Pepr3D

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

Faculty of Mathematics and Physics
Charles University

Contents

Part I

Introduction

Chapter 1

Introduction

In this project, we aim to create an intuitive application that allows the user to interactively color a 3D model and export it in a 3D printable format. This chapter will provide a brief summary of the 3D printing environment, its pipeline and the goals of the project which we set in the beginning.

1.1 3D printing basics

3D printing is a new technology that has seen rapid development 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 purpose 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 transferred 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 cumbersome 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 toothless 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-coloured 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 allows 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).

The main goal is to make the application for desktop PCs, with main development time being focused on the Windows operating system. We did, however, 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]http://slic3r.org/

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

Chapter 2

Related work

Based on our own research and 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. Here we present a simple list of software that could be used to achieve the same results as our program delivers. We highlight the pros and cons of each program to show that our goal is sufficiently unique.

2.1 Autodesk Meshmixer

The closest existing software is Autodesk Meshmixer [1], 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.

It is, probably, meant for more advanced users than Pepr3D. It is (at least in our opinion) unintuitive and can be slow at times. The colouring and segmentation of the model can only be done one color at a time – the user colors one region, separates it and then continues with the next colour, tearing the model into pieces piece by piece.

The export has its problems and we have found issues with the exported objects. The objects were sometimes not printable at all.

It can do a model segmentation, which is analogous to our *Automatic segmentation* tool. It also can select an area based on given parameters and a region border, which is a variation on our *Bucket painter* tool.

On the other hand it is not capable of doing custom strokes like our *Brush* tool or custom text like our *Text* tool.

2.2 Microsoft 3D Builder

Microsoft 3D Builder [2] is another application that handles 3D models but we have not found a way to make it create anything remotely applicable to FDM printing.

It is a really fast 3D model previewer, but it does not support any functionality related to 3D printing, such as separation of objects, colouring of triangles and so on.

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

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

The user can create a very simple 3D models inside this application.

2.3 Complex 3D editors

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-consuming for the user and practically unusable on a larger scale. We discuss **Blender** briefly.

Blender is a fully fledged 3D editor, and therefore is capable of creating any model imaginable. To replicate the desired outcome of Pepr3D, the user has to load the model into Blender, divide the model into several triangle meshes, which only results in the parts of the hull of the object, not a closed polyhedron. The user has to then manually close each part of the hull, fill the holes and ensure the Slic3r can solve the intersections correctly. It does not support any kind of automated separation of one model into several pieces using the triangle colors.

Part II

Developer Documentation

Chapter 3

Architecture

In this chapter we present the overall architecture of the program, describe the individual components briefly and showcase some basic data-flow within the program. In the following chapters we focus on the main components and describe each in detail.

3.1 Architecture overview

The following Figure 3.1 illustrates the main components in the architecture of the Pepr3D application. Each colour represents a separate component, while the arrows denote how data typically flow and how components usually communicate.

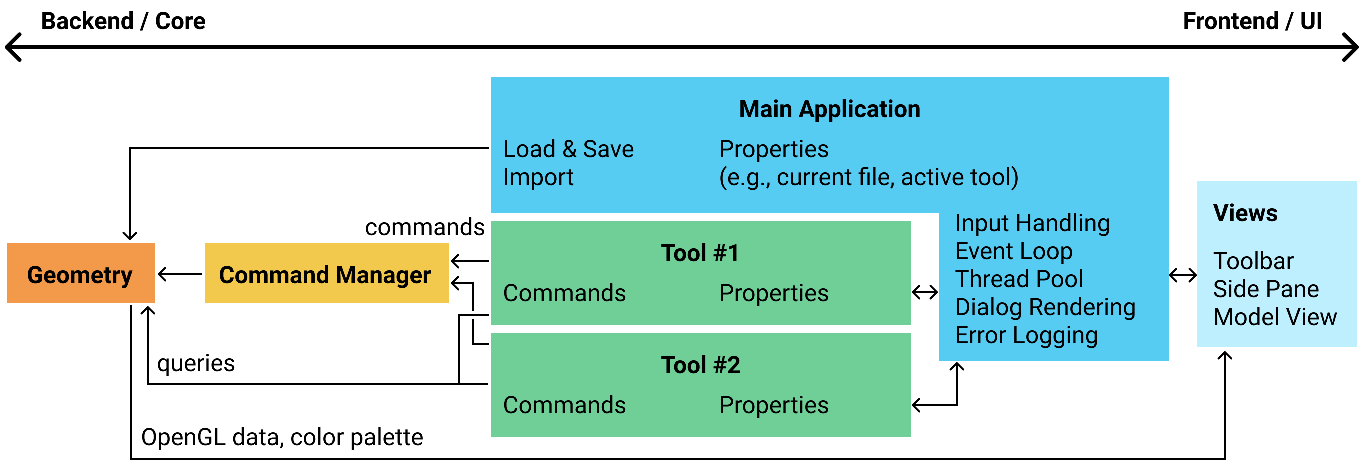


Figure 3.1: An overview of the Pepr3D architecture.

- **Main Application** is the owner of the whole architecture. It contains functionality which is important for the whole application (like thread pooling or error logging) and owns and contains the other data structures. It acts as a single source of truth, holding the data that is valid at every moment (like the selected tool or the current model). This data is then fetched from the `MainApplication` by other components.

- **Views** are the elements that fill the screen of the application. They handle rendering the user interface and allowing the user to perform basic actions to handle Pepr3D like rotating and zooming in on the model in `ModelView`, choosing a tool from the `Toolbar` or changing the tool's properties in `SidePane`.

- **Tools** are the components that allow the user to interact with the data. Each tool has its properties that can be changed by the user via the user interface. If the user wishes to perform an action that would change the geometry (like colouring a triangle), the tool invokes a command which will take care of the action. The tool specifies how the right panel (`SidePane`) will look for each tool and can also query the `Geometry` class for simple information like the hovered triangle id.

- **Command Manager and Commands** are the way Pepr3D handles the *Undo* and *Redo* operations. A `Command` is a single operation that can be ran by the `CommandManager` or joined with other `Commands`. A simple example of a command is the `PaintSingleColor` command. It takes a list of triangle ids and a color, and simply changes the color of those triangles to the new color. It can be joined with other `PaintSingleColor` commands, which have the same color by merging the two triangle id lists. The `CommandManager` then holds the stack of applied commands, saves the snapshots of the current geometry and is able to recreate any state by reverting to the older snapshots and running the commands again.

- **Geometry** is the main data structure of the application. It holds the data of the triangles, their colors, normals, etc. It also generates the buffers for OpenGL and provides an API for the Commands (for non-`const` purposes) and Tools (for `const` purposes such as finding the hovered triangle id).

The arrows between these components show the typical directions of communication. We can see, for example, that only `CommandManager` and the `MainApplication` send data to `Geometry`, while the user interface does not know anything about the `CommandManager`. Not every single communication is represented here, as for example certain Tools need read-only access to Geometry.

During the development we tried to separate the components as much as possible. Since our application is pretty basic, the pipeline is still connected quite a lot. We hope that this design is extendible though, and if the need arises, new components can be easily inserted into the overall architecture as illustrated in Figure 3.1.

3.2 Data flow

In this section we demonstrate the data flow on a simple chain of operations an average user might perform. The sequence is the following and can be executed any time after launching the application: *Import a model → Select Bucket Painter tool → Change settings on Bucket Painter → Paint with the Bucket Painter → Undo the operation.*

1. **Importing a model** causes the `MainApplication` to launch an asynchronous function that loads the new model, calculates the data for both the `MainApplication` and `Geometry` objects. This information includes the data structures for geometry operations or the calculations to re-orient the model correctly in the `ModelView`. After this step is done, `MainApplication` gets updated and holds the new `Geometry` object, which changes the geometry.

2. **Selecting the Bucket Painter** is, according to Figure 3.1, a change of the `MainApplication`'s property. This is done via the `Toolbar` view. Upon selecting the new tool, when the next frame starts rendering, the `SidePane` will query the `MainApplication` object for the current tool, which will be the Bucket Painter. The `SidePane` will then let the Bucket Painter render its own settings within the `SidePane`.

3. **Changing Bucket Painter's settings** is changing the tool's properties. It is done within the `SidePane` view, which the Bucket Painter fills with relevant UI widgets (buttons, checkboxes, etc.). Changing these widgets will change the Bucket Painter's properties.

4. **Painting with Bucket Painter** by clicking on a certain triangle of the model. By clicking into the model view, an event gets created. The event is first processed in the `MainApplication` (e.g. if it is a *drag and drop* event), then gets evaluated in `ModelView` (e.g. if it is a right click to rotate the view) and the `ModelView` then passes it into the tool, if it is required. The Bucket Painter gets the left click, queries the `Geometry` for the hovered triangle id, queries the `Geometry` again for the bucket spread and then executes a `PaintSingleColor` command. This command gets enqueued into the `CommandManager`, which executes it, changing the `Geometry`. In the next rendering frame, the `ModelView` notices, that the OpenGL geometry buffers are dirty, asks the `Geometry` to recalculate the buffers and renders the new colors.

5. **Undoing the last command** by clicking the *Undo* button invokes the `CommandManager`. It restores the last snapshot of the `Geometry`, then re-runs all commands except the last one and updates the `Geometry`. This effectively restores the state before clicking with the Bucket Painter.

This simple sequence of operations describes the majority of the program runtime data flow and control handling. It highlights important concepts of the UI – such as the tool's user interface being rendered by the tool itself or the `CommandManager` semantics.

Chapter 4

Geometry

In this chapter we describe the `Geometry` class in detail. We also discuss the other helper classes which handle geometry in Pepr3D.

4.1 Geometry

4.1.1 Geometry

`Geometry` is the main class of the whole data pipeline in Pepr3D. Here we explain the high-level details of this class, to hopefully give the reader a broader picture of the architecture, which makes navigating the code much easier. Since the class is rather large, we will not present a complete overview as in the following sections, but rather discuss several parts separately.

Public data members

The class itself does not provide any data as public. This is because it holds the geometry model, which cannot and should not be easily altered, since altering requires non-trivial re-computations (such as the acceleration tree structure or the Polyhedral surface data structure).

Private data members

All of the class' data is stored in private members, we present a list of the most important ones here:

```
std::vector<DataTriangle> mTriangles;
std::map<size_t, TriangleDetail> mTriangleDetails;

std::unique_ptr<Tree> mTree;
std::unique_ptr<Tree> mTreeDetailed;

PolyhedronData mPolyhedronData;
std::unique_ptr<PolyhedronData::Mesh> mMeshDetailed;

std::unique_ptr<BoundingBox> mBoundingBox;
ColorManager mColorManager;
```

```
friend class cereal::access;
OpenGlData mOgl;
```

As we can see, the members are of two main functions: the ones holding the actual data (such as `mTriangles` and `mTriangleDetails`) and data structures built upon the data, which help us perform the operations quickly. There are several more conversion `std::map` objects which help the user retrieve the correct triangle data with an arbitrary information about the triangle (an id, a pointer in the surface mesh, etc.), which are not listed here for clarity's sake.

The `mTriangles` array functions as one would expect – this is the main array, which stores the basic information for each triangle (the position, normals, etc.). If the *Brush* or *Text* tools are used, the triangles get split into detailed ones, and `mTriangleDetails` needs to be utilized to look up information about a triangle.

The `Tree` structure is CGAL's AABB hierarchical search tree which allows us to intersect the model with rays. This is especially useful during the user interaction (retrieving the triangle id which corresponds to the user-clicked triangle).

We also point out the duality of many of these data structures – `mTree` and it's detailed version, as well as the `PolyhedronData` object and it's detailed version. This is required, because the user can change the geometry of the model extensively by adding new details. Changing the geometry then alters the behaviour of several algorithms – a *breath-first* search behaves differently on a non-detailed model and a detailed model. However, completely discarding the non-detailed version is not wise either, since some operations may be faster to perform on the non-detailed versions, if the result does not depend on whether the mesh is detailed or not (for example, finding a position of ray-model intersection does not depend on any added details and should utilize the `mTree` member).

The last notable members are the **cereal** friend access and **OpenGL** buffer object. Cereal requires friend access to the class if we want the class to be exported. Since `Geometry` is the most important data object in Pepr3D, we obviously want to be able to save parts of it. The `OpenGlData` object is used to transport the geometry to the front-end rendering part of Pepr3D. The `Geometry` class creates and updates the buffers, and the front-end only re-renders the model when the buffers are changed. This way the front-end does not have to have any knowledge about the geometry and can be kept separate.

Public methods

The main way to work with the `Geometry` class. They can be divided into two obvious categories: the *getters* and the rest.

We provide a lot of getter methods, so the user has a good way of communicating with the object and can extract as much information out as he wants. Most of the getter methods are `const`, since changing certain members is not allowed. There are, however, several which permit the user to change them, since the effect is immediate and doesn't have deeper consequences (like changing the colour palette in the `ColorManager`).

The other category of the methods are the algorithms used to implement Pepr3D's tools, load the data correctly and compute the necessary information. We list a few of the most important ones here:

```
// Loading and computation
void computeSdfValues();
void loadNewGeometry(const std::string& fileName);
void recomputeFromData();
GeometryState saveState() const;
void loadState(const GeometryState&);

// Algorithms
std::optional<size_t> intersectMesh(const ci::Ray& ray) const;
template <typename StoppingCondition>
std::vector<size_t> bucket(const size_t startTriangle, const
   StoppingCondition& stopFunctor);
void paintAreaWithSphere(const ci::Ray& ray, const
   BrushSettings& settings);
void paintWithShape(const ci::Ray& ray, const
   std::vector<Point3>& shape, size_t color, bool
   paintBackfaces = false);
size_t segmentation( ... );
```

The first few methods are the main way to alter the `Geometry` and perform computations on the loaded model. The first batch of methods provides the tools to load and save the model correctly, as well as recompute the data if required (e.g. if a bug was encountered and the data is corrupted). Many of these methods utilize the `GeometryProgress` capability of Pepr3D, which allows the user to know how long the selected operation will run, or at least re-assures him that Pepr3D has not frozen.

The second group of methods are the algorithms employed to implement all the tools in the release version. We describe these methods in greater detail here:

- `intersectMesh(ray)` is the main method used to transform the user's clicks with tools into the geometry pipeline. Given a ray, this returns the id of the triangle which got intersected by the ray. Modern `std::optional` is utilized to avoid various "return codes" (like returning -1 for *no hit*).

- `bucket(start, stopCriterion)` is used in several tools and implements the *breath-first* search over a 3D triangle mesh. The *BFS* traverses the mesh triangle to triangle until it is stopped by the stopping criterion. This criterion corresponds to the user interface of the *Bucket* tool – the user can either select to never stop, to stop upon meeting a different color, to stop on sharp edges or a combination of both. Modern C++ template approach is utilized to replace the runtime `if-else` decisions with compile time template processing.

- `paintWith` are two methods used by the *Brush* and *Text* tools. They allow to paint with selected settings along a target ray. `paintAreaWithSphere` only paints a circular region on the surface of the mesh, where the circle is computed as the intersection of a sphere placed on the surface of the mesh with the mesh's sides. `paintWithShape` accepts a `shape` vector composed of points in 3D space and allows the user to paint a custom shape onto the mesh's surface.

- `segmentation` implements the *Automatic segmentation* tool by using the CGAL library, namely the *Triangulated Surface Mesh Segmentation* [1]. This process is dependant on the SDF values being calculated, which can be done with the computation method `computeSdfValues`.

Private methods

We will only describe a handful of private methods, since they are largely unimportant and only contain technical details which are not required for a general understanding. We will, however, mention the **OpenGL** buffer generators, since this is something not obvious from the first look on the `Geometry` class. As we have already mentioned above, the `Geometry` class handles all the buffer creation and keeps the buffers up-to-date, which allows us to shield the front end from the `Geometry` structure completely. If the structure of the geometry class changes, no further changes in the rendering front end code are required. The only thing which is required is to keep these methods correct. Namely:

```
void generateVertexBuffer();
void generateIndexBuffer();
void generateColorBuffer();
void generateNormalBuffer();
void generateHighlightBuffer();
```

Further information is available in the *doxygen* documentation or directly in the code.

4.1.2 Triangle Detail

`TriangleDetail`

The `TriangleDetail` class allows the user to divide a triangle of the original mesh into smaller, differently coloured pieces. The surface of a `TriangleDetail` can be represented in two formats. We first explain the core concepts of this class before giving a brief overview of its members.

Internal format

The **primary format** is a collection of PolygonSets. For each color in the triangle, a new PolygonSet is created. The union of all polygons from all PolygonSets would exactly fill the surface of the original triangle. These PolygonSets are stored in `mColoredPolys_` map. All painting methods on TriangleDetail, with the exception of painting individual triangles, utilize this representation.

The **secondary format** is a collection of colored triangles, which are the result of a triangulation of the PolygonSets. Usually, this representation is only temporary, as it gets overwritten each time a new painting operation changes the PolygonSets. These coloured triangles are stored in `mTrianglesExact` array.

When the user changes the color of one of these detail triangles (from the secondary format), the PolygonSet representation no longer has the correct color stored. This discrepancy is marked via `mColorChanged` boolean flag. When

[1]https://doc.cgal.org/latest/Surface_mesh_segmentation/index.html

this flag is set, any method that works with PolygonSets must first update the PolygonSets from the coloured triangles.

This duality of formats allows us to use PolygonSets – required for CGAL boolean operations, and triangles – required for rendering and other Pepr3D tools.

Numerical precision

To be numerically stable, TriangleDetail uses an exact Kernel type, provided by the CGAL library. The exact kernel is used for all internal "'Polygon"' and exact triangle operations. While an exact kernel is being used internally, all public methods and their results are using an ordinary, limited-precision kernel. To distinguish between different precision types used in TriangleDetail, the following convention is used:

Types defined on exact kernel (`TriangleDetail::K`) use shorthand CGAL type names: - `Polygon`, `Point3`, `Triangle2`, `Segment2`, etc.

Types defined on inexact kernel (`DataTriangle::K`) use the "Pepr" prefix before their shorthand CGAL type name: - `PeprPoint2`, `PeprVector3`, `PeprTriangle2`, etc.

The "Pepr" prefix signifies the type is the same, as in the rest of the Pepr3D application and can be therefore passed to other methods and tools.

Class overview

We now provide a short overview of the `TriangleDetail` class. Due to the complexity of the class, we will only discuss the public methods of the class (since public members do not exist). We have discussed the private workings of the class in the previous paragraphs. Following, are the most important methods of the class.

```
void paintSphere(const PeprSphere& sphere, int minSegments,
   size_t color);
void paintShape(const std::vector<PeprPoint3>& shape, const
   PeprVector3& direction, size_t color);
void paintShape(const std::vector<PeprTriangle>& triangles,
   const PeprVector3& direction, size_t color);

void addPolygon(const Polygon& poly, size_t color);
void addPolygonSet(PolygonSet& polySet, size_t color);

void save(Archive& archive) const;
void load(Archive& archive);

static std::vector<Triangle2> triangulatePolygon(const
   PolygonWithHoles& poly);
```

There are several groups of methods which work the same way. The `paint` methods take a input object (a sphere or a arbitrary shape) and paint it into the triangle detail – creating new polygons when necessary. The `add` methods can be used to manipulate the internal state of the class, either adding new polygon sets (e.g. if new colours are required) or adding new polygons to increase detail. Since Pepr3D saves objects of this class, there have to be `save` and `load` methods present, for the Cereal library. The `triangulatePolygon` method further highlights this duality of the class, allowing to create triangles from the existing polygons.

4.1.3 Detailed Triangle Id

This `struct` is a very simple object, but important to understand, so we present it here, even though it is not a complex or confusing concept. It is a way to connect the original triangles with the detailed ones, once the user paints with the *Brush* or uses the *Text* tool. Before the user uses these tools, each triangle has its own id – a `size_t` type number. However, once the triangles are subdivided by use of one of these tools, this is no longer the case, since we are adding new triangles. The tools also support re-merging and simplifying the topology if the user removes the details by painting over them again. This would not be possible if we simply added the new triangles to the end and kept indexing by the `size_t`, hence why we use this, admittedly more complex way of indexing.

```
struct DetailedTriangleId {
   DetailedTriangleId()
   explicit DetailedTriangleId(size_t baseId,
      std::optional<size_t> detailId = {})

   size_t getBaseId() const;
   std::optional<size_t> getDetailId();

   bool operator==(const DetailedTriangleId& other) const;

  private:
   size_t mBaseId;
   std::optional<size_t> mDetailId;
};
```

The `BaseId` property is the triangle id of the original triangle, that has been split into many more. The `DetailId` is the id of the one of the smaller triangles, which form the original one (the one identified by `BaseId`).

A triangle which has not been subdivided, will have the `DetailId` empty, which indicates it is a base triangle. A triangle which formed by subdividing an already existing triangle, will have the `BaseId` equal to the id of the triangle which it has subdivided – used to look up the `TriangleDetail` object in `mTriangleDetails`, and the `DetailId` will be used to look up the actual triangle, within this object.

4.1.4 Data Triangle

`DataTriangle` is the main way the geometry is stored in Pepr3D. It is a custom wrapper around *CGAL*'s `Triangle_3` class. Adding this wrapper allows us to attach additional information to each triangle. The information attached in our case is:

- **mColor** which is the color of the triangle. This is kept in a `size_t` variable, because it refers to the index of the color in the color palette.
- **mNormal** which is the normal vector of the triangle face. This is calculated as an average of the vertex normals of the triangle.

A second important class to talk about in this section is the `DataTriangle-AABBPrimitive`. This class provides the conversion between our `DataTriangle` (which contains a `CGAL::Triangle_3`) and the `CGAL::AABB_tree` which requires `CGAL::Triangle_3` as input to build the tree around. We provide the tree our `DataTriangle` array and the tree converts it into `DataTriangleAABBPrimitive`, which it accepts as a geometry primitive.

4.2 Model Importer

One of the classes included in this section is the *Model Importer*. As the name suggests, this class handles the import of a new triangle mesh model into Pepr3D. This class heavily utilizes the *Assimp* library we have mentioned several times in both the specification and documentation. Here is a quick overview of the class:

```
class ModelImporter {
    std::string mPath;
    std::vector<DataTriangle> mTriangles;

    ColorManager mPalette;
    bool mModelLoaded = false;

    std::vector<glm::vec3> mVertexBuffer;
    std::vector<std::array<size_t, 3>> mIndexBuffer;

    GeometryProgress *mProgress;

   public:
    ModelImporter(const std::string p, GeometryProgress
        *progress, ::ThreadPool &threadPool);
    bool isModelLoaded();

    ColorManager getColorManager() const;
    std::vector<DataTriangle> getTriangles() const;
    std::vector<glm::vec3> getVertexBuffer() const;
    std::vector<std::array<size_t, 3>> getIndexBuffer() const;
}
```

We will now go through the public API of the class, explaining in detail what each method does and why it is needed.

Firstly, `ModelImporter` gets initialized using its constructor. It gets the path to the file and our persistent threadpool, as well as a `GeometryProgress` object, which is used to report the import progress to the UI. This setup makes a single ModelImporter object responsible for a single imported mesh file, which follows the *Resource Acquisition Is Initialization* or *RAII* [2] principle, which is prevalent in the C++ scene. Once initialized, the `ModelImporter` can now return all the data it loaded.

A simple check of `ModelImporter`'s ready status can be performed using the method `isModelLoaded()`. This should primarily be used in combination with C++'s `assert` call.

Once the model is loaded correctly, the object now provides the data it has loaded. There are several methods to retrieve data.

- `getColorManager()` is a method which returns a `ColorManager` object, initialized with the colors of the imported file. This means that if a coloured geometry file gets loaded, the `ModelImporter` will automatically create a coloured model which is then displayed in Pepr3D. It will also initialize the color palette with the model's colors, not with the default ones.

- `getTriangles()` is the most important method of the class. This method returns an array of `DataTriangle` objects. This is the main data Pepr3D works on. These returned triangles are preprocessed using both *Assimp* preprocessing options and our own. Examples of preprocessing include triangulating all non-triangle primitives (like quads), removing duplicated vertices and removing degenerate triangles (with a zero area).

- The pair of methods `getVertexBuffer()` and `getIndexBuffer()` is a secondary means of extracting the same geometry data as in the `getTriangles` method, but this time in the "OpenGL" format of a vertex buffer and an index buffer. This data is used only during the construction of the polyhedron model for the *CGAL* library. Extracting the secondary data is easier and faster than transforming the `DataTriangle` array back into this buffer representation just for the polyhedron build.

Most of the private members are very self-explanatory, we will, however, briefly comment on the `mProgress` object. This object takes advantage of *Assimp*'s ability to report the percentage-wise progress during loading and preprocessing. We use this information in the loading dialog, to notify the user on the progress of the import. This makes it clear to the that the program is working as intended and has not crashed or stopped.

4.3 Model Exporter

The `ModelExporter` is an easy to use class which handles the export of the model. Once again, it is initialized once for every geometry object (a model) and can get called multiple times if we are saving the same geometry data (maybe with different parameters) multiple times. Once a new model gets loaded, a new `ModelExporter` needs to be initialized. Here is a simple overview of the class:

```cpp
class ModelExporter {
    const Geometry *mGeometry;
```

[2] https://en.cppreference.com/w/cpp/language/raii

```
    GeometryProgress *mProgress;
    std::vector<float> mExtrusioCoef;

   public:
    ModelExporter(const Geometry *geometry, GeometryProgress
        *progress);

    void setExtrusionCoef(std::vector<float> extrusionCoef);

    bool saveModel(const std::string filePath, const
        std::string fileName, const std::string fileType,
        ExportType exportType);

    std::map<colorIndex, std::unique_ptr<aiScene>>
        createScenes(ExportType exportType);
}
```

There are a few basic methods which we will cover first, as well as an extra degree of freedom for the users of this class, which requires a bit more knowledge, which we explain at the end.

- The *constructor* takes a pointer to the current `Geometry` object and a pointer to a progress indicator. Here we again exploit *Assimp's* progress reporting functionality to let the user know how far along the export is.
- `setExtrusionCoef()` is a simple *setter* which allows the user to specify the depth of extrusion for each segment.
- `saveModel()` is the main method the users will call if they wish to proceed with the export in the current setup. The user specifies the file path, file name and file type of the export, as well as which export should get used, as there are several options.

We mentioned several options of exporting, which are decided by the `Export-Type` enumerator. The whole definition of this `enum` is:

```
enum class ExportType { Surface, NonPolySurface,
    NonPolyExtrusion, PolyExtrusion, PolyExtrusionWithSDF };
```

The only options the end user has are exporting only in `Surface` mode or in `Extrusion` mode, with the ability to turn on *SDF limitation* when in the `Extrusion` mode. The two options prefaced with the keyword `Poly` are there for the developers, since these get invoked when the loaded model cannot be built into a polyhedron structure (which is displayed as a warning dialog, which is covered in greater detail in Chapter 7.

The last public method of the class is `createScenes()`. This is an advanced method which gives the developer access to the segmented scenes. This can be used, for example, to allow the user to preview the export results directly in Pepr3D (as it is done in the *Export Assistant*).

4.4 Font processing

Big portion of the code of this class is based on the *Font23D* library which can be found on GitHub [3]. The team read through the code, heavily modified it to transform it from the *C* language to the modern *C++* and built a new class around what were only free methods in *Font23D*. This object also uses the *FreeType*, *FTGL* and *Poly2Tri* libraries.

The main goal of the `FontRasterizer` class is to take smooth bezier curves of letters from the `.ttf` font files and transform them into a triangle mesh, with a variable rasterization steps. These meshes are then used in the **Text** tool.

A simple overview of the `FontRasterizer` class follows:

```
class FontRasterizer {
  private:
  struct Tri {
      glm::vec3 a, b, c;
   };

   std::string mFontFile;
   bool mFontLoaded = false;

   FT_Library mLibrary;
   FT_Face mFace;
   FT_UInt mPrevCharIndex = 0, mCurCharIndex = 0;
   FT_Pos mPrev_rsb_delta = 0;

  public:
   FontRasterizer(const std::string fontFile);
   std::string getCurrentFont() const;
   bool isValid() const;

   std::vector<std::vector<FontRasterizer::Tri>>
      rasterizeText(const std::string textString, const
      size_t fontHeight, const size_t bezierSteps);
}
```

As we can see, similar to the `ModelImporter` class, this class also uses the *RAII* principle, this time to load and hold the font face, loaded from a `.ttf` file. Once the font file is loaded successfully, the object can then convert any text into a triangle mesh.

The interface of this class is rather simple – the constructor takes a single `std::string`, which is the path to the font file. The API provides a `isValid()` method to check whether the initialization was performed correctly. The API also provides the font file name, without the whole path, which can be accessed by the method `getCurrentFont()`. This is used in the user interface to display the currently selected font.

The main method of the class is the `rasterizeText()`. It takes a `string` containing the text the user wants to convert to triangles, the `fontHeight` integer, which corresponds to the font height commonly found in text editors and

[3]https://github.com/codetiger/Font23D

the variable bezierSteps, which allows the user to control the roughness of the approximation. The useful range of the bezierSteps variable is around 1 – 5. 1 yields very rough results, useful for "blocky" fonts like *Impact*, while 3 should be sufficient for any standard font. For high precision, a higher setting should be used. Please note that the higher the bezierSteps, the more triangles will be generated.

Last thing to note, is the custom triangle object the rasterizeText method returns. Tri is a custom struct declared private in FontRasterizer. This is done to achieve two things:

1. FontRasterizer::Tri should not be used outside of the class. It is a temporary type and is useful only to the FontRasterizer class. The private definition prevents this behaviour. Users should not create more objects of this type at any time.

2. FontRasterizer::Tri is returned by rasterizeText. This is because we want the FontRasterizer to be an independent class, which can be used on its own and not depend on any Pepr3D types. Returning a custom type achieves this behaviour, and it is expected that the users will want to convert the outcome into a custom type more often than not anyway.

4.5 Color Manager

The ColorManager is a simple class which manages the current color palette. At most one ColorManager is active, which is the one MainApplication holds as a source of truth. ModelImporter also creates a new ColorManager while importing an already-coloured model.

Now we list a simplified overview of the class:

```
class ColorManager {
  public:
   using ColorMap = std::vector<glm::vec4>;

  private:
   ColorMap mColorMap;
   size_t mActiveColorIndex = 0;

   friend class cereal::access;

  public:
   ColorManager();
   ColorManager(const ColorMap::const_iterator start, const
      ColorMap::const_iterator end);
   explicit ColorManager(const size_t number);

   void addColor(const glm::vec4 newColor);
   void setColor(const size_t i, const glm::vec4 newColor);
   void replaceColors(const ColorMap::const_iterator start,
      const ColorMap::const_iterator end);
```

```
    void replaceColors(const ColorMap& newColors);
    glm::vec4 getColor(const size_t i) const;

    size_t getActiveColorIndex() const;
    void setActiveColorIndex(size_t index);

    static void generateColors(const size_t colorCount,
       std::vector<glm::vec4>& outNewColors);
}
```

As we can see, `ColorManager` is basically a simple wrapper around STL's `std::vector`, specialized on holding `glm::vec4` and extended with some color-generation features. It also is responsible for handling the current active color in the `mActiveColorIndex` member variable. This is the color the user has currently selected in his color palette widget.

There are several ways to initialize the `ColorManager`. You can initialize it to the default palette, with a list of colors or simply with a number of colors you require. In the last case, the `ColorManager` will generate new colors, which will be *visibly distinct* from each other (you will not get 3 slightly different shades of blue). The generation is done using the `generateColors()` method.

The `ColorManager` API has all the different calls you could expect from a `std::vector` wrapper, like `empty()`, `size()` and `clear()`, which we have omitted from the overview for clarity.

The most important part of the API are the `addColor()`, `setColor()` and `replaceColors()` methods. These allow for changing of the palette on the fly. Note that changing the palette changes the colors on the model in real time. `getColor()` can be used to query the `ColorManager` on any color (for example for user interface purposes), and when combined with the following pair of methods, it provides a vital part of Pepr3D by allowing the color palette widget to work.

We have already mentioned one of the two remaining methods – the getter `getActiveColorIndex()` and the setter `setActiveColorIndex()`. These methods are invoked when the user changes the active color in the color palette widget in the user interface. They are also invoked by various tools when the tool is constructing the recolour command and needs to know which colour the user painted with.

Chapter 5

Commands and Command Manager

In this chapter, we discuss the command system that provides the *Undo* and *Redo* capability of Pepr3D. First we explain the `Command` class in detail and then we show how the `CommandManager` operates to provide a fully functioning *Undo* and *Redo*.

5.1 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 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: colouring a single triangle (triangle painter tool), colouring multiple triangles (like in bucket painter) or changing the color palette.

A single command is a class derived from the `CommandBase` class, which has this structure:

```
template <typename Target>
class CommandBase {
    template <typename>
    friend class CommandManager;
  public:

    CommandBase(bool isSlow = false, bool canBeJoined = false);

    virtual std::string_view getDescription() const = 0;
    bool isSlowCommand() const;
    bool canBeJoined() const;

  protected:
    virtual void run(Target& target) const = 0;
    virtual bool joinCommand(const CommandBase&)

   private:
    const bool mIsSlow;
    const bool mCanBeJoined;
};
```

As we can see, there are only a few methods to implement per each command. Namely `run()` and `joinCommand()`. The only properties of the command are `mIsSlow` and `mCanBeJoined`, which are (through their const getters) accessed by the `CommandManager`.

The `mIsSlow` property notifies the `CommandManager` about a slow command, which means a snapshot should be made to allow for quicker undoing and redoing (we will refer to both of these options as only ”undo” from now on).

The `mCanBeJoined` property allows the `CommandManager` to join two commands of the same type together.

Now that we know how commands look like and work, we can look at the `CommandManager` class and see how the entire feature is implemented in Pepr3D.

5.2 Command Manager

The centrepiece of the `CommandManager` is the **command stack**. The Command Stack is a *LIFO* type structure, with the main purpose to store the executed commands, which allows the `CommandManager` to perform the undo operations. This data structure is then operated by the `CommandManager`. Following is the overview of the `CommandManager` class.

```
template <typename Target>
class CommandManager {
   public:
    using CommandBaseType = CommandBase<Target>;
    using StateType = decltype(std::declval<const
        Target>().saveState());
    static const int SNAPSHOT_FREQUENCY = 10;

    explicit CommandManager(Target& target);

    void execute(std::unique_ptr<CommandBaseType>&& command,
        bool join = false);

    void undo();
    void redo();

    bool canUndo() const;
    bool canRedo() const;
    const CommandBaseType& getLastCommand() const;
    const CommandBaseType& getNextCommand() const;
    size_t getVersionNumber() const;
```

```
  private:
   Target& mTarget;
   std::vector<std::unique_ptr<CommandBaseType>>
      mCommandHistory;
   size_t mPosFromEnd = 0;
   size_t mVersion = 0;

   struct SnapshotPair {
      StateType state;
      size_t nextCommandIdx;
   };
   std::vector<SnapshotPair> mTargetSnapshots;
};
```

As we can see, the class is not complex. It operates over a templated class `Target`, which only has to be able to be able to load its state and save its state using the methods `loadState` and `saveState`. The state is what gets undone during the undo operation.

The three main methods to operate the undo pipeline are `execute()`, `undo()` and `redo()`, which are self-explanatory. The user interface can also use the methods `canUndo()` and `canRedo()` to notify the user with visual cues (like emboldening the undo signs) if the undo and redo actions are available.

The `CommandManager` also keeps its current cumulative version, which gets incremented during each `execute()`, `undo()` and `redo()`. This is a way to keep track of the user's actions and notify him if the current project was modified after it was saved.

The main data is stored in the *command stack*, which have already discussed. The command stack is the member `mCommandHistory`, which holds all past commands in a `std::vector`, which is a sufficient structure to implement a *LIFO* in. It holds pointers to the commands, instead of commands themselves to be able to use **polymorphism** in C++. The pointers are realised with C++'s `unique_ptr` for maximum memory safety.

After consideration, we chose the snapshotting technique we discussed in the *Specification document*. Snapshotting happens inside the `execute()` method and is regulated by one member variable inside the `CommandManager` – the `SNAPSHOT_FREQUENCY`. `mTargetSnapshots` is the array which holds the already taken snapshots. This `std::vector` gets manipulated during `execute()` and `redo()` to correctly perform the undo and redo functions. Snapshots are saved as a `SnapshotPair`, which remembers the current state and the command id which it was taken at.

Let's now illustrate the redo realised by snapshotting on a specific example. Let there be C commands on the command stack, with last snapshot taken at the S-th command, where $S < C - 1$. If we now undo, which means point the current Target state at $C - 1$, the last `SnapshotPair` gets inspected, and we find that the command id of the last snapshot was S. The state gets re-instantiated to the state we found in the `SnapshotPair`. Now we need to re-apply all commands from $S + 1$ to $C - 1$ to be correctly back in the $C - 1$ state.

Chapter 6

Tools

In this chapter, we explain the concepts behind each tool and the Geometry API the tools need.

A tool is the main programmable 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). Our design should allow for later advancements of the software – 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.

Here we take a look at the `Tool` class overview:

```
class Tool {
  public:
   Tool() = default;
   virtual ~Tool() = default;

   // UI properties
   virtual std::string getName() const = 0;
   virtual std::string getDescription() const;
   virtual std::optional<Hotkey> getHotkey(const Hotkeys&
       hotkeys) const;
   virtual std::string getIcon() const = 0;
   virtual bool isEnabled();

   // Action methods
   virtual void drawToSidePane(SidePane& sidePane){};
   virtual void drawToModelView(ModelView& modelView){};

  // Event methods (listed in the Table below)
  ...

   virtual std::optional<std::size_t>
       safeIntersectMesh(MainApplication& mainApplication,
       const ci::Ray ray) final;

   virtual std::optional<DetailedTriangleId>
       safeIntersectDetailedMesh(MainApplication&
      mainApplication, const ci::Ray ray) final;
};
```

As we can see, it is a simple abstract interface class. There are some *UI properties* like `getName` and `getDescription`, as well as some rendering methods like `drawToSidePane`. It contains a few `final` methods, which are just a exception-safe methods to call on the `Geometry` class, implemented in the interface, because every tool needs them. The also are multiple *Event methods*, which we have omitted from the code here and will examine further in the following table.

Name	**Invoked if ...**
`onModelViewMouseDown`	mouse gets pressed down in the model view
`onModelViewMouseDrag`	mouse gets dragged in the model view
`onModelViewMouseUp`	mouse button gets let go of in the model view
`onModelViewMouseWheel`	mouse wheel is scrolled in the model view
`onModelViewMouseMove`	mouse is moved in the model view
`onToolSelect`	this tool is selected
`onToolDeselect`	this tool is deselected
`onNewGeometryLoaded`	new model is loaded (imported or opened)

Table 6.1: The overview of different events available to any `Tool` class.

When implementing a new tool, not all of these events have to be specified. The tool only listening and can act accordingly if it needs to.

The tool's **user interface** is implemented in the `drawToSidePane` method. During the rendering, if the tool is active, this method gets called to fill the SidePane with ImGui widgets, which allow the user to alter the tool's properties.

We will not discuss each and every tool we have implemented, we will, however, mention a few things that most of the tools have in common.

6.1 Specific tool details

Most, if not all of the tools we implemented, need access to the `Geometry` and `CommandManager` classes. Since the `MainApplication` object holds both of these, the tool has to get a reference to the parent `MainApplication` object. This is why most of our tools have this piece of code in common:

```
public:
  explicit PaintBucket(MainApplication& app) :
     mApplication(app) {}
private:
  MainApplication& mApplication;
```

The tool then accesses the `CommandManager` through `mApplication`'s method `getCommandManager()` and the current geometry data through the `getCurrentGeometry()` method.

This way of communication is required as the `MainApplication` is the source of truth of what geometry is currently loaded and what command manager corresponds to it.

Certain tools, such as the Segmentation and Export Assistant, also override Model View OpenGL buffers to display colors that are not in the palette or modified geometry such as the extrusion preview.

Chapter 7

User interface

A *user interface* (UI) is the front-facing part of Pepr3D that the users interact with. It is responsible for managing windows, showing buttons, rendering the 3D model, handling mouse clicks, hotkeys, showing error dialogs, and much more. In case of Pepr3D, it should be an easy-to-use, intuitive, and fast abstraction of the complex 3D geometric algorithms at the backend, see Figure 7.1.

We divide the Pepr3D UI into the following main parts:

- the **main application** corresponds to the whole main window, which consists of the following:
- a **toolbar** with toggleable buttons representing tools, undo/redo, etc.,
- a **side pane** with buttons, checkboxes, sliders, etc., representing configuration of the currently selected tool,
- and a **model view** with the 3D model which the user can rotate, zoom, paint on it, etc.

7.1 Key concepts

In the specification of Pepr3D, we explained several main ideas that the user interface of Pepr3D is built around. We will not repeat every single idea from the specification here as that would be redundant, but we explain the key concepts that we kept in mind while developing the user interface.

To prevent reinventing the wheel, our concepts are based on investigating how other developers implement user interfaces. Our UI mainly consists of *real-time 3D rendering*, i.e., displaying the 3D model that users can interact with, and *widgets*, i.e., the windows, buttons, check boxes, text labels, etc.

7.1.1 Real-time 3D 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 is primarily based on real-time rendering.

As explained in our specification, when we were looking for a 3D rendering library, we mainly focused on finding an easy-to-use abstraction, e.g., for rendering 3D primitives, using custom shaders with uniforms, uploading textures to the

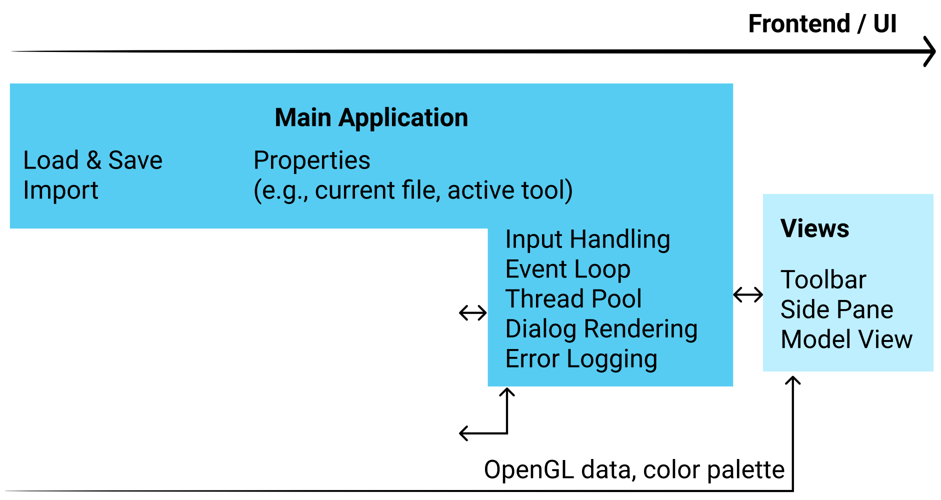


Figure 7.1: An overview of the Pepr3D UI architecture, based on Figure 3.1.

GPU, keeping constant framerate, etc. We decided to use *Cinder*, which is also cross-platform and supports asynchronous events.

7.1.2 Widgets

As Pepr3D is based on real-time rendering, we decided to use *immediate UI*, i.e., a procedural UI redrawn every frame that is mostly stateless. This means that we do not have to program any explicit synchronization between the UI and the backend. Whenever we re-render the UI, it will be rendered with the newest data. Retained state is only where necessary, e.g., for complex calculations not needed in every frame. Our immediate UI is a part of the 3D renderer based on Cinder and OpenGL, where the widgets are rendered on top of the scene.

We also kept in mind that *presentation should be separated from the logic*, i.e., the rest of the application should know nothing about the UI at all. Hence our UI depends on the backend, but the backend does not depend on the UI at all. In theory, we are able to easily replace the UI with another, should it be necessary.

For these reasons and others explained in our specification, we based our widgets on a library called *Dear ImGui*. It provided us with basic handling of mouse clicks, keyboard inputs, etc., and also with 2D drawing such as texts, icons, buttons, etc. But the design of Pepr3D is completely ours and is based on heavily customized widgets based on ImGui primitives.

7.2 Application

The whole application and its main window are executed via the Cinder library. The main file `main.cpp` contains a Cinder macro which corresponds to cross-platform `main` functions. Cinder is responsible for creating an instance of our `MainApplication` class on which it calls the `setup` method.

After the main window is set up, Cinder keeps calling the `update` and `draw` methods every frame in this order. Cinder also handles all necessary events from an operating system and calls the appropriate methods of the `MainApplication`.

7.2.1 Main application

The `MainApplication` class is a router which interacts with other components, mainly the three major UI components: a **toolbar**, a **side pane**, and a **model view**. It instantiates them, listens for events from the operating system, and passes them to the components so that they can handle them themselves.

This also includes delegating hotkeys, open, save, and import commands to their dedicated classes, handling minimized application, etc. We can say that every user interaction gets first through the `MainApplication` which then delegates it to other components. The `MainApplication` also works as a single source of truth that contains the current geometry and current tools. Other components use `MainApplication` as a getter for the current valid state.

Because of the Cinder API, setup is not done directly in the constructor, which only instantiates other components, but in the `setup` method which Cinder calls once it is ready. It creates and configures debug and error loggers, sets the main window properties such as its resolution and icon, initializes ImGui, hotkeys, the default geometry (cube), and constructs all tools.

The `draw` method draws the major UI components and also renders existing modal dialogs and a progress indicator when necessary. The `update` method checks invariants such as that the selected tool is not disabled, application is not rendered if minimized, and unsaved changes in a project are marked with an asterisk * in the title.

7.2.2 Multi-threading and background tasks

Some operations such as importing or exporting a large model may take a long time. If everything was done in the main thread, rendering of the UI would be paused and the window would become unresponsive. In UI applications, we want to avoid this by running slow operations in the background.

For this purpose, we use a very simple thread-pool library consisting of just a single `ThreadPool` class. A single instance of this thread pool is stored as a static member of the `MainApplication`.

Delegating a task into the thread pool is done by calling `enqueue` on the thread pool. This method returns a C++ `future` which we can wait for. However, the UI in the main thread should *not* wait for the tasks to finish in a blocking way, as that would defeat the purpose.

Instead, we can call `dispatchAsync` on the `MainApplication`, which dispatches another callback via Cinder. This callback will be executed before a new frame starts rendering. This enables the background tasks to notify the UI that they have finished.

In summary, the pipeline works like this: the UI enqueues a slow function F (typically a lambda function) to the thread pool and continues rendering. Right before F finishes, the F itself calls `dispatchAsync` with another function A, which will be run on the UI thread at a new frame. A is used to update the UI to the new state, e.g., after loading a new model.

We made a simplified API for exactly this purpose that Tools can use when enqueuing a slow operation such as SDF computation. The `MainApplication` has a templated method `enqueueSlowOperation` to which we can pass the

F and *A* functions. Thread pool and dispatching will be used automatically and a progress indicator will be shown.

7.2.3 Preventing race conditions in UI

Unfortunately, certain operations (e.g., with CGAL) are not thread-safe for the UI even though the UI is read-only when a progress indicator is shown. To solve this problem with possible race conditions, we actually render the whole user interface in a frame buffer (off-screen rendering).

If there is no asynchronous progress going on, we simply show this frame buffer on screen and it looks like we are rendering directly to the screen. If, however, a progress indicator is active, we pause all UI rendering and processing, we show the cached frame buffer on screen and overlay it with the progress indicator. So the indicator is the only part of the UI which is actively rendered, which is thread-safe as the progress relies on `std::atomic` values.

7.2.4 Resources of a minimized or obscured window

Because Pepr3D runs at a vertically synchronized framerate (typically 60 frames per second), it uses CPU and GPU resources even when minimized or obscured. We had to implement a way to pause rendering while the window is not visible. Cinder provided a way to tell if the window was minimized, but it did not solve situations where a window was obscured but not technically minimized.

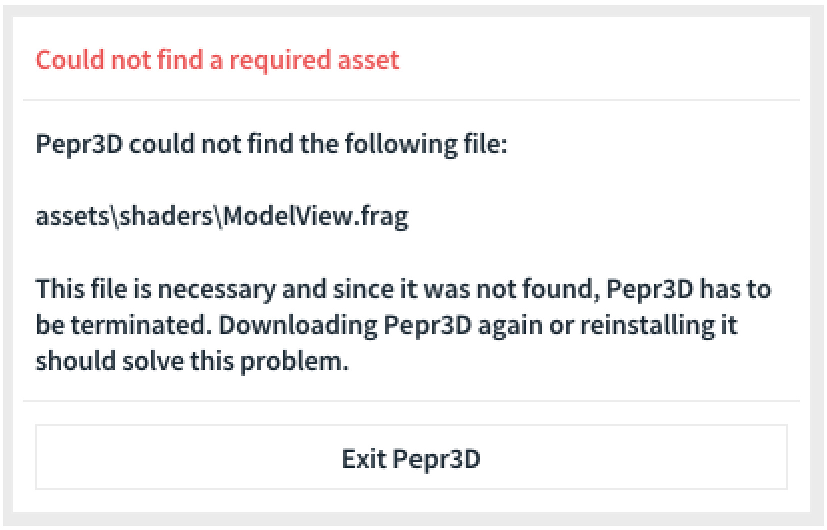
This proved to be a difficult problem that we managed to partially solve for Microsoft Windows by using the Windows API. In regular intervals, we check whether the top left corner, center, and bottom right corner are obscured by another window of another application. When they are, we pause the rendering. This significantly lowers the CPU and GPU usage when Pepr3D is not used.

7.2.5 Dialogs and fatal error handling

In Pepr3D, there are 2 types of dialogs: general modal dialogs and a progress indicator. These dialogs are drawn from the `MainApplication` on top of everything else. While they are shown, all mouse and keyboard input for the rest of the components is interrupted, meaning users cannot interact with any tools.

General modal dialogs and fatal errors General modal dialogs are represented by the `Dialog` class and they can be of various importance, e.g., information dialogs, fatal error dialogs, etc. The dialogs are stored in a priority queue with the most important dialog on top. They can be pushed to the queue by calling `pushDialog` on the `MainApplication` instance.

When we detect an invalid state of the application, e.g., by catching a fatal exception, a *fatal error dialog* (see Figure 7.2) is pushed to the queue. When a fatal error dialog is on the top of the queue, rendering of all components is stopped and only the dialog is shown, because it cannot be guaranteed that other components are in a valid state. Pressing a button on the fatal error dialog terminates the application.


Could not find a required asset

Pepr3D could not find the following file:

assets\shaders\ModelView.frag

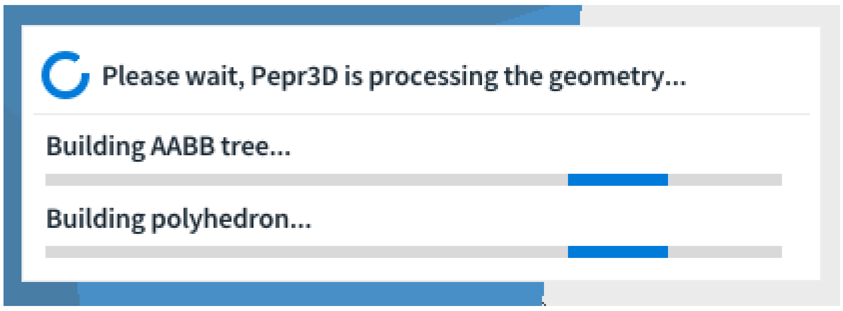
This file is necessary and since it was not found, Pepr3D has to be terminated. Downloading Pepr3D again or reinstalling it should solve this problem.

Exit Pepr3D

Figure 7.2: Example of a fatal error dialog.

Sometimes an error is so fatal that even the fatal error dialog cannot be rendered and Pepr3D is terminated immediately. Nevertheless, all warnings, errors, and fatal errors are logged in `pepr3d.log` files which are backed up in case of a fatal error. When Pepr3D is executed again, an information dialog is shown explaining the user where they can find a related log file with error details.

Progress indicator A progress indicator is an animated dialog (see Figure 7.3) which shows the elapsed and remaining progress required to process the current command. It is used mostly when opening, saving, importing, and exporting geometry files, or computing SDF, because these are slow operations. Its logic is handled in the `ProgressIndicator` class which has a pointer to the `Geometry` which is being loaded. The current status is checked directly from the geometry data every frame.


Please wait, Pepr3D is processing the geometry...

Building AABB tree...

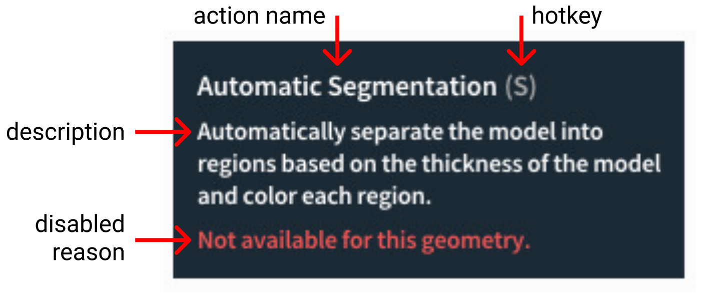
Building polyhedron...

Figure 7.3: Example of a progress indicator during import.

7.2.6 Tooltips

As the user interface has to be as clean as possible to allow fast and easy navigation even for beginners, long and detailed explanations of buttons and input boxes are "hidden" in tooltips. Tooltips (Figure 7.4) are dark informative rectangles that display when user hovers over an interactive widget, e.g., a button.

Tooltips in Pepr3D can show a name of an action, its hotkey (if available), long description of an action (if provided), and an explanation why an action is disabled (if it is disabled). Tooltips are created by calling `drawTooltipOnHover` on the `MainApplication` instance. They are only drawn when the item is actually hovered.



Automatic Segmentation (S)
Automatically separate the model into regions based on the thickness of the model and color each region.
Not available for this geometry.

Figure 7.4: Example of a tooltip.

7.2.7 Hotkeys

Hotkeys (keyboard shortcuts) enable users to perform common actions by pressing a single key or a combination of keys. Using hotkeys, changing active tools and colors is much faster and so is the whole editing process. There is no need to move a mouse cursor over the whole window just to change a color and then move back over the edited geometry.

Hotkeys are managed by the `Hotkeys` class which contains a mapping between `Hotkey` and `HotkeyAction`. There are two maps, one for each direction, i.e., a hotkey to action, and an action to a hotkey. The former one is used for faster event handling, the later one for faster displaying of tooltips that show what keys to press.

A `Hotkeys` instance is managed by the `MainApplication` which loads user specified hotkeys from a file in `assets/hotkeys.json`. If this file does not exist, default hotkeys are loaded. When a key is pressed, `MainApplication` calls `findAction` on the `Hotkeys` instance and then performs the corresponding action. Similarly, `Toolbar` and `SidePane` call `findHotkey` on the `Hotkeys` instance to find which key should be shown in a tooltip. The `getString` method on a `Hotkey`, unfortunately, only supports letters and numbers and a `Ctrl+` modifier. There is no mapping between key codes and UTF-8 symbols.

Hotkeys are customizable by editing the file `assets/hotkeys.json`, however, they are not customizable from the user interface. The file has a simple JSON structure as can be seen in the example:

```
{
   "hotkeys": {
      "keysToActions": [
         {
            "key": {
               "ctrl": false,
               "keycode": 100
            },
            "value": "SelectDisplayOptions"
        },
        {
            "key": {
                "ctrl": false,
                "keycode": 101
            },
            "value": "SelectTextEditor"
        },
        ...
    ]
  }
}
```

where the key codes are based on Cinder and can be found in the user documentation.

7.3 Toolbar and side pane

The toolbar and side pane are two major UI components that users interact with. They are both rendered on top of the geometry, covering the top and right part of the window.

7.3.1 Toolbar

The toolbar is represented by the `Toolbar` class. It is rendered on the top of the window and meets with a side pane at the right side. It consists of 3 main parts: the file drop down on the left, the undo and redo buttons, and the tool buttons that select active tools.

All the buttons in the toolbar are rendered via the templated `drawButton` method. It is a heavily modified ImGui button, because our toolbar buttons also support multiple states (inactive, hovered, held, active, disabled) and a dropdown option which we use for the file drop down.

The tool buttons are not hard-coded, but rendered by dynamically iterating over all `Tool` instances that are part of the `MainApplication`. For example, when we compile in the debug mode, there is an additional debug tool, which is not present in the release version.

7.3.2 Side pane

The side pane is represented by the `SidePane` class. It is rendered on the right side of the window and fills up the whole height of the screen. It consists of 2 main parts: the header which shows the currently active tool, and the "inside" where properties of the active tool are shown.

There is an important concept: *the tools themselves draw the inside of the side pane*. The side pane only calls the `drawToSidePane` method on the currently active `Tool` and the *tool itself* decides what is drawn by calling `SidePane` helper methods such as `drawText`.

So it is the side pane which knows how to draw texts, buttons, color palette, separators, and other UI widgets in the side pane, but what exactly gets drawn is decided by the tools. This is a design decision that makes the `SidePane` independent on the specific tools and enables them to provide various properties that the users can edit in real-time.

Standard widgets Side pane can contain standard widgets. These are for example texts drawn by `drawText`, buttons and coloured buttons by `drawButton` and `drawColoredButton`, separators by `drawSeparator`, checkboxes by `drawCheckbox`, and more. The tools can, however, also use ImGui directly, because all ImGui calls from inside their `drawToSidePane` method get automatically associated with the side pane.

Colour palette The most complex widget available in the side pane is the color palette. This is an advanced and completely custom widget built using basic ImGui components.

It has two modes: the "read-only" color palette only shows color boxes that can be clicked and selected, the "editable" mode allows the user to completely customize the color palette of Pepr3D. The former is shown in most tools such as triangle painter or brush, the later is used in Pepr3D settings.

The color palette is synchronized with the `ColorManager` directly. The editable mode consists of 4 parts: the header, the "add" button and "delete" box, the color boxes (also present in the read-only mode), and a "reset" button.

The drag-and-dropping feature is using the ImGui experimental API built for these purposes. The color boxes hold a payload with their IDs and when they are dropped on a different color box, the colors get swapped or reordered. The difference between swapping and reordering is that the former also swaps the colors in the geometry, while the later only reorders them in the palette. When a color box is dropped on the "delete" box, the color is removed entirely from the geometry and replaced by the first color in the palette.

7.4 Model view

The model view is the 3D part of the UI, represented by the `ModelView` class. It is responsible for rendering the geometry and allowing users to interact with the active tool by clicking and dragging over the geometry. It also handles the camera, i.e., moving around the model, and shows an optional grid representing the printing bed and a wireframe consisting of the triangles of the model.

7.4.1 Model matrix scaling, translations, and rotations

Before even drawing the model, we must first handle its dimensions, position, and rotation. This is because different models, especially models imported from different 3D editors, have various scales and origin points.

We made an `updateModelMatrix` method responsible for scaling, translating, and rotating the geometry with regards to the following rules. The model's largest dimension in the XYZ axes must be 1.0, so that the whole model is visible on the screen without the need to zoom in or out. This is done by computing the axis-aligned bounding box (AABB) of the model and using its dimensions.

The model is also translated so that it is centered over the $(0, 0, 0)^T$ point But the lowest edge of the model must touch the grid at height $y = 0$, so we need to translate in the y axis again, moving the model up a little bit.

Finally, we rotate the model so that the z axis points up, because this is the standard in 3D printing pipelines and corresponds to what the slicer uses. This is different than in OpenGL where the y axis is usually considered to be the up axis. This rotation also means that what users can see in Pepr3D corresponds to what they can see in Blender or Slicer for Prusa printers.

7.4.2 Drawing geometry

Drawing the current `Geometry` instance is the main responsibility of the model view. The `draw` method consists of several steps that we now describe.

First it sets up the OpenGL viewport to only render to the model view part of the window, i.e., to ignore the toolbar and side pane parts. Then we push the camera matrices to OpenGL, which sets up the position and direction from which we look at the model.

Then the `updateModelMatrix` method is called and the model matrix is updated. The scaling, translations, and rotations only happen during the rendering, so the original geometry is not affected at all.

Then we call the `drawGeometry` method, which uploads all necessary data to the GPU via OpenGL. We push the model matrix, update vertex buffer objects, update shader uniforms, and finally draw the batch using OpenGL vertex and fragment shaders. Note that the vertex and index buffers may be very large and may update every frame. For these reasons, we do two optimizations.

First, the buffers are only uploaded when a change is detected. This is handled in the `Geometry::OpenGlData` class which also stores the data on the CPU side. The second optimization is that we do not upload the colors of the model directly to the vertices, instead every vertex has an index to the palette. The color palette itself is uploaded to the GPU as a uniform array and then it is used in the fragment shader.

After drawing the geometry, we also draw the grid that simulates the printing bed of the printer. One cell of the grid measures 0.1×0.1, the whole grid is 1.8×1.8, so that it is always slightly bigger than the model itself.

7.4.3 Overriding buffers

Sometimes it is necessary to draw custom colors that are not in the color palette or to draw objects that are not the current geometry. For example the Segmentation tool needs to provide custom colors so that users can see the different regions. The Export Assistant needs to provide completely custom buffers including vertices, indices, normals, and colors to preview the extruded geometry objects.

For these purposes, `ModelView` provides buffer overrides. There are two of them: `ColorOverrideData` and `VertexNormalIndexOverrideData`. Public setters and getters are provided for overriding them.

7.4.4 Shaders

In order to display the geometry, we need to provide the GPU with two OpenGL shaders written in GLSL. The vertex shader is called on every vertex of a triangle of the model. The fragment shader is then called on every single fragment (pixel) of the displayed model. This is a standard OpenGL pipeline.

In the vertex shader located in `assets/shaders/ModelView.vert`, we primarily just forward vertex attributes to the fragment shader. Additionally, we also generate barycentric coordinates of the vertex. This uses the fact that OpenGL interpolates attributes over the triangles, so if we set the barycentric coordinates in the vertices, they are automatically correctly interpolated for the fragment shader.

In the fragment shader located in `assets/shaders/ModelView.frag`, we need to calculate the final color of every fragment. There are several steps that contribute to the color. Primarily, we use Lambert shading to display the model, where we assume the light source always points from the camera. The main color of the model is found in the color palette with regards to the ID attribute of the vertex.

In case the Brush tool is active, we also calculate whether the current fragment is inside the highlighted region of the brush. And finally if the wireframe rendering is enabled, we use the interpolated barycentric coordinates from the vertex shader to find out whether we are on an edge of a triangle, and if so, we highlight it in a contrast color.

In case of the Export Assistant, we additionally provide a way of discarding pixels that are not in the specified height range. This is done using the `discard;` statement of OpenGL shaders.

7.4.5 Camera

While painting the model, users of Pepr3D need to rotate and zoom the model in order to paint details from all sides of the geometry. Typically, in all major 3D editors, a so called arc-ball camera is used. The camera moves around a pivot point (the model in our case) by dragging the mouse on the screen. Zooming is usually performed with the mouse wheel. Other actions may also be performed and are explained in the user documentation.

We originally used a camera implementation from Cinder called `CameraUi`. Unfortunately, during our testing, we found out that their implementation of zooming is not perfect for all our purposes. We modified the original `CameraUi` so that it supports two types of zooming.

Now, *zoom* in our case means changing the field of view of the camera. *Dolly*, on the other hand, means moving the camera position towards or further away from the pivot point (the model). We also fixed a bug in which dollying too close would move the pivot point and cause unexpected behaviour.

Chapter 8

Testing

In this chapter we describe our testing pipeline. We have several ways how to test if the program behaves correctly. Namely:

- **Unit tests** – the basic testing of several components of Pepr3D. The unit tests are small use-cases crafted to test each functionality of the object individually. The tests are great for catching quick and stateless errors but do not provide any information about more complex operations.
- **Manual tests** – because of the simplicity of unit tests, we have several written manual tests for each tool in Pepr3D. These are executed manually by the person doing the predefined operations, and checking the result against the expected result.
- **Continuous integration** – our Git repository is equipped with Continuous Integration software. This ensures that every pull request and merge is compilable, which ensures every commit in the *master* branch is compilable and runs the unit tests automatically. The merge will not be executed if either of these conditions fail.

In the following sections, we explain these types of tests in detail, as well as provide the descriptions of the manual tests.

8.1 Unit tests

Unit testing is probably the most common way to automatically test software. As such, we will not explain in detail the benefits of this procedure. We use Google Test library [1] as it is one of the best C++ testing frameworks we have found. Several of our team members also already had experience with Google Test.

For better navigation in the code base, we decided to follow the common naming convention: for class `CommandManager`, we have `CommandManager.h` and `CommandManager.cpp` as the implementation files. Now to test this class, we add `CommandManager.test.cpp` file and program all `CommandManager` into this file. This makes searching for tests very easy.

[1]https://github.com/google/googletest

8.1.1 Library test

The test `libraries.test.cpp` is a special case among our unit tests. As the name suggests, this test checks whether our 3rd party libraries are setup correctly. This test should always succeed if it is compiled. If it does not get compiled or linked, the libraries were set up incorrectly.

8.1.2 Class tests

All of the other tests are the ”standard” type of tests – testing the public methods of the class. We will not describe each of the tests individually, since each test has a documentation comment inside describing what the test does, as well as a fitting name.

The general structure of the unit tests is cumulative – this means that if the first test fails, there is a high chance all of the the following tests will fail too. The advantage of this approach is clear once you imagine a different sorting of the tests. If the first test was a complicated behaviour of the class, the test will fail not only if the behaviour is incorrect, but also if the initialization of the class is wrong. This is bad, because the programmer fixing the test will not immediately know which part of the class is incorrect.

8.2 Manual tests

In this section we describe the reasoning behind manual testing. We also list all of the manual tests the team has accumulated during the development.

Manual tests are based on scenarios, i.e., hypothetical stories simulating what users would typically do when using the application. A person performing a manual test is required to proceed exactly as written in the scenario and notice any wrong or unexpected behavior.

There are number of reasons why we use manual testing along with unit testing and continuous integration. One of the most important reasons is that writing fully automated tests for applications with user interfaces is a very slow and demanding process. Often, there are dedicated employees (sometimes called Software Development Engineers in Test, or SDET) writing the software for testing another software. Our team was simply not large enough to handle that.

Another reason is that human testers can quickly identify weird behavior, glitches, or broken look of the user interface. Using automated tests for verifying that the design of a user interface is not broken would require updating the tests every time we add a new button. Automated tests also only do what they were scripted to do, so they do not “explore” the application outside of their boundaries. Humans, on the other hand, do not have to be given precise instructions in every single step and they naturally explore other areas as well.

8.2.1 Basic first-run tests

These scenarios are used for verifying basic common situations that every user of Pepr3D would come across. They basically check that the application as a whole behaves in an expected way. If these tests fail, any user could immediately notice.

Layout and camera control

1. Run Pepr3D. (Note: all scenarios expect the Release build to be run.)
2. Verify that a new window is open, toolbar on the top, side pane on the right, model view in the rest of the window. The title of the main window should say “Untitled* - Pepr3D”. No console window should open.
3. There should be 13 icons in the toolbar and all of them should be active (black) except Undo and Redo that are disabled. Icons are visually divided into 5 parts: File; Undo and Redo; drawing tools; Display Options, Settings, and Information; and Export Assistant.
4. The default active tool is Triangle Painter, the icon is blue.
5. Model view should show a blue cube placed precisely in the middle of a grid, touching the grid at the bottom.
6. Right-clicking and dragging the mouse on the model view should rotate the cube (left-right, top-down). Rotating the cube over 90° at the top or bottom should not be possible.
7. Middle-clicking and dragging the mouse should move the cube *and* the grid without rotating. The same for holding Control and right-clicking and dragging.
8. Scroll wheel should zoom the cube, scrolling up should zoom in, down should zoom out. Zooming too close *may* go inside the cube, but it should be possible to zoom out again and rotate the cube the same as before.
9. In the toolbar, left-click File (the left-most icon). A popup menu should display with 6 options. Press Exit. The application should exit successfully.

Loading, saving, and importing basic files

1. Run Pepr3D, drag-and-drop a valid `.stl` model anywhere onto the main window. Verify that the default cube has been completely replaced by the new model. The new object should be centered on the grid and resized perfectly so it fits on the screen. The title of the main window should say “file_name - Pepr3D”, where the file name should be *without* an extension, *without* a full path, and *without* an asterisk. Repeat with valid `.ply` and `.obj` models.
2. Verify the same files can also be successfully imported via File → Import. Verify that a progress indicator is shown when importing bigger models and that the user cannot interact with the application while the progress indicator is being shown.
3. Verify that when a model that has a color palette defined is imported, the color palette in Pepr3D also imports up to 16 colors from the model.
4. Do a change in the model, e.g., color a single triangle with a different color. Verify an asterisk is now shown in the title denoting an unsaved change.

5. Select File → Save and verify it opens a file dialog with the original file name and path as default. Save the file. Verify the asterisk in the title has disappeared. Do a change in the model. Verify the asterisk has appread again, select File → Save, now the file should be saved to the same file without opening a dialog, and the asterisk should disappear.

6. Select File → Save as, verify a file dialog is now opened and save the file with another name.

7. Exit Pepr3D and run it again. The default cube should be shown. Drag-and-drop the previously saved `.p3d` file and verify it corresponds to the model you saved. Verify the same via File → Open. The overall behavior should be identical to importing models.

8. Verify that when changes are made and a new model is imported or open, Undo and Redo buttons are reset to their disabled state.

Toolbar tooltips and hotkeys

1. Verify `assets/hotkeys.json` file exists and run Pepr3D.

2. Hover mouse above icons in the toolbar and verify tooltips are shown with correct names, descriptions, and hotkeys.

3. Verify every single hotkey by hovering on the icon, checking which hotkey should be pressed, and pressing that hotkey. By default, every icon in the toolbar should have a hotkey assigned. Also check that hotkeys work for the color palette.

4. Exit Pepr3D, remove `assets/hotkeys.json`, and run Pepr3D again. There should be *no* error, Pepr3D should use the default hotkeys.

8.2.2 Tools tests

These scenarios verify that all tools behave as expected, except the Export Assistant, which is checked separately. For all tools, also verify that the side pane shows the current tool options and that tooltips are shown for all options in the side pane. With tools that interact with the models by clicking, hovering mouse over triangles should highlight them, except Brush.

Triangle painter

1. Run Pepr3D, verify a color palette is shown in the side pane. Select different colors in the palette and left-click on the triangles on the cube. They should be colored with the selected colors. Pressing Undo should revert every single click individually.

2. Now left-click and drag the mouse over multiple triangles. They should all be colored in real time. Pressing Undo should revert the whole click-and-drag operation at once.

Paint Bucket

1. Run Pepr3D, select Paint Bucket. Verify the side pane is populated with a palette and options. All options should have a tooltip.
2. Default options should be “Paint while dragging”, “Color based on criteria”, and “Stop on different color”. With these options, select a different color, left-click a triangle, and the whole cube should be colored.
3. Select “Stop on sharp edges”, new options should appear, the default options should be 30° and “Neighbouring triangles”. Select a different color, left-click a triangle. Only the two triangles on a face of the cube should change color. Now click-and-drag over multiple triangles and the changes should be real-time. Unselect “Paint while dragging” and verify that click-and-drag only applies to the first triangle held. Undo should behave as in Triangle Painter.
4. Set angle to 95°. Clicking and coloring should now be applied to the whole cube but still stop at a boundary with a different color.
5. Unselect “Stop on different color”, clicking and coloring should now be applied to the whole cube.
6. Select “With starting angle”. Clicking and coloring should now be applied to the whole cube *except* the one face opposite the one being clicked on.
7. Select “Color whole model”. All options below “Color based on criteria” should disappear. With this option, clicking and coloring should now be applied to the whole cube.

Brush

1. Run Pepr3D, select Brush. Verify the side pane is populated with a palette and options. All options should have a tooltip.
2. Default options should be “Size” number 2, 12 “Segments”, and “Sphere” brush shape. Everything else should be unchecked.
3. Select a different color and verify that hovering mouse over a model highlights a sphere (circle) on the model.
4. Clicking should paint the highlighted shape on the model, but it may not be perfectly circular w.r.t. the number of segments. Try changing this number and verify it also changes the number of segments that are painted.
5. Verify that “Size” changes both the highlight and actual paintings. The maximum size (20) should roughly cover a big part of the model.
6. Check “Respect original triangles”. Verify that a new “Paint outer ring” option has appeared (unckeched). Increase the size of the brush and verify that whole triangles are indeed painted. By checking “Paint outer ring”, triangles should be painted also when the brush is smaller than the triangles.

7. Change shape to “Flat”. Settings below should change. By rotating the model (cube) sideways, verify that “Perspective” and “Normal” settings do change the behavior of painting w.r.t. the camera angle or normals.
8. Verify “Paint backfaces” by painting on the back edge of the model (cube).
9. Change shape back to “Sphere”, import a test model with holes (e.g., fence), and verify that “Continuous” option prevents painting neighboring parts with gaps between them.
10. Change to Paint Bucket and verify that the tool respects the new details created by Brush.
11. Change to Triangle Painter and verify that the tool does *not* respect the details created by Brush and instead uses the original triangle topology.

Text Editor

1. Run Pepr3D, select Text Editor. Verify the side pane is populated with a palette and options. All options should have a tooltip.
2. Default font should be “OpenSans-Regular.ttf”, size 12, 3 Bezier steps, “Pepr3D” text, scale 0.20, zero rotation.
3. Select a different color and verify that hovering mouse over a model shows a 3D line which corresponds to normal vectors.
4. Clicking should show a preview of the text (“Pepr3D”) together with the black normal vector and two additional surface vectors.
5. Clicking multiple times on different positions should update the preview. No text should actually be painted yet!
6. Only after clicking “Paint”, the text should actually be painted. Verify a progress indicator is shown, progress corresponding to the individual letters being painted.
7. Verify that changing the size, scale, and rotation updates the live preview and also changes how the text is painted with “Paint”.
8. Verify that the text can be changed, including empty text and long texts.
9. Try loading a different font and repeat the steps above. If the font is a little bit more complex, try increasing “Bezier steps” to see how curves are more detailed.

Automatic Segmentation

1. Run Pepr3D and import `bunny.obj`. Click “Compute SDF”, which should be the only available option in the side pane. Verify a progress indicator appears. After computing the SDF is finished, “Segment!” button and 2 sliders should appear in the side pane.

2. Keep the default 20% robustness and 30% edge tolerance defaults and click “Segment!”. The bunny should be segmented into 6 regions numbered 0 to 5. A color palette and segment buttons should also appear in the side pane below the previous options, together with “Accept” and “Cancel”.

3. Select “Cancel”, the bunny should revert to its original blue color, side pane should show the main options again.

4. Change edge tolerance to 50% and click “Segment!”. The bunny should be segmented into 5 regions corresponding roughly to 2 ears separately, head, both front legs together, and the rest of the body.

5. Hovering over triangles in model view should highlight the corresponding segment buttons in the side pane. Left-clicking a triangle in a segment should apply a color from the palette to the whole region and also change the color of the segment button in the side pane. The same behavior should apply when one clicks on the segment button in the side pane instead of on a triangle in the model view.

6. Pressing “Accept” if not all regions are colored with palette colors should not do anything. Color all regions and press “Accept”. Verify the bunny is now colored with the corresponding colors and the side pane reverts to the initial options.

7. Set robustness to 0% and edge tolerance to 100%. Two segments should be created, corresponding roughly to the head with ears and the rest of the bunny.

8. Verify that canceling does not affect Undo and Redo. Verify that accepting a segmentation is undoable.

Manual Segmentation

1. Run Pepr3D and import `bunny.obj`. Click “Compute SDF”, which should be the only available option in the side pane. Verify a progress indicator appears. After computing the SDF is finished, a color palette should appear in the side pane.

2. Verify that computing the SDF in Manual Segmentation means that it is no more necessary to also compute it in the Automatic Segmentation, and vice-versa.

3. Select a different color and roughly paint some triangles in both bunny ears. Once triangles are painted, a “Spread” slider with 0% should appear together with unchecked “Hard edges” and “Region overlap” and an “Apply” and “Cancel” button.

4. Increase the spread to roughly 5%, verify that both ears get fully colored and the coloring stops at the place where the ears connect with the head. Increasing towards 35% also colors the head, 100% colors the whole bunny.

5. Verify that Cancel and Accept work as expected similar to Automatic Segmentation, also with Undo.

6. Undo back to the original blue bunny. Select two different colors, color ears with one and tail with another. Verify that increasing the spread increases the colored regions. With “Hard edges” enables, one color should never appear “inside” another region and the “Region overlap” option should be hidden. With “Region overlap”, spread at 100% should color the whole bunny with one color, without “Region overlap” it should not happen.

Display Options

1. Run Pepr3D, select Display Options.

2. With “Dolly”, zooming in is able to get inside the default cube. With “Field of view”, this is not possible. “Reset camera” resets the view to the default position and rotation.

3. “Model roll” rotates with the object in the third axis which is not possible to rotate with right mouse button dragging over the model view. Changing model position moves the model on the grid, when height is changed, the grid moves *together* with the model. “Reset model transformation” reverts all these changes back to the default ones.

4. When “Show grid” is enabled, a grid is shown, and vice-versa. When “Show wireframe” is enabled, every single triangle is highlighted and hovering mouse over them in Triangle Painter highlights them with an opposite color.

Settings

1. Run Pepr3D, select Settings. Two categories should be shown, “Edit Color Palette” and “User Interface”, both open by default. Clicking their headers should toggle between open and close.

2. Verify that changing the “Side pane width” changes the side pane width, with the possible minimum of 200 pixels. Maximum should never be as high to completely cover model view, but icons in the toolbar may be covered.

3. Verify that the color palette shows the same colors as in the tools. By default, there are 4 colors, which can always be reverted by clicking “Reset all colors to default”.

4. Clicking “Add color” adds a new color with a random hue, maximum of 16 is allowed. If more colors want to be added, an error dialog is shown explaining the problem.

5. Drag-and-dropping a color rectangle to the “Drag color here to delete” region removes a color. It is forbidden to remove all colors, an error dialog is shown explaining the problem. When a color is deleted, it is also removed from the model. Verify that by painting the cube. Triangles painted with a removed color should get the first color in the palette.

6. Drag-and-dropping a color rectangle to another color rectangle reorders the colors in the palette, but does *not* change any colors on the model. But when Control is held during the drag-and-drop, verify that the colors are also swapped on the model as well as in the palette.

7. Left-clicking a color rectangle opens a popup where the color can be changed. Verify that changing a color changes it on the model as well in real time.

Information

1. Run Pepr3D, select Information. Verify that the URL of Pepr3D GitHub repository is shown.

8.2.3 Exporting

These scenarios verify that the Export Assistant and exporting geometry work as expected and exported models can be used in 3D printing.

Exporting an extruded cube

1. Run Pepr3D, select Export Assistant. Verify that File → Export also opens Export Assistant. By default, "Depth extrusion" export should be selected, the "Update extrusion preview!" button should be highlighted, and "Export preview" and "Please update the extrusion preview." captions should appear on the top-left corner of the model view. Default extrusion depths are 2.50% absolute, all colors are previewed, and `.stl` is selected without creating a new folder. Preview height is the whole range of zero and hundred percent.

2. Click "Update extrusion preview!". The blue cube should appear, button should not be highlighted anymore and the red caption should disappear. Changing the preview range should "open" the cube from the bottom and top respectively.

3. Select Triangle Painter and color one face with a different color. Select Brush and make a circle with another color on another face.

4. Select Export Assistant. Verify that the update button is highlighted again. Click it. The preview should now get updated reflecting the new coloring. Inspect the interior of the cube by deselecting "Preview" of the various colors and changing the preview height. Verify that the cube is extruded inside!

5. Change the depths of the colors. Verify that the update button gets highlighted. Try setting the depth to 0% for one color and verify that it disables extrusion for that color.

6. Change depth values to "relative to SDF". Verify that the update button gets highlighted. Click it. The extrusion should now get a little bit uneven as the SDF values for a cube are not constant for all vertices. Verify!

7. Check that the extruded model can get exported in all three different formats. Clicking “Export files” should open a file dialog, enter a file name and verify the files got saved. If you followed this scenario, 3 different files should be exported for the model for every format as there were 3 different colors used.

8. Verify that “Create a new folder” indeed creates a new folder when “Export files” is clicked. The folder name should be the same as the name written in the file dialog.

9. Try importing the exported files back to Pepr3D. Inspect them, verify they correspond to the different parts of different colors, and verify they are indeed extruded unless 0% was chosen for the color.

10. Open Slic3r PE and drag-and-drop the 3 files *together at the same time* to the main window of Slic3r PE. Confirm the dialog asking if you want to import them as a single object with multiple parts.

11. Verify that the parts are correctly positioned so that together they form the original cube that you painted. Click “Slice now” (on the right), it should succeed. Go to “Preview” tab (on the bottom) and move the sliders on the right to inspect the layers of the model. It should correspond to the extrusion parameters you set in Pepr3D and it should correspond to the Export Assistant preview.

12. Optionally, try printing the cube.

Exporting surfaces of a cube

1. Run Pepr3D, select Export Assistant. Select “Surfaces only”. Verify that the whole extrusion part of the side pane gets hidden and that a “No preview available for surface export.” caption is shown in the top-left corner of the model view. Nothing except a grid should indeed be shown in the preview.

2. Select Triangle Painter and color one face with a different color. Select Brush and make a circle with another color on another face.

3. Select Export Assistant and try exporting the cube in the various formats.

4. Import the exported parts back to Pepr3D and verify they are indeed only surfaces and they are not extruded. Different colors should still be separated into different files.

Exporting an SDF extruded bunny

1. Run Pepr3D, import `bunny.obj`, paint it using Triangle Painter and Brush.

2. Select Export Assistant, select “relative to SDF” depth values. Keep all depths at a same value. Click to update the preview.

3. Verify that the depth of the extrusion depends on the thickness of the parts of the bunny. For example, ears should be extruded *in lower depth* than the body of the bunny. This is because SDF is lower in the ears, so the depth should also be lower. Inspect various parts of the bunny.

4. Try exporting and verifying in Slic3r PE as explained in the first scenario with the cube.

8.2.4 Error behaviors

These scenarios verify that error situations such as loading a corrupted file are handled correctly. This means that errors should be explained to users in an error dialog. In case Pepr3D crashes entirely, the crash should be logged in a log file and a dialog explaining the log file should appear when the user runs Pepr3D again. All files mentioned in this section are located in the Pepr3D repository in the `assets/models/invalid` directory.

Opening and importing invalid models Run Pepr3D and try opening and importing the following files by drag-and-dropping them onto the main Pepr3D window, by using File → Open, and also by using File → Import. These files are invalid and error dialogs should be shown when trying to handle them in Pepr3D.

- Opening and/or importing `invalid.obj`, `README.md`, or other random files such as JPEG files should result in "Error: Invalid file" dialog. The geometry that was loaded before (e.g., the default cube) should stay opened.

- Importing `invalid_sdf.stl` should work without any problems, but pressing "Compute SDF" in the segmentation tools or using "relative to SDF" in Export Assistant should result in "Error: Failed to compute SDF" dialog. After the dialog is shown, both segmentation tools should be disabled. In Export Assistant, the "relative to SDF" option should be hidden. Exporting should still work, just not with SDF values anymore.

- Importing `invalid_polyhedron.stl` should work but a "Warning: Failed to build a polyhedron" dialog should be shown explaining the model is probably non-manifold. For these models, only Triangle Painter should work, other editing tools should be disabled. Exporting should also work, but the "relative to SDF" extrusion option should be hidden.

- Opening `corrupted.p3d` should result in "Error: Pepr3D project file (.p3d) corrupted" dialog. The geometry that was loaded before (e.g., the default cube) should stay opened.

- Opening `corrupted_crash.p3d` may crash Pepr3D entirely as it looks like valid serialization data but is not. This is a limitation of our serialization library Cereal. Verify that "pepr3d.crashed" file was created. Run Pepr3D again and a dialog "Pepr3D previously terminated with a fatal error" should appear explaining the crash and where the user can find the log file. Pressind "Continue" should close the dialog and everything should be back to normal.

Error when saving or exporting a file Run Pepr3D and open a `.p3d` file that is set as read-only in Windows (right-click the file, Properties, check Read-only). Try to save the file. An error dialog should be shown explaining the file to save into could not be opened. Same should happen when exporting a file to a read-only directory or file.

Other errors Some other error situations (such as removing the only remaining color in a palette) have already been described in other testing scenarios and are not mentioned here again. There are also other error situations which may occur (such as providing a geometry file which is corrupted but in a way that cannot be detected early) and are covered by Pepr3D by showing error dialogs, but we were not able to provide reasonable example files here in the manual testing.

8.3 Continuous integration

Continuous integration is a software engineering term used to describe the work flow of a team based project, which is based on merging the work of many individuals into a main stream often [2]. In particular, *Circle CI* is a free service which can be integrated into GitHub's interface, which allows the users of the repository to perform all kinds of checks and tests before the code is merged into a branch (most commonly the *master branch*.

We performed three checks before allowing the merge into a different branch, namely:

- **clang-format check** – by running clang-format on the whole codebase and comparing it to the one before the run, the software determined if all of the code is properly aligned and follows our coding standards. This benefits us in two ways. Firstly, we minimize the number of git conflicts, because the code is properly formatted. Secondly, this makes the code uniform and such it removes any personal preference in coding styles. The second property is important because it makes reading the code much more programmer friendly – once formatted, you cannot distinguish between your and the others' code, which makes reading it much easier, as you are not bothered by different standards of formatting.

- **ability to compile** – code that does not compile is very dangerous in a repository, especially in the *master* branch. If we need to step back in the *master* branch history to trace the origins of a bug, we want to be building the program and testing it for the bug to find the commit that introduced the bug. If we run into code that does not compile, this methodology is much harder to execute. Our check was performed on a Linux machine, so it had another positive outcome for the team. The team developed on MSVC as our main target was the Windows OS. However, g++ has different and sometime better checking for errors than MSVC, which allowed us to catch some mistakes during compile time on Linux, which we did not see on MSVC.

[2]https://en.wikipedia.org/wiki/Continuous_integration

- **unit testing** – the last check the code needed to pass was the unit tests, which we already discussed. This point is rather simple, if the tests fail, the added code would break Pepr3D, and as such should not be committed.

We used Circle CI [3] and integrated the service into GitHub.

[3]https://circleci.com/

Chapter 9

Building the project

In this chapter, we explain how Pepr3D can be built from the source codes. We assume some knowledge of build systems, compilers, and operating systems as this is a guide for developers.

9.1 Building on Windows

We explain how the 64-bit Pepr3D can be built on Windows 8 and 10, which are the officially supported platforms.

9.1.1 Repository

First of all, the official Pepr3D git repository has to be cloned.[1] This requires git to be installed on the machine and then cloning the repository using the following command in the Windows command line:

```
git clone --recurse-submodules -j8 https://github.com/tomasiser/pepr3d.git
```

If you have already accidentally cloned without submodules, run this command from the root directory of this repo:

```
git submodule update --init --recursive
```

9.1.2 Dependencies

The following dependencies have to be downloaded and/or installed on the machine according to these steps:

1. Download and install the latest **CMake** from https://cmake.org/.
2. Download and install either **Visual Studio 2017** (Community version is enough) or alternatively only **Build Tools for Visual Studio 2017**. Both can be found at https://visualstudio.microsoft.com/downloads/.

[1]Alternatively, use the attached CD (see Appendix II).

3. Download and install **CGAL** from https://www.cgal.org/download/windows.html. Make sure `CGAL_DIR` environment variable is set to the installed CGAL path, which is done by default when using the official installer.

4. Download and install **Boost** from https://www.boost.org/. You can either build Boost yourself or download pre-built binaries for the 14.1 toolset. Make sure to point `BOOST_ROOT` environment variable to the installed Boost path.

5. We use our own built version of **Assimp** from the latest `master` branch. Either build Assimp yourself from https://github.com/assimp/assimp, or download and unzip our prebuilt version[2]. Our version is built with two `.dll`, one for Debug and one for Release. Do not mix them up! Make sure to point `ASSIMP_ROOT` environment variable to the Assimp directory.

6. Download **Freetype** from https://github.com/ubawurinna/freetype-windows-binaries, preferably version 2.9.1. After downloading, it is *necessary* to rename the `win64` subdirectory to `lib`. Make sure to point `FREETYPE_DIR` to the Freetype directory.

All other libraries are part of the Pepr3D repository and will be built automatically by our build system.

9.1.3 Building

From the root directory of the cloned repository, run the following from the command line, which creates a new `build` directory and runs CMake inside:

```
mkdir build
cd build
cmake -G"Visual Studio 15 2017 Win64" ..
```

Now the build project is prepared inside the `build` subdirectory and we can now open `build/pepr3d.sln` in the Visual Studio 2017 application and compile Pepr3D from there.

Building from command line Alternatively, we can build Pepr3D from the command line using the build tools. We have to start **MSBuild Command Prompt for VS2017** or **Developer Command Prompt for VS 2017** from Start Menu, or we can also start the command prompt from a standard command line using:

```
%comspec% /k "C:\Program Files (x86)\Microsoft Visual
   Studio\2017\Community\Common7\Tools\VsDevCmd.bat"
```

In the Visual Studio command prompt, we can build Pepr3D using:

```
msbuild pepr3d.sln /m
```

[2]https://github.com/tomasiser/pepr3d/releases/download/v1.0/Assimp_for_Pepr3D.zip or use the attached CD (see Appendix II)

9.1.4 Running

The executable of Pepr3D should be located in `build/pepr3d/Debug/pepr3d.exe` (or `Release` instead of `Debug`).

After building in Visual Studio, we have to make sure the appropriate `.dll` files are copied next to the executables. If you used Visual Studio 2017 application to build it, some of the `.dll` files should be automatically copied to the executable directory. If you used command line to build, you need to copy `libgmp-10.dll`, `libmpfr-4.dll`, and `assimp-vc140-mt.dll` manually from the `build/` directory to the same directory as `pepr3d.exe`.

Copying correct Assimp `.dll` Note that by default, the Release version of Assimp `.dll` is copied. If you built a Debug version of Pepr3D, you need to replace `assimp-vc140-mt.dll` by the file located in the directory where you unziped our Assimp library. The Debug library is in the `bin/x64-Debug` subdirectory of Assimp instead of in `bin/x64`. If you built Assimp on your own, you need to compile it in the same Debug or Release as Pepr3D.

Copying Freetype `.dll` If you do not have `freetype.dll` as a part of your operating system already, you also need to copy this file next to the executable from the `lib` subdirectory of the Freetype you downloaded as described in the Dependencies subsection.

Running unit tests By default, the Debug executable of all Pepr3D unit tests is build into `build/Debug/pepr3dtests.exe`. It is necessary to also copy the `.dll` files there.

9.2 Building on Linux / Docker container

There is a possibility to build Pepr3D on Linux systems, but please note that is in only supported for verifying that the source codes do compile as necessary for continuous integration (Section 8.3). It is not indended for running and using Pepr3D in release.

We have a Linux Docker container[3] in our special repository at GitHub: https://github.com/tomasiser/docker-cinder. The `latest` image setups a Debian environment to build Cinder applications. The `prebuilt` image actually builds Cinder on top of the `latest` image. Pepr3D can be built in the `prebuilt` container by running `cmake` and `make` commands from the Pepr3D repository.

Note that in order to compile Pepr3D using the container, one needs to have at least a minimal experience with Docker. We advise to follow the tutorials on the official Docker website[4].

[3]https://www.docker.com/resources/what-container
[4]https://docs.docker.com/get-started/

Part III

Progress and results

Chapter 10

Progress of implementation

In this chapter we first cover the tasks and responsibilities of each of the team members. Then we outline the progress of the implementation of this project.

10.1 Responsibilities

Here we list the members of the team alphabetically and summarize all the work each of them has done over the course of the project.

10.1.1 Bc. Štěpán Hojdar

- Implementation of the Geometry class as a data structure, which entails both computational geometry (colouring the triangles, etc.) and rendering capabilities (creating buffers for OpenGL). Testing and integrating the CGAL library into the project and using this library to perform the computational geometry.
- Implementing the following tools both on the backend and the frontend: bucket painter, manual segmentation and automatic segmentation.
- Researching a way to convert a font file (.ttf) into a 2D triangle mesh, implementing the FontRasterizer class and implementing the basics of the text tool, using this knowledge.
- Serializing and deserializing our data using the Cinder library, which allows us to save work in progress as a .p3d file.
- Writing a major part of both the specification and the documentation.

10.1.2 Bc. Tomáš Iser

- The majority of the user interface of the application, including the design of the GUI, the architectural design of the backend of the UI.
- Small widgets – wrappers around Dear ImGui calls to make it easier to use for the rest of the team while developing the UI.

- The color palette editor, shortcuts, correctly scaling and rotating the model, all of the display options (wireframe, two zoom options), tooltips, dialogs, error handling and logging and application settings.
- Prototyping, testing and integrating the Cinder and Dear ImGui libraries which we based the project on.
- The Export Assistant (the visualization part) and Triangle Painter tools.
- Writing a part of both the specification and the documentation (UI, build).
- Connecting our GitHub repository to Circle CI, which is a continuous integration service based on linux. We used the CI throughout the whole process, which made sure every single merge into the master branch was buildable and passed all unit tests.

10.1.3 Bc. Jindřich Pikora

- Communication with Prusa Research s.r.o., including several meetings with our contact in Prusa Research.
- Testing and familiarizing himself with the FDM printing in practice, printing a lot of test subjects to measure the printer's capabilities and downsides, to note during our export development.
- Testing and integrating the Assimp library into the project.
- Implementing the whole import process, with mesh pre-processing, simplification and repairs provided by the Assimp library.
- Implementing the whole export process, researching and developing a usable heuristic to make the process smoother and less error prone. Testing the export by physically printing the results on our printer.

10.1.4 Bc. Luis Sanchez

- Setting up the project environment using CMake, making sure all our libraries compile and link correctly.
- Designing and implementing the whole command architecture, with functioning undo and redo operations on the geometry data.
- Implementing the Brush and Text tool backend and frontend, requiring long and extensive research and developing a brand new way to solve this problem, which we did not find in any available literature or research papers.
- Modifying and extending the Geometry data structure to allow for custom triangle subdivisions.
- Performing complex operations using the CGAL library on the modified Geometry in order to make the brush work correctly.

- Modifying the rest of the tools to be able to work on the new custom modified geometry, as well as to allow it to correctly export and serialize as a .p3d project.

10.2 Timeline of the implementation

In this section, we describe the process of implementing this project from start to finish. We start by explaining the project setup, rules and other measures we employed to get more productive, then we describe the process itself.

10.2.1 Rules and project setup

Team management

To manage the work in the team, we set several goals. We met regularly each week with our supervisor, and had a structured meeting. The first part of the meeting had each of us tell the rest of the team what we worked on the last week, describe what went well and if/where we got stuck. This had two effects – firstly, it allowed us to help the stuck member and not waste too much time, and secondly, this made sure that all the team members are up to date with the progress of the whole project, which motivated further progress. The second part of the meeting had us setting goals for the next week, assigning clear and doable tasks to each team member, which would get reviewed on the next meeting. Each meeting took around an hour, including a general discussion after these two structured points.

Git repository

The second major part of the teamwork was our git repository, which we setup on GitHub [1]. We employed several measures to ensure the quality of the code and to avoid issues like a master branch that cannot be compiled.

Firstly, we disabled any way to push directly into the master branch. This had the effect that every contribution has to go through GitHub's **pull request** mechanism. We also set the pull request merge to require one approving review. This means that for each merge (and therefore commit), two people were required to read the code - the one who wrote and tested it, and the reviewer, whose only job was to go over it, try to compile it and point out any weak programming in the code.

Secondly, as we already mentioned in the previous section, one of our team member set up a continuous integration service, called Circle CI [2]. We required the CI to be run on every pull request that was sent towards the master branch. The CI would notify us if the pull request either did not compile, or did not pass all of the unit tests. Because Circle CI utilizes a linux server to build the program, it also meant that we would be sure it compiles on both Windows and Linux all the time.

[1]https://github.com

[2]https://circleci.com/

Last but not least, we also made sure to set the CI to check the code formatting. We have discussed and configured the clang-format tool [3] to format our code in the same way, to avoid mixed coding standards. If the code was not formatted correctly, the pull request wouldn't go through.

These two measures helped us immensely and made the code more reliable in the long run.

10.2.2 The process of the implementation

Here, we will go over the process of the implementation, week by week, as can be seen in the git log, starting on 01.10.2018, when we sent the specification to the committee.

01.10. - 08.10.

By now, we have had a functioning repository, since we used it to create the specification as well. We also had the continuous integration working. This week we started to implement the basic functionality, so far in separate projects, because the CMake of the whole project was not finished yet, so the project didn't include all the necessary libraries (e.g. CGAL or Assimp). The application was running, but there were no responses to the buttons and nothing to render. The basics of the command manager also got implemented, even though they would wait for another month before being applied to the Geometry class.

09.10. - 16.10.

This week we added the basic Geometry class and ray-shooting capabilities using CGAL, because the CMake finally accepted the CGAL library. We started rendering the geometry in the ModelView (for now a triangle) and could shoot rays.

17.10. - 24.10.

This week the Assimp library got added into CMake, and the ModelImporter was merged, which meant we could import models into the geometry, and render them in the ModelView window. We had begun to try to debug normals of the mesh Assimp gave us, which will take a bit more time, since the library is not clear on what it does in the documentation.

25.10. - 1.11.

We added ray-casting from the model view, which happens on a mouse click, which allowed us to finally get the Triangle Painter functionality to be complete – we could click on a triangle and change its color. We also started adding unit testing, for now only for the Geometry class. Drag and drop was now also a supported way to load a model. We redid the color palette as a integer based data structure, instead of RGB color notation.

[3]https://clang.llvm.org/docs/ClangFormat.html

2.11. - 9.11.

This week we struggled with the CGAL library and managed to get the bucket painter to do a breath-first-search over the model. We also modified the shader to accept the the colors as integers and then a color palette array, which allows for real-time color swapping done in the color palette.

10.11. - 16.11.

Assimp finally stopped loading degenerate triangles, which was due to the wrong setup and what we believe is a bug in the library, which we solved by double checking the output. We also extended the bucket painter tool to allow for stopping on different criteria (like edge sharpness or color), and modified the Command Manager to be more memory friendly and customizable. The UI received a highlight for the hovered triangle, and an editable color palette. We also added some basic error handling, bucket painter UI, and made the Undo and Redo work on bucket painter.

17.11. - 23.11.

Geometry got a big refactoring, which removed a lot of lower quality code that got detected in a code review and fixed a few warnings that were showing up on g++ and not on MSVC. The camera handling got improved and now always fit the model, instead of always pointing in the same direction. Bucket painter got a prettier UI and better stopping criteria. A working export is finished, and the team notices a few cases, which break the export. Research and testing will continue in the following weeks to try to find a way to make the export more robust.

24.11. - 1.12.

Loading a new geometry is now done in parallel, using a threadpool. Brush development is starting and the backend for automatic segmentation is done. The frontend for automatic segmentation is developed, and a new way of rendering custom colors is added. Work also starts on implementing the manual segmentation. Tools that are disabled (because the user loaded a non-valid model) are now greyed out and cannot be selected. Dialogs get implemented to show the user progress while loading a new model.

2.12. - 9.12.

Manual segmentation is done, but the team is not happy with the handling. We discuss the behaviour during a meeting and the behaviour is changed to a different one, which we are happy with. UI Tooltips get implemented, both for Tools and for the tool configuration. Work is also starting on serializing and deserializing the geometry. The brush starts to work, but is really slow and needs optimization.

10.12. - 17.12.

Brush is still getting improved, now is able to undo and redo the operations. Serializing and deserializing is done, the unsaved asterisk mark gets added, as well as *Save* and *Save as* options. The OpenGL buffers get redone, so they do not recalculate every frame. A few crashes are fixed and a lot of refactoring is done on the existing code.

18.12. - 01.01.2019

Hotkeys are updated, more tooltips get added. The brush is getting UI settings (like size of the brush) and gets a highlight around the cursor. It's Christmas, so work gets slowed down. Work on export is done, using the CGAL library to determine the thickness is accepted by the team as a viable way to prevent the majority of bugs.

01.01. - 08.01.

Font conversion from a .ttf into triangle meshes gets researched and added. The UI is getting final polishing, spell-checking and gets a scrollbar. The new geometry from Brush is getting fixed in other already existing Tools, and a lot of error dialogs and crash prevention is done. Logging is improved and exception handling for multi-threaded load and import is fixed.

09.01. - 16.01.

Export is getting reviewed, brush is getting reviewed and bug fixed. The functions that convert the Brush tool into the Text tool get added. The team decides to start writing this document, since the program is almost feature complete.

17.01. - 24.01.

This document gets started. The brush is still getting polished and the export is getting merged into master. The attention of half of the team is shifted towards this document, while the remaining two members finish the few remaining features of the program.

25.01. - 01.02.

Documentation is being written, mostly by Štěpán as other team members need to focus on fixing the remaining bugs. After meeting with Oskar, we decide to implement the Export Assistant tool to improve the export process significantly.

02.02. - 09.02.

The Export Assistant tool is finished, exporting is being verified. The remaining bug in the Brush tool is being debugged and fixed. The documentation is being finished. We focus on the documentation and finding a good model to showcase the results of our work.

10.02. - 17.02.

Debugging the application and trying to find every case of a crash and make the program print out an error message, instead of crashing. Documentation is being written by half the team, while the other half handles the final polishing.

18.02. - 25.02.

More memory crashes are fixed and tests for the `TriangleDetail` class are added.

26.02 - 05.03.

More polishing on the program – fixing serialization, tests, removing unused debug code. The build process for UNIX operating systems has been fixed and Pepr3D can now be built on Linux machines.

06.03. - 13.03.

Improving the UI of the newly implemented tools (brush and text). Geometry class details have been added to the programmer documentation.

14.03. - 21.03

Meeting with doc. Ing. Jaroslav Křivánek, Ph.D. and Ing. Vojtěch Bubník from Prusa Research s.r.o. has taken place. We demonstrated the software to both gentlemen and decided it was ready for submitting.

22.03. - 31.03.

Final polishing, filling in the last pieces of documentation, compiling a bunch of models that we would like bundled with the application for users to be able to try the application.

Chapter 11

Comparison to minimal requirements

In this chapter, we focus on comparing the finished product with the minimal and advanced requirements we set in the specification, before we started to implement the project.

11.1 Minimal requirements

We go through the minimal features one by one and elaborate on if and how well we achieved this goal.

- *Loading a model from a basic 3D format* – our application supports .STL, .OBJ and .PLY, which are the three most used formats on the 3D printing market today.
- *Export a multi-coloured .STL file, which can be entered into the slicer* – Upon discovering the slicer more, and having time to physically print something, we realised, that the slicer does not actually support a single multicoloured .STL file. We changed this goal to exporting a single .STL file for each color, which only contains the triangle of the chosen color. Our application supports this export, even though it is simpler and more error prone than the other (fully 3D) supported export.
- *Bucket painter with a simple criterion* – Our bucket painter currently supports edge sharpness, whose threshold the user can alter, a different color stopping, or the combination of both.
- *Basic form of edit history with undo and redo steps* – This feature is working exactly as promised, with infinite amount of steps to *Undo* and *Redo*. It is not a tree-like structure and it will overwrite the future upon *Undoing* and then applying new commands. We observed that this is the case in many 3D applications.
- *Functional 3D UI allowing zooming and rotating the model* – We tried several methods of zooming (which are selectable in the settings menu) and we fit the model into the default view. This means that the size of the loaded model does not matter, it will always fit into the view when loaded.

Using this summary we conclude that we met the minimal requirements.

11.2 Additional features

We will now discuss the advanced features we disclosed in the specification, and compare the proposed feature with the implemented one.

11.2.1 Automatic and semi-automatic segmentation

While writing the specification, we thought that the semi-automatic segmentation (called *Manual segmentation* in the application) would be the most used feature of the program. Upon implementing both segmentations, we actually think the automatic segmentation achieves the goal of quickly colouring the model much better. Meanwhile the manual segmentation is better for fine tuning some parts of the model, because it allows for colouring one part consistently, while leaving the rest intact (which the automatic one cannot do).

In conclusion, we placed the automatic segmentation as second to last on our feature list, but we strongly disagree with the placement in hindsight and think the tool is one of the most usable tools in the application.

11.2.2 Text tool

In the specification, we discussed two extension to this tool: **text projection** and **fonts**.

Text projections Starting with text projections, we added more than just X/Y/Z – the user is now able to click on individual triangles, and the projection angle will be taken as the clicked triangle's normal vector. A real-time preview is displayed (the text is hovered above the clicked triangle) so the user can see what is going to happen after projecting. We chose this option because it was not much harder to do than the promised X/Y/Z, while adding a lot of functionality. The main perk of this method is the ability to reproduce the results reliably (the angle will be the same every time you click on a particular triangle), while giving the user more freedom than just X/Y/Z.

On the other hand, we did not implement the cylindrical or any other special projections, mainly because we lacked the manpower to do everything we set out to do. The second reason is a more practical one. The application focuses on the *WYSIWYG* pattern – *What you see is what you get* to be as intuitive as possible. Dealing with cylindrical and other complicated projections is not a task we can expect from a basic user.

Fonts Regarding the fonts, we were able to implement a class, called FontRasterizer, which takes the .ttf file and a font string, and creates triangle meshes out of it. This allows us to work on any font the user provides. However, the library we used (you can get more details about this class in the implementation section of the documentation) seems to have trouble with the non-letter characters (like the WiFi icon) we mentioned in the specification, which means this extension goal was fulfilled half way.

11.2.3 Brush and adaptive triangulation

This topic is very in-depth, and we would advise the reader to read through the implementation chapter first, but simply put, our implementation is the closest we could get to making it safe to use. This means that repeated painting on the same spot of the model, with the same color, does not subdivide the triangulation more. The brush also tries to simplify the topology already created – for example, if you select the red color and paint over a blue detail, erasing it, the triangles of the detail do not stay, but get merged back together, which simplifies the topology.

11.2.4 Hotkeys and customizability

As we mentioned in the specification, very few users generally use hotkeys. However, we wanted to provide the option of changing the hotkeys anyway. In our application, the hotkeys are saved as a .json file, structured as the following example illustrates.

```
{
   "key": {
      "ctrl": false,
      "keycode": 112
   },
   "value": "SelectPaintBucket"
},
...
```

This .json is readily available next to the application's main executable file to edit by the user, as he sees fit. The *keycode* values are provided in a separate file next to it, in the following format:

```
KEY_a     = 97,
KEY_b     = 98,
KEY_c     = 99,
...
```

We understand that this is not the most user-friendly way to change the hotkeys, but we believe, that if the user is advanced enough to want to customize his hotkeys, this process is simple enough as to not cause any issues.

11.2.5 Radial menu

In the specification we mentioned the possibility of adding a radial menu around the cursor. In the end, we did not implement this feature. This decision was made, because we saw many more areas of the application that could be improved and focused on instead. We thought that the users would benefit more from these improvements than the radial menu feature.

11.2.6 Triangle subdivision and decimation

While writing the specification we put this feature as the lowest priority feature, because we thought only the most advanced users would be able to utilize it. While developing the application, we downloaded and tested many models that are on the internet for anyone to download and print. The websites we used include Thingiverse [1] and yeggi [2]. We noticed on many of these models, that many are unoptimized, include holes, unreasonably small or wastefully many triangles. From this observation we concluded that the users do not generally optimize and micromanage their models since a few operations done in software like Blender can reduce this waste by a big percentage. All of this made us decide to not include the feature, as we generally do not believe the users would use it or be able to use it to greatly improve the model.

11.2.7 Model exporting

For completeness' sake, we discuss the model exporting feature here. We did a lot of research and tried many different approaches, and the one implemented in the application looked to the team as the best solution. You can read more about the chosen method in the implementation part of the developer documentation.

Here we state that this feature was a priority for us, as we have shown in Section 10.1, one of our team members spent a big amount of time trying to optimize this feature. We believe we came up with a way that should at least help, if not solve clipping and other unwanted occurrences in the majority of the scenarios, though we do have examples of wrong behaviour. We also add manual control over the feature, to allow the user to fix the issue manually, should any issue arise.

[1] www.thingiverse.com
[2] www.yeggi.com

Chapter 12

Results

In this chapter, we showcase the pipeline we have managed to create on a simple, low polygon-count model. We also attach all the necessary files to recreate the steps taken here. We use a model downloaded from the internet, which is the expected use case of our application.

12.1 Acquiring and preprocessing the model

As a beginner, the user will not create his own model, but download the free ones from the internet. Here we hope to demonstrate the pipeline from the user's perspective, so we will do the same. We will be using this model [1] from Thingiverse [2].

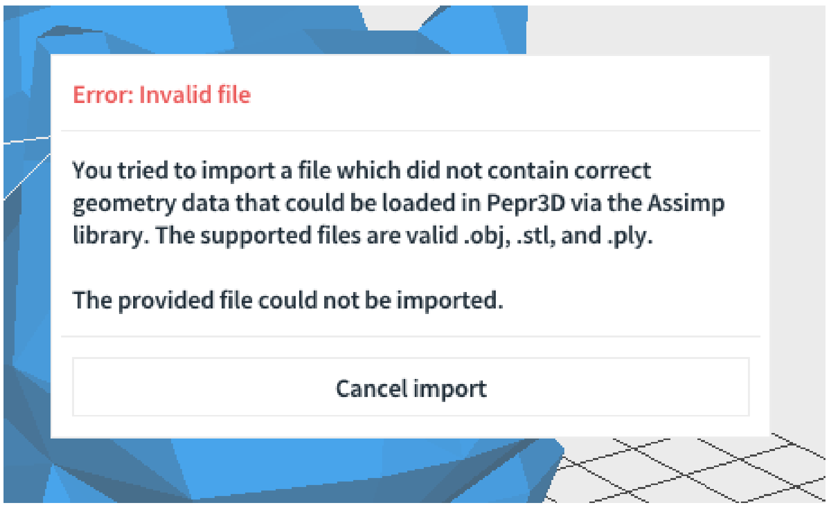
Before we can get to Pepr3D, we unfortunately have to pre-process the model somewhat, as the artist forgot to specify the model's normals. This is a very easy correction in Blender, but already showcases the fact, that the models found on the internet suffer from a plethora of problems, some of which are detectable and correctable within the program and some of which are not. We attach the already corrected .STL file as well.

12.2 Loading the model into Pepr3D

Once we have our model cleaned up – removing all duplicate vertices, reducing the model to a manifold object and making sure the normals are pointing out, we can load the model into Pepr3D. This is done by selecting **File** – **Import** or simply dragging and dropping the .STL file into Pepr3D.

In our case, the model gets loaded successfully in under a second. Larger models (like the bunny.obj, which we have attached), load slower and an asynchronous dialog informing the user about the loading progress is displayed. Other models can be corrupted, the files do not correspond to a single object or be otherwise unloadable. In this case, a dialog is displayed, notifying the user that the file is damaged and explaining what can be done to prevent this. In some scenarios, Pepr3D remains usable with a limited functionality, in others, the model does not load. You can refer to the following figures 12.1 and 12.2 for illustration.

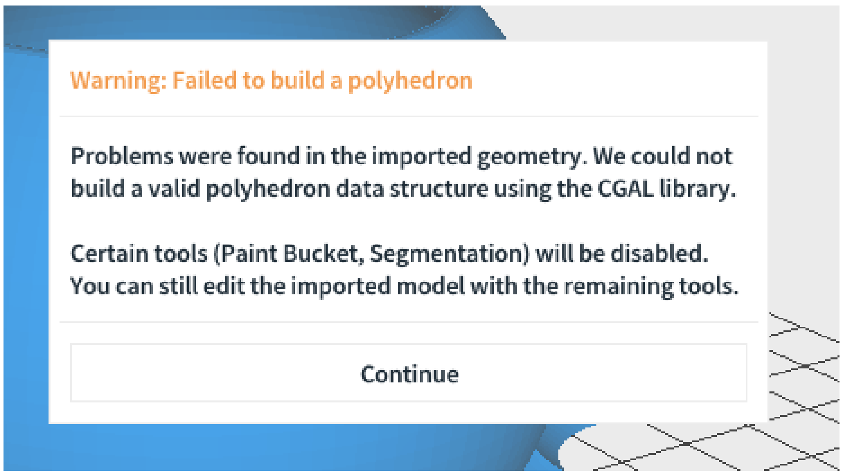
[1] https://www.thingiverse.com/thing:327753
[2] https://www.thingiverse.com


Error: Invalid file

You tried to import a file which did not contain correct geometry data that could be loaded in Pepr3D via the Assimp library. The supported files are valid .obj, .stl, and .ply.

The provided file could not be imported.

Figure 12.1: Loading a file that does not contain any geometry will result in this error.


Warning: Failed to build a polyhedron

Problems were found in the imported geometry. We could not build a valid polyhedron data structure using the CGAL library.

Certain tools (Paint Bucket, Segmentation) will be disabled. You can still edit the imported model with the remaining tools.

Figure 12.2: While this file could be loaded, the file does not conform to a certain assumption of some of the algorithms. Tools using these algorithms will be disabled, but the other tools will work correctly.

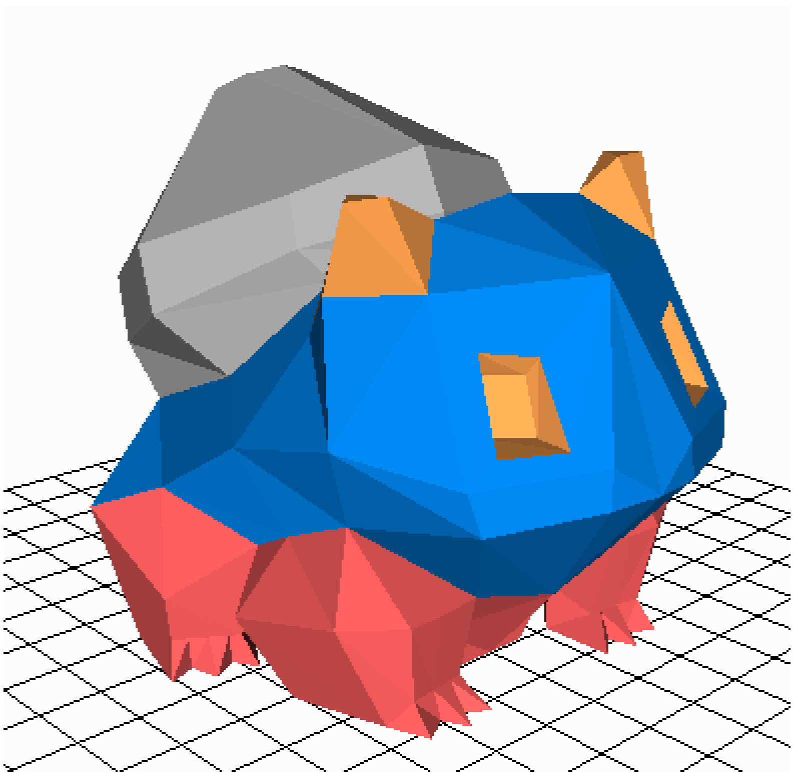


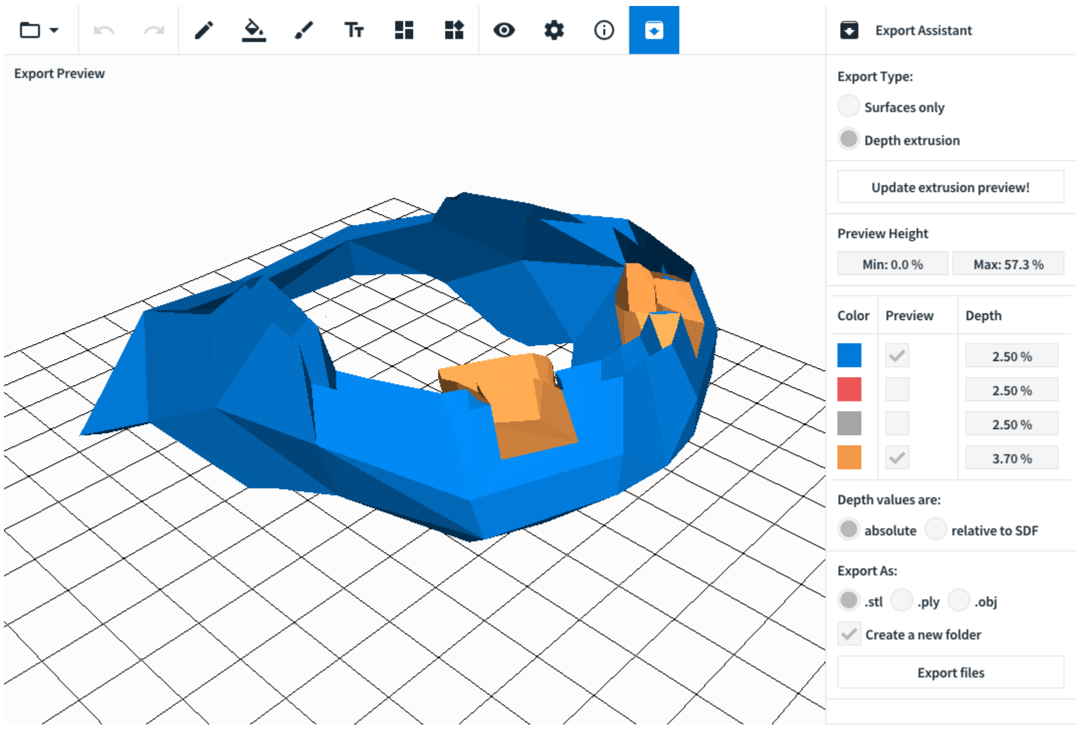
Figure 12.3: Our demonstration colouring, achieved by using the Triangle and Bucket painters.

12.3 Colouring the model

Once the model gets loaded, the user is free to select any available tools and color the model as he wishes. We have opted for a quick colouring of triangles, with all four colors. Our result is showcased in Figure 12.3.

12.4 Exporting the model

After we are happy with our colouring, we go to the **Export Assistant**. This is done by either navigating the menu *File – Export* or clicking the icon from the toolbar. We are now presented with the user interface seen in Figure 12.4. There is a plethora of options here and all the options are described in detail in chapter 17. For our simple model, we can leave the options to their default values, since 2.5% is a good extrusion throughout the whole print. We also select the checkbox to create a new folder for the exported files. After that, we select the volumetric export (*depth extrusion*) and select the .STL format for our files. A new folder is created, containing four different .STL files. We can now also save the project as the Pepr3D project file – **.p3d**, in case we want to alter our colouring later. We include this coloured model in our attachments.


Export Preview

Export Assistant

Export Type:

Surfaces only [✓ Depth extrusion]

Update extrusion preview!

Preview Height

Min: 0.0 % | Max: 57.3 %

Color	Preview	Depth
🟦	✓	2.50 %
🟥		2.50 %
⬜		2.50 %
🟧	✓	3.70 %

Depth values are: [✓ absolute] relative to SDF

Export As: [✓ .stl] .ply .obj

✓ Create a new folder

Export files

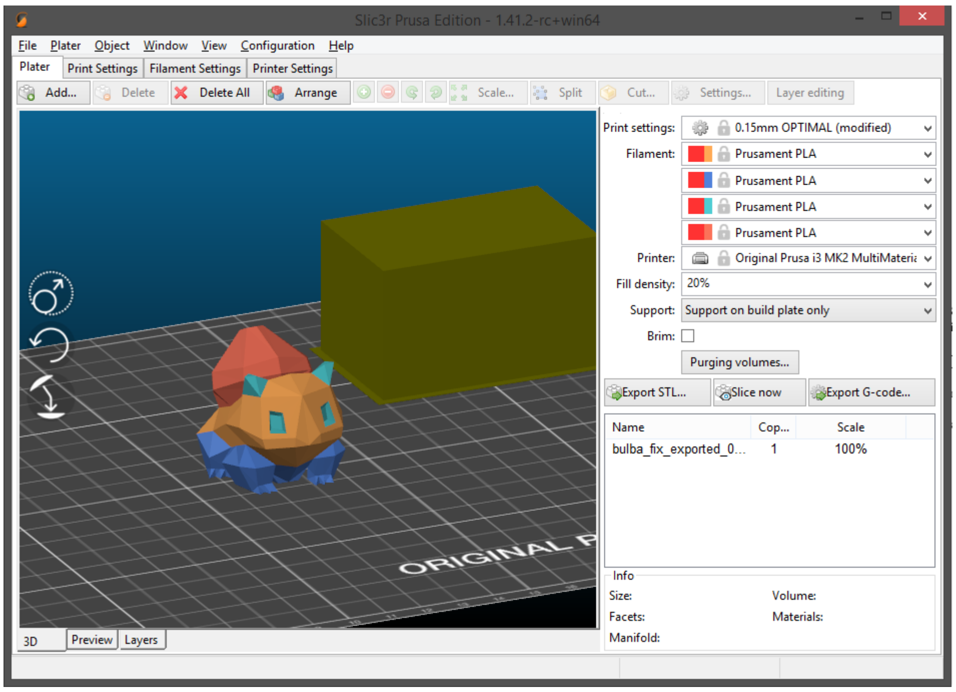
Figure 12.4: The demonstration of the GUI of Export Assistant. Here we select the parameters mentioned in the text and can preview all our export options.

12.5 Putting the files together in Slic3r

In this section, we showcase how the parts we exported in the previous section look in the Slic3r application. In Figure 12.5, the model is already loaded into Slic3r. This was done by loading one of the exported .STL files and then adding parts to it, in the *Settings* menu of the object in Slic3r. We can now slice the model, and prove that the Pepr3D export worked correctly.

12.6 Printing the result

After we are happy with the slicing we got, as shown in Figure 12.6, we can print the model. We include a picture finished print of the model in Figure 12.7, as well as a compilation of other coloured models we printed with our application in Figure 12.8.



Print settings: 0.15mm OPTIMAL (modified)
Filament: Prusament PLA
Prusament PLA
Prusament PLA
Prusament PLA
Printer: Original Prusa i3 MK2 MultiMateri
Fill density: 20%
Support: Support on build plate only
Brim: ☐

Name	Cop...	Scale
bulba_fix_exported_0...	1	100%

Info
Size:
Volume:
Facets:
Materials:
Manifold:

Figure 12.5: The parts exported from Pepr3D loaded correctly into the Slic3r software.

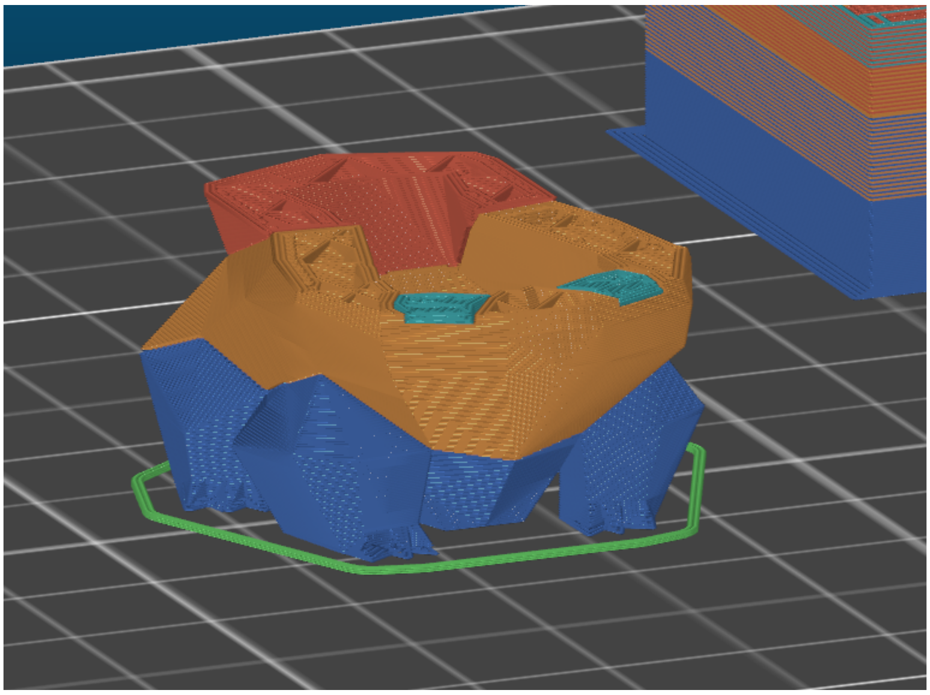


Figure 12.6: The sliced, multimaterial model, ready to be printed.

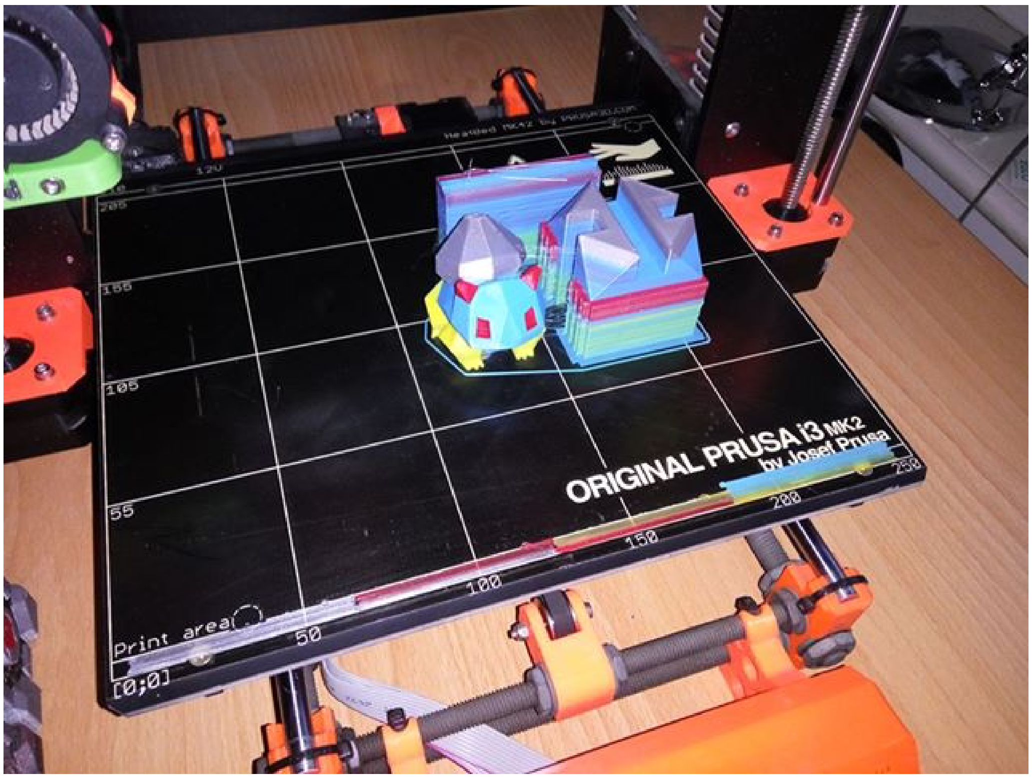


Figure 12.7: The printed model, with a custom wipe tower next to it.

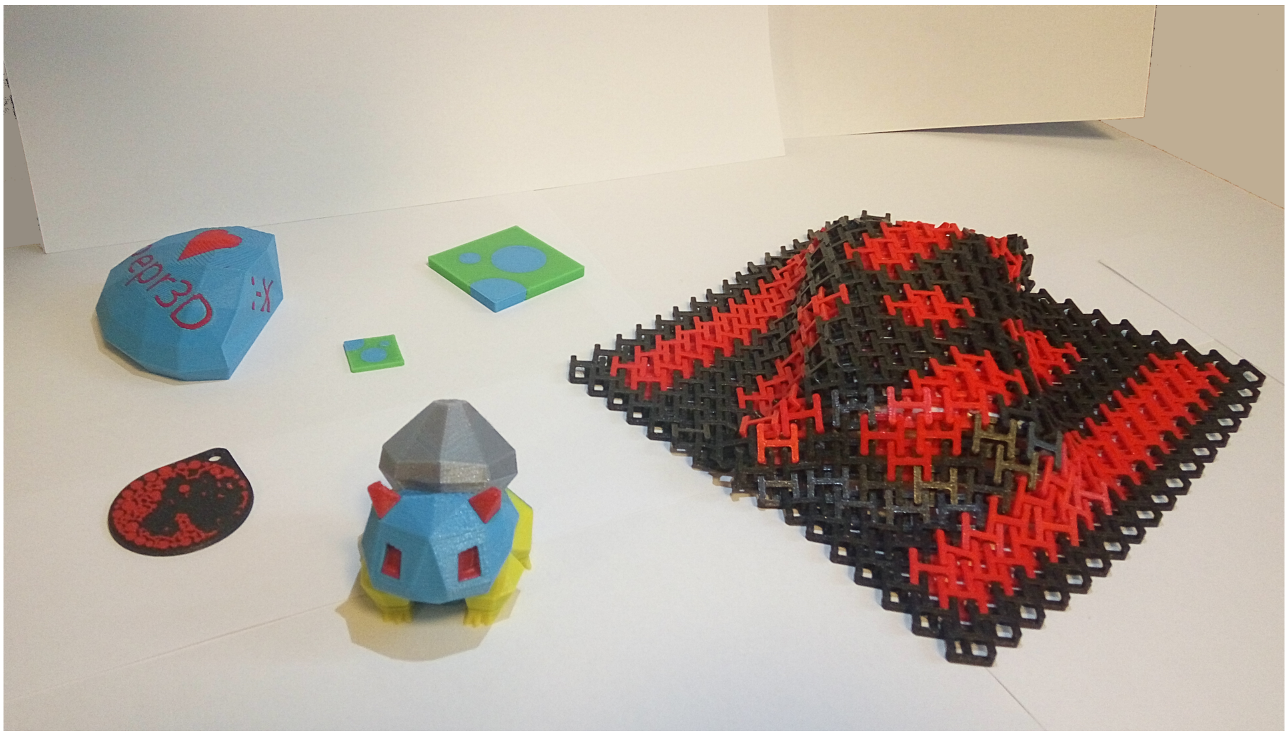


Figure 12.8: Other printed models, all painted in Pepr3D.

Chapter 13

Conclusion

In this chapter, we summarize the project, outline our experience with the 3rd party libraries that we have used, and elaborate on future work that can be done on the project.

13.1 Summary

In our software project, we aimed to design and fully implement an intuitive application for interactive colouring and exporting 3D models for 3D printing. Within this documentation, we described the whole process behind this work together with our results.

We explained the basics of 3D printing and related works in Part I. In Part II, we continued with a developer documentation describing our architecture and how we decided to implement the software. Finally, here, in Part III, and especially in Chapters 11 and 12, we prove that we have successfully fulfilled the goals and requirements initially set in our specification.

The whole application has been successfully verified together with our supervisor Oskár Elek and with our consultants Vojtěch Bubník from Prusa Research, Jaroslav Křivánek, and Tobias Rittig. Possible future work that can be done together with Prusa Research is discussed later in this chapter.

13.2 3rd party libraries

Now we provide a quick summary of our experiences with the 3rd party libraries we decided to use.

13.2.1 Cinder

Cinder [1] is a C++ library which serves as a wrapper around OpenGL. It provides a multi-platform solution to creating an OpenGL window, handling keyboard and mouse input, simplifies the OpenGL buffer handling and much more. It is written with a modern C++11 standard and seemed like a good fit for our project.

[1] https://libcinder.org/

Our experiences were mixed. On one hand, the library performed all the tasks we required and allowed us to spend little time worrying about Linux compatibility. On the other hand, the library itself forced a few very non-practical decisions on us, like the already mentioned C macro in the `main.cpp` file, which we discussed in Section 7.2.

We also suspect, that the Cinder library is the main culprit behind our rather long compile times, since the problem existed basically from the beginning of the development, when the project did not include so much code.

13.2.2 Dear ImGui

Dear ImGui [2] is a graphical user interface library for C++. We already discussed our reasons for choosing this library in the Section 7.1.2.

Our experience with this library has been very positive, as we have expected. The library is simple to use and simple to pick up. One of our team members was able to quickly start working with the library within a few days, without any prior knowledge. The limitation of this graphical user interface is the limited support for skinning, though this feature is not important at this time. It is difficult to use for an entirely custom user interface though, as some things are not yet exposed in the API, such as more advanced column layouts.

13.2.3 CGAL

The Computational Geometry Algorithms Library (CGAL) [3] is the main library we chose to solve the geometry computations for this project, since it included several useful features, which we described in more detail in Chapter 4.

The team's experience with this library is conflicting. On one hand, the library performed everything we hoped for very well, did not raise many issues and we did not find any bugs, weird or wrong behaviour. On the other hand, the library is so heavily templated, that sometimes it is very hard to navigate. This problem is furthermore highlighted by its documentation, which is lacking in several places (with phrases like `Advanced feature.` as the only explanation to a public method). Since it looks to be generated from the code, the code does not provide any more information.

The members who have worked with the library were not satisfied with the library mainly for the user-friendliness, however, it is important to state that once you figure out the API and the general ideas behind the library, it performs well.

13.2.4 Assimp

Assimp [4] is a library that handles importing and exporting of the models. This library holds a unique space in the C++ libraries for geometry loading and saving, because it is basically the only one which supports so many different formats for both importing and exporting. This makes it almost a must-include in a C++ geometry project and we hoped for an easy and fast integration.

[2]https://github.com/ocornut/imgui
[3]https://www.cgal.org/
[4]http://www.assimp.org/

In reality, we have had the most issues with Assimp out of all the libraries used. While the documentation is not plainly generated from code and explains a lot of concepts and ideas, it is not complete and the only thing left is reading the internal source code (not just the header files providing the API, the actual implementation as well). This happened several times during our development (while implementing post-processing during the import, while trying to use Assimp's progress reporters). Exporting a custom built scene (that did not get loaded by Assimp earlier), is only explained in the *Issues* tab on Assimp's GitHub page, which also further highlights the lacking documentation.

We also encountered weird behaviour in the post-processing during the import phase – we set Assimp to completely remove all degenerate triangles (triangles with an area equal to zero), which is described at length in Assimp's documentation. We found out that this pre-process, while configured exactly as the documentation stated, did not, in fact, remove all degenerate triangles, and we had to implement one additional check, after Assimp finishes.

In conclusion, while this library is the best on the market right now, it still has long ways to go, at least in our experience.

13.2.5 Cereal

Cereal [5] is a header-only library for (de)-serialization. The library has a minimalistic API and a solid documentation which explains all the major concepts behind it.

In our experience, it was very easy to pick up and add to our project and worked really well. So far we have not encountered any issues or found any bugs within the library. We have, however, discovered one limitation which the library imposes on the code it is used on. Any object which is to be loaded from serialized data either has to have a default constructor, or be stored by a pointer. If you have an object stored by value and it is not default constructible, the library will not know how to load it.

13.2.6 Threadpool

Threadpool library [6] is a *very simple* C++ library providing a simple threadpool. We used this library since we required a basic threadpool without too many features or overhead. This free code is simple, easy to check and functional.

13.2.7 FreeType, FTGL and Poly2Tri and Font23D

While doing research for the *Text* tool on how to take a font file and a text string, and transform the bezier curves into triangle meshes, we found the **Font23D** library/project.

Font23D is a library/project on GitHub [7] without too much activity, but solving exactly the issue we faced as well. It incorporates the **Freetype**, **FTGL** and **Poly2Tri** libraries to solve the issue and we used it parts of this project in

[5]https://uscilab.github.io/cereal/

[6]https://github.com/progschj/ThreadPool

[7]https://github.com/codetiger/Font23D

our own. You can read the exact development discussion about this library in Section 4.4.

We made severe improvements and adjustments to the code from the repository, as the code is mainly written in **C**, instead of C++.

13.2.8 Boost

Last but not least is the Boost library which came as a pre-requisite for Cereal. Since we already had this library in the project, we decided we might as well use it. In the end, we did not use it for any major features but it was still handy to have around.

We will not discuss the quality of the documentation or the performance of this library, since it is a staple in the C++ environment.

13.3 Future work

In this section, we discuss the future work that could be done on this project. We divide the improvements that could be implemented into several categories:

- Improving existing core features
- Adding *quality of life* (QoL) changes to the GUI
- Extending the toolset of the application

13.3.1 Improving existing core features

A few of Pepr3D's features and algorithms were developed by the team from scratch, since no solution satisfying our needs existed. These features are mainly the **Brush tool** and the **volumetric Export**.

The brush tool uses computational geometry to subdivide triangles on the fly, which is not an easy task. Further work could be done by optimizing the brush tool to create better subdivisions and increasing the speed of the tool on bigger and more complex models. Our finished product is the best the team was able to come up with but with some more research, the tool can probably be optimized further.

The volumetric export (meaning the export which extrudes the faces inwards) is also a very complicated task, for which we have not found many solutions in any academic research or commercial products. We think that making this feature more robust would greatly improve the Pepr3D user experience.

13.3.2 New quality of life features

Since Pepr3D is a user-targeted application, the range of features the users have come to expect from the GUI of the program is vast. We implemented the basic subset of, what we think, are the most useful and important features – such as hotkeys, tooltips and clear and simple user interface. However, there are many more features the users might benefit from, for example the radial menu around the mouse cursor, which we already discussed in Section 11.2.

Other quality of life feature we got asked about by our colleagues during the development was a *branching Undo & Redo history*. This means that the command history would not be linear, but the user could go back a few commands from version B to version A, make new changes to version A, which would take him to version C. He could then compare versions B and C, which are both based on A and decide which he likes better.

The export GUI could additionally benefit from a tighter integration with 3D printing slicers. They could show the user in real time how the exported segments will look like layer by layer after being sliced for 3D printing. Our current visualization is not as advanced as we do not have the necessary data and algorithms for actually slicing the objects. This would make exporting the objects faster as users would not need to run another application.

13.3.3 Extending the toolset

When we designed the application's architecture, we put strong emphasis on allowing a potential developer to extend the toolset by adding other tools. We think we achieved this goal very well, because several of the tools require the same Geometry and Command API, which means we could add the tools and extend the functionality without implementing any additional functionality into Geometry or adding new Commands. This is the intended behaviour for the potential future developers.

If the new tool should require extending either the Geometry or Commands API, we strived to make the code educational – if you need to create another command, you can read through one or two existing commands and then have a good understanding of how you should create your own.

Appendix I: User Documentation

Chapter 14

System requirements and Installation

In this chapter we will describe the system requirements of Pepr3D and the installation process.

14.1 System requirements

We divide the system requirements into **must have** items and recommended ones. The **must have** requirements are the following:

1. a 64-bit CPU with SSE instructions
2. a GPU supporting OpenGL version 3.2

These two requirements are mandatory and Pepr3D might not work if you do not meet one or both of these.

Now we mention recommended system parameters. These are derived from what the team has been developing the software on, since we do not have an access to any larger data.

- **System:** Windows 8 / 10 (64-bit)
- **Processor:** Dual core Intel CPU with clock speed 2.0 GHz or higher and 64-bit and SSE instructions
- **Memory:** 2 GB or more
- **GPU card:** GPU card compatible with OpenGL 3.2
- **Storage:** 200 MB

14.2 Installation

Installing Pepr3D is very easy. If you use the attached CD, you can run the executable files directly (see Appendix II). Otherwise, a compressed archive can be downloaded[1] and unpacked into a folder anywhere on your hard drive. Pepr3D should now be ready to run.

[1]https://github.com/tomasiser/pepr3d/releases

Chapter 15

First run

In this chapter we show the usage of Pepr3D for complete beginners. It covers every step from starting Pepr3D to exporting a simple colored model including importing, manipulating and using tools.

15.1 First look at Pepr3D

When you run Pepr3D, you will see a cube at the center of the application. There is a toolbar at the top of application which contains file menu, undo/redo buttons, set of tools and some settings. There is also a side pane on the right with settings of individual tools as you can see in figure 15.1.

15.1.1 Model manipulation

You can manipulate the model by using your mouse. There are several ways to manipulate so you can reach and see any part of the model:

- **Rotation** – Click and hold right mouse button and move.
- **Translation** – Press Ctrl + right mouse button and move, or press and hold the middle mouse button (mouse wheel) and move the mouse.
- **Zoom** – Scroll with the mouse wheel.

Left mouse button is dedicated to using selected tool.

15.2 First model

Now we can start working on a simple model with Pepr3D. First we have to acquire a 3D model, it should be in one of these file formats: `.stl`, `.ply`, `.obj`. The simplest way to acquire model is choose any model on the internet and download it. Or you can use any 3D modelling software and create one on your own. In this tutorial we use a simple low-polygon model of a Bulbasaur downloaded from Thingiverse[1].

To import the model we can use a drag and drop gesture with the model file or we can browse for a model file after clicking *Import* in the file menu.

[1]https://www.thingiverse.com/thing:327753

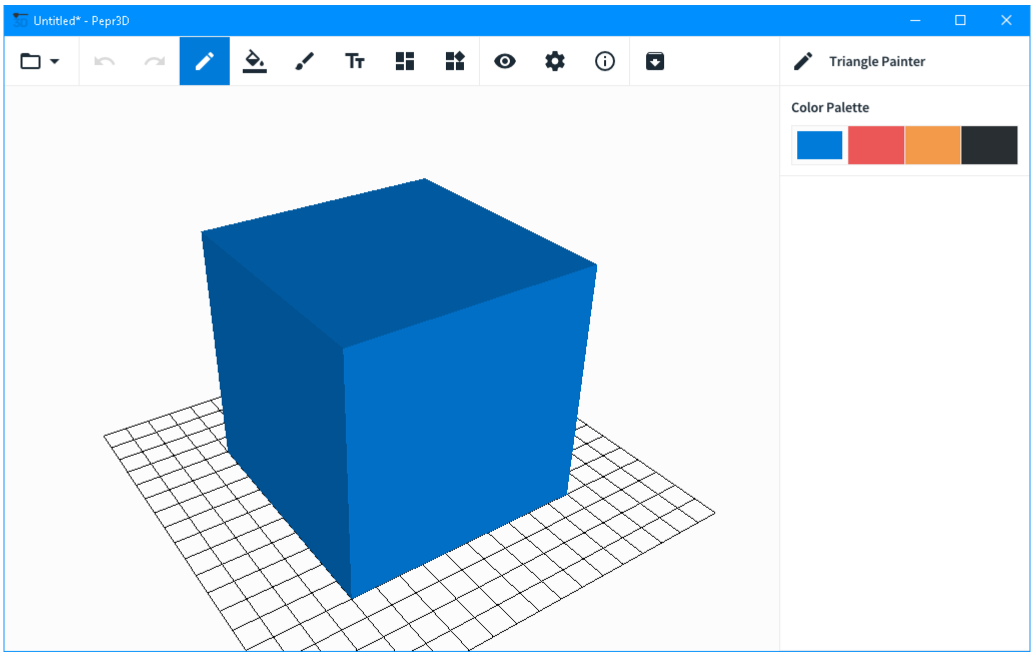
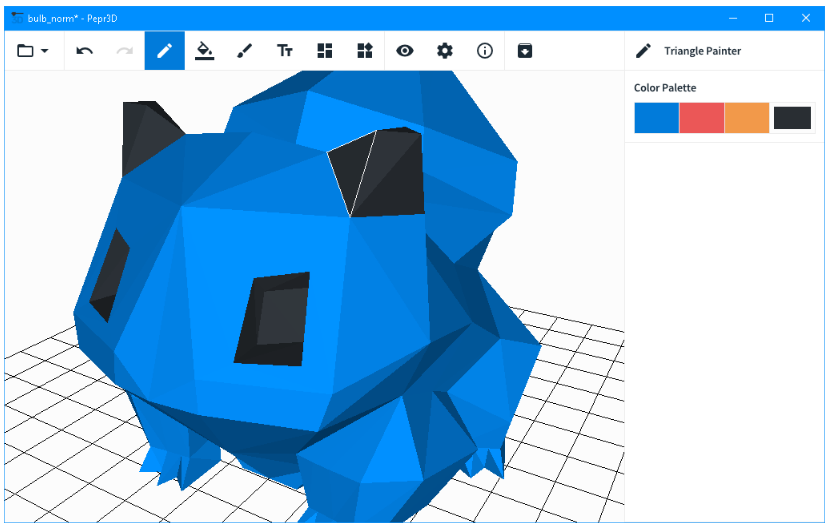


Figure 15.1: Pepr3D appearance after start-up.

15.2.1 Painting the model

After importing the model we can use any tool that our application provides to color the model as we want. In a few steps, we will show how to quickly color the imported model of our Bulbasaur with basic tools.

1. Select the *Triangle Painter* tool, choose black color in the color palette.
2. Paint all triangles in each eye by clicking on them with the left mouse button. It is possible to click and drag to paint multiple adjacent triangles at once.
3. Use the same technique to paint its ears.

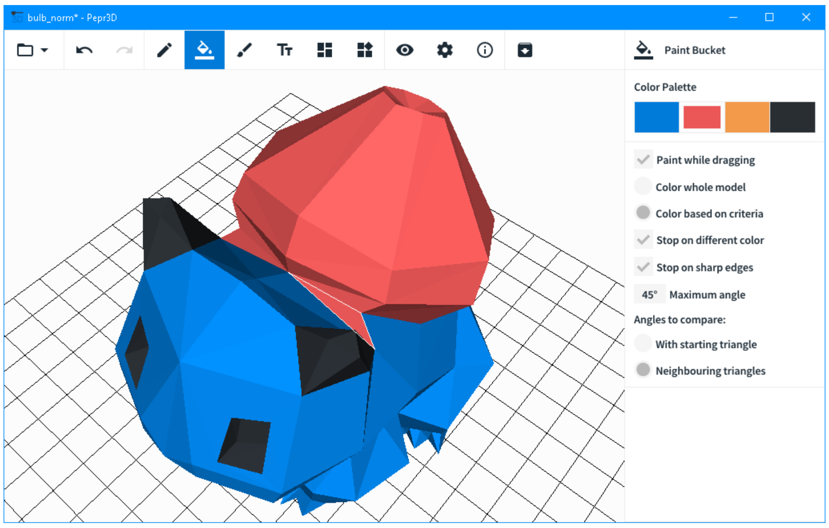


4. Choose another color (red) and select the *Paint Bucket* tool.

5. Check *Stop on sharp edges* in the side pane and set the *Maximum angle* to 45°.

6. Use the *Paint Bucket* on any triangle on the ”onion” on the back of the Bulbasaur. Click two more times on any unpainted triangle to paint the whole ”onion” with red color.

7. Select the *Triangle Painter* and the first color (blue) and recolor two triangles near the neck which have been painted extra by the *Paint Bucket* in the previous step.

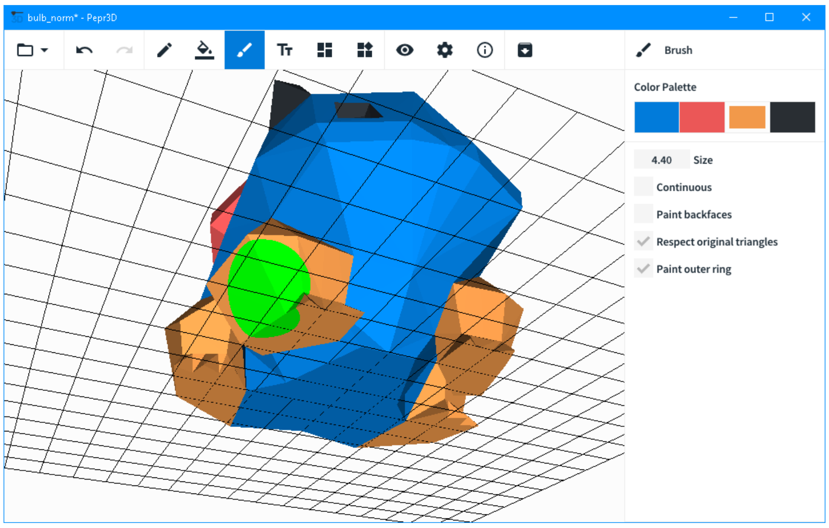


✓ Paint while dragging
Color whole model
● Color based on criteria
✓ Stop on different color
✓ Stop on sharp edges
45° Maximum angle
Angles to compare:
With starting triangle
● Neighbouring triangles

8. Select the *Brush* tool and check both the *Respect original triangles* checkbox and the *Paint outer ring* checkbox in the settings of the tool.

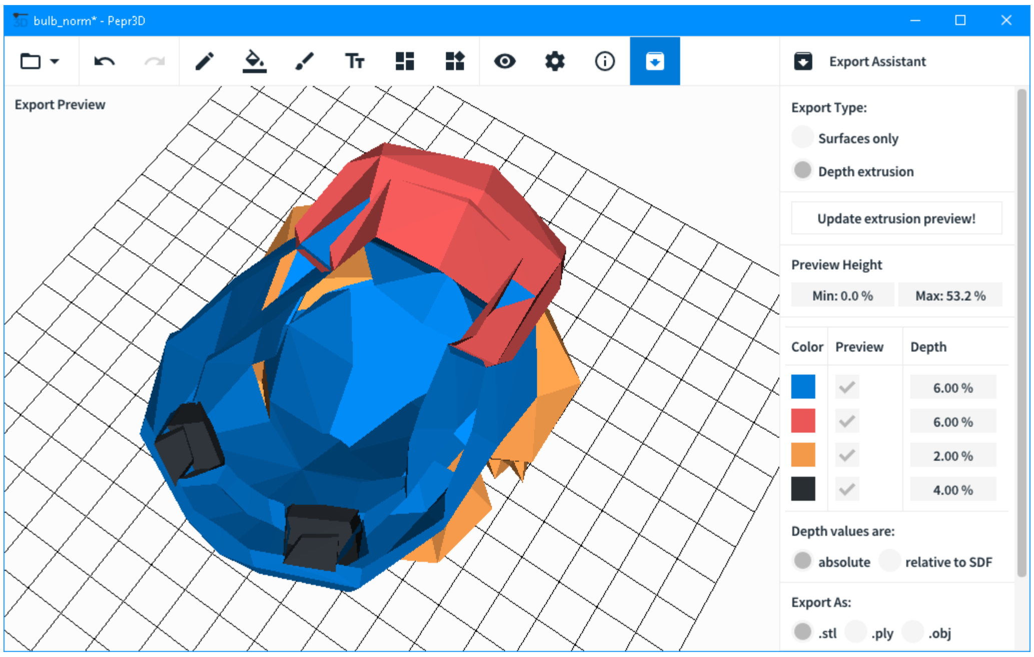
9. Set the brush size to about 4.0.

10. Choose orange and paint each leg. Do not forget to paint the legs from below.



4.40 Size
Continuous
Paint backfaces
✓ Respect original triangles
✓ Paint outer ring

You can undo any step you did with any tool. For example, if you paint on a incorrect triangle, you can press *Undo* (left arrow) in the toolbar to revert the mistake.



Export Type:
Surfaces only [✓ Depth extrusion]

Preview Height
Min: 0.0 % Max: 53.2 %

Color	Preview	Depth
🟦	✓	6.00 %
🟥	✓	6.00 %
🟧	✓	2.00 %
⬛	✓	4.00 %

Depth values are:
[✓ absolute] relative to SDF

Export As:
[✓ .stl] .ply .obj

Figure 15.2: Example of *Export Assistant* with colored model.

15.2.2 Exporting the model

Now the model is painted and we can proceed to model exporting. Before exporting the model itself we need to set the depth of color extrusion into the model. Exporting can be summarized in the following steps:

1. Open the *Export Assistant* on the toolbar or click *Export* in the file menu.
2. Click the *Update extrusion preview!* button.
3. Lower the percentage of *Max Preview height* to see into the model and see the thickness of model walls – the extrusion depth.
4. Adjust the percentage of *Depth* for the desired extrusion depth.
5. Update the preview by clicking on the *Update extrusion preview!* button.
6. Repeat adjusting the depth and updating the preview until you are satisfied.
7. Click on *Export files* and complete the export.

Exported files can be now imported into any supported slicer and printed on a multimaterial 3D printer.

Chapter 16

Tools

This chapter covers all the tools the user has at his disposal. We explain each tool's purpose and all the parameters the user can set.

16.1 Triangle Painter

Triangle Painter is the simplest tool of Pepr3D. It allows the user to color a single triangle with a selected color. This can be performed either by a single click on the model's triangle or by dragging the mouse over several triangles with the left mouse button pressed down.

The triangle that is currently hovered (has the mouse cursor over it) will be highlighted on the model's surface with a different border color.

The only property the user is able to select in this tool is the current color from the color palette.

Pressing the *Undo* will undo the last stroke of the triangle painter. This means it will undo **the whole** stroke, if the user dragged the mouse over several triangles.

16.2 Bucket Painter

Bucket Painter is a simple tool that can be used to achieve sophisticated results easily. This tool works as one is used to from image editing software like GIMP [1] or Adobe Photoshop [2] – it starts colouring every triangle it can reach, starting with the triangle the user clicked on.

16.2.1 Properties

The properties of this tool revolve around the spread of the bucket. This is something we call *stopping criteria.* We now list all properties of the tool and explain each one in detail.

- **Color Palette** – This widget allows the user to select the current active color. The selected color will be spread by the bucket. Customizing the palette can be performed in the *Settings* panel.

[1]https://www.gimp.org/
[2]https://www.adobe.com/products/photoshop.html

- **Paint while dragging** – *On / Off* – This checkbox specifies whether the bucket painter will only function by clicking on single triangles (*Off*) or will bucket spread continuously if the user drags the mouse in a stroke (*On*). We recommend leaving this *On* unless it disrupts you or the model you are working on is very big.
- **Color whole model** – *On / Off* – We have mentioned *stopping criteria* in the beginning. This is the first choice the user can make that affects the stop of the bucket spread. If the user selects this option, the Bucket Painter will simply color the whole region of the model. If the model is a single mesh, it will color the whole model. Turning this *On* will hide the other options. Turning this *Off* allows the user to specify the *stopping criteria.*
- **Stop on different color** – *On / Off* – The simplest *stopping criterion.* The spread will only re-paint triangles which have the same color as the triangle the user clicked on. Additionally, the spread will stop if a new color is met. If this is the only criterion that is enabled, the Bucket Painter will work exactly as we are used to from image editors. This is the default setting of the tool.
- **Stop on sharp edges** – *On / Off* – A second *stopping criterion* which can be enabled or disabled. Enabling it expands the user interface to allow the user to modify the criterion. This criterion will not care about the color the user clicked on, and only stops from spreading to the neighbouring triangle, if the neighbouring triangle is at a greater angle than specified. The exact behaviour is specified by the following properties.
- **Maximum angle** – 0°–180° – Specifies the angle which the two neighbouring triangles have to be angled at for the bucket painter to stop spreading. If the angle between the two triangles is greater than this value, the spread will not color the triangle and will stop.
- **Angles to compare** – *With starting triangle / Neighbouring triangles* – The last choice in the sharp edges *stopping criterion.* If the user selects *With starting triangle*, the angle will be measured between the triangle the user clicked on and the triangle currently being coloured. For example, if this option is chosen, the angle is set to 95° and a single face of the cube is clicked, all faces of the cube except the opposite one are coloured. This is because the opposite face is at an 180° angle. If the user selects *Neighbouring triangles* and uses the 95° setting again, the whole cube will get coloured, because there are no faces on the cube that are at an angle greater than 95°.

Both of the *stopping criteria* can be selected together. The spread stops when one of the criteria is not fulfilled – both of the criteria must be fulfilled for the spread to continue.

16.3 Brush

Brush tool is more complicated tool for coloring the model. User can draw strokes with mouse depending on a brush properties. Just like the *Bucket Painter*, this tool works as a brush in ordinary image editing software when you are drawing on flat side of model. It has different behavior on the edges which depends on set properties.

16.3.1 Properties

There is a list of properties that user can change to customize the tool.

- **Brush Size** – *float number* – Defines the size of the brush. Larger size means larger brush diameter so that the user can paint larger area at once using the brush. The size number is equal to world units.
- **Number of Segments** – *positive integer* – This setting adjust the shape of the brush. The shape is a polygon and the parameter corresponds to a number of polygon sides. If it is high enough, the shape of the brush looks like a circle. The smallest number can be set to 3, it means that the brush will have triangular shape.
- **Paint backfaces** – *On / Off* – If the property is set off the brush will paint only visible faces that direct towards the user. If it is set on, it will paints parts of the model even if they are facing away from the camera. The entire area that is within the scope of the tool will be painted.
- **Spherical brush shape** – Turns on spherical brush (this is set by default). Spherical brush paints everything within a radius from a mouse cursor, and will create additional edges to smooth transition over triangle boundaries.
  - **Continuous painting** – *On / Off* – With this option turned on user can paint only triangles that are connected inside the painting radius. This prevents accidentally painting two parts connected only via an air-gap.
  - **Respect original triangles** – *On / Off* – Turning this feature on prevents brush from creating new triangles. This makes *Brush* tool behave like a *Triangle Painter* tool with a radius.
  - **Paint outer ring** – *On / Off* – This can be set on only if *Respect original triangles* option is set on. It allows the user to paint the whole original triangle even if it is not fully inside the brush.
- *Flat brush shape* – Turns on flat brush shape. Flat brush paints the shape directly to the surface, ignoring any distance limitation. With the **Paint backfaces** turned on it will paint the whole cylinder through the model.
  - **Flat brush settings** – *Perspective / Normal* – With *Perspective* option the brush paints from the direction of the camera. If *Normal* option is set, it will paint area aligned against triangle normal.

16.4 Text Editor

Text editor tool allows user to write some text onto the model. User can preset properties of the text and place the text on the model by left mouse button click on a specific location on the model. After that a floating text appears beside this location. Pepr3D also shows a normal vector in the center of the floating text that determines the direction in which the text will be projected onto the model. After showing the floating text, user can still adjust the tool properties to improve the appearance.

16.4.1 Properties

The properties that user can adjust are following.

- **Load new font** – *button* – This button opens a file dialog to select and import user's own text font from the computer in `.ttf` format. However, text tool does not support complicated fancy fonts such as ornamental or picture fonts.
- **Font size** – *10–200* – By adjusting this property user can change size of the text. It is base font size in font-units.
- **Bezier steps** – *1–8* – This parameter specifies how much the edges of the letter of the text will be smooth. Higher number of *Bezier steps* will increase painting time.
- **Text** – *text field* – In this text field user can type custom texts that he or she want to paint on the model. The preview floating text will be changing during typing into this text field.
- **Text scale** – *0.01–1.0* – Another way to change size of the font is adjusting this parameter. It allows user to make really big or small texts on models.
- **Text rotation** – 0°–180° – This parameter allows user to rotate the text around the normal vector. With the default value (0°) the text lays horizontally, no matter what angle the normal vector has.
- **Paint** – *button* – Finally, the last option paints the prepared floating text onto the model. The text is projected along the normal vector. The computation may take some time.

16.5 Automatic Segmentation

Automatic segmentation is a powerful tool which allows the user to quickly achieve the baseline colouring of the model, which then can be detailed to the user's liking. This is achieved by separating the model into several segments based on the thickness. These segments then can be quickly coloured individually. The user can select the sensitivity of this segmentation, which allows him to control the level of detail (for example, low sensitivity might only segment the body and limbs of a character, whilst high sensitivity will also segment the fingers, ears and horns).

16.5.1 Properties

- **Compute SDF values** – *button* – Before anything can be done in this tool, the user is asked to compute the SDF values. This is the data that is required to perform a successful segmentation. This computation might take a long time to perform, depending on your model size. For low-poly models (e.g. 1000 triangles), this computation is instantaneous. If you already performed the computation in a different tool (like export or the other segmentation), this option will not be visible.
- **Segment!** – *button* – This button starts the segmentation process. The default values are set so the segmentation returns viable results in most cases. If you did not set any of the following properties and the segmentation returned an undesirable number of segments (like only one or too many), modify the following properties.
- **Robustness** – 0%–100% – a magic parameter. The meaning of this parameter is somewhat obscured but the best way to imagine this setting is the quality or robustness of the segmentation. Higher values take longer to compute but might give better results – a higher value might merge two segments that are somewhat related, which the lower values will not recognize. The default setting is the team's best effort to balance the performance and quality.
- **Edge tolerance** – 0%–100% – This parameter specifies how the algorithm should understand ”thickness”. If you set this value very high, the algorithm will tend to merge more segments together, resulting in a lower amount of segments. If you set this very low, every nook and crease will signal the algorithm to create a new segment, thus resulting in a higher amount of segments. This is the **primary means** to control this tool and we recommend adjusting this slider over the previous one to change the main behaviour.
- **Color Palette** – Same as in the previous tools, this widget allows the user to select a color to assign to each segment.

16.5.2 Segmentation

To adjust the segmentation, we recommend first trying to adjust the *Edge tolerance* slider, and only after experimenting with this slider to change the *Robustness.*

After these settings are adjusted and the user clicks on the *Segmentation* button, a list of segments will appear, along with the number of segments created. The user is then instructed to assign a color from the color palette to each segment. This can be done in two ways – either by clicking directly on the model or by clicking on the ”Segment #” button. After clicking on one of these two regions, the color selected in to color palette will get assigned to the segment.

After **all segments** have been assigned a color from the color palette, the user is able to click *Accept* to color the model this way. Should the user be dissatisfied with the colouring, he can either click *Cancel* or *Segment!* again, to completely undo the whole segmentation and start from scratch.

16.6 Manual Segmentation

Manual segmentation is a similar tool to the Automatic segmentation we discussed in the last section. The difference in these two tools is that while Automatic segmentation is a very global tool (since it segments the whole model at once), Manual segmentation is designed to be local. The user can color a handful of triangles with a single color and then manually adjust the spread of this color over the segment the triangles define. This description is rather abstract, but hopefully it will get clearer once we discuss the properties.

16.6.1 Properties

- **Compute SDF values** – *button* – before anything can be done in this tool, the user is asked to compute the SDF values. This is the data that is required to perform a successful segmentation. This computation might take a long time to perform, depending on your model size. For low-poly models (e.g. 1000 triangles), this computation is instantaneous. If you already performed the computation in a different tool (like export or the other segmentation), this option will not be visible.

- **Color palette** – once SDF values of the object have been computed, the user is presented with a sidepane very similar to the Triangle painter tool. This widget allows the user to select the current color. The color will be used while initializing the segments on the model.

- **Spread** – 0%–100% – once a single triangle is coloured on the model, additional options appear. One of them is the *Spread* slider. This slider is analogous to the *Edge tolerance* slider in Automatic segmentation, since it controls how much each coloured triangle will spread its color among its local neighbourhood. If the spread is 0%, only the triangle is coloured. Increasing it to 100% will color all triangles of the model (unless a second segment is competing).

- **Hard edges** – *On / Off* – if this option is turned on, the spread of one color will stop upon meeting a second one's border and will not attempt to color any other triangles. Use this option if the segments you are colouring are well defined and differ in thickness a lot.

- **Region overlap** – *On / Off* – once the user turns this option on, the spreading regions will overlap freely and the last color (rightmost) will always win if *Spread* is turned to 100%. Use this option if the borders of the segments converged a little soon or late and you would like one color to expand a little more. This option is not good if the spread is turned to high percentages, since a lot of the segments will overlap and information will be lost.

16.6.2 Segmentation behaviour

If no checkboxes are turned on, the color spreads to the global optimum segmentation. This means that if you set the *Spread* to a high percentage, there might be several discontinuous segments coloured by the same color, because that is globally optimal to the input. This setting is a good starting point, since it does not restrict the spread too much, but does not allow it to roam unlimited as well. Use the other options to tweak the spread after you understand how this model will get segmented.

The spreading algorithm uses the SDF function (thickness of the model) to calculate segments. This means that this tool is not able to grow segments on a model which does not vary in thickness. It is also very important to be aware of the fact that several non-neighbouring parts of the model might have the same thickness (like ears, hand fingers and feet fingers) and thus the global optimum colors all of these with the same color (use *Hard edges* to counter this behaviour).

After the user is satisfied with the spread, clicking *Apply* will confirm this re-colouring. Clicking cancel will return the model into the state before Manual segmentation started.

As a last tip, we recommend using a single-triangle strokes to initialize the segments at the beginning. This is using many triangles accelerates the spread a lot, which ultimately gives the user less fidelity and time to see what is happening.

We want to stress that this tool is an advanced tool, which should be used to fine-tune an already coloured model's details, not as a means to primarily color the model upon importing it (Automatic segmentation is a lot better tool in that case).

Chapter 17

Import, Export and Saved projects

In this chapter we explain in-depth how the users should import their models, all the different methods of exporting their work and the ability to save their work as a *Pepr3D project* to continue at a later date.

17.1 Importing a model

Importing the model is the first step in the Pepr3D workflow. There are several ways how to import a model and all of them are equally easy and the choice is entirely up to you.

1. **File → Import** will open a typical *Open* dialog of you respective operating system. Navigate to the model you want to import and click the button *Open*.
2. **Drag and drop** is a very fast way to import the model if you already have it located in any file explorer. The model can be dropped into any part of Pepr3D.
3. **Control + I** is the keyboard shortcut for importing a model. Upon pressing this shortcut, the *Open* dialog of you respective operating system will open. Navigate to the model you want to import and click the button *Open*.

After you perform either of the two previous steps, the model will start loading. There is a detailed dialog which explains what is currently happening and Pepr3D is trying to give you accurate information about the progress of the process. However not every computation has a well known length, so several loading bars will just cycle through until the loading is complete. Please be patient, loading a large model can take a long time. See Table 17.1 for a rough idea about the loading times.

17.2 Exporting a colored model

When you finish painting a model in Pepr3D and you would like to use it in 3D printing or another application, it is necessary to *export* the colored model.

File size [MB]	**Estimated loading time [s]**
80	3
15	2
5	1
1.2	0.5
<1	<0.5

Table 17.1: Loading times during the import into Pepr3D.

This saves your work to files which are compatible with other software. To be more precise, 3D printing a model with different colors typically requires separate colors in separate files, which exporting in Pepr3D does.

To export a model you have painted in Pepr3D, use our **Export Assistant**. You can access it by clicking on its **icon in the Toolbar**, by using the default hotkey **Ctrl + E**, or from the File menu **File → Export**.

Once you select the Export Assistant, you can change its options in the side pane. On the left side, a preview of the export will be shown, but only for certain options described later.

In the side pane, as the very first decision, you can choose between exporting *only the surfaces*, or exporting with *depth extrusion of the colors*. As these two options influence the rest of the export process, we describe them separately in the following subsections.

17.2.1 Exporting surfaces only

Exporting only surfaces of your painted model is the simplest option. All triangles with the same color are grouped together and saved to a separate file. The result are multiple files containing the differently colored surfaces of the model.

This option is useful when you want to use your painted model in another 3D editor such as Blender. It is not entirely useful for 3D printing as common slicers are not capable of actually printing these exports reasonably. That is because volumetric information, i.e., how deep the colors should be extruded, is missing in this export.

The only options in the side pane are the file formats and then a simple **"Export files"** button which opens a file dialog to save the files. Follow Section 17.2.4 for more details.

17.2.2 Exporting extruded colors

Unlike surface export, extrusion export has various options and also provides you with a preview of the options. The biggest difference is that extrusion export is much more suitable for 3D printing as it provides volumetric information, i.e., how deep the colors should be extruded. The user can specify this extrusion information in the various options.

At any time you can press the **"Update extrusion preview!"** button and a preview of the export will be shown on the left side of the Pepr3D window. The point of the preview is so that you as the user can see how the model is going to look like after being exported. Generally, the preview should look exactly as the model you painted, unless there are errors and the options are wrong.

By changing the range in the **"Preview Height"** option, you can see inside the model. Raising the minimum height removes a bottom part of the model. Lowering the maximum height removes an upper part of the model. This only affects the preview, not the actually exported files!

Being able to see inside the model is very useful in extrusion export, as it helps you understand how deep the colors will actually penetrate. One of the most important things to remember is that the extrusion *should never penetrate* the original surface! If the extruded part penetrates the surface on the opposite side, you have to lower the depth of that color!

This can be done by changing the **"Depth"** in the table of colors. The depth percentage is with regards to the size of the object, where 100% corresponds to the size of the whole object. This means that if you set an extrusion that high, it will almost for sure penetrate the surface and be wrong. We should always operate with lower extrusion depths such as a few percent only.

In models that have SDF (shape diameter function) values available, you can set the depths to be **"relative to SDF"** instead of "absolute". This is very useful for complex models with varying thickness, e.g., models with spikes, little details, etc. Relative depths are not only based on the percentage you set, but they also vary locally with regards to the local thickness of the model. So for a certain color and percentage you set, the depth will be *higher* than your percentage in thick parts of the model and *lower* in thin parts of the model. This is different than in "absolute" thickness where all triangles are extruded the same.

To verify how the extrusion looks inside the model, you may also disable the **"Preview"** of certain colors in the table of colors. This will simply hide all triangles and extrusions of a certain color from the preview, but not from the exported files that you actually export.

The export is finished using the **"Export files"** button (see Section 17.2.4).

17.2.3 Advice for 3D printing

When exporting for 3D printing, try to follow the following recommendations:

- Always use the **"Depth extrusion"** option unless it gives completely wrong results with whatever extrusion options you try. In that case, it might be necessary to use **"Surfaces only"** and do the extrusions manually in a 3D editor such as Blender.

- Make sure the extruded parts never penetrate the object surface from the opposite side. This penetration will be visible in the 3D printing! If that happens, make sure to lower the **"Depth"** of that color. On the other hand, too low depths may be impossible to print as the model would be too hollow. It is necessary to find the right balance.

- For certain slicers such as Slic3r Prusa Edition, the extrusions *may* intersect in the interior of the model. The slicer will fix this automatically when generating the G-code for printing.

- In models with various thickness in different parts, e.g., models with spikes, little details, etc., try to use depths **“relative to SDF”**.
- If parts of a model with a certain color are too deep or too shallow but changing the percentage for the whole color breaks in another part of the model, consider adding a new color to the palette. You can paint the two parts with different colors but still print them with the same color! Remember that in the 3D printing slicers, you can set multiple parts from Pepr3D to be printed with the same color. **The Pepr3D color palette does not have to correspond to the colors you actually print!**
- If you want a part of a model to be filled instead of partially hollow, in certain slicers such as Slic3r Prusa Edition, try to import the original model as one of the extrusion parts. The slicer will then try to fill in the part completely. A similar trick may be achieved by setting color depth to 0.00%, but not all versions of slicers can successfully import these.
- Always verify the extrusion by importing the exported files to a slicer, generating the G-code, and previewing the model layer by layer, if it is supported by the slicer.

17.2.4 Supported formats

When exporting the files, before clicking on the **“Export files”** button which opens a file dialog to save the files, there are options to choose from. First, you can choose from 3 file formats described below. Then, you can also check the **“Create a new folder”** option, which means that a new folder will be created and all the exported files will be saved in the new separate folder (directory). This is useful when using multiple colors and neatly organizing your exports.

The supported file formats are:

- binary **.stl** (stereolithography) files, they are suitable for example for 3D printing with Prusa printers and Slic3r Prusa Edition,
- binary **.ply** (Stanford Triangle Format) files, they are supported by common 3D editors,
- non-binary **.obj** files that are also saved with their corresponding **.mtl** files, also supported by common 3D editors.

17.3 Saving and opening a project

17.3.1 Saving a project

Saving a project to work on it later is very simple in Pepr3D. There are two save options in Pepr3D:

1. **File** → **Save** will overwrite your last save file with the current state of the model. If you have not yet saved the project at all, this option also acts as *Save As*. The keyboard shortcut for *Save* is **Ctrl + S**.

2. **File** → **Save As** will prompt you with a *Save As* dialog of your respective operating system. Upon selecting the folder and choosing the name, the project will be saved inside the folder with the chosen filename. There is no keyboard shortcut for *Save As*.

If your project has been modified since the last save, you will see an asterisk (*) next to the project's name.

Please note that Pepr3D **does not** save your work undo history. If you save a project and re-open it, you **will not** be able to undo any operations done by the previous session.

17.3.2 Opening a project

Opening a project can be done through **File** → **Open** or simply by pressing **Ctrl + O**. Both of these options will display the *Open* dialog of you respective operating system. Here you can choose the **.p3d** file and press open.

Opening a project can also be performed by **drag and drop**. Simply grab your **.p3d** file and drop it anywhere into Pepr3D.

As we have mentioned in the section about saving projects, keep in mind that Pepr3D **does not** save your work undo history.

Chapter 18

Additional options and settings

In this chapter, we showcase and explain all the preferences and settings Pepr3D contains. There are two main categories of the settings the user can alter in Pepr3D: **Display options** and **Settings**. **Display options** contains all settings regarding the displaying of the model and handling the camera. The **Settings** tab contains the important *Color Palette* manager and some extra UI settings. We now discuss each category in more detail.

18.1 Display options

This tab of the interface contains all properties related to the user's view and interaction with the model. These preferences are divided into three segments and we will explain every one in detail.

18.1.1 Camera zoom behavior

This option provides the choice between two most common camera zoom behaviors, namely the **dolly** and **field of view**. Both of these are widely used in computer graphics and there is no clear consent on which of these two is better.

- **Dolly** – as the short text next to the option explains, this setting physically moves the camera within the space of the program, while the field of view stays the same. This option is the default setting, since the zooming in and out is more intuitive because it replicates moving the observer's eye closer to the object. The disadvantage of this method lies in editing fine details on the surface. This is because the **dolly** allows the camera to enter the inside of the object, which means the user is not able to zoom as close as he might like.

- **Change field of view** – the second widely used option amongst the graphics editors. This option does not move the camera but only changes the field of view (FOV for short). This means the camera will never fly inside the object and is able to provide as close of a look as the user desires. The main disadvantage of this approach is slightly worse handling when zoomed in really close.

- **Reset camera** – a simple button to reset the camera to the default position. This is the position the object appeared in after the model import was done.

Feel free to change this camera zoom behavior to the one you like, it does not affect the performance or model quality in any way.

18.1.2 Model transformation

While Pepr3D tries to orient your model correctly from the start, it only succeeds if your model follows the Blender axis alignment. This is the **X left-right, Y forward-backward, Z down-up** axis alignment. This is the most common spread axis system and is the one Pepr3D supports without any adjustments. This is the axis system that Prusa Slic3r uses, so if your model works in Slic3r, it should natively work in Pepr3D. Should you need to use a different system, this segment of the settings provides you with means to correct for this change manually.

- **Model roll** – $0°$ – $180°$ – If the model is incorrectly oriented, you can rotate the model with this slider. You can control the camera with the right click drag, which rotates the camera in two axes around the model. This option rotates the model along the third axis, which allows you to fully customize the object's orientation. If your model is incorrectly oriented, this slider should always be able to fix the issue.
- **Model position** – -1.00 – 1.00 in X/Y/Z axes – Pepr3D tries to center your model onto the guiding grid. Should Pepr3D not succeed, these options allow you to position the model correctly.
- **Reset model transformation** – *button* – a simple button to reset the model to the default position. This is the position the object appeared in after the model import was done.

18.1.3 Guidance graphics

The last section contains a couple of guiding tools that the user might want to use to help his orientation in the space or understand the model's geometry better.

- **Show grid** – *On / Off* – determines whether the model should be positioned on a guiding grid or not. This is done to simulate the printing bed of the 3D printer itself. It also should help the user with orientation, if the model is symmetrical.
- **Show wireframe** – *On / Off* – if enabled, every triangle on the model will have its borders displayed in a high contrast color. This option is turned off by default since it is distracting on complex models but might be a really useful tool to see how the model geometry looks like and how it got changed by the Brush or Text tool.

18.2 Settings

For now, there are only a couple options in the Settings menu. They are, however, very important to understand.

18.2.1 Editing the color palette

This is probably the most important setting in Pepr3D, because it affects every tool and the outcome of the export. Here you can **add or remove** additional colors into the palette, **change** the already existing colors, as well as **reorder** them or **reset** them into the default stage. Let's talk about these options in more detail.

- **Add color** – *button* – adds a new color into the palette. The hue of the color is randomized. See *changing the color* to learn how to change the hue. The maximum number of colors for now is **16**. Even the most advanced FDM printers cannot print more than a few colors and 16 offers a lot of flexibility while keeping the user interface simple and easy to use.
- **Delete color** – *drag and drop field* – deleting a color is as simple as grabbing it and dragging it onto the red zone with the *Drag color here to delete* description. Be careful, since this **will permanently delete** all information associated with the color. Any triangles painted by this color will get a new color (the previous one in the palette). This operation **is undoable**.
- **Reordering the colors** – *drag and drop within palette* – you can rearrange the colors by dragging and dropping them accordingly. Note that this **does not change** the colouring of the model and is **only cosmetic**. This operation **is undoable**.
- **Changing one color** – *right click on the color* – if the user wishes to change the the color, it is done by left-clicking the color patch. Once clicked, a color picker gets displayed and the user can change the color there. Note that this **does change** the color on the model in real time, which allows the user to preview the change. This operation **is undoable**.
- **Resetting colors to the default values** – *button* – once clicked, the colors will be reset into the default values and the model will be re-coloured according to the positions of the first four colors. This operation **is undoable**.

18.2.2 User interface preferences

This section allows the user to modify the appearance of the Pepr3D software.

- **Side pane width** – *in pixels, relative to Pepr3D's width* – this modifies the width of the side pane area. This is especially useful if the user is working with more than four colors or on a high resolution display.

Appendix II: CD Attachment

Chapter 19

CD contents

The contents of the accompanying CD are organized as follows:

- **Assimp_for_Pepr3D** contains our build of Assimp library for compiling.
- **Pepr3D-Documentation** contains this documentation in .pdf.
- **Pepr3D-SampleModels** contains a set of 3D models as examples.
- **Pepr3D-Windows-x64** contains an *executable file* and other files required for running the application.
- **Source** contains source codes of Pepr3D and certain libraries.

The organization of the files corresponds to our GitHub repository releases that are available at https://github.com/tomasiser/pepr3d/releases.