FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Glub - OpenGL configurator
Student:	Jakub Drgoň
Supervisor:	Ing. Jiří Chludil
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Cílem práce je vytvořit konfigurovatelné rozhraní, jehož výstupem bude kompilovatelný OpenGL projekt so zvolenými knihovnami.

1. Analyzujte současné řešení kompilace OpenGL projektů.
2. Pomocí metod softwarového inženýrství navrhněte generování OpenGL projektů se specifickými knihovnami
3. Implementujte REST API, které bude vytvářet projekt se specifickou konfiguraci
4. Implementujte prototyp webového rozhraní pro vytvořené REST API
5. Vytvořte automatické testovaní projektů generovaných prostřednictvím REST API

Electronically approved by Ing. Michal Valenta, Ph.D. on 28 January 2022 in Prague.

Bachelor's thesis

GLUB - OPENGL CONFIGURATOR

Jakub Drgoň

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Jiří Chludil
May 10, 2022

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Drgoň Jakub. *Glub - OpenGL configurator.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

List of Figures

List of Tables

List of code listings

First and foremost, I would like to thank my fiancé, who has stood by me through all the hard times while I was working on the thesis. She offered her neverending love and support, she was always there to listen and tried to help whenever she could. A big thanks also belongs to the supervisor of this thesis, Ing. Jiří Chludil, for his insightful suggestions on writing this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 10, 2022 .....

Abstract

This thesis aims to simplify the compilation of graphical applications built with OpenGL. Since no solution that would allow the user to set up an OpenGL project with only a few clicks exists, significant emphasis was put on analyzing the technologies that can be used to build glub. This document contains a comparison of multiple technologies and clarifies why CMake is the most suitable one. Based on selected tools in the analysis, the design chapter proposes a structure of how various components should be built and connected. The implementation details are described in the following chapter, which thoroughly explains the inner workings of each component within glub. Despite glub being an open-source project, the implementation chapter is still worth reading. Because apart from describing how glub works, it contains reasons why it is implemented a certain way. The glub significantly simplifies the compilation process of OpenGL projects, so it achieved the main objective. A summary of the whole development process and future plans can be found at the end of the document in the conclusion.

Keywords OpenGL, CMake, C++, compilation, REST API

Abstrakt

Cieľom tejto práce je zjednodušiť kompiláciu grafických aplikácií, ktoré sú postavené na technológií OpenGL. Keďže doposiaľ neexistuje riešenie, ktoré by dovolilo užívateľom nakonfigurovať OpenGL projekt len niekoľkými klikmi, bol kladený veľký dôraz na analýzu technológií, ktoré by mohli byť použité na vytvorenie nástroja glub. Tento dokument obsahuje porovnanie viacerých technológií a zdôvodňuje prečo bol vybraný práve CMake. Na základe nástrojov vybraných v analýze, bol vytvorený návrh komponentov tvoriacich glub a komunikácie medzi nimi. Implementačné detaily sú obsiahnuté v nasledujúcej kapitole, ktorá dôkladne popisuje ako každý komponent funguje. Napriek tomu že glub je open-source projekt, kapitola o jeho implementácií sa vyplatí prečítať. Okrem detailov fungovania vysvetľuje prečo boli niektoré časti implementované určitým spôsobom. Keďže glub výrazne uľahčuje kompiláciu OpenGL aplikácií, hlavný cieľ práce bol splnený. Zhrnutie celého vývoja a plány do budúcnosti sa nachádzajú na konci dokumentu.

Kľíčová slova OpenGL, CMake, C++, kompilácia, REST API

Abbreviation list

API	Application Programming Interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MSVC	MicroSoft Visual C++
MVC	Model View Controller
MVT	Model View Template
OpenGL	Open Graphics Library
RAM	Random Access Memory
REST	REpresentational State Transfer
URL	Uniform Resource Locator
vCPU	virtual Central Processing Unit
VPS	Virtual Private Server

Chapter 1
Introduction

This thesis will focus on creating software that addresses particular issues programmers face when working with OpenGL. First of all, OpenGL must be briefly introduced; afterward, the focus will shift to identifying complex processes in OpenGL development that take a significant amount of time and can be automated.

OpenGL specification defines an interface that can access the graphics subsystem without knowledge of subsystem architecture and its OpenGL implementation. Such a unified API allows programmers to focus on the graphics development rather than implementing support for as many graphics component vendors as possible. However, this API is still relatively complex and low-level, which is why programmers prefer to use libraries for common graphics components.[1]

The integration process of a library into an OpenGL project depends on the tools that are used for building the project. Usually, the programmer has to visit the library's official website, download and unpack the compressed library folder into the project, after which the actual integration only begins. Library files might require changes to work with the programmer's build tools, after which the build tool must be instructed on how to compile and link the library, leading to a long and tedious process.

Programmers trying to learn OpenGL often struggle to compile the project they are working on and ask for help on various platforms. These programmers are often met with rude answers, as popular forums are filled with similar questions which do not seem to stop appearing anytime soon.[1] Programmers with multiple OpenGL projects use one of their previous projects as a template for a quick start. However, if they want to try a new library, they must go through the whole process nevertheless.

There are attempts to decrease the time spent on configuring the compilation process in the form of templates. These templates are publicly available OpenGL project repositories containing instructions on installing the required libraries and compiling the project. Some of the templates contain the libraries, so the programmer does not have to install the libraries but can immediately compile the project. If the libraries are included in the project, this solution significantly decreases the time needed to start the development. However, it has an apparent limitation; libraries can not be easily added or removed. As of April 2022, there is still no better solution for decreasing the complexity of the OpenGL project compilation than the aforementioned templates.

Creating a tool that would help with configuring OpenGL projects must start with clarifying what is expected from such a tool. In order to be a unique solution, it must allow the user to select which libraries should be included in the project. Users will be able to compile the project with selected libraries in only a few simple steps. These steps will be uniform for any combination of selected libraries the user might choose from the list of available libraries. The same project must be compilable across various development environments, meaning it should

[1] Few of these questions are illustrated in Image A.1 in the Appendix.

be fully IDE agnostic, work on Linux and Windows, and be able to utilize various compilers. This will ensure that the target audience is not limited to a specific development environment but also enable the collaboration of developers using different environments on the same project.

If the objectives mentioned above are fulfilled, glub will be a unique tool for creating OpenGL projects. However, it must offer a wide range of libraries to make it an excellent option that programmers would enjoy using. Considering the estimated amount of work that library integration will require and the constant releases of new library versions, a single person can not maintain and expand the list of available libraries for free. One option would be to offer glub as a paid tool, but among other things, companies that would be able to afford such a tool usually have their own and more advanced toolchain. The other option would be to distribute glub as an open-source project. If people enjoy using it and it would save their time, they might be more willing to implement a newer version of their favorite library into glub.

Since no advanced tool for simplifying OpenGL project creation with libraries was previously created, viable tools for compilation will need to be thoroughly analyzed. The selected technology that glub will be built on must be able to fulfill all the previously defined objectives. There will have to be a well-thought design that would enable relatively simple library integrations, so volunteers will not struggle with contributing. Even after thorough analysis, excellent design, and implementation is done, it still can not be decided that the project was successful. The success of glub will be decided in the months following its release. It will need constant maintenance, so it is up to the effort of contributors to decide if glub will be a successful and growing tool or disappear as an impractical idea.

Chapter 2
Analysis

This chapter will analyze and compare suitable tools for building glub. Since glub can be separated into three independent parts, tools will be analyzed for each one of these parts separately.

The central part is the build tool, which will compile the OpenGL project. Which build tool will be used must be decided first, not just because of the critical role that it plays in the scope of glub, but also because tools used in other parts of glub will depend on this decision.

In order to keep the list of available libraries up to date, a service providing such a list alongside instructions for building these libraries will be required. Multiple concepts can be used to build such backend services; however, only REST API technologies will be considered and analyzed. REST API is a popular and straightforward concept that provides all the necessary functionality to be a suitable option for all the build tools that will be analyzed.

The third part of the glub is the interface that users will use to request data necessary to build the OpenGL project. The technology used to build such an interface will heavily depend on which build tool was chosen. If glub is integrated into IDE as a plugin, technologies used to develop plugins for the particular IDE will have to be used. However, the ideal option would be to create a website so the target audience is not limited to a single IDE or platform.

It is expected that glub will provide OpenGL project generation containing multiple libraries with various versions. Testing combinations of libraries and versions must be automated. This service will not be a part of glub, but glub will be requesting test results for specific library configurations from this service. The technology used for building the test environment depends entirely on the selected build tool.

Based on selected technologies and expectations from a tool such as glub described in the introduction, specific functional and non-functional requirements can be established. The last section of this chapter will be dedicated to listing and describing these requirements.

2.1 OpenGL build tools

The main focus of this thesis is to facilitate the compilation of source code with specific properties. There are multiple ways of managing source code compilation, but choosing the right tool is crucial as it must retain certain qualities that will impact all aspects of the final product.

One of the essential qualities of the tool that will be used for the compilation is **cross-compatibility**. If the Linux or Windows system that glub is run on has OpenGL support and a C++ compiler installed, the build tool must be able to compile the source code with the available compiler on both platforms. If the tool can compile a project on both platforms, it will receive 2 points. One point will be added when integrating cross-compilation into glub will be reasonably simple, meaning estimated implementation will take no longer than 4 hours. If

no additional configuration is needed from the user, when migrating the project between the platforms, another point will be added. The fifth point can be gained if the tool can compile the project with various compilers.

Since glub will rely heavily on contributions from volunteers, the selected build tool must have a certain level of **popularity**. If the build tool would not be a commonly used technology, volunteers might get discouraged from contributing because of their unfamiliarity with the tool. Build tools will receive points for popularity based on multiple factors, but mainly on how often the technology is used in the OpenGL and C++ development.

OpenGL projects are almost always making use of multiple existing libraries, these libraries can be usually downloaded as a zip file from their website, but many of them have a Github repository as well. When choosing the build tool, ease of **integrating libraries** must be considered, as they play an important role in OpenGL development.

The end user's perspective must be considered as well, there is no point in using glub if it will be hard to use and the generation of the project will take a lot of time. The build tool is influencing how the **distribution** of glub and the generated project will look like. Depending on how difficult it is to install and start using the tool, up to 2 points can be added to the score. If the tool is standalone and does not force the user to use a particular development environment, 1 point will be added. When new libraries are integrated into glub, these updates must be easily and quickly distributed; another 2 points can be added for this quality.

2.1.1 IDE build tools

IDEs often provide their own build tools (e.g., Microsoft's MSVC in Visual Studio), in such cases, glub could be directly integrated into an IDE as a plugin. However, glub will be bound by the properties of the IDE it is integrated into, so selecting the right IDE would be crucial.

Cross-compatibility

If the right IDE is selected, glub does not have to consider different platforms and compilers. However, not all IDEs are compatible with Windows and Linux; some do not have a compiler bundled in, which narrows the options of IDEs to pick from and relies on the user to provide a compiler supported by IDE. Estimating the time of cross-compatibility implementation might be a bit difficult, but IDEs like CLion or Visual Studio Code seem to fulfill the cross-compatibility criteria for all 5 points.

Popularity

Various IDEs use various technologies for developing plugins, for example Intellij uses Java and Kotlin [2], but Visual Studio Code uses Typescript [1]. Despite the popularity of these IDEs and the development of many plugins for them, the technologies used to build the plugins are not commonly used in the OpenGL and C++ development, so the score for popularity is 2.

Library integration

It is possible to retrieve library files and configure how they should be linked; however, this would require quite a bit of additional code for every library. Integrating new libraries should still be pretty straightforward, which deserves a score of 3.

Distribution

IDEs have their own plugin marketplace, on which glub can be published and regularly updated. The project would be generated on the user's system, so no additional downloads apart from the libraries would be needed. IDE plugin is clearly an excellent option for simple

[1] JavaScript can be used for Visual Studio Code extension development as well, however TypeScript is recommended [3]

distribution of glub itself and the projects it generates, but it will be limited only to the IDE that glub is built for; therefore, the score is only 3.

2.1.2 GNU Make

GNU Make is a tool that controls the generation of the executable from source code. The whole generation can be configured in *makefile*, this is quite beneficial for glub as it would only need to generate the correct *makefile* instead of configuring the whole project when used in an IDE. [2]

Cross-compatibility

GNU Make itself can be run on Windows and Linux; however, glub will need to perform certain tasks differently on each platform. When using GNU Make, the number of such tasks seems substantial, but it can be done in a way that the user will not be required to take any additional actions when migrating between platforms. Because it will take some effort to implement the cross-compatibility when using GNU Make, the score is 4.

Popularity

GNU Make is not that easy to use, but people that work with C++ most likely used GNU Make at some point or at the very least heard of it. Since GNU Make is a well-known tool in the targeted community, the score for popularity is 4.

Library integration

Libraries can be cloned from their Github repositories; most of them include *makefile* which can be used to link and build the library. However, properly building and linking each library with GNU Make requires additional non-trivial configurations to be made; that is why the score is 3.

Distribution

In order to compile a project with GNU Make, only *makefile* is needed. It is possible to distribute glub as an IDE plugin, but the same generation process can be used for various IDEs. Since only a single file is required to configure a project's compilation process, this file can also be generated by a web service and downloaded by the user. Generating the file in a web service would also ensure that the user always gets the latest version generated by glub without taking any additional action. Using GNU Make provides a lot of flexibility in distribution, so the score is 5.

2.1.3 CMake

CMake is a bundle of tools that uses *CMakeLists* to control the compilation of projects. It is not only able to compile a project, but it also generates native *makefile* or even a Visual Studio solution. It is designed to support complex directory hierarchies and applications dependent on several libraries. [3]

Cross-compatibility

The same *CMakeLists* file can generate an executable on both Linux and Windows. Also, it is simple to configure platform or compiler-specific tasks. It will take minimal effort to implement cross-compatibility, and the user will not have to worry about any additional work when migrating the project between platforms. The score for cross-compatibility is 5.

[2] All the information about GNU Make from section 2.1.2 can be found in the GNU Make manual on the official GNU Operating System website [4]

[3] All the information about CMake from section 2.1.3 can be found on the CMake website [5]

Popularity

CMake is a prevalent tool in the community of OpenGL and C++ enthusiasts. It is well integrated into popular IDEs. Most members of the community interested in using glub should have some experience with CMake; therefore, the score is 4.

Library integration

Similar to GNU Make, integrating a library would require a clone from Github; however, most of the libraries include *CMakeLists* file, so a library can be integrated just by providing basic information about the library and library-specific tasks that should be performed. Libraries can be integrated relatively easily with CMake if they contain the required configuration file, so the score is 4.

Distribution

The CMake version of glub can be distributed similarly as if it would be built with GNU Make. The project's build configuration can be generated inside an IDE plugin and downloaded from a web service. Leaving *CMakeLists* generation up to a web service also ensures the latest version of the file is provided to the user. Because of the distribution flexibility, it is possible to cover a large number of developers using various tools in their workflow, and the final score is 5.

2.1.4 Custom build tool

A whole new build tool can be developed to satisfy all the requirements. This tool can utilize commands and APIs available on each platform. Apart from providing all the needed functionality, it will require significant work to keep glub up-to-date with all supported systems.

Cross-compatibility

Since everything will be implemented using low-level platform-specific APIs, glub will not be bound by the availability of 3rd-party software on certain platforms. Since the implementation and maintenance of cross-compatibility will take a significant amount of effort, one point will be subtracted from the overall score instead of not adding a point for this quality. This solution will fulfill all the other qualities, so after subtraction, the score is 3.

Popularity

In this option, glub will be an independent software, which might discourage developers from using it, as it is not integrated into the development tools that they are using. To enable the community to contribute to glub, it can be written in C++. Despite this, it will still have to work with various complicated APIs that the targeted community is unfamiliar with. Combining the separation of glub from the usual workflow of developers with the difficulty of creating contributions, the score is 2.

Library integration

Each library will require a significant amount of code to be integrated into glub. This code might change with every new release of a library. Integrating and maintaining libraries will require considerable work, which lowers the score to 1.

Distribution

As a standalone software, glub will be available to download and install on its own website, forcing users to take additional steps before using the tool itself. Whenever the generation of the project is updated, locally installed glub would have to be manually updated as well. The advantage of creating a new build tool would be the generation of the project, as it can be entirely

generated directly into a folder the user selected. However, this feature does not compensate for the many downsides of the glub distribution, and the score is 1.

2.1.5 Overview

From Table 2.1, it is clear that building a custom tool is not suitable from every considered aspect. Integrating glub into an IDE is a viable option, as glub would be directly integrated into the users' workflow. This integration can also be achieved by using GNU Make or CMake, as the configuration file will be provided by a web service, and an IDE plugin can also request it. That is why the most suitable tools are GNU Make and CMake. However, CMake is a slightly more viable tool, as library integration and cross-compatibility can be achieved more easily than GNU Make.

Table 2.1 Build tools comparison

Build tool	Cross-compatibility	Popularity	Library integration	Distribution
IDE build tools	●●●●●	●●	●●●	●●●
GNU Make	●●●●	●●●●	●●●	●●●●●
CMake	●●●●●	●●●●	●●●●	●●●●●
Custom tool	●●●	●●	●	●

2.2 REST API technologies

In order to build the OpenGL project with CMake, only specific text files are required. These text files will be generated and served by a REST API.

Since the API will be performing only elementary tasks, it should also be built with simple technology. That is why the most important quality for the backend is **simplicity** of the used technology. The code must be readable and with minimal obfuscation; since the size of the API will be small, keeping related functionalities close together in the source code will be helpful.

Some updates to add new functionalities might be required, and used technology must provide simple **extensibility** of the API. Adding a new endpoint or a section into the generated files must take as few lines of code as possible.

Size of the minimal working REST API of the technology is also essential. Since glub will not use advanced functionalities of the technologies, it is counter-productive to store and compile big applications.

2.2.1 PHP & Laravel

PHP is a well-established scripting language in web development. By combining it with the Laravel framework, it becomes a powerful tool for developing web APIs. Laravel provides its own packaging system and helps structure the project into an MVC application but still keeps the simplicity, making it a viable option for a smaller web API. [4]

Simplicity

[4] All the information about Laravel from section 2.2.1 can be found in the documentation of Laravel[6]

Laravel follows an MVC design pattern, which might be a disadvantage in such a small application as it might cause slight obfuscation. Starting the development with Laravel is not that simple, it requires a considerable amount of dependencies to be installed apart from PHP and Laravel itself. If set up correctly, this technology might be simple enough, but it still brings unnecessary complexity, and the score is 2.

Extensibility

Laravel offers reasonable opportunities of extending the API, thanks to the MVC design pattern. Adding a new endpoint would require registering it in the router and referencing a new Controller. For illustration, an example controller is shown in Code listing 1. For an MVC design pattern, the amount of work needed to add an endpoint is reasonable; however, for the size of API that glub requires, it might be a bit long, so the score is 2.

```php
<?php

namespace App\Http\Controllers\API;

use App\Http\Controllers\Controller;
use Request;

class CMakeController extends Controller {
  public function getCMake() {
    return response()->json(
      [ 'cmake' => 'Generated CMake contents' ]
    );
  }
}
```

Code listing 1 Laravel controller

Size

The size of a basic API itself is 40 MB in size, but in order to start developing in PHP using Laravel, a considerable amount of packages need to be installed beforehand. This is not very suitable for such a small API, and the score is 2.

2.2.2 Node.js & Express

Node.js is a JavaScript runtime environment that allows building web APIs with JavaScript. JavaScript itself is a highly readable language, which in combination with the Express framework, makes building APIs a very fast process.[5]

Simplicity

Starting to build an API with Node.js and Express takes a few minutes. The only prerequisites are installing Node.js itself and a package manager like npm, through which Express can be installed. Setting up a project using Node.js is reasonably simple, it also uses easy-to-understand scripting language, so the score is 4.

Extensibility

[5] All the information about Express from section 2.2.2 can be found on the official Express website[7]

Any required changes that need to be made will be straightforward and will not take any more work than necessary. Adding a new endpoint to the API can be accomplished by adding only a few lines of code, as can be seen in Code listing 2. Complex extensions to the API will require proper designs to be made before being implemented, as Express does not enforce any kind of project structure, and it might lead to obfuscation. For a simple API that glub will require extensibility is sufficient and straightforward, so the score is 4.

```
1  app.get('/cmake', (req, res) => {
2    res.send('Generated CMake contents');
3  });
```

Code listing 2 Node.js endpoint

Size

The size of all the components needed to develop an API with Node.js and Express is minimal. Node.js is a small program of 20 MB, and the size of a basic API with Express is 2 MB. Necessary tools do not take any unnecessary space, so the score is 5.

2.2.3 Deno & oak

Deno is a new tool heavily inspired by Node.js, Ryan Dahl [6] is public about addressing issues of Node.js in Deno. Compared to Node.js, Deno is written in Rust instead of C, but mainly it does not uses a package manager. Dependencies are imported by providing a URL. Also, it does not require any additional packages to support TypeScript. Framework oak further simplifies the API development by providing a quick and easy way to set up an HTTP server.[7]

Simplicity

Installing Deno is enough to start working on the API. Dependencies like oak will be included via URLs in the source code. In many ways, TypeScript is similar to JavaScript, the main difference is that TypeScript allows specifying data types of variables, which is quite helpful when dealing with objects provided by the user. Creating a Deno project could not be simpler; typescript is also as readable as JavaScript but offers more flexibility, so the score is 5.

Extensibility

Deno will face similar issues with extensibility as Node.js, mainly that the future work on extensions will depend on the current design and quality of code. Deno might have an advantage by using data types, as APIs are working with a considerable amount of variables with different data types, so having them well defined should make future updates to the API easier. Extending the API with a new endpoint takes a similar effort as in Node.js, which is demonstrated in Code listing 3; based on this, the score is 4.

```
1  router.get('/cmake', ({ response }) => {
2    response.body = 'Generated CMake contents';
3  });
```

Code listing 3 Deno endpoint

[6] Developer of Node.js and Deno.
[7] All the information about Deno from section 2.2.3 can be found on the official Deno website[8]

Size

Deno itself takes up 20 MB, all the required dependencies are cached and not directly included in the project, so the size of the basic API is only a few KB. Since there are no unnecessary components that would take space in the project, the score is 5.

2.2.4 Python & Django

Another high-level readable popular language is Python, as a general-purpose language, it enables building a web API. Django is an open-source web framework that helps with fast and straightforward web application development. Django is using MVT architecture, which will not be utilized in glub API, as it does not require any access to a database.[8]

Simplicity

Python is already a great programming language for small and simple projects, which is the case of glub API. Web framework Django simplifies the building of web APIs even further. However, first Python must be installed in the development environment, after which Django can be manually installed, which slightly slows down preparations for the development compared to Deno, so the score is 4.

Extensibility

Python is a highly readable language, which is always an advantage when adding new features. Django defines a structure of the API source code, adding a new endpoint would require registering it in the list of URLs and creating a new view for the endpoint, which is demonstrated in the Code listing 4. Implementing a new endpoint seems fairly simple, but it requires a bit of redundant work considering the size of the API, so the score is 3.

```
1  import json
2  from importlib import import_module
3  from django.http import HttpResponse
4
5  def index(request):
6    return HttpResponse(json.dumps({\
7      "cmake": "Generated CMake contents"\
8    }), content_type="application/json")
```

Code listing 4 Django endpoint

Size

The size of an API build with Python and Django is minimal; however, installing all the tools required will take 100 MB, which is still a reasonable amount, so the score is 4.

2.2.5 Ruby on Rails

High-level, general-purpose languages are usually readable and easy to work with, which benefits the simple glub API. Ruby belongs to the group of languages with these properties; therefore, it should not be left out. Ruby on Rails is a web application framework that utilizes the MVC

[8] All the information about Django from section 2.2.4 can be found on the official Django website[9]

pattern to help with building the front and back end of application.[9]

Simplicity

Even though Ruby takes a different take on programming language design, the simplicity and ease of use in both languages are comparable. However, Ruby on Rails is optimal for more extensive applications and brings a bit of unnecessary complexity into very small applications, so the score for simplicity is 3.

Extensibility

Since glub will utilize only the controllers from the MVC or MVT patterns, the extensibility of Ruby on Rails is the same as with Python and Django. An example of a simple endpoint can be seen in the Code listing 5, and the score is also 3 for the same reasons as Python and Django.

```
1  class Api::CMakeController < ApplicationController
2
3    def index
4      render json: { "cmake": "Generated CMake contents" }
5    end
```

Code listing 5 Ruby on Rails endpoint

Size

In order to have working Ruby on Rails installed on the system alongside Ruby itself, it will take over 700 MB, which might cause problems with some very limited API hosting services, so the score is 2.

2.2.6 Overview

The glub API will be minimalistic, only serving generated text files. Based on the comparison in Table 2.2, the most suitable tools are Node.js and Deno. Node.js is already a widely used tool that has been proven to work well, but Deno offers some new out-of-the-box features that work well, especially on small projects like glub. API used by glub will be developed in Deno, as it has more to offer than Node.js, and minimalistic APIs like glub do not need to be built on technology that has been around for decades.

Table 2.2 REST API comparison

Technology	Simplicity	Extensibility	Size
PHP & Laravel	●●	●●	●●
Node.js & Express	●●●●	●●●●	●●●●●
Deno & oak	●●●●●	●●●●	●●●●●
Python & Django	●●●●	●●●	●●●●
Ruby on Rails	●●●	●●●	●●

[9] All the information about Ruby on Rails from section 2.2.5 can be found in the Ruby on Rails documentation[10]

2.3 REST API hosting

Deno is a relatively new technology, and not many hosting services support it yet. That is why it must be easy to **deploy** a Deno application on the selected hosting service. It is not expected that glub will be generating significant traffic, so the **pricing** of the hosting service is a critical parameter. Since for version control GitHub will be used, a possibility of deploying the API directly from GitHub to the hosting service (**GitHub integration**) would be helpful.

2.3.1 Fly.io

In order to deploy the Deno application onto Fly.io, it requires a command-line tool to be installed and a configuration file included in the application folder. The deployment itself is launched from the command line tool, the steps necessary to deploy are simple, but the initial configuration requires some additional actions. Hence, the score for **ease of deployment** is 3.

Fly.io offers a limited free usage of the hosting service, in which 160 GB of bandwidth can be used each month, 3 GB of permanent storage are available, and 3 shared CPUs can be used for over 2 000 hours every month. These limits are more than enough for API that will glub use, so the score for **pricing** is 5.

Deno application stored on GitHub can be deployed onto Fly.io by using GitHub Actions. A configuration file will be added to the GitHub repository, including commands for deploying the application onto Fly.io. Since **GitHub integration** is achievable by providing one configuration file, the score is 4.[10]

2.3.2 Clever Cloud

Deployment onto Clever Cloud looks very similar to deployment onto Fly.io. Installation of Clever-tools CLI will be required, and a configuration file must be provided. The score for **ease of deployment** is, therefore, the same - 3.

Clever Cloud does not offer free usage of its services. The cheapest service Clever Cloud offers costs 4.50 € per month. This service includes 256 MB of RAM and 1 shared CPU. These prices are pretty high considering provided resources, so the score for **pricing** is 2.

Via the Clever-tools CLI, an application can be created. This application is then used to track the repository and automatically deploy it. Since only set up with Clever-tools is required to automatically deploy from GitHub, the score for **GitHub integration** is 4.[11]

2.3.3 Deno Deploy

Deno Deploy is a hosting service developed by the team that is developing Deno itself. This is a great advantage, as Deno is a still-growing technology, in case of any changes, hosting will be compatible with Deno since they will be aware of the changes. If the repository is hosted on GitHub, the deployment can be done through the Deno Deploy website with only a few clicks. Therefore the score for **ease of deployment** is 5, since it can not be any simpler.

Deno Deploy is currently in its initial public beta release and offers free but limited deployment. There can be a maximum of 1 000 requests per minute, only 512 MB of RAM is available, and the CPU is available only for 50 ms per request. These limits are quite strict, but still more than what glub API would require, so the score is 4.

GitHub repository can be deployed onto Deno Deploy with only a few clicks. However, if any further building steps are required, it is possible to include a GitHub Action and perform these

[10] All the information about Fly.io from section 2.3.1 can be found on the Fly.io website[11]
[11] All the information about Clever Cloud from section 2.3.2 can be found on the Clever Cloud website[12]

steps. Deno Deploy works well with GitHub, it is simple to use but not limited in functionality. the score for **GitHub integration** is 5.[12]

2.3.4 Overview

Fly.io offers many resources for free, but Deno Deploy is easier to use and has a certain guarantee of working well with Deno. For the purposes of glub API, Deno Deploy is the ideal candidate. In case the traffic to the API increases, hosting can be switched to Fly.io at any time. The comparison of API hosting services can be found in Table 2.3.

Table 2.3 API hosting comparison

Hosting service	Deployment	Pricing	GitHub integration
Fly.io	●●●	●●●●●	●●●●
Clever Cloud	●●●	●●	●●●●
Deno Deploy	●●●●●	●●●●	●●●●●

2.4 Test environment

Verification of various library configurations would be a lengthy manual process, considering there are constantly newer versions of libraries being released. That is why it is vital that testing for library compatibility is an automated process. The parameters that need to be tested are the success of the build process and the ability to execute the output of the compilation on Linux, and on Windows, the visual output of the project does not have to be tested. This can be done with one of the continuous integration services. However, the options are limited, as many of them do not provide an environment with access to a GPU. In some cases, this problem can be overcome by using an implementation of OpenGL that requires only a CPU to run, like Mesa[14]. Since most likely additional work will be required to test OpenGL programs, the important quality of the test environment is **simplicity** of preparing the environment to run the tests. In order to build a project, the test environment must download all the necessary libraries and compile them, which can take a significant amount of time. That is why **price** should be considered as well. Compiling the libraries and the project itself will take a significant amount of time, which will depend heavily on the performance of used **CPU**. CMake supports multithreaded compilation, which might decrease the build time of the project mainly if it consists of multiple independent parts like libraries. So not only the frequency at which the CPU operates should be considered but also the count of the CPU cores provided.

2.4.1 GitHub Actions

Since glub API will be open-sourced and hosted on GitHub, it might be convenient to have the source code and tests in one place. GitHub Actions can be run on Linux and Windows; however, GPU is not accessible. As part of the test process, Mesa has to be installed before building the project, which will increase the usage of resources. Apart from this complication setting up a GitHub Action is a simple process, so the score for **simplicity** is 4.

[12] All the information about Deno Deploy from section 2.3.3 can be found on the Deno Deploy website[13]

GitHub offers a plan for students, which includes 3 000 execution minutes per month. The initial amount of tests needed to be executed is high, but after they are processed, this limit should be sufficient, so the score for **price** is 5.

GitHub-hosted runners provide a 2-core CPU, for which 2 points will be added to the score. However, no more information on this topic is available in the documentation. In January 2022, Magno Logan published an article in which he analyzed the security of GitHub runners. From his Ubuntu runner reconnaissance, it seems the runner uses an Intel Xeon Platinum CPU running at 2.6 GHz, which will be evaluated only with one point.[15] The total score for **CPU** is 3.[13]

2.4.2 CircleCI

CircleCI provides access to a GPU, but first, CUDA must be installed. The configuration files of CircleCI are no more complicated than the configuration files of GitHub Actions, so the score for **simplicity** is 4.

CircleCI has a free plan offering 6 000 minutes. However, in this plan, access to GPU is not provided. The amount of execution minutes is sufficient, so the score for **price** is 5.

In the free plan, CircleCI provides only 2 vCPUs, so the compilation might take a while. It is hard to estimate the exact performance of the vCPU, but if a rough approximation is set and the performance of a vCPU is similar to the performance of one CPU core, then 2 points can be added to the CPU score. Since details of the used hardware are not available, an average frequency of 2.5 GHz will be assumed, and one point will be added to the **CPU** score, which sums to 3.[14]

2.4.3 VPS

Using VPS to run tests is a viable option, as it is possible to run tests as long as needed. However, setting up the whole environment to run the tests will take a considerable amount of time. Therefore the score for **simplicity** is 2.

Since the tests must be run on Linux and Windows, there is a need for 2 separate VPS, which doubles already fairly high price. The price is usually a few hundred dollars per month. The cheapest services like VPS Mart starts from 45$ per month.[18] There are options of VPS that provide access to a GPU, but for obvious reasons, those costs even more than the VPS without any access to GPU. VPS is an expensive option, so the score for **price** is 1.

In their cheapest plan, VPS Mart provides an Intel Xeon X3440 CPU with 4 cores running at 2.53 GHz.[18] The CPU frequency will add one point to the score, but 4 CPU cores are a significant advantage, and 3 points will be added, so the score for **CPU** is 4.

2.4.4 Overview

Running a VPS would be a time-consuming and expensive choice. Despite the freedom it provides, it is not a viable option. GitHub Actions and CircleCI provide very similar services. Since there are no significant differences, GitHub Actions can be used as a test environment due to the convenience of keeping it in one place with the source code of glub API.

2.5 Web framework

There are countless web frameworks available. No matter in which one the glub web interface will be written in, there will always be a web framework that is arguably slightly better. Since

[13] All the information about GitHub Actions from section 2.4.1 can be found in the GitHub Actions documentation[16]

[14] All the information about CircleCI from section 2.4.2 can be found in the CircleCI documentation[17]

Table 2.4 Test environment comparison

Test environment	Simplicity	Pricing	CPU
GitHub Actions	●●●●	●●●●●	●●●
CircleCI	●●●●	●●●●●	●●●
VPS	●●	●	●●●●

the glub will require only a small website with no advanced capabilities like user registration, a suitable web framework should provide fast and simple website creation. One such web framework is Svelte, it does not enforce any complex project structure, and thanks to its component system, it is modular and easily extensible.[15]

Parts of the website that create one logical component are kept in a single file that consists of an HTML template, a style that is used, and a script that will be executed. An example of such component structure can be seen in the Code listing 6.

```
1  <script>
2  // logic goes here
3  </script>
4
5  <!-- markup (zero or more items) goes here -->
6
7  <style>
8  /* styles go here */
9  </style>
```

Code listing 6 Svelte component example

Each component can be used as an HTML tag anywhere on the website. Apart from great modularity, Svelte makes good use of logic-less templates inside the HTML that will greatly simplify the presentation of data gathered from the glub API. Despite being a fairly new web framework, it has readable and concise documentation, but more importantly, it is effortless to work with. Alongside other good qualities of Svelte, using it for the development of the glub website will be quite efficient.

2.6 Functional requirements

2.6.1 Backend

F1.1 Provide library metadata

Information about all the included libraries can be requested, and the response will include a list of available libraries. Each library in the list will provide metadata about the library, like the category it belongs to and available versions of the library.

F1.2 CMake file generation

The backend will be capable of generating CMake file needed to build a project based on the configuration provided in the request. The request will have to contain libraries and their

[15] All the information about Svelte from section 2.5 can be found in the Svelte documentation[19]

versions that will be included in the project, but also additional project metadata like the name of the project, path to source files, and more. Based on this request, the backend will find fragments of code needed to build the library with the specified version and combine these fragments, which can be afterward inserted into the correct place in the *CMakeLists.txt* and returned in the response.

F1.3 Additional C++ files generation

In order to make project generation even quicker, the backend will be providing additional C++ files containing source code to a simple OpenGL showcase program so that it can be immediately built and executed after project generation. C++ files will be generated separately from *CMakeLists.txt* but will work on the same basis as the generation of CMake files, meaning based on the configuration in the request, certain fragments of code will be inserted into specific places in the C++ file templates and returned in a response with a file structure that is expected in the project.

F1.4 Library configuration testing

Tests to ensure CMake and C++ source code fragments are compatible with other libraries will be done on the backend. Based on the list of libraries with their versions, a project will be generated with *CMakeLists.txt* and C++ files. This project will be built, then executed, and the result will be recorded.

F1.5 Test results management

After a library configuration test is performed, the result must be stored. These results can be retrieved by making a request with a list of libraries with their versions. Indicating only if the test was successful or unsuccessful for a given library configuration is insufficient. The backend must provide information on whether the given configuration was already tested or not, if libraries are incompatible by design, or if the source code fragments need to be fixed.

F1.6 New library releases detection

Library repositories need to be periodically checked for new releases. In case a new release is detected, it should be tested. This way, it will be ensured that source code fragments for libraries are always up to date or need to be updated.

2.6.2 Web interface

F2.1 Project configuration

Users must be able to configure the project structure to suit their needs. This includes parameters like the name of the project, version, description, but also in which directory the resources and source code should be located.

F2.2 Library selection

When accessing the web interface, a list of available libraries must be retrieved from the backend and displayed in an organized manner. Users will be able to select and deselect libraries that will be used for project generation.

F2.3 Library version selection

Next to each library in the list, a dropdown with available versions for the given library will be displayed. Users will be able to choose a version of any selected library that will be used for project generation.

F2.4 Library compatibility

After any changes to library or library version selection were made, the result for the selected library configuration will be retrieved and displayed.

F2.5 CMake download

User will be able to request a *CMakeLists.txt* generation. A request to a backend will be made with all the project metadata and library configuration made by the user. Users can either download generated *CMakeLists.txt* or copy its contents into the clipboard.

F2.6 Project zip file download

Similarly to requesting *CMakeLists.txt* generation, the user can request a generation of the whole project. In this case, two requests will be made, one for generating *CMakeLists.txt* and the other for generating the additional C++ files. A zip file will be generated from retrieved files containing all the necessary files in a structure defined by the user.

2.6.3 Community tools

F3.1 User guide

If the user finds glub confusing or has no experience using CMake, a brief user guide will be provided with all the necessary information.

F3.2 Request a library

If the user uses a library but does not find it in the list of available libraries, a request to integrate the library into glub can be made. This request will be submitted in the form of a GitHub issue. In order to be considered a valid request, it will require the user to provide all the necessary information about the library, like the GitHub repository with source code that can be easily built with CMake.

F3.3 Guides for library modifications

Contributors willing to help with improving glub can integrate the new library into glub or modify outdated source code fragments for newer versions of already integrated libraries. Documentation of how all the files are generated is needed, but also to make the work of contributors as easy as possible, simple guides containing examples of common library integration procedures will be included alongside instructions on how the final pull request should look like.

F3.4 Pull request testing

After a contributor submits a pull request, an automatic test should be performed before the source code is reviewed. The test can be launched via GitHub Action and prove that the contributor's implementation integrated the library correctly and is ready for source code review.

F3.5 Bug reporting

Suppose a user cannot compile only a particular library configuration, but the web interface shows the given configuration as successfully tested. In that case, the user can submit a bug report in the form of a GitHub issue. This issue will contain a predefined template, requiring the user to fill out all the necessary information to identify the problem. This information can contain the user's environment, requested library configuration, and build process output.

2.7 Non-functional requirements

NF1 Platform compatibility

Compatibility with Windows and Linux must be established across all parts of glub. Each library configuration test must be performed on Linux and Windows to ensure generated projects

will work on both platforms. User guides will also require separate instructions for both platforms, and similarly, all contribution guidelines will have to take cross-compatibility into account.

NF2 Project generation time

The idea behind glub is time-saving compared to existing solutions for OpenGL project generation. That is why a generation of the project must be quick and efficient, but also because, as an open-source project, there will not be any high-end hardware available, which means processing each request must take less than a second, even on a slower CPU.

NF3 Easy to follow guidelines

User guides must be clear and simple enough so that even a user with no prior experience with CMake must be able to follow them. Similarly, guidelines for contributors must leave no space for ambiguity and specify precisely what is needed from the contributor.

NF4 Technology requirements

Based on the previous analysis of multiple technologies, the most suitable technologies should be used to build glub. As the core build tool, use CMake to build the backend that provides the generation functionality, use Deno for frontend development, use Svelte, and use GitHub's Actions for creating the test environment.

Chapter 3
Design

3.1 Workflows within glub

In order to design the architecture of glub, it must be determined what interactions will be taking place between glub and user, but also inside the various components of the tool. These interactions will be modeled with a UML activity diagram, which allows modeling processes as activities and actions. The activity diagram will reveal the overall workflow, which will later help design smaller glub components.[20]

Activity diagram 3.1 showcases processes that take place when using glub. When an OpenGL programmer that wants to use glub visits the glub website, it will display the website contents and request the latest list of available libraries from the backend. When everything is loaded, the user can configure the project. On any change to selected libraries, the frontend will request library compatibility status from the backend. The response to this request will non-intrusively notify the user if the libraries are compatible.

If the user is satisfied with configurations to the project, generation of only the *CMakeLists.txt* or the whole OpenGL project can be requested. This request will be processed on the backend, which will first generate requested files and send the results in the body of the response. When generating the whole project, the generation of *CMakeLists.txt* can run in parallel with the generation of C++ files. However, before sending back a response, both parallel processes must be finished to insert the generation results into the same response body.

Upon receiving the generated files, the frontend can display them as downloaded. However, in the case of OpenGL project generation, it will first create a zip file and further work only with the zip. After saving the files, the user can either place the *CMakeLists.txt* into the existing OpenGL project or extract the contents of the zip file into the desired directory. The generated project is now ready for compilation, which can be started by running CMake. The output of the compilation will be an executable OpenGL application.

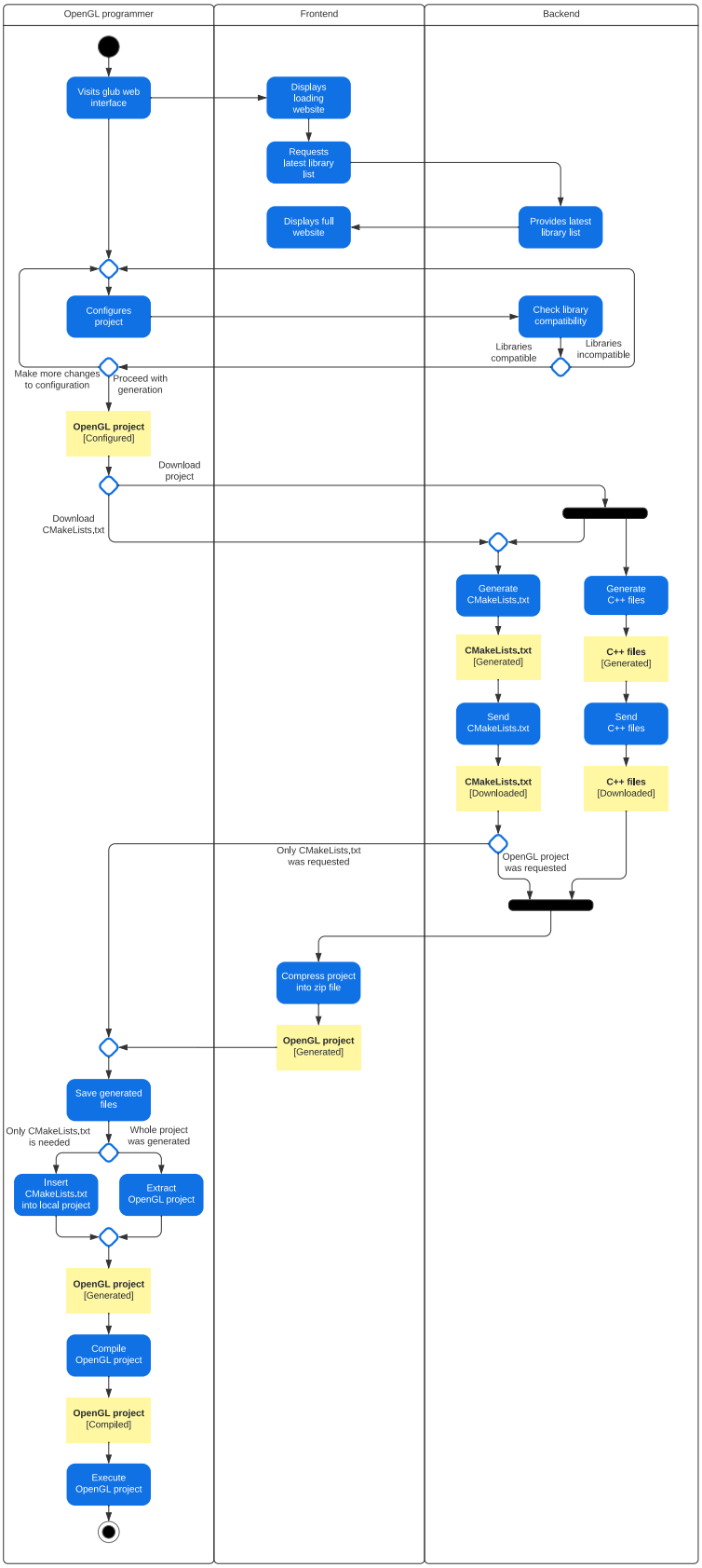

Figure 3.1 Activity diagram for usage of glub

One more activity diagram is needed, this diagram is modeling a very different process related to glub, but since glub will be open-source software, it is a vital process. Activity diagram 3.2 focuses on processes that will ensure smooth and organized integration of new libraries or updates required to support a new release of a library.

When an OpenGL programmer that uses glub notices that one of the commonly used libraries is not available in glub, the user can submit a request, so the library is integrated into glub. This request will require specific information about the library to be filled, mainly a URL pointing to a library repository that can be compiled using CMake.

Before starting the integration of the requested library, the request will be verified. If the library is not relevant to the OpenGL development or is not commonly used, the request can be dismissed. If it has missing or incorrect information, the contributor can request the user to update the request and block the library integration until the information is complete and correct.

After the library request is verified, it can be accepted. A contributor can pick up an accepted library request and integrate the library into glub. If the contributor is done with integrating the library, a pull request can be submitted. The test environment will automatically run tests on the branch with the new library integration. These tests will decide if changes are required or if the library is ready to be reviewed. If the source code also does not require any changes, the library can be merged into the main branch and deployed, after which it will appear in the glub's list of available libraries.

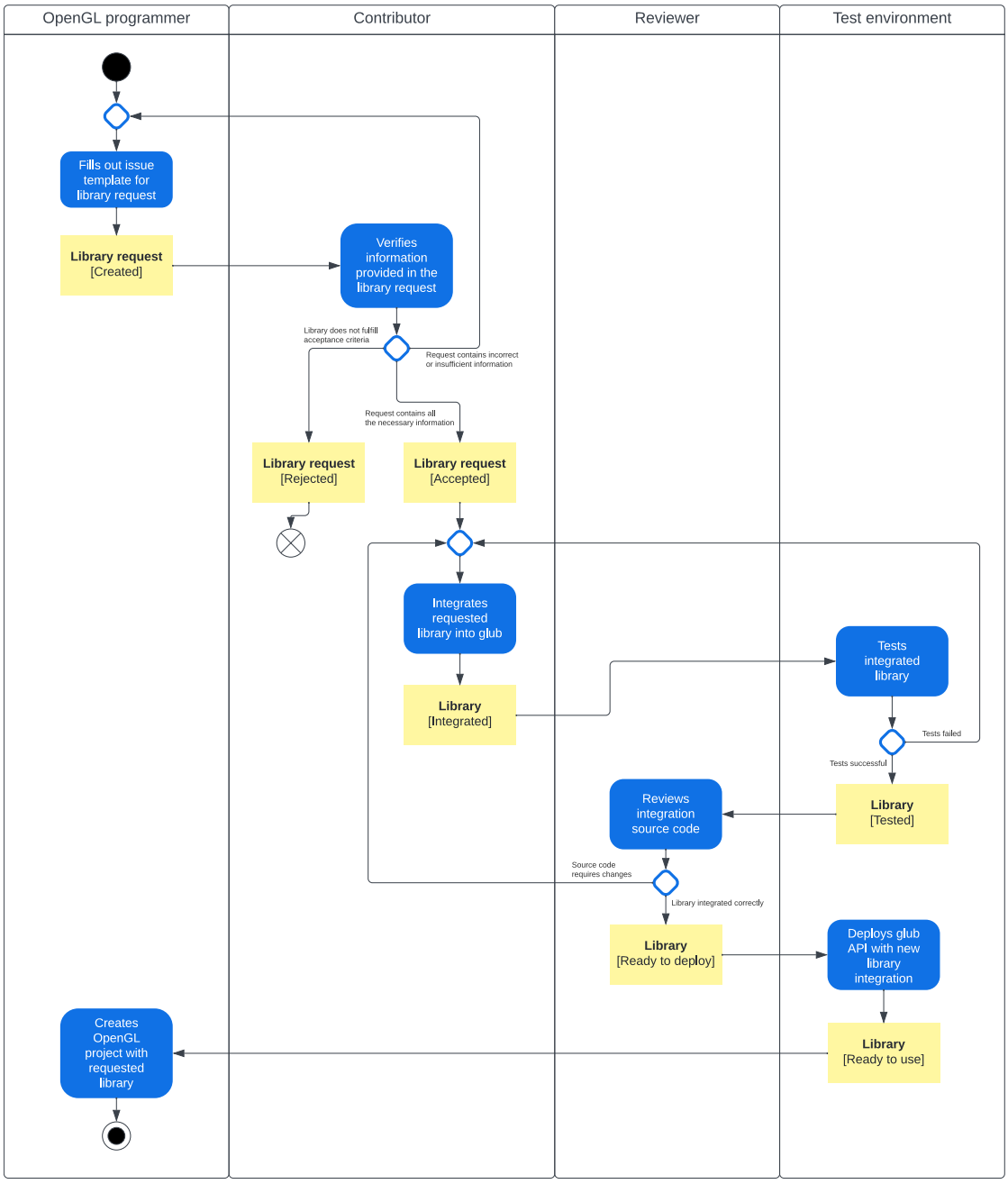

Figure 3.2 Activity diagram for library requests

3.2 Use cases

Section 3.1 provides a detailed overview of processes taking place within glub. Based on these processes, actions available to various actors that can interact with glub can be determined. Actions will comprise of multiple processes from activity diagrams in section 3.1. To concisely represent actions and actors, use case diagrams will be used. Use case diagrams utilize use cases to present actions connected to actors. The simplicity of use case diagrams allows to effortlessly verify that the designed system fulfills all the functional requirements. Each use case will fulfill one or more requirements. If a requirement is not fulfilled by any use case, either the design should be changed to fulfill this requirement, or it should be determined if the requirement is needed for glub to function properly.

First, the use cases of users will be modeled. The users of glub will be OpenGL programmers that wish to create a new OpenGL application. Available use cases for programmers that relate to creating a new project are presented in the Use case diagram 3.3. Programmer can either configure properties of the generated project, request generation of the OpenGL project, or generate the CMake file only. Backend will react to the programmer's actions, before the programmer starts editing the configuration, it will provide the programmer with an up-to-date list of available libraries. If the programmer changes the selection of libraries included in the project, the backend will verify the compatibility of a particular library configuration.

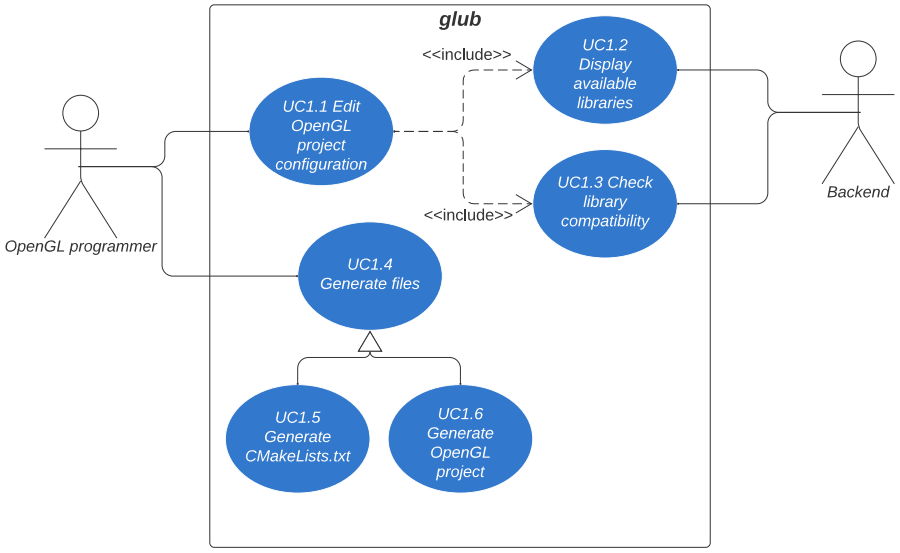

Figure 3.3 Use case diagram for glub usage

Use cases from diagram 3.3 can be used to verify the fulfillment of functional requirements related to project generation. **UC1.1** provides the OpenGL programmer with possibility to configure the OpenGL that will be generated. Configuration of project metadata (**F2.1**), libraries (**F2.2**) and library versions (**2.3**) will be available. OpenGL programmer will be able to generate whole OpenGL project (**UC1.6**) or only the CMake configuration file (**UC1.5**), which fulfills the requirements **F1.2** and **F1.3**. Backend will be providing up-to-date list of available libraries (**UC1.2**) and compatibility verification of selected libraries (**UC1.3**), which will fulfill requirements of providing library metadata (**F1.1**), test results management (**F1.5**) and library compatibility verification (**F2.4**). Whole overview of requirements fulfilled by Use case diagram 3.3 is represented by Table 3.1.

Table 3.1 Functional requirements fulfillment for glub

	Use Case					
Requirement	**UC1.1**	**UC1.2**	**UC1.3**	**UC1.4**	**UC1.5**	**UC1.6**
F1.1		●				
F1.2				●	●	
F1.3				●		●
F1.4						
F1.5			●			
F1.6						
F2.1	●					
F2.2	●					
F2.3	●					
F2.4			●			
F2.5					●	
F2.6						●

The test environment will be testing various project configurations by compiling and executing them. A test can be triggered manually by a contributor to glub's repository or automatically by a new library release as seen in the Use case diagram 3.4. The test result will be stored in the REST API and used by the user interface to inform the user about the compatibility of selected libraries.

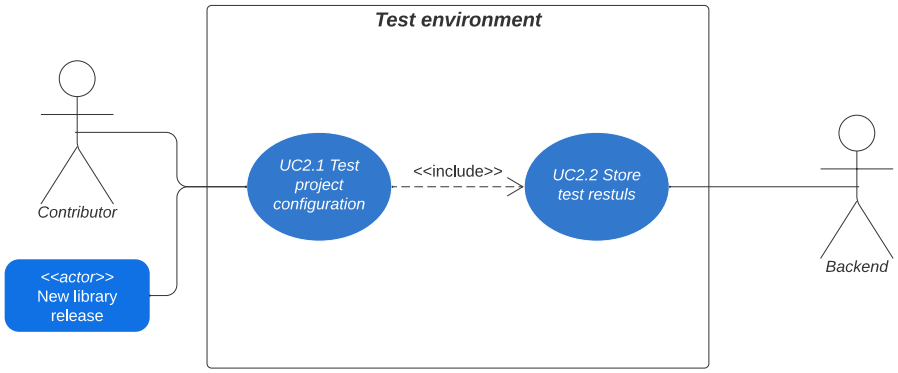

Figure 3.4 Test environment use case diagram

Requirements for executing the tests (**F1.4**, **F1.6**) are fulfilled by Use case **UC2.1**. The management of the test results (**F1.5**) is included within the Use case **2.2** as illustrated in Table 3.2.

Table 3.2 Functional requirements fulfillment for test environment

	Use Case	
Requirement	**UC2.1**	**UC2.2**
F1.1		
F1.2		
F1.3		
F1.4	●	
F1.5		●
F1.6	●	

Apart from generating OpenGL projects, the programmer will have the opportunity to contribute to glub. Library requests can be submitted if the OpenGL programmer uses a library that is not in the list of available libraries. This library request will be verified by a contributor, who will check if the library request contains all the necessary information and if it is appropriate to be integrated into glub. If the contributor accepts a library request, the requested library can be integrated into glub, after which a pull request with this integration can be created. Pull request will be reviewed by a reviewer and tested in the test environment. Similar to a library request, a bug report can be submitted if provided compatibility results do not match the output of the programmer's project compilation. The user interface of glub will be intuitive, but programmers that use glub may be beginners; to help them out, user manuals will be available. For contributors, contribution guidelines will be provided, as library integration is a well-defined process.

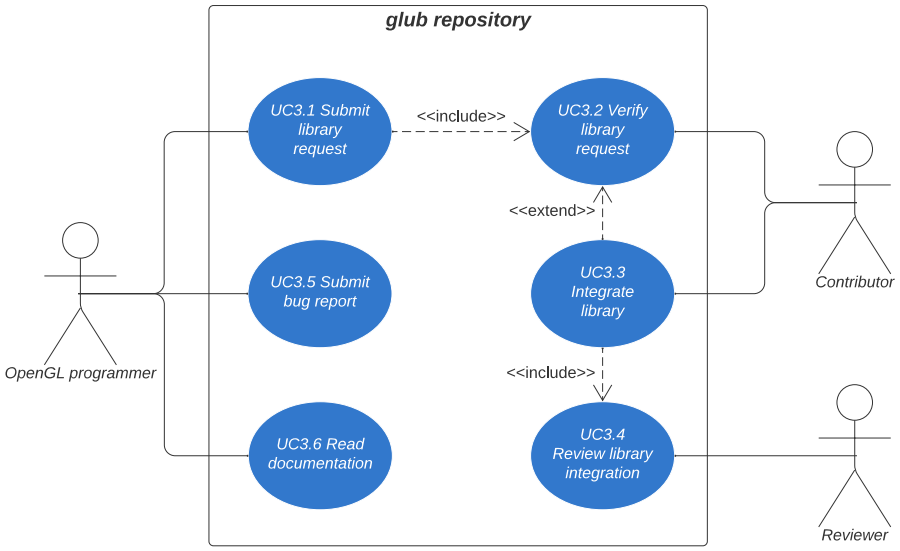

Figure 3.5 Contributions use case diagram

Most of the use cases from diagram 3.5 will be realized within the GitHub repository of glub. Integration of a library (**F3.2**) and bug fix (**F3.5**) can be requested by creating an issue on GitHub (**UC3.1**, **UC3.5**). The request can be verified (**UC3.2**) and the library can be integrated

(**UC3.3**) within the process of a library request (**F3.2**). After the library is integrated, a pull request can be created, which will be reviewed by a reviewer (**UC3.4**), who will review the source code and request a test in the test environment (**F3.4**). If the user or a programmer runs into any problems, documentation will be available (**UC3.6**). User guide (**F3.1**) will demonstrate how the glub should be used, and contribution guidelines will document all the necessary processes within library integrations (**F3.3**).

Table 3.3 Functional requirements fulfillment for glub repository

	Use Case					
Requirement	**UC3.1**	**UC3.2**	**UC3.3**	**UC3.4**	**UC3.5**	**UC3.6**
F3.1						●
F3.2	●	●	●			
F3.3						●
F3.4				●		
F3.5					●	

3.3 Wireframe

Interactions between OpenGL programmer and glub are now clearly defined; based on these requirements, a wireframe for glub's web interface can be constructed. As can be seen in Figure 3.6 the website will be split into multiple sections. Section with number 1 is used to configure project metadata, like name, version, or source code directory. Sections marked with the number 2 are lists with available libraries that can be added or removed from the generated project. Each library entry will also provide a dropdown containing available library versions. Section 3 includes buttons to generate configured project and displays test result of selected libraries. This section also contains useful links, like a link to glub's GitHub repository, bug report form, or library request form. If the user is unsure how to use glub properly, section 4 will provide an interactive guide walking the user through project configuration all the way to compiling the generated project. The user might not be aware that multiple useful libraries used in OpenGL development exist. Section 5 displays a website preview of the last selected library, after clicking the preview, the website will open, and the user can find more details about the library.

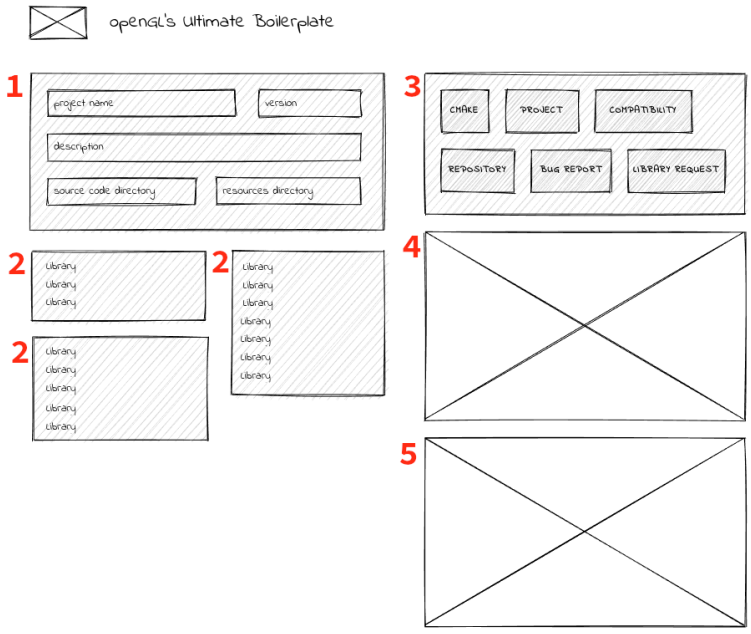

Figure 3.6 Wireframe for web interface

3.4 Backend

The website will be retrieving the required data from REST API endpoints described in Section 3.4.1. Section 3.4.2 defines the structure of data within the REST API and how a response to a request will be formed.

3.4.1 Endpoints

To cover all the functionalities provided by glub, REST API will serve four endpoints. Design proposal for these endpoints will be made by using the tool Swagger, which bases endpoint definitions on OpenAPI specification. Images 3.7, 3.8, 3.9 and 3.10, each contain specification to one of the REST API endpoints.

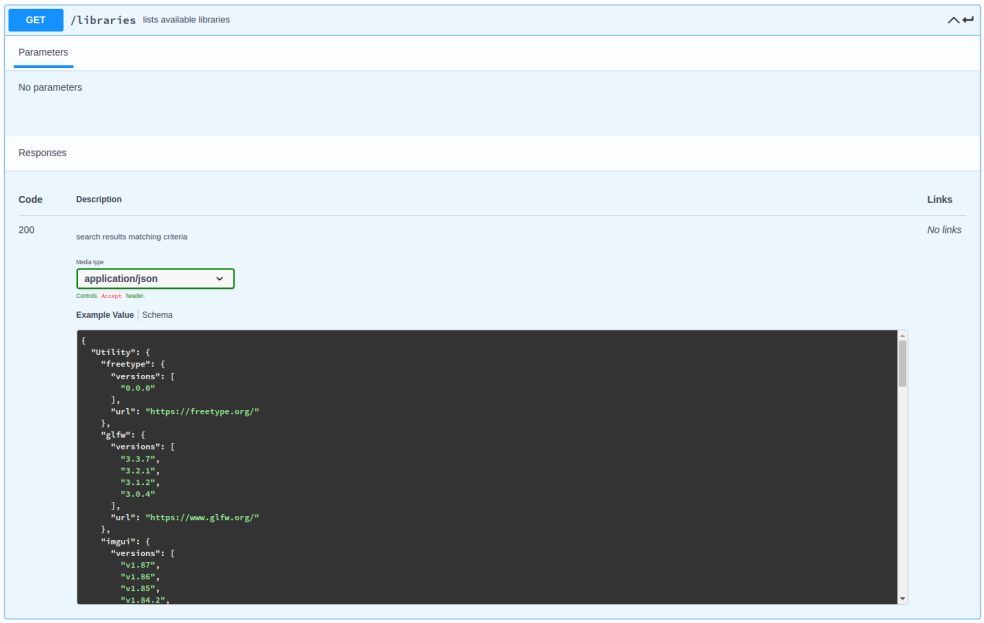

Figure 3.7 Libraries endpoint specification

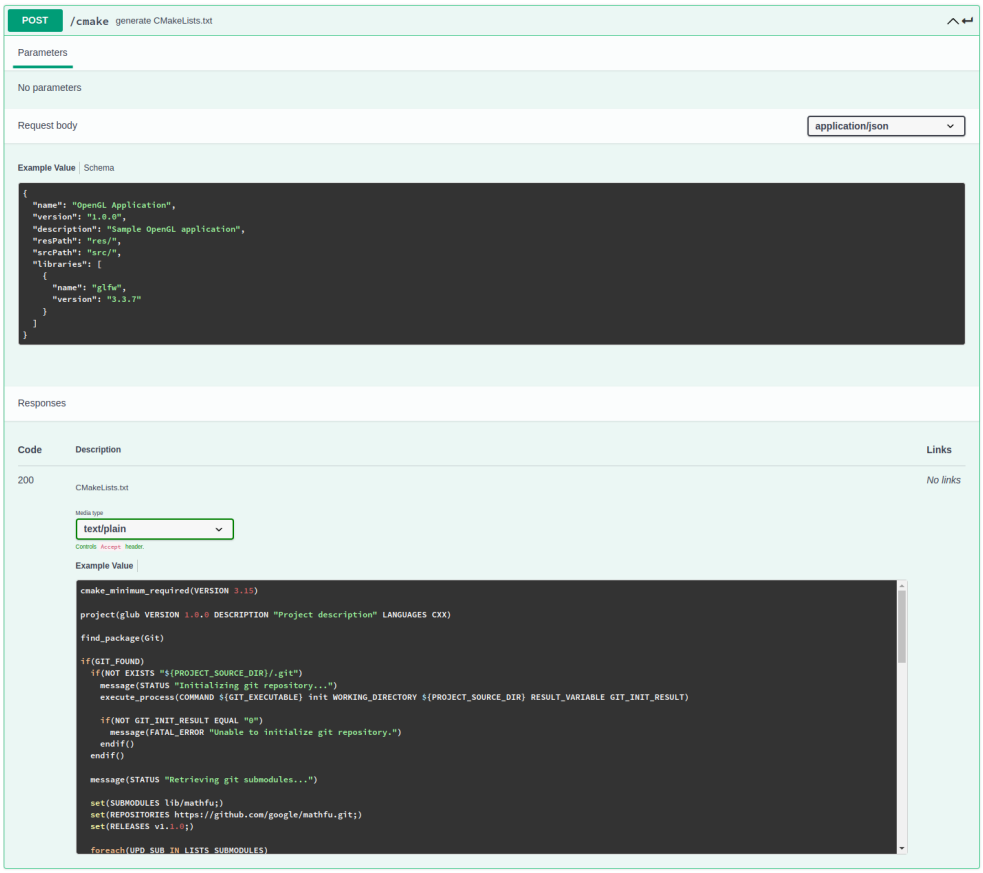

Figure 3.8 CMake endpoint specification

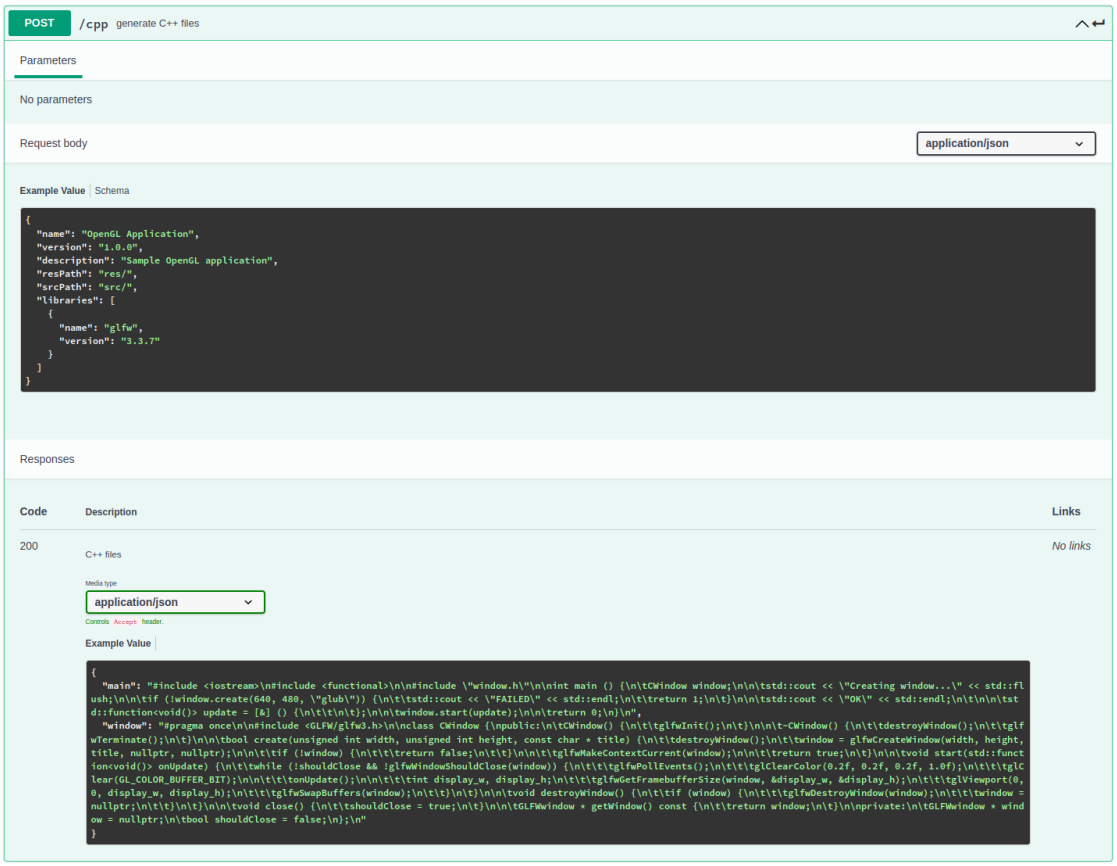

Figure 3.9 C++ endpoint specification

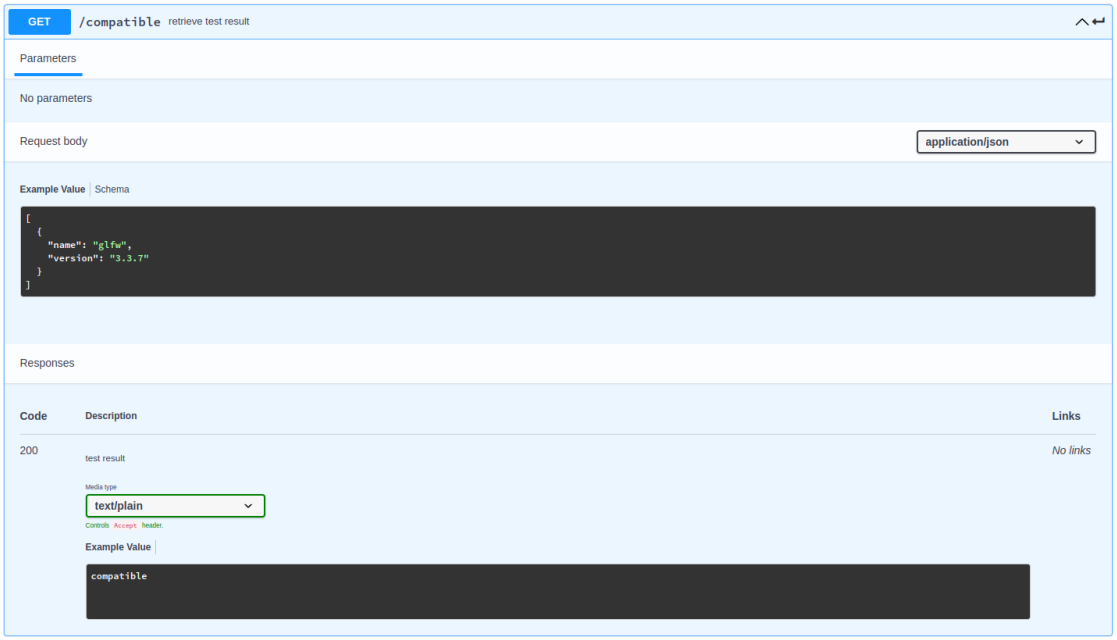

Figure 3.10 Compatible endpoint specification

3.4.2 Domain model

Image 3.11 illustrates the structure of data within the REST API, which will be generating responses from this data. Library compatibility data will be stored elsewhere, but the diagram contains all the data required for OpenGL project generation. The endpoint that provides libraries will utilize function `getLibraries()` in the Category class to group libraries together and pass them to the frontend. Endpoints for generating *CMakeLists.txt* and C++ files will be able to retrieve necessary data based on the version tags of selected libraries in the request by utilizing function `getVersion(tag)` in the Library class and retrieving the code snippets and files from the Version class.

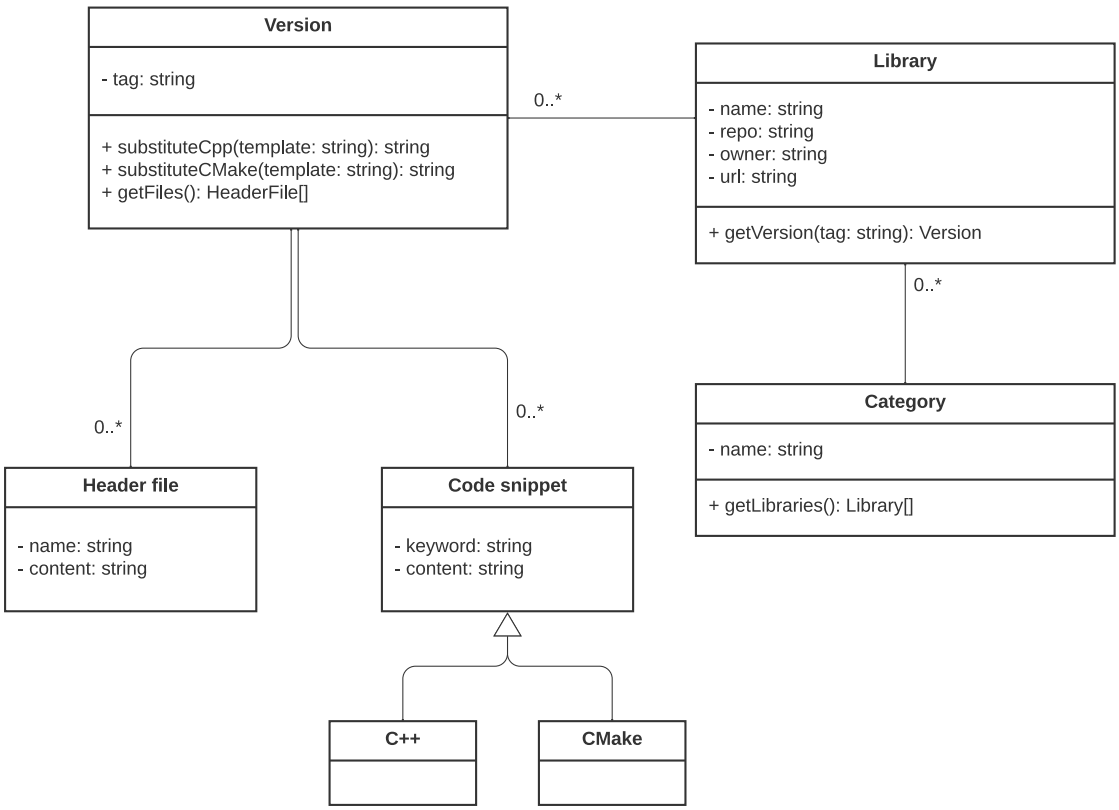

Figure 3.11 REST API domain model

Chapter 4
Implementation

Analysis and design are important processes in software development, but implementation is the process that previous work was leading towards. This chapter will describe the build process of glub using tools selected in analysis based on the designs from the previous chapter. Since no similar tool has been created so far, there is a chance that the realization of glub might not be viable or possible as designed. The output of the implementation process will serve as a proof of concept, and even if glub does not fulfill the requirements to become an effective tool, it might still create a space for further build automatization of projects that use low-level APIs.

4.1 Generation templates

In the analysis, it was decided that CMake will be the most viable tool to distribute the build instructions. Before generating the required textitCMakeLists.txt file, there must be a well-defined template into which each library will be able to insert instructions needed to link and build the library.

First, a plain *CMakeLists.txt* that is able to compile OpenGL applications without any libraries will be created in the section 4.1.1. Alongside the plain *CMakeLists.txt* a basic OpenGL project will be created in the section 4.1.2, it will also use no libraries and created *CMakeLists.txt* will be able to compile it.

The basic templates can then be extended to link and compile with a library. *CMakeLists.txt* including instructions needed to link and compile a project with a library will be described in section 4.1.3. Finally, in the section 4.1.4 files used to compile the project will be analyzed, and the structure that can be used for generating these files will be defined.

4.1.1 Bare *CMakeLists.txt* template

To construct a simple *CMakeLists.txt* that is capable of compiling an OpenGL project without any libraries, commands needed for this minimal configuration will be introduced. After necessary command definitions, commands can be combined into a working *CMakeLists.txt* example.

There are two commands that do not directly specify the compilation or linking process but define the behavior of CMake. `cmake_minimum_required()` defines policies that CMake will use for configuring the project. Specifying a minimal CMake version that can be used to build the project ensures that all the needed commands will be present and policies from newer versions that could break the building process will not be enabled.[21] Picking the right minimal CMake version is not a critical decision, so version 3.16, released in late 2019, will be used. Since version 3.16 was not released recently, most users should be able to use it and can easily change

it to one of the newer versions. Using command `project()` metadata about the project can be provided. The user will specify the project name, version, and description in the frontend. These metadata will be inserted as arguments of this command alongside an argument specifying the programming language used in the project. CMake enables C and C++ by default, but by specifying that only C++ will be used in the project, CMake will not have to waste time by detecting available C compilers.[21]

```
1  cmake_minimum_required(VERSION 3.16)
2
3  project(glub VERSION 1.0.0 DESCRIPTION "glub description" LANGUAGES CXX)
4
5  if(UNIX)
6    add_compile_definitions(UNIX)
7  elseif(WIN32)
8    add_compile_definitions(WINDOWS)
9  else()
10   message(FATAL_ERROR "Detected platform is not supported!")
11 endif()
12
13 if(EXISTS ${PROJECT_SOURCE_DIR}/res/)
14   message(STATUS "Copying resources...")
15 file(COPY ${PROJECT_SOURCE_DIR}/res/
16   DESTINATION ${CMAKE_BINARY_DIR}/res/)
17 endif()
18
19 message(STATUS "Setting up build options...")
20 file(GLOB_RECURSE SRC_FILES "./src/*.h" "./src/*.cpp")
21
22 if(UNIX)
23   add_executable(glub ${SRC_FILES})
24 elseif(WIN32)
25   add_executable(glub WIN32 ${SRC_FILES})
26 endif()
27
28 message(STATUS "Linking...")
29 find_package(OpenGL REQUIRED)
30 target_link_libraries(glub ${OPENGL_LIBRARIES})
31
32 if(UNIX)
33   target_link_libraries(glub X11)
34 endif()
```

Code listing 7 Basic *CMakeLists.txt* template

The next important step is to add an executable target. C++ files passed as an argument of this command will be compiled and included in the generated executable. Each file can be provided as a separate argument, but glub will group all files with extension `.cpp` or `.h` located in the source code directory specified by the user and pass them as an argument to the command.

Even though this *CMakeLists.txt* example is not using any additional libraries, as an OpenGL application, it will still need to link OpenGL API. First OpenGL package will be found by using the command `find_package(OpenGL REQUIRED)`. After locating the OpenGL package, it can

be linked with the target by using the command `target_link_libraries()`.

The commands mentioned above are sufficient to compile an OpenGL application. Code listing 7 demonstrates a simple *CMakeLists.txt* configuration, that is capable of compiling an OpenGL application without any additional libraries. Apart from already presented commands, it contains a few more useful configurations. To effortlessly determine the operating system in the C++ source files, CMake will define a suitable macro using command `add_compile_definitions()`. If the project contains any resources, glub will insert them into the output directory alongside the compiled executable. And finally, in case the target is compiled for the Unix system, glub will link X11 libraries to enable execution in environments that use X Window System.

4.1.2 Basic OpenGL project template

To verify that the CMake configuration from Code listing 7 is capable of generating a working OpenGL application, a simple OpenGL project should be created and compiled using the configuration. The generated application will be only capable of creating a window, initializing OpenGL context, and freeing allocated resources after the window is closed.

The source code of the OpenGL application will consist of two parts. A main file that will contain the `main()` function, which will be using `CWindow` class to create a window. The other part will be the implementation of `CWindow` class. By using the macros from the section 4.1.2 it can be determined if implementation of the window will be using the X Window System API or the Windows API. Implementation of the `CWindow` using both APIs will be based on the OpenGL documentation by The Khronos Group Inc.[22]

4.1.3 Library integration

Working *CMakeLists.txt* can now be expanded to link a library that is commonly used in OpenGL development. Integrating a library into glub will help identify all the procedures that need to occur before using the library in the application. A library that will be integrated is GLFW. GLFW will remove the need to call platform-specific APIs and demonstrate multiple procedures that libraries might need to be integrated into glub. Simplicity will not be as important as extensibility when integrating GLFW because the integration of other libraries will use the same pattern.

Currently, the project does not include any source files of a library. The source code of libraries will be retrieved from the GitHub repository by using Git. Command `find_package(Git)` can be used to detect Git on the user's system, after which variable `GIT_EXECUTABLE` will contain a path to a file that can be used to execute Git commands. Before running Git commands in the project directory, Git must be initialized in the particular directory. If directory `.git` exists in the project directory, it means Git already is initialized, but if it does not, glub will have to run `init` command for Git.

Now the libraries can be retrieved from GitHub repositories as submodules. Each submodule can be updated with Git command `submodule update`. If the submodule update was not successful, it might mean that submodule was not yet added to the project. By executing Git command `submodule add`, the submodule will be added to the project, and submodule update can be attempted again. If the submodule update was successful, the project now contains the library with the most recent changes. However, users might not want the library API to change while working on their project, so only a specified version of the library should be included in the project. This can be done by utilizing GitHub releases. Each release has a tag that can be used to switch the repository to the same state as when the new version was released by using `checkout` command. The complete process of adding libraries to the project is demonstrated in the Code listing 8.

Libraries with user-defined versions are now present in the project, their compilation and linking can be configured. Libraries might need some specific configuration, mainly defining include directories. It must be ensured there will be space reserved for these configurations. After the necessary configurations, the library can be linked with the project, which concludes a library integration.

In the sample project source code, libraries should demonstrate a simple functionality to verify that they were linked and compiled correctly. In the case of GLFW, it can be window management. The main source code file does not have to change, but the platform-specific implementations of `CWindow` class must be overwritten by an implementation that utilizes GLFW window management.

4.1.4 Template structure

Working examples of OpenGL projects with and without libraries were presented. These examples were created to propose a structure of files generated by the glub REST API. Sections that will be common across files generated with or without libraries will be kept in a template. Library-specific sections will be inserted into the templates by the REST API.

In the *CMakeLists.txt* are three sections that will need to be inserted by the REST API. The submodules update from Code listing 8 should be inserted only if the project uses at least one library. Each library will be required to provide the library's name, version, and repository URL so that the library can be added to the project. Another section that REST API will have to insert is configurations that each library requires to be performed. And lastly, libraries should be linked to the project.

In the main C++ source file, REST API should allow libraries to initialize before entering the update loop. Also, if a library needs to perform any updates, it should be able to add required tasks into the update loop. Libraries should be able to add additional header files into the project and overwrite existing ones.

```
find_package(Git)

if(GIT_FOUND)
  if(NOT EXISTS "${PROJECT_SOURCE_DIR}/.git")
    message(STATUS "Initializing git repository...")
    execute_process(COMMAND ${GIT_EXECUTABLE} init WORKING_DIRECTORY
    ${PROJECT_SOURCE_DIR} RESULT_VARIABLE GIT_INIT_RESULT)

    if(NOT GIT_INIT_RESULT EQUAL "0")
      message(FATAL_ERROR "Unable to initialize git repository.")
    endif()
  endif()

  message(STATUS "Retrieving git submodules...")

  set(SUBMODULES lib/glfw;)
  set(REPOSITORIES https://github.com/glfw/glfw.git;)
  set(RELEASES 3.3.7;)

  foreach(UPD_SUB IN LISTS SUBMODULES)
    execute_process(COMMAND ${GIT_EXECUTABLE} submodule update --init
    --recursive --remote -- ${UPD_SUB} WORKING_DIRECTORY
    ${PROJECT_SOURCE_DIR} RESULT_VARIABLE GIT_SUBMOD_RESULT)

    list(FIND SUBMODULES ${UPD_SUB} SUB_INDEX)
    list(GET REPOSITORIES ${SUB_INDEX} SUB_URL)
    list(GET RELEASES ${SUB_INDEX} SUB_RELEASE)

    if(NOT GIT_SUBMOD_RESULT EQUAL "0")
      execute_process(COMMAND ${GIT_EXECUTABLE} submodule add ${SUB_URL}
      ${UPD_SUB} WORKING_DIRECTORY ${PROJECT_SOURCE_DIR})
      execute_process(COMMAND ${GIT_EXECUTABLE} submodule update --init
      --recursive -- ${UPD_SUB} WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}
      RESULT_VARIABLE GIT_SUBMOD_RESULT)

      if(NOT GIT_SUBMOD_RESULT EQUAL "0")
        message(FATAL_ERROR "Unable to retrieve submodule ${UPD_SUB}")
      endif()
    endif()

    execute_process(COMMAND ${GIT_EXECUTABLE} checkout ${SUB_RELEASE}
    WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}/${UPD_SUB}
    RESULT_VARIABLE GIT_SUBMOD_RESULT)

    if(NOT GIT_SUBMOD_RESULT EQUAL "0")
      message(SEND_ERROR "Unable to checkout branch ${SUB_RELEASE}
      of repository ${UPD_SUB}")
    endif()
  endforeach()
else()
  message(ERROR "Git not found.")
endif()
```

Code listing 8 Retrieving libraries from GitHub

4.2 Project generation

The project will be generated by REST API built with Deno using Typescript. The generation will be based on generation templates proposed in the section 4.1. *CMakeLists.txt* will be used as a template, containing keywords at strategic positions identified in section 4.1.4, which libraries will utilize to insert library-specific source code. A strict structure of data used for project generation must be defined to keep library integrations as simple as possible.

4.2.1 Data structure

All the necessary library-specific source code should be kept together. Each library will have a separate directory containing folders with version tags of the library. It is expected that source code that is able to add a library into the project might be changing over time, but glub should be able to add older versions of the library into the project. The folder for every released library version is redundant, as the library integration will mostly not change. Version folder will integrate specific release of a library, every newer release of a library will use the same integration. If the integration does not work with one of the more recent releases, a new version folder will be added with new library integration, which will be used in the same manner for newer releases.

Library integration will consist of code snippets that should be inserted into the *CMakeLists.txt* template, C++ files, and header files that should be added or replaced in the project. CMake code snippets will be located in the files with extension `.txt`, C++ code snippets will be in the files with `.cpp` extension and header files in `.h` files. CMake and C++ files will match the file's name with a keyword in the template and replace it with its content. Some of the keywords might be reserved, and instead of replacing the keyword in the template, code snippets from all included libraries will be combined and inserted in place of the keyword.

Libraries do not have to implement all the keywords in the templates. In case no library replaces a keyword in the template, the keyword might still need to be replaced by a default code snippet. The default code snippets will be defined in a separate folder but in the same way, as library integration was defined. The default CMake snippets can define different code snippets for projects containing at least one library and for projects that do not contain any library by including the code snippet in a file `[keyword]_raw.txt`.

Figure 4.1 Structure of files used for project generation

```
data
├── default ..... default code snippets
│   ├── [cmake_keyword].txt ..... replaces keyword in CMakeLists.txt
│   ├── [cmake_keyword]_raw.txt ..... replaces cmake keyword if no library is present
│   ├── [cpp_keyword].cpp ..... replaces keyword in C++ files
│   └── [file_name].h ..... adds [file_name].h to the generated project
├── libraries
│   └── [library_name]
│       ├── [version]
│       │   ├── [cmake_keyword].txt ..... replaces keyword in CMakeLists.txt
│       │   ├── [cpp_keyword].cpp ..... replaces keyword in C++ files
│       │   └── [file_name].h ..... adds or replaces [file_name].h in the generated project
│       └── meta.json ..... library metadata
└── CMakeLists.txt ..... CMake template
```

4.2.2 REST API implementation

REST API will include four endpoints; one endpoint will provide a list of available libraries with their versions, another endpoint will provide test results, and the other two endpoints will accept generation requests. The endpoint that provides test results will be implemented in section 4.3, the other endpoints will work with files described in section 4.2.1. Reading the files on each generation request would significantly slow down the response time. Therefore, all the data should be loaded into memory on startup. Code snippets for library integration are specified only for a certain range of library versions. All library releases should be retrieved from GitHub API and assigned code snippets that are necessary to add a particular library version to the project. Since the number of releases can be overwhelming, glub will only consider major or minor version changes. Patch version changes will be discarded, and only the most recent patch version will be used. Apart from loading library-specific code snippets into memory, the default code snippets should be loaded as well.

After all the data is ready, the server can start responding to requests. Endpoint `/libraries` will be a simple *GET* endpoint, which returns list of available libraries in JSON format. Libraries will be categorized by their functionality, and each library will include a list of detected releases. An example of a response with a single category and library with multiple versions is demonstrated in Code snippet 9.

```
1  {
2    "Utility": {
3      "glfw": [ "3.3.7", "3.2.1", "3.1.2", "3.0.4" ]
4    }
5  }
```

Code listing 9 Example response of `/libraries` endpoint

Endpoint `/cmake` will return a plain text response containing generated *CMakeLists.txt*. Generation of the *CMakeLists.txt* will start with the CMake template. Keywords from the template will be replaced with code snippets from various sources. Some keywords like `name`, `version` or `srcPath` will be replaced with data provided in the request. Other keywords will require combining code snippets from included libraries and replace the keyword with the generated code snippet. Combined code snippets are used in multiple places, like linking the libraries to the project or a list of included library repositories in the submodule update process. The rest of the keywords should be replaced with code snippets from the default folder. These default code snippets will get replaced if a library provides a code snippet with the same keyword.

Endpoint `/cpp` will be providing C++ files of a basic OpenGL project in a JSON format. The JSON in the response will contain name of the file as a key with the contents of the file as its value. The keyword insertion will work the same way as in the CMake generation, but it will need to go through all the included files. Also, the libraries will have to be able to add files to the project and replace the default files.

Endpoint `/cmake` and endpoint `/cpp` will be *POST* endpoints. The request for both endpoints will contain the same data. It will include metadata for the generated project and a list of library releases that should be compiled with the project. Request example for these endpoints is illustrated in Code snippet 10.

4.3 Testing

Once the REST API is ready to serve responses, the library configuration tests can be implemented. The test environment uses the REST API to generate a project of a configuration that

```
{
  "name": "glub",
  "version": "1.0.0",
  "description": "glub test",
  "resPath": "res/",
  "srcPath": "src/",
  "libraries": [
    {
      "name": "glfw",
      "version": "3.0.4"
    }
  ]
}
```

Code listing 10 Request example for `/cmake` and `/cpp` endpoints

is being tested. The result of the project's compilation and execution will be saved and later used to verify library compatibility. The test environment will be implemented via GitHub Actions, and it will require two separate workflows. One workflow will be triggered manually with a specific library configuration, and the other will periodically check for new releases of libraries.

To test a specific library configuration, a list of selected libraries must be present in the inputs of the action. The action will start two different jobs, one running on Ubuntu and the other on Windows. The job running Ubuntu will first install packages necessary to build the generated project, but afterward, both jobs will perform the same tasks. The selected libraries from the workflow input will be passed into a script as an argument. The script will request project generation from the currently deployed REST API and place the generated project into the current working directory of the job. The generated project can then be compiled with CMake and executed. However, the execution is consistently falling on Ubuntu runner. The program fails to create a window and returns with exit code 1. The documentation on this topic is lacking, but it can be assumed that the Ubuntu runner is a limited environment that does not allow window creation with OpenGL context; therefore, the application will be executed only on the Windows runner.

The last step of each job will be to set a flag indicating a successful test on the runner's platform. The third job will read these flags and write the final result of the test into the file `data/compatibility.json`. The output file will consist of an object containing library configuration as a key and test result as a value. The key will always contain tested versions of all libraries in alphabetical order. If a library was not present in the tested configuration, the character "0" will be used instead of a version. Some libraries may be incompatible by design, which makes all the configurations containing such libraries incompatible no matter which versions of the libraries are used. In these cases, the character "*" can be used to define test results for a configuration containing the library at any version.

The second workflow will be triggered automatically once a week. It will compare each library's most recent release tag available on GitHub with the latest tested release tag. If there is a newer release on GitHub, the automatic workflow will trigger the previously described workflow with the new library release as an input. The latest tags will be retrieved from GitHub and compared to the latest tested tags by using JavaScript. Upon completing the list of libraries that should be tested, the list will be transformed from a JSON object into a GitHub Action matrix. This matrix can be used to start a separate job for each library that should be tested. Each job will trigger the manual workflow and pass the received library configuration as a workflow input. However, triggering a different workflow from a job created by a matrix and passing an item of

the matrix as an input into the workflow proved to be a complex task to implement. The error logs from the job runs are not just ambiguous and do not provide any meaningful information, but also GitHub lacks detailed documentation for these specific issues. Based on these grounds, the development of automatically triggered tests will have to be postponed.

The test endpoint mentioned in section 4.2.2 can now be implemented since the source of test results is available. `/compatible` will be a *POST* endpoint, requiring a list of libraries with their versions in the request. The test result will be retrieved from `data/compatibility.json` by creating a regular expression that will match a key that follows previously mentioned formatting. If it does not match any key, it means the configuration was not yet tested, which should be stated in the response. If a key was matched, the value of this key can be returned in the response.

4.4 Frontend

The web interface that the user will access will be built with the framework Svelte. The website will contain a single page designed in section 3.3. As seen in the Image 3.6, the page will be split into multiple sections. Each section will be contained within a reusable Svelte component. Each component will contain all the necessary data needed to correctly display the component, including styles and scripts.

Most of the components will be either reading or modifying the project configuration. Therefore, the whole configuration will be provided to the components by utilizing Svelte's store contract. The store contract will define the properties of the project; this way, the components will be able to modify the project configuration and subscribe to changes in any of the properties. Component with inputs for the project's metadata will be writing any changes to properties in the store. Similarly, components that provide library selection will add and remove selected libraries from the store or change their selected version. The compatibility status function will subscribe to any changes made to the library selection and retrieve compatibility for the current selection with a debounce.

The website is accessible at https://glub.drgy.dev/ by using GitHub pages. GitHub pages can deploy only static pages, since the glub website is created with Svelte, it will require a build step. This will be done via a GitHub action triggered by any new commits on the main branch. The action will build the Svelte website, which will generate a static website that can be deployed on GitHub pages.

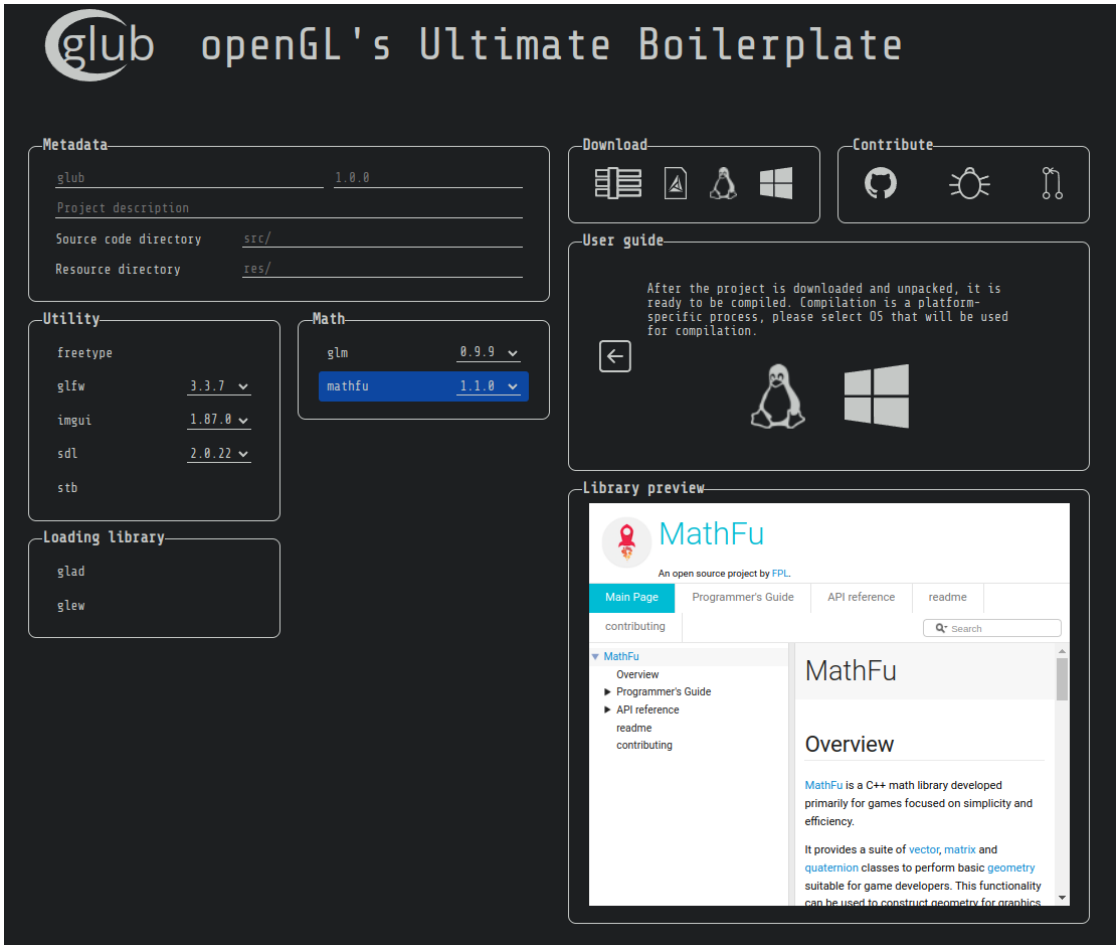

Figure 4.2 Web interface for glub

4.5 Documentation

4.5.1 User guides

It is straightforward to generate an OpenGL project using glub. However, some prerequisites must be met. It is recommended to generate a project using glub's website, but a direct request to glub API can also be made. The website can be accessed using software capable of rendering HTML documents with JavaScript. The compilation of generated projects requires CMake and access to CMake compatible compiler. Compilation and execution are possible only on Windows and Linux-based operating systems that implement OpenGL API.

The website contains an interactive user guide that walks the user through the whole process of generating and compiling the OpenGL project. All the functionalities and actions that users can take within glub's website are described in section 3.3. After downloading the OpenGL project, it can be compiled with CMake. There are multiple ways of compiling a project with CMake. The simplest one is to create a folder in the project directory and run the following commands from the created folder: `cmake ../ && cmake --build .`. On Windows, it is recommended to use Visual Studio for compilation. If the C++ development components and CMake are installed within the Visual Studio, after opening the generated project, it can be compiled simply by clicking the "run" button. All of this information and more is included in the interactive user

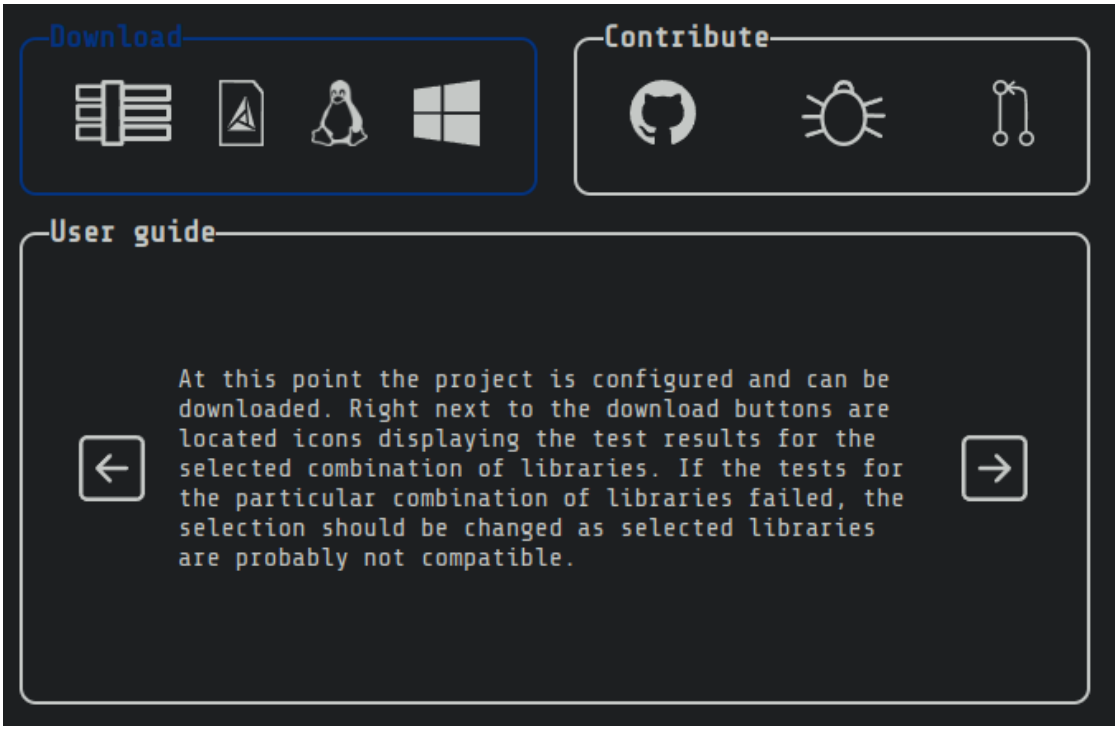

Figure 4.3 Interactive user guide

guide on glub's website as shown in the Image 4.2. Apart from the project's build instructions, the user guide describes steps for generating the OpenGL project and draws the user's attention to the section related to the currently displayed step by highlighting the section's borders. This can be seen in the Image 4.3.

4.5.2 Contributions

Users can influence further development of glub by reporting wrong test results or submitting library requests. To encourage users to take these actions, they are present on the website right next to buttons for project generation. Users will be redirected to a form for submitting a new issue. These templates state what information is expected to be present in the issue in order to be accepted. Template issue for incorrect test results is in the Image A.2 and library request is in the Image A.3.

For contributors that would like to integrate a library into glub, Wiki pages in the repository will be helpful. Wiki pages contain detailed information on the keywords substitution in the templates, a list of available keywords, and an example of the library integration process. Readme files of the glub and glub-web repositories contain information on running glub on the local machine.

Chapter 5

Conclusion

The motivation for creating glub was the lack of solutions for a simple compilation of OpenGL applications. Even though glub fulfills the requirements for an efficient compilation tool, it can not be determined if it solved all the existing issues with OpenGL compilation, as user feedback and usage statistics are not available as of yet. However, glub can still be compared to other solutions that programmers use. Visual Studio provides a GUI for configuring the compilation and linking of libraries. Because of the number of available compilation configurations, the GUI becomes obscure and also requires the user to know how the library should be compiled and linked. This solution is obviously limited only to programmers that use the Visual Studio IDE for the development. Anyone else that would decide to contribute to the project would have to configure the compilation themselves or use the Visual Studio IDE.

Some of the programmers share their project templates that are able to compile an OpenGL project with predefined libraries, usually configured with CMake. Most of the templates still require the user to download the required libraries or change the configuration if any library should be removed or added. Configuring the project by using GUI tools like Visual Studio or searching for the already created template are inefficient ways of OpenGL project configuration compared to using glub. With glub, it is possible to generate a configuration with only a few clicks. No knowledge of how the included libraries are compiled and linked is required. Also, the user does not even have to download the libraries manually, the generated project will automatically download everything needed. Libraries can be easily added or removed, the selection is limited only to libraries integrated into glub. However, if a popular library is missing, the user can submit a library request or integrate the library by following the contribution guidelines. Since the compilation configuration can be generated through a web service, it is not limited to programmers using only a specific set of development tools.

Creating a tool like glub is a task that needs to be broken down into smaller objectives. It was important to analyze available technologies for building glub, especially the technology used for the compilation configuration. In the chapter 2 technologies for each component of glub were analyzed, and CMake was determined as the most viable technology for configuring the compilation process. Chapter 3 defines how the user will interact with glub and communication of glub's components with each other. Following the designs from chapter 3, all of the components were implemented. Details of the implementation process are described in chapter 4. The structure of templates used for the generation of compilation configuration was defined in a way that allows simple library maintenance and addition. REST API that uses these templates to generate compilation configurations and projects requested by users was implemented with Deno. The REST API is currently deployed via Deno Deploy and serves requests on https://glub.deno.dev/. Deployed REST API is used to generate projects for the test environment, which is running on GitHub Actions and verifies that various libraries and library versions are working together. The

test results from the test environment and endpoints that the REST API provides are utilized by a website built with the Svelte framework, which currently serves as an interface that users can use to generate an OpenGL project on https://glub.drgy.dev/. Source code of all the glub components is freely available on GitHub, containing user guides and documentation for library integration, which allows users to integrate their favorite OpenGL libraries into glub.

All of the crucial objectives are fulfilled, and glub is available for all the OpenGL programmers on https://glub.drgy.dev/. The success of glub depends on the number of people that decide to use it and maintain the libraries. The hope is that glub will become helpful for many people, which will, in return, update outdated integration of libraries or integrate libraries that they use. To better blend into user's workflows, glub will be available as a plugin to popular IDEs in the future. Thanks to the practical design, creating a glub plugin will not be difficult, as all the necessary data can be retrieved from the REST API. If glub becomes a popular tool amongst OpenGL programmers, it is extendable to support other graphical APIs like Vulkan or DirectX.

Appendix A

Images

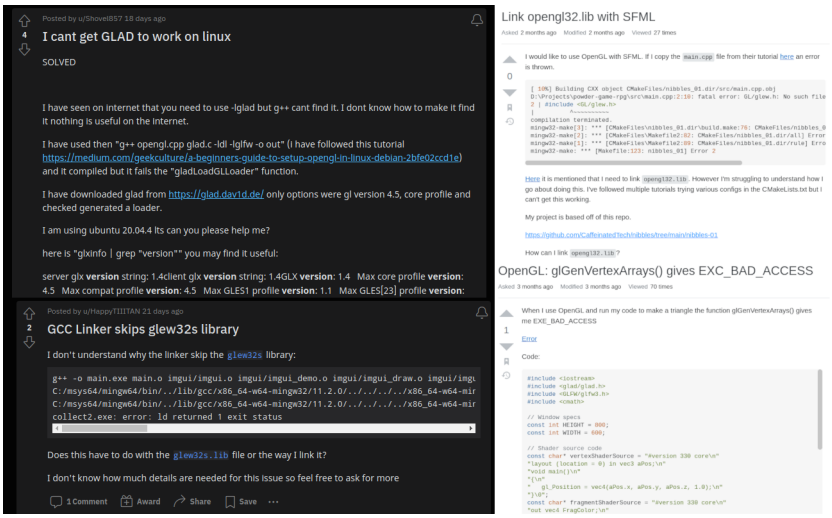

Figure A.1 Collection of questions related to OpenGL compilation

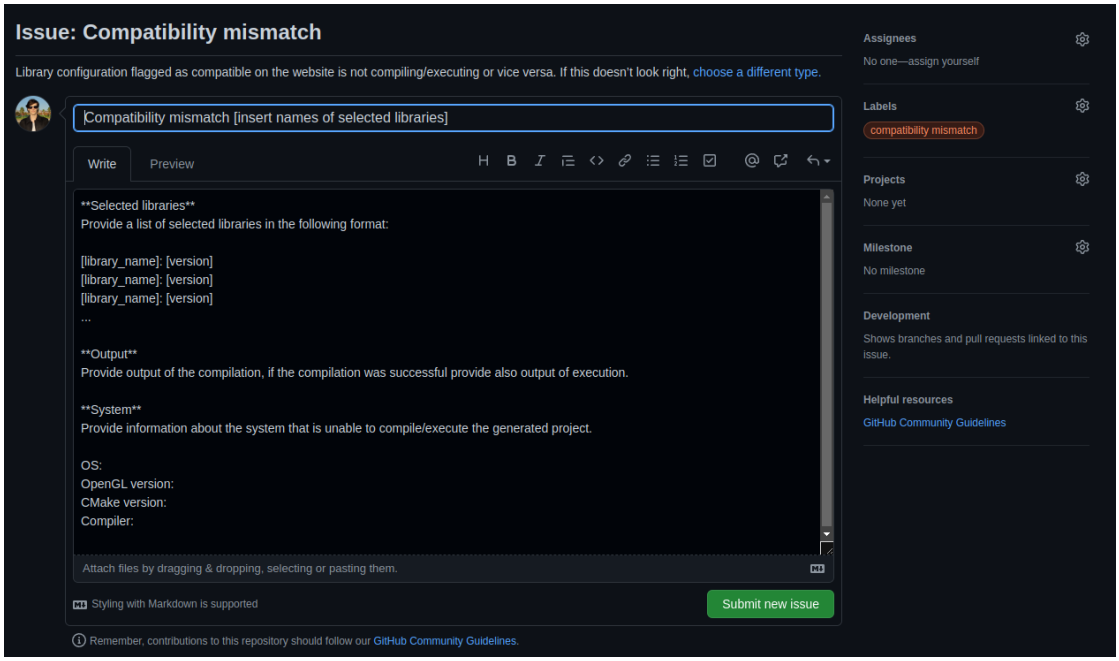

Figure A.2 Issue template for submitting an incorrect test result

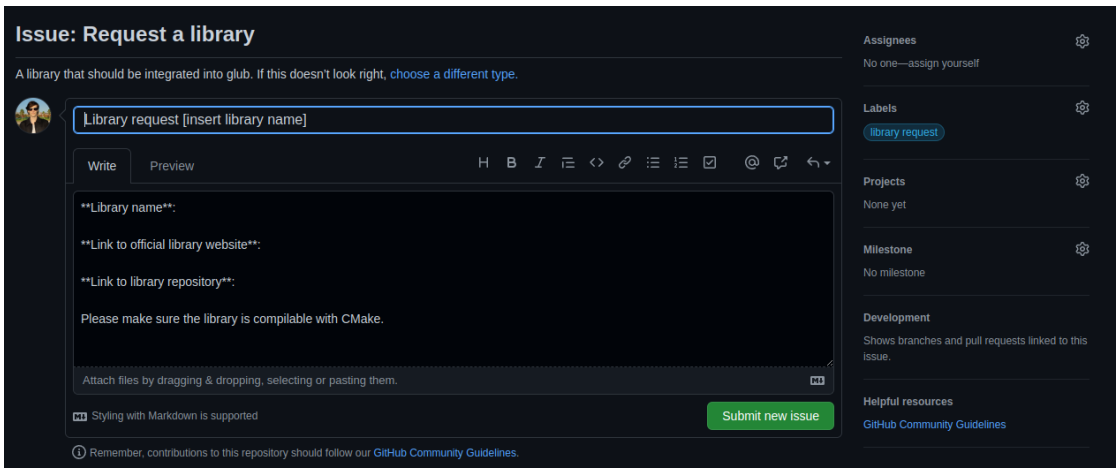

Figure A.3 Issue template for submitting a library request

Bibliography

[1] G. Sellers, J. Richard S. Wright, and N. Haemel, *OpenGL Superbible: Comprehensive Tutorial and Reference*, Seventh Edition. New Jersey, USA: Pearson Education, Inc., 2016, ISBN: 9780672337475.

[2] JetBrains s.r.o., *The IntelliJ Platform*, 2022. [Online]. Available: `https://plugins.jetbrains.com/docs/intellij/intellij-platform.html`.

[3] Microsoft Corporation, *Visual Studio Code - Extension API*, 2022. [Online]. Available: `https://code.visualstudio.com/api`.

[4] Free Software Foundation, Inc., *GNU Make manual*, 2022. [Online]. Available: `https://www.gnu.org/software/make/manual/make.html`.

[5] Kitware, Inc., *CMake*, 2022. [Online]. Available: `https://cmake.org/`.

[6] Laravel LLC, *Laravel documentation*, 2022. [Online]. Available: `https://laravel.com/docs/9.x`.

[7] StrongLoop, *Express*, 2022. [Online]. Available: `https://expressjs.com/`.

[8] L. Casonato, R. Dahl, *et al.*, *Deno*, 2022. [Online]. Available: `https://deno.land/`.

[9] Django Software Foundation, *Django*, 2022. [Online]. Available: `https://www.djangoproject.com/`.

[10] R. M. França, A. Patterson, *et al.*, *Ruby on Rails documentation*, 2022. [Online]. Available: `https://api.rubyonrails.org/`.

[11] Fly.io LLC, *Fly.io*, 2022. [Online]. Available: `https://fly.io/`.

[12] Clever Cloud, *Clever Cloud*, 2022. [Online]. Available: `https://www.clever-cloud.com/`.

[13] L. Casonato, R. Dahl, *et al.*, *Deno Deploy*, 2022. [Online]. Available: `https://deno.com/deploy`.

[14] M. Blumenkrantz, A. Rosenzweig, E. Anholt, M. Olšák, *et al.*, *Mesa documentation*, 2022. [Online]. Available: `https://docs.mesa3d.org/index.html`.

[15] M. Logan, *GitHub Actions Runners*, 2022. [Online]. Available: `https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/github-action-runners-analyzing-the-environment-and-security-in-action`.

[16] GitHub, Inc., *GitHub Actions documentation*, 2022. [Online]. Available: `https://docs.github.com/en/actions`.

[17] Circle Internet Services, Inc., *CircleCI documentation*, 2022. [Online]. Available: `https://circleci.com/docs/`.

[18] Database Mart LLC, *VPS Mart - GPU VPS*, 2022. [Online]. Available: `https://www.vps-mart.com/gpu-server`.

[19] R. Harris, *Svelte documentation*, 2022. [Online]. Available: `https://svelte.dev/docs`.

[20] J. Arlow and I. Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*, Second Edition. New Jersey, USA: Pearson Education, Inc., 2005, ISBN: 9780321321275.

[21] R. Świdziński, *Modern CMake for C++*. Birmingham, UK: Packt Publishing Ltd., 2022, ISBN: 9781801070058.

[22] The Khronos Group Inc., *Khronos OpenGL documentation*, 2022. [Online]. Available: `https://www.khronos.org/opengl/wiki/Main_Page`.

Contents of attachment

- `glub` — REST API source code
  - `.github`
    - `ISSUE_TEMPLATE` — templates for GitHub issues
    - `workflows` — GitHub Actions source code
  - `data`
    - `default` — code snippets for project generation
    - `libraries` — library specific code snippets
    - `CMakeLists.txt` — template for generation
    - `compatibility.json` — test results
  - `src` — REST API implementation
  - `tests` — scripts used by test environment
  - `README.md` — instructions for running glub API locally
- `glub-web` — web interface source code
  - `.github`
    - `workflows` — GitHub Actions source code
  - `src` — glub website implementation
  - `static` — static resources for glub website
  - `README.md` — instructions for running glub website locally
- `glub.wiki` — contribution guides from glub's GitHub repository
- `thesis.pdf` — thesis document