Pepr3D

Authors: Bc. Štěpán Hojdar, Bc. Tomáš Iser, Bc. Jindřich Pikora, Bc. Luis Sanchez
Supervisor: Mgr. Oskár Elek, PhD.
Consultants: doc. Ing. Jaroslav Křivánek, Ph.D., Ing. Vojtěch Bubník (Prusa Research s.r.o.)

Faculty of Mathematics and Physics
Charles University

Contents

Part I

Introduction and Functional Requirements

Chapter 1

Introduction

1.1 3D printing basics

3D printing is a new technology that has seen rapid developement in the last years. It comes in many different forms, melting plastic, fusing metals, shining UV on photopolymers, etc. Fused Deposition Modelling (FDM) is the most popular and accessible to the general public and for the purpose of this project, when we talk about 3D printing, we will always mean FDM printers, unless stated otherwise.

FDM printing is a relatively simple process - a printer head melts the plastic filament and deposits it on a preheated platform layer by layer, from the bottom towards the top. The printer has to regulate the temperature of both the filament in the head and the moving platform for the deposited material to bond correctly. Several types of filaments are used, namely PLA, ABS, PET and others.

1.2 Prusa environment

The Prusa environment is very similar to the general description we provided in the section 1.1. For the puprose of our project, the most important concept in the Prusa environment is the slicer. The slicer is a program that receives the 3D model the user wishes to print out and creates the instructions for the Prusa 3D Printer – a G-code file. The file is then transfered to the printer, which then executes the commands in the G-code file. The slicer has to plan the movement of the head for the whole print. This includes several crucial things:

- Covering the whole area of each layer
- Reinforcing the walls of the object to make them sturdier
- Filling the inside of the object with a rougher print, because it won't be visible when finished
- Planning the path so the head can stay in one Z level - an "Eulerian path".
- Switching the materials for multimaterial printing (more in 1.3)

Prusa develop their own slicer - a forked branch of an open-source program called Slic3r [1], called Slic3r Prusa Edition [2]. This slicer can do all we listed above very well.

1.3 Multimaterial printing

Multimaterial printing is a very new concept, even in the fairly new world of 3D printing. Many of the simpler and cheaper 3D printers can only print one material models - one color for the whole object. However, many users would like to print models that include more than one color. Even though the more advanced printers are capable of combining up to four different materials into one print, the process to achieve this is rather cumbersom for the end user - the user has to manually split the 3D mesh of the object into parts that he wishes to have a different color.

For example, if we are printing a dragon, want the dragon to be black and have white teeth, we have to take the dragon model, and split off each individual tooth. Then tell the slicer that the remaining file - the toothles dragon should be black and the teeth should be white.

This model splitting has to be done in a full 3D editing software like Blender or 3ds Max, which is difficult to control for newcomers and overly complex.

1.4 Our project

Our project aims to make printing a multi-colored object a lot easier, by developing an application that will allow the user to simply paint on the 3D model (i.e. the dragon) with different colors (i.e. color the teeth white), then simply click export and generate the files of the split-off models automatically.

Our application should allow for free hand painting as well as some forms of guided painting - bucket fill and some smarter tools, for example a bucket fill that studies the object's geometry and stops the filling if it detects a sharp edge (i.e. the transition of the tooth into the dragon).

Our aim is to make the application for desktop PCs, with main developement time being focused on the Windows operating system. However, we are trying to use software engineering tools that can also be ported to a plethora of other platforms like Linux based OS, Mac OS and mobile, if the need should arise.

1.5 Related works

Based on the analysis of the experts from Prusa Research s.r.o, there, at the moment, does not exist a software that does what this project is trying to achieve.

The closest existing software is Autodesk Meshmixer [3], which is very complicated and is not targeted for FDM printing specifically. As such, it includes a lot of features that are not important for the FDM users and end up being confusing.

[1] http://slic3r.org/

[2] https://www.prusa3d.com/slic3r-prusa-edition/

[3] http://www.meshmixer.com/

Microsoft 3D Builder [4] is another application that handles 3D models but we have not found a way to make it create anything remotely applicable to FDM printing.

Any 3D computer graphics program designed to handle 3D models which allows for the model to be created or split by colors manually. This section would include software as 3ds Max, Maya or Cinema4D. Using these applications, however, would be very time comsuming for the user and practically unusable on a larger scale.

1.6 Challenges in this project

This section should briefly familiarize the reader with some of the parts of the application we think will be difficult to implement correctly, before we present the full program specification.

1.6.1 Handling the geometry during editing

We want our application to be able to emboss text on the surface of the object, detect edges and stop painting the color during bucket fills, allow the user to paint fine details on a rough triangle mesh. All of these things require some degree of subdividing the triangle mesh to allow the user to create small details. We think that this potentially could involve some difficult problems - we have to allow the user to subdivide the triangle mesh enough to actually allow him to create fine details on the surface. However, the if the user goes overboard with the subdivision, the model will be too complex to print or even handle inside a desktop PC.

1.6.2 Exporting the finished objects

After the user is done painting, the application will have to separate the designated objects and areas into distinct meshes. This is potentially a very complicated task to do correctly for non-convex meshes. For example: if we are writing a text on a ball, we really only want the text to be carved deep enough into the ball for the printed material to hold firmly, we do not want it to be too deep. However, if we want the dragon's teeth to be white, we would prefer the whole tooth to be white, not just its surface. The distinction between these two cases could be non-trivial.

1.6.3 Performance

Handling complex geometry is a very taxing task for the user's computer. This application is targeted on beginner-level customers of simple and non-expensive 3D printers. Therefore the application cannot be too hardware demanding – it has to run smoothly on an average 3 year old PC or laptop. We expect it is going to be hard to ensure this is the case.

[4] https://www.microsoft.com/en-us/p/3d-builder/9wzdncrfj3t6?activetab=pivot%3Aoverviewtab

Chapter 2

Use case

2.1 Wireframe of the application

Figure 2.1 provides a simple wireframe sketch of the application graphical user interface (GUI). The window consists of several usual components:

- A horizontal toolbar at the top, allowing for a fast selection of tools and file manipulation
- A 3D preview window, with live preview of the object. This window allows rotating and magnifying the object, as well as the application of selected tools (e.g. painting with a brush). A simple grid and a 3D cross is provided to ensure the user is always aware of object orientation, as it is important for 3D printing.
- An options window on the right allows the user to customize the settings of the currently selected tool (e.g. selecting the color for a brush).

2.2 Workflow and editor tools

In this section we describe the intended workflow for a user who has a 3D model he wishes to color and then print on a multicolor FDM printer. This application is intended for users of varying degrees of experience and our goal is to create as easy-to-use application as possible.

2.2.1 Importing the model

First, the user has to import the model he found or created. Clicking on the *import button* creates a dialog, the user selects the 3D model and the model gets loaded. The application should accept at least a few standard formats – namely Wavefront .obj and .stl [1], both of which are widespread and well known among the 3D printing community.

The user should also be able to continue on an already existing project made earlier with Pepr3D.

[1] https://en.wikipedia.org/wiki/STL_(file_format)

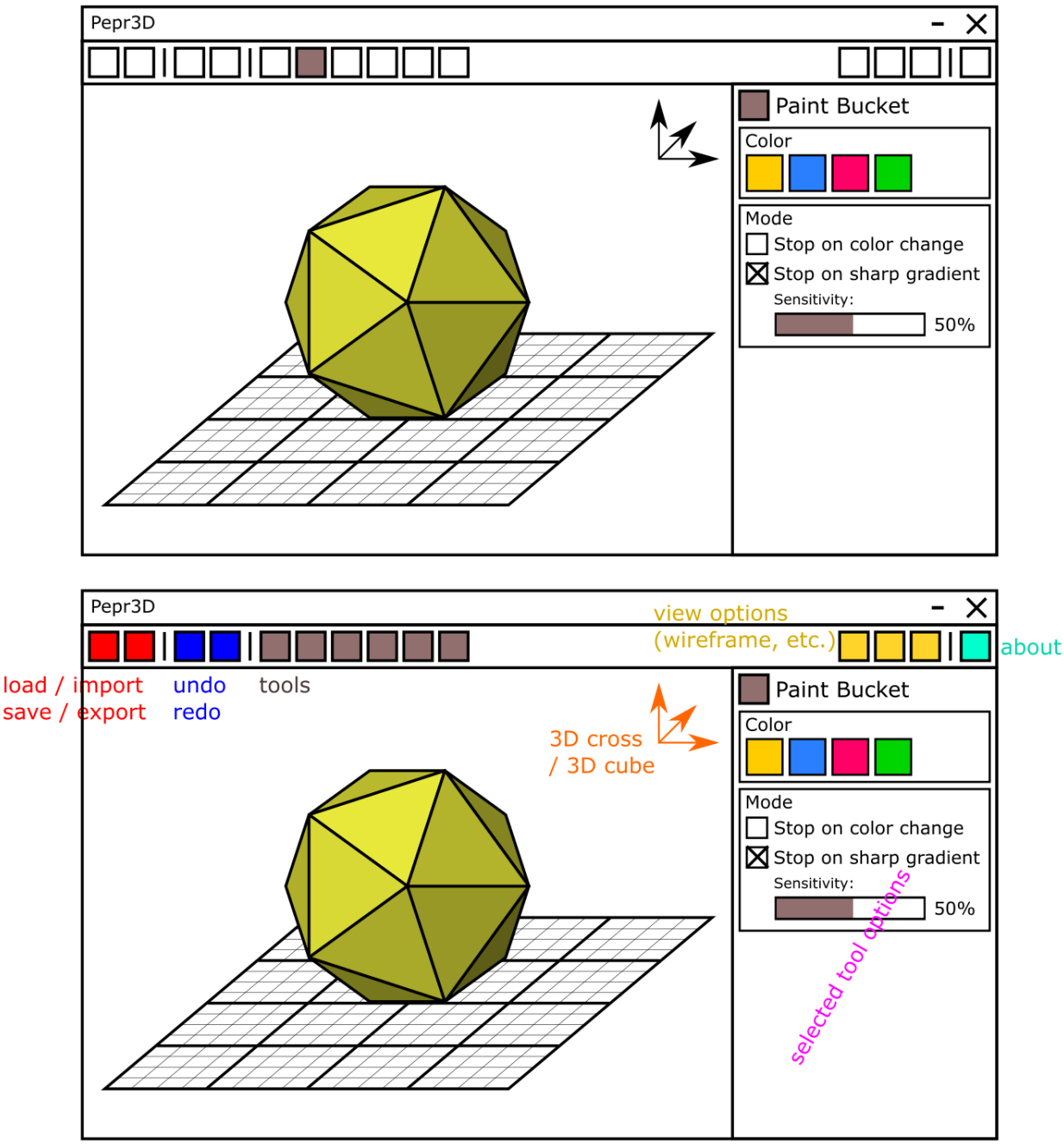

Figure 2.1: A simple wireframe sketch of the application.

After the model is loaded, it will be rendered in the 3D preview window. The window allows the user to rotate, zoom in and out and preview the wireframe of the model (rendering only edges and no faces of the model). The user is able to set the number of colors he wants to use (by default 4 because current Prusa printers support up to four colors). The model is colored with the first color by default.

The user then selects one of the tools from the toolbar. This will bring up the *Tool options* menu on the right hand side allowing the user to customize the tool.

2.2.2 Tools

Edit history

The user is always able to revert his last action by using an *Undo button* or a keyboard shortcut. Depending on the technical difficulties, this feature could also persist through different sessions.

Save as Pepr3D project

Saves the current project as a Pepr3D project file. Upon re-opening Pepr3D, the user can load the project back and continue the work as if he never left. Does not include exporting the file into a slicer-compatible format.

Export

Export the file into a slicer-compatible format. This file is then handed to the slicing program (e.g. Slic3r Prusa Edition we mentioned earlier) and can be printed directly.

Triangle painter

After selecting a color, the user can assign said color to triangles he clicks on. Backside filtering is always on, so the user can only ever color a triangle that faces towards him, which should prevent a lot of accidents a lesser experienced user might make.

This is the simplest way to color the user's model and may be desireable as a quick way to correct any small errors made by either the user or the automatic segmentation itself. This tool also allows for very quick and easy coloring of very simple models (like a cube) that have a very low triangle count when the user wishes for a simple outcome – for example a playing dice with six differently colored sides.

The *properties* of this tool are very simple – the user chooses a color to assign to the triangles that he clicks on. Depending on testing and remaining time, this tool could be expanded to include a radius and instead of coloring the single triangle the user clicks on, we could color all triangles in the vicinity of the clicked triangle instead.

Bucket painting

The user selects a color and by clicking anywhere on the model paints all triangles with selected color until an edge criterion is met. The simplest and most intuitive edge criterion is continuity (a hole stops the bucket spread). Several more criterions could be useful when in 3D, namely the sharpness of the normal (if two neighbouring triangles are at an angle greater than X, stop.) or a big gradient in a *shape diameter function* (SDF).

A little more complex but a very intuitive tool that allows the user to get a quick initial paint on the model, which can later be adjusted with the Triangle painter tool, or made more complex with the finer Brush tool.

We already mentioned the main *property* of the tool — the edge criterion. A color-picker is, of course, neccessary as well.

Automatic segmentation

Pepr3D fully automatically colors the whole model using the selected colors, according to a edge criterion as discussed in the *Bucket painting* section. The user can then decide if he wants to merge some segments together, reducing the number of colors.

This tool should serve as a reliable way to color simple models which can be very distinctly separated into a number of regions, like a guitar which has two main parts – the body and the neck.

The *properties* of this tool are as following:

- The number of regions
- The colors assigned to the regions
- The edge criterion settings

Semi-automatic segmentation

The user roughly paints over triangles in areas that should have distinct colors, as indicated by Figure 2.2. The program then finishes the coloring by executing a clever flood-fill algorithm utilizing SDF, sharp edges, etc.

This tool is primarily aimed at models that do not clearly separate into a few regions, and as such the computer would have a hard time guessing which regions the user had in mind. It is quick to use, since it requires non-precise brush strokes and should be a significant speed up when compared with just using the Bucket and Triangle painters to do all the work manually. Some manual adjustment is expected from the user but the main bulk of the painting should be done automatically.

The properties are very similar to automatic segmentation, the only exception being that the program will actually require the user to paint with the colors assigned to each region, after the user chooses the colors. If any of the brush strokes are missing, the program can either completely skip the color or generate the missing region automatically. The specific behavior will be chosen once both approaches are tested by the team.

Figure 2.2: Semi-automatic segmentation as seen from the user's perspective. The ears of the rabbit are yellow as indicated by one stroke on each ear. The body is orange as indicated by the stroke on its back. The rabbit's feet are pink – four pink strokes.

Brush

A simple to use brush tool that allows to paint onto the model with a selected color. This tool allows the user to paint finer details, even though the geometry does not include them. For example painting the nose of the rabbit from Figure 2.2 black – there is no distinct edges on the nose, but the user can color only the nose by fine strokes of the brush.

The implementation of this tools is harder, because the program needs to adaptively subsample the triangle mesh to allow for finer details. This poses a lot of problems, which will later be discussed in the implementation parts of the document.

The *properties* of the Brush tool include the color the user wishes to paint with, the shape of the brush itself and the size of the brush. A fail-safe setting that will stop subdividing triangles if they are smaller than the set number should be present, to avoid the user accidentally creating a model too complex for their computer to handle.

Text

Using the *tool options* window, the user selects a font and types a custom text into a window. The text gets projected onto the model using some sort of projection transformation (customizable by the user from a limited range of projections). The software also allows extruding the projected text in the direction of the surface normal to create a 3D effect.

We anticipate that this will be a very popular feature, especially among com-

Figure 2.3: Three stages of triangle numbers. The bunny on the left has the most triangles and most complicated geometry. Several decimations can reduce the number of triangles but also the number of details as shown on the second and third bunny.

panies, since printing their own promotional items, with the ability to emboss the items with the company's name and logo seems very useful.

The *properties* of this tool include the standard feature-set of text editing – the font, size, style (bold, italics, underlined, etc.) as well as the color of the letters. The more advanced settings of this tool include the projection type, and since we allow the extrusion of the text above the surface itself, the height will have to specified as well.

Triangle subdivision/decimation

The user selects a section of triangles and then presses either subdivide or decimate, which will either make the geometry more complicated, or more simple. See Figure 2.3 for visual aid. Please note that our tool is not a sculpting tool and such this tool might not allow the same extent of modifying the model as some 3D editors do.

We anticipate this to be a niche tool, used only by the more experienced users. We chose to include it, since it allows the proffesional printers to fine-tune their models for better performance or more precise coloring or printing.

Part II

Non-Functional Requirements

Chapter 3

Architecture

Now that we understand the background and use case of Pepr3D, we can propose a software architecture for the project. In this chapter, we show an overview of the whole architecture. Further details are then available in next chapters.

3.1 Overview

A good software architecture should be easy to maintain and refactor, it should be possible to replace parts of it with new ones. It should be as simple as possible, resistant to bugs and errors, and programs built with such architecture should run reasonably fast without any major bottlenecks.

This can be achieved with *modularity*, i.e., separating a project into multiple parts that are as independent as possible. It is important to define data flows and dependencies between the modules. Keeping the data flows and dependencies as simple as possible makes it easy to replace modules in case it is necessary, e.g., when a certain library is not developed anymore or new technology is created.

In our case, we tried to come up with a modular and simple architecture. As Pepr3D is a 3D editor, it consists of both a complex *backend* with geometry manipulation, and a complex *frontend* for displaying and editing this geometry by a user. We propose an architecture consisting of several parts ranging from backend to frontend. Details are in Figure 3.1 and in the next section.

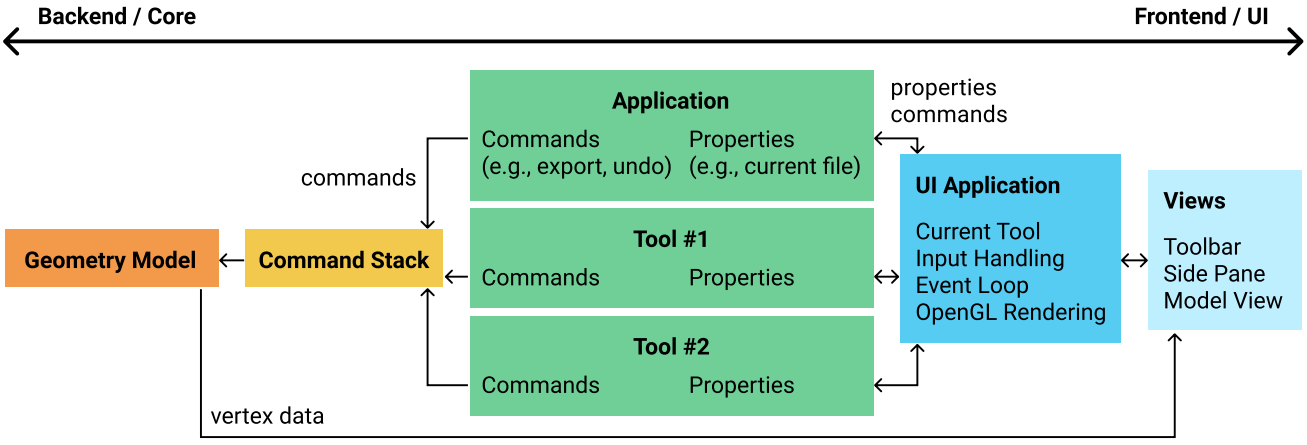

Figure 3.1: An overview of the Pepr3D architecture.

3.2 Modules

For the Pepr3D architecture, we propose the following modules:

- We need a module responsible for the geometry of the edited 3D object, including importing, exporting, and editing the object. **Geometry model** is a data structure describing everything Pepr3D needs to know about the 3D model. A 3D view is also rendered based on this structure.

- Then we need a module allowing *undo* and *redo* functionality. In editors, this is mostly done using a *command pattern*, in our case encapsulated in the **command stack** module. This module receives commands and changes the geometry model accordingly.

- The commands are created and sent either directly from the **application** module or from **tools**. Application is responsible for generic commands such as loading and saving files, importing and exporting, or invoking undo and redo. Tools are responsible for their corresponding actions such as changing a color of a triangle or running a segmentation. Both application and tools also have *properties*, e.g., a brush size of a brush tool.

- The application and tools are managed by a **UI application** module which is a bridge between what the user sees and what the application does. It manages a window, handles events such as user input, processes asynchronous events in the event loop, manages an OpenGL context, etc.

- Finally, **views** are responsible for actually displaying information on user's screen. They describe a toolbar, side pane, and a 3D model view. They correspond to buttons, text inputs, icons, numerical sliders, etc. The user interacts with Pepr3D through the views.

3.3 Data flow

In software such as editors, when we interact with the application, data first flow from frontend to backend (1.), and then from backend to frontend again (2.).

1. Front to back In our case, when a user uses a certain tool, e.g., a brush, and paints on the 3D model, this painting gets first registered in a **view**. An asynchronous event is created through the **UI application** and a corresponding command is invoked from a current **tool** according to its properties. The command is processed through the **command stack** to allow us to undo it in the future. The command then modifies the **geometry model** accordingly.

2. Back to front After changing the **geometry model**, the command gets resolved and the **command stack** saves it to its history. If no other commands are required by the **tool**, it notifies the **UI application** that the asynchronous event has finished. The new geometry data are then immediately visible in the **view**, which renders the new geometry model.

3.4 Choosing a language

When selecting a programming language for a certain project, there are basically two important things to consider. First, do we already have experience with the language or at least with a similar one? Second, does this language provide all features necessary for the project? Are there libraries available to help us building the software? Is it not too complicated for the use case?

The first question is important because building a big project with a brand new language is very difficult. It is too easy to start with a project and halfway through find out that one has been using the language concepts wrong the whole time. We should have at least one person with some experience before starting.

The second part is important as some languages might be better for certain use cases. For example, it would not make much sense making a web application in C++ instead of JavaScript, and vice-versa for complex geometry applications.

In our case, we had quite a lot of experience with C++, C#, JavaScript, and Python. We want Pepr3D to be cross-platform and do heavy manipulations with 3D geometry as fast as possible. We prefer compiled languages with optimized compilers, complex debuggers, profilers, and fast 3D libraries.

Other 3D editors are mostly developed with C++ and most suitable libraries (see future sections and chapters) also primarily target C++. It is also a language with cross-platform heavily optimized compilers. Hence, we decided to choose **C++** as a primary language for Pepr3D.

3.5 Dependencies

In software development, it is a good idea *not* to reinvent the wheel. It means that if there is a library available for a certain task that we would like to do, it makes sense to consider using such library.

3.5.1 Why yes, why not

Libraries are useful for solving complex error-prone tasks that might be too difficult to implement ourselves. As libraries typically have other users, there is a chance they have already spotted important bugs and reported them. Hence there is a high chance they have already been fixed, which is not the case when we decide to develop our brand new custom solution. Also, when a library is actively developed, it can keep improving and we do not even need to touch it.

Obviously, for certain tasks, a library might not exist, it might be too old and not maintained anymore, or it might not be in a good condition in general. Using third party libraries makes maintenance of the project more difficult. We need to keep checking if our dependencies are up to date, if there are known bugs or security issues in them. Also in case we need to modify the library, we need to keep our custom fork of it and keep merging latest changes. Different libraries also tend to use their own classes and structures for the same thing, which makes the source codes more difficult to understand.

3.5.2 Libraries used by Pepr3D modules

We have made a list of libraries that we would like to use to implement Pepr3D. Details about why we have decided to use exactly *these* libraries are available in specific chapters related to the modules. This is just a brief overview:

- **Assimp**[1] library should be used for importing and exporting 3D models for the *Geometry Model* module.
- **Cereal**[2] library should be used for (de)serialization of *Geometry Model* and *Application* data.
- **CGAL**[3] library is considered for complex 3D geometry and topology calculations in the *Tools* commands affecting the *Geometry Model*.
- **Cinder**[4] library will be used in the *UI Application* for managing cross-platform windows, input handling, asynchronous event loop, **OpenGL** context handling, and other related tasks.
- **Dear ImGui**[5] library will be used in *Views* as a UI widgets library for managing buttons, text inputs, checkboxes, and other UI widgets.
- This list of dependencies may not be final. Other libraries are also considered, such as **spirit-po**[6] for translating UI into different languages.

[1] http://www.assimp.org/
[2] https://uscilab.github.io/cereal/
[3] https://www.cgal.org/
[4] https://libcinder.org/
[5] https://github.com/ocornut/imgui
[6] https://github.com/cbeck88/spirit-po

Chapter 4

Core modules

This chapter takes a closer look at the portion of the architecture higlighted in Figure 4.1. We will explain how each task is performed once the UI element has received the input from the user. The UI portion of the architecture will be covered in the next chapter.

4.1 Commands

Commands are the primary means of altering the geometry model. Each of them gets executed and placed on the command stack, which allows for the *Undo and Redo* operations to function correctly. The commands then interact with the geometry model (this interaction will be explained in greater detail in the chapter regarding the geometry model) to modify it according to the user's wishes.

Because each command gets put on the command stack, and each Undo step removes one command from the stack, each command has to have a visual impact on the user's work. This means that internal computations, such as geometry queries, cannot be represented as commands, because pressing the Undo button would not have any visual effect and would confuse the user. Examples of commands include: coloring a single triangle (triangle painter tool), adding a brush stroke (as in semi-automatic segmentation we looked at in Chapter 2) or a single step of triangle mesh subdivision.

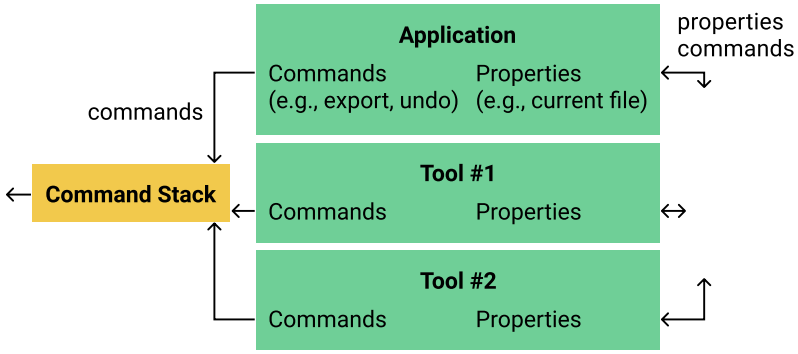

Figure 4.1: The part of architecture covered in this chapter.

Implementation details

A command will be a class with three primary methods.

- *constructor* – Once the command gets created, the main thing to do is remember the state the geometry model was in before the command modified it. This will allow for the command to restore the model if *Undo* is pressed. The advantage of saving the state in the constructor and not in the execute function itself is because of the *Redo* mechanism – if the user repeatedly presses the combination of *Undo + Redo*, the model gets saved only once as opposed to each execute.

- *execute()* – The execute function will perform the task the command is created to do. This will typically get called immediately before the command gets placed onto the command stack.

- *undo()* – This method will take the state of the model saved by the constructor of the command and apply it to the current geometry model, effectively reversing the command.

4.2 Command Manager and Command Stack

4.2.1 Command Stack

As the name suggests, the Command Stack is a *LIFO* type structure, its main purpose to store the executed commands to allow for the *Undo and Redo* operations to be performed. As we have outlined in the previous section, each command carries all neccessary information to perform both actions. This means that the Command Stack can be a simple container without any significant logic behind it.

4.2.2 Command Manager

The command manager is a object to manage the command stack. It receives the commands from tools, executes them and stores them in the command stack. When the user wishes to *Undo* an operation, the command manager retrieves the top command from the stack and invokes its undo() method.

4.3 Tools

A tool is the main programable component which connects the low-level command structure we outlined above and the high-level UI components (such as color-pickers, the user performing brush strokes and file operations like exporting the file). This design should also allow for later advancements of the software easily – adding a tool to the software should be a matter of writing the new tool's Tool class, unless the tool is advanced and needs some complicated custom geometry processing functions.

Each tool is composed of two main components:

- *properties* – a methodless object holding the tool's properties which can be customized by the user. This includes, for example, the color which gets assigned to the triangles in the triangle painter tool, the number of colors for automatic-segmentation, the subdivision level or the gradient thresholds for region detection in bucket painting. The information in this object gets changed directly when the user interacts with the UI.
- *commands* – Each tool generates at least one command, which it creates, fills with all neccessary information the command needs to execute itself, and passes it to the command manager. More complicated tools can create more commands, as we will illustrate in the following section. The tool is able to do some pre-processing before a command is issued, in case the pre-processing isn't visible on the screen, as shown in example 4.1.

4.4 Examples of data flow within the center section of the achitecture

We include a few example use cases which should illustrate what happens in the application (normal font) when a user interacts with the UI (italics). The first example shows the need for preprocessing power within the tool class, while the second example illustrates the need for multi-command tools.

4.4.1 A preprocessing example - triangle painter

The user selects the triangle painter from the tool box, and in the Side Pane, he changes the color from default red to green.

This action makes the UI manager change the color property of the TrianglePainter tool from red to green.

The user then clicks on a triangle that he sees on the screen, in Model View.

This makes the UI generate a ray in a direction of the user's mouse input. It then calls the TrianglePainter tool, passes the ray and tells it to color it, according to the tool's settings. The tool first calls the Geometry Model, to retrieve the triangle that gets intersected by the given ray, then creates a command to color the triangle with the color it has in its properties. The command is then passed to the CommandManager and executed. The query for the ray intersection is not passed as a command, because it is not visible for the user, hence should not be reversible.

4.4.2 A multicommand tool example - the semi-automatic segmentation tool

The user selects the number of colors – 4. The user also selects which colors he would like to use – C, M, Y, K.

The UI updates the tool's properties to reflect these values.

The user then selects the color C, and performs a stroke.

The UI first tells the tool that the current color is C. Then the tool receives the parameters of the stroke (much like the ray for a click), creates the command for a stroke with color C and sends it to the command manager.

This gets repeated for the other 3 colors. So far the tool generated 4 commands. If at any point the user presses Ctrl + Z, only one of the strokes disappears.

The user confirms the hint strokes are complete and the segmentation can start.

The tool now generates the final command to complete the segmentation and passes it to the manager.

If the user presses Ctrl + Z now, only the segmentation will get removed, with the strokes still remaining on the object.

4.5 Geometry model

The geometry model is responsible for keeping the geometry data (triangles of the model) in memory and implementing geometric operations, that then get used in commands, to perform the tasks specified by the tools.

4.5.1 Implementation variants

There are two main approaches to programming the geometry model. One approach is to look at the model as a state-less chunk of data. The functions that operate on the model (e.g. *return the triangle that intersects this ray*) are free functions that just get called on the geometry model. The second approach is to implement the model as a full object. This means that the model has its private data – the triangles of the user's model, and it has its methods – the geometric operations it allows the user (i.e. the programmer of the commands and tools) to perform upon the private data.

Both solutions have their pros and cons and we, at the time of writing, do not know which will shape up to be a better fit to the application. We will briefly mention some of the reasons we might choose either one. The state-less *struct* version is better for handling the data. We will need to make some kinds of copies both for saving the model on the disc and for the *undo Undo and Redo* operations. This approach would allow us to simply copy the struct and not waste any space or time. The object-oriented approach is easier to work with and probably less confusing for new programmers trying to implement a new tool or other extensions to the application.

4.5.2 Libraries

There are several libraries that the geometry model could benefit from. We have been searching on the internet and did research and we found the following libraries that we will try to use to augment the geometry model.

- **Assimp**[1] – a portable Open Source library to import various well-known 3D model formats in a uniform manner[2]. This library is very important to our application as it will allow us to support a plethora of 3D formats for the users to use. This should help the less experienced users by not forcing

[1] http://www.assimp.org/

[2] Description taken from http://www.assimp.org/

them to convert their objects to different formats before our program can import them.

- **cereal**[3] – cereal is a header-only C++11 serialization library. cereal takes arbitrary data types and reversibly turns them into different representations, such as compact binary encodings[4]. We need to serialize data to enable saving the current state of the user's project on disc, and then be able to de-serialize them when the user wishes to continue working on the saved project and presses *load existing project*. We have had positive experience with cereal and that is the reason we chose this library.

- **The Computational Geometry Algorithms Library or CGAL**[5] – a software project that provides easy access to efficient and reliable geometric algorithms in the form of a C++ library[6]. We have heard very positive feedback on CGAL and therefore will attempt to use the library for the more complex geometry problems. We have, however, not had any previous experiences with CGAL, so we are unsure as to how well it will fulfill our requirements and might drop the use of it if it does not meet our expectations.

[3]https://uscilab.github.io/cereal/
[4]Description taken from https://uscilab.github.io/cereal/
[5]https://www.cgal.org/
[6]Description taken from https://www.cgal.org/

Chapter 5

User interface

A *user interface* (UI) is the front-facing part of Pepr3D that the users are going to interact with. It is responsible for managing windows, showing buttons, texts, rendering the 3D model, handling mouse clicks, and much more. In case of Pepr3D, it needs to be a cross-platform, easy-to-use, intuitive, and fast abstraction of the complex 3D geometric algorithms at the backend, see Figure 5.1.

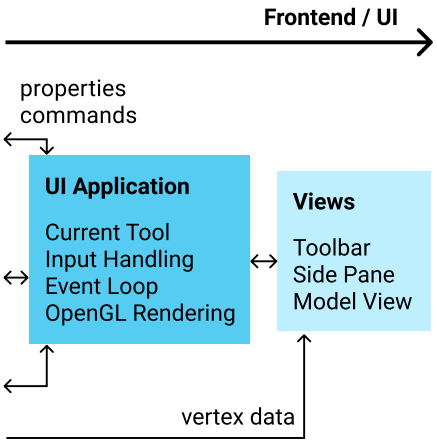

Figure 5.1: An overview of the Pepr3D UI architecture, based on Figure 3.1.

5.1 Introduction

We did not want to reinvent the wheel, so we investigated how other developers recommend to implement user interfaces. In our case, we have *3D rendering*, i.e., displaying the 3D model, and *UI widgets*, i.e., the windows, buttons, check boxes, text labels, etc. Here we describe what we found important to understand.

5.1.1 Existing patterns

Let us have a look at existing generic UI architectural patterns. A detailed overview of them was written for example by Derek Greer[1]. The most common pattern is called Model-View-Controller (MVC), where *model* is a state, *views* visualize the state, and *controllers* react to user input to manipulate the model. The MVC pattern got so famous that there are a lot of alternatives nowadays

[1]https://lostechies.com/derekgreer/2007/08/25/interactive-application-architecture/

based on the similar principles, like Model-View-Viewmodel (MVVM), Model-View-Presenter (MVP), or Presentation-Abstraction-Control (PAC).

But we can go even further: Johannes Norneby mentions[2] a common paradigm of UI, which he objects is *not valid*: *"The user interface and / or visualization of any program is inherently stateful."* He objects that this is a broken paradigm and devotes his book into explaining the so called *immediate user interface*, which instead provides a *stateless* alternative to rendering UI.

The main difference between *immediate* and *retained* modes is that in the latter, the visualization library *retains* internally a complete model (state) of objects to be rendered, while the former is procedural and redrawn every frame.[3] The major benefit of a stateless immediate UI is the ability to maintain and reuse it much easier. Norneby suggests that every *view* should be as *pure* as possible, meaning that in languages like C++, all views should in fact be *free functions*.

5.1.2 Immediate vs. retained

Even when one sticks to MVC principles, there does not seem to be a consensus for which applications one should prefer the *retained* mode over *immediate* and vice versa. At its core, MVC principles can be used in both of them. Norneby goes as far as saying that MVC *and* immediate UI are two implicitly connected concepts. Arguments were made[4] for both approaches without a clear winner.

The main downside of retained UI is the necessity to maintain a UI state. This often leads to complex libraries that are difficult to learn to work with and introduces out-of-sync bugs that are hard to fix. This is why video game and interactive applications developers (including Blizzard Entertainment) support immediate UI, because it *interlocks* the application data and the current state of the UI, meaning the state and UI never get ouf of sync. The libraries are also usually very simple to use.

The main downside of immediate UI is a poor separation of logic and presentation and the necessity to rerender the UI more often. What developers at uiink[5] suggest is to just use the best of the both worlds. And it is not different from what Norneby actually proposed in his never-finished book. We should only use immediate UI in the actual *views*, which are just free functions procedurally explaining how the UI should be rendered each frame. The rest of the application should know nothing about immediate UI. In theory, we should be able to swap immediate UI and retained UI, or use both of them together without the need to touch the rest of the application.

5.1.3 Real-time rendering

In Pepr3D, a regular user interface with a few buttons and texts is not enough. We primarily need real-time 3D rendering and manipulation of the 3D model that the user is editing. Hence the whole user interface needs to take this into account and should be primarily based on real-time rendering.

[2] http://www.johno.se/book/immvc.html
[3] http://msdn.microsoft.com/en-us/library/windows/desktop/ff684178(v=vs.85).aspx
[4] https://gamedev.stackexchange.com/questions/24103/immediate-gui-yae-or-nay
[5] https://uiink.com/articles/data-driven-immediate-mode-ui/

As already mentioned, real-time applications such as video games favor immediate UI. They need to rerender the whole screen all the time anyway. For Pepr3D, it is perfectly possible to make immediate UI a part of the renderer.

5.1.4 Presentation separated from logic

Nowadays, a lot of UI is being developed for web applications. We can investigate the most used frameworks and libraries for single-page applications[6]: React by Facebook, Angular by Google, or Vue.js.

We observed that these tend to follow the principle that a *view should be just a thin front-facing layer only responsible for displaying data.* Calculations and data manipulation should be done in other parts of the application.

It is not a surprise that even Qt[7], a widely used C++ UI framework, encourages people to eliminate data consistency problems by using separate *views*.[8] Even though there are so many different UI libraries and frameworks, they all seem to share the same common principles about the separation of presentation.

5.1.5 Internationalization and accessibility

There are many other observations one can make when studying existing applications that heavily rely on user interface. A lot of energy has been invested into creating standards and guidelines for them. It is not in the scope of this work to list all details about building good user interfaces, but we should still mention at least two more concepts: internationalization and accessibility.

Typically, when applications are used by users from different countries, the UI needs to support *internationalization* (abbreviated as *i18n*)[9], i.e., different languages, number formats, time formats, etc.

Applications should also be *accessible* (accessibility, abbr. *a11y*), meaning they should support keyboard navigation for people who cannot use mouse, screen readers for people who are blind, high contrast themes for people with worse eyesight or color blind users, etc. Especially in the "web world", there exist important accessibility guidelines called WCAG.[10]

5.2 Our requirements

Based on the observations made in the previous section, on the expected usage of our application, and on general advice gathered from Vojtěch Bubník from Prusa Research s.r.o., we decided on the following set of requirements for the user interface of Pepr3D.

The user interface of Pepr3D and the library we are going to use for it should:

1. separate presentation from application logic, i.e., in theory, we should be able to easily replace the UI with another one, should it be necessary,

[6]React: https://reactjs.org/, Angular: https://angular.io/, Vue.js: https://vuejs.org/
[7]https://www.qt.io/
[8]http://doc.qt.io/qt-5/modelview.html
[9]https://blog.mozilla.org/l10n/2011/12/14/i18n-vs-l10n-whats-the-diff/
[10]https://www.w3.org/WAI/standards-guidelines/wcag/

2. support real-time 3D rendering of the 3D model, provide an easy-to-use abstraction, e.g., for rendering 3D primitives, using custom shaders with uniforms, uploading textures to the GPU, keeping constant framerate, etc.,
3. look visually good and allow us to unify the design of the 3D rendering part and the rest (toolbar, controls, etc.), e.g., by supporting custom themes,
4. be cross-platform at least on desktop (Windows, Mac, Linux), ideally on tablets as well (Android, iOS), i.e., the "cross-platformity" of Pepr3D should not be limited by the UI library,
5. support keyboard navigation, e.g., tabbing to buttons and input elements, using keyboard to enter values,
6. support high DPI, e.g., Apple Retina displays, Microsoft Windows scaling,
7. support asynchronous events, e.g., long calculations on background should not affect the UI thread,
8. support internationalization including plurals, time formats, UTF-8, and
9. the license of such library should be as least restrictive as possible, e.g., allowing commercial usage and redistribution, should the development of Pepr3D continue after this initial school project is finished.

5.3 Choosing a 3D rendering library

There are many cross-platform C++ libraries for 3D rendering and for creating user interfaces. Picking the right ones for Pepr3D is not an easy task. Fortunately, as we built a list of requirements in the previous section, we can easily disregard the libraries that do not conform to our requirements. Let us now describe how we chose a library for the 3D rendering part of Pepr3D.

In order to support multiple platforms and even older computers, we decided to use OpenGL rendering API instead of Microsoft DirectX or alternatives like Vulkan. We cannot use OpenGL on its own as we need a library to handle cross-platform windows, contexts, keyboard and mouse inputs, etc., so it is necessary to find a library that can help us with that.

There are many libraries like SDL, GLFW, Cinder, Ogre3D, or bgfx,[11] and some UI libraries like Qt can also help with that. Some of these libraries depend on others from the list, for example bgfx uses SDL for windows and input handling, and Cinder uses native code for Windows and OS X but GLFW on Linux.

We had previous knowledge of Cinder and bgfx. The other libraries we examined did not seem to provide any advantages over these two, either because they were already included in the two, or because they were too heavy.

We found that Cinder conforms to our requirements better than bgfx: Cinder has built-in high DPI support, font rendering, event loop for asynchronous events, a big set of tutorials and examples, and much more. We have decided to choose **Cinder** as the library for real-time 3D rendering.

[11] https://www.libsdl.org/, https://www.glfw.org/, https://libcinder.org/, https://www.ogre3d.org/, https://github.com/bkaradzic/bgfx

5.4 Choosing a UI widgets library

Now that we know what library to use for 3D rendering, we need to choose a way to display *widgets* such as toolbars, buttons, check boxes, or text labels. Again, there are famous cross-platform libraries that already exist for these purposes, so it would not make much sense to make our custom solution.

5.4.1 Why not wxWidgets nor GTK+

We should definitely mention retained UI libraries wxWidgets and GTK+.[12] They are both cross-platform and used by famous software like GIMP or Audacity. Unfortunately, we found major flaws with both of them.

Regarding wxWidgets, we did not really like its default appearance. It uses native controls where possible making theming very limited and also undocumented. Hence, we would not be able to easily unify the design of the 3D view and the rest of the application. Also, only desktop is supported.

Regarding GTK+, they do support theming up to some degree, they also added OpenGL rendering widgets a few years ago. Making Cinder and GTK+ work together would probably cost us some effort as we did not find any already working solution. The problem with GTK+ is that a lot of developers who actually use it are not satisfied and warn others about using it.[13,14,15]

They say that GTK+ documentation is very bad and that different versions of GTK+ break existing applications, extensions, and themes, because the API and ABI is changing rapidly providing no guarantees. It did not seem that using GTK+ for Pepr3D would be a good long-term idea should anyone continue with its development in the future.

5.4.2 Qt

We have already mentioned Qt on previous pages of this specification. It is a rather large actively-developed library providing a lot of features including their own internationalization solutions and so on. Qt conforms to all our requirements stated in the previous section. There are two major drawbacks with Qt: its controversial licenses[16] and its huge size.

The licensing is controversial because either one can pay for the commercial license, or one can use the LGPLV3 one, but it requires dynamic linking, providing users the ability to relink the application, and also the necessity to deliver complete Qt source codes to users including all changes made if any. The huge size is also an issue, because using only the basics of Qt (widgets, GUI, and core) is already around 17 megabytes in libraries, which together with Cinder would lead to a very large size of the final Pepr3D application. It is also uncertain whether it would be a good idea to use Cinder together with Qt, so we would possibly need to rely on a different library.

[12] https://www.wxwidgets.org/, https://www.gtk.org/

[13] https://davmac.wordpress.com/2016/07/05/why-do-we-keep-building-rotten-foundations/

[14] https://fosspost.org/opinions/are-gtk-developers-destroying-linux-desktop-with-their-plans

[15] https://www.reddit.com/r/linuxmasterrace/comments/7xkcwo/

[16] https://www1.qt.io/licensing-comparison/

5.4.3 UI libraries for OpenGL

There are also libraries that do not use native controls at all, but rather generate draw instructions and lists that can be used by renderers like OpenGL directly. The libraries itself do not handle window creation, native calls to operating systems, etc. Users of such a library need to bind the input handling and draw commands of these libraries to their own OpenGL/DirectX/other renderer. In our case, we would need to connect the library to Cinder, which handles windows, inputs, and rendering itself.

There are many such libraries, e.g., Dear ImGui, Nuklear, NanoGUI, and FlatUI.[17] While some of them like NanoGUI and FlatUI seem to be rather small, without that many users, and not under active development, Nuklear and Dear ImGui are still under active development and maintenance.

Nuklear is an ANSI C header-only library with a C API and C naming conventions. We did not manage to find existing Cinder–Nuklear bindings that we would be able to use, so we would need to develop them ourselves. For this reason, we did not continue investigating Nuklear, because we found an alternative.

Dear ImGui (or just ImGui) is a modern bloat-free C++11 library that we already mentioned in the previous sections. It is backed by large companies like Blizzard Entertainment or NADEO. Its community is very active providing different bindings for different renderers and libraries including Cinder. It is easily themable and we have created a prototype with completely custom controls.

5.4.4 Final decisions

For our final decisions, we have selected two libraries: **Qt** and **ImGui**. When looking at our requirements from Section 5.2 (numbering refers to Section 5.2):

1. separation can easily be achieved in both using Views and Models,
2. OpenGL rendering is implicit in ImGui, there is a widget in Qt,
3. theming in Qt: QSS stylesheets, in ImGui: styles and custom drawing,
4. both are cross-platform even on mobile devices,
5. both support keyboard navigation, ImGui since version 1.60,
6. high DPI possible in Qt, for ImGui we can use Cinder high DPI support,
7. Qt has its own thread pool, signals, and promises, for ImGui we can use C++11 and Cinder/ASIO event loop using dispatchAsync,
8. Qt has its own i18n support, for ImGui we can use PO files and gettext[18] together with a translation editor, e.g., open-source PoEdit[19],
9. Qt is unfortunately commercial or LGPLV3 (see above), ImGui has much less restrictive MIT License.

As we can see, there is no clear winner: both libraries have positive and negative attributes. In fact, Qt and ImGui are very different. Qt is a large retained UI library and ImGui is a small bloat-free immediate UI library.

[17] https://github.com/ocornut/imgui, https://github.com/vurtun/nuklear, https://github.com/wjakob/nanogui, https://github.com/google/flatui

[18] Free i18n system commonly used on Linux, see https://www.gnu.org/software/gettext/

[19] https://poedit.net/

Using Qt together with Cinder is rather questionable as they both overlap in certain areas like window management and event handling. Whereas ImGui needs a renderer and a window handler anyway, so using it with Cinder seems to be a good idea. ImGui is much closer related to the actual OpenGL rendering and offers us quite a bit more flexibility. It also has a very simple source code that one can read in an evening, meaning we can actually learn a lot about how such a library is made.

We think that Qt would probably be an unecessary huge piece of middleware that we would have to learn just for the sake of this project. We have decided to use **ImGui** for the Pepr3D user interface.

5.5 Final proposal

In Section 2.1 and in Figure 2.1, we have already proposed a wireframe of Pepr3D and explained why we find it reasonable and easy to use. In this chapter, we explained our requirements and investigated existing patterns for actually implementing the UI. Let us now propose how we can use these together.

5.5.1 Overview

We propose to divide the Pepr3D UI into the following main parts that correspond to the wireframe in Figure 2.1 and to Figure 5.2:

- a **toolbar** with toggleable buttons representing tools, undo/redo, etc., implemented as an ImGui widget rendered with Cinder,
- a **side pane** with buttons, checkboxes, sliders, etc., representing configuration of the currently selected tool, also implemented as an ImGui widget rendered with Cinder,
- and a **model view** with the 3D model which the user can rotate, zoom, paint on it, etc., implemented in OpenGL using Cinder.

Even though the toolbar and side pane are to be implemented in ImGui, they will in fact be rendered using Cinder and OpenGL as well. The whole window is managed by Cinder and has a single large OpenGL drawing context.

5.5.2 Stateless views / widgets

Inspired by the immediate UI advice, we propose to use *stateless views* that will be responsible for drawing the UI. We can implement these views as C++ free functions without encapsulating them in any classes. This is a common pattern in C++, e.g., in the standard library, because unlike in languages like Java and C#, functions in C++ do not need to be enclosed in classes.

Having stateless views means that we do not have to program any explicit synchronization between the UI and the backend. Whenever we rerender the UI, it will be rendered with the newest data. We should try to avoid retained state in the UI as much as possible, but it might be necessary in specific situations, e.g., for scrollbar positions or complex calculations that we do not want to do each frame. The ImGui library provides ways for retaining states. Where necessary, we can also use regular C++ classes and their instances.

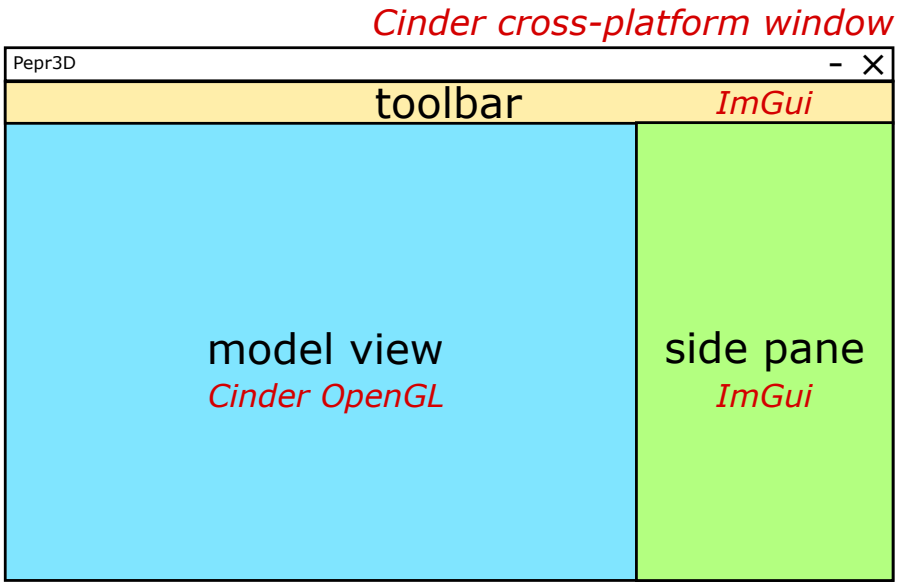

Figure 5.2: An overview of the Pepr3D UI.

There is also a huge ImGui community at GitHub, where we can find both simple and complex widgets available from other users of the library. We should aim at making our UI composed of reusable ImGui widgets.

5.5.3 Internationalization and accessibility

It should be possible to change a language of the application. For this purpose, we propose to use PO files that are used by a lot of Linux distributions and applications including PHP. Loading the PO files can be handled by lightweight libraries that already exist and are suitable for our project, at a first glance spirit-po[20] looks perfect for our purposes. The files themselves can be generated from source codes and translated using for example the already mentioned PoEdit.

The UI should provide basic keyboard navigation including hotkeys (shortcuts). It is especially important that the currently selected tool can be changed by pressing a single key on a keyboard. We also considered adding a radial menu to a specific mouse button (e.g., middle button click), but it is not a top priority. The default theme of Pepr3D should provide sufficient contrast and it should be easy to see which tool is selected and what options are enabled.

[20] https://github.com/cbeck88/spirit-po

Part III

Execution and minimal implementation

Chapter 6

Execution of the project

6.1 Todo by Jindra

Chapter 7

Minimal Implementation

7.1 Todo by Jindra