



**FACULTY  
OF MATHEMATICS  
AND PHYSICS**  
Charles University

**SOFTWARE PROJECT**

**World's best 3d printer driver**  
Project specification

Authors: Bc. Bohuš Brečka,  
Bc. Matúš Goliasš,  
Bc. Qingqin Hua,  
Bc. Vojtěch Kužel,  
Bc. Vojtěch Tázlar

Supervisor: Tobias Rittig, B.Sc., M.Sc.

Consultant: doc. Ing. Jaroslav Křivánek, Ph.D.

Prague 2019

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	3D-Printing basics . . . . .	3
1.2	Previous work . . . . .	3
1.3	Our project . . . . .	4
<b>2</b>	<b>Functional requirements</b>	<b>5</b>
2.1	Workflow . . . . .	5
2.1.1	Model loading . . . . .	5
2.1.2	Visualization . . . . .	6
2.1.3	Model transformations . . . . .	6
2.1.4	Viewports . . . . .	7
2.1.5	Optimization . . . . .	7
2.1.6	Output for printers . . . . .	8
2.2	UI design . . . . .	8
<b>3</b>	<b>Software design</b>	<b>12</b>
3.1	Requirements . . . . .	12
3.2	Architecture . . . . .	12
3.3	Modules . . . . .	13
3.3.1	Vulkan module . . . . .	13
3.3.2	Geometry module . . . . .	14
3.3.3	Pipeline module . . . . .	18
3.3.4	Preview rendering module . . . . .	20
3.3.5	Controller module . . . . .	21
3.3.6	UI (Qt) - module . . . . .	24
<b>4</b>	<b>Execution of the project</b>	<b>26</b>
4.1	Team . . . . .	26
4.1.1	Supervisor . . . . .	26
4.1.2	Consultant . . . . .	26
4.1.3	Team members . . . . .	26
4.2	Difficulty and timetable . . . . .	26
4.3	Stages of the project . . . . .	27
4.3.1	Preparation . . . . .	27
4.3.2	Specification . . . . .	27
4.3.3	Initial working solution . . . . .	27
4.3.4	Main development . . . . .	27
4.3.5	Polishing . . . . .	28
4.3.6	Documentation . . . . .	28
<b>5</b>	<b>Minimal implementation</b>	<b>29</b>
5.1	Geometry module . . . . .	29
5.2	Vulkan module . . . . .	29
5.3	Pipeline module . . . . .	29
5.4	Preview rendering module . . . . .	30

5.5	Controller module . . . . .	30
5.5.1	Continuation without controller module . . . . .	31
5.6	UI (Qt) - module . . . . .	31
<b>Bibliography</b>		<b>33</b>

# 1. Introduction

The fast development of 3D printing technology over the last few years significantly increased the availability of 3D printers. They are able to fabricate objects with various shapes, appearances, and mechanical properties.

Research in the Computer Graphics Group at Charles University is seeking to improve the state of the art methods for full-color 3D printing [Elek et al., 2017, Sumin et al., 2019]. The aim is to make the printout’s appearance match what the designer intended on a virtual counterpart. One particular project addresses textured objects to appear as close as possible to the user’s expectations despite the printing material having significant subsurface scattering properties. These lead to blurred texture features and reduce the perceived contrast.

The goal of this project is to create an easy-to-use application that would allow users to access this cutting-edge research and provide them with the ability to accurately preview (soft-proof) their printout before printing. This functionality cannot be found in existing applications and is novel to our efforts.

## 1.1 3D-Printing basics

In additive manufacturing or 3D printing, multiple approaches exist to form three dimensional objects from a digital description. All of them have in common, that objects are divided into layers (slices) by a driver software (slicer) which are then replicated by the printer on top of each other to form an object.

In this project, we target Material Jetting (synonyms: Inkjet 3D printing, PolyJet™) where a liquid photopolymer resin is jetted through an inkjet head onto the build plate and then cured with UV light. The advantages of this technique are a high resolution of up to 1200 dpi and multi-material capability (up to 8) which allows for full-color mixing using CMYK inks.

The materials are highly translucent for the UV curing process to work and to allow the discrete inks (CMYK + White) to mix subtractively through light scattering underneath the surface. While the translucency allows for more saturated colors than pure opaque materials, the disadvantage is that texture features and edges get blurred over the surface and the contrast gets limited.

This can be intuitively explained by the fact, that light is irrecoverably absorbed by darker inks, in turn, reducing the brightness of white parts in the vicinity. This means that the overall absorption should be minimized to preserve a high contrast and that absorptive layers should be placed close to the surface to reduce the lateral blurring effect.

## 1.2 Previous work

The field of Appearance Fabrication, where our project is also located, has mostly focused on faithful color reproduction in the past.

Brunton et al. [2015] showed how halftoning can be used to discretize arbitrary texture values into a volumetric composition of printing base materials. It uses a single forward model to distribute the color error over the surface and

extrude the material mixture into the depth. Babaei et al. [2017] eliminated the halftoning noise by stacking layers of base materials with various thicknesses into the depth (contoning) instead of distributing the mixture laterally. The line of work by Brunton et al. [2018] and Urban et al. [2019] addresses the problem of how translucency can be controlled if the input texture has an alpha channel.

Our research group co-authored two articles on opaque texture reproduction. In our work by Elek et al. [2017], we were the first to address texture blurring caused by unwanted lateral light scattering underneath the surface of planar slabs. The second work [Sumin et al., 2019] proposes scattering compensation for arbitrary 3D geometry and improves color reproduction on thin geometry, where color cross-talk limits the gamut. Both publications rely on an inverse optimization loop consisting of forward simulation and backward refinement.

All these works produce a passive, reflective-based appearance by varying absorption volumetrically and thus modulating environmental light. This appearance can only be previewed digitally with an accurate light transport simulation, as the surface color depends on the complex mixture of base materials underneath it.

## 1.3 Our project

Our project focuses on transforming cutting-edge research from this year’s publication [Sumin et al., 2019] into an interactive application for users with little expertise in the field. Our efforts face two main challenges - connecting with an existing optimization framework and designing a UI that provides, both, control over the framework and functionality of a Slicer software.

As of now, the optimization algorithm is accessible as a console application that lacks any interactivity with a user. It is also difficult to install since it requires numerous dependencies, which need to be acquired manually beforehand. The framework is now a mixture of C++ code and Python scripts connecting individual parts together. Part of our work would be to make it more consistent (leaning towards C++) and define an external API so it can be easily connected to different systems (firstly to our UI).

The UI must provide basic Slicer functionality - loading of a textured 3D model, potentially checking correctness and printability of the model to avoid any problems during subsequent processing and placement of the models on the printing table. After that, the user can run the optimization pipeline on any subset of models before finally generating slices for the whole printing table. Allowing for interactive manipulations, an adequate level of control over the pipeline must be provided. Our key feature is a realistic visualization of the virtual objects that allows for accurate previewing before the actual printing process. This way, expensive material can be saved and the design process is significantly sped up as one does not have to wait for a long print job and post-processing to inspect the results.

## 2. Functional requirements

Our first and foremost objective is to provide a good software solution for use and previewing of results of the aforementioned optimization algorithm for 3D printing. To achieve this objective, the simplicity and predictability of controls in our software are extremely important. We need to keep in mind that our users will be mostly people who don't know how 3D printing optimization works.

### 2.1 Workflow

Because of the nature of the optimization framework, we expect our application to mostly follow a traditional slicing software workflow with the twist of an additional but optional optimization step:

1. Load model
2. Transform models and place them on the printing table
3. Run optimization pipeline on selected models
4. Preview expected results before printing
5. Generate slices for the 3D printer

This is a high-level overview of the workflow which might differ in order as we may load multiple models and for each of them, steps 2-5 may be done in an arbitrary order (or even repeated multiple times). For further details about these steps, refer to the following subsections.

#### 2.1.1 Model loading

To start actual work, users need to load a model. We will support widely used formats (e.g. obj, 3ds, fbx) which can be exported from 3D modeling applications. Since we are focused on full-color 3D printing, the model should also have some textures assigned to fully utilize the potential of the optimization pipeline even though it is also possible to optimize models without a texture (e.g. models with a single color). Loading will run in the background and the user will be informed about its progress via a loading bar placed in the UI. It will also be possible to queue the loading of multiple files. When the loading successfully finishes, newly loaded models will be added to the list of models and reasonably placed on the printing table (so the models won't intersect each other if it can be avoided), otherwise - in case of failure - appropriate message will be displayed.

An additional feature of loading is checking the printability of the loaded model. Since the loaded model can be corrupted in multiple ways (holes, inverted normals, self-intersection, etc.) the user should be notified of these issues before he decides to print the model. In case it is not printable or if the shape of the printed result would differ a lot with the shape of the loaded model, the user will be prompted to adjust the model and load it again. Currently, it is not fully clear

to us which of the problems can be fixed automatically and which ones can only be detected, but we are actively investigating this further.

After loading multiple models, it may be tedious for the user to orientate in the scene only by the viewport. Instead, it is beneficial to have a possibility to see the scene content as a list of the model's names, where one can select objects by clicking on their name. This idea can be extended to a hierarchical view of the scene allowing the representation of ownership between scene objects and other entities, like textures. We plan to add a panel with this functionality.

### **2.1.2 Visualization**

Our application will provide two means of visualization of the models, both of them have different advantages and are meant for different purposes.

The first one is a real-time renderer displaying each object's surface and texture with a simple reflection model. In this mode, the user will be able to see a virtual printing table and loaded models on top of it - he will be able to inspect them from different angles and also manipulate them (see the next section) in real-time. This is intentionally interactive so the user will see an immediate response to his actions. Because of the fast and simple way of rendering used (no ray tracing or physically based shading), the displayed model will be different from what they will look like after printing.

The second (precise) rendering system is here to give the user an idea about how the model will look like after it is printed. It is done with physical based light transport simulation taking into account (measured) properties of the printing materials. This requires a significant amount of computation, so it is currently impossible to achieve a framerate sufficient for features supported in real-time visualization. Our first goal is to support this visualization in an on-demand fashion - the user will navigate the scene in the real-time mode and then executes this realistic rendering to compute a single frame using the current scene and camera state. Optionally he may choose to preview only selected models, so the computation power won't have to be wasted on models of no interest (e.g. objects in the background). Later on, we would like to achieve an interactive performance of this rendering mode - so it would interactively show the same view (e.g. camera orientation, model placement) as its corresponding real-time visualization at all times.

### **2.1.3 Model transformations**

One of the necessary properties of software for generating slices for 3D printers is the ability to perform basic transformations - translation, rotation, and scaling - to organize objects on the printing table, set desired sizes and reduce the amount of support material that needs to be used. We plan to support the setting of these properties with absolute values (e.g. in properties of the selected object) but also with gizmos which show up on top of selected models in the real-time visualization and can be manipulated with a mouse (e.g. dragging of the model on the printing table). Both of these approaches are present in most of the digital content creation (DCC) tools so anyone who has worked with 3D models before, shouldn't have any problems with them.

Considering the gizmos themselves, we have several options to choose from. They might be implemented as a part of the UI in which case, the gizmos would be rendered on top of the selected objects and user selection of axis handles would essentially happen in two dimensions. The other option would be to introduce gizmos as new objects in the scene rendered alongside the geometry. This rendering would be executed by the real-time renderer. A big advantage of this approach is that the gizmos would be partially hidden by objects they are assigned to which would look more natural. This effect could be achieved in the screen space as well but it would be more difficult and most likely not accurate for non-convex geometry. Also, we can work with two types of object space gizmos, one represented as geometry and the second one as primitive shapes, mainly a single line for each axis handle. Then we need to choose whether we want to intersect the geometry or analytically calculate distances between lines in 3D. Currently, we are leaning towards the analytical object-space representation because it seems like the most general and manageable option.

### **2.1.4 Viewports**

So far we have described different means of visualization and features allowing users to interact with loaded objects on the printing plate. All this will be drawn in a dedicated window, which we call viewport.

While doing transformations of the scene objects, users may require to look at the scene from multiple directions at once (it is common for 3d modeling applications to have perspective in combination with orthographic views like top, left, front, etc. open at the same time). Likewise, we decided to have an option to open multiple viewports which support our two displaying systems (real-time and precise) available (and there could be more in the future).

There is still an ongoing discussion about whether to show the results of the precise visualization in the viewport or open an independent window. If it acts as an interactive renderer, it would be better to put it in the viewport, since the user could move the realtime viewport and watch the precise image change in some other viewport. However, as we already mentioned in the Visualization section, the precise renderer will not be interactive at first. The user will just request rendering of the scene's current state using the camera in the selected viewport. After that, the rendering will not react to viewport changes, so there is no need to see more viewports at the same time. Independent windows would also have several advantages - they could be easily dragged to other monitors, resized to full screen without seeing the application panels, etc. In the end, both of these possibilities may even merge into a window that is dockable in the place of the viewport - effectively achieving both of the variants at the same time.

### **2.1.5 Optimization**

After the desired model transformations are done (namely rotation and scaling), the optimization can be started. The optimization does not take into account the position of the model on the printing plate as well as other models placed there, so each model can be optimized independently. This can be used to allow users



to run an optimization only on a selected subset of the models to get the result faster on the models of interest.

Since the optimization is a long process, the user needs to be informed about its progress. We will provide a progress bar as well as all the statistical information that users could be interested in. Currently, the optimization can only be stopped manually - e.g. when the user deems the result to look acceptable. There could be some automatic stopping criteria, but their development is not meant to be a part of our work.

The optimization process generates a lot of intermediate data, some of which can be interesting for the user, e.g. the intermediate assignment of the materials to the voxels. Their difference between iterations shows what the optimizer changes and the magnitude of changes can serve as an ad-hoc stopping criterion. Here we plan to use the scene hierarchy view - the volumes created in each iteration can be attached to their respective object where users can select one of them for visualization in the realistic preview rendering.

It is important to allow users to visualize the results of the optimization (so they can decide when to stop the optimization), therefore a connection between the pipeline and the precise renderer is required. When the user runs the precise rendering while also running an optimization, the most recent version of the volumetric object representation will be used. Alternatively, he might choose one of the intermediate versions of the volume, saved during the optimization process, including the initial one.

### **2.1.6 Output for printers**

As a final output of our application, we need to generate printer instructions that fabricate the object in our memory. We can control each individual voxel's material assignment using a stack of slices (in the Z direction) that is read back by the manufacturer's printer driver and subsequently sent to the printer itself. Each slice consists of a simple .png image where each pixel has one of 8 colors that map to a corresponding base material (Cyan, Magenta, Yellow, Black, White, Transparent, Support) or air. We additionally output an XML file which describes this mapping and additional printing parameters. The number of slices depends on the Z height of the tallest object, whereas their dimensions depend on the XY extent of all objects on the printing table.

## **2.2 UI design**

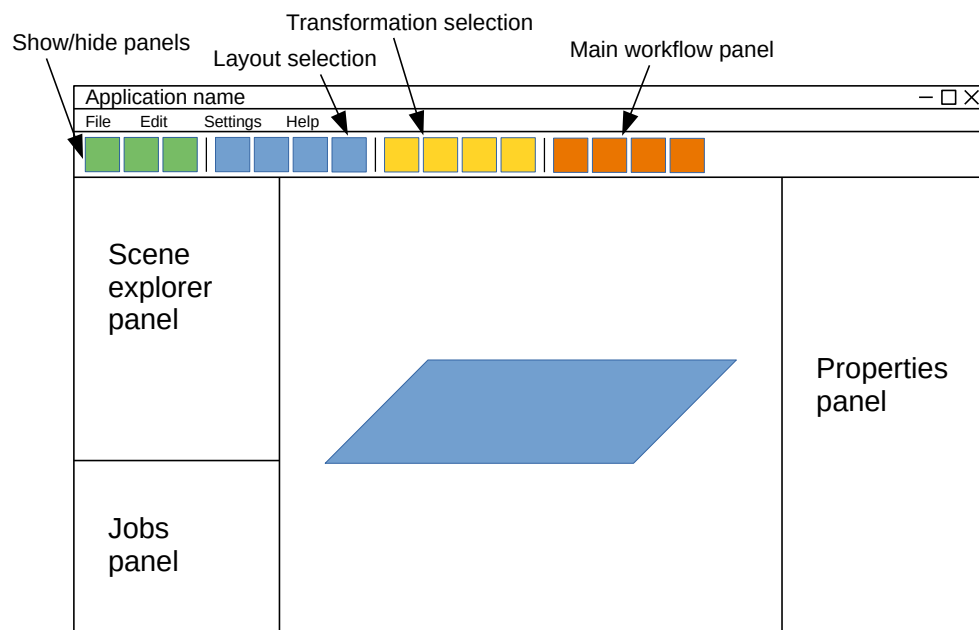
It is natural that our UI design is greatly inspired by existing Slicer (e.g. PrusaSlicer, GrabCAD Print) and 3D applications (e.g. Autodesk 3ds Max, Blender). It is because our application will mainly function as a Slicer with the additional functionality of the optimization pipeline which on the UI side (e.g. predictive visualization) behaves in a similar way as renderers inside of the DCC tools. We also respect our users' habits from existing software and not radically change the user experience with our approach.

The design of the UI must encompass all of our functional requirements in an as simple as possible way. So to accomplish all the requirements we need to provide:

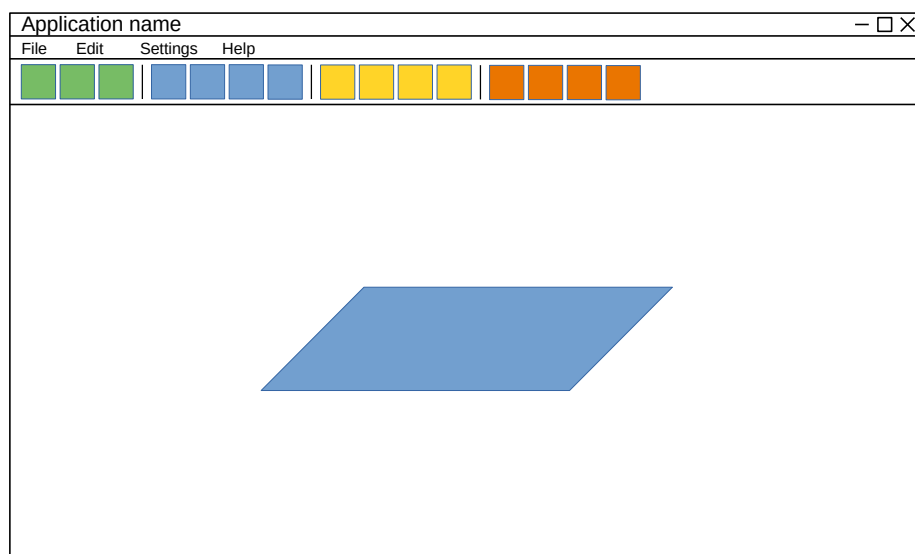
- Support for steps from the main workflow
  - Toolbar with buttons for the main actions
  - Submenus or additional toolbars for additional subactions
- Viewport(s) for the real-time visualization
  - We need a big section of the application screen for visualization purposes which may be divided into multiple viewports with independent camera settings
    - \* Layouts of multiple viewports will follow common conventions:
      - Single viewport
      - Two viewports next to each other
      - (Potentially) 3 viewports - one on the left and two on top of each other on the right (or the other way around)
      - Four viewports in a grid
      - User can switch between different layouts and individual viewports must be resizable
  - When a set of models is selected - corresponding gizmos for the transformations should be displayed in all of the viewports
- Displaying the precise visualization
  - The exact solution not yet decided, as there are two possible approaches:
    - \* As a different type of the viewport
      - Users would be able to switch the type of each of the aforementioned viewports between real-time and precise visualization
    - \* Open an additional application window which works as an image viewer for the precise renders
      - Additional functionality (e.g. difference images) could be provided way more easily
- Scene explorer panel
  - Tree-like organization of all the models in the scene
  - Models can be selected from this menu or by clicking on them in the viewport
  - Option to show/hide individual models or their groups
  - Additional data may also be present - e.g. finished precise renders of the model may be added under it (which may be selected and displayed on demand)
- Properties panel
  - A panel containing properties of a selected model (e.g. for the setting of transformations with explicit numbers) or of a selected tool (e.g. optimization pipeline or precise rendering)

- \* Both settings of optimization pipeline and precise visualization may end up in up to two additional independent panels - depending on the future testing
- Jobs panel
  - There are three long-running tasks which we need to keep track of - loading of models, run of the optimization pipeline and progress of the precise visualization rendering
  - Jobs panel should provide a progress bar for each run of one of these long term operations (and history of the finished ones)
  - Together with a progress bar, we plan to provide stop/cancel button and button for the opening of logging window (e.g. containing logging output of the optimization pipeline)
    - \* This way additional detailed information about the progress of the jobs will remain available but it won't clutter the application environment
- Toolbars
  - Other than a toolbar for the main workflow there should be additional toolbars
    - \* For switching between viewport layouts
    - \* For showing/hiding of the panels
    - \* For the selection of model transformation
  - Their position (on top or on the sides) and exact look (icons or text) is not yet specified
- Manipulation of the UI
  - All the panels and toolbars should be dockable - users should be able to move them around according to preference
  - Panels should be resizable
  - This way a user will be able to customize the UI for his needs
  - Because of that, we would need to save settings of the UI (docking, sizes) on closing the application and load it back during its startup

When we put all of these parts together we get a simple UI design, which can be easily modified to fit the user's needs because individual panels and toolbars are dockable and/or resizable.



Example of default UI composition.



Example of customized UI.

## 3. Software design

### 3.1 Requirements

The timeframe of the software project is limited but we want to achieve the most we can. The goal is to put together something that we can build on, improve and add new functionality step by step.

For this reason, the strongest requirement is to make the architecture extensible and each part should be easy to switch for a new, better variant. To achieve this, the development of particular components should not be driven by the underlying technology, but rather by expected functionality. Also, the limitations and rules given by the technology should not spread out of a component where it is used.

We aim for a reliable, well-tested and well-documented piece of software. To have enough time for this, we need to carefully select the minimal required functionality and prioritize it over other parts.

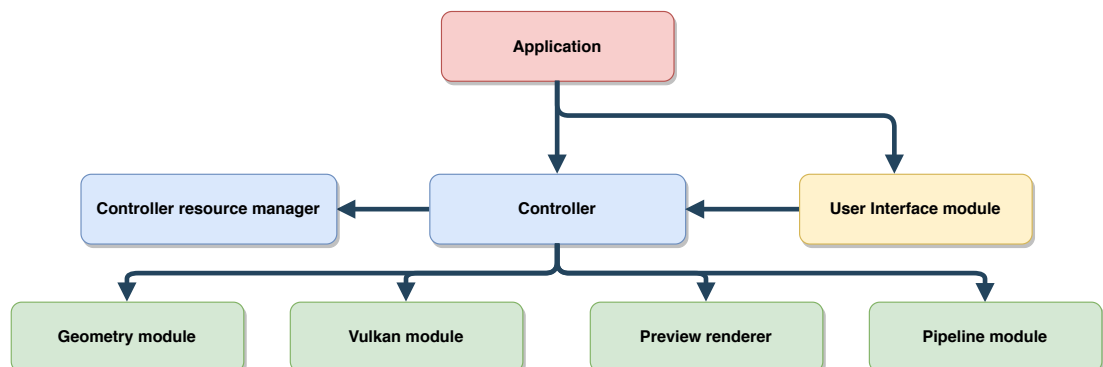
Since this product has the possibility to be used in the real industry, we need to keep in mind what the real users would need and what is really valuable for them.

To achieve maximum possible users, we want to target our application for Windows, Linux and possibly macOS, but at first, we will start developing on Windows and port it to Linux later on. The reason is that all of our team members have only a little experience with Linux and no experience with macOS development.

### 3.2 Architecture

Since the work needs to be split between five people and since the project can be naturally divided into several functionally distinct parts, we divided the project into modules. A module is a well-defined and enclosed functionality pack that communicates with other modules via their interfaces. In our case, every module is a separate project in the solution. Each of these projects except one compile into a statically linked library and can be referenced by other projects via their API headers. The main project compiles into an executable file and it is supposed to start up the application and cleanup running modules on termination.

The structure is visually explained on a dependency graph:



The main project launches the User Interface (UI) module and the Controller. The UI sends requests based on the user input to the controller to access resources and issue user commands. The Controller processes those requests by assigning work to helper modules. For model loading and checking, we have the Geometry module, the Vulkan module is for real-time rendering whereas the Preview renderer is for path-tracing the scene and seeing the models in a more realistic manner. The Controller resource manager, which maintains the 3D models and images, etc., and finally the Pipeline module, which takes care of the communication between the existing optimization pipeline and provides its results.

While designing this architecture, we took special care to avoid interdependencies, therefore any communication against the arrows on the graph is handled via callbacks.

## 3.3 Modules

### 3.3.1 Vulkan module

This module is responsible for displaying the user's scene in real-time. It is intended to run together with some UI editing elements (gizmos) that will allow for example translating, rotating, and scaling the scene objects. The users will be able to check and modify the printing table layout and also check the appearance of the scene objects, and they can make sure the objects have correct UV mapping, etc.

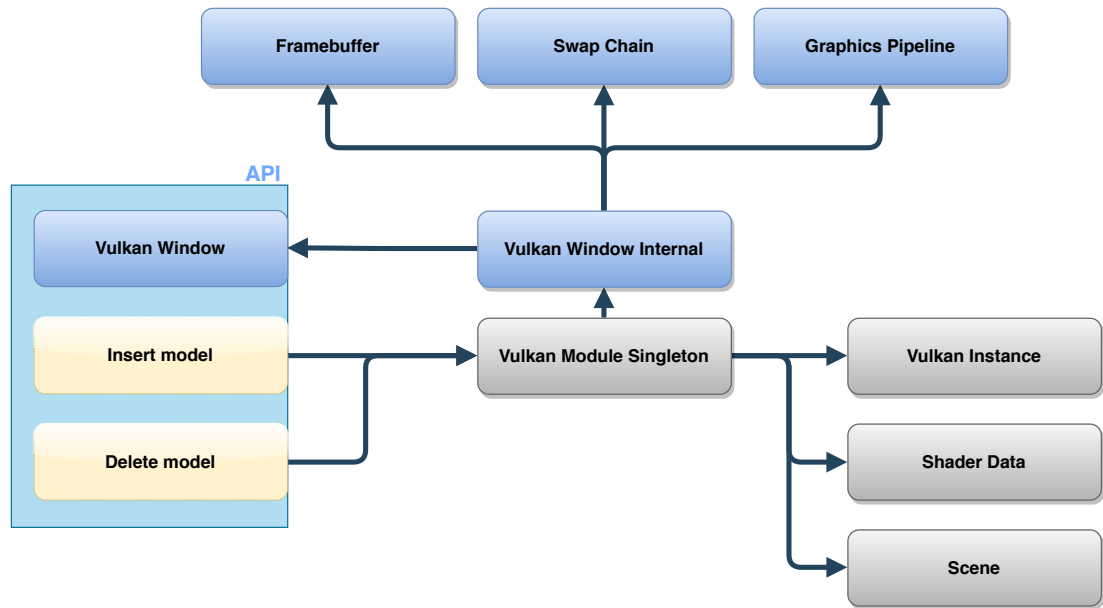
As the name suggests, we have chosen to use Vulkan API to implement this renderer. We want to allow the users to have high-poly scenes that will be rendered in real-time, therefore Vulkan API is a good choice, as it allows for a lot of optimizations and unlocks many threading options.

The main purpose of the renderer is to have a textured preview of the objects to be printed, but we will try to also make it as pretty as possible, if we have time. We could possibly add shadows and other visual features to the shaders, to make the viewports easy on the eye.

#### Structure

As the UI will be capable of splitting the screen into multiple viewports, the renderer needs to be able to render it into all of them at the same time. There will be a representation of a Vulkan registered window, and the renderer will manage all the references.

There will also be a core rendering engine, that communicates with the graphics card driver and manages objects shared between the windows. One of the shared objects will, for example, be the scene, as all of the objects to be rendered will be loaded into graphics card memory, which will be shared for all of the windows. The module will be capable of uploading new scene objects and also deleting them from the graphics card memory.



The Vulkan Module Singleton node represents all of the shared data, that need to be accessible from each and every one of the windows. This includes the scene, hence the Insert and Delete model nodes from the API are connected to the singleton node.

An external window can be wrapped into Vulkan module functionality via the Vulkan Window, as hinted by the blue API node. Some of the rendering objects and their functionality should not be exposed to the API, that is why there is a Vulkan Window Internal node, which will serve as a point of communication between the API window and the singleton. This internal window will hold Vulkan objects like the framebuffer, the swap chain, the graphics pipeline, etc.

## Shaders and their compilation

The shaders for Vulkan are going to be written in glsl and compiled into SPIR-V by a Vulkan SDK compiler. For easier usage of the compiler, there will be a tool capable of using this compiler to compile all the necessary shaders into binary format, when Vulkan SDK is properly installed.

### 3.3.2 Geometry module

The Geometry module is responsible for loading the model file selected by the user inside our application and check whether the imported model is printable or not. The model file formats we support are frequently-used in industry, including but not limited to obj, 3ds, fbx.

This module has two main parts, loading, and checking. In the loading part, all the geometry data will be loaded from the model file and later transferred into the Controller resource manager and used by the renderers and the pipeline.

The checking part will have the responsibility to determine whether the model is printable or not. Users will know the result through UI. This is done by checking the different properties of each model individually, so the user can be notified which property is not satisfying the conditions for printing.

## Structure

The Geometry module will be able to load models with textures in different 3D-file-formats. It will also do the checking job to make sure the imported models can be printed. For each part, we are using external libraries. In the loading part, we are using Assimp. In checking part, we are using OpenVDB for two purposes, checking and voxelization, with the possibility to add another geometry processing library for more robust checking.

## Loading

We choose Assimp for loading purposes, a library for import and export of various 3D-file-formats. Assimp supports multiple model formats that can satisfy our needs. It also includes post-processing functionality, eg, triangulation, and UV-flip, which ensures loading of models that can be rendered. Assimp post-processing can also generate missing normals and invert normals which are facing inwards - which allows loading of models with these issues.

Since we are using our own geometry representation for usage in different modules, we need to convert the default Assimp representation to our internal geometry representation. Our representation is straightforward, it includes vertex positions, vertex normals, UV coordinates, and texture reference paths.

Besides transforming geometry data to internal geometry representation, we need to consider how to order the data inside the geometry representation. A single file might contain multiple models with multiple meshes. In this case, we need different ways to split these models. For rendering purposes, we need to split each model to multiple meshes by material, which will be used for both real-time rendering buffer and the pipeline. For the checking purpose, we need to split models in a way that each model is a single mesh. After the geometry loading process finishes, the Controller module can retrieve the data, store them or pass them to other modules for further processing.

## Checking

Most likely input models will not be perfect, they may have holes or insufficient wall thickness, which may lead to corrupted printouts. In this case, the users should have feedback on whether their model is good enough for printing, to avoid any negative surprises.

A perfectly printable model should satisfy many requirements, but it's hard to find general consent on how many points need to be checked for a robust printable geometry. Every 3D printing software has its own checking framework.

We made a survey of the common checking functionality that a 3D printing software should support to ensure the model is printable:

1. No inverted normals - all the normals should be facing the same direction - outwards.
2. Sufficient object size and wall thickness - these parameters are related to the resolution of a particular 3D printer. The geometry shouldn't be too small or too big, also the walls of the model shouldn't be too thin.



### 3. Water-tightness and no self-intersections.

In the checking stage of the geometry module, we need to check all these properties. Results will then be transferred to the controller module and UI module so that the user will, in the end, get the message whether there are any issues with the model or not.

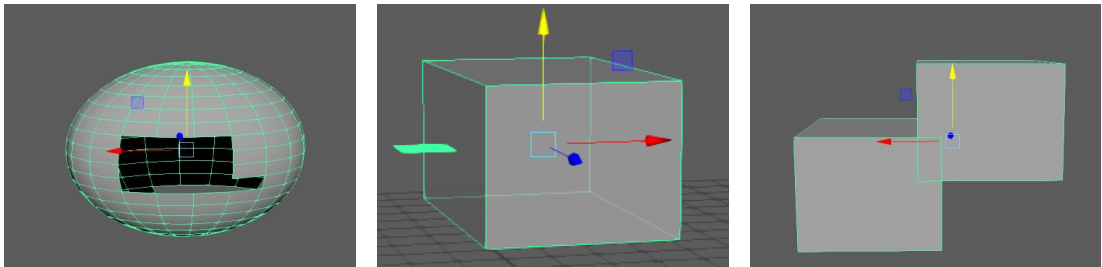
Due to the communication between other modules, we need a structure for storing the result of each property that needs to be checked. For water-tightness and manifold, we will use two simple booleans. For object size and wall thickness, we will save a floating-point number (because our software allows resizing of models, this value may change). For inverted normals, we will use Assimp post-processing capability to fix the flipped normals during the loading stage.

We may use an external library to implement the checking. There are two possibilities, first of them is to use a geometry processing library, which provides rich functionality for geometry data inquiries, such as Libigl, CGAL, and Openmesh. The second one is to use a voxelization library, e.g. OpenVDB. A voxelization library converts meshes into a volume representation. It may handle slightly incorrect models and it also allows us to convert the volume back to a triangle mesh - we are still conducting research to find out which properties can be reliably tested this way (for further discussion see next chapter).

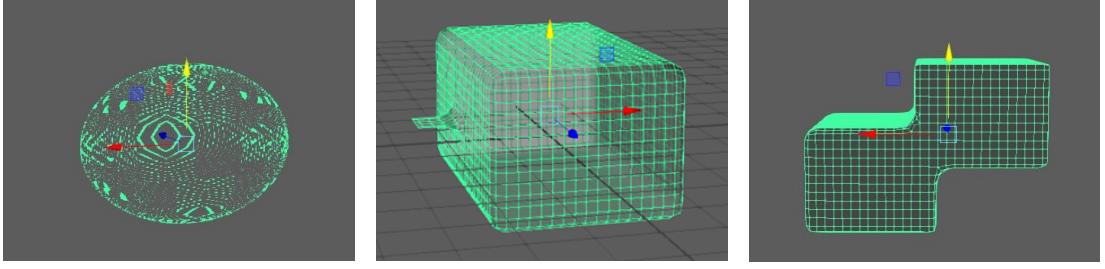
### Open questions

Designers and artists may not understand the technical requirements for input meshes and their otherwise good looking models may not have all the properties that we expect. Also, the transformation from other representations (e.g. NURBS) to triangular meshes can yield incorrect models. In this case, it is reasonable to try to deal with minor imperfections of the models.

We described in the previous chapter what are the perfect printable models, here we are going to show examples of imperfect models and how well the reconstructions works for them.

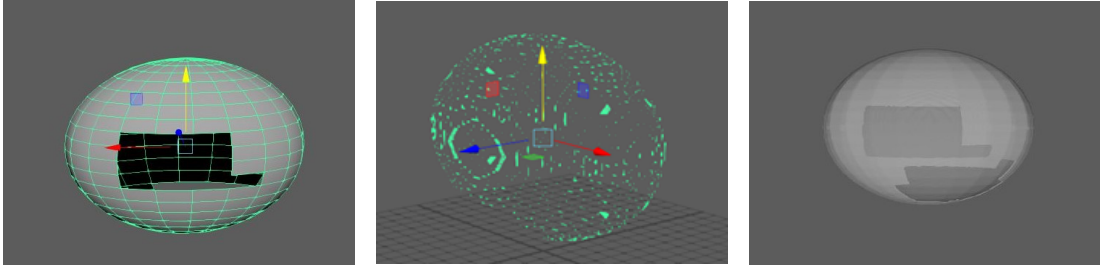


Different types of imperfect models. The left one has an open surface. The middle one has one horizontal plane with no-thickness. The right one has self-intersections. All of these three models are non-manifold.



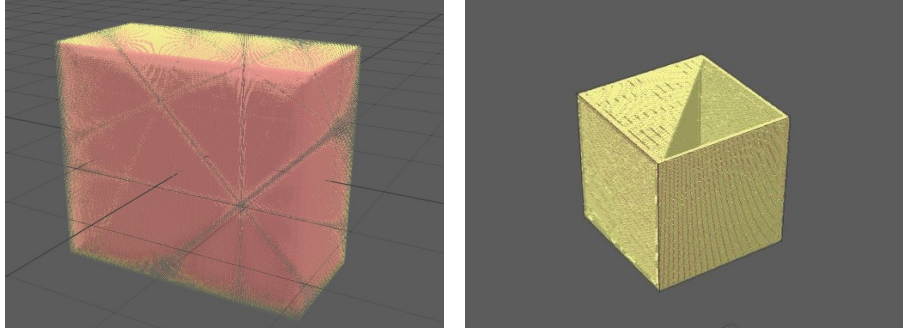
Results of the reconstruction of previous images. As we can see, the reconstruction does not work for the left model. In the other two cases, it successfully created correct manifold meshes from non-manifold ones - we could take advantage of this and allow slightly incorrect models in some cases.

As the OpenVDB documentation explains it has limitations when it comes to geometry with open surfaces. We did some experiments to see how well the reconstruction with open surfaces works in OpenVDB. It turned out that even though the reconstruction runs without an error, the results are far from the shape of the original models.



When the OpenVDB library gets an open surface (left image), it voxelizes only the surface polygons and leaves the insides empty, which most likely leads to reconstruction which was not expected by the user. The reconstruction allows a setting of a parameter to define the “tightness” around the voxels of the created surface. Setting the value of this parameter above 0 moves the surface edge outside while negative values will move it inwards. We can see the results for different values of the parameter: 0.01 (middle), 0.05 (right). Nonetheless, insides of the voxelized model stay empty.

It is clear that we cannot perform a correct reconstruction of the open surface models - the interior of the models stays empty, which is a major problem for the optimization pipeline. It would view the object as a thin geometry trying to optimize also the look of the inner surface instead of assigning it an opaque filling material.



Visualization of voxelization functionality built into OpenVDB. The yellow points represent exterior voxel points, the pink ones represent interior voxel points. The left image shows that a closed cube is properly voxelized (has correct interior and exterior voxels). The right image shows that an open-faced cube has only exterior voxelized, while the interior stays empty.

Regarding this issue, we still need more tests to decide how we approach the mesh with missing faces. So far it seems that OpenVDB can handle some non-manifold models, while it fails to reconstruct open surface ones. A plausible solution is to add an algorithm that would check the water-tightness of the input model and notify users in case they load a model with the open surfaces since it would most likely lead to a completely incorrect result.

### 3.3.3 Pipeline module

This project builds on the optimization framework developed by the researchers at Charles University [Sumin et al., 2019]. The purpose of this module is to expose optimization functionality for the rest of the application. It is in our interest to rewrite all the existing source code to a cleaner, more extensible and readable form, but because of the sheer extent of the framework, some parts will be used with only minor changes in the scope of the software project.

The current codebase consists mostly of Python scripts and C++ code, a huge amount of work is offloaded to 3rd party libraries, like OpenVDB, numpy, boost, etc. In the final state, we would like to have the Python scripts replaced by C++ and use as few 3rd party libraries as possible. One of the software engineering goals at the beginning is to design the work in a way that the codebase can be incrementally rewritten and we can have a fully working application at any stage of the process.

#### Structure

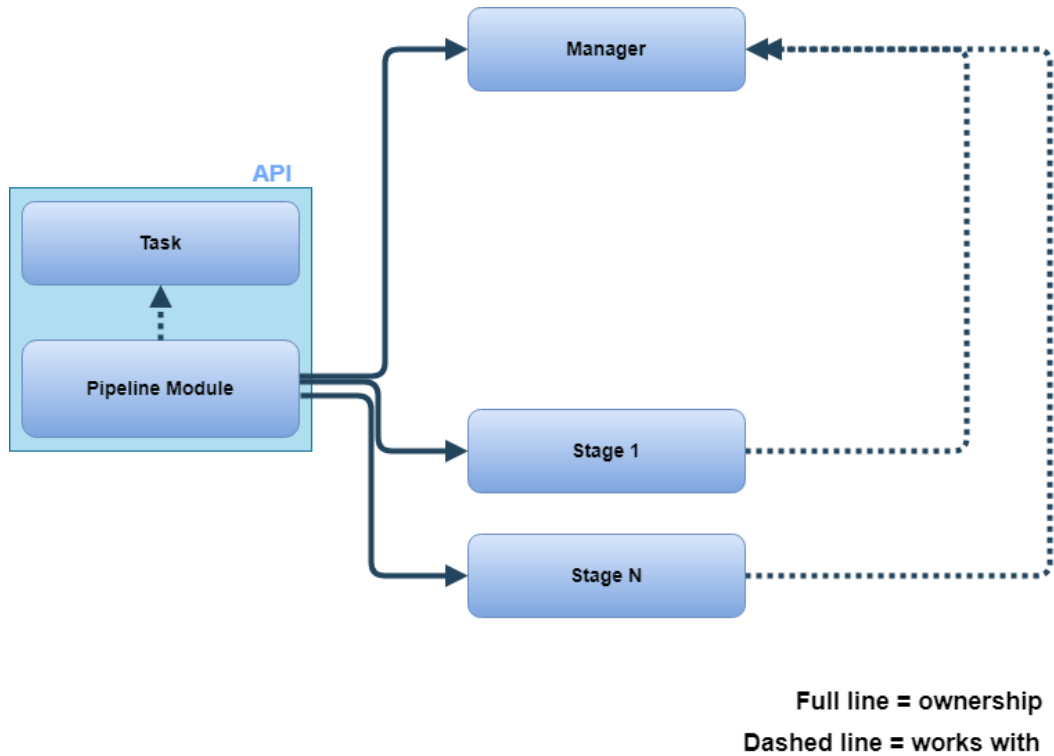
Since we are building on existing code, we will first describe the architecture in the current state.

The entry point of the framework is a set of Python scripts responsible for the configuration of the pipeline and preparing various input parameters. The pipeline naturally consists of multiple stages, each stage is executed either as Python function or as an executable from the command line in case of the stages written in C++. The data interchange between the particular stages is done via saving intermediates in files, whose file-paths are distributed to respective stages by the configuration scripts.

A crucial part of the optimization process is predictive rendering. Currently, Mitsuba, a research-oriented open-source rendering system, is used for this purpose. Its main perks are the modularity allowing to easily add plugins and built-in support for distributed rendering, which is very useful in performance demanding applications like this one.

The optimization internally uses a voxel representation of the object's inner volume. To work with that, OpenVDB is used. It provides sparse volumetric data representation as well as a set of algorithms built on top.

We inspire our architecture with the current version of the framework. There will be a top-level class providing API for the rest of the application. The API will contain basic functions for running the pipeline, canceling and observing the progress as well as functions for the setting of various parameters. The data needed for one pipeline run will be enclosed into a Task class. To be able to easily pass parameters to other functions, there will be a Manager class holding the task data, settings and managing the temporary files paths. The pipeline consists of multiple stages, each represented by one class with a run function, taking the Manager class as a parameter. The structure is sketched in the image below:



The minimal requirements for this module to work is to acquire all the dependency libraries used by the pipeline (they cannot be just taken from the current framework, since it is developed only on Linux and our target platform is Windows) and create a C++ wrapper around the current Python scripts / C++ code and define the API mentioned above.

Further goals, as we already mentioned, are to rewrite the Python scripts to C++, refactor the C++ code and create one library where the calls of particular stages will be only function calls instead of running executables in the command line. The next thing is to eliminate the number of intermediate files used to pass

data between different stages of the pipeline. Replacing that with direct passing of data in parameters in function calls would lead to performance increase and overall clarity of the code. Since the progress when working on this is really unpredictable and it clearly cannot be done completely in the timeline of the software, we do not set goals on how much of the scripts should be rewritten.

### 3.3.4 Preview rendering module

While the Vulkan module renders the printing plate in real-time allowing users to place objects on the printing plane or inspect the shape and texture by rotating the camera around them, this module shows the user how the object will look like after it is printed. It will be achieved by physically correct simulation of light transport in heterogeneous media, which is, in this case, the partially translucent material used in the printers. This is a crucial feature since it can save time and expensive printing material because the object does not have to be printed multiple times before achieving the desired appearance.

The ability to predict the final look of the printed model is not only beneficial for the user, but it is also an integral part of the optimization pipeline. The optimizer is comparing the desired appearance with the prediction and is subsequently distributing the difference between the two into the current volume. Since this functionality is already a part of the pipeline, we do not have to build this module from scratch.

Currently, it uses the volumetric path tracer available in Mitsuba renderer. The pipeline contains Python scripts that provide the scene and other necessary data and execute the rendering. Mitsuba supports multithreading, so there is no need for extra steps to achieve that.

In the scope of the software project, we are going to reuse this solution, since replacing Mitsuba with an in-house renderer would be very time consuming and we would lose also other features of Mitsuba which are currently used in the framework. In order to incorporate that in our application, we plan to create a C++ wrapper around Mitsuba, allowing us to feed it with data in the format used by the other modules and hide the boilerplate code for initializing Mitsuba and creating the scene.

The API class will provide functions to set parameters, create a scene, run and cancel rendering and observe rendering progress. The whole module will consist mainly of the API class, some minor settings, utility classes and the native Mitsuba classes which are documented in the Mitsuba documentation.

There is one important difference to the existing solution. The prediction in the pipeline only renders the inner volume using a special camera that shoots rays perpendicular to the object's surface, which yields the prediction of the texture appearance, as if it was unwrapped to 2D. On the other hand, we are interested in displaying the printed object as if it was seen through a real camera, in 3D. To display an object faithfully, we need more than just light transport in the volume inside of the object, we also need to count on the surface - one of the most visible effects are the specular highlights caused by reflection on a glossy finish. It is up to a discussion, what glossiness do we assign to the object's surface, since the output of the printer is rough and the glossiness is achieved by consecutive polishing.

The ultimate goal is to make the renderer interactive, meaning that it will automatically re-render the image after the user makes a change in the scene, for example, moves the camera. This feature is quite popular among the users of production rendering systems like Corona Renderer or V-Ray. Implementation of this feature is not that difficult, but reasonable usability requires fast scene building and fast rendering to provide users with a close-to-real time response. These two things are not present in the current solution used for rendering, which is the reason why we classify this feature as optional in the scope of this project.

### 3.3.5 Controller module

Creating multiple modules in our project spawned a discussion on the way these modules are connected, what are the dependencies between them and how to handle the communication among the modules. Our solution to this problem is creating a controller that will serve as an intermediate layer in our application. This means that most of the communication will pass through this module. We cannot say that every message passed in the application will be processed by the controller as there are exceptions that would warrant it but we believe a central module will make the application management easier.

In addition, our decision is influenced by the need for resource management and scene description which are inseparable from a 3D rendering and processing application. Therefore we chose to place scene and resource management functionality in the central module. The module will expose different types of resources, currently, we have in mind geometry data structure provided by the geometry module and textures which would be handled by the resource management subsystem. Furthermore, the module is going to expose the scene description in the form of a scene graph containing all of the necessary information for the other modules. For example, rendering modules will be able to traverse the graph acquired from the controller, read visible geometry objects with textures assigned to them and render them to the selected window. Unfortunately, the exposed objects cannot be part of the controller directly due to other dependencies and we will place them inside a separate submodule with working title Controller Scene Module. We would like to include all scene management under the controller module, and thus we will most likely create this connection by placing the word “controller” within the submodule’s name.

Following that, we are wary of a certain redundancy within the module since it might only pass calls of the modules around, which would result in duplicated functions. However, we believe that the monitoring capabilities of such a module, the abstraction of asynchronous behavior executed upon the functions of the other modules and unification of the core features of all the APIs will outweigh the risks. An additional caveat is that adding new visible functionality to some module will result in the need to update the controller module as well as the UI module but we think that such a problem is easily manageable and it gives us the opportunity to double-check the viability of the API design of the functions.

The main component with which the user communicates in our application is our user interface module. We do not want every part of the application to be an extension of the user interface, and thus, we split it into modules and create an abstraction layer. As a result, we expect that the majority of the messages

will pass from the UI to other parts of the application and our controller module will encompass those functions needed for the user interface. We expect that the rest of the communication will not require any interference from the UI module. To give an example, the Vulkan module will provide the option to render into a window using some camera. In addition, the Vulkan module will give an option to create GPU buffers for the geometry objects in our application. Then, the controller module would include in its API the rendering function but the creation of the buffers would be excluded because it is part of resource management and UI does not need to know about buffers. There will likely be another part of the API which handles the loading of files selected by the user, which would deal with the intricacies of buffer management.

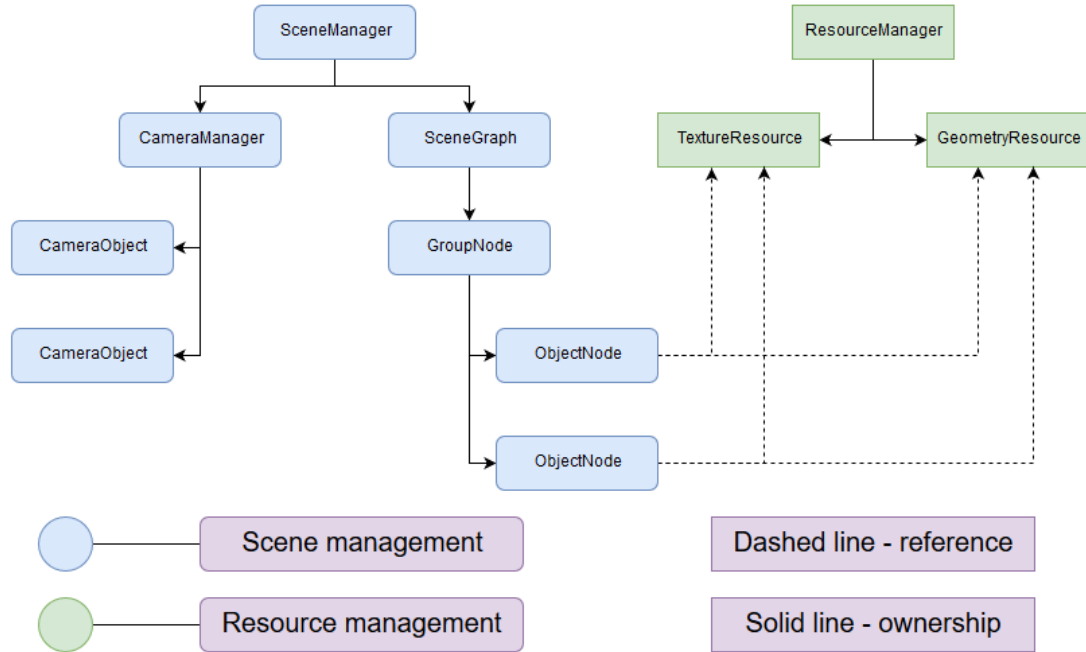
Finally, we would like to abstract asynchronous implementations of the individual functions from the user interface module. There has to be the notion of callbacks but we do not want the UI to handle thread generation and management. Therefore, unless the specific module executes its long-running functions asynchronously, we will implement such behavior in the intermediate layer - the controller module.

## **Scene management**

Every application which handles three-dimensional scenes and sets of objects requires some sort of scene management. We decided that our scene will be represented as a tree-like graph. It is expected that the graph will not be extensive and there will be no need for optimizations upon the structure due to the low amount of objects which can be present in a single scene. Our intention is to have a virtual table with several, possibly very detailed, models and we do not plan explicit handling light sources or other objects.

Alongside the graph of objects, we need to store cameras that will be tied to individual windows. A camera object is part of the scene but it will probably not be an internal node of the scene graph because of the special connection between rendering windows and cameras. Nonetheless, every object which can be considered as a part of the scene will be owned and managed by the scene submodule of the controller. Other modules will then be able to access these objects selectively based on the executed operation, for instance, the rendering module will have access to cameras and the scene graph for rendering.

It is important to note that we do not consider the scene as the owner of heavy resources. Let us consider an object node that will handle a single instance of some geometry. This node should contain the transformation of the object and then the reference to the actual geometry which is transformed when needed. Ownership of potentially large resources is separated from the scene management and the nodes of the graph will only reference an existing resource. This way we will be able to reuse the same geometric data to create multiple instances of the same object. We will handle textures in a similar manner. In the following diagram, we show how we imagine the connection between scene management and resource management.



Finally, we will implement a way of synchronization between the application and the contents of the scene graph. We expect the long-running tasks to be executed in parallel which means that we need to take care of processing the scene graph and adding new nodes to it at the same time. Our intended solution to this problem is to lock the graph when it is edited or processed. However, we expect difficulties in locking the processing functions, and thus, we would like to distinguish between the real graph in the scene manager and a local copy of the graph. This local copy would not be affected by the changes in the real graph, and therefore, safe to traverse.

## Resource management

The scene graph should not be the owner of large objects it might contain, specifically in the case of geometry where we want to be able to create instances. Similar thoughts can be applied to texture handling because multiple objects might want to use the same texture.

We decided to create a resource manager submodule as a part of the controller alongside the scene graph. This submodule will store data loaded by the application and provide it through the controller module to the rest of the application. It will also handle the loading process of resources that do not have dedicated management in other modules, such as textures. We will use one of the open-source image loading libraries, such as OpenImageIO, FreeImage or STB for texture loading. We require a permissive license and support for all of the major file formats. The second requirement puts SBT into unfavorable position because it does not support OpenExr images. We are leaning towards the FreeImage library because it is easy to use, does not contain a lot of files and dependencies and supports all of the file formats we might need. Furthermore, the resource manager will handle the storage and possibly deletion of resources with respect to the available memory. However, we do not expect that our application will need



rigorous memory management, and therefore this is a feature of the submodule which will be resolved when it is needed.

In addition, the modules in our application will require different views on the same data, for instance, libraries behind preview and real-time rendering expect geometry in different forms. We could store the resources in one format and let the modules transform it when they access the resource. However, we believe that a central solution to the resource transformation problem will be easier to manage. Thus we decided to incorporate this functionality into the resource management module so that the other modules will be able to access the required data without the need to care about their format.

In conclusion, the resource manager will serve as a data handler behind the scene management submodule which will reference the data for instanced use. Finally, the controller will use both submodules to handle data of our application.

### **Gizmo implementation**

One part that is not fully decided is the placement of the gizmo implementation within our module hierarchy. It can be fully implemented inside the UI module or it can be split between the controller module and the real-time renderer.

There are multiple problems with the implementation of gizmos and for the object space-oriented methods we found two libraries, `tinygizmo` and `LibGizmo`, which would most likely need several adjustments according to our tests. In the case of screen space gizmo, we would need to implement the logic from scratch. Therefore, we will likely opt for the analytical object-space oriented approach.

### **3.3.6 UI (Qt) - module**

The UI module provides a graphical interface for the user to ease up the handling of the functionality provided by other modules. As such it does not manage any resources or other modules. It processes user input and based on it sends requests to make specific operations to the Controller module. How are they handled is then completely irrelevant to the functionality of the UI module.

On the other hand, what the UI module manages are all the UI widgets that are visible to the user - e.g. windows, viewports, toolbars... - but that does not mean that the UI module itself needs to handle all that in a given UI widget happens (e.g. real-time preview is rendered into the viewports from external - Vulkan - module). Nonetheless, the UI module processes the user input and other events in the event loop and manages communication between individual widgets.

### **Technology**

Because it is necessary for our project to be cross-platform and most of our codebase will be in C++ we had to select a cross-platform library for the implementation of the UI as well. One more consideration that had to be taken into an account was ease of extensibility, because the software development on this application may be continued after the end of the software project.

We have looked into the immediate mode GUIs like Dear ImGui, Nuklear, and NanoGUI and even though they offer extremely lightweight solutions for the graphical side of the UI, they don't handle user input where other libraries must

be used and some of the functionality like docking of windows was hardly supported by them. In the end an implementation based on this kind of technology would soon grow into multiple interconnected libraries due to our non-trivial demands. Part of the functionality which is provided by other more sophisticated frameworks would need to be reimplemented as well.

For these reasons, we have focused on more full-fledged UI frameworks like wxWidgets, GTK+, and Qt. Because there are no problems with licensing of these frameworks we have ended up with the biggest (with most features available) and most commercially used (and as such tested) one - which is Qt. We don't think that its deployment size (starting around 30 MB) - its biggest disadvantage - , is an issue when individual 3D models may be way bigger.

## 4. Execution of the project

### 4.1 Team

#### 4.1.1 Supervisor

Tobias Rittig, B.Sc., M.Sc. is a doctoral student at CGG and joined first-author of the publication on which this project is based. He has co-supervised the software project “Pepr3D” together with Oskar Elek, a former member of the computer graphics group.

#### 4.1.2 Consultant

doc. Ing. Jaroslav Křivánek, Ph.D. is an associate professor at CGG with major experience in the content production industry from his rendering startup “Corona”. He supervises, among others, the Appearance Fabrication efforts at CGG consisting of three Ph.D. students.

#### 4.1.3 Team members

- Bc. Bohuš Brečka (team leader) - master student of computer graphics at MFF UK currently working as a developer of Corona Renderer (Chaos Czech a.s), with a history of game development and 3D visualizations.
- Bc. Matúš Goliaš - master student of computer graphics at MFF UK with previous projects focused on game development and data visualization.
- Bc. Qingqin Hua - master student of computer graphics at MFF UK with previous experience as a software developer in MORE VFX, China.
- Bc. Vojtěch Kužel - master student of computer graphics at MFF UK previously working as a developer for VRgineers, Inc., currently working as a junior software engineer for Bohemia Interactive Simulations.
- Bc. Vojtěch Tázlar - master student of computer graphics at MFF UK currently working as a research intern for Chaos Czech a.s.

### 4.2 Difficulty and timetable

The expected finish date of this project is in April 2020.

A great deal of work was already done before the project was approved, we were discussing the possible solution, inspecting the existing framework, writing the proposal and during the summer (July - September) we started the actual work on the project.

In the upcoming months, we estimate 8-16 hours per week per team member spent on the project.

## 4.3 Stages of the project

### 4.3.1 Preparation

This part is already considered complete. It included:

- Studying the current codebase provided by the researchers, coming up with options on how to plug it into the rest of the application
- Discussing different technologies and libraries that need to be used (e.g for the UI)
- Discussing which features need to be implemented
- Dividing the work among the team members according to their abilities and interest

### 4.3.2 Specification

First two months since the project approval (November 26th). We will use the information gathered in the first stage and in the already launched implementation process to write this document which you are currently reading.

### 4.3.3 Initial working solution

The goal of this stage is to assemble the basic project structure and have a notion of the necessary functionality in terms of interfaces. Each module should provide all the functionality that is required of it in a simplified form or using mockups so that all the connections between the modules can be made before this stage is considered finished. That is important so that we can test that our design is sufficient, and so that we are able to modify this fundamental design in time if need be.

Since the proposal approval, we made most of the modules work in a simplified form and started connecting them together. We plan to finish this stage until the end of November.

### 4.3.4 Main development

When the basic structure is in place and all the modules are connected, it will be necessary to implement all the functionality as described in the specification. This stage is a time for adding features, implementing missing parts, and improving the already implemented code.

Since the work on this stage may include also the implementation of nice-to-have features, it's time frame should be set reasonably so that we have enough time for the remaining stages, which are more important. We expect to finish this stage by the end of January 2020.

### **4.3.5 Polishing**

The polishing stage will be a window allocated for bug-fixing, performance improvement, and refactoring. This stage tends to be underestimated as we know from our experience, so we decided to allocate at least two months for it (until the end of March 2020).

### **4.3.6 Documentation**

The software will be submitted with Doxygen documentation, which is being written along with the code, as well as with a detailed document explaining how to install the application, documentation for users and description of the architecture from the developer's point of view. We expect to finish this stage as well as the whole project in April 2020, the documentation can start during the Polishing stage, since it is connected in many things.

## 5. Minimal implementation

Working on a project in a group of people brings certain risks. The more people the bigger is the risk that someone will leave the project for various reasons. Reduction of the team size will definitely lead to slower progress and redistribution of the work will be necessary. There are also other possible causes of progress decline, like problems with selected technology, unexpected difficulties concerning the work with the research code, etc.

For these reasons, there is a need for a backup plan. Our backup plan includes a simplified and trimmed down version for each of the components. They are described in the following.

### 5.1 Geometry module

The ability to load a 3d model is crucial for further use of our application, therefore we definitely need to implement this. Fortunately, it is almost entirely carried out by the Assimp library, including the check of correctness of input files for making sure the model can be rendered correctly.

What is a bigger challenge is the check of printability of the loaded models. With the help of the OpenVDB library, we can reproduce high-quality geometry models for printing. However, OpenVDB itself has the limitation for processing open surface geometry, therefore, the way for reproducing robust geometry needs to be handled carefully.

In the minimal implementation, the geometry module will be able to load and print closed geometry, and leave the case of taking care of open surface geometry to the users. This can be improved by the additional check of open surfaces, and further investigation on OpenVDB.

In the worst case, if the person is not able to work on this module anymore, the geometry module will not implement any checking job, therefore, the user will get no warnings if the incorrect models are imported.

### 5.2 Vulkan module

To be in its state of minimal implementation, this module needs to render the scene with shared model memory onto possibly multiple viewports. The scene should be properly textured and use the normals and uvs from the input model file.

The techniques required for simple rendering are multi-buffering (double and more, to prevent drawing directly to the display buffer) with some advanced techniques like mailboxing (to prevent screen tearing), multisampling, depth-buffering, and they should all be present in the minimal implementation.

### 5.3 Pipeline module

Unlike other modules, this module builds on a fair amount of already working code. The only thing preventing the possibility to directly use the code is that it

was developed on Linux with no apparent intention of an easy port on Windows. We thoroughly discuss the problems connected with the port in the Solution chapter, the most significant ones are the dependencies and compiler-specific errors in the C++ code.

In order to make the framework usable on Windows in the shortest time possible, there is a possibility to use a container (e.g. Docker). This is not a preferable solution since it would introduce yet another dependency and as well as a lot of additional complexity.

Since we have already obtained all the necessary dependencies on Windows, it is probably easier to try to run the optimization framework on Windows without major code changes described in the Solution chapter (code cleaning, rewriting to C++). The work that was already done regarding this can be kept for later.

In case the person working on this module was not able to continue working on the project, someone else would have to be assigned to do it since the optimization is the core idea of the application. Luckily, the new person would not have that much work to achieve a working version of the module, as we described in the previous paragraph.

## 5.4 Preview rendering module

Although this is a really important feature, it is already partially implemented in the current optimization framework. It allows creating renders of optimized objects from various angles during the optimization process. The intermediate images can be provided to the user.

From this point, various improvements can be made. We can start at allowing users to see the images in some temporary folder when they are saved, the next step could be to show them in the viewport. The final step, as we described in the Solution chapter is to allow users to render own images from the viewport camera.

Unlike other modules, this one does not have a strictly assigned person who will be working on that. Therefore the worst-case scenario, in this case, is that there will not be anyone having time to work on this module while also working on the module that he/she has been assigned. The application can function even when this module will be completely omitted because no other module depends on its functionality. In the case at least a little time can be allocated for this module, we can use the existing functionality in the pipeline described at the beginning of this chapter.

## 5.5 Controller module

This module consists of two major parts. Those are the controller, which serves as an intermediate communication layer for the other modules, and scene management, which stores information about the scene and handles application resources.

Most of the functionality of the controller can be moved to the UI because that is the main user of the communication. This would result in the disappearance of the intermediate layer which would cause a less organized project structure but it would not hinder the development in a significant way.

On the other hand, scene management is a core requirement of the application. We need a way to represent the scene, store information about it and to handle the resources which are referenced in the application. In the minimal implementation, this includes storing objects with their transformation and cameras used to render them into viewports. In addition, objects need to reference some geometry and texture. These two resources need to be stored during the application runtime and in the case of textures loaded in the first place. All information about objects has to be provided to the other modules so that they can render it or process it in some other way.

Both parts of the controller module can be improved in a lot of ways. We can manage executed operations in the application asynchronously, gather information about the progress of different tasks or cache information so that it does not need to be recomputed on every call. For scene management, we can create a more complex scene representation, handle resources in a more general manner, create our own file format for loading and saving of our scenes or allow various operations on the objects or on the tree.

### 5.5.1 Continuation without controller module

There are lots of problems that might arise during the development and it is possible the developers of the controller module will drop out of the project. In this case, it would be very difficult to finish this module, and thus we would like to present a possible workaround for the missing controller.

The first part of the controller which is the intermediate communication can be moved to the user interface module due to our architecture and every attempt at asynchronous execution and event or message handling can be omitted.

The second part which is scene and resource management is more important from the application standpoint. However, we do not need a full scene structure with complex camera handling, scene importing and exporting, grouping objects and similar features to execute the workflow of our application. Therefore, it is possible to store all loaded geometry with its textures in a simple array of objects and work with a fixed amount of basic cameras. On top of that, it is not necessary to allow deleting or duplicating objects and they could be hidden by a flag instead. Such scene handling would not be efficient but it would allow the rest of the team to continue working on the application without the controller.

## 5.6 UI (Qt) - module

Compared to other modules implementation of the UI module directly depends on the functionality provided by other modules, which needs to be accessible on the user side and as such be part of the user interface.

Nonetheless, we definitely need to support the basic functionality necessary for the main workflow as it is described in the UI design chapter. As such the main points of the UI design must be implemented at least on a basic level:

- Support for steps from the main workflow
- Viewport(s) for the real-time visualization



- Window(s) for the precise visualization
- Scene explorer panel
- Properties panel
- Jobs panel
- Toolbars

The scope of additional functionality and available customization level of the UI is hard to predict because it depends on the level of implementation of other modules. But we want to achieve as much of the functionality described in the UI design section as possible.

# Bibliography

- Vahid Babaei, Kiril Vidimče, Michael Foshey, Alexandre Kaspar, Piotr Didyk, and Wojciech Matusik. 3d color contoning. In *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication*, SCF '17, pages 8:1–8:2, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4999-4. doi: 10.1145/3083157.3092885. URL <http://doi.acm.org/10.1145/3083157.3092885>.
- Alan Brunton, Can Ates Arikan, and Philipp Urban. Pushing the limits of 3d color printing: Error diffusion with translucent materials. *ACM Trans. Graph.*, 35(1):4:1–4:13, December 2015. ISSN 0730-0301. doi: 10.1145/2832905. URL <http://doi.acm.org/10.1145/2832905>.
- Alan Brunton, Can Ates Arikan, Tejas Madan Tanksale, and Philipp Urban. 3d printing spatially varying color and translucency. *ACM Trans. Graph.*, 37(4):157:1–157:13, July 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201349. URL <http://doi.acm.org/10.1145/3197517.3201349>.
- Oskar Elek, Denis Sumin, Ran Zhang, Tim Weyrich, Karol Myszkowski, Bernd Bickel, Alexander Wilkie, and Jaroslav Křivánek. Scattering-aware texture reproduction for 3d printing. *ACM Trans. Graph.*, 36(6):241:1–241:15, November 2017. ISSN 0730-0301. doi: 10.1145/3130800.3130890. URL <http://doi.acm.org/10.1145/3130800.3130890>.
- Denis Sumin, Tobias Rittig, Vahid Babaei, Thomas Nindel, Alexander Wilkie, Piotr Didyk, Bernd Bickel, Jaroslav Křivánek, Karol Myszkowski, and Tim Weyrich. Geometry-aware scattering compensation for 3d printing. *ACM Trans. Graph.*, 38(4):111:1–111:14, July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322992. URL <http://doi.acm.org/10.1145/3306346.3322992>.
- Philipp Urban, Tejas Madan Tanksale, Alan Brunton, Bui Minh Vu, and Shigeki Nakauchi. Redefining a in rgba: Towards a standard for graphical 3d printing. *ACM Trans. Graph.*, 38(3):21:1–21:14, May 2019. ISSN 0730-0301. doi: 10.1145/3319910. URL <http://doi.acm.org/10.1145/3319910>.