University of Applied Sciences and Arts
Northwestern Switzerland

+5.65
50
Zimmer 2
+5.15
Treppenhaus
Bad Kinder
Zimmer 1
80
2.40
+2.75
Obergeschoss
+2.75
2.00
Gartenhaus
Büro
35
+2.40
Wohnen
gewachsenes Terrain
+-0.00
Erdgeschoss
±0.00
neues Terrain
Weinkeller
-0.42
Garage
Garagentor 6.00x2.125m
-2.82
Untergeschoss
-2.82

Modelling Building Envelopes Report

“IIT22, Vereinfachte Modellierung von Gebäudehüllen”

Authors

Ackermann Patrick, Siffer Florian
Degree Program: Computer Science and Information Technology

Customer

Stalder Patrick
PlanFabrik GmbH

Advisor

Doris Agotai
Schubiger Simon
Institute for Interactive Technologies | FHNW

Windisch 2020

Abstract

In order to offer various planning and validation services to the construction industry, architectural firms, such as the Planfabrik GMBH, must calculate building envelope surface area totals by material and facing. These calculations are done automatically, by modeling buildings with software. This modeling process takes too long in relation to its complexity with architectural modeling software such as Plancal Nova. Taking into account data models and available software stacks, a Sketchup extension is the most appropriate approach to improve this workflow. Indeed, the Sketchup extension ‘Envelop’, developed as part of this project, allows users to model building envelopes with the required amount of detail and precision four to five times faster.

Table of Contents

1. Introduction

In the construction industry there are various tasks that require the calculation of building envelope surface area totals. Calculating building envelope surface area totals can be understood as wrapping a building tightly in wrapping paper and then measuring how much wrapping paper was used. That area has then to be divided by material and orientation. For example, building A has ten square feet of exterior wall type one facing north, three square feet exterior door facing south and so forth.

Such measurements are for example used to verify buildings adhere to energy efficiency standards, calculate expected energy usage for heating or plan the construction of floor heating. These processes are done for new buildings still in the planning phase as well as for existing buildings. Therefore, there is a wide variety in regards to the documents available to an architecture firm performing such services.

Usually, the process to calculate building envelope surface areas starts with a collection of two dimensional plans. The architecture firm then has to either manually calculate all surface areas or model the building in appropriate software to calculate the measurements automatically. Manual calculations are much slower than modeling, thus modeling is preferred if appropriate plans are available.

The customer, Planfabrik GmbH, is an architectural firm, offering the services mentioned above to the construction industry. To model buildings, they used Plancal Nova, a specialized architectural modeling software with extensive capabilities. Depending on the model complexity, starting from the digitized building plans, it took about 60 to 75 minutes to calculate the building envelope surface area totals. The aim of this project was to reduce the time required by four factors while still retaining most of the precision. Because there is always some discrepancy from the plans to the actual buildings, a few percentage points of error are permissible.

This project evaluated different approaches to this problem, based on different data models and software stacks in chapter 2. It was concluded that a Sketchup extension would be the most appropriate approach to solve the customers' problems within the scope of this project. Before the extension could be implemented, Sketchup had to be analysed in more detail in chapter 3. Moreover, a new improved workflow had to be designed in chapter 4. An important part of this project was also the verification that the efficiency and precision requirements were met in chapter 5. And indeed, the Envelop extension is about four to five times faster than the previous workflow with Plancal Nova and still has sufficient precision. A technical analysis of the extensions is offered in chapter 6.

2. Context Evaluation

In the following chapters, various approaches to solutions to the challenges described above will be discussed. This includes a more theoretical analysis of data models as well as a more practical comparison of concrete software stacks. At the end of the chapter the decisions mode for the Envelop extension will be justified.

2.1 Data Models

This section will compare various different approaches to modeling the data, with special attention to performing area measurements, the development effort and it's flexibility in terms of constructable geometry.

2.1.1 Polygon Mesh Based Model

A mesh based data model consists primarily of points, also called vertices, and lines connecting those points, called edges. Areas or faces are then delineated by those edges, which are used to construct polygons and finally surfaces stitched together to objects. All of these elements can be seen disassembled in figure [n].

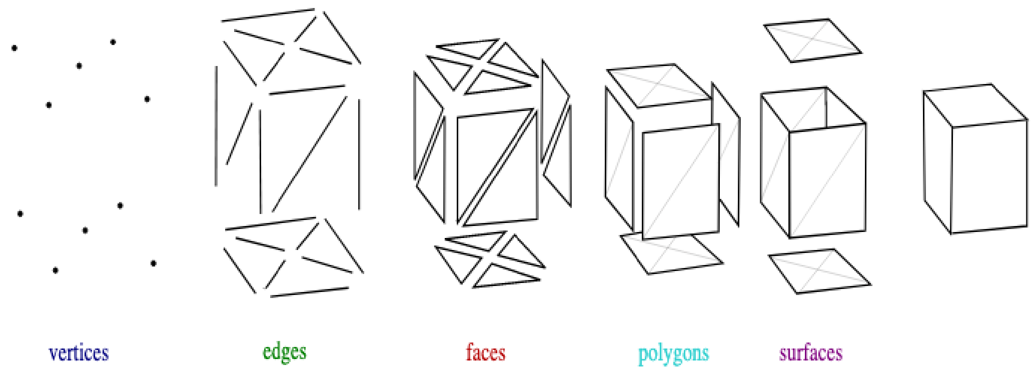


Figure n: All elements of a mesh based model of a cuboid, assembled step by step.

Clearly, calculating areas based on a mesh based model is very straightforward - using the vectors defined by the vertices and edges combined with the facing of surfaces the ares can directly be determined. Moreover, such a data model offers the greatest flexibility in terms of object complexity.

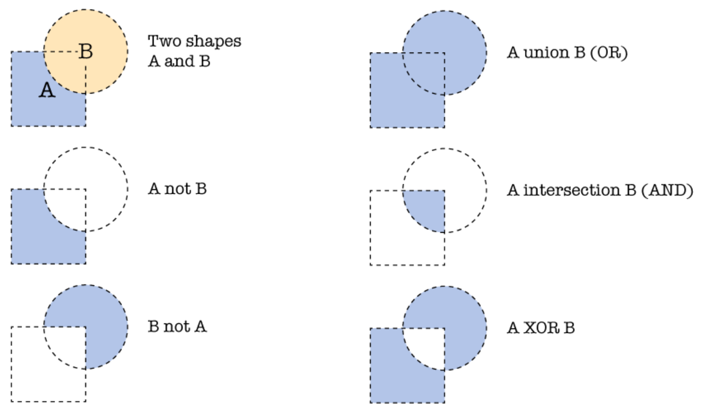
However, there are two significant challenges to using such models, both centering around model correctness. Firstly, there is the issue of water tightness. A model is called not water tight, or non-manifold if it has a hole or holes and thus cannot be unfolded into a 2D surface with all its normals pointing the same direction. The second issue is about hidden surfaces. Inside a 3D polygon mesh based object, there might be additional polygons not visibles from the outside. This would lead to additional or duplicate surface area within the model, which would invalidate the results. Detecting and resolving such issues can be very challenging manually, as missing or superfluous polygons might be hidden inside the model.

There are algorithms and complete software packages to either fix meshes or wrap existing models with a new mesh resulting in watertight models without any superfluous surfaces. However, the mesh fixing tools are mostly built for 3D scans of real life objects. Such scans often have errors that are relatively easy to fix. Modeling errors, like the ones likely made when modeling a building envelope, are rarely caught. Moreover, these tools fail often completely and destroy a model with such errors. Re-wrapping tools would fix modelling errors, however, they also commonly remove some level of detail. Even more damning, they also result in completely new faces. The faces however, have to fit the different materials and their borders on the building plans. Thus, none of these tools can be used to fix a model with modeling errors.
Re-wrapping tools can be used to recalculate the total surface area of a building model and then check if the previous total area matches up. If it doesn't, there are likely hidden surfaces or holes in the model.

Using the modeling techniques mentioned in the next subchapter, the water-tightness and hidden surfaces issues can be avoided.

2.2.1 Boolean Operation Based Model

Using primitives, meaning simple shapes, and combining them using boolean operations such as NOT, AND, or XOR complex shapes such as buildings can be assembled. This approach can be used with polygons, as described in figure [n], or with simple 3D shapes such as cylinders, boxes and circle as shown in figure [n]. This is called “Constructive Solid Geometry”, short CSG.



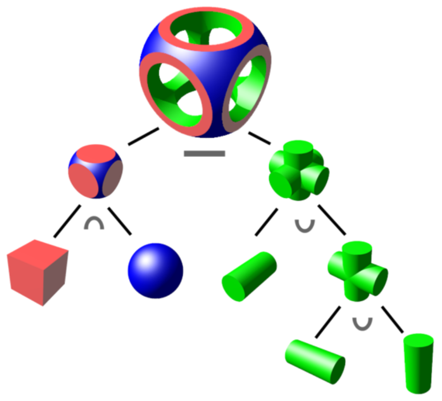


Figure n, n: On the left hand side, boolean combinations of polygons are shown. The figure on the right shows an example of boolean combinations of 3D shapes.

The one of the biggest benefits of using a model based on boolean operations, is that its near unlimited flexibility in terms of what can be built. Moreover, each intermediate step is humanly understandable and verifiable. Finally and most importantly, all models build with CSG are watertight. This means that there are no gaps in the resulting object and no unnecessary hidden areas or walls within the object.

Unfortunately, calculating areas, or more generally, forming meshes to calculate areas and visualize complex CSG objects, is extremely complex. Developing algorithms to do so is beyond the scope of this project. If however, a mesh including different materials can be generated, determining the areas involved becomes trivial again. Thus, a boolean operation based model remains a valid option if existing software can be used to process the final CSG object.

A third option is to use a specialised data model, that can only be used to model buildings.

2.2.2 Specialised Model of Building Envelopes

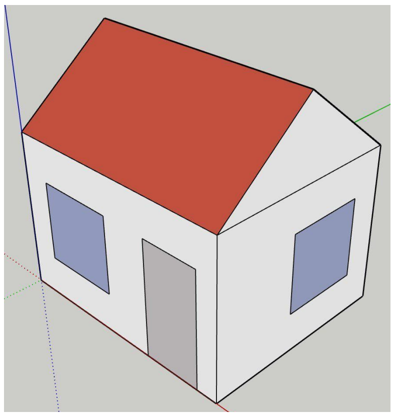
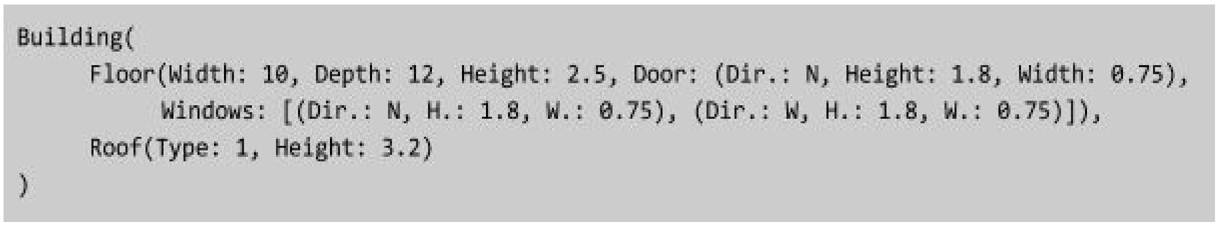


Figure n: Very simple building

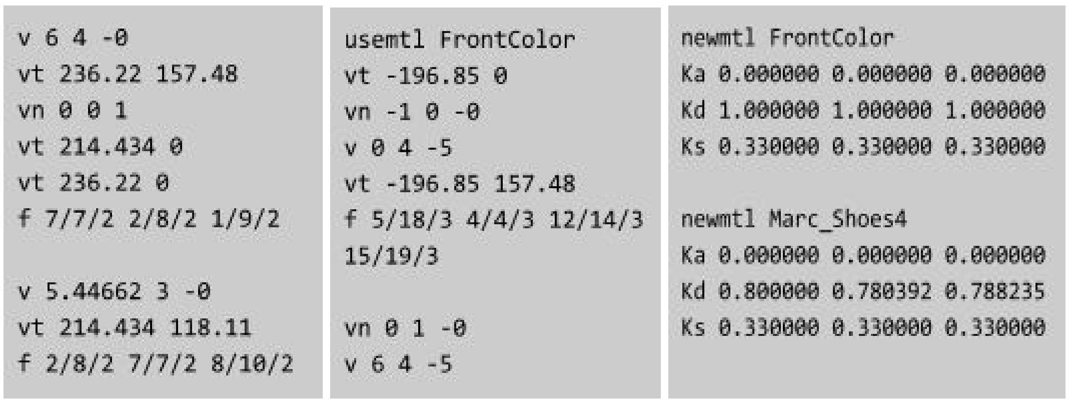
To illustrate what is meant by “specialised building envelopes model” consider the 3d model of a very simple building in figure [n]. A specialised data model might describe the building as follows in figure [n].



```
Building(
     Floor(Width: 10, Depth: 12, Height: 2.5, Door: (Dir.: N, Height: 1.8, Width: 0.75),
          Windows: [(Dir.: N, H.: 1.8, W.: 0.75), (Dir.: W, H.: 1.8, W.: 0.75)]),
     Roof(Type: 1, Height: 3.2)
)
```

Figure n: Imaginary specialised data model to describe simple building. [M]

However, using a generic mesh based geometry definition file format, such as Wavefront .OBJ, the description of the vertices, polygons, normals, etc. takes up comparatively enormous amounts of space and is neither humanly readable nor manageable. As exported by Trimble Sketchup, the building in figure [n] is about 2.5 KB on disk. Figure [n] shows excerpts of said file.



```
v 6 4 -0
vt 236.22 157.48
vn 0 0 1
vt 214.434 0
vt 236.22 0
f 7/7/2 2/8/2 1/9/2

v 5.44662 3 -0
vt 214.434 118.11
f 2/8/2 7/7/2 8/10/2
```

```
usemtl FrontColor
vt -196.85 0
vn -1 0 -0
v 0 4 -5
vt -196.85 157.48
f 5/18/3 4/4/3 12/14/3
15/19/3

vn 0 1 -0
v 6 4 -5
```

```
newmtl FrontColor
Ka 0.000000 0.000000 0.000000
Kd 1.000000 1.000000 1.000000
Ks 0.330000 0.330000 0.330000

newmtl Marc_Shoes4
Ka 0.000000 0.000000 0.000000
Kd 0.800000 0.780392 0.788235
Ks 0.330000 0.330000 0.330000
```

Figure n: Very Small excerpts of .OBJ file describing building in figure [n]. [M]

Looking at these data model, it becomes immediately obvious that, with a simplified specialised data model, the area calculations the application has to perform are much easier than interpreting a generic model. Moreover, developing a generic model, that is implicitly much more powerful than a specialised one, is much more complex and thus error prone and more time intensive to develop.

The primary drawback to a specialised data model is its inability to adapt to yet unforeseen requirements. This also means that any additional geometries supported must be implemented through the whole software stack, down to the data model. This could lead to significant additional effort during the development phase. Whether or not that effort exceeds the effort of developing a generic data model in the first place, depends on how well the to be supported geometries are studied and considered ahead of the development of the data model. Moreover, if new requirements arise after the completion of the project, the solution could turn out to be unable to fulfill them and might require significant changes to do so.

Please note, that a generic data model does not necessarily mean a widely supported data format such as .OBJ. Even if a generic data model is chosen, it is still unclear whether or not a custom file format or an existing standard should be used. The considerations in chapter “3.3 Software Stacks” will raise further points relating to the usage of a custom or standard file format.

2.2.3 Summary Data Models

	Development Complexity	**Area Calculations**	**Human Readability**	**Flexibility**
Mesh Based	Extremely High	Hard	No	High
Boolean Volume Operations	High	Extremely Hard	Step: Yes Total: No	High
Specialised Model	Initially Low	Easy	Yes	Low

Table n: Summary of evaluation of different data models. [m]

The previous considerations and the summary in table n lead to the conclusion, that CSG boolean volume operations should be used when building the model, to prevent common modeling errors while still retaining the same flexibility mesh based modeling offers. The very hard area calculations should be covered by the appropriate software stack choice however, as translating CSG models into meshes is beyond the scope of this project. If this is covered, the development complexity of a CSG based solution suits the resources of this project though.

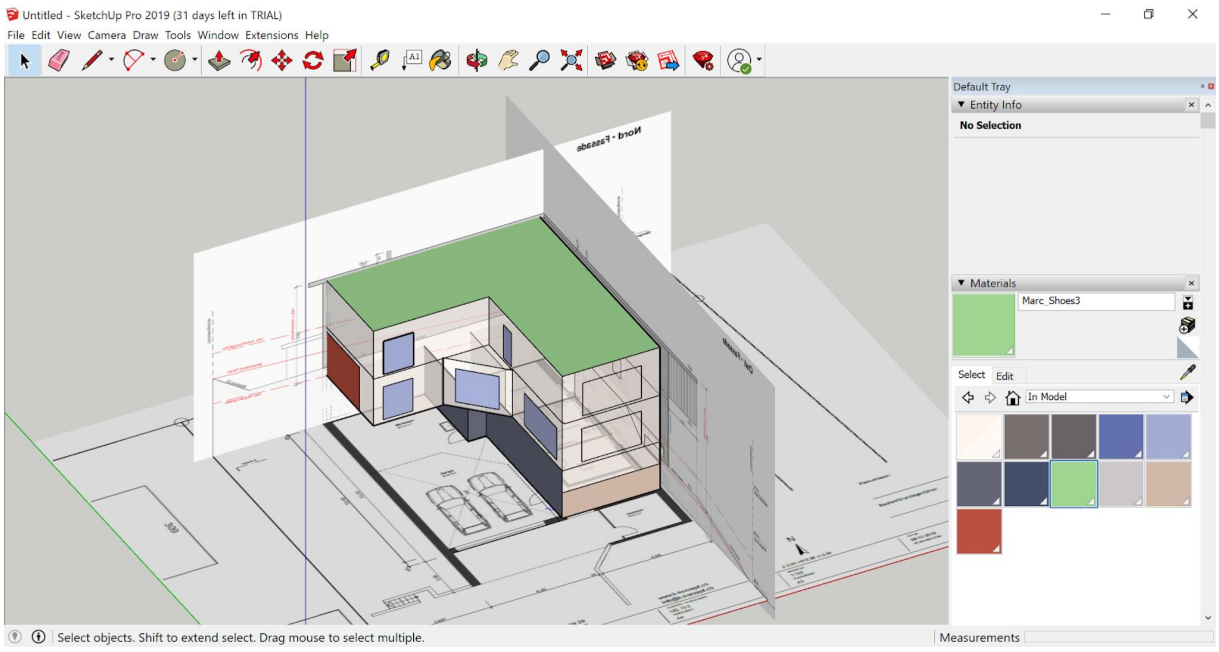
2.3 Software Stacks

The following chapters will evalue software stacks with which a solution to the initially described problem could be realised. Basic requirements for these software stacks included three dimensional visualisation of the building model, interactive tools to create that model, importing of pdf and image files and calculating surface areas of the model.

2.3.1 Trimble SketchUp Extension

SketchUp

Trimble SketchUp in figure n is a 3D modeling Software. It is available for free as a web application and a paid downloadable desktop application. The Free version lacks certain features including the extensibility with extensions. Extensions for SketchUp are written in Ruby.


Untitled - SketchUp Pro 2019 (31 days left in TRIAL)
File Edit View Camera Draw Tools Window Extensions Help
Nord - Fassade
Default Tray
Entity Info
No Selection
Materials
Marc_Shoes3
Select Edit
In Model
Measurements

Select objects. Shift to extend select. Drag mouse to select multiple.

Figure n: Sketchup with very early Envelop extension experiments. [m]

The free option of SketchUp cannot be used, as the ability to enhance SketchUp with an extension is only available in the Pro version. A pro plan license is available for $299 USD a year per user.
The Ruby environment has many libraries called gems but SketchUp discourages using them. Libraries that can be imported using raw ruby files however work reliably.

An extension for a fully functional 3D Modelling Software such as Trimble SketchUp has the huge advantage that 3D Modelling tools and data structures are already there. However problems with those tools are often hard to fix. For example, it is not possible to change the underlying data representation of the mesh. The main limiting factor that needs careful evaluation are the capabilities and limitations of the exposed API. Especially regarding the calculation of mouse interaction points in 3 dimensions and access to the textures on that interaction point.

Extensions for SketchUP have the ability to create basic input Dialogs or more advanced dialogs that display html content with javascript. These UI capabilities should suffice. SketchUp even has a built in area calculation that can calculate the surface area where a certain material was used.

The key advantages of using the Sketchup software stack are intuitive built in push and pull tools, simple materialisation, built in surface area calculations, constructive solid geometry tools and extensive UI possibilities for extensions through HTML and Javascript. Disadvantages include having to use Ruby, a language the project team has no experience with, no Ruby on Rails support within Sketchup and as such on the outside of the Ruby community - even more so with no Ruby gems support. Moreover, extensions are only supported in the paid pro version.

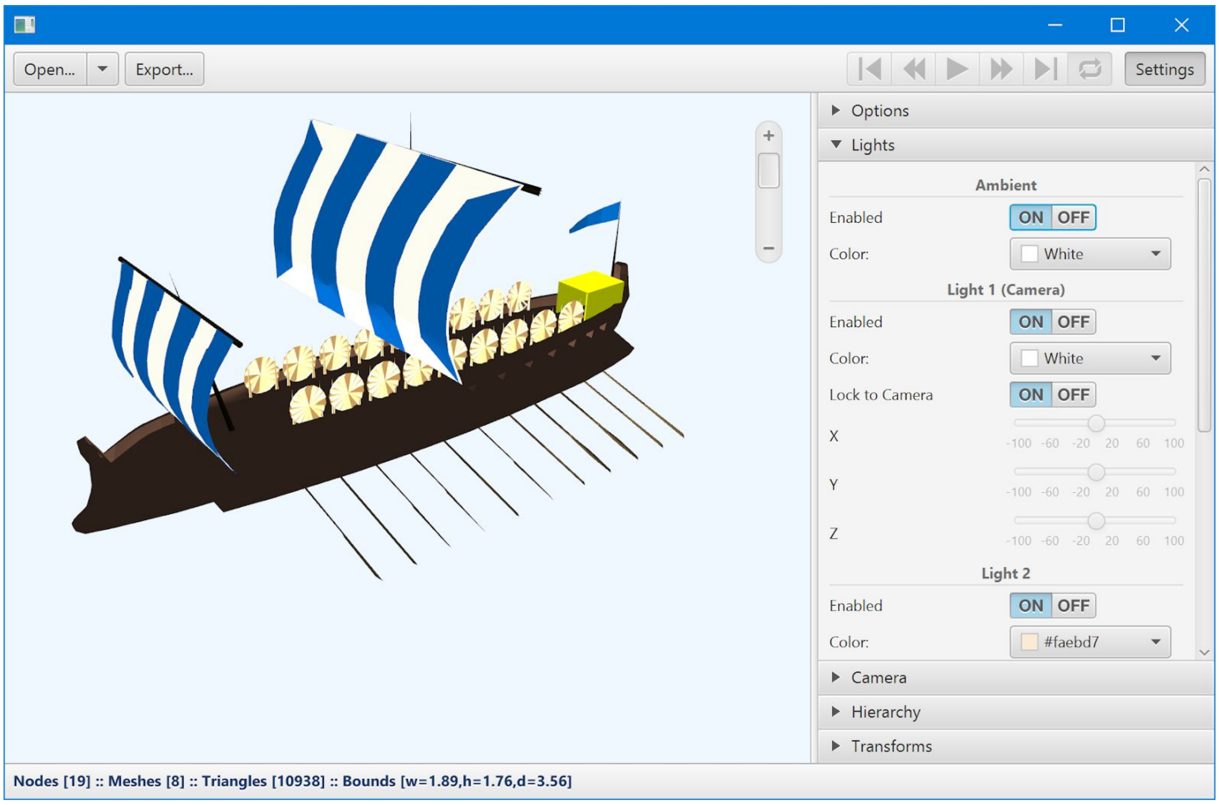
2.3.2 JavaFX Application & Visualisation

JavaFx supports basic 3D rendering. It is possible to display basic 3D shapes such as boxes and cylinders as well as custom meshes. JavaFx also supports orthographic as well as perspective cameras and lights. Surfaces can be textured with diffuse, specular and other texture maps. Besides the various textures that can be applied to an object there is very little control over the visual appearance of objects because custom shaders are not supported. Transparency is available but the transparent nodes are not automatically sorted by distance from the camera and therefore may require manual sorting.

This Software stack offers full control over the data representation of the model. But to display it, a conversion into a triangle mesh would be needed. Fortunately, there is a free library, JCSG, that supports constructive solid geometry operations in Java.

As with the data model there is complete freedom on how to design the tools for creating the model. This is on one hand more work but also offers total control over the result. Figure n gives an impression of the look and feel JavaFX applications generally have, using the built in UI components.

There exist libraries to convert pdf files to images with java. For example Apache PDFBox. The Image could then be applied as a diffuse texture onto a surface.



Ambient

Enabled [✓ ON] OFF

Color: White

Light 1 (Camera)

Enabled [✓ ON] OFF

Color: White

Lock to Camera [✓ ON] OFF

X -100 -60 -20 20 60 100

Y -100 -60 -20 20 60 100

Z -100 -60 -20 20 60 100

Light 2

Enabled [✓ ON] OFF

Color: #faebd7

```
Nodes [19] :: Meshes [8] :: Triangles [10938] :: Bounds [w=1.89,h=1.76,d=3.56]
```

Figure n: Generic JavaFX application with a three dimensional model. [m]

The most important advantages of choosing the JavaFX software stack include its widespread usage, which means there are a lot of libraries and community support, the availability of a CSG librarie and the complete freedom in terms of UI and tools developed. However, this freedom comes with the heavy price of having to implement basic tools and functionality that would be readily available in the other software stacks. Other disadvantages are limited shading capabilities and simple lines always having a thickness of one pixel.

2.3.3 Blender Plugin

Blender is a free and open source 3D creation software. It supports not only modeling but also many other tasks related to rendering images. Plugins for Blender are written in Python.

The modeling in Blender is done in a 3D viewport. Blender already provides many tools to manipulate a 3D model. The general approach is to start with a simple 3D primitive such as a cube and then adding or removing from that solid. There are also tools to manually edit faces.

The major advantage of Blender is that its plugins are written in Python, a widely used language. That means that there exist third party libraries to overcome general problems like converting pdf files to images. Blender comes bundled with its own Python interpreter and does not rely on the Python install on the operating system. Fortunately it is possible for plugins to install third party libraries for use inside Blender.

Another advantage of Blender is that the plugin API is well documented and due to its popularity has a large community supporting it.

The key disadvantage of Blender is that the general modeling approach is not as intuitive as in Sketchup. There are no simple push and pull tools available, for example. There are however tools for constructive solid geometry operations, such as adding and removing solids. Unfortunately these operations do not work reliably.

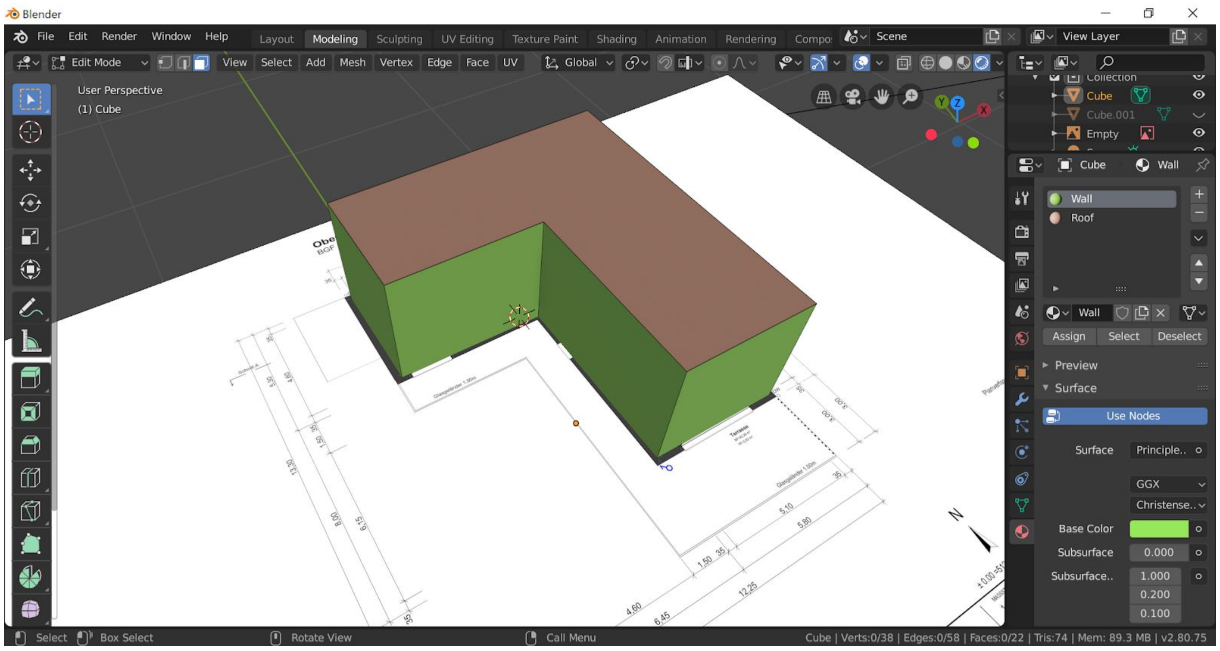

Blender
File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compo
Scene
View Layer
Edit Mode
View Select Add Mesh Vertex Edge Face UV
Global
User Perspective
(1) Cube
Collection
Cube
Cube.001
Empty
Cube
Wall
Wall
Roof
Assign Select Deselect
Preview
Surface
Use Nodes
Surface Principle..
GGX
Christense..
Base Color
Subsurface 0.000
Subsurface.. 1.000
0.200
0.100
Select Box Select Rotate View Call Menu
Cube | Verts:0/38 | Edges:0/58 | Faces:0/22 | Tris:74 | Mem: 89.3 MB | v2.80.75

Figure n: Blender with early Envelop related experiments. [m]

Blender, shown in figure n, has an overly complex UO for the simple task of modeling building envelopes. This can be mitigated to some degree by creating a custom workspace that only shows the relevant panels but things like the material creation remains overly complex. Finally, Blender offers only limited UI capabilities to plugins.

2.3.4 Summary Software Stacks

The following table n summarizes the evaluations from the three previous chapters.

	Blender Plugin	**JavaFX Application**	**Trimble SketchUp Extension**
Licence Fee	free	free	299$ per year per user
Flexibility	medium	high	low
Development Effort	medium	high	low
Enduser Complexity	high	low	medium

Table n: Summary of software stack evaluations. [m]

A Blender Plugin is free but the modelling workflow has significant differences to the envisioned Envelop workflow. This would require a significant amount of work. Additionally, its constructive solid geometry tools do not work reliably. As a whole, Blender seems too complex for this project, where the goal is a simple and efficient solution.

JavaFx has its limitations regarding the visual capabilities but it offers total freedom in the design of modeling tools. However, a large chunk of the allotted time would have to be used to implement basic things like moving the camera around and zooming. Therefore less time would be available to actually fine tune the application to the needs of the customer.

This leads to the third option, a SketchUp Extension. This would incur additional costs, as the pro version is needed. But SketchUp provides an already very intuitive modelling workflow that can be nicely incorporated into the new building envelope modeling workflow. Some areas, such as the importing of pdfs will need additional effort because existing Ruby libraries cannot be used easily. But key capabilities, such as the creation of additional tools, work seamlessly.

2.4 Conclusion

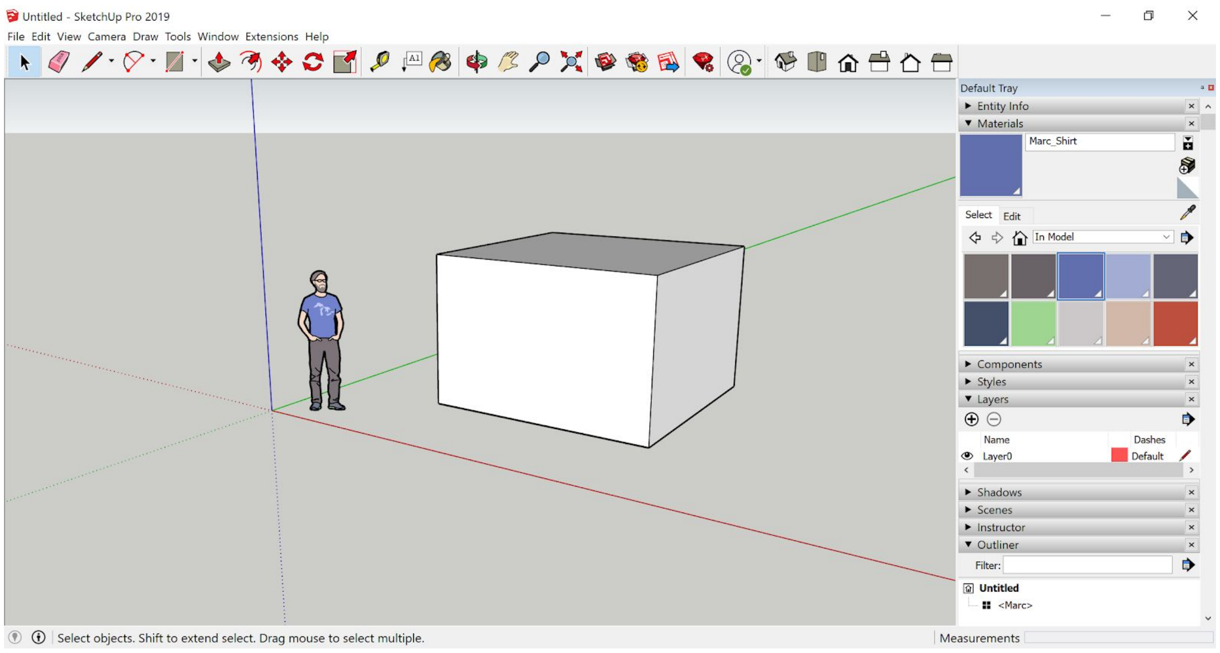
A Sketchup extension seems the most promising approach to develop a simple to use, fast and sufficiently precise way to calculate building envelope surface area totals by modeling buildings interactively.

This approach supports the use of solid constructive geometry operations, offers a simple interactive modeling environment on which Envelop can be built and has a sufficiently powerful extensions API. These major advantages offset the downsides of using a less popular and less familiar language as well as the licence fee. Moreover, Envelop will offset the Sketchup licencing fees quickly, but saving a lot of time.

3. Analysis of Sketchup

This Chapter takes a detailed look at how Sketchup works and what capabilities an extension has.

3.1 User Interface



Select objects. Shift to extend select. Drag mouse to select multiple.

Figure n: Sketchup UI without any extensions. [M]

Sketchups user interface consists of several parts, as shown in figure n. The main area is taken up by the view of the model. On the top are the menus and various toolbars. The toolbars can be moved around and customized as needed. On the right there is by default the tray docked. The tray is a customizable collection of panels that show the materials, hierarchy of the model and various other additional information. The bottom line is occupied by the statusbar. The statusbar displays general messages as well as detailed descriptions of the options the user has after he selected a specific tool. On the right of the statusbar is a special textbox called the value control box, short VCB. The VCB is used to display measurements like the length of a line being drawn. Moreover, the user can enter exact measurements if for example he wishes to create a line of ten meters.

3.1.1 Modeling

The general workflow of Sketchup consists of drawing a shape with a pen tool and then pulling it out to create a volume or pushing it inside an existing object to subtract a volume. The PushPull tool of Sketchup always creates new faces perpendicular to the original face. Unfortunately, operations of this tool are not performed using CSG.

Sketchup also supports assigning materials to the outside of faces. These materials are characterized by a color and a name and can be managed in the materials tray. Clicking on a material enables the material tool, which will assign that material to any clicked face.

3.1.2 Groups

Sketchup can group elements of the model hierarchy. Selecting several elements and creating a group has several effects. Initially, the geometry and faces of the grouped elements are merged to preserve volume and envelope surface area, but face divisions and internal geometry is lost.

Editing a group instead of individual elements requires the user to double click into the group. The groups outline is then highlighted and the user can only edit elements within the group. To leave the group editing mode a single click outside of the group suffices.

The CSG operations Sketchup offers require groups.

3.1.3 Further Information

Additional and more detailed information about the Sketchup UI can be found in the Trimble help center: https://help.sketchup.com/ja/sketchup/getting-started-sketchup.

3.2 API

The following chapters will provide a high level view at important areas of the Sketchup Ruby API for extensions.

3.2.1 Extensions

Sketchup can be enhanced with extensions written in Ruby. The Extension Warehouse contains a large number of readily available extensions submitted by developers. The extensions from the warehouse are reviewed and can be purchased and downloaded directly from within Sketchup.

It is also possible to install an extension without using the warehouse. An extension is simply a zip file containing all the necessary files with the filename extension renamed to '.rbz'. At the root of the extension archive must be a ruby file that registers the extension within Sketchup.
[https://help.sketchup.com/en/extension-warehouse/extension-development-best-practices]

3.2.2 Libraries

Ruby is a mature language and has many libraries, so called gems, available. But these ruby gems cannot be used as is in Sketchup. In a normal application the developer has control over all the software used but inside Sketchup extensions possibly loaded at the same time are developed by different people and version conflicts between separate versions of a ruby gem could easily occur. Another problem is that some ruby gems use native c/c++ code compiled for the specific platform it is running on and Sketchup does not ship with such a compiler built in. It is possible to use gems by manually compiling them but these compiled gems are then platform dependent and rely on the Ruby version shipped with Sketchup. They will break if a newer version of Sketchup ships with a newer Ruby version. It is however possible to include code from a ruby gem into an extension by copying and including the required source files. Of course, this only works if all dependencies can be included this way too.

3.2.3 Scene Hierarchy

The Sketchup Ruby API has several modules. One of them is the ‘Sketchup’ module, that contains all the classes specific to Sketchup. An extension can access the model that is opened in Sketchup by calling ‘Sketchup.active_model’.

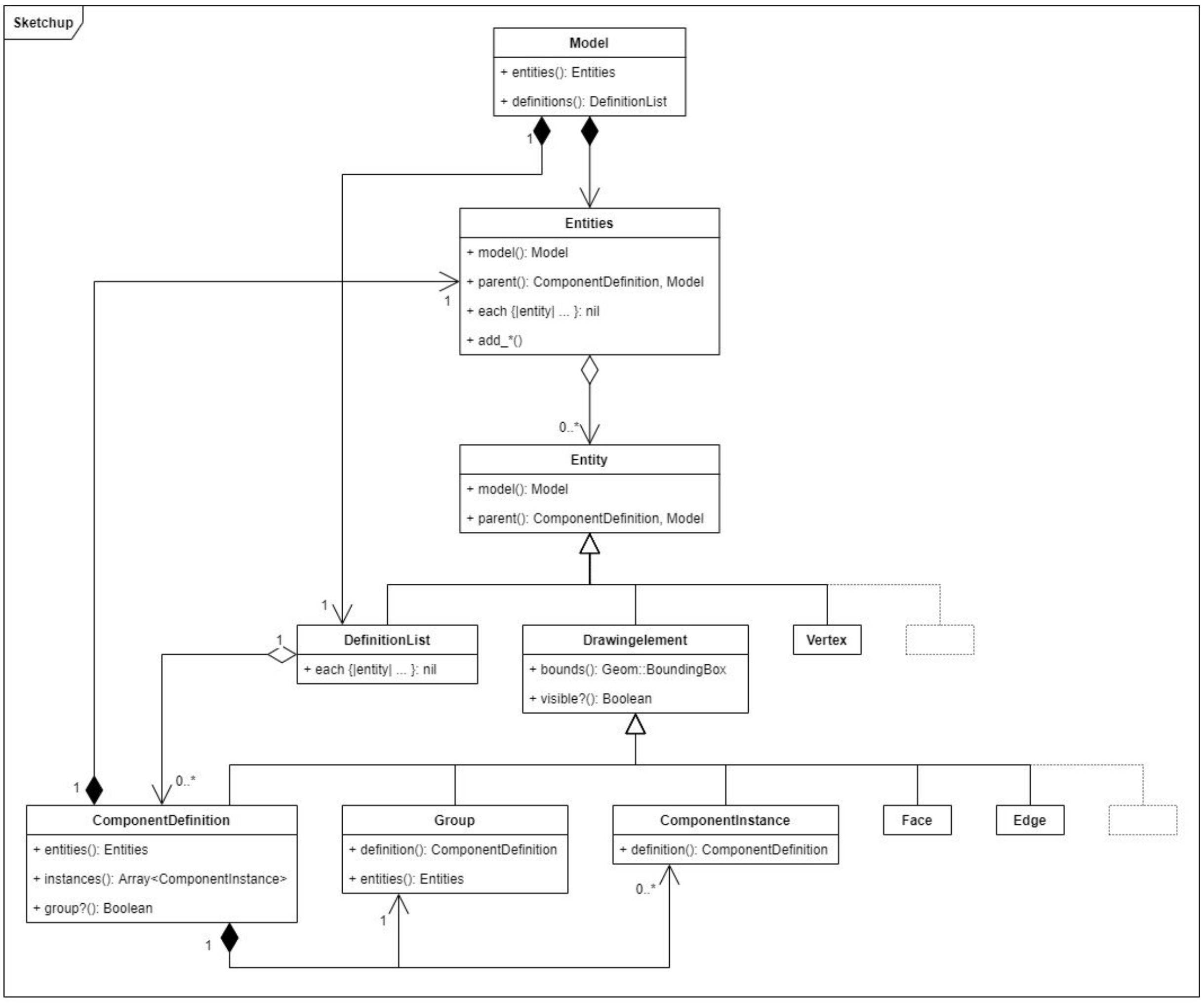

Sketchup
Model
+ entities(): Entities
+ definitions(): DefinitionList
Entities
+ model(): Model
+ parent(): ComponentDefinition, Model
+ each {|entity| ... }: nil
+ add_*()
Entity
+ model(): Model
+ parent(): ComponentDefinition, Model
DefinitionList
+ each {|entity| ... }: nil
Drawingelement
+ bounds(): Geom::BoundingBox
+ visible?(): Boolean
Vertex
ComponentDefinition
+ entities(): Entities
+ instances(): Array<ComponentInstance>
+ group?(): Boolean
Group
+ definition(): ComponentDefinition
+ entities(): Entities
ComponentInstance
+ definition(): ComponentDefinition
Face
Edge
1
0..*
1
1
1
1
0..*
1
1
0..*

Figure n: Sketchup Model architecture [https://ruby.sketchup.com/Sketchup/Model.html]

The ‘Model’ class has methods to access the defined materials, layers and also the geometry. Everything inside the Model inherits from ‘Entity’. Things that have a visual representation like faces and edges inherit from ‘DrawingElement’, that in turn inherits from ‘Entity’. The ‘entities’ method of the ‘Model’ class returns an ‘Entities’ class containing all ‘Entity’ in the root of the model hierarchy as seen in Figure n. ‘Entities’ is not a standard ruby collection but a Sketchup class that behaves like a collection and has some factory methods to add new entities.

Sketchup allows the user to group primitive entities together. The contents of a group are stored in ‘ComponentDefinition’ classes. All the ComponentDefinitions of a model are accessible via the ‘DefinitionList’ class that lists all component definitions. Sketchup has two ways of grouping things together. One is with the ‘Group’ class for easier handling. The other is for cases where several instances of the same object are needed. In that case the ‘ComponentInstance’ class is used. Several ‘ComponentInstance’ can reference the same ‘ComponentDefinition’ whereas every ‘Group’ has its own ‘ComponentDefinition’. The ‘ComponentDefinition’ class has, like the ‘Model’ class, an ‘entities’ method that allows access to the contained entities.

Navigating the model from root to leaf is done by recursively following the references to the definition of a ‘Group’ or ‘ComponentInstance’ and then its ‘Entities’. On the other hand moving from a leaf to the root is not always possible in an unambiguous way because the ‘parent’ method returns a ‘ComponentDefinition’ and in case of a component there could be multiple ‘ComponentInstance’s it could have originated from. In such cases the model must be searched for the specific entity from the root if the parent is required.

3.2.4 Picker

Picking is the act of calculating which objects or coordinates are at the screen position of the mouse. The Sketchup Ruby API has two distinct ways for picking.

Firstly there is the (InputPoint)[https://ruby.sketchup.com/Sketchup/InputPoint.html] class. Its main use is to get the 3D coordinates of the mouse position. The ‘InputPoint’ class has an instance method called ‘pick’ that takes a view, an x and y coordinate and optionally another InputPoint and sets the InputPoints state to the result. The resulting 3D coordinate is returned by calling ‘position’ on the InputPoint. Sketchup can use inferencing for calculating the result, which means it can snap to existing geometry and, if the optional InputPoint parameter is supplied, to points on the same axis as that point.

If the goal is to get an entity like a face below the cursor, then the ‘PickHelper’ is the way to go. The PickHelper class has methods to retrieve all the potential hierarchy leaf entities sorted by how good Sketchup thinks the pick is. These methods helpfully also return the path from the model root to the entities.
[http://www.thomthom.net/thoughts/wp-content/uploads/2013/01/PickHelper-Rev3.2-18-03-2013.pdf][https://ruby.sketchup.com/Sketchup/PickHelper.html]

3.2.5 Custom Tools

Extensions can create tools similar to Sketchups native tools. To do so, an extension has to define a class that defines a method for all the callbacks the custom tool should respond to. Some of the methods are non optional. By passing an instance of such a custom tool class to the models ‘select_tool’ the custom tool gets activated.

3.2.6 Observers

To let extension react to events in Sketchup many classes have a method to add an observer. An observer is just a class that implements certain callback methods. For example, to react to the deletion of an ‘Entity’, an instance of a class that implements ‘onEraseEntity’ has to be passed to the ‘add_observer’ method of the entity. When the entity is deleted the respective method is called.

Special care has to be taken when relying on observers to be around after a new file is opened. Objects of the new file do not have the observer attached automatically. To circumvent this problem an observer that listens to ‘onNewModel’ has to be added directly to ‘Sketchup.add_observer’.

3.2.7 HTML Dialog

The ‘HtmlDialog’ class allows extensions to create complicated and highly customizable windows. A HtmlDialog is a chromium browser running in a separate window, with HTML, CSS and Javascript sources provided by the developer.

To create such a dialog, the extension has to create a new instance of ‘HtmlDialog’ with an options dictionary as a parameter. The options declare the size, title and other settings of the dialog. After that the HTML content of the newly created dialog can be specified by calling ‘set_file‘ to specify a file or ‘set_html’ to specify a string. It is also possible to specify an URL as the content of the dialog. By calling ‘show’ on the dialog it finally turns visible.
The Ruby code can call Javascript functions from the ‘HtmlDialog’ and the Javascript code can call ruby callbacks previously specified on the ‘HtmlDialog’ object.

Data to be displayed has to be transferred from the Ruby code to the ‘HtmlDialog’. However, the ruby code cannot just call a Javascript function right after the creation of the dialog because ‘HtmlDialogs’ are created asynchronously. The HTML site has to load its content and Javascript code first. In order to transfer data, the Ruby code has to create the dialog, create a callback and then call the show method of the dialog. Now the dialog gets displayed and the Javascript code of the ‘HtmlDialog’ is executed. The Javascript components can now, after they’ve initialized themselfs, request the data by calling the predefined Ruby callback that in turn calls a Javascript function with the data as an argument. The process is visualized in Figure n.

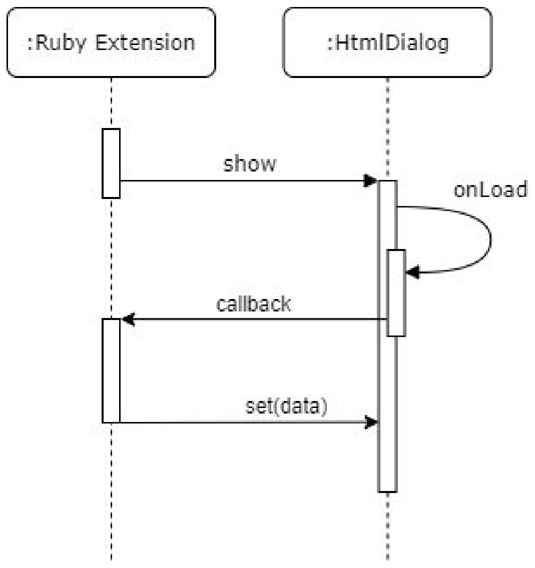


Figure n: Sequence diagram of Sketchup extension communicating with ‘HtmlDialog’. [m]

With this technique any data, for example base 64 encoded images or json strings can be transferred to the HtmlDialog.

A huge benefit of ‘HtmlDialog’s is that both on Sketchup for Windows and Mac runs the same browser and therefore the Javascript, HTML and CSS code developed for the ‘HtmlDialog’ works on both platforms. Moreover, Javascript libraries can be used freely within HtmlDialogs.

4. Modeling Building Envelopes with Envelop

This chapter will explain how to model building envelopes with the Envelop extension from a user perspective.

4.1 Overview

The following describes the general modeling workflow Envelop was optimised for. Note however, that any of these steps can be performed in any order. This flexibility allows the user to correct mistakes after the fact and choose his own even more efficient workflow as he becomes more experienced. The following list is intended as an overview, in order to understand the workflow from a higher level perspective. If more thorough explanations are needed, please refer to the ‘Quick Guide’ reference in chapter 4.2 or the detailed manual referenced in chapter 4.3.

1. Import plans
   The user uses the file explorer or drag & drop operations to add all required plans for the building he is trying to model.
2. Insert lowest floor plan
   The user selects, crops and inserts the lowest floor plan from the ones he has imported in step one.
3. Trace Floor
   Using the pen tool, the user traces the outline of the lowest floor. If the floor is square, two clicks are quired, otherwise the user has to simpli click each corner once.
4. Insert side views
   The user inserts all side views, correctly positioned and scaled relative to the already created floor outline. Whenever a new plan has to be positioned and scaled, all the other ones are hidden. This ensures the user orientates all the plans to the same reference point - the model.
5. Pull up floor
   Using the simple push and pull tool, the user can pull up the outlined floor in order to create the full volume of the first floor.
6. Create remaining floors
   The following steps are repeated, until all of the floor volumes except the roof are created.
   a. Inset next floor plan
      The user inserts the next floor plan, positioning it on top of the previous floor volume.
   b. Fix floor outline
      If the new floor has a different outline than the previous, that user must correct that outline using the pen tool.

   c. Pull up floor
      The push and pull tool is used by the user to create the volume of the current floor.
7. Create roof
   Flat roofs can be created like floors. The following steps are therefore intended for more complicated roof shapes.
   a. Trace the roof from a side view
      Using the pen tool the user traces a side view of the roof, just like the floors were traced previously.
   b. Pull the rood over the building
      To create the roof volume, the user uses the push and pull tool to drag the side view of the roof across the building.
   c. Fix roof
      For even more complicated roof shapes, the above two steps might have to be repeated. Moreover, the user can remove volumes from the existing roof, using the pen tool to draw its outline and using the push and pull tool in its alternate mode.
8. Create dormers
   User draws the outlines of all dormers using the pen tool. Then each dormer can be completed by double clicking with the push and pull tool into these outlines.
9. Draw windows and other details
   The user draws any windows or wall separations that are required, for example if a single wall requires two different materials due to half of the wall being below the ground. All of this is done using the pen tool.
10. Apply materials to faces
    Using the materialisation window to the right, the user can click the material he would like to apply and then all the surfaces that should have that material.
11. Scale building
    Using the scale tool, the user should trace a known distance on the building or on any of the plans. After entering that distance, the model is scaled.
12. Set north direction
    By default north is set so that up on the floor plans correspond to north. If this is not correct, the user can set north using the orientation tool.
13. Export surface area
    The user can now open the building envelope surface area totals table in order to view and copy the results.

This workflow is intended to maximize efficiency and usability, by minimizing the number of different tools that have to be learned and by following an interactive, step by step construction.

4.2 Quick Guide

Part of the Envelop extension is a ‘Quick Guide’ in the form of a wizard. It is best viewed within Sketchup, as it features a task list with additional information on how to achieve each step and short videos, showing how to perform these actions.

To view this guide, please install Envelop as described in the manual in chapter 8.2 and open the wizard. The installation guide can also be found in the repository in chapter 8.6.

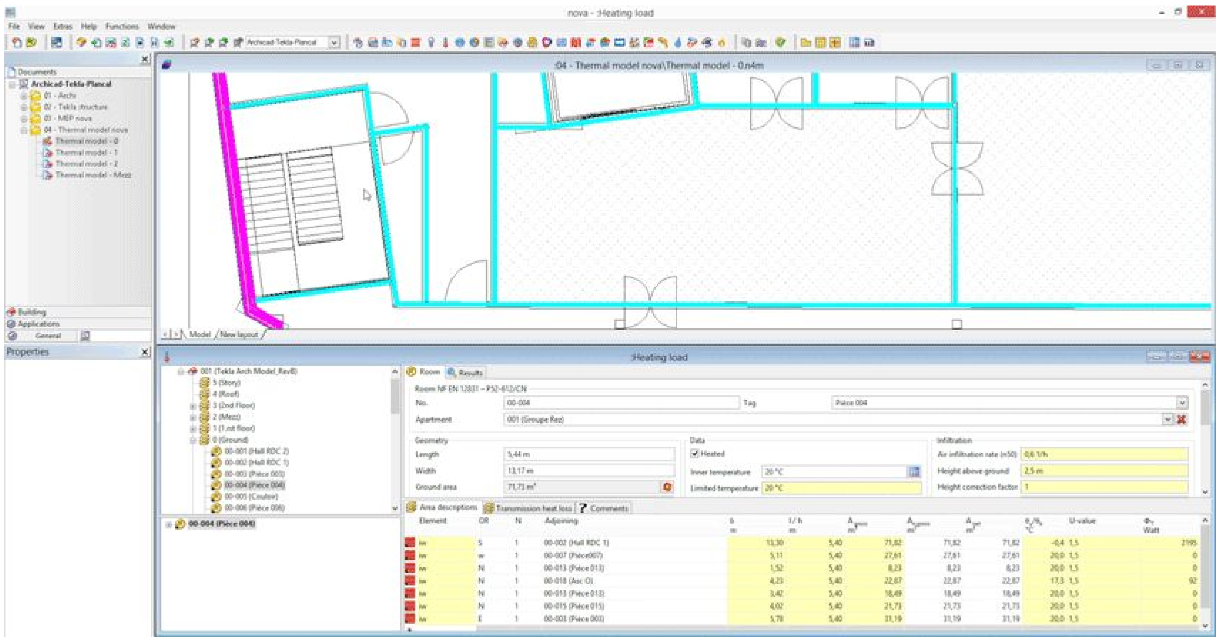
4.3 Detailed Manual

A detailed manual for all the functionality Envelop offers has been written. It covers installation, as well as usage - including how to model outside of the Envelop tools and how to recover broken models. Please consult chapter 8.2 to view this manual.

5. Productivity and Usability Evaluation

The following subchapters will validate that Envelop, using the workflow described in the chapter 4.1 and the in-app wizard, fulfills the precision and efficiency requirements.

5.1 Previous Workflow



Room NF EN 12831 – P52-612/CN

No.	00-004	Tag	Pièce 004
Apartment	001 (Groupe Rez)		

Geometry		Data		Infiltration	
Length	5,44 m	✓ Heated		Air infiltration rate (n50)	0,6 1/h
Width	13,17 m	Inner temperature	20 °C	Height above ground	2,5 m
Ground area	71,73 m²	Limited temperature	20 °C	Height correction factor	1

Element	OR	N	Adjoining	b m	l / h m	A_{gross} m^2	$A_{Opening}$ m^2	A_{net} m^2	θ_u/θ_a °C	U-value	Φ_T Watt
iw	S	1	00-002 (Hall RDC 1)	13,30	5,40	71,82	71,82	71,82	-0,4	1,5	2195
iw	w	1	00-007 (Pièce007)	5,11	5,40	27,61	27,61	27,61	20,0	1,5	0
iw	N	1	00-013 (Pièce 013)	1,52	5,40	8,23	8,23	8,23	20,0	1,5	0
iw	N	1	00-018 (Asc O)	4,23	5,40	22,87	22,87	22,87	17,3	1,5	92
iw	N	1	00-013 (Pièce 013)	3,42	5,40	18,49	18,49	18,49	20,0	1,5	0
iw	N	1	00-015 (Pièce 015)	4,02	5,40	21,73	21,73	21,73	20,0	1,5	0
iw	E	1	00-003 (Pièce 003)	5,78	5,40	31,19	31,19	31,19	20,0	1,5	0

Figure n: Plancal Nova UI, with a building plan visible. [M]

Previously, the buildings had to be fully modeled using Plancal Nova. This included detailed information regarding windows, doors and materials. Not only did Plancal Nova require much more information than was necessary to perform the required measurements, entering this information was also very slow and burdensome, requiring the navigation of many layers of menus and having very little support for a visual and interactive modeling workflow. The above screenshot of Plancal Nova in figure n gives a sense of the kind of old but powerful software that it is.

Having to enter more information than would be necessary and having to use work with Plancal Novas user interface meant that even for very experienced users, going from two dimensional building plans to calculated building envelope surface area totals, required about an hour to an hour and a half of concentrated working.

5.2 Test Setup

In the repository referenced in the appendix 8.5 there are several plan collection examples. For each of these, Envelop was used to make a model and determine envelope surface area totals. Where possible, these totals were compared with the numbers provided by the customer using the previous workflow. This was done in order to verify the precision of Envelop.

For each of the models, the overall time required, number of clicks and tool switches were recorded. The essential metric here is the overall time required, to validate the efficiency increase compared to the previous workflow. The clicks and tool switches were recorded to gain further insights into the bottlenecks of the new workflow and to better understand the differences of buildings with varying complexity.

The models and measurements generated during the testing can be found in full alongside the source plans in the repository in appendix 8.5.

5.3 Test Results

The next two subchapters will present and evaluate the results of the testing procedure described above. These chapters will contain summaries of the individual test results, the complete testing artifacts can be found in appendix 8.6.

5.3.1 Precision

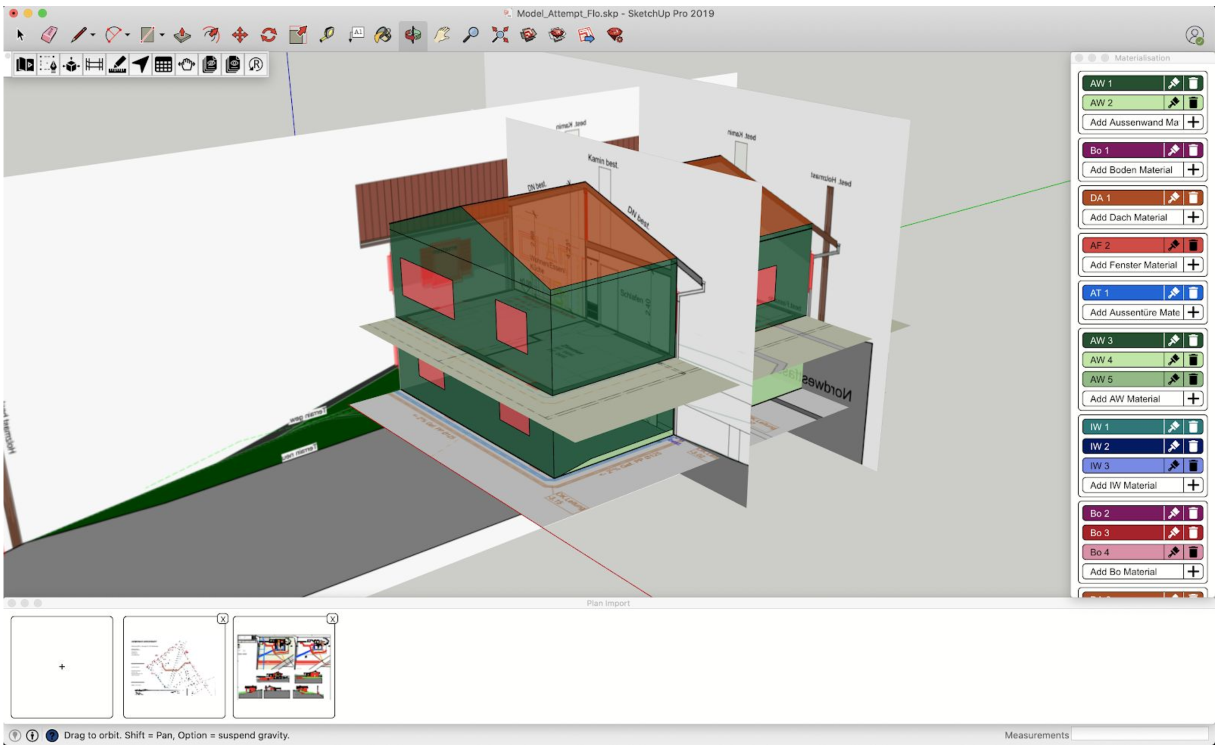


Figure n: Building P19101891001 modeled in Envelop. [m]

The customer provided four example buildings with his area calculations using the old workflow. These buildings were modeled and measured using Envelop, as for example with building P19101891001 in figure n. The table n contains all the buildings the customer provided with measurements.

Model	**Total Area Provided**	**Total Area Envelop**	**Percentage Error**
P19101721004	626 m^2	600 m^2	~ 4 %
P19101771001	546 m^2	590 m^2	~ 8 %
P19101891001	402 m^2	399 m^2	~ 1 %
P19102081001	317 m^2	325 m^2	~ 3 %

Table n: Verification of sufficient accuracy with Envelop compared to old workflow. [m]

These models and measurements were analysed together with the customer. It was determined that the Envelop has sufficient precision, as errors in these percentage ranges occur naturally in the process of releasing a theoretical construction plan. Moreover, the larger deviations were caused due to difficult plans and insufficient modeling detail, such as with stairwells.

5.3.2 Efficiency

In order to verify the efficiency and speed of modeling and measuring a building with the Envelop extension, all the buildings the customer provided plans for were modeled and measured according to the test setup in chapter 5.2. The results of these testings are presented in the following table n.

Model	**Customer Estimated Old Workflow Time**	**Envelop Time**	**Envelop Clicks**	**Envelop Tool Switches**
P19101721004	1h - 1.5h	12:07	266	49
P19101771001	1h - 1.5h	08:50	244	49
P19101891001	1h - 1.5h	10:09	221	51
P19102081001	1h - 1.5h	10:47	230	54
Average	1h - 1.5h	10:28	240	51

Table n: Efficiency measurements of Envelop compared to the old workflow. [M]

TODO

6. Technical Examination of Envelop

In this chapter the developed Envelop extension will be presented from a technical perspective. The goal is both to offer insight into the technical achievements by Envelop as well as to enable the continued development and support of Envelop.

6.1 Architecture

6.1.1 Overview

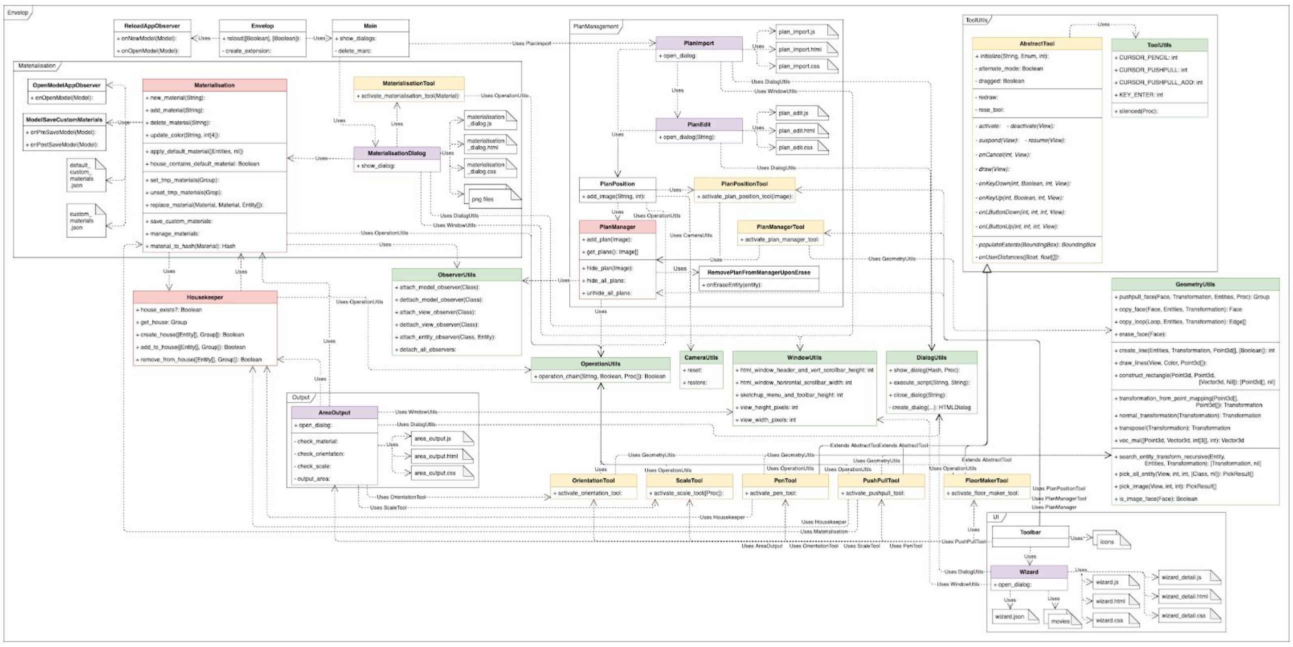


Figure n: Envelope Architecture Overview. The full scale version is in the appendix 8.3. [M]

The diagram in figure n provides an overview of the components, their most important functions and their relationships that make up the Envelop extension. If not indicated otherwise, this diagram is to be read using UML conventions.
There are four major types of components, which are color coded in the above diagram. Red marks important centerpieces, which manage and modify key data. Violet marks dialogs, which are windows of any kind that are shown to the user. Yellow indicates custom tools that the user can use to interact with the model. Finally, utilities are colored green.
The diagram does not list any attributes, instead all listings are functions. No parentheses indicates no parameters and no type indicates no return values. The functions shown here represent only a selection of the functions of each component. This selection was made with the goal of increasing the readability and usability of the diagram.
Because the extension is written in Ruby, most of the relations are simply usings - components generally don't instantiate each other and instead just call functions. Moreover, there is only one abstract class that is inherited from.

The extension can logically be divided into six major areas of concern. These are extension initialisation, house management, plan management, materialisation, area output and modeling. In the next few chapters the basic workings of the first four of these areas of concern will be discussed in order. Modeling can be understood as a collection of custom tools.

As mentioned earlier, such custom tools, as well as custom dialogs and utilities are also important architectural features of Envelop. Therefore these patterns will be explained in more detail as well.

6.1.2 Envelop Main

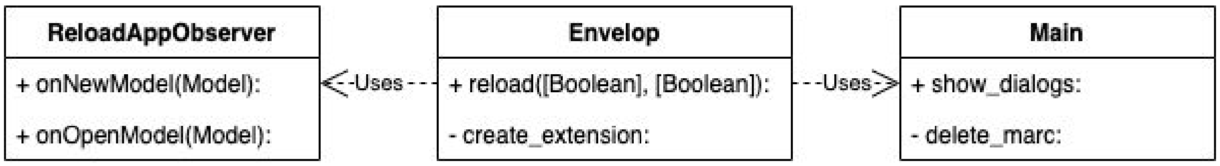


Figure n: ‘Envelop’ and ‘Main’ Module with their primary functions. [M]

The above figure n displays the core modules responsible for loading and initialisation of the extension. The ‘Envelop’ module is loaded by Sketchup and has three primary functions. Most importantly, it registers the extension with Sketchup using the ‘create_extension’ function. It also kicks off the loading and initialization of the remaining code through the ‘Main’ module upon being loaded. It’s third functionality is an extension reload, which was very important to reduce testing times during the development of Envelop. It enabled the testing of new code without restarting Sketchup. In the released version however, this reload functionality is used when new files are created or opened to ensure a clean internal state for each file. The class ‘ReloadAppObserver’ is used to catch the events which have to induce an extension reload. This observer class is attached and managed with the ‘OberserverUtils’ described in chapter nnn.

The ‘Main’ module completes the initialization by importing all remaining code, deleting Marc from the model using the private ‘delete_marc’ and opening the default windows with ‘show_dialogs’. This opens the materialisation dialog and the plan import.
Marc is the figurine that is added to every model to help users scale their models correctly. Since Envelop comes with its own custom scaling tools, this figurine is not needed.

Any remaining initialization is done on a per module basis and is performed by each module on its own. Ruby code is executed upon being loaded, thus initialization can happen automatically.

6.1.3 Housekeeper

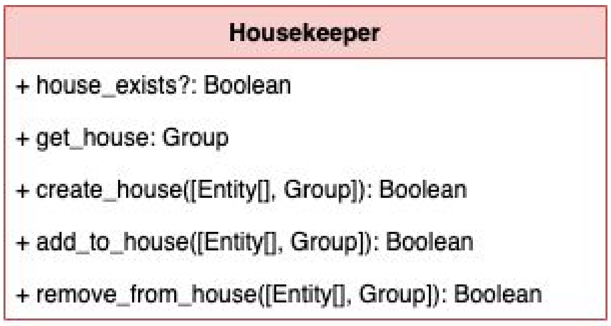

Housekeeper
+ house_exists?: Boolean
+ get_house: Group
+ create_house([Entity[], Group]): Boolean
+ add_to_house([Entity[], Group]): Boolean
+ remove_from_house([Entity[], Group]): Boolean

Figure n: The ‘Housekeeper’ module with its primary public functions. [M]

The ‘Housekeeper’ module in figure n is responsible for managing and exposing the most important internal state of the extension. That is, it keeps track of the Sketchup Group that contains the house that is being modeled. It does so with an internal Ruby reference, because that is faster than searching the entire model, whenever that group is needed. However, this reference is very brittle and can break in many ways. For example saving and loading files and undoing or redoing actions.

This problem is solved using the ‘AttributeDictionary’ class and the related functions provided by Sketchup. They allow extensions to save arbitrary data into the model hierarchy. This data is then preserved correctly in files and during any user actions. Moreover, changes to these attributes are, like any other change to the model, properly synchronized and are thus much safer than modifying internal state in functions that are executed many times each second.

These attributes are used by Envelop in various ways, for example in tools, as covered in chapter nnnn. The ‘Housekeeper’ uses them to mark the house group and recover the reference if it breaks for any reason. The recovery is done with an internal method that searches the model hierarchy recursively for the previously set attribute.

The ‘house_exists’ and ‘get_house’ functions expose the internal reference to the rest of the codebase. However, in order to modify the house group, the ‘add_to_house’ and ‘remove_from_house’ functions should be used. These functions perform additional checks. For example, arguments are ensured to be non-nil and the group is only changed if the volume remains manifold. This is done using boolean volume operations as described in chapter 2.2.1. Moreover, these functions preserve any and all already assigned materials and surface divisions, by assigning unique new temporary materials to all faces and restoring the original materials after the boolean operation. This is done using the ‘Materialisation’ module described in chapter 6.1.5.

Finally, the ‘create_house’ function creates and saves a group as the house, if the supplied argument is already manifold.

6.1.4 PlanManager

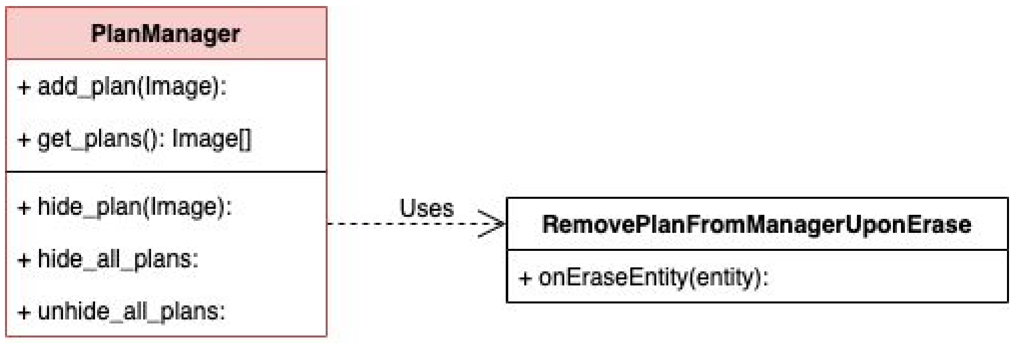


Figure n: the ‘PlanManager’ module and its RemovePlanFromManagerUponErase class. [M]

The ‘PlanManager’ module in figure n concerns itself, as the name suggests, with the plans added to the model. Its functionality is two fold: on one hand it keeps track of plans added with ‘add_plan’ and returns a list of all plans with ‘get_plans’. On the other hand, it can hide individual plans or all plans with ‘hid_plan’ and ‘hide_all_plans’ respectively. Of course, plans need to become visible again, which is done with ‘unhide_all_plans’.

Just like in the ‘Housekeeper’ module, the plans are tracked with Ruby references as well as attributes within the model, to maximise both performance and stability of the system. To reduce the need to recursively search the model for plans to ensure none of them have been deleted since the previous access the class ‘RemovePlanFromManagerUponErase’ is used. It is instantiated and attached to each plan as an EntityObserver. If that entity, in this case a plan, is erased from the model for any reason, it removes itself from the PlanManager. This observer, like any other that is part of the extension, is managed using the previously mentioned ‘ObserverUtils’ module.

Plan importing and editing is explained in more detail in the respective tool and dialog chapters, that is nnnn, nnnn and nnn.

6.1.5 Materialisation

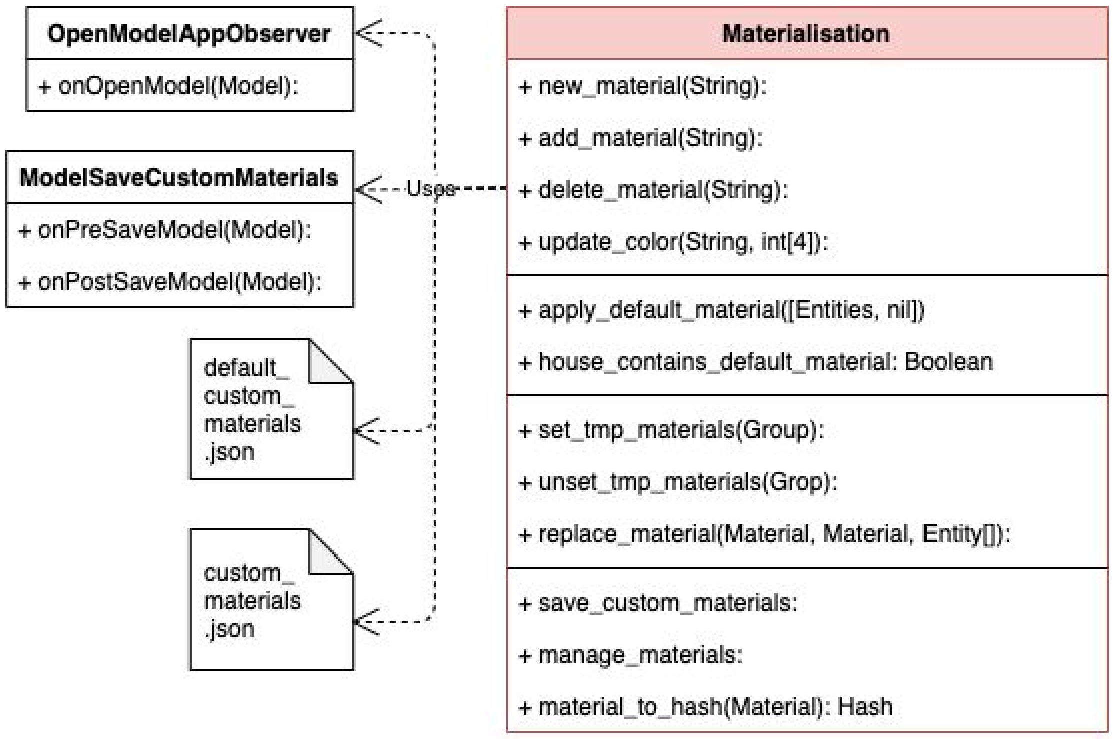

OpenModelAppObserver
+ onOpenModel(Model):
ModelSaveCustomMaterials
+ onPreSaveModel(Model):
+ onPostSaveModel(Model):
default_custom_materials.json
custom_materials.json
Uses
Materialisation
+ new_material(String):
+ add_material(String):
+ delete_material(String):
+ update_color(String, int[4]):
+ apply_default_material([Entities, nil])
+ house_contains_default_material: Boolean
+ set_tmp_materials(Group):
+ unset_tmp_materials(Grop):
+ replace_material(Material, Material, Entity[]):
+ save_custom_materials:
+ manage_materials:
+ material_to_hash(Material): Hash

Figure n: The 'Materialisation' module with its functions and the observers it requires. [M]

The 'Materialisation' module pictured in figure n manages materials. Material management is quite involved, therefore it is split into four groups of functions to ease its understanding.

Firstly, there are the functions to create new materials, add new one based on an existing material, delete materials and change their colors. All of these functions rely on a string string id to identify the material to delete, to change the color of, or based on which to generate a new one. Such new materials use a sligh color deviation of the source material and use the same name with the next unused number. For example, if no other materials exist, a green 'ADA 1' material as source material generates 'ADA 2' in lime green.
Materials must only be changed using these functions, as each material carries an extensive list of custom attributes that must be kept up to date in order to ensure 'Materialisation' continues to work correctly. Moreover, 'delete_material' warns the user if he is about to delete a material that is still in use. Lastly, after each change the list of custom materials is saved using 'save_custom_materials'.

The management of the default material for parts of the house is another responsibility of the 'Materialisation module'. The Sketchup default material is opaque, however, Envelop users should be able to peer into their model, to more easily spot mistakes. Thus, each surface added to the house has its surfaces set to the Envelop default using 'apply_default_material'. In order to warn the user if he tries to print the results without assigning a custom material to each face of the house, there is also the function 'house_contains_default_material', to detect if there are any surfaces with the default material left.

The third functionality group supports other modules in their tasks. The 'Housekeeper' module for example needs to set and unset temporary materials as mentioned in chapter

6.1.3. This is done using the ‘set_tmp_materials’ and ‘unset_tmp_materials’ functions. There is a third utility function ‘replace_material’, that is mostly used by the ‘Materialisation’ module itself. It replaces all instances of one material with another. Like most operations affecting the whole model hierarchy, this has to be done recursively.

Finally, there is the issue of saving custom materials so that a user can use his materials in new models as well. At the same time, files must be usable on other machines too, thus custom material must be saved locally as well as in the save files. This leads to another problem, as these two sets of materials must be joined - but several kinds of conflicts can occur during this join-operation. There are three cases:

- The material name exists only in one of both sets. In this easy case it is added to the final list of materials.
- The material name exists in both sets, but all additional information, such as its color is identical. Again, an easy case as the material saved in the file can be used, so that no surfaces must be changed. However, such materials are marked as not originating from the save file.
- The material name exists in both sets and there is additional conflicting information. In this case, the material saved in the file has priority, as there may be surfaces with that material in the model. These materials are marked as originating from the save file.

The join operations to resolve the above cases are further complicated by the fact that Sketchup automatically saves and loads materials used in a model in that model’s save file. Lastly, the question arises how to save changes the user makes to the resulting list of materials. The policy Envelop uses is optimised for usability and user expectations. As such, changes made to materials that originate from the model save file are saved to that file but not locally. Vica versa, changes made to local materials are only saved locally.

To achieve the above described behaviour, ‘Materialisation’ uses the ‘ModelSaveCustomMaterials’ observer, to mark all materials used as originating from the save file, just before Sketchup automatically writes them into the file and to remove these marks shortly after again. This doesn’t happen if the materials are saved locally using ‘save_custom_materials’. The material set joining happens automatically if a file is opened, at which point the ‘OpenModelObserver’ calls ‘manage_custom_materials’. At this point in time, the custom materials saved in the file are already added. The function has six steps.:

1. Unused materials are removed from the model.
2. The default material is added to the model, if it doesn’t already exist.
3. The locally saved materials in ‘custom_materials.json’ are added to the model. If none exist, the file is initialized with ‘default_custom_materials.json’.
4. Identical materials are merged as described above.
5. Conflicting materials that originate from the local machine are hidden.
6. The materialisation dialog, see chapter 6.3.1, is reloaded so that it displays the new final list of materials.

Saving and displaying the materials require a Json serialisation, which is available with ‘material_to_hash’.

6.1.6 AreaOutput

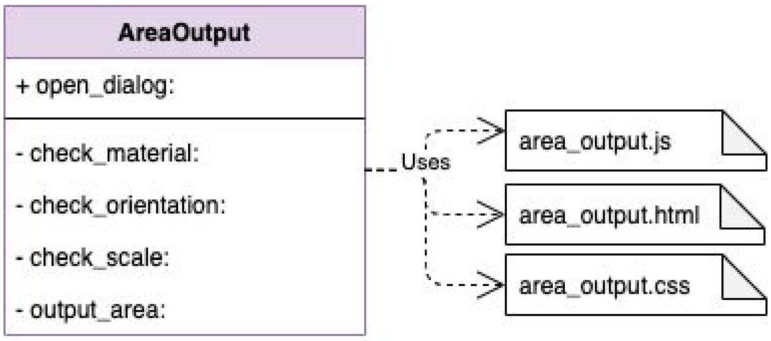


Figure n: AreaOuput dialog with required external files and a selection of functions. [M]

The ‘AreaOuput’ module is a custom dialog and as such uses ‘DialogUtils’, ‘WindowUtils’ and external HTML, Javascript and Css source files. This pattern of custom dialogs will be discussed in further detail in chapter 6.1.8 ‘Custom Dialogs’.

The only public function ‘open_dialog’ uses a chain of private functions to ensure that the model has no default materials left with ‘check_material’, its orientation has been set with ‘check_oriantation’ and finally that it has been scaled with ‘check_scale’. If any of these conditions fail, the user is prompted to fix them and the appropriate tool is started. If all of these conditions are met, the final tally of surface areas is opened using ‘ouput_area’.

To be able to display the surface area totals, this module must perform several calculations. For each surface that is part of the house, it’s area is translated into the unit the user has selected as the default for the Sketchup model, it is sorted into the correct user defined orientation, then its orientation is localised into german and finally added to the resulting table. Once these calculations are finished, the resulting table is sent to the Html dialog as discussed in chapter 6.1.8.

6.1.7 Custom Tools

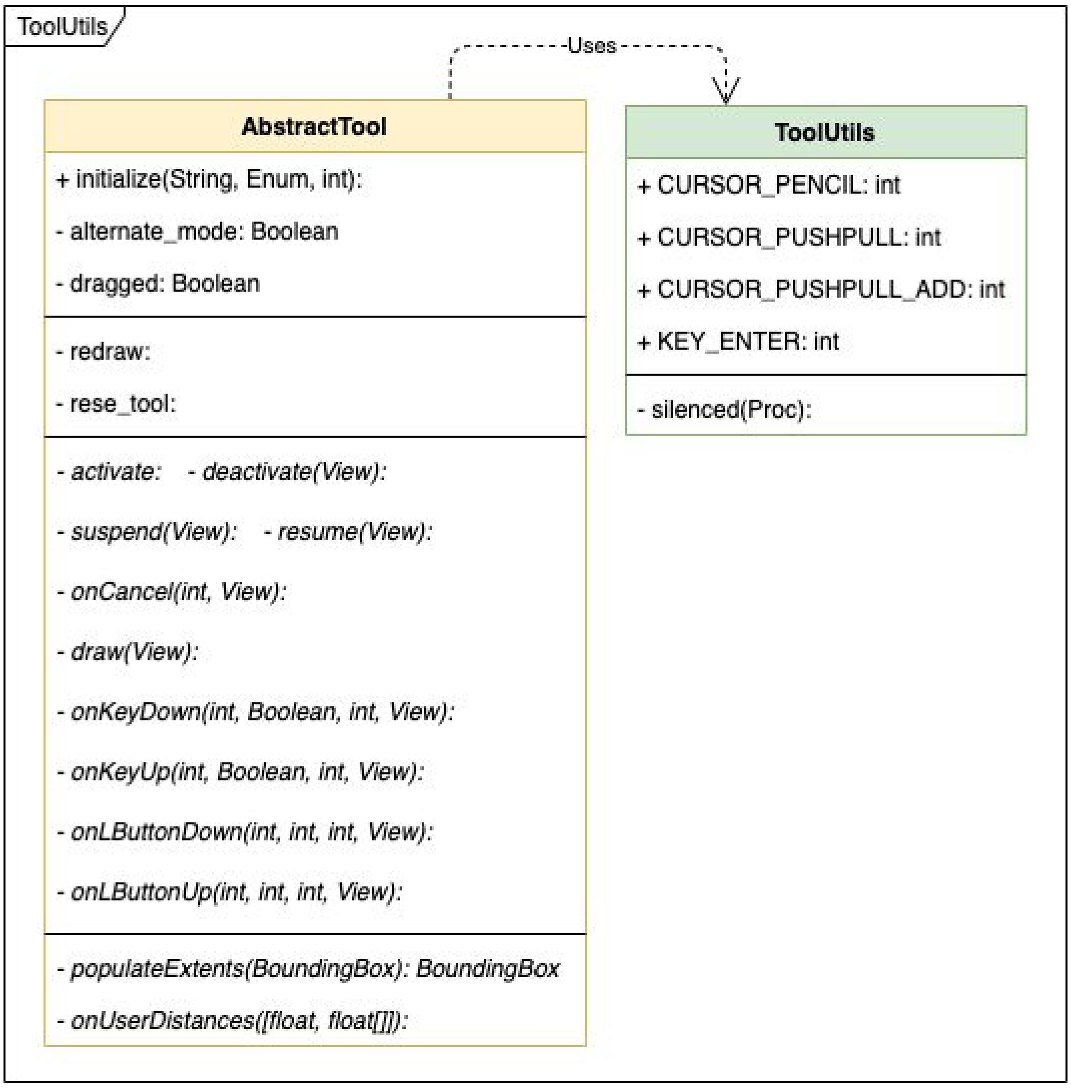

ToolUtils
Uses
AbstractTool
+ initialize(String, Enum, int):
- alternate_mode: Boolean
- dragged: Boolean
- redraw:
- rese_tool:
- activate:
- deactivate(View):
- suspend(View):
- resume(View):
- onCancel(int, View):
- draw(View):
- onKeyDown(int, Boolean, int, View):
- onKeyUp(int, Boolean, int, View):
- onLButtonDown(int, int, int, View):
- onLButtonUp(int, int, int, View):
- populateExtents(BoundingBox): BoundingBox
- onUserDistances([float, float[]]):
ToolUtils
+ CURSOR_PENCIL: int
+ CURSOR_PUSHPULL: int
+ CURSOR_PUSHPULL_ADD: int
+ KEY_ENTER: int
- silenced(Proc):

Figure n: ‘AbstractTool’ class with its abstract functions and the ‘ToolUtils’ module. [M]

Custom tools, such as the modeling tools, should inherit from ‘AbstractTool’ pictured in figure n and override necessary functions. To reduce the need to rely on static values, the constants in the module ‘ToolUtils’ should be used.

Instead of just implementing a class that fulfills the requirements for a custom tool defined by Sketchup, inheriting from ‘AbstractTool’ is much more convenient. Common actions that all or most tools need to perform are already covered by inherited code and the concrete class can focus on its actual functionality. However, because the abstract functions perform important updates to the internal state of the concrete tool, overridden functions must be called from the concrete tool using ‘super’. ‘AbstractTool’ simplifies modeling a tool with phases, alternate modes with keys, supports dragging as well as clicking start and end points, simplifies the handling of mouse coordinates, the handling of user input text, the drawing of previews and finally also the instantiation and management of internal tool state. Any redraws required by the above are performed, massively reducing the need for redraw calls in concrete tools. All of these topics will be covered in the following paragraphs.

For many tools it is useful to model their logic in phases. For example, an initial phase, one after the first user click and one after the second, for a tool that moves an object using two clicks. ‘AbstractTool’ supports this model, if it is provided with a Enum of phases as the second argument to ‘initialize’. It then treats the first entry of the enum as the initial phase, for the purposes of tool resetting. The current phase can be accessed and written to with ‘@phase’. This phase-model is also very useful for concrete tools to show different status texts at the bottom right. ‘AbstractTool’ expects concrete tools to implement ‘set_status_text’ in which the status text is updated. This function is then called before every redraw.

Most tools also feature alternate modes. For example this could be free moving of an object versus moving it along an axis. ‘AbstractTool’ fully supports this, with both keeping an alternate mode key pressed and being in alternate mode while the key is pressed and with pressing the key once to enter alternate mode until it is pressed again. The keys detected are Alt, Command, Control and Shift. The concrete tools can check for alternate mode with ‘@alternate_mode’. These keys are detected in ‘onKeyDown’ and ‘onKeyUp’ and ‘@alternate_mode’ is updated accordingly. The system time is also saved and compared in order to compare with ‘TAP_HOLD_THRESHOLD_MS’ - if the time difference is below this constant the press is treated as a single click. Otherwise it is treated as press and hold.

‘AbstractTool’ supports dragging and clicking in a similar fashion, by recording system time in ‘onLButtonDown’ and ‘onLButtonUp’ and comparing it against ‘CLICK_DRAG_THRESHOLD_MS’. Concrete tools can then look at ‘@dragged’ in their implementation of ‘onLButtonUp’ to determine if the user just ended a drag.

Sketchup requires that custom tools map the cursor to model space in a ‘onMouseMove’ function. ‘AbstractTool’ covers this too and concrete tools can access the current model space input point with ‘@ip’, which will always be the last valid model space input point determined. Invalid input points, such as when the mouse is outside of the Sketchup window, will not be written to ‘@ip’. ‘AbstractTool’ manages the input point tooltip and therefore concrete tools do not need to concern themselves with this

Sketchup allows the user to input exact distances via text for most of the built in tools, with the VCB field. ‘AbstractTool’ simplifies supporting this feature too. Instead of having to override ‘onUserText’ and having to parse string values, concrete tools can implement ‘onUserDistance’ which abstract will call with either a single float or a list of floats, representing the values entered by the user. The values are split using regex and parsed using internal Sketchup functions.

Previews are a central concern of any custom tool, as they are required to visualize for the user what will happen if he finishes the interaction. This is essential for high usability. Apart from actually drawing the preview, which is left to the concrete tool, Sketchup also needs to be told the location and size of the preview and to be redrawn. Usually this is done by implementing ‘getExtents’ and invalidating the active view. This is somewhat verbose though, so concrete tools can simply call ‘redraw’ when after having drawn the preview and can

implement ‘populateExtents’ which already comes with a Extends object instantiate, which must simply be supplied with values.
Another issue showing previews is adding elements to the model temporarily and removing those elements very shortly after again. For example, if the preview is updated on each mouse move elements end up being added many dozens of times each second. Updating internal references to those elements in Ruby has led to significant performance issues and random crashes. Thus, preview elements must be marked as such using the attribute dictionaries provided by Sketchup. Instead of Ruby references, the model must be searched recursively when the elements should be removed. As mentioned in chapter 6.1.3, this method resolves any concurrency issues.

‘AbstractTool’ has a significant amount of internal state, which must be initialised and reset at various points in time, for example, if the user hits the escape key. This happens within ‘reset_tool’. Concrete tools are expected to reset their own state in their overriding version of ‘reset_tool’ and then call ‘super’.

For convenience, ‘AbstractTool’ also contains trace logging statements in order to simplify debugging.
Most custom tools will also want to use the utility function in OperationUtils, which will be described in detail in chapter nnn.

Using this abstract base class for feature rich custom tools not only simplifies their development, but it also ensures a tool palette with consistent usability features on each tool. Moreover, it eliminates a lot of easy to make mistakes and increases the readability of custom tools. As such it is a key architectural feature of Envelop.

6.1.8 Custom Dialogs

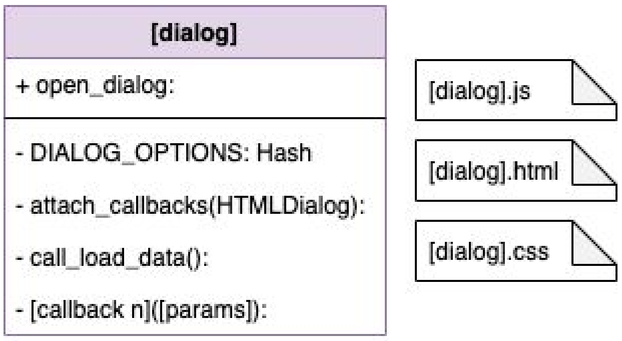


Figure n: Generic custom dialog, with the required functions and external components. [M]

Any window that is opened by Envelop is a custom dialog. This covers plan importing, plan editing, the wizard, the materialisation palette and the area output. All of these dialogs are implemented using the patterns described in the following, using a few functions and external resource files, as pictured in figure n. Custom dialogs heavily rely on both ‘DialogUtils’ and ‘WindowUtils’, their functionalities are described in more detail in the chapters nnn and nnn. The chapter about ‘DialogUtils’ also covers ‘DIALOG_OPTIONS’.

Custom dialogs generally have one public function to open the dialog, usually named ‘open_dialog’. This function takes the private hash ‘DIALOG_OPTIONS’ and the function ‘attach_callbacks’ and sends both of these to the ‘DialogUtils’ in order to instantiate the dialog and attach any needed callbacks using ‘add_action_callback’ provided by Sketchup. In the above module these callbacks are hinted at with ‘callback n’. One of these callbacks will send any required data to the HTMLDialog, in the above example its ‘call_load_data’. After the Javascript component of the custom dialog has finished loading it can call this function to start the data transfer. By loading the data delayed like this, it can be ensured that any receiving functions have been fully initialized. For further details how functions are called and data is shared between Javascript and Ruby, please see chapter 3.2.6.

Writing custom dialogs like this allows each dialog to focus on its own settings and functionality, while all logic regarding instantiation and opening is dealt with in ‘DialogUtils’. This improves the readability and reliability of custom dialogs.

6.1.9 Utilities

Besides the core modules, custom tools and custom dialogs, utility modules are the fourth central architectural pattern used in Envelop. However, each utility is quite different, as discussed in the chapters following 6.4.

The utilities are especially useful to encapsulate interactions with the Sketchup API, as it requires quite a bit of boilerplate code, see chapter nnn GeometryUtils, and can be difficult to navigate correctly, for example in chapter nnn DialogUtils.

Envelop also uses a number of external libraries, in Ruby and Javascript as well as CSS. Unlike the Envelop own utilities, these libraries do not interact with Sketchup, instead they are a mixture of Frameworks such as JQuerry, utilities such as ‘color_math.rb’ or ‘image_size.rb’ and components such as ‘pdf.js’ or ‘cropper.js’. The usage of external libraries is not without complications, as mentioned in chapter 3.2.2. Still, they allow Envelop code to focus on the core issues at hand, and as a result be more readable and reliable.

6.2 Tools

In this chapter, all custom tools as described in chapter 6.1.7 will be discussed. The focus will lie on technical details of these tools and their unique inner workings. The patterns covered in chapter 6.1.7 will not be reiterated in order to favour brevity.

6.2.1 PenTool

PenTool
+ activate_pen_tool:

Figure n: The ‘PenTool’ with its activation function. [M]

The pen tool in figure n has three phases, namely ‘INITIAL’, ‘FIRST_POINT’ and ‘MULTIPLE_POINTS’. In the initial phase, no previews are being drawn. As mouse clicks are registered, the phases are advanced, first to ‘FIRST_POINT’ and then to

‘MULTIPLE_POINTS’ where the tool remains until it is reset by the user or by completing the polygon that is currently being drawn.

The high level routing, as to which internal functions should be called, happen within the ‘onLButtonDown’ and ‘onReturn’ functions. These functions decide according to the current phase and the set of previous positions, ‘@previous_points’, which action to take. The normals of surfaces on which points are placed are also tracked, in order to determine the normal of the resulting polygon. This tracking is done with sets, in order to ensure each entry is unique.

Another important decision happens within the ‘finish_with_points’ function. There the resulting polygon is added to the house, in order to test if the resulting shape is still manifold. If not, the changes are undone and the polygon is created without being added to the house. This decision happens using ‘OperationUtils’ which makes undoing operations very easy.

In this tool, previews don’t change quickly and thus internal references can be used to track the preview elements. In ‘add_construction_geometry’ the preview elements are created and added to ‘@construction_entities’. The function ‘erase_construction_geometry’ on the other hand is responsible to clear the preview elements again. Additionally, each draw call draws the final preview line from the last point to the cursor.

6.2.2 PushPullTool

PushPullTool
+ activate_pushpull_tool:

Figure n: The ‘PushPullTool’ with its activation function. [M]

The push and pull tool pictured in figure n has only two phases, ‘INITIAL’ and ‘FACE_SELECTED’. If the tool can find a face using ‘GeometryUtils.pick_best_face’ under a click or drag start, it remembers the face, its center and normal. This data is used to calculate the distance and direction the face should be moved during preview drawing and when the user finished the operation. A double click to create a dormer is handled separately in ‘onLButtonDoubleClick’.

To decide if the operation should add or remove by default, the volume to be added or removed is intersected with the current house. If the resulting intersection has the same volume as it had before the operation, then all of the volume is inside the house. In this case the tool defaults to removing volume, otherwise the default is adding.

Otherwise this tool is fairly standard and uses many of the patterns described in chapter 6.1.7. As such it is a good example to familiarize oneself with the Envelop codebase. In terms of usability this is one of the most important tools offered by Envelop, as such operations have proven to be highly intuitive.

6.2.3 FloorMakerTool

FloorMakerTool
+ activate_floor_maker_tool:

Figure n: The ‘FloorMakerTool’ with its activation function. [m]

The ‘FloorMakerTool’ shown in figure n is an example of a very simple tool, that still benefits from inheriting from ‘AbstractTool’. However, due to its simplicity it does not need any phases. Instead it consists mostly of its preview drawing and floor-making if the user clicks.

The separation of the house is done by generating a huge new surface at the required elevation and then intersecting that surface with the existing house. The resulting edges are then either handed over to the ‘HouseKeeper’ to finish making a floor or they are colored and marked as preview elements using the techniques described in chapter 6.1.7.

6.2.4 ScaleTool

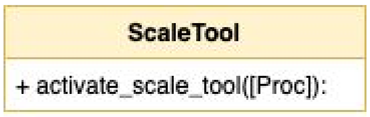


Figure n: The ‘ScaleTool’ with its activation function. [m]

The ‘ScaleTool’ that is shown in figure n has three phases, that is ‘BEFORE_FIRST_POINT’, ’BEFORE_SECOND_POINT’ und ‘BEFORE_SCALE’. This follows naturally from it’s usage in three steps: selecting two points and entering the real world distance into a dialog field. Again, most of the decision making is contained within the ‘onLButtonDown’ method.

In order to scale the whole model, all elements are put into a single temporary group. This is much easier than enumerating all elements of the model and scaling them individually. Because scaling is mandatory before the surface area totals can be calculated, Envelop must remember that the user did so. This would be impractical with Ruby state, thus another attribute dictionary is used to save this information into the model and ensure it is saved and loaded from files.

6.2.5 OrientationTool

OrientationTool
+ activate_orientation_tool:

Figure n: the ‘OriantationTool’ with its activation function [m].

The ‘OrientationTool’ in figure n has logically two phases, one before the compass origin point has been set and one before the direction has been set. In a way it is structured very similarly to the ‘ScaleTool’ - it too saves it’s orientation into the model using attributes.

However, its preview process is more involved. The compass is drawn using 100 subdivisions of a circle, 16 snap lines, four lines for the cardinal directions and letters with ‘draw_text’ provided by Sketchup. As none of the elements are Sketchup model elements and instead just temporary lines and texts, nothing needs to be removed when the tool is aborted or that changes are finalised.

6.2.6 MaterialisationTool

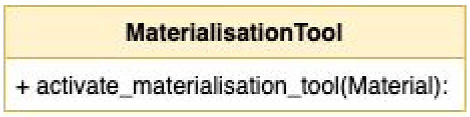


Figure n: The ‘MaterialisationTool’ and its activation function. [m]

The ‘MaterialisationTool’ in figure n differs from other tools in that its activation function takes a parameter. This way it can be activated for a specific material, which will be applied to surfaces that are clicked by the user. To detect these surfaces it uses ‘GeometryUtils’. The ‘MaterialisationTool’ therefore contains only code to actually assign the chosen material and is, as a result. One of the simplest tools.

6.2.7 PlanManagerTool

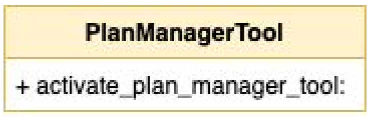


Figure n: The ‘PlanManagerTool’ and its activation function. [m]

The ‘PlanManagerTool’ in figure n has three phases, which can be better understood as modes. They are ‘NEUTRAL’, ‘DRAGGING’ and ‘MOVING’. Without a plan selected the tool is in ‘NEUTRAL’, but then switches to either ‘DRAGGING’ or ‘MOVING’ depending on if the user is moving the plan by dragging it along its axis or if he is moving it by selecting it and then clicking again to confirm it’s new position.

As long as the tool is not in ‘NEUTRAL’, the selected plan is being moved in ‘onMouseMove’. This is both the preview as well as the actual translation of the plan. Thus to confirm, the internal state of the tool simply has to be reset. On the other hand, if the movement is canceled, the already performed translation has to be undone.

To find the position to move to, the mouse cursor is mapped to the nearest point on a line extending from the center of the plan both in negative and positive direction along its normal. If the cursor however is over a point, edge or face other than the plan being moved, this point is used directly instead of searching the nearest point along the normal.

Finally, hiding a plan with a double click is handled independently from all of the above in the ‘onLButtonDoubleClick’ method.

6.2.8 PlanPositionTool

PlanPositionTool
+ activate_plan_position_tool(Image):

Figure n: The 'PlanPositionTool' with its activation function. [m]

The 'PlanPositionTool' in the above figure also takes an activation argument. In this case it's the plan that has to be positioned. Each individual step of this positioning is fairly easy to understand, however, because there are a lot of steps, this tool has a lot of phases.

At first the plan has to be positioned correctly. This takes two clicks and accordingly is mapped to two phases: 'BEFORE_FIRST_POINT' and 'BEFORE_MOVE'. Then the plan has to be scaled, which again takes two clicks and two phases: ' BEFORE_SECOND_POINT' and 'BEFORE_SCALE'. This tool has a special cleanup phase as well, called 'FINISHED'. All the decision making taking these phases into account happens in 'onLButtonDown'.

Part of resetting the tool is hiding all other plans in order to make positioning the plan easier. Therefore, when the tool is excited the plans have to be made visible again. Apart from this mechanism is this tool easy and straightforward to understand because of the phases model it uses. Other patterns used have been discussed in the preceding chapters.

6.3 Dialogs

In the following chapters the inner workings of all custom dialogs will be discussed. Again, the patterns presented in chapter 6.1.8 will not be reiterated. Instead the focus is on significant differences and key structures.

6.3.1 MaterialisationDialog

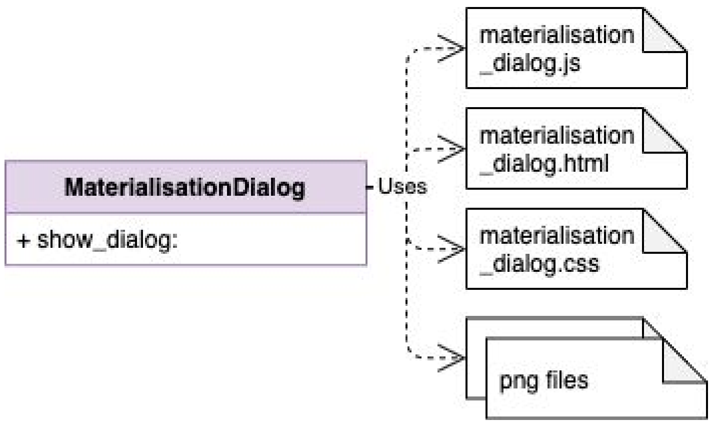

MaterialisationDialog
+ show_dialog:
Uses
materialisation _dialog.js
materialisation _dialog.html
materialisation _dialog.css
png files

Figure n: 'MaterialisationDialog' with the opening option and the required external files. [m]

The 'MaterialisationDialog' in figure n has more external files than other dialogs, because it also uses quite a few image files for its buttons.

The Ruby component of this dialog is fairly straightforward, as it features simply the dialog configuration and the required callbacks for all actions that can be performed within the dialog. More specifically, one callback to update the materials displayed in the dialog, one to start the ‘MaterialisationTool’, one to delete a material, one to create a completely new one and one to create a new material based on an existing.

The Javascript component is also easily understood, as it has a single task displaying lists of materials sent to it from Ruby. This is done with a simple enumeration over the materials sent, sorted by group and name. The HTML file has templates for groups and materials, which are copied during the enumeration. The only thing left to do is then attaching the callbacks to the newly instantiated elements. The icons are switched between black and white by assigning different CSS classes based on the l-value of the hsl encoding of each material color. Finally, the color picker is implemented using ‘color-picker.js’.

6.3.2 Wizard

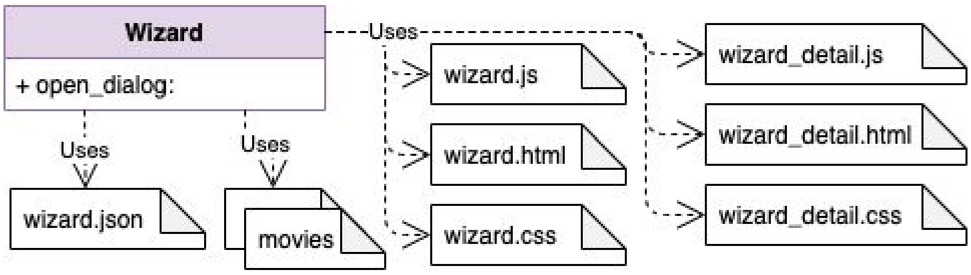


Figure n: The ‘Wizard’ dialog with all the required external files. [m]

The ‘Wizard’ in figure n is the only custom dialog that has two sets of HTML, Javascript and CSS files. Additionally it also requires a collection of movie files and a data file in the form of ‘wizard.json’. To be able to use the two different sets of files, the Ruby component also has two different dialog configurations. One of the dialogs displays the overview of the workflow and the second can display the detail view of a single step. To be able to open the second with the first, there is a second callback besides the data loading one.

The content of both of these dialogs is loaded dynamically from ‘wizard.json’, so that it’s easier to edit and update, independent of any changes to the codebase. This file contains the workflow steps and substeps, as well as the detail steps and the links to the movie files.

The Javascript components function much like the ones for the ‘MaterialisationDialog’, taking on content and then displaying it by enumerating and copying HTML template elements.

6.3.3 PlanImport

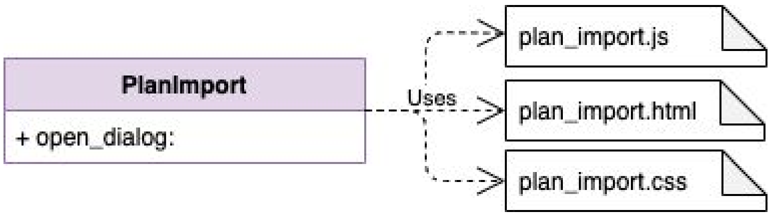


Figure n: The ‘PlanImport’ dialog with its external files. [M]

The ‘PlanImport’ dialog in figure n uses mostly the patterns described in previous chapters. However it has two important distinctions. The first regards its content origins. This is the only dialog that can add content on its own, with just the Javascript component. To do so, it uses the template instantiation technique, combined with HTML5 canvases. The file contents are loaded using ‘readAsDataURL’. If it's an image, it can be loaded directly in the canvases, otherwise ‘pdf.js’ is used to display the PDF data in a canvas.

The second important difference to other dialogs is that this dialog converts all its important plans to a base 64 encoding and then saves them in this form into the model. Of course, if a new model is opened any previously important plans are decoded and displayed in the ‘PlanImport’ dialog again.

6.3.4 PlanEdit

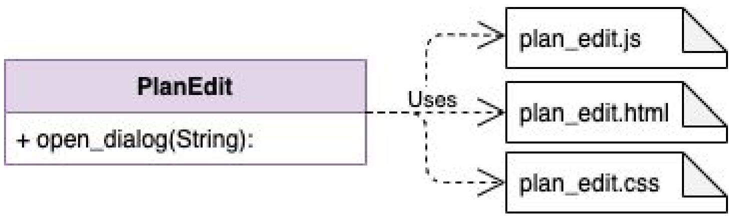


Figure n: The ‘PlanEdit’ dialog with its external files. [M]

The ‘PlanEdit’ dialog in the above figure uses two data loading callbacks. One for the plan that has to be edited, which is sent to ‘PlanEdit’ in the form of a base 64 encoded string. This string is the forwarded to the dialog with the first data loading callback. The second data loading callback tells the Javascript component which direction the user has configured to be north. This allows Envelop to name the insertion buttons accordingly. The cropping element in the dialog is provided by ‘cropper.js’, resulting in a clean and easy to read Javascript component for ‘PlanEdit’.

6.3.5 AreaOutput

Please refer to chapter 6.1.6 to learn more about the ‘AreaOutput’ dialog.

6.4 Utilities

The various Envelop utilities that have been developed and used will be listed and explained in the following chapters. The focus will lie on the technical aspects that are important for the continued development of Envelop. See chapter 6.1.9 for a more general explanation of the utilities pattern used.

6.4.1 OperationUtils

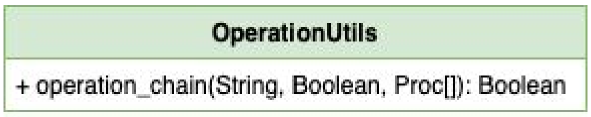


Figure n: ‘OperationUtils’ with its single function. [m]

‘OperationUtils’ in figure n have one single public function. Operations are Sketchups way to allow extensions to modify and manage the Undo/Redo stack. Each action that the user performs with Envelop may consist of several actions behind the scenes. ‘OperationUtils’ allows all of these actions to be committed with a single name to the undo stack. To do so, a code block with all of the actions to be executed is handed over to ‘operation_chain’. It can even consume and resolve an array of code blocks.

‘OperationUtils’ also facilitates undoing previous actions in a block if a later one fails - to do so, just return false from the code block. This can be very useful to make temporary changes to the model, take measurements of the change model and then return to the unmodified model again.

Finally, it is also possible to commit transparent code blocks. This means that any changes to the model are merged with the previous action in the Undo/Redo stack. Unfortunately, this stack is limited to 100 entries, thus committing a lot of transparent actions is not recommended.

6.4.2 CameraUtils

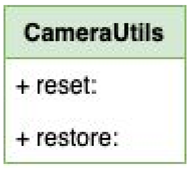


Figure n: ‘CameraUtils’ with its two functions. [m]

The ‘CameraUtils’ in figure n are only used to import plans into the model. This utility is needed because the ‘add_image’ method provided by Sketchup does so in relation to the current camera angle. To ensure that angle is consistent, before inserting ‘reset’ is called. This method saved the current camera position and angle and resets it to a known standard value. After the plan is inserted, the previous camera angle and position can be restored using ‘restore’. Because the view is not refreshed in between these steps there is no flickering and the operations finish very quickly.

6.4.3 ToolUtils

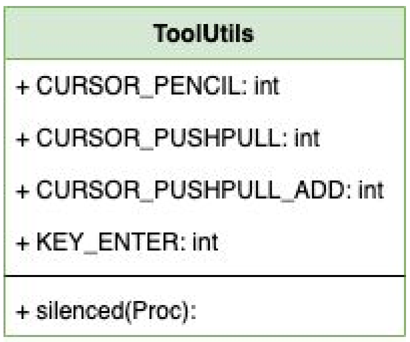


Figure n: ‘ToolUtils’ with a number of constants. [m]

The ‘ToolUtils’ in figure n are mostly a container for the ‘AbstractTool’ class described in chapter 6.1.7. This class will not be covered here again.

The first four entries in the above figure are constants. The first three allow custom tools to select a cursor to use from this predefined list, instead of having to use a constant value. The fourth is a constant of the key value assigned to the enter key. This makes detecting key presses in custom tools easier. Finally, the ‘silenced’ function takes a block of code and executes it, without printing anything to the standard output. This is helpful if actions are only performed in order to do some kind of calculations but will be undone quickly again - such actions do not need to be logged.

6.4.4 DialogUtils

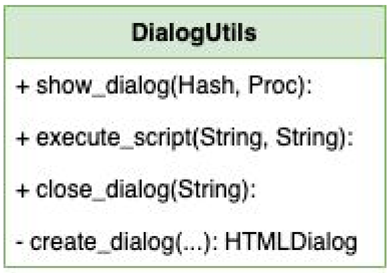

DialogUtils
+ show_dialog(Hash, Proc):
+ execute_script(String, String):
+ close_dialog(String):
- create_dialog(...): HTMLDialog

Figure n: ‘DialogUtils’ with its most important functions. [m]

The most important function of the ‘DialogUtils’ in the above figure is ‘show_dialog’ which, if necessary, creates a new dialog and shows it. Internally, the created dialogs are tracked using a hash object, where the key is a string id per dialog. This id, just like all of the other settings required, is defined by the custom dialog using the ‘DialogUtils’ as part of the ‘DIALOG_OPTIONS’ hash that is the first parameter to the function. The second parameter is a function which in turn takes the created HTMLDialog as a parameter. Custom dialogs can use this parameter to assign callbacks to the dialog and with this establish communication with the dialogs Javascript components.

The string id of each dialog mentioned above is also required for the ‘execute_script’ function, to identify the dialog instance to which to send the command. It is also needed in the ‘close_dialog’ function.

The internal function ‘create_dialog’ is needed if ‘show_dialog’ is called with a dialog id for which no dialog has been created yet. Its parameters are populated directly from the hash parameter in ‘show_dialog’. This hash must adhere to the following structure in table n.

Parameter Name	Type	Default	Description
path_to_html	String	Required	Absolute path to the HTML file for the dialog.
title	String	Required	Title of the window for the dialog.
id	String	Required	Unique string identifying the dialog for future references.
height	int	Required	Height of the window in pixels. Will be ignored if Sketchup has already stored a user preference.
width	int	Required	Width of the window in pixels. Will be ignored if

			Sketchup has already stored a user preference.
pos_x	int	Required	Horizontal position of the window for the dialog. Will be ignored if Sketchup has already stored a user preference.
pos_y	int	Required	Vertical position of the window for the dialog. Will be ignored if Sketchup has already stored a user preference.
center	Boolean	false	If set to true, the dialog will be created in the center of the screen, overriding any other parameters affecting the position.
can_close	Boolean	false	If set to true, the dialog cannot be closed by the user.
resizeable_height	Boolean	false	If set to true, the dialog can be vertically resized by the user.
resizeable_width	Boolean	false	If set to true, the dialog can be horizontally resized by the user.
min_height	int	0	If the window is resizable, then this value sets a lower limit for the window height.
min_width	int	0	If the window is resizable, then this value sets a lower limit for the window width.
dont_save_prefs	Boolean	false	If this parameter is set to true, the dialog will not remember any user preferences regarding size and position.

Table n: Elements of the hash parameter to the ‘show_dialog’ function. [m]

Sketchup saves the positions and size of dialogs but offers neither a mechanism to detect if these settings exist nor a way to reliably set custom values if there are stored values. ‘DialogUtils’ therefore creates its own external files to track which dialogs have been created previously and which have not in order to decide if the position and size must be overridden. This mechanism too uses the dialog id.

6.4.5 WindowUtils

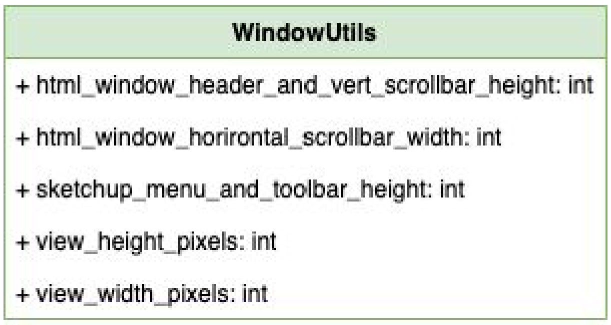
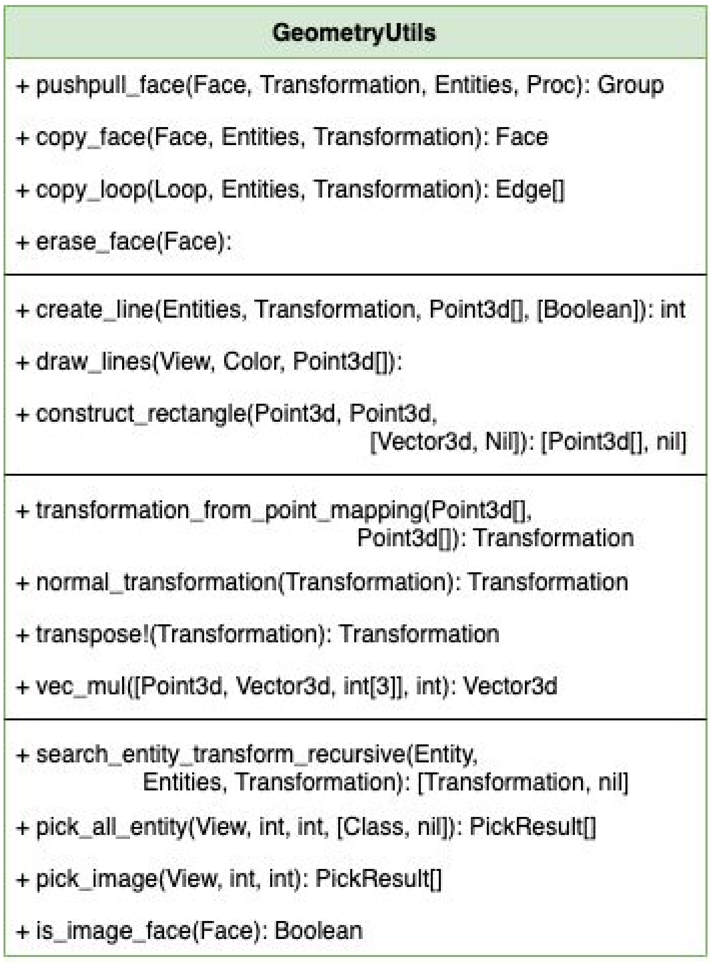

WindowUtils
+ html_window_header_and_vert_scrollbar_height: int
+ html_window_horirontal_scrollbar_width: int
+ sketchup_menu_and_toolbar_height: int
+ view_height_pixels: int
+ view_width_pixels: int

Figure n: ‘WindowUtils’ with its public functions. [m]

The ‘WindowUtils’ in figure n helps Envelop open its dialogs in appropriate sizes and positions across different operating systems. For example on Mac OSX and Windows, the scrollbars take up different amounts of space within a dialog. All of the above functions enable custom dialogs to be configured without taking the operating system into account, instead this logic is encapsulated here. The values have been determined using trial and error, thus there is a significant risk that they do not hold in all situations. However, the dialogs can be resized and positioned by the user and those settings will be remembered the next time the same dialog is opened. Therefore, if mistakes are made, the user can fix them himself and has to do so only once.

The operating system is detected using ‘os.rb’ a third party library.

6.4.6 Geometry Utils



```
GeometryUtils

+ pushpull_face(Face, Transformation, Entities, Proc): Group
+ copy_face(Face, Entities, Transformation): Face
+ copy_loop(Loop, Entities, Transformation): Edge[]
+ erase_face(Face):

+ create_line(Entities, Transformation, Point3d[], [Boolean]): int
+ draw_lines(View, Color, Point3d[]):
+ construct_rectangle(Point3d, Point3d,
                                  [Vector3d, Nil]): [Point3d[], nil]

+ transformation_from_point_mapping(Point3d[],
                                  Point3d[]): Transformation
+ normal_transformation(Transformation): Transformation
+ transpose!(Transformation): Transformation
+ vec_mul([Point3d, Vector3d, int[3]], int): Vector3d

+ search_entity_transform_recursive(Entity,
            Entities, Transformation): [Transformation, nil]
+ pick_all_entity(View, int, int, [Class, nil]): PickResult[]
+ pick_image(View, int, int): PickResult[]
+ is_image_face(Face): Boolean
```

Figure n: ‘GeometryUtils’ with most of its functions, grouped by areas of concern. [m]

The‘GeometryUtils’ in figure n is the utility that simplifies interacting with the Sketchup 3d model and other performing related 3d calculations. The size and complexity of this utility have been broken down into the groupings in the above figure. In the following paragraphs these groupings will be discussed in order. Due to the generally high complexity of these methods, they have been extensively documented using comments within the sourcefile. It is recommended therefore to also take a look at the source file in appendix 8.6.

The first grouping contains methods to directly modify elements of the model hierarchy. Just like many native Sketchup methods, these often contain an ‘Entities’ parameter into which the resulting elements will be added and an Transformation parameter, which is the transformation that is applied to the source elements in the hierarchy. Transformations are not consumed upon being applied and instead become part of the element.
More explicitly, the first grouping contains methods to move a face with ‘puspull_face’, to copy a Sketchup Face or Loop and to delete a face. The first method is implemented using the copy methods, which also support a transformation argument or block.

The second grouping contains methods to create new elements. While ‘create_line’ and ‘draw_lines’ directly insert elements into the model, ‘construct_rectangle’ just creates the Points as Ruby instances and returns them.

The third grouping contains methods to perform mathematical vector and matrix operations. Since ‘Transformation’ is Sketchups way to represent a matrix, these methods mostly work with that argument type. The first method finds the transformation required to translate a set of points to a second set of points. The other functions perform simple calculations: adapting a transformation matrix for normal vectors, transposing a transformation matrix and scalar multiplication of vectors.

The last grouping contains functions relating to navigating and searching the model hierarchy. For example the ‘search_entity_transfrom_recursively’ method, which sums up all the transformations while navigating through the hierarchy to a specific entity that is part of the model. ‘pick_all_entity’ on the other hand finds the entities of a given type under the current cursor position. The Sketchup API is powerful but relatively complicated. Using this guide, this function returns the results conveniently with their transformations, parents and without any image components. Because image components, in Envelops case plans, are filtered out, there is also the ‘pick_image’ method, which works similarly, but instead only looks for images. The final method checks if a given ‘Face’ is part of an image, because such faces should not be treated like regular surfaces in most cases.

6.4.7 ObserverUtils

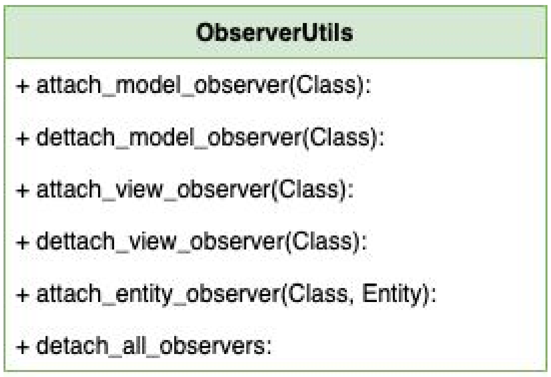

ObserverUtils
+ attach_model_observer(Class):
+ dettach_model_observer(Class):
+ attach_view_observer(Class):
+ dettach_view_observer(Class):
+ attach_entity_observer(Class, Entity):
+ detach_all_observers:

Figure n: ‘ObserverUtils’ with its public functions. [m]

The ‘ObserverUtils’ in the above figure manges Sketchup observers, such as the ‘EntityObserver’ provided by Sketchup. ‘ObserverUtils’ makes sure each observer class is only instantiated once per target, by keeping track of instances and targets in internal hash objects. These hash objects are also used to find the instances to detach again. For example, if the extension is reloaded, all observers must be detached. In this clean state, all modules can reattach their observers.

‘ObsercerUtils’ supports attaching and detaching three different kinds of observer targets: Views, Models and Entities. Other observers must be managed by the module requiring it.

7. Conclusion

This chapter will review the work that has been done as part of the “IIT22, Vereinfachte Modellierung von Gebäudehüllen”.

7.1 Accomplishments

As part of this project the following was accomplished:

- Analysis of the customer requirements in terms of accuracy, input data, materialisation and expected output format. This also required the analysis of the existing workflow in Plancal Nova in order to determine how the usability and speed of the process can be increased.
- Analysis and evaluation of different approaches to modeling data models as well as already available software solutions in relation to the previously determined requirements. This includes a more thorough evaluation of Sketchup and Blender involving initial experiments with Ruby and Python.
- Iterative development of the Envelop extension. This process includes demos and user testing in order to ensure the final product meets the customer demand.
- Thorough documentation of Envelop, both from a user and from a technical perspective, in the form of this report and the manual in appendix 8.2.
- Release of the complete source code and documentation to the open source community through the repository in appendix 8.5.

7.2 Conclusion

Powerful generic software can solve many problems, but as this project demonstrates such software can be overly complicated and inefficient for specific purposes. In this case, generic modeling software such as Blender or Sketchup and even architectural modeling software Plancal Nova take a lot more time to calculate building envelopes than what would be expected considering the task complexity and required precision. Specific solutions, such as specialised plugins, can be much faster. In this case, Envelop is four to five times faster, as demonstrated in chapter 5.3, while still fulfilling the precision requirements for this task. Moreover, the Sketchup Ruby API is powerful enough to support the development of extensions that aim to specialize and optimize Sketchup for specific tasks. Finally, visual interactive tools, such as pens to draw shapes, pulling and pushing surfaces or tracing images are very efficient for taks where a few percentage points of inaccuracy are permissible.

7.3 Outlook

How to continue the development of Envelop from a technical perspective is covered in the manual reference in appendix 8.2. In terms of analysis, there are three lenses under which continued development of Envelop can be understood:

7.3.1 Maintenance

As with all software, the Envelop extension might contain bugs that are only discovered during more intensive regular usage. Further issues might be introduced or discovered as Sketchup releases new versions as well. Fixing such bugs can be understood as software maintenance. If any additional resources can be spent on the continued development of Envelop, this would be the most urgent and important kind of development.

7.3.2 Refinement

During the development of Envelop, there have been features and feature refinements that have been deemed as out of scope for the initial version. These ideas can be found as issues in the repository in appendix 8.5, as described in the chapter about the development environment in the manual in appendix 8.2. Minore code style refinements have also been documented using todo comments, directly in the source. Resolving these issues refines the existing Envelop extension. As this development builds on the existing codebase, the effort is usually less than implementing new features as in 7.3.3.

7.3.3 Improvement

It is to be expected that during regular and intensive use, users will discover additional features they would like in order to further increase the usability of Envelop and with that, the speed of calculating building envelope surface area totas. Implementing new features like this represents the most effort while they are likely the least urgent.

8. Appendix

8.1 Quick Guide

The Envelop extension contains a ‘Quick guide’ in the form of a wizard that can be viewed inside Sketchup. To view it, install Envelop as described in the manual in appendix 8.2 and open the wizard which too is described in the manual.

8.2 Manual

There exists a detailed user manual, covering the installation, usage as well as continued development of the Envelop extension. It can be found in the repository mentioned in appendix 8.5 or with this direct link: [url]

8.3 Architecture Overview

The full scale version of the architecture overview diagram can be found in the repository mentioned in appendix 8.5 or directly with this link:
https://github.com/FSiffer/IP6_ModellingBuildingEnvelopes/blob/master/docs/6.1.1%20Architecture%20Overview.png

8.4 Envelop Extension

The Envelop Sketchup extension can be found in the repository mentioned in appendix 8.5 or directly with this link: [url]

8.5 Source Code

Any and all source code can be found on the following Github page: https://github.com/FSiffer/IP6_ModellingBuildingEnvelopes. The repository also contains the final version of this report, the results of the workflow testing mentioned in chapter 5. Evaluation of New Workflow" and executables of the final version of [TDB].

8.6 Testing Artefacts

The testing artifacts, that is the models, their respective results, plans and measured metrics as well as any calculations provided by the customer can be found in the repository referenced in chapter 8.5 or directly with this link: https://github.com/FSiffer/IP6_ModellingBuildingEnvelopes/tree/master/Planbeispiele

8.7 References

8.8 Declaration of Honesty

We hereby confirm that the present Report is solely our own work and that if any text passages or diagrams from books, papers, the Web or other sources have been copied or in any other way used, all references – including those found in electronic media – have been acknowledged and fully cited.

Signature: ______________________________ Date & Place: ______________________________

Signature: ______________________________ Date & Place: ______________________________