation of a project graph of a model with a RGGRoot and two custom nodes A. The projectgraphNodes list holds the nodes themselves with their node id. This ID is again used in triples of the projectgraphEdges list to define the source (first entry) and the destination (second entry). The third value in the triple represents the edge bits to seperate for example successor and branch. Finally the projectgraphRoot represents the root node of the project graph.

2.4.8 APIWorkbenchManager

The APIWorkbenchManager extends the CLIWorkbenchManager due to the very similar requirements, the only change is that the APIWorkbenchManager creates APIWorkbenches instead of CLIWorkbenches. The hashmap-based approach is suitable for the API since it creates a unique identifier for each workbench that does not change if another workbench is closed.

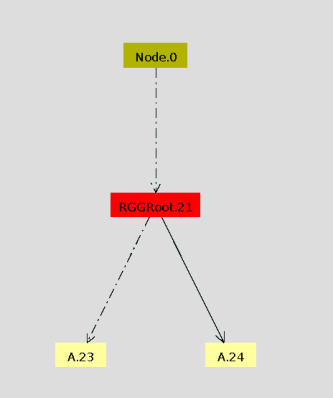


Figure 2.8: A simple example project graph.

2.4.9 XLinLineRunner

The XLinLineRunner is a class mainly copied from the ShellFilter, that allows the execution of XL queries without the start of a new thread as the ShellFilter does. This gives the API the ability to run an XL-query and read the results from the XLConsole.

2.5 Usage

The above-described implementation is available on Gitlab and is included in the release of GroIMP 2.1. Based on the chosen design the GroLink API server can be used without any specific client. As a stateless HTTP-API, it can be addressed by any software or programming language that can create HTTP requests, for instance, any web browser. In this section, the start of GroLink in a local and a containerized environment will be shown followed by an introduction to the commands and the additional data handling.

2.5.1 Start and connect

As mentioned in the introduction, GroIMP is designed for handling multiple applications. To define which application is supposed to be executed the command line parameter -a is used. The default application is the “normal” graphical user interface, this application is started when no parameter is defined. The currently available options are de.grogra.GroIMP (the “normal” GUI),CLI and API. Therefore the command in listing 2.5 is used to start the GroLink API on the port defined in the GroIMP preferences.

```
java -Xverify:none -jar core.jar -a api
```

Listing 2.5: The terminal command to start GroLink on the port defined in the preferences. The name of core.jar can be different based on way GroIMP was installed.

If the port is supposed to be different it can be defined by the additional parameter -Xport. For instance, the command to start GroLink on port 5051 is shown in listing 2.6.

```
java -Xverify:none -jar core.jar -a API -Xport=5051
```

Listing 2.6: The terminal command to start GroLink on port 5051.

Additionally, it is possible to add `--headless` to avoid the appearance of the splash screen.

In any case, the response (Lst. 2.7) should be the same, except for the port. Afterward, there is no more interaction with this terminal necessary and it will only print out eventual error logs.

```
Welcome,
the GroIMP HTTP API is running on port:58081.

 You can now connect to it via HTTP or by using a client libary
```

Listing 2.7: The start message of GroLink in the terminal. In this case the API is running on port 58081.

The API will then be accessible on the server it is running on via the specified port. If the API is running in an environment, the server address is localhost, otherwise, it is the IP address of the server. If the API is executed on a remote server it must be clear at any time that GroLink does not include any security mechanism that prevents people from executing harmful Java code embedded in an RGG function on the server.

The GroLink API can now be accessed by this address by entering API calls in a web browser, this can be seen in figure 2.9. Besides that it is also possible to execute these API calls from any other software project, this is shown in the use case “Using GroIMP as a backend for a forestry game”(sec. 4.3).

Starting in a containerized environment

One goal of the design was the usage of GroLink on remote systems or in virtual containers that holds all needed dependencies. The requirements for both are similar and therefore only the usage in the container system Docker is shown using the default GroIMP Docker container which can be created with the Dockerfile in listing 2.9. By starting the container with the API the port has to be forwarded to be reachable from outside the container (Lst. 2.8).

```
docker run -dp 127.0.0.1:58081:58081 groimp2.1:latest -a api
```

Listing 2.8: The command to start the container with port 58081 forwarded to the "real" system.

Afterward, the GroLink server can be used in the same way as a locally started version.

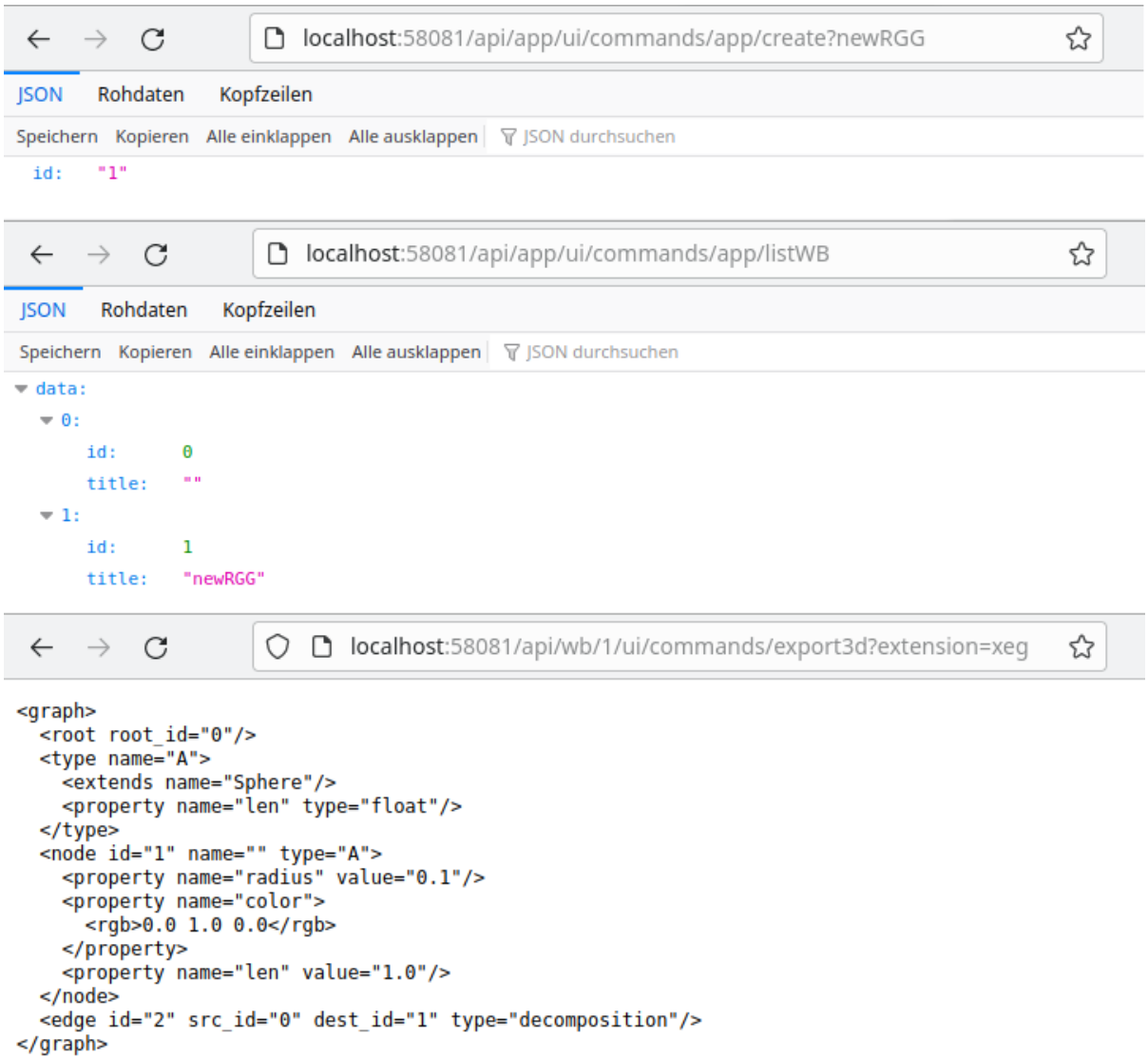


Figure 2.9: An example of using GroLink in a web browser (Firefox). It starts by creating a new project, then lists all the open workbenches and finally exports the newly created model as XEGgraph. It can be seen that the first two commands use “app” as a reference, while the last one refers to the ID of the created workbench.

```
FROM openjdk:11
COPY ./GroIMP-2.1-all.deb /usr/src/GroIMP-2.1-all.deb
RUN apt-get update && apt-get install -y pocl-opencl-icd /usr/src/GroIMP-2.1-all.deb
WORKDIR /usr/share/GroIMP/
ENTRYPOINT ["java", "-Xverify:none", "-Xss200m", "-jar", "core.jar"]
```

Listing 2.9: The Dockerfile used to create a contianer of GroIMP 2.1 based on the Debian installer package.

2.5.2 Commands

Besides the address of the server, the path to the commands and the required parameters are needed to execute the API Call. Currently, 24 commands are included in the GroLink plugin. A list of all commands and their usage can be found in appendix A as well as on the Gitlab repository of the API plugin `https://gitlab.com/grogra/groimp-plugins/api`.

In comparison with table 1.1 from the outline, these commands cover all functionalities except for the get and set of node attributes and the deletion of nodes. Nevertheless, through the usage of XL queries, even the manipulation of node attributes and nodes is possible.

In general, it is important to mention that post commands can only handle files that are added directly, not with flags or in groups. Besides the commands design for the API , it is possible to address any other command defined in the GroIMP registry as a CommandItem by its absolute path, but this functionality is very limited since it is not possible to give information to this command or receive results besides the console, the log or changes on the model.

2.5.3 Data handling

To keep independence from shared file systems it was necessary to provide a way to transfer external data (list/tables) in both directions. The shown approaches are additions to the existing ways, it is still possible to refer to external files and write the data there.

Read data from a project file

To send files such as CSV or XLS with the simulation to the API , they can simply be added to the project in the same way as RGG files and then read with the newly created library function `getInputStreamFromProject("fileName")`.

For example, if a file of coordinates called points.CSV is added to the project described in the listing 2.10, it can be accessed as an input stream and then handled in the same way as any other Java input stream. In this case, the file is read as buffered input and the coordinates are added as points.

```
1  import java.io.InputStream;
2  import java.io.BufferedReader;
3  import java.io.InputStreamReader;
4
5  protected void init()[
6      Axiom==>Floor;
7  ]
8
9  public void draw ()
10 {
11     //get the data from the file and create the stream and the reader
12     InputStream inp = getInputStreamFromProject("points.csv");
13     InputStreamReader inpr = new InputStreamReader(inp);
14     BufferedReader bf = new BufferedReader(inpr);
15     int i=0;
16     [
17         Floor ==> Node()
18             //loop over the lines of the file
19             for(String line; (line = bf.readLine()) != null; )(
20             {
21                 //parse the line and add the points
22                 String[] lp = line.split(",");
23                 float x = Float.parseFloat(lp[0]);
24                 float y = Float.parseFloat(lp[1]);
25                 float z = Float.parseFloat(lp[2]);
26                 i++;
27             }
28             [Point(x,y,z)]
29             //having less children per node is always a good idea
30             if(i>100)(
31                 Node()
32                 {i=0;}
33             )
34         );
35     ]
36     //close the stream and the readers
37     bf.close();
38     inpr.close();
39     inp.close();
40 }
```

Listing 2.10: An example using the new getInputStreamFromProject function, that reads coordinates from a CSV file and adds them as points to the model. The input stream created in line 12 is added to a reader and a buffered reader, therefore it is possible to read the document line by line without loading the whole document at once. The readers and the stream must be closed (lines 37-39) like any other readers and streams in Java.

This way enables the changes of the data with the same API commands that also edit RGG files.

Yet changes in the data can only reach the simulation after the model was compiled.

Writing data to Datasets

To return data from a simulation to the client the GroIMP datasets can be used. In GroIMP, they are used in the same way as for plotting, meaning that a DatasetRef is created and the new data is inserted by rows.

GroLink then provides a command to list all available datasets and another one to load a dataset in CSV format.

2.6 Developing

Since the design of GroLink follows the GroIMP plugin concept, it is possible to add new functionalities, templates and examples to the API without changing the GroLink plugin itself. In the following, it will be explained how to create such additions, first for functionalities and then for templates or examples.

2.6.1 How to add new API commands and functions

There are several reasons why a specific API command needed in GroIMP might not be added directly to the GroLink plugin, for example, if it is for a specific project, or if the command is associated with another plugin that is not necessarily installed. In the following a new plugin is created, the example commands are introduced and then implemented in Java and linked in the plugin.xml.

Creating a new Plugin

To keep the following steps simpler, it is considered that the new function would be in a new plugin called APIPlus. The creation of a GroIMP plugin is described on the GroIMP wiki in the developer guide Creating a GroIMP plugin.

Following this guide and adding a Java class called SayHi.java will lead to the structure below:

- pom.xml
- src
  - main
    - java/de/grogra/apiplus/SayHi.java
    - resources/plugin.properties -resources/plugin.xml
  - assembly
    - assembly files used for packaging

The commands

The idea of these new functions is very simple, the API server will greet the user with the given name. The plugin will contain two commands, one application command and one workbench command, which only differ on the fact that the workbench command adds the name of the workbench that was responding.

The application command would respond to the call:

`<host>:<port>/api/app/ui/commands/app/sayhi?name=<name>`

with the following JSON data:

```
{
    "data": [
       "Hi <name>"
    ]
}
```

The workbench command would respond to the call:

`<host>:<port>/api/wb<id>/ui/commands/sayhi?name=<name>`

with the following JSON data:

```
{
    "console": [],
    "data": [
        "Hi <name>, sended by <wb-name>
    ],
    "logs": []
}
```

The Java Code

As with any other function referenced by a CommandItem in GroIMP, the functions are static and have three parameters, the “item” referencing the CommandItem itself, “info”, a variable to transfer information and “context” which defines on which workbench the command is executed. In the case of GroLink, info is an instance of APIRunner, the class that holds the information about the API requests, handles the parameters and returns the result to the client.

The app function of the example shown in listing 2.11 does not need the context and therefore the code is only three lines. First, in line 4, info is cast to an APIRunner, then in line 8, the parameter is read from this runner. In this case the static variable `APIRunner.INP_NAME` is used but any String works as long as it is the same as the key in the HTTP query (`url?<key>=<value>&<key2>=<value2>`).

Additionally, the second value of getParameter defines if the parameter is allowed to be null. If it is not allowed and the parameter is not defined an error is returned to the client.

At last, in line 16, the response is added to the runner as a key-value pair, once again the key could be any String.

```
1  // A static function that follows the structure of any GroIMP command
2  public static void appSayHi(Item item, Object info, Context ctx) throws Exception{
3      //If a command ist called by the API, the info variable is always the APIRunner
4      APIRunner a = (APIRunner)info;
5
6      //The API runner contains all parameters of the request
7      // the second parameter defines if the parameter is allowed to be Null.
8      String name=a.getParameter(APIRunner.INP_NAME, false);
9
10     /*****
11      * This would be the place for "real" code
12      ****/
13
14
15     //Information that is supposed to be returned to the client must be added to the api
           runner:
16     a.addResponseContent(APIRunner.RETURN_DATA, "hi "+name);
17 }
```

Listing 2.11: This static function uses the info variable as APIRunner (line 4) to receive the name from the query of the API call (line 8) and return a String with the name included as JSON response data in line 16.

The code for the workbench request (Lst. 2.12) is the same except that the context is used to resolve the workbench and to get the name of the workbench (line 3).

```
1  public static void wbSayHi(Item item, Object info, Context ctx) throws Exception{
2      //The context of a command holds the connection to the workbench.
3      String wbName= ctx.getWorkbench().getName();
4      APIRunner a = (APIRunner)info;
5      String name=a.getParameter(APIRunner.INP_NAME, false);
6      a.addResponseContent(APIRunner.RETURN_DATA, "hi "+name +", best "+wbName);
7  }
```

Listing 2.12: This static function first gets the name of the workbench it is executed on from the given context in line 3 and then follows the same steps as the example for the app command.

Adding to the registry

To create the connection between GroLink and the newly created functions, the functions must be added to the registry. For that, hooks are used. A hook is a collection of tasks that are executed

based on certain events in GroIMP. Similar to directories it is possible to add elements to existing hooks.

```
1  <registry>
2     <ref name="hooks">
3         <!-- A hook exectured if a new APIWorkbench is created-->
4         <ref name="apiLaunch">
5             <!-- adding workbench commands -->
6             <insert target="/ui/commands">
7                <command name="sayHi" run="de.grogra.apiplus.SayHi.wbSayHi"/>
8             </insert>
9             <!-- adding application commands-->
10            <insert target="/ui/commands/app">
11                <command name="sayHi" run="de.grogra.apiplus.SayHi.appSayHi"/>
12            </insert>
13        </ref>
14    </ref>
15 </registry>
```

Listing 2.13: The registry of the example plugin, with the code used to add the commands to the apiLaunch hook. If this hook is called, the insert command in line 6 will add the CommandItem with the name sayHi, that references to the wbSayHi function in the SayHi class, to the registry directory /ui/commands. Therefore an API call with the path /ui/commands/sayHi can reach it. The same is done in line 10/11 with the appSayHi command and the /ui/commands/app/ directory.

In this case, the hook apiLaunch is executed every time an API Workbench is created, and the insert command added will create CommandItems in the registry directories `ui/commands` and `ui/commands/app` (Lst. 2.13). The location of the CommandItem does not need to be one of those directories, since the absolute path is defined in the API call, yet it makes the structure simpler.

Testing

If the code above is compiled and added to GroIMPand GroLink is running on port 58081, the following URL can be opened in any web browser.

`http://localhost:58081/api/app/ui/commands/app/sayHi?name=tim`

It should return `{data: "hi tim"}`, or any other name set in the query, as defined in line 16 of the function appSayHi (Lst. 2.11).

For testing the workbench command a workbench has to be created, for example with the following API call, that creates a workbench based on the “newRGG” template with the name “myWB”:

`http://localhost:58081/api/app/ui/commands/app/create?name=newRGG&newName=myWB`

Assuming this new workbench has the id 1, the following API call would address the newly created function wbSayHi (Lst. 2.12).

```
http://localhost:58081/api/wb/1/ui/commands/sayHi?name=tim
```

This call returns beside the console and the log (Lst. 2.14), the data tag defined in line 6 (Lst. 2.12) including the name read from the parameters (Lst. 2.12 line 5) and the name of the newly created workbench resolved in line 3.

```
{
    console []
    data    "hi tim, best myWB"
    logs    []
}
```

Listing 2.14: The response of the example API call addressing a workbench called myWB with the name parameter: tim.

2.6.2 How to add new FilterSourceFactories

The CLI and the API use the same way to load examples and templates, this way is different from the way the GUI uses in GroIMP 2.1 and lower. **Therefore the examples and templates added as follows will not show in the GUI.**

With the new approach, a template or an example is a .gsz file that is added to the registry by using the newly introduced FilterSourceFactory.

It is possible to add examples and templates to an existing plugin or to create a simple new one. For simplicity, the following example is based on a new plugin.

Plugin structure

Since this plugin will not contain any Java code and does not need to be compiled the file structure is very simple:

- myPlugin:
  - plugin.xml
  - plugin.properties
  - myTemplate.gsz
  - myExample.gsz

Adding to the registry

Theregistrycontains a directory for examples and one for templates. GSZ files can be added using the FilterSourceFactory with a resource as shown in listing 2.15.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <plugin
3      id="de.grogra.myPlugin"
4      version="0.0.1"
5      xmlns="http://grogra.de/registry">
6      <registry>
7          <ref name="ui">
8              <ref name="examples">
9                  <FilterSourceFactory name="MyExample">
10                     <resource name="myExample.gsz" />
11                 </FilterSourceFactory>
12             </ref>
13             <ref name="templates">
14                 <FilterSourceFactory name="MyTemplate">
15                     <resource name="myTemplate.gsz" />
16                 </FilterSourceFactory>
17             </ref>
18         </ref>
19     </registry>
20 </plugin>
```

Listing 2.15: The example plugin.xml file used to add an example in line 9-11 and a template in line 14-16, using the FilterSourceFactory.

Additionally, the new entries must be described in the plugin.properties file as shown in listing 2.16. These properties are later shown and used for filtering, by theCLI and GroLink

```
pluginName = myPlugin
provider = grogra.de

/ui/examples/MyExample.Name = my Example
/ui/examples/MyExample.ShortDescription = a very simple example
/ui/examples/MyExample.Tags = tutorial


/ui/templates/MyTemplate.Name = newFunProject
/ui/templates/MyTemplate.ShortDescription = a new template to create a fun project
```

Listing 2.16: The plugin.properties file of the example project, which defines the name and the description of the example and the template. Additionally the example has a tag for filtering.

Adding templates or examples to a plugin with code

If a plugin already contains Java code and must be compiled the steps above are the same except that the .gsz files must be stored in src/main/resources. This should end up in a structure like the following:

- pom.xml
- src
  - main
  - java
    - de/grogra/myPlugin/SomeCode.java
  - resources
    - plugin.properties
    - plugin.xml
    - myTemplate.gsz
    - myExample.gsz
  - assembly
    - files used for packaging

Chapter 3

The API client library

3.1 Introduction

With the result of the previous chapter, it is now possible to interact with GroLink and it would also be possible to create software to do so by sending APICalls as HTTP requests. An example of that is shown in the use case "Using GroIMP as a backend for a forestry game" (sec: 4.3). Yet the direct interaction with the API does come with some inconveniencies, first of all, the calls are long and unhandy and the responses have to be interpreted. Secondly, in the case of small changes on the API server such as changing the name or the location of a command, all APICalls in all client applications would need to be changed. One common way to address these issues is to create client libraries for an API. A client library is a set of functions written in the programming language used by the client software that aliases the API calls to simple names. In the case of this project, the API Call `http://localhost:58081/api/wb/1/ui/commands/runRGGFunction?name=run` could be defined as the function `runRGGFunction("run")`. Additionally, a client library can already include the interpretation of the response of the API, leading to a user experience similar to a fully native function.

3.2 Requirements

To work with the GroLink server a client library must fulfill some requirements, starting with the sending of the request. To do so the library must be able to send HTTP requests including POST requests that contain decoded files. It must be possible to define the host and the port to enable the usage of GroLink in containers and on remote servers. Moreover, due to the plugin structure of GroIMP and the open approach of GroLink, the library should include the possibility of defining custom commands by giving the registry path to a CommandItem.

For the interpretation of the responses, an important requirement is the handling of the different

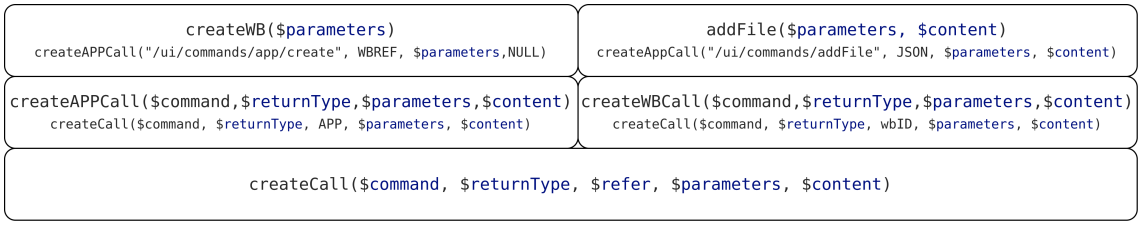


Figure 3.1: The structure of the creation of calls in the client library design. The functions call each other from top to bottom resulting in the fact that, all call objects are created by the "createCall" function. In the case of the GroLink class shown in the left stack, the createAPPCall function forwards the parameters to the createCall function and adds the app as the reference. In the case of the WBRef extension of the GroLink class on the right, the createWBCall function adds the workbench ID of the instance to the forwarded parameters. In both cases, the functions on top can be created by only adding the path and the return type.

return types. If an API Call returns a JSON object this object has to be parsed while a returned binary file should be untouched by the client library.

Finally, based on the ability to handle multiple workbenches, it would be required that the client library has a proper way to handle reference objects for workbenches making clear which workbench is addressed at any time.

3.3 Design

Since most of the requirements are highly dependent on the programming language used, this general design of a client library will focus mainly on the structure and the overall workflow.

3.3.1 Objects and Structure

Even so, a client library could completely consist of static functions that take the address of the server and the id of the workbench as additional parameters it seems more convenient to create an object-oriented approach. In that case, a workbench reference class has functions like getSourceFiles that can get the workbench ID and the server address from the instance of the class. Moreover, an individual call addressing the API can also be treated as a kind of reference that could be executed several times (like executing an RGG function 4 times in a row).

Based on these ideas of an object-oriented approach and the need for an entry point, the following three classes are designed.

Call

A call is a class that holds all information needed to send an API Call and to interpret the result. Therefore a call is defined by a link to the server, a command, a reference to either a workbench or the app (MainWorkbench), parameters, binary content and a return type. While the other parameters are needed to create the request, the return type defines how the response will be

interpreted. For the current state of the API, three return types are proposed, “JSON” for JSON data, “file” for decoded files and “WBRef” for calls that return the id of a workbench which can directly be turned into an instance of the WBRef class.

A call has two functions: run() to send the APICall and wait for the result and read() to interpret the result of the run function. Based on this structure it is easily possible to run the same call several times and it is optional to interpret the result which might not be of any interest.

GroLink

The GroLink class describes the basic connection to the API server, it is initialized with the address of this server and is used to create all calls that address the app reference (a list can be found in Appendix A.1), such as creating or opening a workbench, listing open workbenches or available examples.

The base function of the GroLink class is createCall which takes all information needed to create a call as parameters. This function is in this class only used by the createAppCall function which defines the reference as app and requires all other parameters. Every other function in this class calls createAppCall with different sets of predefined parameters (Fig. 3.1). For example, a function to open a workbench would define the command accordingly, the return type to WBRef and will in the end only require the path to the file as a parameter. This structure makes it very easy to create all different functions and to let users create new functions on the run.

WBRef

This class extends the GroLink class and adds the id of the workbench as additional information. The class uses mainly the function createWBCall which works similar to the createAppCall function but with the reference to the workbench id of the instance. And again all other functions defined in WBRef call createWBCall (Fig. 3.1). An instance of the class WBRef is normally created by calls with the return type WBRef and therefore a client application would never handle the id of a workbench as a simple variable. Nevertheless, it is possible to create a WBRef element around a given ID, this was done in the additional analysis notebook of the use case “Using GroIMP as a backend for a forestry game” (sec: 4.3).

3.3.2 Workflow

Based on the structure described above the basic workflow of a library following it would start by creating an instance of the GroLink class with the address of the running server. To create a new workbench this instance would then be used to generate the call object representing an APICall to create a workbench. After the call is executed it can be read and the result creates a WBRef object which can then be used to generate new workbench-related calls.

In listing 3.1 a pseudo code script shows the basic steps of connecting to the server, creating a workbench and executing workbench commands.

```
1 link = GroLink("localhost:58081")
2 wbCreateCall = link.createWB()
3 wbCreateCall.run()
4 wbref1 = wbCreateCall.read()
5
6 runRGGCall = wbref1.runRGGfunction("run")
7
8 runRGGCall.run()
9 runRGGCall.run()
10 runRGGCall.run()
```

Listing 3.1: A pseudo-code script showing the theoretical usage of a GroLink client library, starting with the initialization of the GroLink object in line 1. The creation of calls can be seen for app commands in line 2 and for WB commands in line 6. These commands are executed in line 3 and in lines 8-10. The usage of the read function of a call is shown in line 4 with the return of the WBRef element wbref1.

3.4 GroPy

GroPy is the Python implementation of the designed concept of a GroLink client library.

Python was chosen not only because it is one of the most used programming languages [24], but because it is used in many (data)-science projects including OpenAlea [25]. As introduced in section 1.3.2 there were already first steps towards co-simulation between OpenAlea and GroIMP. Therefore a client library could be the first step to recreate the link between the two projects and create new links to other projects.

3.4.1 Implementation

The GroPy implementation follows the complete design for client libraries described above and therefore contains a Call, a WBref and a GroLink class. All classes are stored in one file to simplify including the library without installing (described below). It is developed and tested for Python 3.9 and 3.12 but should work for every Python version higher than 3.7.
To handle the different types of HTTP requests the Python library “requests”[26] is used. Besides that, no other external library is needed since Python comes with a well-working JSON parser.

3.4.2 Deployment and installation

Installing with pip

The GroPy library can be installed using the pip package manager with a custom index URL. Pip is the most common way to install Python packages and is installed with any newer version of Python

[27]. To install the GroPy library the command in listing 3.2 is used. This command works similar to other pip commands, including `--upgrade` for updating to a new version or `pip uninstall GroPy` to remove the library.

```
pip install GroPy --index-url https://gitlab.com/api/v4/projects/50527255/packages/pypi
    /simple
```

Listing 3.2: The command to install the GroPy library using the PIP package manager.

Linking the source

If installing with pip is not feasible, it is also possible to simply download the Python file from the repository, place it in the same directory as the project and import it as a local file.

Deploying the pip package

The GroPy project is hosted on Gitlab and uses a deployment pipeline [28] to create the package for the pip installer. This pipeline is executed any time a new version tag is created and then runs a chain of commands that build the package [29] and push it to the package registry of the repository [30]. To do so the Python packages build [31] and twine [32] are used.

To deploy a new version of the library with this pipeline it is only necessary to change the version of the library in the file “pyproject.toml” and create a new tag by either pushing or using the web interface.

3.4.3 Usage

To show the usage of the GroPy library some basic operations are shown in the following. It is at any point necessary that GroIMP is running the GroLink application. A collection of more complex examples can be found in appendix C.

Connecting to the GroLink server

After importing the GroPy library (Lst. 3.3 line 1) the connection to the server can be defined. As described in the design, the GroLink object is the base element of the client library and must be initialized with the path to the server. This is shown in listing 3.3 in line 2, where the link object is created.

This link object by now did not interact with the server, it is only a collection of commands to create calls. These commands can be found in the appendix B.1.1.

```
1 from GroPy import GroPy
2 link = GroPy.GroLink("http://localhost:58081/api/")
```

Listing 3.3: Example code of importing the GroPy library and creating an instance of the GroLink object based on the address of the Server, in this case localhost:58081. If the GroPy library was not installed but linked (see above) the first line canges to "import GroPy".

Create a new workbench reference

Using the above-created object link, it is now possible to create application calls, for instance, to create a new project (Lst. 3.4 line 1). If this call is then run (Lst. 3.4 line 2) it sends the API Call to the server. On the server side, this call creates a new APIWorkbench and returns the identification of it. This return value is captured by the call object and if this object is read (Lst. 3.4 line 3) it creates a new reference to the workbench (WBRef).

In conclusion the code in listing 3.4 does the same as the menu entry 'File /new/RGG Project' on the GUI.

```
1 wbCreateCall = link.createWB("newRGG") # create the API call object
2 wbCreateCall.run() # execute the API call object
3 wb1 = wbCreateCall.read() # reading the results of the API call and create a WBRef
      object
```

Listing 3.4: The creation of the first workbench using the GroPy library, and the concepts of calls as described in the library design.

On a more abstract level the code in listing 3.4 shows how GroPy handles calls. A call is created by the functions of the GroLink object 'link'. This call represents an API request but did not interact with the GroLink server yet. This happens when the `run()` function is executed. Afterwards, the response is stored in the call object and is interpreted when the `read()` function is executed. The process can be simplified as shown in listing 3.5, yet this only affects the structure of the code not the way of execution.

```
wb1 = link.createWB("newRGG").run().read() # an inline solution of the code above
```

Listing 3.5: An example for the inline execution of an APICall in GroPy, the creation of the call, the sending and the interpretation are done one by one.

Using the now-created WBRef object wb1, a list of commands to create new calls can be executed. This list can be found in the appendix B.1.2.

Receiving information

With the WBRef object 'wb1' it is now possible to get information about the workbench, such as the project graph (Lst. 3.6 line 2) or available RGG functions (Lst. 3.6 line 4).

```
1 #print the Project graph
2 print(wb1.getProjectGraph().run().read())
3 #get the list of RGG functions:
4 functions = wb1.listRGGFunctions().run().read()
5 for f in functions['data']: # reading all entries form the data entry
6     print(f)
```

Listing 3.6: Using GroPy inline calls to receive and print the ProjectGraph (line 2), and to store the available RGG functions in a python object (line 4). This object is then processed in lines 5 and 6.

By default, each function that is defined with the return type 'JSON' returns a Python dictionary that then can be processed as usual (Lst. 3.6 line 5,6). For workbench functions, this dictionary always includes 'console' for the content of the XLConsole and 'log' for application information, as described in the design of the API server. Additionally, in most cases, the data that was requested is returned in 'data' (Lst. 3.6 line 5). In some more complex results such as the ProjectGraph, the response is divided into several dictionary entries as shown in listing 2.4.

Executing RGG and XL

As the name says, interactive modelling is a key feature of GroIMP and therefore also for GroLink and the client libraries. This can be done by RGG functions and XL queries, which are usable in GroLink and as shown in listing 3.7 in GroPy.

```
1  # create the call that counts all A nodes
2  countA = wb1.runXLQuery("count((*Model.A*));")
3  # create the call that executes the rgg command 'run'
4  execRun=wb1.runRGGFunction("run")
5  # execute the count and read the results
6  print(countA.run().read())
7  # execute the run function 10 times without printing because the result is not needed
8  for i in range(10):
9      execRun.run()
10 # execute the count and read the results a second time
11 print(countA.run().read())
```

Listing 3.7: Example usage of RGG functions and XL queries with the GroPy library. First, the Call objects for an XL query (line 2) and the RGG function (line 5) are created. These calls are then run multiple times (lines 6, 9, 11) concluding in a workflow of counting the number of A nodes, triggering the RGG function "run" 10 times and then counting the A nodes again.

XL queries as shown above (Lst. 3.7 line 2) work similarly to the XLConsole in the GUI, including the usage of rewriting rules. Only the variables of the XLConsole are not usable with the GroLink project.

The call created with runRGGFunction (Lst. 3.7 line 5) starts the same process as the button in the RGG toolbar in the GUI.

```
1  # update the Model.rgg file like in JEdit in the GUI
2  wb1.updateFile("Model.rgg","""
3  module B(float len) extends Sphere(0.1)
4  {
5      {setShader(GREEN);}
6  }
7
8  public void change ()
9  [
10     B(x) ==> F(x) B(x);
11 ]
12 """).run()
13 # compile to build the new scene
14 wb1.compile().run()
15 wb1.addNode("xeg","""
16 <graph>
17   <root root_id="0"/>
18   <type name="B">
19     <extends name="Sphere"/>
20     <property name="len" type="float"/>
21   </type>
22   <node id="1" name="" type="B">
23     <property name="radius" value="0.1"/>
24     <property name="color">
25       <rgb>0.0 1.0 0.0</rgb>
26     </property>
27     <property name="len" value="1.0"/>
28   </node>
29   <edge id="2" src_id="0" dest_id="1" type="decomposition"/>
30 </graph>
31 """).run()
32 wb1.runRGGFunction("change").run()
33 print(wb1.export3d("xeg").run().read().decode('utf-8'))
```

Listing 3.8: A GroPy pipeline to add an XEG file, manipulate it and export the result as a new XEG.

Adding and exporting graph structures

In order to show the abilities of the GroPy library regarding adding and exporting graph structures, the following example (Lst. 3.8) is introduced.

This example uses the same workbench as created above (sec. 3.4.3) but changes the environment by updating the Model.rgg file (Lst. 3.8 line 2-12) to create a module B and an RGG function change containing the rule to replace a B with an F and a B (Lst. 3.8 line 10). After the changes are compiled (Lst. 3.8 line 14) an XEG containing one B Module is added(Lst. 3.8 line 15-31). Finally, the newly created change is executed (Lst. 3.8 line 32) and the now changed project graph is exported as XEG and printed (Lst. 3.8 line 33).

The printed result can be seen in listing 3.9, including an F node (line 15) and a B node (line 7). In comparison to the added XEG in listing 3.8 line 16-30, the additional F shows that the rule defined in the change function was applied.

```
1  <graph>
2    <root root_id="0"/>
3    <type name="B">
4      <extends name="Sphere"/>
5      <property name="len" type="float"/>
6    </type>
7    <node id="2" name="" type="B">
8      <property name="radius" value="0.1"/>
9      <property name="color">
10       <rgb>0.0 1.0 0.0</rgb>
11     </property>
12     <property name="len" value="1.0"/>
13   </node>
14   <edge id="3" src_id="3" dest_id="2" type="successor"/>
15   <node id="3" name="" type="F">
16     <property name="length" value="1.0"/>
17     <property name="diameter" value="-1.0"/>
18     <property name="fcolor" value="-1"/>
19   </node>
20   <edge id="5" src_id="0" dest_id="2" type="decomposition"/>
21   <edge id="6" src_id="0" dest_id="3" type="decomposition"/>
22 </graph>
```

Listing 3.9: The results of the example XEG pipeline, including an F and a B node.

Saving and closing the workbench

The last usability introduced at this point is the saving and closing of workbenches, and it is important to mention that if the workbench is not saved before closing the changes will be lost without additional questions.

As shown in listing 3.10 two ways of saving exist. The first one (lines 2-5) reads the GSZ file from the body of the response and handles it then like a “normal” Python object. While the second way (line 8) just defines the path where the server is supposed to store the project.

```
# Saving the returned data
data=wb1.save().run().read() # receive the gsz file as binary data
f = open("result.gsz",'wb') # open a binary file
f.write(data) # save the file
f.close() # close the file

# save to a path on the System
wb1.save(path="/absolute/path/to/the/saved/project.gs").run()
```

Listing 3.10: The two ways how GroPy can saves projects.

The two different ways exist for two different use cases: The first one does not require a shared file system between the server and the client. This means it is possible to run GroIMP on a different system such as a remote computer or a container (e.g. Docker). Therefore the first way also only supports .gsz files, since all information is returned as one file. The second is handy in other scenarios and supports GS and GSZ.

The closing itself is then quite simply the "close" command shown in listing 3.11.

```
# close the workbench
wb1.close().run().read()
```

Listing 3.11: Closing a workbench using GroPy

Afterward, the API can be closed by crl+c in the console or with the command `link.close().run()`.

3.5 GroR

GroR is an R client library that does not fully follow the design concept, it is missing most of the object-oriented approach and the usage of the Call class.

The selection of R as a client library was inspired by the project of Benjamin Spehle [17] (introduced in section 1.3.3), how used R Shiny, an R-based web visualization tool [33]. Moreover, R is well-known and used in the fields of Bioinformatics and data science [34].

3.5.1 Implementation

In contrast to the GroPy implementation, the GroR implementation is much simpler. The object structure is replaced by two groups of functions: GroLink functions and WBRef functions which do not create call elements but send the request directly. Furthermore instead of object-based the functions are static and use an additional parameter for the reference to the API. The GroLink functions use simply the URL as an additional parameter and the WBRef functions use a tuple of

the URL and the ID of the workbench.

The GroR library depends on the httr2 package [35] for HTTP requests and the dplyr package[36] for the pipe element `%>%`.

3.5.2 Installation

The GroR package is not added to any public repository and must therefore be installed by hand or added directly as a source.

Adding the R file as a Source

It is possible to download the GroR.R file from the GitLab repository and link it to an R file using the source command (Lst. 3.12). If this way is used the command in listing 3.12 replaces the usage of `library(GroR)` in listing 3.14.

```
source("path/to/GroR.R")
```

Listing 3.12: Adding the GroR library directly as an R file to a project.

Package

The releases on the GitLab repository contain R-packages that can be installed directly with either the R command in listing 3.13, or with the R Studio package manager by selecting “install from package archive file”.

```
install.packages("path/to/GroR_x.x.x.tar.gz", repos = NULL, type = "source")
```

Listing 3.13: Installing the GroR library from the released package.

3.5.3 Usage

To show the usage of the GroR library the same basic operations as for the GroPy library are shown in the following. Since the general tasks are the same this section will be briefer by mostly referring to the usage of the GroPy library (sec: 3.4.3).
It is at any point necessary that GroIMP is running as the API server.

Creating the workbench

Since the commands in GroR are static the creation of a workbench (Lst. 3.14 line 7) takes the address of the server as an parameter. Therefore no other initialization of the library is necessary, except for importing it and its requirements (Lst. 3.14 line 2-4).

```r
#import the needed libraries
library(dplyr)
library(httr2)
library(GroR) # can be replaced with source(.../GroR.R) based on the installation

# create a new workbench identification
wbID1<-GroLink.create("http://localhost:58081/api")
```

Listing 3.14: An example showing how to create a new workbench using the GroR library.

The library contains a collection of functions starting with "GroLink" to manage the application, a list of them can be found in the appendix B.2.1.

The newly created object wbID1 is used in the following to define which workbench is addressed by calls. Using this object it is possible to use the second collection of functions of the library, which start with "WBRef". The list of these functions can be found in the appendix B.2.2.

Receiving information

Similar to the usage of GroPy it is possible to use the wbID1 object to receive information from the created project (Lst. 3.15). The listRGGFunctions returns parsed results that can be queried (Lst. 3.15 line 4) while getProjectGraph does not. It's that way because the JSON parser that is included in the library is quite minimalistic and parsing the ProjectGraph is not properly possible with it. The solution is to use a proper JSON library, this is done in the use case: "Interactive web interfaces for RGG-based GroIMP models" (sec. 4.5).

```r
print(WBRef.getProjectGraph(wbID1)) #getting the project graph

functions <- WBRef.listRGGFunctions(wbID1) # requesting the list of rgg functions
functions$data # the part of the result that is interesting
```

Listing 3.15: GroR function to receive the Project Graph as a String (line 1) and the RGG functions as a list (line 3).

Executing RGG and XL

Using the GroR library the same interactions with the simulation as with the GroPy library are possible. As shown in listing 3.16, this includes executing RGG functions (line 6) and the evaluation of queries (lines 2,10). Moreover, the usage of rewriting rules would be possible as well.

```
# run a query that counts the amount of A nodes in the Model
WBRef.runXLQuery(wbID1,"count((*Model.A*))")

#execute the run function 10 times
for(i in c(0:10)){
  WBRef.runRGGFunction(wbID1,"run")
}
# run a query that counts the amount of A nodes in the Model

WBRef.runXLQuery(wbID1,"count((*Model.A*))")
```

Listing 3.16: A GroR example to show the execution of RGG functions and XL queries, based on the reference object wbID1.

Adding and exporting graph structures

The example in listing 3.18 is in its execution identical to the GroPy example described in section 3.4.3. Therefore the Model.rgg file from the workbench referenced in wbID1 is updated in lines 1-10 and then compiled. After adding an XEG (lines 14-28) and running the change function, a new XEG is exported. This graph is identical to the one from the Python example and can be seen in listing 3.9.

Saving and closing the Workbench

Besides the way shown in listing 3.17, it is also possible to receive the project as binary data from the workbench. Yet at this place, only the saving to a path is shown (line 1) since the handling of binary data in R would lead quite far.

```
WBRef.save(wbID1, "/path/to/result.gsz") #save the project
WBRef.close(wbID1) # close the workbench
```

Listing 3.17: The GroR commands for saving and closing a workbench.

```
WBRef.updateFile(wbID1,"Model.rgg",'
  module B(float len) extends Sphere(0.1)
  {
    {setShader(GREEN);}
  }

  public void change ()
  [
    B(x) ==> F(x) B(x);
  ]')

WBRef.compile(wbID1)

WBRef.addNode(wbID1,"xeg",'<graph>
  <root root_id="0"/>
  <type name="B">
    <extends name="Sphere"/>
    <property name="len" type="float"/>
  </type>
  <node id="1" name="" type="B">
    <property name="radius" value="0.1"/>
    <property name="color">
      <rgb>0.0 1.0 0.0</rgb>
    </property>
    <property name="len" value="1.0"/>
  </node>
  <edge id="2" src_id="0" dest_id="1" type="decomposition"/>
</graph>')

WBRef.runRGGFunction(wbID1,"change")

print(WBRef.export3d(wbID1,"xeg"))
```

Listing 3.18: A GroR implmentation of a simple XEG manipulation pipeline.

Chapter 4

Use cases

4.1 Introduction

In order to show the abilities of the newly created application and the client libraries, use cases were designed, inspired by previous projects and the fields of application defined in the introduction.

These use cases are only an advance in the different directions, designed to highlight the different possibilities GroLink provides in the above-discussed areas of application. Additionally, these different directions are briefly discussed for each use case.

All following examples work with the GroLink version released within GroIMP 2.1 and are tested in a containerized scenario.

4.2 An XEG processing pipeline

The workflow of this example was inspired by the work of Qinqin Long [12] on the co-simulation of GroIMP and OpenAlea, which was introduced in section 1.3.2. Yet for this project, only the GroIMP side of the approach was considered, meaning the importing and processing of XEG files. Instead of the OpenAlea integration, the client side is designed as a command line tool, inspired by the shortly discussed field of application "GroIMP as a file converter" (sec. 1.4.1).

4.2.1 Description

The final tool takes the path to an XEG file as a parameter, runs a pipeline and returns the changed XEG file. This pipeline runs a GPUFlux light model and adds the absorbed light to the leaves of a presented XEG file.

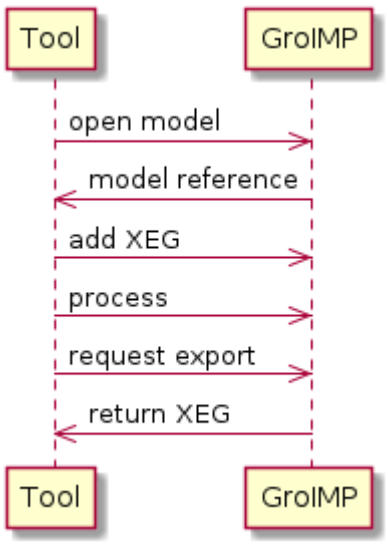


Figure 4.1: A UML sequence diagram showing the workflow of a simple XEG pipeline.

4.2.2 Concept

To properly interact with an XEG that includes custom nodes these nodes must be defined as RGG modules in the GroIMP project before adding the XEG file. Therefore the interaction with GroLink (figure. 4.1) starts with the definition of the GroIMP environment, which also includes the definition of the used light system and the function to calculate the absorbed light.

After the environment is initialized the XEG file can be added, processed and exported. The processing executes the GPU raytracing, queries all leaves and sets the absorbed light as a parameter.

4.2.3 Implementation

The implementation is separated into the GroIMP environment/model that processes the simulation and the client software that initializes the pipeline. In this case, the client is developed in Python using the GroPy library.

GroIMP

The RGG code of the model can be seen in the listing 4.1. The leaves used in this model are parallelograms with an additional parameter energy for the amount of light absorbed which is initialized with zero (Lst. 4.1 lines 1-5). Due to the current implementation of the XEG interface in GroIMP, the light source must be added after the XEG file is imported and removed before the new XEG is exported. Therefore the RGG function that calculates the light is also adding the light node to the RGGRoot-node (Lst. 4.1 line 10) and removing it later (4.1 line 20). In between that, the light model is computed in line 13 and the leaves are updated in line 16.

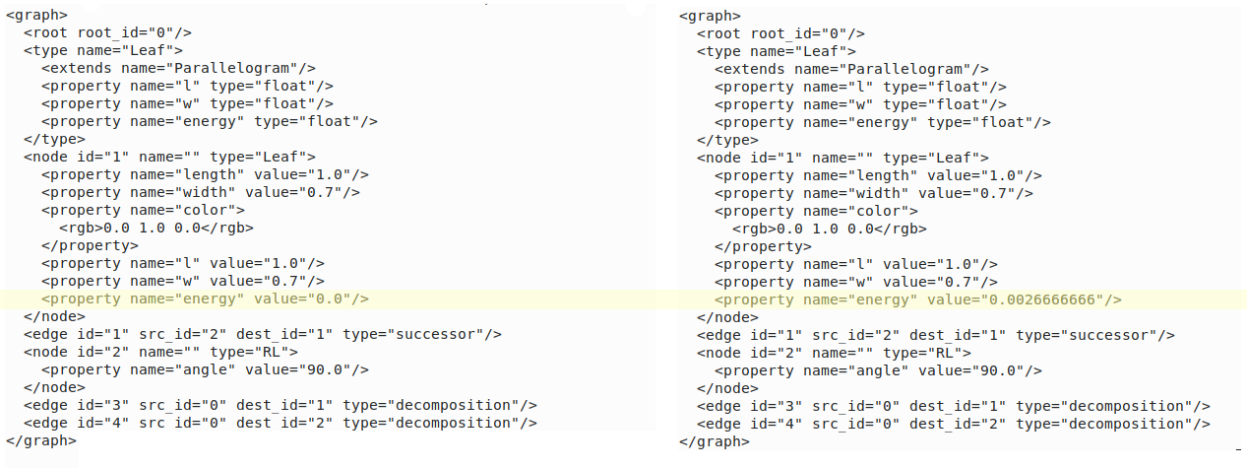
```
1  module Leaf(float l, float w) extends Parallelogram(l,w)
2  {
3      float energy=0;
4      {setShader(GREEN);}
5  }
6  static FluxLightModel lm = new FluxLightModel(50000, 5);
7
8  public void calc_light(){
9      //add LightNode
10     [ ==>^ [M(50) RL(180) LightNode() ];]
11     //calculate light
12     derive();
13     lm.compute();
14     //Update leaf energy
15     [
16     le:Leaf::>{le.energy=lm.getAbsorbedPower(le).integrate();}
17     ]
18     //remove LightNodes
19     [
20         M(50) RL(180) LightNode ==>;
21     ]
22 }
```

Listing 4.1: The example model used for the GPUFlux based XEG processing pipeline. Besides defining the needed RGG module and the light model, the calc light function is the core of this model. This function adds a simple light environment, calculates the light model, updates existing leaves and removes the light environment. In this way a before included graph is left behind with processed leaves and no artifacts.

GroPy

Using the above-described model with the GroPy library can result in the pipeline shown in listing 4.2. This pipeline starts with the creation of the link to the GroLink API server (line 3), as described in the section about the usage of GroPy (sec. 3.4.3). Afterward, the GSZ file (based on lst. 4.1) is read as binary and sent to the server to create a workbench and a workbench reference (Lst. 4.2 line 4). In the same way, the XEG file which is defined by the first command line parameter (`sys.argv[1]`) is read and sent to the workbench in line 5. To process the now imported graph, the `calc_light()` function (Lst. 4.1 lines 8-23) is executed using the runRGGFunction in line 6. Finally, the processed graph can be exported to XEG, which then has to be decoded to utf-8 before printing (lst 4.2 line 7). Since after the printing, the workbench is no longer needed it is closed in line 8. This tool could be executed with a command like this: `python xegPipeline.py simpleTree.xeg`.

The results of a simple example of only a single leaf can be seen in figure 4.2, the pipeline was also tested and validated with more complex tree structures.



```
<graph>
  <root root_id="0"/>
  <type name="Leaf">
    <extends name="Parallelogram"/>
    <property name="l" type="float"/>
    <property name="w" type="float"/>
    <property name="energy" type="float"/>
  </type>
  <node id="1" name="" type="Leaf">
    <property name="length" value="1.0"/>
    <property name="width" value="0.7"/>
    <property name="color">
      <rgb>0.0 1.0 0.0</rgb>
    </property>
    <property name="l" value="1.0"/>
    <property name="w" value="0.7"/>
    <property name="energy" value="0.0"/>
  </node>
  <edge id="1" src_id="2" dest_id="1" type="successor"/>
  <node id="2" name="" type="RL">
    <property name="angle" value="90.0"/>
  </node>
  <edge id="3" src_id="0" dest_id="1" type="decomposition"/>
  <edge id="4" src_id="0" dest_id="2" type="decomposition"/>
</graph>
```

```
<graph>
  <root root_id="0"/>
  <type name="Leaf">
    <extends name="Parallelogram"/>
    <property name="l" type="float"/>
    <property name="w" type="float"/>
    <property name="energy" type="float"/>
  </type>
  <node id="1" name="" type="Leaf">
    <property name="length" value="1.0"/>
    <property name="width" value="0.7"/>
    <property name="color">
      <rgb>0.0 1.0 0.0</rgb>
    </property>
    <property name="l" value="1.0"/>
    <property name="w" value="0.7"/>
    <property name="energy" value="0.0026666666"/>
  </node>
  <edge id="1" src_id="2" dest_id="1" type="successor"/>
  <node id="2" name="" type="RL">
    <property name="angle" value="90.0"/>
  </node>
  <edge id="3" src_id="0" dest_id="1" type="decomposition"/>
  <edge id="4" src_id="0" dest_id="2" type="decomposition"/>
</graph>
```

Figure 4.2: The same XEG before and after running through the pipeline. As shown in the highlighted row the energy variable changes due to the calculation in GroIMP.

```
1 from GroPy import GroPy
2 import sys
3 link = GroPy.GroLink("http://localhost:58081/api/")
4 wb1 = link.openWB(content=open("tree_calc.gsz",'rb').read()).run().read()
5 wb1.addNode("xeg",open(sys.argv[1],'rb').read()).run()
6 wb1.runRGGFunction("calc_light").run()
7 print(wb1.export3d("xeg").run().read().decode('utf-8'))
8 wb1.close()
```

Listing 4.2: The example client tool of the XEG processing pipeline.

4.2.4 Evaluation

This simple example works nicely and gives a first impression of the abilities of GroLink regarding the integration into other software such as OpenAlea and the usage for automatization of tasks. Moreover, this example shows that GroLink should be able to provide the GroIMP side of the original design approach from the GroAlea plugin by Christophe Pradal [37]. At the current state, pipelines like this one need a lot of specification in the GroIMP model, yet hopefully, this will change with future improvements of XEG. Additionally, more advanced functions for adding and exporting graph structures could be very beneficial improvements to the GroLink API. This could include the ability to reference a parent node for adding or exporting.

4.2.5 Perspective

This use case shows that only a few lines on the client side are needed to create an automated task. Therefore more advanced approaches could be used in different ways, either for small tools aiming at simple tasks like file transformations or for fully implemented co-simulation as intended by Qinqin Long and Christophe Pradal. For the second perspective, it would even be possible to go further and create Visualea nodes for each GroLink API Call, which would enable full interaction.

4.3 Using GroIMP as a backend for a forestry game

This scenario was discussed by different teams in the past and is mostly focused on the interaction with the 3D scene: The life integration of a GroIMP model into a game engine. This is not a very scientific use case and would most likely only be interesting for presentation or teaching but it was here added because it highlights the fact that the GroLink API server can be used without any client library and from software that is not focusing on any similar topic. Additionally, it shows some similarities to the GroIMP part of the work done by Dirk Lanwert [11] (introduced in section 1.3.1) and also to the approach of Marek Fabrika on the SIBYLA platform [38].

4.3.1 Description

For this use case, the "game" is rather simple, a player can walk through a forest stand, cut or plant trees, get basic information about the trees and trigger new growth steps. The idea is that each of the player's actions (except movement) triggers an interaction with a GroIMP model by either XL queries (getting information), XL rules (plant or cut trees) or RGG functions (growth step).

4.3.2 Concept

The concept of the game described above can be separated into 4 sequences of communication between the user, the game and GroIMP.

Initialization and model selection

The initial sequence shown in figure 4.3 starts by presenting the possible scenarios that can be played. After the user selects one of them, GroIMP loads the model, initializes the simulation and returns the model ID to the game. Using this ID the game can request a 3D model of the scene which then is displayed to the user.

Growing the forest

The growth process (Fig. 4.4) is started when the user triggers an event by pressing a button or a key. This event leads the game engine to send an API call to the GroLink server, which requests

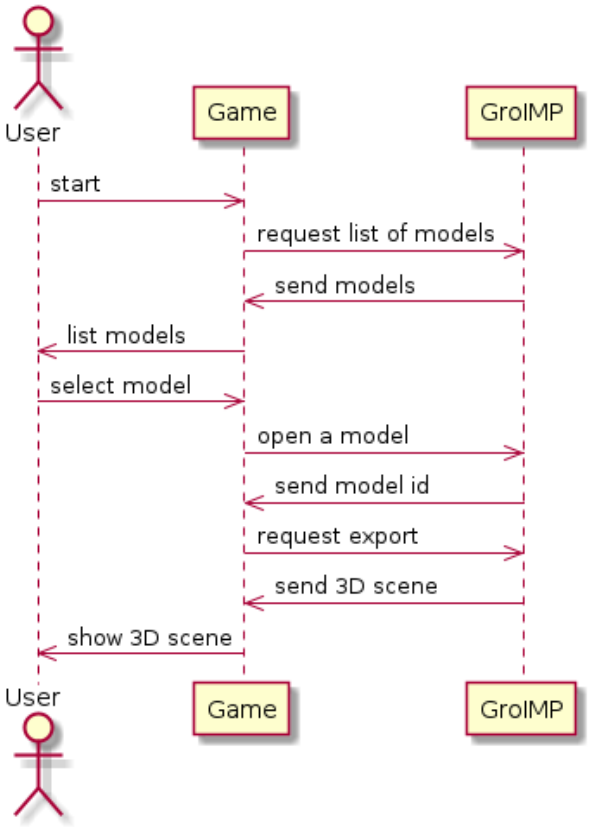


Figure 4.3: A UML sequence diagram showing the different steps needed to start a GroIMP-based game out of a list of possible models.

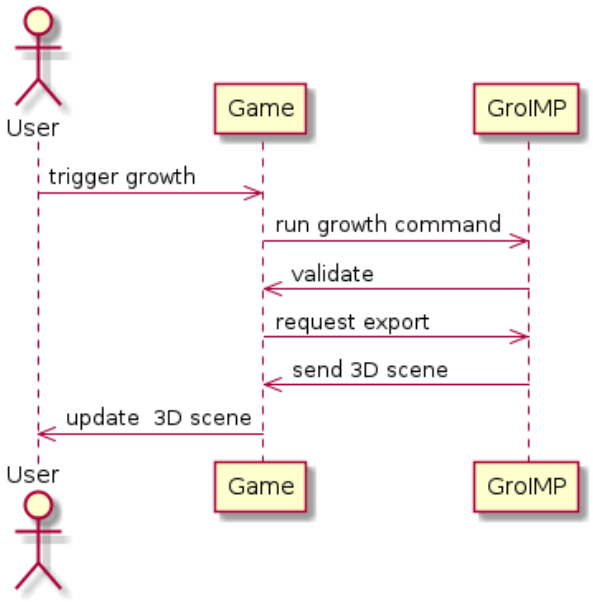


Figure 4.4: A UML sequence diagram showing the different steps required to execute an RGG growth function and to update the 3D scene in the game.

the execution of a growth function. Afterwards, the game requests a 3D scene which it then uses to update the environment of the game.

Cut a tree

Figure 4.5 shows the sequence that is started if the user selects a tree with the cut option. The game sends an XL query to GroIMP that replaces the selected tree with nothing. Afterwards, the game requests a 3D scene which it then uses to update the environment of the game.

Plant a tree

Planting a tree works very similar to cutting one, the user selects a location where a new tree is supposed to be placed and the game sends an XL query to add a tree object at the given location (Fig. 4.6). Afterwards, the environment is updated the same way as in the previous sequences.

Get Info

To get information about a tree the user selects the tree and the game sends a query to GroIMP. GroIMP returns the information and the information is shown (Fig. 4.7).

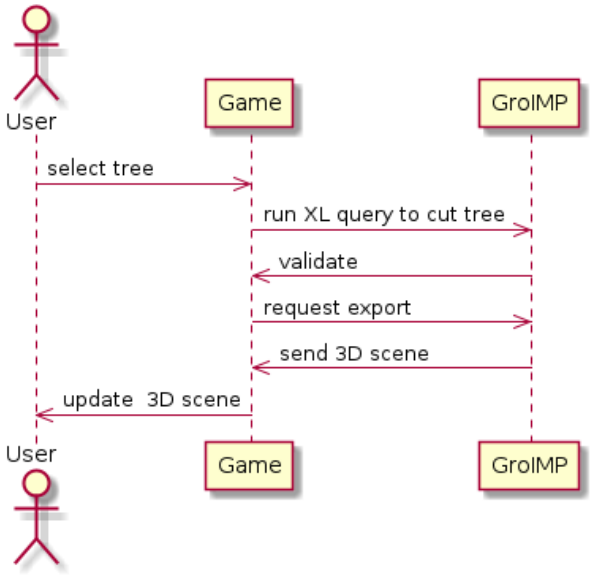


Figure 4.5: A UML sequence diagram showing the different steps required to cut a selected tree and update the 3D scene

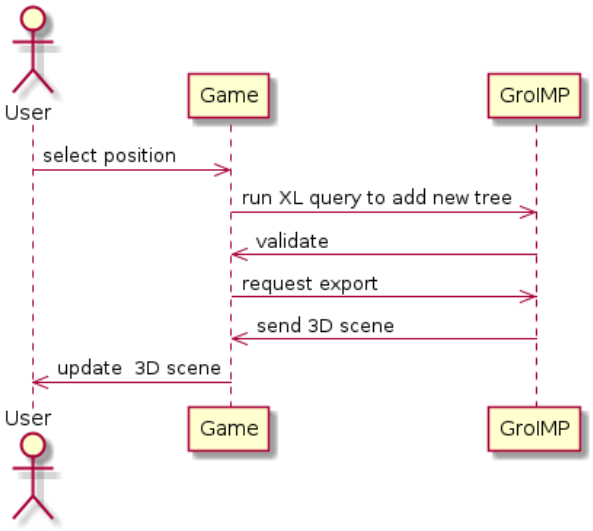


Figure 4.6: A UML sequence diagram showing the different steps add a tree at a a specific tree based in GroIMP and update the 3D scene

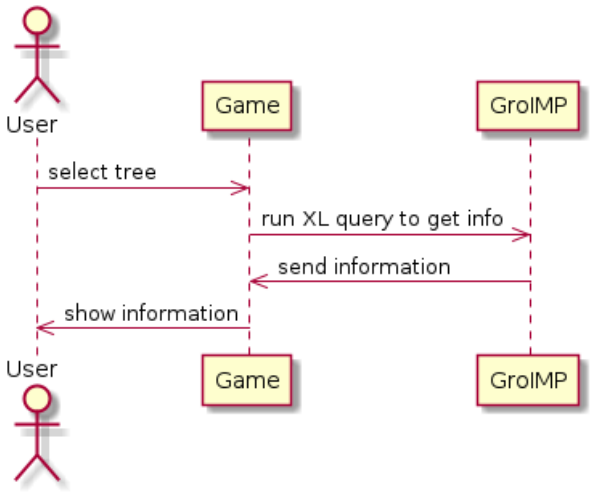


Figure 4.7: A UML sequence diagram showing the different steps required to receive information about a tree based on an XL query

4.3.3 Implementation

For the implementation of the game, the open-source game engine Godot [39] was used. The engine combines a drag-and-drop interface editor with its own programming language for event handling. This programming language is capable of HTTP requests which qualifies the platform as an GroLink API client.

To simplify the API handling and to further test the abilities of the new FilterSourceFactory the forest models for the game were added to GroIMP in a newly created plugin and registered as examples with the tag forester.

The current version contains three different forest models, which each contain two different types of trees, a conifer and a deciduous. The first model visualizes the trees as simple cylinders with either a sphere or a cone. In this model, the growth is simulated based on a very simple linear age-height relation. The second model uses a cone-based approach to estimate the spatial competition for each tree. This competition is used as a factor in the growth simulation of the tree. The last model (figure 4.9) is based on parametric functional structural tree models from a previous project [40] and its growth function is based on a much more complex approach including light competition.

For the 3D file format, OBJ was the only option since it is the only format supported by both platforms and even for that some changes were necessary. First, the additional gd-obj project [41] was used to enable live parsing of OBJ files in Godot and OBJ color support was enabled in both the gd-obj project and GroIMP.

Figure 4.8: The menu of the forest simulator game after the connect button is pressed. The available scenarios are listed and on the right a small summary can be seen. The wooden panel was created by Alexandra Alarie and published on public domain [42].

Initialization and model selection

After starting the forest game, the address of the running GroLink server must be added (the dialog can be seen in figure 4.8). The connect button then executes an HTTP request querying the examples based on the tag "forester". The result is then listed in the list box below the address (Fig. 4.8) and if a model is selected the description of the example, added in the GroIMP plugin, is shown on the right.

The selected model can now be started by clicking the run button in the bottom right corner (Fig. 4.8) and it is possible to select if the scenario is supposed to include grown trees or not, by the checkbox above the run button.

To start the model the game calls the API and requests to load the selected example and stores the returned ID for future communication. Additionally, the game requests the graph rewriting rule for creating the trees used in this model. This is necessary since the different scenarios have different trees and it is solved by a simple RGG function that both GroIMP models include and which just prints the requested queries in the console which later can be read by the client software. Afterward, if selected, a function to add grown trees is executed.

Finally, with the ID and the rules to create trees, the actual game is initialized by changing to a new Godot scene, requesting an OBJ export of the current state of the simulation and adding the

Figure 4.9: The interface of the loaded game, with the control options at the top of the screen. The panel in the middle holds the selectable actions: cutting, planting conifers, planting deciduous and getting information as well as the button to grow the forest. On the right, a small help button can be found next to the close button. In the top left corner an example variable cash is shown which changes depending on cutting and planting trees. The grass used at the bottom was created by LuminousDragonGames and published under public domain [43].

result to the new scene (Fig. 4.9). The user can now use keyboard control to navigate through the simulation.

Growing the forest

The simplest action to implement was the growth of the forest which just sends an API Call to run the growth function of the GroIMP model (Lst. 4.3) and then updates the 3D scene by overwriting the existing scene with the newly exported OBJ model. The growth is triggered by either clicking on the small hourglass in the menu (Fig. 4.9) or by pressing whitespace.

```
$http/grow.request(
    Global.getURL()+"/api/wb/"+id+"/ui/commands/runRGGFunction?name=run"
)
```

Listing 4.3: Godot code to create an HTTP GET request based on the address of the server (getURL()), the ID of the workbench and the path to the API call for executing RGG functions. This request triggers the execution of the rgg function "run" which is simulating the growth.

Cut a tree

To cut a tree the little axe in the menu must be selected (Fig. 4.9) and the tree must be clicked with the mouse. Since the whole stand is exported as one OBJ file it is not possible to properly select a tree in Godot and get any identification. Therefore the position of the mouse projected on the floor is used to run an XL query to detect the tree closest to this position and remove it. The Godot code to create this query can be seen in listing 4.4.

Additionally, the volume of the tree is returned and used to increase the cash variable of the game. Finally, the 3D scene is updated and the selected tree is gone.

```
1  $http/cut.request(
2      Global.getURL()+"/api/wb/"+id+"/ui/commands/runXLQuery",
3      ["Content-Type: application/json"], HTTPClient.METHOD_POST,
4      "[
5          t:Model.Tree,(location(t).x > "+str(x-0.15)+" &&
6                        location(t).x < "+str(x+0.15)+" &&
7                        location(t).y > "+str(z-0.15)+" &&
8                        location(t).y < "+str(z+0.5)+")
9              ::>{println(t.getVolume());[t==>>;]}
10     ]"
11 )
```

Listing 4.4: The Godot code of the API request for an XL query to cut a tree. The function uses the address that was set in the menu and the id of the workbench to create a POST request (lines 2 and 3). It was necessary to use a POST command since the request contains an XL query (lines 4-10) that includes characters that are in conflict with the URL standard. This query selects a tree based on given coordinates and replaces the tree with nothing(removes it). Besides that, the volume of the tree is printed so it can be found in the returned information later. In lines 7 and 8 it can be seen that GroIMP and Godot handle coordinates differently therefore the z coordinate in Godot is the y coordinate in GroIMP.

Plant a tree

To plant a tree the projected mouse position is used again but as values for a GroIMP translation which then is connected with the earlier received graph rules to create new trees of the selected type (conifer or deciduous see figure 4.9). This combined graph description is then added to the

floor of the GroIMP simulation and therefore a new tree is created on the given location (Lst. 4.5). Afterwards, the 3D scene is updated.

```
$http/plant.request(
    Global.getURL()+"/api/wb/"+id+"/ui/commands/runXLQuery?xl=[Floor==>Floor[Translate("
        +str(x)+","+str(z)+",0)"+Global.getcTree()+"];]"
)
```

Listing 4.5: The Godot code to create the API call that requests the execution of the XL query that places a new tree. This function can be a GET request, therefore only the URL has to be defined based on the address (getURL()) and the workbench as well as the coordinates the user selected and the graph rewriting rule for creating trees in the selected model. As mentioned above this rule is requested from the model during the initialization.

Get Info

To receive information about a tree (lst: 4.6) the same technique as for cutting trees is used, but instead of rewriting, the getInfo() (line 9) function is executed. This function prints information about the tree to the console which then can be read and displayed by the client.

```
1  $http/info.request(
2      Global.getURL()+"/api/wb/"+id+"/ui/commands/runXLQuery",
3      ["Content-Type: application/json"], HTTPClient.METHOD_POST,
4      "[
5          t:Model.Tree,(location(t).x > "+str(x-0.15)+" &&
6                        location(t).x < "+str(x+0.15)+" &&
7                        location(t).y > "+str(z-0.15)+" &&
8                        location(t).y < "+str(z+0.5)+")
9                  ::>{t.getInfo();}
10     ]"
11 )
```

Listing 4.6: A Godot function that uses a POST HTTP request to receive information about a tree. This function is almost identical to the function to cut a tree.

4.3.4 Usage & evaluation

The game works and can be used as intended and with the more complex tree simulation, even effects like light competition can be seen. Nevertheless, the selection of the trees is not that smooth and on the more complex model, the updating of the scene can take some time. Yet these limitations are not connected to the GroLink project, but to the limitation of the obj-based model transmission. The fact that every geometrical object is deconstructed into a collection of points and meshes, transferred to the client and reconstructed there is suboptimal. This could be solved with a more suitable 3D format, which also might be able to handle the selection of the trees.

However, regarding the GroLink project, this use case shows that it is possible to use the API included in another project without any client library and that the manipulation of running simulations without recompiling works well with the use of XL queries. Moreover, a Jupyter notebook with the GroPy library was used to access the same running simulation and to apply XL queries for statistical visualization and rule-based tree management.

4.3.5 Perspective

As mentioned in the beginning, the approach is mostly interesting for visualization or education. Yet with the ability to fully interact with the simulation, more complex scenarios would be possible like presenting an interactive greenhouse or teaching students about forest stands and their treatment.

Besides the direct approach, two functionalities were shown in this use case, first, the parameter-based model interaction using complex XL queries and second the usage of GroIMP as a 3D model provider.

The parameter-based interaction could be used to integrate new types of input (for instance sensors) into a GroIMP model. The usage as a 3D model provider could be useful for any type of software that requires realistic plant models for tasks like city planning or garden design.

4.4 Parallelized sensitivity analysis

The ability of GroLink to handle multiple workbenches in a nonblocking way, combined with the possibility of the automatized setting of parameters, clears the way for parallelized sensitivity analysis. This was done by Marek Jäger for the GroIMP version of the Lignum model [44],<> [45], [46], using the GroLink project and the GroPy library. Therefore only the basic workflow will be introduced at this point and for further information the work of Marek Jäger: “Parameter fitting and sensitivity analysis in two plant models based on measurements” [44] should be considered.

4.4.1 Basic workflow

The individual steps for each set of parameters of the analysis can be seen in figure 4.10, starting with the creation of a workbench and updating the parameters.

Currently, the easiest way of changing a large number of variables in a GroIMP model using GroLink, is by storing the variables in a source file that then can be updated. This source file can be directly an RGG file or any parsable file like CSV or XLS. The second approach uses the newly created getInputStreamFromProject library function which was explained in the implementation (sec. 2.5.3).

In this way, it is possible to open the project and define a set of parameters using GroLink.

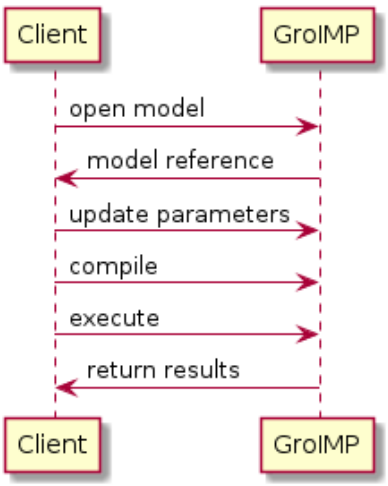


Figure 4.10: A UML sequence diagram showing the paramterised execution of a GroIMP model.

Afterward, the model can be executed and the results can be received either by reading the console or by using datasets (Fig. 4.10). This by itself could already be used for a very simple iterative approach of a sensitivity analysis, by just looping over a set of parameters and storing the results of each iteration.

Yet in most cases, the hardware executing the simulation would be capable of executing it multiple times, therefore a parallelized approach can be considered. For that, it is important to keep in mind that GroIMP does not include any limitation for computation or balancing, meaning that it will try to execute every incoming call at the same time regardless of the available resources. Therefore the client must define the amount of executions that are supposed to be parallelized. This number of possible parallelized executions is fully dependent on the hardware and the simulation.

In theory for an analysis of m parameter sets and n possible parallel executions the basic steps would be to create n workbenches of the model. These workbenches combined with the first n parameter sets are pushed to a parallelized pipeline of updating compiling and running. Afterwards, all results are collected and the next n parameters are matched to the workbenches. A very simple example showing these basic steps can be found in listing 4.7.

```
1  import multiprocessing
2
3  def update_and_run(val): # val represents a tuple of the parameter set and the workbench
4      val[1].updateFile("Model.rgg","""
5        public static float x ="""+str(val[0][0])+""";
6        public static float y ="""+str(val[0][1])+""";
7
8        public void calc(){
9          float result = x **y;
10         println(result);
11     }
12     """).run()
13     val[1].compile().run().read()
14     data =  val[1].runRGGFunction("calc").run().read()['console'][0]
15     return data
16
17
18
19 if __name__ == '__main__':
20     inp = [[i ,j] for j in range(1,6) for i in range(1,6)] #create paramter sets
21     wbs = [link.createWB().run().read() for i in range(0,5)] #create 5 new RGG
           workbenches
22     for i in range(0,9):
23         inp_batch = list(zip(inp[i*5:(i+1)*5],wbs))  #combining 5 parameter sets with
               the 5 workbenches
24         pool = multiprocessing.Pool(processes=5) # creating a pool of 5 parallel
               processes
25         res = pool.map(update_and_run,inp_batch) # executing the pool of processes with
               the tuple of parameter set and workbench
26         results=results + res
27     print(results)
28     for wb in wbs:
29         wb.close().run()
```

Listing 4.7: A GroPy example implementation of the parallel data generation for a sensitivity analysis. In this example the "simulation result" is the first input parameter to the power of the second input parameter (line 9). The script starts with the creation of 25 parameter sets and 5 workbenches in lines 20 and 21. Then the input sets are processed in batches of 5 (exemplaril assumed to be the number of possible parallel executions) by matching them with the workbenches. The multiprocessing library can then execute the update and run function (lines 22-25) on these input sets in parallel. This function then calls the API server to change the values and execute the calc function (lines 4-14). When all 5 workbenches are done the results of the batch are stored. After the last batch all workbenches are closed (lines 28,29).

4.4.2 Evaluation

As shown in the work of Marek Jäger [44], it is possible to use this approach in a real simulation with several hunderd parameter sets. Moreover, the possibility of using Python or R for such a task is quite promising since these languages are well-equipped for sensitivity analysis.

Yet the fact that GroIMP does not handle any execution limits makes the usage a more advanced task. Therefore regarding the future integration of this technique in GroIMP, the main discussion should be about the design of some kind of load balancer across running workbenches that limits the number of executions and stops GroIMP from failing. Besides that, more advanced handling of data sources as an extension to datasets would be very useful, especially in combination with the ability to clone workbenches at any state of the simulation without recompiling. This would enable to only parallelize the part of the simulation that depends on the parameters.

4.4.3 Perspective

Future usage will show the potential of this approach in different types of analysis. To simplify that a first step could be the usage of existing sensitivity analysis libraries such as SALib [47]. Moreover, the distribution of computation on a cluster would be an interesting step to improve the parallelized approach.

4.5 Interactive web interfaces for RGG-based GroIMP models

The graphical user interface of GroIMP with all its functionalities and options, can be too complex for some scenarios. It might be of interest to create simpler user interfaces that highlight the functionalities of a specific model, especially with a focus on publishing or presenting an interactive result. This and the work of Benjamin Spehle on a GroIMP web interface in RShiny [17] (sec. 1.3.3), inspired the following use case.

4.5.1 Description

This use case contains two parts, a generalized viewer for GroIMP models and a specified interface for the interaction with one model. The generalized viewer shows the 3D scene and the project graph, interacts with the RGG functions of the selected model and enables the user to apply XL queries.

The second interface handles a specific simulation of a simplified plant growing toward light. The idea is that the user can trigger growth steps and manipulate the power of the light to see the changes in the growth direction of the plant. To do this, the plant is placed between four light sources of equal power, and with each growth step, it produces four leaves, each pointing towards one of the sources. It is then possible to rotate the x and y axis of new internodes based on the light absorption of the different leaves. The plant can be seen in the result in figure 4.16. The simulation

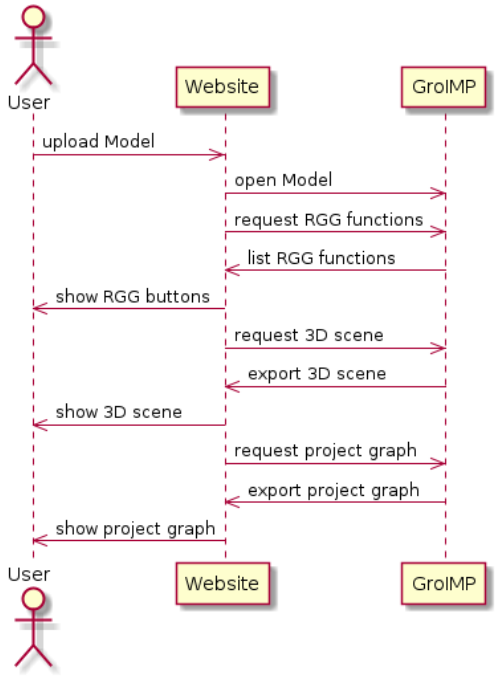


Figure 4.11: A UML sequence diagram showing the conceptional loading and initializing of the general model viewer

is visualized in 3D as well as in a plot showing the light absorption of the leaves separated by the different directions.

4.5.2 Concept

Initializing the generalized viewer

The generalized viewer is a website where a user can upload a model which then is forwarded to GroIMP. After the model is initialized the website can request and show the available RGG functions, the current 3D view and the project graph based on APICalls (Fig. 4.11).

Running RGG and XL

The RGG functions are displayed as buttons linked to the name of the function which then can be used to call the API to execute this function. After an RGG function is executed, the 3D view and the graph view are replaced with newly exported versions. Moreover, the current content of the XLConsole is displayed (Fig. 4.12). The same steps are done after an XL query is executed.

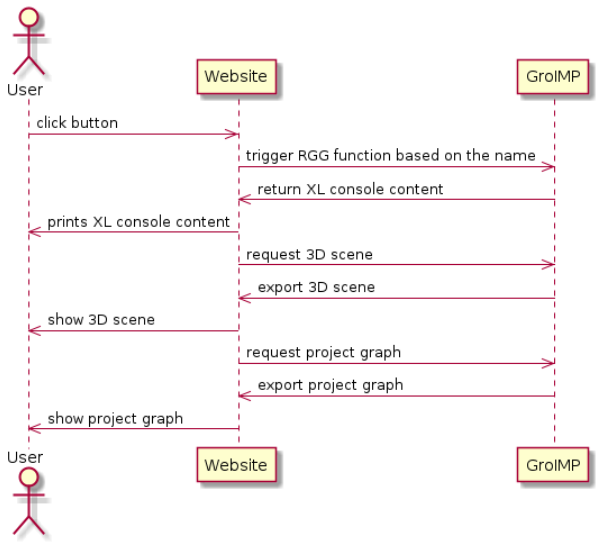


Figure 4.12: A UML sequence diagram showing how the website should handle RGG input. This will be similar for XL queries.

Initializing the specific viewer

For the specific user interface, the website can be much more static because the available RGG function will not change. Moreover, the model is not selected by the user and can be initialized directly (Fig. 4.13).

Additionally to the buttons for growth and reset and the 3D view, a plot of the light is added based on a dataset received from the model. The growth button can be based on an RGG function and would work similarly to the sequence in figure 4.12, with the addition that the growth function should write the absorbed light in a dataset that then later can be read and plotted.

Manipulating the light of the specific viewer

To manipulate the power of the light an input field for each source must be provided as well as a way to update according to this input. This updating can be done with XL queries and does not need any more updates on the viewer since it does not visibly change the model (Fig. 4.14).

4.5.3 Implementation

The implementation is based on RShiny, an extension for the R programming language that enables the creation of web applications with a focus on data visualization [33]. Since it is an R extension the GroR library was used for the connection to the GroLink server. Additionally, the visNetwork [48] library was used to visualize the project graph, ggplot2 [49] for the plotting of the light absorption and the javascript X3D library X3Dom [50] for the visualization of the 3D scene.

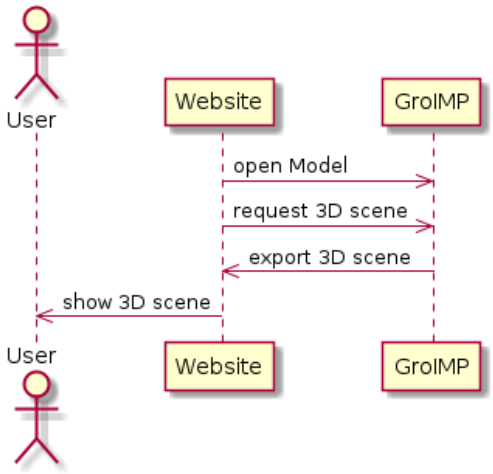


Figure 4.13: A UML sequence diagram showing the conceptional loading and initializing of the specific model viewer. This sequence starts directly by loading the website without user interaction since the model is preselected.

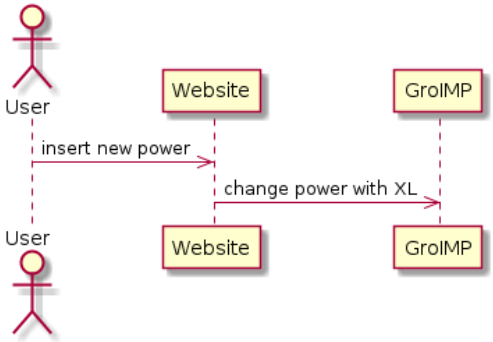


Figure 4.14: A UML sequence diagram showing the steps to change the power of a light source using an XL query.

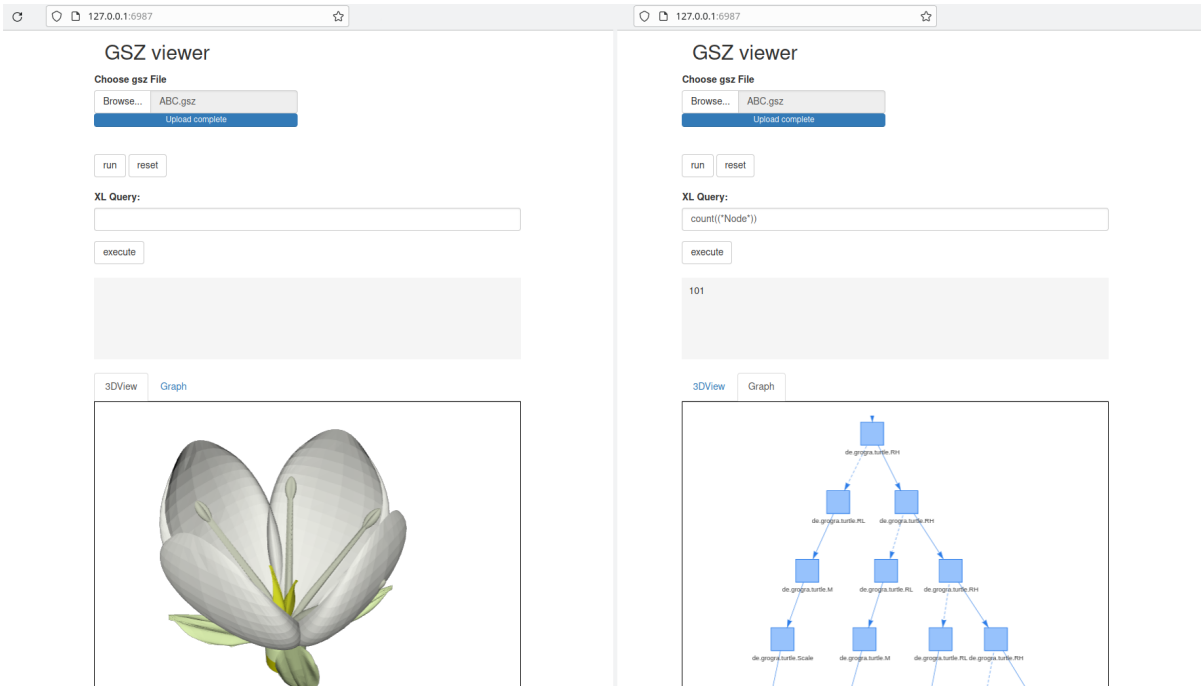


Figure 4.15: The general GSZ viewer with the ABC.gsz model from the GroIMP examples, left with the 3D view and right with the zoomed-in graph visualization and the results of a simple XL query.

Both web applications follow the basic structure of RShiny which contains a data structure representing the user interface and one function that is executed by every interaction with an instance of the web application. This function uses input, output and session as parameters that handle the input of the user, the possible output and the current session. A session in RShiny is defined as one opened instance of the web application and ends by either leaving or reloading the page. The session can store variables that are only defined in that specific session [51].

General viewer

As shown in figure 4.15, it is possible to upload GSZ files to the web application, this upload triggers a function that checks if there is already a workbench opened in this session and if so close it before reading the selected file as binary content and opening it in a new workbench which then is stored in the session. Using this newly created workbench it is now possible to generate buttons for each RGG function (in the case of the ABC model in figure 4.15 this is “run”). To dynamically change the web application in that way the javascript binding of RShiny is used.

To visualize the 3D scene an exported X3D representation is combined with the HTML header of x3dom and placed in a shiny HTMLOutput object. For the project graph, the already existing RShiny binding of visNetwork was used with a fitted version of the JSON project graph.

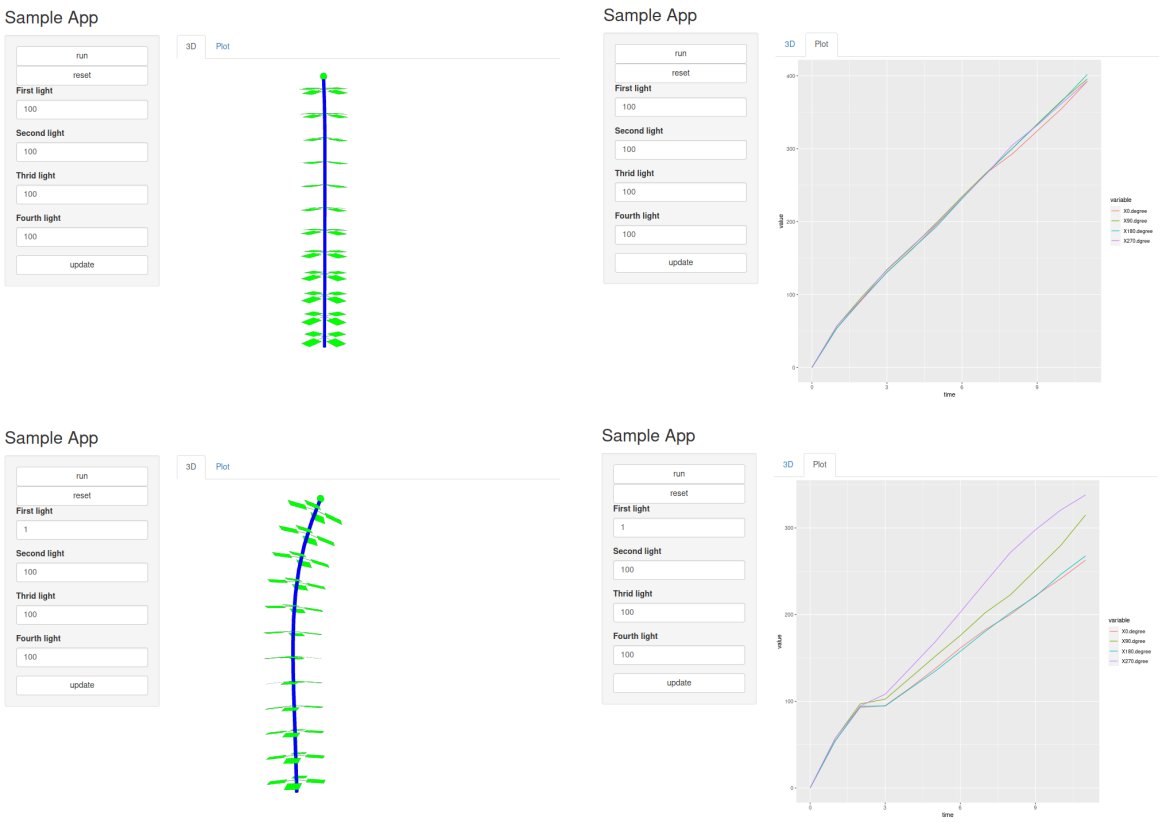


Figure 4.16: The Shiny app of the plant growing towards the light. Shown in two scenarios: without changing the light(upper row) and with changes in the light (lower row). With the 3D view (left) and the plot of the dataset (right).

After the model is initialized, the buttons can be pressed which then triggers GroR to send an APICall to GroIMP to execute the RGG function linked to the button. Afterwards, the 3D model and the graph view will be recreated. Sadly these recreations do not keep the old viewport (zoom, position…) meaning the view jumps always back to the initial position. The content of the XLConsole that might be created within the executed RGG function is printed in the gray box (Fig. 4.15). The execution of an XL query entered in the text field triggers the same update of the web application. The XL queries can be used to gain information as shown in the right part of figure 4.15 or to execute rewriting rules that manipulate the opened model.

Leaving the web application or opening a new project closes the workbench, saving the changes would be possible but is not implemented.

Specific model

The initialization of the web application targeting the light following growth model is much simpler since the model and the usage are clearly defined and the interaction (RGG /XL) can be hard coded (Fig. 4.16). Therefore the web application just lets the API create a workbench for each session and add the initial version of the 3D model using the same X3D-based technique as described above.

The run button triggers GroR to execute the growth function of the model and afterwards updates the 3D view. Additionally, it reads the dataset of the model and turns it into a ggplot graph showing the light absorption of the leaves over time (Fig. 4.16). The four text fields and the update button are used to change the power of the different light sources, by executing XL rewriting rules for each of them. This changes the light without resetting the simulation which allows users to change the light during the growth. Therefore it can be seen how the plant reacts to the change and how the light increase changes in the diagram.

4.5.4 Evaluation

The combination of RShiny and GroR makes it quite simple to create basic web applications that can interact with GroIMP models. It is possible to gain (visual) information about a model, to visualize data created during the simulation and to interact with the running model.

The parts that created issues during the testing of this approach, were mainly on the 3D view due to some issues within the handling of X3D and on the performance of the graph viewer on large models. In addition, visNetwork's ability to select nodes makes the lack of functionality for reading and writing node properties very clear.

One important fact to mention here is that the upload of custom models as well as the direct access to the XL filter through text input, allows users to execute custom Java code on the server. This is one big feature of GroIMP which is very handy in many situations but could become a security issue on the internet. Yet if a web application only works with a predefined model and predefined XL queries this can be avoided.

4.5.5 Perspective

For the presentation of a specific model to a selected audience this use case holds a lot of potential. A similar approach could be used to create a web application with restricted access over the internet using GroIMP and RShiny on a server managed by a fully equipped web server like Nginx or Apache. This fully equipped web server then can be used to allow authenticated users access to the RShiny web application but not GroLink directly (simplified in figure 4.17). This would allow users to access the web application and the web application to interact with the API but not the user to interact with the API.

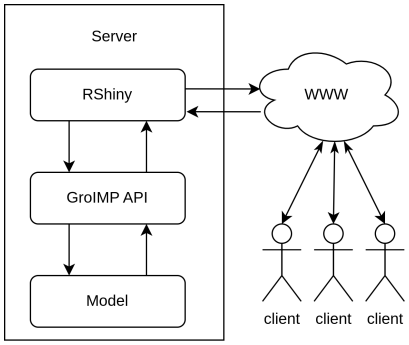


Figure 4.17: A structure that could be used to present an interactive GroIMP model to an online community.

Chapter 5

Discussion & Prospects

5.1 Reviewing the outline

In the introduction, an outline (sec. 1.4) for the project was created to clarify the requirements of a GroIMP API server. In the following, this outline will be reviewed to evaluate the implementation of the GroLink project and the use cases.

5.1.1 Areas of application

The areas of application were introduced to point out possible scenarios where a project like GroLink could be used. In the following, GroLink-based approaches for these areas are discussed based on the use cases and examples created.

GroIMP as the source of 3D models

The ability to export the 3D view of a model is implemented in the GroLink server and the client libraries and was used in three use cases. Moreover, it is possible to first manipulate the model with RGG functions or XL queries. Therefore it can be said that this area is basically fulfilled by GroLink. Nevertheless, two major limitations must be pointed out. First, it is not possible to export fractions of the 3D scene, this was a limitation for the XEG processing pipeline (sec. 4.2) where the light was not supposed to be exported. Moreover, it is not possible to properly interact with the elements in the exported 3D scene, therefore in the foresty game (sec. 4.3) a costly workaround using locations and distances was necessary. Additionally, for specific usage of some formats, the file system independent export is not fully supported. For example, if a model using image shaders is exported as X3D to the HTTP body, the image shaders are not transferred and will therefore be missing.

GroIMP as a file converter

The XEG processing pipeline (sec. 4.2) shows an example of not directly a file converter but a file manipulator. Similarly, it would be possible to export a project with an imported graph structure into a different format. While this should work out of the box for most conversions, some issues need to be addressed. First, as mentioned above, the export of a subset of the model is not supported by GroLink. Moreover, only the file extension is used to determine the import and export filters, therefore for formats like ply which are used for point clouds as well as for 3D models, the selection of the right filter would not be possible.

GroIMP as an interpreter of FSPM models/commands

The interpretation of models and the usage of specific user interfaces as described in this area, was shown in the two web applications of the use case "Interactive web interfaces for RGG-based GroIMP models" (sec. 4.5). The use case shows that it is possible with GroLink to receive and execute RGG commands and XL queries/rewriting rules, to load data from datasets and to access and display the project graph. Nevertheless, the same use case highlights that it is not yet possible to edit the attributes of a node of the project graph or the project graph directly.

GroIMP as an external service for manipulating models

The steps explained in this area were in different parts implemented in the XEG processing pipeline (sec. 4.2) and in the simplified example of a sensitivity analysis (Lst. 4.7). Both together show that it is possible to create a new RGG environment with new functions, to add graph structures from files and to process them before exporting them again. The same steps were shown in simple examples for GroPy (sec. 3.4.3) and GroR (sec. 3.5.3). The limitations of the XEG processing named above (sec. 5.1.1) must also be considered as general limits for this approach.

Orchestration of multiple simulations from an external script

This area was mostly explored in the thesis of Marek Jäger [44] and is therefore only directly introduced in the small example in section 4.4. Nevertheless, the other use cases and examples show that opening and addressing workbenches works very well. Yet an unexpected issue on larger scales is that closing many workbenches at the same time is quite time-consuming due to bad access management by the project manager.

Regarding parallel execution, the workbench and thread management work well and can for some simulations be a performance improvement. Nevertheless, the fact that the project comes without proper cross-workbench resource management makes the usage much more challenging for users than necessary.

To enable the manipulation of configuration files the getInputStreamFromProject function was created. But although this works well for instance with the .fun files from the Lignum model,

this approach is limited by the fact that the project has to be compiled (and therefore reset) after changing such files.

5.1.2 Requirements & needed functionality

Based on the areas of application, table 1.1 was created to summarise the required functionalities. Comparing this table with the GroLink HTTP commands in the appendix A shows that only the node-attribute related functions and the remove node function are not implemented. Yet it has to be mentioned that resetting a model is in GroLink done by recompiling which is not the same way as in the GUI. Moreover, although, the table of needed functionalities is almost covered, it is important to keep in mind that this table lists only a fraction of the possible functions of GroIMP. For instance, the management of shaders, curves or preferences is not included.

Regarding the requirements on the API server (sec. 1.4.2), GroLink can run without a GUI and in a container environment, which was tested with all use cases. Moreover as shown in section 2.6.1 it is possible to add new functionalities with additional plugins without changing the original plugin. Yet the new functions must be designed specifically for the use of GroLink which is a limitation that might been avoidable with proper toolkit management.

The ability to execute several commands on different workbenches in a nonblocking way was already discussed above in the revie about the orchestration of multiple simulations.

Finally, the idea to open a project with two GroIMP applications at the same time is considered in the design, for instance by the separation of workbench and project, yet it was not implemented in this work. It turned out to be much more complex than expected. Moreover, the way the workbench and project were split needs to be discussed in future implementations. It might not be suitable to have two registries but rather have a project registry that is interpreted by the workbench.

5.2 Theoretical application on related work

In the beginning, four projects were introduced to show possible applications of a general API. In the following, it will be discussed if and how GroLink could have been used in these projects.

5.2.1 Funktions- / Strukturorientierte Pflanzenmodellierung in E-Learning-Szenarien

The GroIMP model of Dirk Lanwert's work [11] handles not only the simulation but also the interaction using the GroIMP HTTP server and the automated startup function. This second part could be replaced by GroLink without any bigger changes to the simulation itself. Moreover, it would be possible to hold a connection to the simulation and therefore interact with it. The use

case “Using GroIMP as a backend for a forestry game” (sec: 4.3) follows a quite similar structure, yet with a much simpler simulation and with the generation of the 3D scene in GroIMP.

5.2.2 The integration of different functional and structural plant models

The work of Qinqin Long [13] was one of the main inspirations for this work as well as for the use case “An XEG processing pipeline” (sec. 4.2). Therefore the basic workflow on the GroIMP side can be seen in that use case and the limitations were already discussed above. Yet some of these limitations were also part of the original approach since they are partly rooted in the current implementation of the XEG interface.

Besides the possibility of linking the original functionalities to OpenAlea, it could also be possible to link all GroLink functions to OpenAlea, enabling even more complex co-simulation. Nevertheless, first, a real link between OpenAlea and GroLink must be created to prove the, until now only theoretical, approach.

5.2.3 Conveying the Effects of Climate Change - Interfacing virtual Riesling with an interactive R shiny application

This project by Benjamin Spehle [17] used a quite common workaround for integrating GroIMP based on configuration files and the headless mode. As shown in the use case “Interactive web interfaces for RGG-based GroIMP models” (sec. 4.5) this could be avoided in most cases using GroLink

5.2.4 A framework for a Digital Twin of a semi-closed greenhouse using the FSPM platform GroIMP

This project could directly benefit from GroLink in two ways, first the R and Rmarkdown-based report pipeline could be recreated using the GroR library, avoiding several workarounds of the original implementation. Moreover, with the manipulation of the simulation through XL queries the GroLink project could be used to update model parameters during a running simulation. And, in the other direction, it would also be possible to receive simulation results as direct responses. This could result in the kind of exchange a digital twin is aiming for.

Besides these approaches, it would also be possible to explore other directions, such as the creation of a user interface similar to the software used to manage the physical twin with the addition of a way to place and analyze plants. Or an optimization toolkit to find simulation parameters that create the same results as measured in the physical twin.

Since these ideas burst the frame of this work was only a small proof of concept regarding the interaction of GroLink and the semi-closed greenhouse was created. The results can be found in appendix D as well as in interactive form in the attachments.

5.3 Discussion

After reviewing the outline and the associated projects, it is clear that although most of the outline issues have been achieved and the example projects could benefit from the current version, the project is still in its infancy. On all three main levels, the generalized foundation for GroIMP applications, the server and the client libraries, improvements would be either necessary or beneficial.

Regarding the new generalized foundation, it is mostly about the split between the workbench and the project as mentioned above and if the GUI should be following the same structure, some general rework and improvement.

The GroLink server plugin would greatly benefit from the use of a proper toolkit to open the project to GroIMP's full potential. This rework could be combined with some reworking of command names and parameter keys to follow a proper naming convention rather than a grown structure. Moreover, this rework may include a proper starting page listing all available commands and their description, similar to the main page of the GroIMP HTTP server.

Another subject, that is placed somewhere between the generalized foundation and the server, is the cross-workbench resource management which was already mentioned and should be addressed properly.

Finally, for the client library, the main concern regarding the general approach is that it might have been overly complex to define the calls as individual objects that can be recalled. As shown in GroR it can work well with a simpler approach of just functions directly addressing the API server. Yet, while this works well in GroR, the R library is at several parts quite too simplistic for instance regarding the handling of binary files and JSON objects. With proper knowledge of the R programming language, this could be improved.

5.4 Prospects

As mentioned above this work is only the first step and an initial version of the GroLink project. Several improvements and possibilities were already pointed out during this work. In the following, only the main possible paths for the future will be introduced.

Regarding more general possibilities, the most interesting, but difficult, step could be to move the GUI application to the new generalized foundation. This would not only simplify the code structure in GroIMP it would also improve the possibilities of future interaction of the different GroIMP applications including access to the same project from multiple workbenches. Based on the new generalized foundation it also is easily possible to create more GroIMP applications for specific use cases such as user interfaces or automatization approaches.

Additionally as mentioned in the discussion a proper solution for managing the computation

resources among multiple workbenches should be considered to improve the abilities of parallel execution. Moreover, the tests on automatization and orchestration pointed out that cloning workbenches including the current state of the simulation could be a great improvement.

Regarding the GroLink plugin itself, the next important step will be the creation of an APIToolkit that is able to transfer GroIMP panels into JSON responses and can interpret user input on a generalized level. This could enable the API to execute the same functions as the GUI and would therefore enable access to more GUI functionalities and bring GroLink a step closer to accessing all of GroIMP's functionalities. In the long term, this could enable the creation of a fully functional web interface for GroIMP and with that the possibilities of a cloud-based modelling platform.

Some other possibilities for client applications using GroLink were already introduced with the use cases. Each of them could be extended to a full project or be integrated into existing work. In particular, the connection to OpenAlea and the abilities for sensitivity analysis hold great potential. But also the usage of the API as a small part of automated projects holds countless possibilities. With the creation of new projects, the creation of new client libraries could be considered as well, for instance in programming languages like JavaScript or Julia.

References

[1] O. Kniemeyer, "Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling," Georg-August-University Göttingen, 2008.

[2] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. in The Virtual Laboratory. New York, NY: Springer New York, 1990. doi: 10.1007/978-1-4613-8476-2.

[3] D. Gourley, B. Totty, and M. Sayer, *HTTP: The definitive guide*, 1. ed. in Understanding web internals. Beijing Köln: O'Reilly, 2002.

[4] "HTTP response status codes - HTTP | MDN." Accessed: Oct. 18, 2023. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`

[5] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API Property Inference Techniques," *IIEEE Trans. Software Eng.*, vol. 39, no. 5, pp. 613–637, May 2013, doi: 10.1109/TSE.2012.63.

[6] "Simple Object Access Protocol (SOAP) 1.1." Accessed: Aug. 30, 2023. [Online]. Available: `https://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383487`

[7] J. Tihomirovs and J. Grabis, "Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics," *Information Technology and Management Science*, vol. 19, no. 1, Jan. 2016, doi: 10.1515/itms-2016-0017.

[8] "What is a REST API?" Accessed: Aug. 30, 2023. [Online]. Available: `https://www.redhat.com/en/topics/api/what-is-a-rest-api`

[9] V. Stavridou, "Integration in software intensive systems," *Journal of Systems and Software*, vol. 48, no. 2, pp. 91–104, Oct. 1999, doi: 10.1016/S0164-1212(99)00049-7.

[10] A. I. Wasserman, "Tool integration in software engineering environments," in *Software Engineering Environments*, vol. 467, F. Long, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 137–149. doi: 10.1007/3-540-53452-0_38.

[11] D. Lanwert, "Funktions- / Strukturorientierte Pflanzenmodellierung in E-Learning-Szenarien," Georg-August-University Göttingen, 2007.

[12] Q. Long, "The integration of different functional and structural plant models," Georg-August-University Göttingen, 2019. doi: 10.53846/goediss-7554.

[13] Q. Long, C. Pradal, and W. Kurth, "Co-simulation with OpenAlea and GroIMP for cross-platform functional-structural plant modelling." Accessed: Jul. 31, 2023. Available: https://inria.hal.science/hal-03059527/file/abstract8.pdf

[14] Q. Long, W. Kurth, C. Pradal, V. Migault, and B. Pallas, "An Architecture for the Integration of Different Functional and Structural Plant Models," in *Proceedings of the 7th International Conference on Informatics, Environment, Energy and Applications*, Beijing China: ACM, Mar. 2018, pp. 107–113. doi: 10.1145/3208854.3208875.

[15] C. Pradal, S. Dufour-Kowalski, F. Boudon, C. Fournier, and C. Godin, "OpenAlea: A visual programming and component-based software platform for plant modelling," *Functional Plant Biol.*, vol. 35, no. 10, p. 751, 2008, doi: 10.1071/FP08084.

[16] "3. MTG file — openalea.mtg 2.1.2 documentation." Accessed: Oct. 19, 2023. [Online]. Available: https://mtg.readthedocs.io/en/latest/user/intro.html#mtg-file

[17] B. Spehle, "Conveying the Effects of Climate Change," in *Book of Abstracts - fspm23*, p. 151. Accessed: Nov. 01, 2023. Available: https://www.fspm2023.net/_files/ugd/38ef44_04f09eb2900942d8bb4c3d24d3a7a39a.pdf

[18] T. Oberländer, "A framework for a Digital Twin of a semiclosed greenhouse using the FSPM platform GroIMP," Georg-August-University Göttingen, 2023.

[19] "R Markdown." Accessed: May 18, 2023. [Online]. Available: https://rmarkdown.rstudio.com/

[20] "GroIMP v 2.0 · Grogra / GroIMP · GitLab." Accessed: Aug. 31, 2023. [Online]. Available: https://gitlab.com/grogra/groimp/-/releases/GroIMP_2_0

[21] "Grogra / GroIMP plugins / CLI · GitLab." Accessed: Oct. 25, 2023. [Online]. Available: https://gitlab.com/grogra/groimp-plugins/cli

[22] "JLine." Accessed: Oct. 25, 2023. [Online]. Available: https://github.com/jline/jline3

[23] "Nano – Text editor." Accessed: Oct. 25, 2023. [Online]. Available: https://www.nano-editor.org/

[24] "Most used languages among software developers globally 2023." Accessed: Oct. 12, 2023. [Online]. Available: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

[25] "OpenAlea community — OpenAlea community website." Accessed: Oct. 12, 2023. [Online]. Available: http://virtualplants.github.io/

[26] "Requests: HTTP for Humans™ — Requests 2.31.0 documentation." Accessed: Oct. 12, 2023. [Online]. Available: https://requests.readthedocs.io/en/latest/

[27] "Project Summaries — Python Packaging User Guide." Accessed: Oct. 12, 2023. [Online]. Available: https://packaging.python.org/en/latest/key_projects/#pip

[28] "CI/CD pipelines | GitLab." Accessed: May 18, 2023. [Online]. Available: https://docs.gitlab.com/ee/ci/pipelines/

[29] "Packaging Python Projects — Python Packaging User Guide." Accessed: Oct. 12, 2023. [Online]. Available: https://packaging.python.org/en/latest/tutorials/packaging-projects/

[30] M. Tichenor, "How To Upload Private Python Packages to Gitlab." Accessed: Oct. 12, 2023. [Online]. Available: https://medium.com/@matt_tich/how-to-upload-private-python-packages-to-gitlab-2999e9604603

[31] "Project Summaries — Python Packaging User Guide." Accessed: Oct. 12, 2023. [Online]. Available: https://packaging.python.org/en/latest/key_projects/#build

[32] "Twine 4.0.2 documentation." Accessed: Oct. 12, 2023. [Online]. Available: https://twine.readthedocs.io/en/stable/

[33] "Shiny." Accessed: Oct. 12, 2023. [Online]. Available: https://shiny.posit.co/

[34] F. M. Giorgi, C. Ceraolo, and D. Mercatelli, "The R Language: An Engine for Bioinformatics and Data Science," *Life*, vol. 12, no. 5, p. 648, Apr. 2022, doi: 10.3390/life12050648.

[35] "Httr2." Accessed: Oct. 12, 2023. [Online]. Available: https://httr2.r-lib.org/articles/httr2.html

[36] "A Grammar of Data Manipulation." Accessed: Oct. 12, 2023. [Online]. Available: https://dplyr.tidyverse.org/

[37] "GitHub - openalea-incubator/groalea: Interoperability between GroIMP and OpenAlea." Accessed: Oct. 16, 2023. [Online]. Available: https://github.com/openalea-incubator/groalea

[38] "SIBYLA Triquetra." Accessed: Nov. 17, 2023. [Online]. Available: https://sibyla.tuzvo.sk/index.html

[39] G. Engine, "Godot Engine - Free and open source 2D and 3D game engine." Accessed: Oct. 12, 2023. [Online]. Available: https://godotengine.org/

[40] T. Oberländer, "A parametric functional-structural tree model based on graph rewriting," Georg-August-University Göttingen, 2020.

[41] Dylan, "Gd-obj." Accessed: Oct. 12, 2023. [Online]. Available: https://github.com/Ezcha/gd-obj

[42] A. Alarie, "Game wood panel." Accessed: Nov. 07, 2023. [Online]. Available: https://opengameart.org/content/game-wood-panel

[43] LuminousDragonGames, "Blended textures of dirt and grass." Accessed: Nov. 07, 2023. [Online]. Available: https://opengameart.org/content/blended-textures-of-dirt-and-grass

[44] M. Jäger, "Parameter fitting and sensitivity analysis in two plant models based on measurements," Georg-August-University Göttingen, 2023.

[45] J. Perttunen, "LIGNUM: A Tree Model Based on Simple Structural Units," *Annals of Botany*, vol. 77, no. 1, pp. 87–98, Jan. 1996, doi: 10.1006/anbo.1996.0011.

[46] K. Smoleňová, "Rule-based integration of LIGNUM into GroIMP | FSPM2013 Proceedings," Accessed: Nov. 24, 2023. Available: https://ojs.silvafennica.fi/index.php/fspm2013/article/view/850

[47] J. Herman and W. Usher, "SALib: An open-source Python library for Sensitivity Analysis," *JOSS*, vol. 2, no. 9, p. 97, Jan. 2017, doi: 10.21105/joss.00097.

[48] datastorm, "visNetwork." Accessed: Oct. 13, 2023. [Online]. Available: https://datastorm-open.github.io/visNetwork/

[49] "Create Elegant Data Visualisations Using the Grammar of Graphics." Accessed: Oct. 13, 2023. [Online]. Available: https://ggplot2.tidyverse.org/

[50] J. Behr, P. Eschler, Y. Jung, and M. Zöllner, "X3DOM: A DOM-based HTML5/X3D integration model," in *Proceedings of the 14th International Conference on 3D Web Technology*, Darmstadt Germany: ACM, Jun. 2009, pp. 127–135. doi: 10.1145/1559764.1559784.

[51] "Session object — session." Accessed: Oct. 14, 2023. [Online]. Available: https://shiny.posit.co/r/reference/shiny/1.7.2/session.html

[52] "Project Jupyter." Accessed: Nov. 27, 2023. [Online]. Available: https://jupyter.org

Glossary

client A software or hardware tool used to access a service provided by a server. For example, an email client like Thunderbird accesses the service provided by the email server. 4, 5, 6, 7, 10, 12, 16, 17, 18, 19, 21, 29, 30, 37, 38, 39, 45, 46, 47, 48, 49, 51, 54, 59, 60, 63, 67, 68, 71, 72, 73, 83, 87, 88

container A container describes an executable software unit that includes the needed dependencies. It is a light but less independent form of software virtualization. A well known implementation of this concept is Docker. 12, 33, 45, 54, 85

containerized A software and its dependencies installed in a container. 32, 33, 59

factory In software design, a factory is a creational pattern for objects without knowing the exact class. In this way, a function can create different types of objects based on the different factories (following the same structure) without any knowledge about the class of the object. In GroIMP such factories are stored in the registry. 4, 23, 24, 25, 67

filter A filter is used to process an incoming stream in order to create information usable for the software. 3, 23, 41, 42, 67, 81, 84

foundation The new generalized foundation for GroIMP applications is a set of interfaces and base structures that are intended to generalize the way applications interact with the GroIMP platform. This aims to simplify the creation and improfment of application. 13, 15, 20, 27, 87

GroIMP application In GroIMP an application describes the function executed after the boot process. This is used to start for instance the GUI. 13, 20, 23, 27, 29, 85, 87

GroLink function A collection of functions in the GroR library similar to the idea of the GroLink class. 54

GroLink object A GroLink object in the client library is an instance of the GroLink class. 49, 50

GroLink class The GroLink class in the client library describes the connection to the GroLink API server and is used to send app calls such as open or create a workbench. 47, 48, 93

headless see headless mode. 2, 4, 7, 8, 9, 12

headless mode A software running in a headless mode is not using any (graphical) user interface or any kind of input of a mouse or keyboard. 3, 7, 10, 86, 94

hook An extendable collection of commands that is executed on a certain event in GroIMP. Similar to directories it is possible to add elements to existing hooks. 2, 16, 39, 40

JobManager A JobManager in GroIMP receives commands and organizes their execution and synchronisation with the project. If no specification is given the commands are executed in order of their arrival. 2, 4, 15, 17, 18, 21, 25, 29, 30

orchestration Automated management and coordination of multiple software processes. 10, 85, 88

pipeline The concatenation of different applications or processes designed to stepwise process information or tasks. 1, 6, 8, 9, 59, 60, 61, 73, 83, 84, 86

plugin A software package for a specific purpose that can be added to an existing software. 2, 3, 6, 7, 10, 12, 13, 16, 20, 23, 24, 25, 27, 35, 37, 38, 41, 42, 45, 62, 67, 85, 87, 88

registry The registry in GroIMP holds almost all information or links to them and makes them available to the different parts of the software. 2, 3, 4, 10, 15, 16, 18, 20, 23, 24, 35, 39, 41, 45, 49, 85

server A software or hardware tool that provides a service that can be accessed by a client such as a web or email server. 4, 5, 6, 7, 12, 13, 16, 17, 18, 19, 21, 32, 33, 38, 45, 46, 47, 49, 50, 51, 53, 54, 55, 61, 63, 68, 77, 81, 83, 85, 87

Swing One of the most common Java libraries for graphical user interfaces. 4

thread The smallest sequence of instructions that is managed by the operation systems scheduler. Multi-threading allows computers to execute several tasks in parallel. 4, 15, 18, 29, 32, 84

toolkit A widget toolkit is a set of functions used to generate basic UI widgets such as buttons or text fields. 4, 15, 16, 26, 85, 87, 88

virtual see virtualization. 12, 33

virtualization Virtualization is the process of using software layers to create hardware abstract environment. For instance a virtual operation system or a docker container. 93, 94

web application An application that can be used in a webbrowser similar to a website. 7, 9, 77, 79, 80, 81, 84

Acronyms

API Application Programming Interface. 2, 5, 6, 8, 9, 10, 12, 13, 15, 16, 17, 18, 19, 20, 21, 25, 28, 29, 30, 31, 32, 33, 35, 36, 37, 38, 40, 41, 45, 46, 47, 50, 51, 54, 55, 61, 62, 63, 67, 68, 69, 72, 76, 81, 83, 85, 87, 88

CLI Command-line interface. 13, 14, 15, 24, 25, 26, 27, 32, 41, 42

CSV Comma-separated values. 35, 37, 72

GPU Graphics Processing Unit. 2, 59, 60

GroIMP Growth Grammar-related Interactive Modelling Platform. 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 22, 23, 25, 26, 29, 30, 32, 33, 35, 37, 38, 40, 41, 45, 48, 49, 51, 54, 55, 56, 59, 60, 62, 63, 65, 67, 68, 69, 70, 71, 72, 73, 75, 76, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88

GroLink The newly created GroIMP application that provides an API access. 13, 14, 24, 27, 28, 29, 32, 33, 34, 35, 37, 38, 39, 40, 42, 45, 48, 49, 50, 51, 52, 59, 60, 61, 62, 63, 67, 68, 71, 72, 77, 81, 83, 84, 85, 86, 87, 88

GS The file ending of the core file of a GroIMP project. If GSZ is not used this file is selected as the saving location. 54

GSZ The zip compressed version of a saved GroIMP project. 3, 19, 41, 53, 54, 61, 79

GUI Graphical User Interface. 3, 4, 9, 11, 12, 13, 15, 26, 41, 50, 52, 85, 87, 88

HTTP Hypertext Transfer Protocol. 5, 6, 7, 17, 18, 29, 30, 32, 38, 45, 48, 55, 67, 68, 83, 85, 87

JSON JavaScript Object Notation: A file format for transferring and storing information in human-readable text. 6, 30, 31, 38, 46, 47, 48, 56, 79, 87, 88

MTG Multiscale Tree Graph. 7

OBJ Wavefront OBJ: an open format for 3D objects. 11, 67, 68, 69, 70

RGG Relational Growth Grammars. 1, 2, 4, 7, 9, 10, 11, 18, 20, 24, 26, 33, 35, 36, 46, 51, 52, 53, 56, 60, 63, 68, 72, 75, 76, 77, 79, 80, 81, 83, 84, 86

RSML Rootsystem Modelling Language. 11

UI User Interface. 13, 14, 15, 20, 22, 23, 25, 26, 27, 29

URL Uniform Resource Locator. 5, 18, 40, 48, 54, 55

X3D Extensible 3D: An XML-based format for describing 3D scenes with a focus on web presentation. 77, 79, 81, 83

XEG EXchange Graph. 7, 9, 11, 34, 53, 57, 59, 60, 61, 62, 83, 84, 86

XL eXtended L-system language. 1, 9, 10, 11, 12, 32, 35, 51, 52, 56, 63, 65, 70, 72, 75, 76, 77, 80, 81, 83, 84, 86

XLS ExceL Spreadsheet. 35, 72

Appendix A

HTTP commands

The following list contains all commands specifically implemented for the usage of GroLink.

A.1 Application

A.1.1 Create new workbench

This command creates a new project in a new workbench similar to File/new/... in the GUI. The command then returns a workbench ID that is needed for later calls.

The new projects are created based on templates, these templates can be listed with the list templates call.

Path: `app/ui/commands/app/create`

Parameter:

Table A.1

key	description	type	required	default
name	the template that will be used	GET	optional	newRGG
newName	the name of the new project	GET	optional	newRGG

Return values:

Table A.2

name	description	format
id	the id of the new workbench	JSON

Example: `http://localhost:58081/api/app/ui/commands/app/create?name=newEmptyRGG&newName=myTest`

A.1.2 Open project

There are two different ways to open a project, the path-based command is a bit faster due to less encoding and decoding but the path must be accessible from the server. After transferring the file both commands work identically, they open a workbench the same way as in the GUI or the CLI and then return a workbench ID that is needed for later calls.

Path-based

Path: `app/ui/commands/app/open`

Parameter:

Table A.3

key	description	type	required	default
path	absolute path to the project	GET	required	-

Return values:

Table A.4

name	description	format
id	the id of the new workbench	JSON

Example: `http://localhost:58081/api/app/ui/commands/app/open?path=/absolute/path/to/your/project.gs`

Call body based

Path: `app/ui/commands/app/open`

Parameter:

Table A.5

key	description	type	required	default
-	a gsz project as byte code	POST	required	-

Return values:

Table A.6

name	description	format
id	the id of the new workbench	JSON

Example:

`http://localhost:58081/api/app/ui/commands/app/open` + file as request body

A.1.3 List examples

The examples as known from the GUI (File/Show Examples) can be listed and queried by tags. With the first version, this is quite misleading, since the parameter key is "tags" but only takes one argument. Additionally, this `?tags=` with nothing follows will lead to an empty result.

Path: `app/ui/commands/app/listExamples`

Parameter:

Table A.7

key	description	type	required	default
tags	the tag the examples should have	GET	optional	(all examples are shown)

Return values:

Table A.8

name	description	format
data	A list of all examples with title, tag, id and description	JSON

Example: `http://localhost:58081/api/app/ui/commands/app/listExamples?tags=FSPM`

A.1.4 Load example

This command loads an example from GroIMP based on its id, into a new workbench and returns the id of that workbench.

Path: `app/ui/commands/app/loadExample`

Parameter:

Table A.9

key	description	type	required	default
name	the id of the project given in the list of examples	GET	required	-

Return values:

Table A.10

name	description	format
id	the id of the new workbench	JSON

Example: `http://localhost:58081/api/app/ui/commands/app/loadExample?name=ABC`

A.1.5 List templates

GroIMP comes with a list of different templates (that can easily be extended) for creating new projects. This list can be viewed with this command.

Path: `app/ui/commands/app/listExamples`

Parameter: None

Return values:

Table A.11

name	description	format
data	A list of all templates with title, id and description for each	JSON

Example: `http://localhost:58081/api/app/ui/commands/app/listTemplates`

A.1.6 List open workbenches

A command that lists all opened workbenches with title and id. It is important to mention that the workbench with id=0 is the main workbench and should not be used for any commands.

Path: `app/ui/commands/app/listWB`

Parameter: None

Return values:

Table A.12

name	description	format
data	A list of all workbenches with title and id	JSON

Example: `http://localhost:58081/api/app/ui/commands/app/listWB`

A.1.7 Close GroIMP

This command stops the API server and closes GroIMP.

Path: `app/ui/commands/app/close`

Example: `http://localhost:58081/api/app/ui/commands/app/close`

A.2 Workbench

A.2.1 List RGG functions

This command lists all RGG functions by name, this includes the “Run <name>” functions which in the GUI can be used for infinite running. These functions can not be used in the API.

Path: `wb/<id>/ui/commands/listFunctions`

Parameter: None

Return values:

Table A.13

name	description	format
data	A list of all RGG functions of this workbench	JSON
console	for this command empty	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/listFunctions`

A.2.2 Run RGG function

This command runs the RGG function given by the name and returns eventually printed results in the console return value. It does the same as clicking a button on the RGG-Toolbar in the GUI.

Path: `wb/<id>/ui/commands/runRGGFunction`

Parameter:

Table A.14

key	description	type	required	default
name	the name of the function	GET	required	-

Return values:

Table A.15

name	description	format
console	returns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/runRGGFunction?name=run`

A.2.3 Run XL query/rules

This command handles the same XL queries as the XL console in the GUI/CLI does except for the storage of variables. With the command, the graph can be analyzed as well as manipulated.

The command can be executed in GET and POST because some XL queries are in conflict with the standard of a URL, such as `&&`.

GET

Path: `wb/<id>/ui/commands/runXLQuery`

Parameter:

Table A.16

key	description	type	required	default
xl	The query	GET	required	-

Return values:

Table A.17

name	description	format
console	the retuned content of the query	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/runXLQuery?xl=[Node==>F F ;]`

POST

Path: `wb/<id>/ui/commands/runXLQuery`

Parameter:

Table A.18

key	description	type	required	default
-	The query	POST	required	-

Return values:

Table A.19

name	description	format
console	the retuned content of the query	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/runXLQuery` + body

A.2.4 Compile

This command compiles the RGG files and resets the project graph

Path: `wb/<id>/ui/commands/compile`

Parameter: None

Return values:

Table A.20

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/compile`

A.2.5 List source files

This command lists all source files the same way the file explorer does.

Path:

`wb/<id>/ui/commands/getSourceFiles`

Parameter: None

Return values:

Table A.21

name	description	format
data	A list of all source files of this workbench	JSON
console	for this command empty	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/getSourceFiles`

A.2.6 Add source file

There are two different ways to add a source file, the path-based command sends the connection to an existing file but only works when the API server can access that file. The request body-based command can directly transfer the text content of the file as the body of the request.

After adding a file the file is activated which compiles the project.

If GroIMP is not able to compile the added File it returns "message": "pfs:<name>.rgg could not be opened."

Path-based

Path: `wb/<id>/ui/commands/addFile`

Parameter:

Table A.22

key	description	type	required	default
path	the absolute path to the file	GET	required	-

Return values:

Table A.23

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/addFile?path=/path/to/your/file.rgg`

Request body-based

Path: `wb/<id>/ui/commands/addFile`

Parameter:

Table A.24

key	description	type	required	default
name	the name of the new file	GET	required	-
-	content of the file	POST	required	-

Return values:

Table A.25

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/addFile?name=test.rgg` + body with code or data

A.2.7 Rename source file

This command can change the name of a source file.

Path: `wb/<id>/ui/commands/renameFile`

Parameter:

Table A.26

key	description	type	required	default
name	the current name of the source file	GET	required	-
newName	the new name of the source file	GET	required	-

Return values:

Table A.27

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/renameFile?name=Model.rgg&newName=Mooodel.rgg`

A.2.8 Update source file

This command changes the content of a source file.

Path: `wb/<id>/ui/commands/setSourceFileContent`

Parameter:

Table A.28

key	description	type	required	default
name	the name of the file	GET	required	-
-	content of the file	POST	required	-

Return values:

Table A.29

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/setSourceFileContent?name=Model.rgg`
+ body with code or data

A.2.9 Get source file content

This command returns the content of the given source file.

Path: `wb/<id>/ui/commands/getSourceFileContent`

Parameter:

Table A.30

key	description	type	required	default
name	the name of the file	GET	required	-

Return values:

Table A.31

name	description	format
-	the content as raw data	text

Example: `http://localhost:58081/api/wb/1/ui/commands/getSourceFileContent?name=Model.rgg`

A.2.10 Remove source file

This command removes a given source file

Path: `wb/<id>/ui/commands/removeFile`

Parameter:

Table A.32

key	description	type	required	default
name	the name of the file	GET	required	-

Return values:

Table A.33

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/removeFile?name=param/parameters.rgg`

A.2.11 Get project graph

This command returns the project graph of the selected workbench in a JSON format separated into a list of nodes with the id and the type and a list of the edges as vectors of three components (parentID, childID, edgeByte)

Path: `wb/<id>/ui/commands/getProjectGraph`

Parameter: None

Return values:

Table A.34

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON
projectgraphNodes	the nodes of the project graph	JSON
projectgraphEdges	the edges of the project grpah	JSON
projectgraphRoot	the id of the root node	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/getProjectGraph`

A.2.12 Add external file to scene

This command behaves similarly to 'Objects/insert file' on the GUI, it allows the adding of files of different memeTypes(e.g.XEG, OBJ, DTD..) to the scene.

The command can be used in two different ways depending on the use case and the infrastructure.

Path-based

This way requires that the server can access the location of the file.

Path: `wb/<id>/ui/commands/addNode`

Parameter:

Table A.35

key	description	type	required	default
path	the path to the file	GET	required	-

Return values:

Table A.36

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/addNode?path=/absolute/path/to/your//file.obj`

Request body-based

In this way, the imported object is stored in the body of the request either as bytecode or as text.

Path: `wb/<id>/ui/commands/addNode`

Parameter:

Table A.37

key	description	type	required	default
extension	the extension to the given file file	GET	required	-

key	description	type	required	default
-	the content of the file	POST	required	-

Return values:

Table A.38

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/addNode?extension=xeg`+ body with XEG-graph

A.2.13 Export the scene

This command exports the scene from the project graph similar to View/export in the 3D view of the GUI. The command either stores the export in a given file (path-based) or returns the export directly in response to the request (Response-based).

Path-based

The given path must be accessible for the API server.

Path: `wb/<id>/ui/commands/export3d`

Parameter:

Table A.39

key	description	type	required	default
path	the path for saving the export	GET	required	-

Return values:

Table A.40

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/export3d?path=/absolute/path/where/the/file/will/be/stored.x3d`

Response-based

In this way, the graph is exported to the format of the given extension and returned in the body of the response.

Path: `wb/<id>/ui/commands/export3d`

Parameter:

Table A.41

key	description	type	required	default
extension	the extension to export in	GET	required	-

Return values:

Table A.42

name	description	format
-	The exported graph as text/code	file/text

Example: `http://localhost:58081/api/wb/1/ui/commands/export3d?extension=obj`

A.2.14 List existing datasets

Data collected during a simulation can be stored in datasets, which are distinguished by their names. This command gives a list of the available datasets of the workbench.

Path: `wb/<id>/ui/commands/getDataset`

Parameter: None

Return values:

Table A.43

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON
data	the list of all datasets	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/getDatasets`

A.2.15 Get the content of a dataset

This command returns the content of a specific dataset as a CSV file in the response body.

Path: `wb/<id>/ui/commands/export3d`

Parameter:

Table A.44

key	description	type	required	default
name	the name of the dataset	GET	required	-

Return values:

Table A.45

name	description	format
-	the csv representation of the content of the dataset	csv /text

Example: `http://localhost:58081/api/wb/1/ui/commands/getDataset?name=data1`

A.2.16 Save the project

The project opened in the workbench can be saved either at a given location (path-based), or by just returning the GSZ file as the content of the response (response-based).

Path-based

This supports GS and GSZ files but requires the server to be able to access the path.

Path: `wb/<id>/ui/commands/saveas`

Parameter:

Table A.46

key	description	type	required	default
path	the path for saving	GET	required	-

Return values:

Table A.47

name	description	format
console	retuns the same as the XL console (e.g. print/println)	JSON
log	contains eventual errors	JSON

Example: `http://localhost:58081/api/wb/1/ui/commands/saveas?path=/absolute/path/to/the/saved.gs`

response-based

In this way, the GSZ file is returned in the body of the response.

Path: `wb/<id>/ui/commands/saveas`

Parameter: None

Return values:

Table A.48

name	description	format
-	The saved project	gsz

Example: `http://localhost:58081/api/wb/1/ui/commands/saveas`

A.2.17 Close Workbench

This command closes a workbench without saving.

Path:
`wb/<id>/ui/commands/close`

Parameter: None

Return values: only empty JSON: "{}"

Example: `http://localhost:58081/api/wb/1/ui/commands/close`

returns the same as the XL console (e.g. print/println)

A.3 Custom commands

GroLink is designed in a way that can also execute any other command that is registered in the GroIMP registry as a CommandItem. Therefore the absolute registry path to the command must be added to the call similar to the following:

```
app/<absolute-registry-path> wb/<id>/<absolute-registry-path>
```

However, this function is very limited since it is not possible to add parameters or define the return value. This will return only the log and the console.

Appendix B

Client library functions

B.1 GroPy functions

In the following the functions implemented in the GroPy library are explained.

B.1.1 GroLink

Table B.1 contains the application commands that are available as functions of a GroLink instance.

Table B.1: List of functions implemented on the GroLink object.

function	return of read()	description
createWB(template="newRGG", name=None)	WBRef	creates new project of the given template
openWB(path = None, content=None)	WBRef	opens an existing project from a path or a binary(content)
loadExamle(name)	WBRef	loads an example project by name
listWB()	JSON	lists all open workbenches
listTemplates()	JSON	lists all available tamplates
listExamples(tag=None)	JSON	lists all examples of all of the given tag
close()	-	stops the API server

B.1.2 WBRef

Table B.2 contains the workbench commands that are available as functions of a WBRef instance.

Table B.2: List of functions implemented on the WBRef object.

function	return of read()	description
listRGGFunctions()	JSON	lists all RGG functions of the model
runRGGFunction(name)	JSON	execute the given RGG function
runXLQuery(query)	JSON	execute the given XL query
listFiles()	JSON	lists all source files of the model
renameFile(name,newName)	JSON	renames the given source file
addFile(path,content=None)	JSON	adds a file from a given path or from content, if content is defined, the path is considered as the name of the file
updateFile(name,content)	JSON	set content as the content of given source file
getFile(name)	File	returns the given file
removeFile(name)	JSON	deletes the given source file
getProjectGraph()	JSON	returns the project graph in JSON syntax
export3d(extension)	File	exports the 3d view to a file of the given extension
export3d_toFile(path)	JSON	exports the 3d view to the file defined in the path
listDatasets()	JSON	lists all exiting datasets
getDataset(name)	File	return given dataset
save()	File	return the current project as gsz file
save(path)	JSON	saves the current project at the given path
close()	JSON	closes the project without saving

B.2 GroR functions

In the following the functions implemented in the GroR library are explained.

B.2.1 GroLink

GroR-GroLink functions take the URL of the API as an argument, this URL is defined as follows:

```
http://<ip-address>:<port>/api/
```

- The IP address is either localhost or the address of the device running GroIMP
- The port is defined by the start of the GroIMP API, by default it is: 58081

The available GroLink functions are listed in B.3:

Table B.3: List of the GroR application functions

function	return	description
GroLink.create(url)	wb	creates a new workbench of the emptyRGG template
GroLink.create(url,type)	wb	creates a new Workbench of the given template
GroLink.open(url,path=path)	wb	opens the project at the given path

B.2.2 WBRef

Each WBRef function takes a workbench reference (wb) as a parameter to clarify on which workbench the command is supposed to be executed. The implemented functions are listed in B.4

Table B.4: The implemented workbench functions of the GroR library.

function	return	description
WBRef.listRGGFunctions(wb)	JSON	lists available RGG functions
WBRef.runRGGFunction(wb,name)	JSON	execute the given RGG function
WBRef.runXLQuery(wb,query)	JSON	execute the given XL query
WBRef.compile(wb)	JSON	compile
WBRef.listFiles(wb)	JSON	lists all source files
WBRef.renameFile(wb, name, newName)	JSON	rename a file
WBRef.addFile(wb, path)	JSON	add file from the given path

function	return	description
WBRef.addFile(wb, name, content)	JSON	add the content as a new file with the given name
WBRef.updateFile(wb,name,content)	JSON	set the content as content of the file of the given name
WBRef.removeFile(wb,name)	JSON	remove the given file with the given name
WBRef.getFile(wb, name)	String	returns the content of the given file
WBRef.getProjectGraph(wb)	String	returns the project graph as String [1]
WBRef.addNode(wb, path)	JSON	adds the file given by the path to the scene.(e.g. obj or xeg)
WBRef.addNode(wb,extension,content)	JSON	adds the content as a file of the type of the extension to the scene.
WBRef.export3d(wb,extension)	String	returns the content of a file of the given extension exported from the 3d scene
WBRef.export3d_toFile(wb,path)	String	exports the 3d scene to the given path
WBRef.listDatasets(wb)	JSON	lists the existing datasets of the model
WBRef.getDataset(wb, name)	String	returns the content of a datasets
WBRef.save(wb)	String	returns the project as a decoded GSZ file
WBRef.save(wb,path)	JSON	saves the project to the given path

1.) The JSON parser included in the library is quite limited and for the ProjectGraph an external one is more suitable, therefore the getProjectGraph function returns a String even though the content is JSON.

Appendix C

GroLink-GroPy Notebook

On the following page, the results of a Jupyter notebook [52] are added to provide a deeper understanding of the capabilities of the GroLink project using the GroPy library.

This interactive notebook is also available as an attachment including the 3D visualization that is missing in the PDF version.

Python as a client for GroLink

December 15, 2023

Basic examples of the interaction with the GroIMP API through the Python client library GroPy.

This Nodebook requires the GroIMP API application to run on an address that is reachable for the Nodeook. How to start the API is described here.

Contents

1 Installation

The following command uses the pip package manager with the custom repository of GroPy and should work without any other preparations. If pip is not an option in your scenario please see here.

[ ]:
```
pip install GroPy --upgrade --index-url https://gitlab.com/api/v4/projects/
↪50527255/packages/pypi/simple
```

1.1 Initializing the connection

To create workbenches and workbench references the link to the API server must be defined at first.

[1]:
```
from GroPy import GroPy
link = GroPy.GroLink("http://localhost:58081/api/")
```

2 Creating a model and getting started

This section is about the very basic usage of GroIMP and how to receive the first information.

2.1 Create a new Workbench

Using the link defined in 0.1 the call to create a workbench can be generated. A call holds all the information needed to trigger a command at the API. Run send the call to the server and read interprets the result. In this case that ends up telling the API to create a workbench and returning the workbench reference wb1.

[2]:
```
wb1 = link.createWB().run().read()
```

2.2 Read basic content

To get an overview of the new workbench the source files and the currently implemented RGG functions.

This also works with calls, yet in that case the calls return JSON files including the requested content and allways the console (like the XL console) and the logs(like the messages)

[3]:
```
# create a call
fileListCall = wb1.listFiles()
# execute the call
fileListCall.run()
# interprete the result
fileList = fileListCall.read()
# show the JSON content
fileList
```

[3]:
```
{'console': [], 'data': ['Model.rgg', 'param/parameters.rgg'], 'logs': []}
```

[4]:
```
# process the recived data

i=1
for f in fileList['data']:
    print(str(i)+")"+f)
    i+=1
```

```
1)Model.rgg
2)param/parameters.rgg
```

[5]:
```
# similar to the two blocks above just a bit shorter
wb1.listRGGFunctions().run().read()['data']
```

[5]:
```
['run', 'Run run']
```

2.3 Execute RGG functions

The above-listed functions can now be executed similarly to pressing the button in the Gui. The call again will return the console and log, in the example they are both empty but if the RGG function contains print commands the results will be in the console.

[6]:
```
# create the call based on the function name
runner = wb1.runRGGFunction("run")
runner.run().read()
```

[6]:
```
{'console': [], 'logs': []}
```

[7]:
```
# A call can also be executed multiple times
for i in range(2):
    print(runner.run().read())
```

```
{'console': [], 'logs': []}
{'console': [], 'logs': []}
```

2.4 Model visualisation

With the export3d call, it is possible to export the model the same way as the export menu in the view3d panel. For the following example, x3d is used because it is very simple to visualize x3d in a web browser by linking it to some javascript.

[8]:
```
# The call returns the content of the exported file, this must be decoded
data = wb1.export3d("x3d").run().read().decode('utf-8')
```

[9]:
```
# just help for visualizing the model

from IPython.display import display, HTML
js ="""<head>
    <script type='text/javascript' src='https://www.x3dom.org/download/x3dom.
 ↪js'> </script>
```

```
    <link rel='stylesheet' type='text/css' href='https://www.x3dom.org/download/
 ↪x3dom.css'/>
</head>"""
display(HTML(js+data))
```

```
<IPython.core.display.HTML object>
```

2.5 Reading the Graph

A key part of GroIMP is the project graph, this graph can be read with the API. The returned graph is formatted in JSON and separated into a list of nodes and a list of edges. The edges are defined as a triple of the parent node ID, child node ID and edge byte. Additionally, the root is defined.

[10]:
```
graph = wb1.getProjectGraph().run() .read()
graph
```

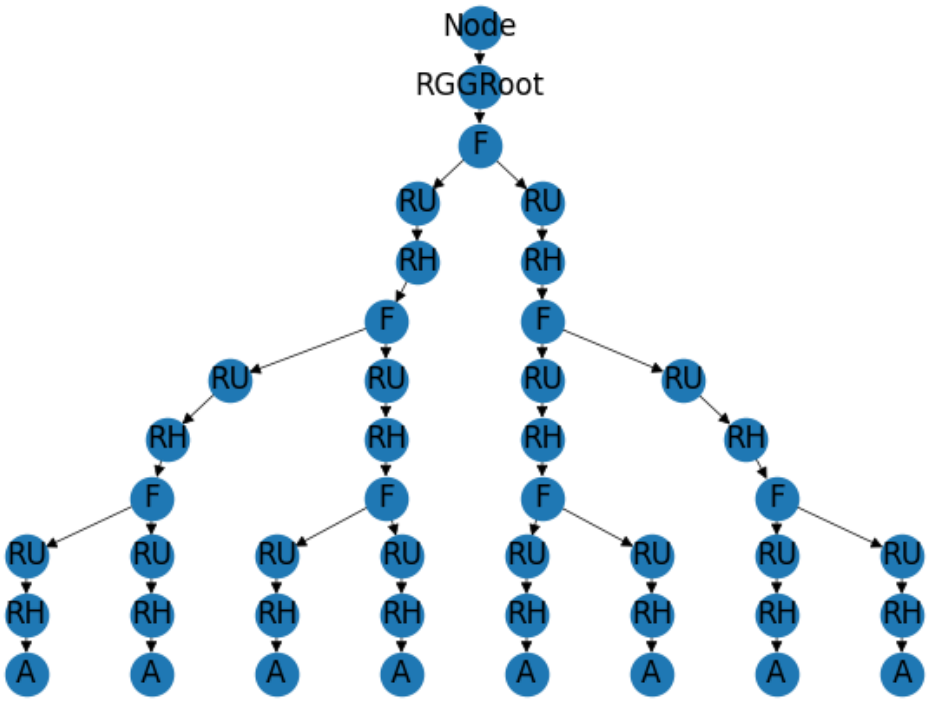
[10]:
```
{'projectgraphNodes': [{'id': 0, 'type': 'de.grogra.graph.impl.Node'},
  {'id': 21, 'type': 'de.grogra.rgg.RGGRoot'},
  {'id': 24, 'type': 'de.grogra.turtle.F'},
  {'id': 25, 'type': 'de.grogra.turtle.RU'},
  {'id': 26, 'type': 'de.grogra.turtle.RH'},
  {'id': 31, 'type': 'de.grogra.turtle.F'},
  {'id': 32, 'type': 'de.grogra.turtle.RU'},
  {'id': 33, 'type': 'de.grogra.turtle.RH'},
  {'id': 45, 'type': 'de.grogra.turtle.F'},
  {'id': 46, 'type': 'de.grogra.turtle.RU'},
  {'id': 47, 'type': 'de.grogra.turtle.RH'},
  {'id': 48, 'type': 'Model.A'},
  {'id': 49, 'type': 'de.grogra.turtle.RU'},
  {'id': 50, 'type': 'de.grogra.turtle.RH'},
  {'id': 51, 'type': 'Model.A'},
  {'id': 35, 'type': 'de.grogra.turtle.RU'},
  {'id': 36, 'type': 'de.grogra.turtle.RH'},
  {'id': 52, 'type': 'de.grogra.turtle.F'},
  {'id': 53, 'type': 'de.grogra.turtle.RU'},
  {'id': 54, 'type': 'de.grogra.turtle.RH'},
  {'id': 55, 'type': 'Model.A'},
  {'id': 56, 'type': 'de.grogra.turtle.RU'},
  {'id': 57, 'type': 'de.grogra.turtle.RH'},
  {'id': 58, 'type': 'Model.A'},
  {'id': 28, 'type': 'de.grogra.turtle.RU'},
  {'id': 29, 'type': 'de.grogra.turtle.RH'},
  {'id': 38, 'type': 'de.grogra.turtle.F'},
  {'id': 39, 'type': 'de.grogra.turtle.RU'},
  {'id': 40, 'type': 'de.grogra.turtle.RH'},
  {'id': 59, 'type': 'de.grogra.turtle.F'},
  {'id': 60, 'type': 'de.grogra.turtle.RU'},
```

```
 {'id': 61, 'type': 'de.grogra.turtle.RH'},
 {'id': 62, 'type': 'Model.A'},
 {'id': 63, 'type': 'de.grogra.turtle.RU'},
 {'id': 64, 'type': 'de.grogra.turtle.RH'},
 {'id': 65, 'type': 'Model.A'},
 {'id': 42, 'type': 'de.grogra.turtle.RU'},
 {'id': 43, 'type': 'de.grogra.turtle.RH'},
 {'id': 66, 'type': 'de.grogra.turtle.F'},
 {'id': 67, 'type': 'de.grogra.turtle.RU'},
 {'id': 68, 'type': 'de.grogra.turtle.RH'},
 {'id': 69, 'type': 'Model.A'},
 {'id': 70, 'type': 'de.grogra.turtle.RU'},
 {'id': 71, 'type': 'de.grogra.turtle.RH'},
 {'id': 72, 'type': 'Model.A'}],
'console': [],
'projectgraphEdges': [[0, 21, 512],
 [21, 24, 256],
 [24, 25, 512],
 [24, 28, 512],
 [25, 26, 256],
 [26, 31, 256],
 [31, 32, 512],
 [31, 35, 512],
 [32, 33, 256],
 [33, 45, 256],
 [45, 46, 512],
 [45, 49, 512],
 [46, 47, 256],
 [47, 48, 256],
 [49, 50, 256],
 [50, 51, 256],
 [35, 36, 256],
 [36, 52, 256],
 [52, 53, 512],
 [52, 56, 512],
 [53, 54, 256],
 [54, 55, 256],
 [56, 57, 256],
 [57, 58, 256],
 [28, 29, 256],
 [29, 38, 256],
 [38, 39, 512],
 [38, 42, 512],
 [39, 40, 256],
 [40, 59, 256],
 [59, 60, 512],
 [59, 63, 512],
```

```
  [60, 61, 256],
  [61, 62, 256],
  [63, 64, 256],
  [64, 65, 256],
  [42, 43, 256],
  [43, 66, 256],
  [66, 67, 512],
  [66, 70, 512],
  [67, 68, 256],
  [68, 69, 256],
  [70, 71, 256],
  [71, 72, 256]],
 'projectgraphRoot': '0',
 'logs': []}
```

More over it is possible to visualize the Graph...

[11]:
```
import networkx as nx
import matplotlib.pyplot as plt
G = nx.DiGraph()
labels={}
pos = {0: (0, 0)}
for n in graph['projectgraphNodes']:
    G.add_node(n['id'])
    labels[n['id']] = n['type'].split('.')[-1]
for e in graph['projectgraphEdges']:
    G.add_edge(e[0], e[1])
pos = nx.nx_agraph.graphviz_layout(G, prog="dot")
nx.draw_networkx_labels(G, pos, labels)
nx.draw(G, pos,width=0.5)
plt.tight_layout()
plt.axis("off")
plt.show()
```



2.6 Closing a workbench

A workbench opened with the API must be closed with the API or it stays open til the API is stopped. wb1.close().run()

3 XEG Pipeline

The following example shows how a subgraph can be imported, changed and exported with the API.

3.1 Create an empty project with a function to manipulate

To generate a basic environment for the pipe line a new workbench is created and the source file is changed to create the needed environment.

[19]:
```
# creat a new workbench
wb2 = link.createWB().run().read()
```

[20]:
```
# update the Model.rgg file like in JEdit in the GUI
wb2.updateFile("Model.rgg","""
module B(float len) extends Sphere(0.1)
```

```
{
        {setShader(GREEN);}
}


public void change ()
[
        B(x) ==> F(x) B(x);
]

""").run().read()

# compile to build the new scene
wb2.compile().run().read()
```

[20]:
```
{'console': [], 'logs': []}
```

Now the project graph is empty and does and only contains a not-used axiom.

[21]:
```
wb2.getProjectGraph().run().read()
```

[21]:
```
{'projectgraphNodes': [{'id': 0, 'type': 'de.grogra.graph.impl.Node'},
  {'id': 30, 'type': 'de.grogra.rgg.RGGRoot'},
  {'id': 31, 'type': 'de.grogra.rgg.Axiom'}],
 'console': [],
 'projectgraphEdges': [[0, 30, 512], [30, 31, 256]],
 'projectgraphRoot': '0',
 'logs': []}
```

3.2 Add an XEG graph to the model

It is possible to add objects defined by files similar to using 'Objects/insert file' in the GUI. In this example, XEG is used.

[22]:
```
wb2.addNode("xeg","""
<graph>
  <root root_id="0"/>
  <type name="B">
    <extends name="Sphere"/>
    <property name="len" type="float"/>
  </type>
  <node id="1" name="" type="B">
    <property name="radius" value="0.1"/>
    <property name="color">
      <rgb>0.0 1.0 0.0</rgb>
    </property>
    <property name="len" value="1.0"/>
  </node>
```

```
  <edge id="2" src_id="0" dest_id="1" type="decomposition"/>
</graph>
""").run().read()
```

[22]:
```
{'console': [], 'logs': []}
```

[23]:
```
# The graph now contains a node called Model.B as the one defined in XEG
wb2.getProjectGraph().run().read()
```

[23]:
```
{'projectgraphNodes': [{'id': 0, 'type': 'de.grogra.graph.impl.Node'},
  {'id': 30, 'type': 'de.grogra.rgg.RGGRoot'},
  {'id': 32, 'type': 'Model.B'}],
 'console': [],
 'projectgraphEdges': [[0, 30, 512], [30, 32, 256]],
 'projectgraphRoot': '0',
 'logs': []}
```

3.3 Execute the RGG function to change the sub-graph

Now the in 'main.rgg' defined function can be executed to change the scene graph.

[24]:
```
wb2.runRGGFunction("change").run().read()
```

[24]:
```
{'console': [], 'logs': []}
```

3.4 export to XEG again

The project graph can now be exported back to XEG in the same way as above in 1.4. The newly created XEG contains an additional F.

[25]:
```
print(wb2.export3d("xeg").run().read().decode('utf-8'))
```

```
<graph>
  <root root_id="0"/>
  <type name="B">
    <extends name="Sphere"/>
    <property name="len" type="float"/>
  </type>
  <node id="2" name="" type="B">
    <property name="radius" value="0.1"/>
    <property name="color">
      <rgb>0.0 1.0 0.0</rgb>
    </property>
    <property name="len" value="1.0"/>
  </node>
  <edge id="3" src_id="3" dest_id="2" type="successor"/>
  <node id="3" name="" type="F">
    <property name="length" value="1.0"/>
    <property name="diameter" value="-1.0"/>
```

```
    <property name="fcolor" value="-1"/>
  </node>
  <edge id="5" src_id="0" dest_id="2" type="decomposition"/>
  <edge id="6" src_id="0" dest_id="3" type="decomposition"/>
</graph>
```

[26]:
```
wb2.close().run()
```

[26]:
```
<GroPy.GroPy.Call at 0x7fd4d9596c40>
```

4 XL queries and rules

Similar to rgg functions it is also possible to manipulate the model with xl queries and rewriting rules.

[21]:
```
# first a workbench is created based on a empty rgg template
wb3 = link.createWB(template="newEmptyRGG").run().read()
wb3.getProjectGraph().run().read()
```

[21]:
```
{'projectgraphNodes': [{'id': 0, 'type': 'de.grogra.graph.impl.Node'},
  {'id': 8, 'type': 'de.grogra.rgg.RGGRoot'},
  {'id': 9, 'type': 'de.grogra.rgg.Axiom'}],
 'console': [],
 'projectgraphEdges': [[0, 8, 512], [8, 9, 256]],
 'projectgraphRoot': '0',
 'logs': []}
```

[22]:
```
# simple rewriting
wb3.runXLQuery("[Axiom==> F;]").run().read()
```

[22]:
```
{'console': [], 'logs': []}
```

[23]:
```
wb3.getProjectGraph().run().read()
```

[23]:
```
{'projectgraphNodes': [{'id': 0, 'type': 'de.grogra.graph.impl.Node'},
  {'id': 8, 'type': 'de.grogra.rgg.RGGRoot'},
  {'id': 10, 'type': 'de.grogra.turtle.F'}],
 'console': [],
 'projectgraphEdges': [[0, 8, 512], [8, 10, 256]],
 'projectgraphRoot': '0',
 'logs': []}
```

[24]:
```
# simple querying for information
print("number of F nodes:"+ wb3.runXLQuery("count((*F*))").run().
  ↪read()['console'][0])
print("number of nodes:"+ wb3.runXLQuery("count((*Node*))").run().
  ↪read()['console'][0])
```

```
number of F nodes:1
number of nodes:2
```

[25]:
```
# more rewriting
wb3.runXLQuery("[F==>F [F F];]").run().read()
```

[25]:
```
{'console': [], 'logs': []}
```

[26]:
```
wb3.getProjectGraph().run().read()
```

[26]:
```
{'projectgraphNodes': [{'id': 0, 'type': 'de.grogra.graph.impl.Node'},
  {'id': 8, 'type': 'de.grogra.rgg.RGGRoot'},
  {'id': 11, 'type': 'de.grogra.turtle.F'},
  {'id': 12, 'type': 'de.grogra.turtle.F'},
  {'id': 13, 'type': 'de.grogra.turtle.F'}],
 'console': [],
 'projectgraphEdges': [[0, 8, 512],
  [8, 11, 256],
  [11, 12, 512],
  [12, 13, 256]],
 'projectgraphRoot': '0',
 'logs': []}
```

[27]:
```
# changing one edge between two nodes
wb3.runXLQuery("[a:F +> b:F ==> a > b;]").run().read()
```

[27]:
```
{'console': [], 'logs': []}
```

[28]:
```
wb3.getProjectGraph().run().read()
```

[28]:
```
{'projectgraphNodes': [{'id': 0, 'type': 'de.grogra.graph.impl.Node'},
  {'id': 8, 'type': 'de.grogra.rgg.RGGRoot'},
  {'id': 11, 'type': 'de.grogra.turtle.F'},
  {'id': 12, 'type': 'de.grogra.turtle.F'},
  {'id': 13, 'type': 'de.grogra.turtle.F'}],
 'console': [],
 'projectgraphEdges': [[0, 8, 512],
  [8, 11, 256],
  [11, 12, 256],
  [12, 13, 256]],
 'projectgraphRoot': '0',
 'logs': []}
```

[ ]:
```
wb3.close().run()
```

5 Reading Data

To receive more data measured during the simulation, datasets are used, which can be read through the API. To demonstrate that a simple existing model with two light sources and two "plants" is used.

[29]:
```
import pandas as pd
import io
from matplotlib import pyplot as plt
```

5.1 Opening an existing model

An existing model can be opened by either sending a path to the project file or by sending a .gsz file as binary content.

[30]:
```
# The model is read in Python as binary (rb) and the content is transferred to␣
 ↪the API
wb4 = link.openWB(content=open("left_right_light.gsz",'rb').read()).run().read()
```

[31]:
```
# reading the content of the Model.rgg file
print(wb4.getFile("Model.rgg").run().read().decode("utf-8"))
```

```
// import other compiled files (.rgg, .java, .xl) without subdirectory path.
import parameters.*;
static FluxLightModel lm = new FluxLightModel(50000, 5);
module A(float len) extends Sphere(0.1)
{
        {setShader(GREEN);}
}

module pL();

module pR();

DatasetRef lightSum = new DatasetRef("lightSum");

module Light(float pow) extends LightNode(){
                        DirectionalLight dl= new DirectionalLight();
                {
                        dl.(setPowerDensity(pow));
                        setLight(dl);
                }
                void setPow(float po){
                        dl.(setPowerDensity(po));

                }
        }
```

```
module lL(float po) extends Light(po);

module lR(float po) extends Light(po);


module Leaf(float x, float y)==>leaf(x,y);

protected void init ()
[
        {
                lightSum.clear();
                lightSum.addRow()
                        .setText(0,"left")
                        .setText(1,"right");

        }
        Axiom ==> [Translate(1,0,0)RH(180)pR()A(parameters.length)]
        [Translate(-1,0,0)pL()A(parameters.length)]
        [
        Translate(5,0,10)RU(205) lR(100)
        ]
        [
        Translate(-5,0,10)RU(-205) lL(100)
        ]



        ;
]

public void grow ()
[
        A(x) ==> F(x) [RU(80) RH(90)Leaf(x,x)] [RU(-10) RH(90) A(x*0.9)];

        {
        lm.compute();
        lightSum.addRow().set(0,sum(lm.getAbsorbedPower3d((*pL -descendants->
Leaf*)).integrate())).set(1,sum(lm.getAbsorbedPower3d((*pR -descendants->
Leaf*)).integrate()));
        }
/*      l:Leaf::>{
                print(lm.getAbsorbedPower3d(l).integrate());
        }*/
]
```

5.2 Growing and reading data

With the execution of the growth function, the plants grow and add data to the dataset. This data set can then be plotted.

[32]:
```
# define the calls
grower = wb4.runRGGFunction("grow")
reader = wb4.getDataset("lightSum")
```

[33]:
```
# Run the Model 3 times
for i in range(3):
    grower.run()
```

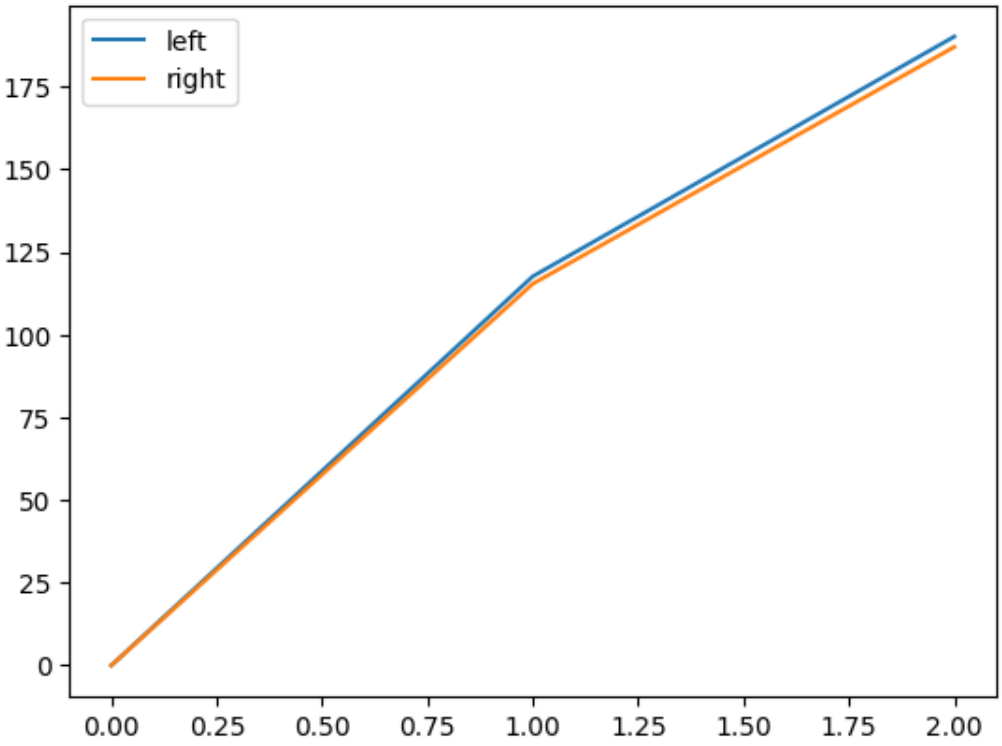
The received dataset is formated as csv and can directly be transfered into a pada dataframe

[34]:
```
data = pd.read_csv(io.StringIO(reader.run().read().decode('utf-8')))
data
```

[34]:

	left	right
0	0.000000	0.000000
1	117.676081	115.423834
2	190.185858	187.045337

[35]:
```
# based on the dataframe a simple plot can be created
plt.plot(data['left'],label="left")
plt.plot(data['right'], label="right")
plt.legend(loc="upper left")
plt.show()
```



5.3 Changing conditions

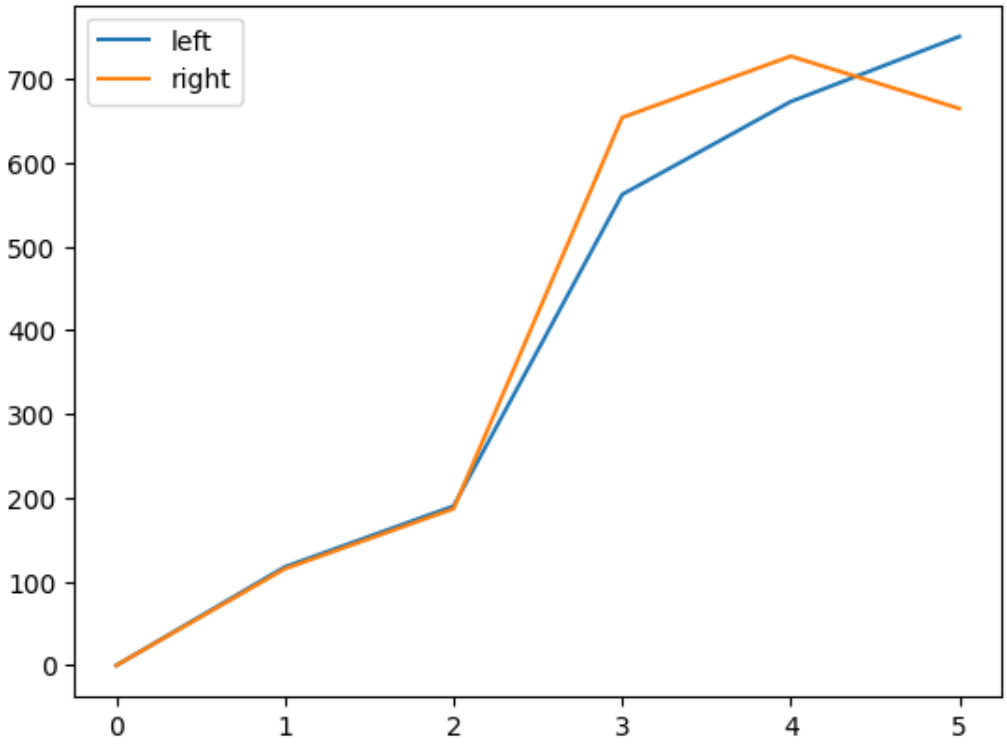
To show the ability to interact with a running simulation the left light source will be increased to 400 before running additional growth steps.

[36]:
```
wb4.runXLQuery("[Model.lL(x)==>Model.lL(400);]").run().read()
```

[36]:
```
{'console': [], 'logs': []}
```

Redo step 4.2

[37]:
```
for i in range(3):
    grower.run()
data = pd.read_csv(io.StringIO(reader.run().read().decode('utf-8')))
plt.plot(data['left'],label="left")
plt.plot(data['right'], label="right")
plt.legend(loc="upper left")
plt.show()
```



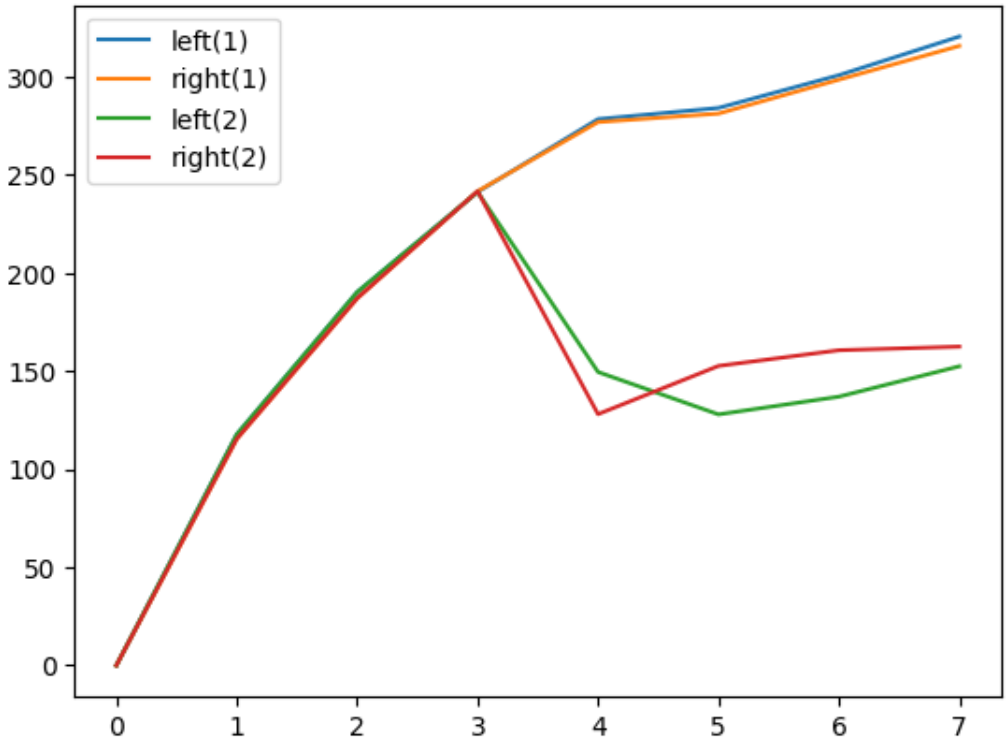
[ ]:
```
wb4.close().run()
```

5.4 combination

The steps above can also be used to create a direct comparison of the influence of a change in the light source.

[38]:
```
# open the project twice
opener = link.openWB(content=open("left_right_light.gsz",'rb').read())
wb41=opener.run().read()
wb42=opener.run().read()

# define the calles
grower1 = wb41.runRGGFunction("grow")
reader1 = wb41.getDataset("lightSum")
grower2 = wb42.runRGGFunction("grow")
reader2 = wb42.getDataset("lightSum")


# grow both models
for i in range(4):
    grower1.run()
```

```
    grower2.run()

# turn of the left light on one of them
wb42.runXLQuery("[Model.lL(x)==>Model.lL(0);]").run().read()

# grow both models.
for i in range(4):
    grower1.run()
    grower2.run()

# read the datasets
data1 = pd.read_csv(io.StringIO(reader1.run().read().decode('utf-8')))
data2 = pd.read_csv(io.StringIO(reader2.run().read().decode('utf-8')))

# close the models
wb41.close().run()
wb42.close().run()

# plot
plt.plot(data1['left'],label="left(1)")
plt.plot(data1['right'], label="right(1)")
plt.plot(data2['left'],label="left(2)")
plt.plot(data2['right'], label="right(2)")

plt.legend(loc="upper left")
plt.show()
```



6 Running in Parallel

GroLink executes commands in a nonblocking way, meaning that several commands can be pushed to the API and they are executed in parallel. This is shown by a simple example of a function that just waits for 5 seconds and then prints hello.

5.1 creating 50 workbenches with the same command

[39]:
```
%%time
# create a new RGG workbench
wb5= link.createWB().run().read()
# define function and compile
wb5.updateFile("Model.rgg","""
      public void runner(){
        Thread.sleep(5000);
        println("hello");
    }
""").run()
wb5.compile().run()

# saving the newly created workbench as a python variable
data_tmp = wb5.save().run().read()
```

```
wb5.close().run()


# open 50 instances based on the saved model
wbs=[]
for i in range(0,50):
    wbs.append(link.openWB(content = data_tmp).run().read())
```

```
CPU times: user 222 ms, sys: 64 ms, total: 286 ms
Wall time: 6.89 s
```

6.1 Compare running one model to running 50

[40]:
```
# a help function for paralellisation
def callIt(wb):
    return wb.runRGGFunction("runner").run().read()
```

running the function on one Workbench

[41]:
```
%%time
callIt(wbs[0])
```

```
CPU times: user 7.73 ms, sys: 803 µs, total: 8.54 ms
Wall time: 5.04 s
```

[41]:
```
{'console': ['hello'], 'logs': []}
```

Running it on all Workbenches

[42]:
```
%%time
import multiprocessing
if __name__ == '__main__':
    # the processes variable defines how many requests are done in parallel, at this point that is the only limt(the API server will just try to run all you put in)
    pool = multiprocessing.Pool(processes=50)
    results = pool.map(callIt,wbs)
results
```

```
CPU times: user 112 ms, sys: 425 ms, total: 537 ms
Wall time: 6 s
```

[42]:
```
[{'console': ['hello'], 'logs': []},
 {'console': ['hello'], 'logs': []},
 {'console': ['hello'], 'logs': []},
 {'console': ['hello'], 'logs': []},
 {'console': ['hello'], 'logs': []},
 {'console': ['hello'], 'logs': []},
 {'console': ['hello'], 'logs': []},
 {'console': ['hello'], 'logs': []},
```

```
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []},
{'console': ['hello'], 'logs': []}]
```

[ ]:
```
%%time
for wb in wbs:
    wb.close().run()
```

Appendix D

GroLink-Greenhouse Notebook

On the following page, the printed version of a Jupyter notebook [52] is added to show a small proof of concept regarding the connection of the GroLink API with the already mentioned semi-closed greenhouse project. This interactive notebook and a version of the Greenhouse model are also available as an attachment.

GroLink-greenhouse

December 20, 2023

This notebook is designed to point out the abilities of the newly created GroLink API in connection with a larger GroIMP project, a semi-closed greenhouse with tomato plants.

Contents

1 Create the link and have a first look

After connecting to the GroLink server the Greenhouse project can be opened and the files and available RGG functions listed. This gives a first impression of the project.

[263]:
```
from GroPy import GroPy
link = GroPy.GroLink("http://localhost:58081/api/")
```

[264]:
```
wb_greenhouse = link.openWB(content=open("Greenhouse.gsz",'rb').read()).run().
↪read()
```

[265]:
```
wb_greenhouse.listFiles().run().read()['data']
```

[265]:
```
['DigitalTwin.rgg',
 'Greenhouse.rgg',
 'plants/KL.java',
 'plants/PlantInterfaces.rgg',
 'plants/Tomato.rgg',
 'plants/PlantHandler.rgg',
 'plants/PlantHandlerImpl.rgg',
 'Shaders/Shaders.rgg',
 'Objects/Objects.rgg',
 'Parts/Heating.rgg',
 'Parts/GlassParts.rgg',
 'Parts/SteelParts.rgg',
 'Parts/Shading_screens.rgg',
```

```
 'Parts/Gutters.rgg',
 'Parts/Floor.rgg',
 'Parts/VentilationPipes.rgg',
 'ComplexParts/Roof.rgg',
 'ComplexParts/Walls.rgg',
 'Utils/Cameras.rgg',
 'Utils/Config.rgg',
 'Utils/Logging.rgg',
 'Sensors/Sensors.rgg',
 'Sensors/SensorHandler.rgg',
 'Sensors/SensorHandlerImpl.rgg',
 'LightEnvironment.rgg',
 'input/config.properties',
 'input/planting_plan.txt',
 'input/light_sensor_position.txt']
```

[266]:
```
wb_greenhouse.listRGGFunctions().run().read()['data']
```

[266]:
```
['day', 'hour', 'checkSensors', 'Run day', 'Run hour', 'Run checkSensors']
```

2 Change initial configuration

The Greenhouse model comes with three parameter files that can predefine the simulation environment. With the newest version of the greenhouse, these files can be added to the gsz file by using the new getInputStreamFromProject("<fileName>") function. The most important one is the config.properties file, this as well as the planting_plan.txt (position and type of the plants) and the sensor_position.txt (location for the sensors) can be edited.

In the following the config.properties file is edited to change the logging to the API logger which enables the logging of individual plants. Moreover, the planting_plan is changed to add four plants in two different growth states to the greenhouse.

[267]:
```
wb_greenhouse.updateFile("input/config.properties","""

# ---- init ----
groIMPPath=/home/tim/programme/java/openlogic-openjdk-8u262-b10-linux-64/bin/
 ↪java -Xss1g -Xms1g -Xmx1g   -jar /home/tim/programme/GroIMP/dev/app/core.jar

# ---- input ----

GrowPlan = pfs:/input/planting_plan.txt
SensorList=pfs:/input/light_sensor_position.txt

# ---- output ----
fly_path= output/vid1
grow_path= output/vid2
sensor_log_path=output/
```

```
report_path=output/


# ---- Render ----
ImgWidth = 320
imgHeight = 240


# ---- Sky model ----
latitude = 51
distance = 40000
distfact= 300
attenuation = 0.19
sky_nodes=20

# ---- Light model ----
rays= 500000
reflections =5

#-----Logger------
## 0:none 1:base, 2:api
logger = 2



# ---- play book ----

## init
greenhouse= True
sky = True
sensors = True
plants = True


### Start
doy=140
hour=10

## run
loops= 24
plantGrow=True

### treatment
averageMovingHeight=150

# ---- Report ----
sensorSinglePlot=True
```

```
sensorComparePlot=True
sensorHeatMap=False
sensorHeatMapWidth=21
sensorHeatMapLength=13

plantSingelPlot=True
plantComparePlot=True


""").run().read()
```

[267]:
```
{'console': [], 'logs': []}
```

[268]:
```
wb_greenhouse.updateFile("input/planting_plan.txt","""15,15,1
25,5,2
20,4,1
4,10,1
""").run().read()
```

[268]:
```
{'console': [], 'logs': []}
```

To apply the changes above the model needs to be recompiled

[269]:
```
wb_greenhouse.compile().run().read()
```

[269]:
```
{'console': [], 'logs': []}
```

3 Work with the plants

Using XL queries it is possible to add plants to the current simulation or remove them, without recompiling. Moreover, with the usage of RGG functions, it is possible to trigger the growth steps of the simulation and it is possible to receive information about the plants in the greenhouse, by using datasets.

[270]:
```
# add a plant of type 1 to the 10nth place on the 10nth gutter
wb_greenhouse.runXLQuery("(*PlantHandler.PlantHandler*).addPlant(10,10,1)").
  ↪run().read()
```

[270]:
```
{'console': [], 'logs': []}
```

[271]:
```
# run the simulation for 10 days
for i in range(0,10):
    wb_greenhouse.runRGGFunction("day").run().read()
```

[272]:
```
# list the available log files/datasets
wb_greenhouse.listDatasets().run().read()['data']
```

[272]:
```
['p_10x10',
 'p_15x15',
 'p_25x5',
 'p_20x4',
 'p_4x10',
 's_2000x2000x140',
 's_2000x1000x140',
 's_2000x4000x140',
 's_2500x4000x140']
```

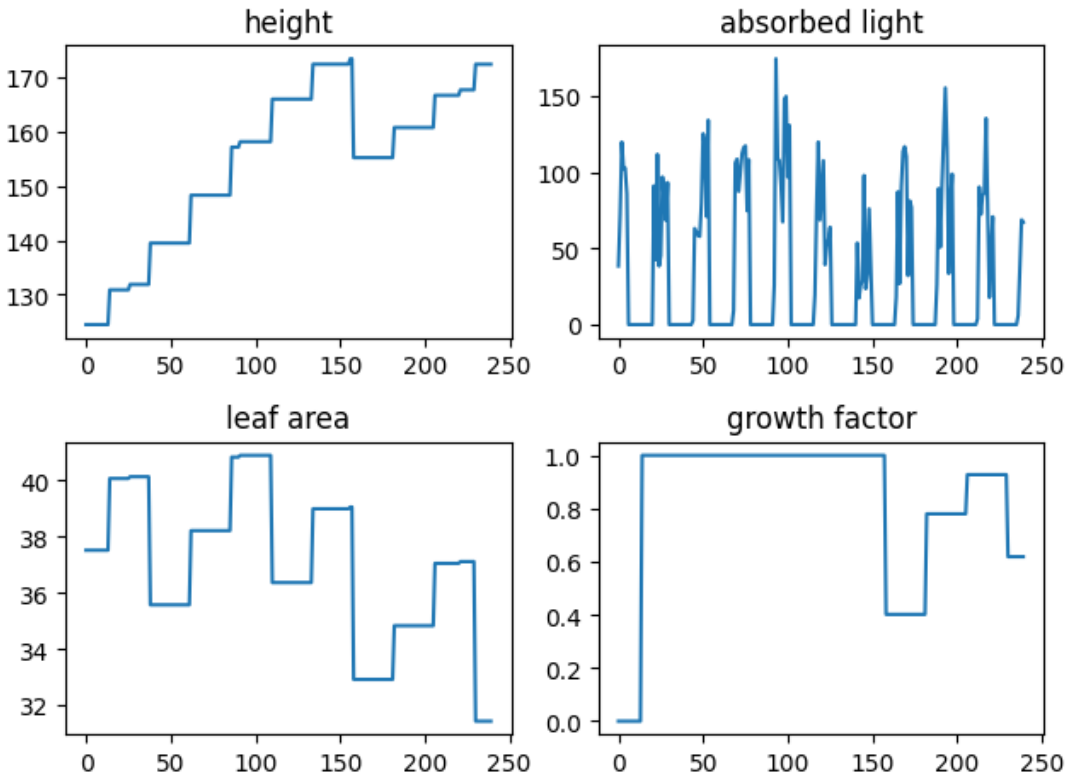
[275]:
```
# turn the log data form the plant at a position 10x10 to a pandas dataframe
import pandas as pd
import io
df = pd.read_csv(io.StringIO(wb_greenhouse.getDataset("p_10x10").run().read().
 ↪decode('utf-8')), header=None)
df.rename(columns={0: "height",1: "absorbed light",2:"leaf area",3: "growth␣
 ↪factor"}, inplace=True)
df
```

[275]:

	height	absorbed light	leaf area	growth factor
0	124.515854	38.450417	37.516018	0.000000
1	124.515854	72.546930	37.516018	0.000000
2	124.515854	119.797270	37.516018	0.000000
3	124.515854	102.523560	37.516018	0.000000
4	124.515854	102.648470	37.516018	0.000000
..	…	…	…	…
235	172.359440	0.000000	31.415749	0.618732
236	172.359440	5.854876	31.415749	0.618732
237	172.359440	36.322630	31.415749	0.618732
238	172.359440	68.743380	31.415749	0.618732
239	172.359440	67.115860	31.415749	0.618732

[240 rows x 4 columns]

[276]:
```
# plot the data from the plant at 10x10
import matplotlib.pyplot as plt
fig, axis = plt.subplots(2, 2)
axis[0,0].plot(df["height"])
axis[0,0].set_title("height")
axis[0,1].plot(df["absorbed light"])
axis[0,1].set_title("absorbed light")
axis[1,0].plot(df["leaf area"])
axis[1,0].set_title("leaf area")
axis[1,1].plot(df["growth factor"])
axis[1,1].set_title("growth factor")
fig.tight_layout()
plt.show()
```



About the plots:

height: The height in cm increases until the average of all plants is higher than 150 cm (as defined in averageMovingHeight in config.properties) then the upper part of the plant is moved side wise like a curtain and therefore the height decreases because side wise growth is not considered here.

leaf area: The total leaf area of the plant depends on the increase of the leaves and the development of new leaves as well as on the removal of the lower leaves, which is a common treatment in greenhouse horticulture.

absorbed light: The light absorbed by the leaves of the plant

growth factor: A relative value that defines how much of the carbon needed for growing is available.

3.1 The Sensors and the LightEnvironment

The sensors added to the simulation are logging their measurements on an hourly base, this can be read from the API into a dataframe and then plotted as shown below. That plot shows the difference in light interception of the sensors and the change over time.

[277]:
```
#Requesting a dataset and turning the response into a pandas dataframe
df_sensor0 = pd.read_csv(io.StringIO(wb_greenhouse.
  ↪getDataset("s_2000x2000x140").run().read().decode('utf-8')), header=None)
#updating the name
```

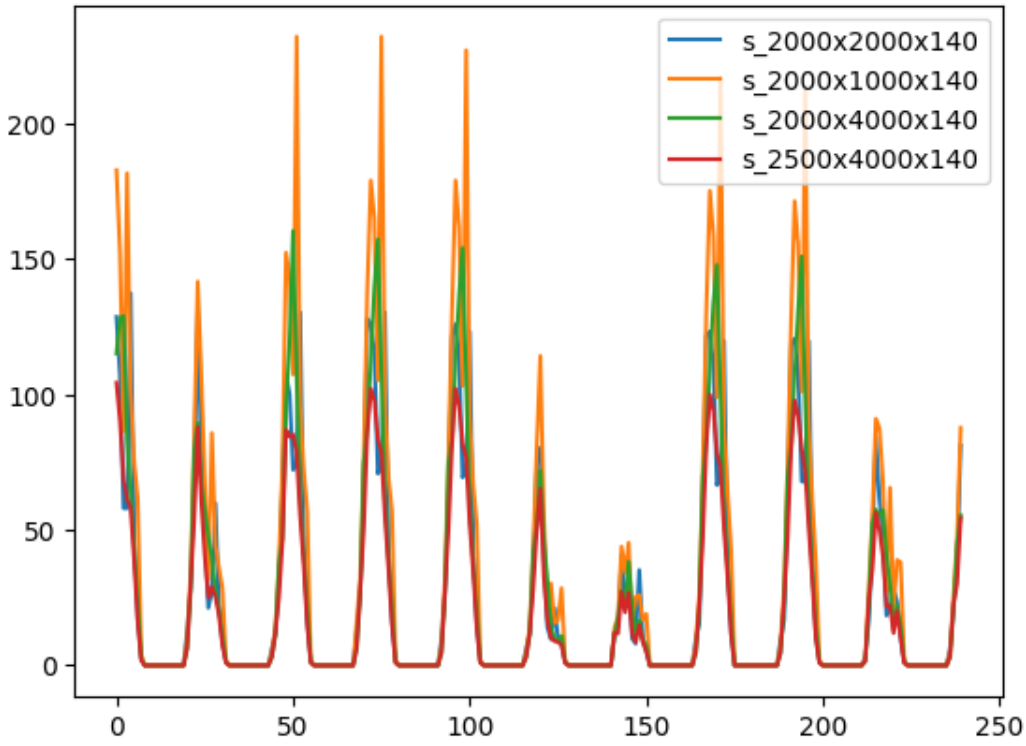
```
df_sensor0.rename(columns={0: "s_2000x2000x140"}, inplace=True)
df_sensor1 = pd.read_csv(io.StringIO(wb_greenhouse.
 ↪getDataset("s_2000x1000x140").run().read().decode('utf-8')), header=None)
df_sensor1.rename(columns={0: "s_2000x1000x140"}, inplace=True)
df_sensor2 = pd.read_csv(io.StringIO(wb_greenhouse.
 ↪getDataset("s_2000x4000x140").run().read().decode('utf-8')), header=None)
df_sensor2.rename(columns={0: "s_2000x4000x140"}, inplace=True)
df_sensor3 = pd.read_csv(io.StringIO(wb_greenhouse.
 ↪getDataset("s_2500x4000x140").run().read().decode('utf-8')), header=None)
df_sensor3.rename(columns={0: "s_2500x4000x140"}, inplace=True)

#merging the dataframes of the different sensors
df_sensor = pd.concat([df_sensor0,df_sensor1,df_sensor2,df_sensor3], axis=1).
 ↪reindex(df_sensor0.index)
df_sensor
```

[277]:

	s_2000x2000x140	s_2000x1000x140	s_2000x4000x140	s_2500x4000x140
0	128.611680	182.784130	115.098820	104.322900
1	106.964250	152.004040	128.300000	90.160710
2	58.028465	86.327680	128.900000	68.448135
3	58.006783	181.620970	83.450390	61.386044
4	137.331530	101.722860	64.423360	58.188854
..	…	…	…	…
235	0.000000	0.000000	0.000000	0.000000
236	6.259963	5.066280	3.027400	3.250860
237	22.195906	25.904583	21.128096	22.516033
238	38.484325	46.585457	44.802210	30.216904
239	81.140564	87.695070	55.377155	54.427340

[240 rows x 4 columns]

[278]:
```
# Plotting the measurements of the sensors
df_sensor.plot()
plt.show()
```



Since the light environment is a GroIMP module it can be used in XL queries to either receive or update information. This is used at this point to make sure it is daytime in the simulation. This is necessary for the step below.

[279]:
```
print(wb_greenhouse.runXLQuery("(*LightEnvironment.LightEnvironment*).time").
  ↪run().read())
print(wb_greenhouse.runXLQuery("(*LightEnvironment.LightEnvironment*).doy").
  ↪run().read())
```

```
{'console': ['10'], 'logs': []}
{'console': ['150'], 'logs': []}
```

[280]:
```
wb_greenhouse.runXLQuery("(*LightEnvironment.LightEnvironment*).time=11").run().
  ↪read()
```

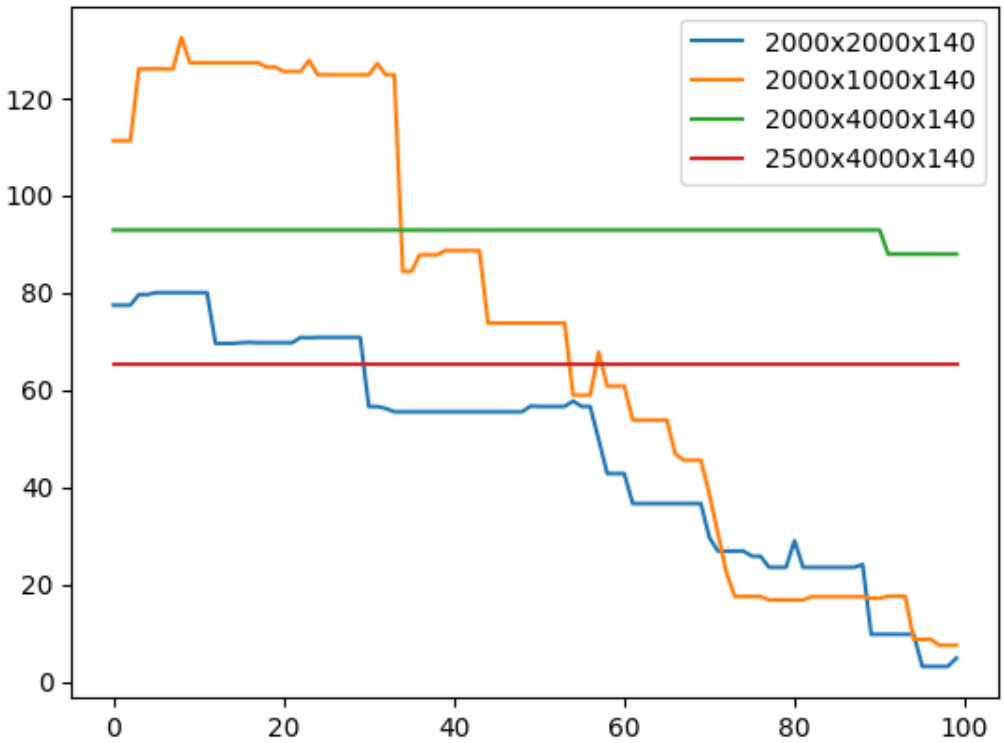
[280]:
```
{'console': ['11'], 'logs': []}
```

To test the effect of the shading screens in the simulation these shading screens are closed step by step in 100 steps and the sensors are read after every step.

The result shows that the sensors inside the greenhouse (2000x2000x140 and 2000x1000x140) are decreasing over time. The increase in between is most likely due to either the reflections of the shading screens or measurement errors.

[281]:
```
lables=[d.split(':')[0] for d in wb_greenhouse.runRGGFunction("checkSensors").
 ↪run().read()['console']]
data=[[] for i in range(0,len(lables))]
for i in range(0,100):
    wb_greenhouse.runXLQuery("(*Shading_screens.ShadingScreen*).
 ↪setClosed("+str(i*0.01)+")").run().read()
    j=0
    for d in wb_greenhouse.runRGGFunction("checkSensors").run().
 ↪read()['console']:
        data[j].append(float(d.split(':')[1]))
        j+=1
```

[282]:
```
import matplotlib.pyplot as plt
x=[i for i in range(0,len(data[0]))]
for i in range(0,len(lables)):
    plt.plot(x,data[i],label=lables[i])
plt.legend(loc="upper right")
plt.show()
```



[ ]:
```
# Close the workbench
wb_greenhouse.close().run().read()
```