Article

An Automated Prefabricated Facade Layout Definition for Residential Building Renovation

Kepa Iturralde [1,2,*], Samanti Das [1,2], Aravind Srinivasaragavan [1,2], Thomas Bock [1] and Christoph Holst [2]

[1] Chair of Building Realization and Robotics, School of Engineering and Design, Technical University of Munich, 80333 Munich, Germany; ge75guy@mytum.de (S.D.); bockrobotics@web.de (T.B.)
[2] Chair of Engineering Geodesy, School of Engineering and Design, Technical University of Munich, 80333 Munich, Germany; christoph.holst@tum.de
* Correspondence: kepa.iturralde@tum.de

Abstract: The rising global energy demand has made it essential to reduce energy consumption in the residential building stock. Adding a new insulating envelope with Renewable Energy Sources (RESs) onto the existing buildings' facade is one way to achieve zero energy consumption in residential buildings. The ENSNARE project aims to semi-automate this process by using prefabricated facade modules and developing new building data acquisition techniques. Prior to this research project, an analysis was carried out and several research gaps were identified. One of the obstacles to using prefabricated modules with RES is that the layout needs to be drawn and adjusted during different phases of the project. That is time-consuming. For this reason, this article describes two new solutions: (1) automated drafting of the optimized layout of prefabricated modules of the facade and the number of solar panels based on the existing residential building model, and (2) automated adjustment of the layout depending on the phase of the renovation project and the accuracy of the measurements in each step. The proposed semi-automated approach has the potential to significantly reduce the time used in drafting the layout of the prefabricated modules, which benefits the whole renovation process, contributing to a more sustainable future for the residential building stock.

Keywords: online; building model; prefabrication; renovation

check for updates

Citation: Iturralde, K.; Das, S.; Srinivasaragavan, A.; Bock, T.; Holst, C. An Automated Prefabricated Facade Layout Definition for Residential Building Renovation. *Buildings* **2023**, *13*, 2981. https://doi.org/10.3390/buildings13122981

Academic Editor: Asimina Athanatopoulou-Kyriakou

Received: 30 September 2023
Revised: 15 November 2023
Accepted: 20 November 2023
Published: 29 November 2023

1. Introduction

One of the key strategies for addressing the challenge of climate change is to reduce global energy consumption [1]. In recent years, efforts to achieve zero-energy consumption of residential buildings have focused on insulating existing structures and installing renewable energy sources (RES) on their rooftops [2–4]. Other approaches include optimizing building envelopes to facilitate better harvesting of solar energy [5–7]. However, manually implementing these measures carries the potential risks of user intrusion, disruptions, and hazardous activities carried out at elevated heights. To circumvent these issues, prefabricated modules are now being manufactured off-site, incorporating insulation, RES, windows, and waterproofing elements [8]. Previous studies have explored the use of automated, robotic facade renovation with prefabricated modules, which can be categorized into three sub-categories [9–11]: information or data flow, off-site manufacturing of the modules, and on-site installation of the modules (see Figure 1).

The adoption of prefabricated elements in residential building renovation has not yet gained widespread market acceptance due to its lack of competitiveness compared to traditional manual methods. The ENSNARE research project [12] aims to reduce data acquisition and processing time by 90%, while also reducing the duration of the manufacturing and installation processes. It is crucial to ensure that any reduction in working time does not result in a loss of quality and adheres to regulations and standards [13]. Errors in the data flow can lead to deviations in the manufactured modules, resulting in water and heat leaks, collisions, or impeded installation processes.

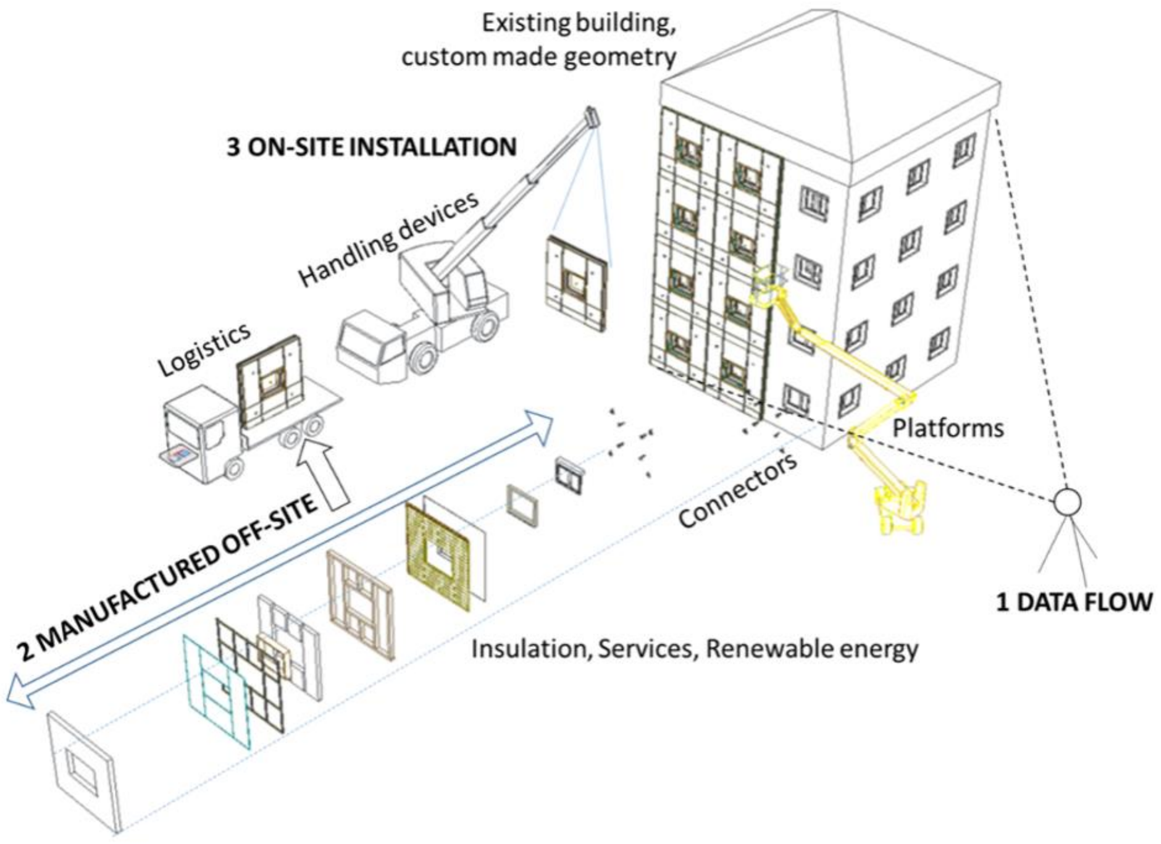

Figure 1. Building renovation with prefabricated modules.

The successful implementation of robotics and automated solutions in residential building renovation largely depends on their economic feasibility [14–17]. Furthermore, it is essential to address the issue of managing and preparing such technologies for the market [18–20]. Previous research initiatives, including BERTIM [21] and HEPHAESTUS [22], have examined this matter. Earlier studies [23] have identified up to 15 research gaps (RG) in this field.

In the field of residential building renovation, the utilization of prefabricated modules for energy-saving and generation purposes often presents less competitive alternatives compared to manual procedures due to the requirement for more comprehensive and detailed planning. When it comes to prefabricated modules for building renovation, the involved parties, including building owners, promoters, or engineers, must possess a comprehensive understanding of the building's solar energy generation capabilities, investment costs, and the necessity for insulation in the preliminary stages of the project. To achieve this, it is crucial to have a geo-located three-dimensional (3D) model of the building that can represent the building's shape, structure, and capability to accommodate prefabricated modules and solar panels. In this context, the layout of prefabricated walls and solar panels assumes a significant role, providing a clear depiction of the number of solar panels that can fit onto the building envelope, the requisite amount of insulation, and the corresponding investment costs. Concerning data acquisition and flow, and to explain the context of this article, it has been determined that the following measures are necessary:

- **RG1.1:** In the initial phases, it is necessary to generate a building model online, meaning without visiting the site. In previous stages of the research, a method to generate building models online was achieved based on facade images and OpenStreetMaps data.
- **RG1.2:** Typically, in building renovation, the initial definition of the prefabricated module layout is a manual process and can take up to 5–20 h for a low-rise building.

The aim should be to automatize defining the layout of prefabricated modules with the information of the building modeled online in RG1.1.

- **RG1.3:** On the next stages of the project, once the "client" or building owner approves the budget, the engineering team can afford to visit the building and measure accurately. Using Total Station or even a 3D Laser Scanner can be a time-consuming activity. A faster method was developed based on OpenCV Apriltags [24].
- **RG1.4:** The measurement accuracy is much higher in RG1.3 than with the technique described in RG1.1. This means that the layout needs to be re-adjusted based on more accurate measurements. Adjusting the layout by manual means is time-consuming. Therefore, we developed a technique to adjust the layout automatically.

This article focuses on describing solutions in order to mitigate research gaps **RG1.2** and **RG1.4** in the following sections. As said before, RG1.1 and RG1.3 were developed in a previous phase of the research.

2. Addressing RG1.2 by Defining the Layout of the Prefabricated Modules

In this section, we explain RG1.2, which is based on the ***ENSNARE_MODULE_PLACEMENT***, which is a FreeCAD [25] plugin developed to generate the layout of prefabricated facade modules with solar panels for residential building upgrading. The tool represents a concerted effort to improve the cost-effectiveness and efficiency of residential building renovation within the ENSNARE project. The flow chart in Figure 2 explains the process of the solution.

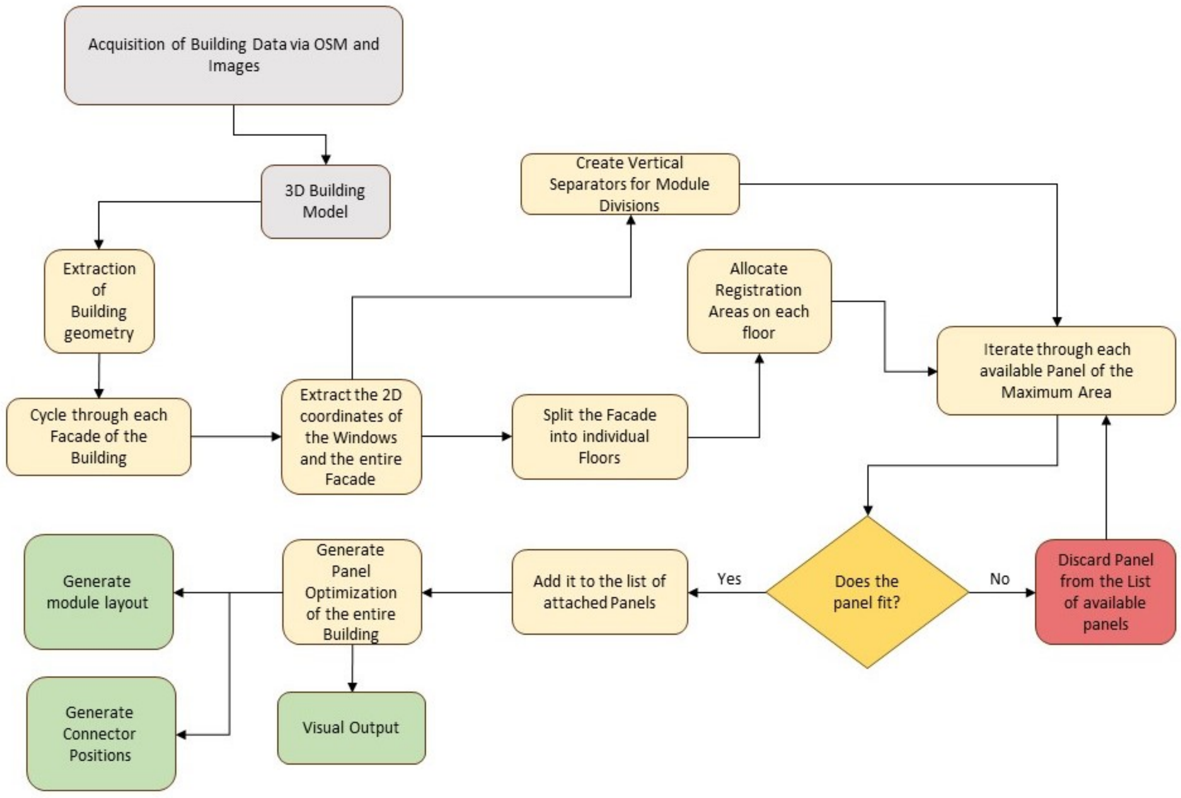
Acquisition of Building Data via OSM and Images
3D Building Model
Extraction of Building geometry
Cycle through each Facade of the Building
Extract the 2D coordinates of the Windows and the entire Facade
Create Vertical Separators for Module Divisions
Split the Facade into individual Floors
Allocate Registration Areas on each floor
Iterate through each available Panel of the Maximum Area
Does the panel fit?
Yes
No
Discard Panel from the List of available panels
Add it to the list of attached Panels
Generate Panel Optimization of the entire Building
Generate module layout
Generate Connector Positions
Visual Output

Figure 2. Flow-chart of the solution RG1.2. The inputs are shown in gray, the outputs in green, and the steps required in yellow.

Before applying RG1.2, the ENSNARE project has defined a technique to draft the building model online. In Figure 3, one of the four pictures of a demo building is shown. This demo-building is part of the ENSNARE research project and it is located in Milan.

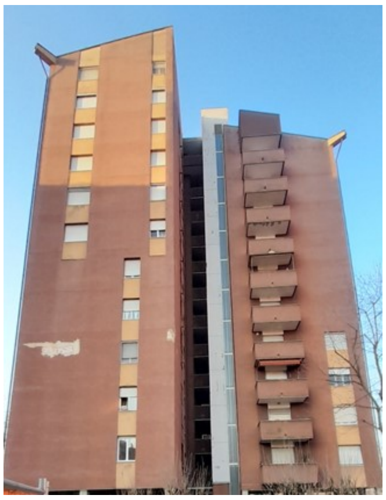

Figure 3. Actual photo of a demo-building in Milan. With images like this and the OSM floor-plan, the building model can be generated as shown in Figure 4.

The building model is generated with the images of the building and the OSM information. The building model type that is achieved is shown in Figure 4.

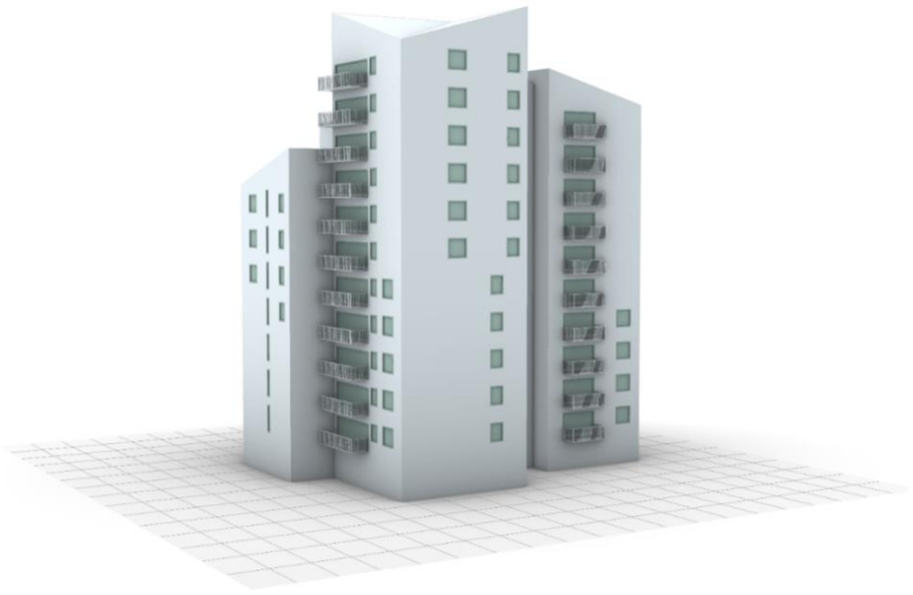

Figure 4. Building model generated with building images and OSM, in this case, the demo-building in Milan.

This building model is the primary input for RG1.2. In the next subsections, each of the steps and the code of RG1.2 are explained.

2.1. Tools Associated with the Plugin

The FreeCAD plugin ENSNARE_MODULE_PLACEMENT comprises several tools that facilitate the generation of the layout of prefabricated solar panel modules with the output of the 3D building model. Each of these tools serves primarily two sets of functions. Namely, a set of tools to extract the relevant information about the 3D building geometries and another set of tools to place solar panels based on the geometric information as explained in Figure 5.

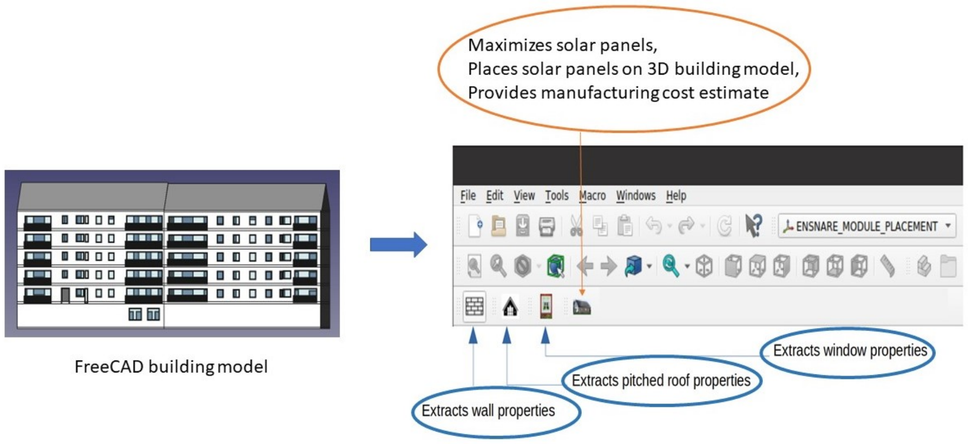
Maximizes solar panels,
Places solar panels on 3D building model,
Provides manufacturing cost estimate
FreeCAD building model
File Edit View Tools Macro Windows Help
ENSNARE_MODULE_PLACEMENT
Extracts window properties
Extracts pitched roof properties
Extracts wall properties

Figure 5. Several tools and functions of the plugin.

2.1.1. Extraction of Relevant Information

This research has devised an algorithm for maximizing the number of solar panels of diverse dimensions that can be accommodated on a two-dimensional facade. To implement this algorithm, it is necessary to extract pertinent information related to the building's facade from the 3D building model generated by the plugin team in ENSNARE.

ENSNARE_MODULE_PLACEMENT is equipped with several tools that enable the extraction of essential information from the online 3D building model, as depicted in Figure 5. The subsequent subsection provides a comprehensive description of these tools:

1. *Tool: exportWallProperties*

This tool facilitates the extraction of essential data related to a building facade, such as its height, width, and four corner points. The user can access this tool by selecting the desired facade and clicking on the corresponding icon. Upon execution, the tool generates a JSON file containing the aforementioned information. This tool is implemented in Listing 1 as shown below:

Listing 1. A Python function which generates a JSON file containing building geometries.

```

def exportWallProperties():

    clicked_face = freecad_utils.get_clicked_face()
    facade_height,
    facade_width,
    wall_vertices_list = face_dimensions(clicked_face)
    wall_dict = { "wall_height": facade_height,
                "wall_width": facade_width,
              "vertexes_anticlockwise":wall_vertices_list}

    with open(f"{constants.path}/wall_properties.json","w")
             as write_file:
        json.dump(wall_dict, write_file, indent=4)

    FreeCADGui.Selection.clearSelection()
```

This function uses another function internally: *face_dimensions*, which shows how exactly the corner vertices and the height and width are determined from the FreeCAD data. This function returns those values to the user. This procedure is described in Listing 2:

Listing 2. A Python function integral to the previous one.

```python

def face_dimensions(face: 'Part.Face') ->
                    Tuple[float, float]:
    vertexes = [v.Point for v in face.Vertexes]
    corner_points_list_bottom = []
    corner_points_list_top = []
    max_z = 0
    min_z = 0
    for point in vertexes:

        if point.z == min_z:
            corner_points_list_bottom.append(point)
        if point.z > max_z:
            max_z = point.z

    for point in vertexes:

        if point.z == max_z:
            corner_points_list_top.append(point)

    if corner_points_list_bottom[0].x >
            corner_points_list_bottom[1].x:
        right_bottom_vertex = corner_points_list_bottom[0]
        left_bottom_vertex = corner_points_list_bottom[1]
    else:
        right_bottom_vertex = corner_points_list_bottom[1]
        left_bottom_vertex = corner_points_list_bottom[0]

    if corner_points_list_top[0].x >
            corner_points_list_top[1].x:
        right_top_vertex = corner_points_list_top[0]
        left_top_vertex = corner_points_list_top[1]
    else:
        right_top_vertex = corner_points_list_top[1]
        left_top_vertex = corner_points_list_top[0]

    wall_vertices = [left_bottom_vertex,
    right_bottom_vertex, right_top_vertex, left_top_vertex]

    ref_point = wall_vertices[0]
    for point in wall_vertices[1:]:
        if point.z == ref_point.z:
            fac_width = abs(point.sub(ref_point).Length)
        else:
            fac_height = abs(FreeCAD.Vector(ref_point.x,
            ref_point.y, point.z).sub(ref_point).Length)

    assert fac_height and fac_width

    wall_index_list = []
    for index, point in enumerate(vertexes):
        if point == wall_vertices[0]
        or point == wall_vertices[1]
        or point == wall_vertices[2]
        or point == wall_vertices[3]:
            wall_index_list.append(index)

    wall_vertices_list = []
    for item in wall_vertices:
        item_list = math_utils_.vector2list(
                            item,scale=0.001)
        wall_vertices_list.append(item_list)

    return fac_height, fac_width, wall_vertices_list
```

2. *Tool: exportTriangularRoofProperties*

This tool serves the purpose of extracting information related to pitched roofs. In some cases, buildings consist of facades with pitched roofs on top. To obtain relevant information such as the height and width of the facade, the topmost point of the roof, and the vertices of the roof, the user must select the appropriate roof and click on the corresponding icon.

The code for generating the required information from a pitched roof is presented in Listing 3:

Listing 3. A Python tool which extracts relevant geometric information associated with pitched roofs.

```python

def face_dimensions_roof_triangle(face: 'Part.Face') ->
                                Tuple[float, float]:
    vertexes = [v.Point for v in face.Vertexes]
    corner_points_list_bottom = []
    corner_points_list_top = []

    z_list_vertexes = []

    for point in  vertexes:
        point.z = math_utils_.truncate(point.z, 4)
        z_list_vertexes.append(point.z)

    max_z = max(z_list_vertexes)
    min_z = min(z_list_vertexes)

    for point in vertexes:

        if point.z == min_z:
            corner_points_list_bottom.append(point)
        if point.z > max_z:
            max_z = point.z

    for point in vertexes:

        if point.z == max_z:
            corner_points_list_top.append(point)

    # roof_start
    print(corner_points_list_top)
    print(corner_points_list_bottom)
    # roof_end
    wall_vertices = [corner_points_list_bottom[0],
                     corner_points_list_bottom[1],
                     corner_points_list_top[0]]
    roof_topmost_point = corner_points_list_top[0][2]*1000
    fac_width = abs(corner_points_list_bottom[1].sub(
                corner_points_list_bottom[0]).Length)
    side1 = abs(corner_points_list_top[0].sub(
                corner_points_list_bottom[0]).Length)
    side2 = abs(corner_points_list_top[0].sub(
                corner_points_list_bottom[1]).Length)

    area = math_utils_.area_triangle(side1, side2,
                                fac_width)
    fac_height = 2 * area / fac_width

    assert fac_height and fac_width

    wall_vertices_list = []
    for item in wall_vertices:
        item_list = math_utils_.vector2list(item,
                                scale=0.001)
        wall_vertices_list.append(item_list)

    return fac_height, fac_width, wall_vertices_list,
            roof_topmost_point
```

3. Tool: exportLBCProperties

This tool is intended to determine the left bottom corner points of a building facade. The need for this tool arises due to the manner in which FreeCAD software version 0.19.3 stores information. In later stages, a 2D projection of the facade is required, and it is necessary to know which portion of the 3D facade corresponds to its left bottom corner. Since it is challenging to select and extract the left bottom corner point from FreeCAD, the sketch file of the left bottom-most window must be chosen, and the button needs to be clicked to obtain the left bottom points of the facade.

4. Tool: exportRBCProperties

This tool has a similar purpose as the previous one, with the difference being that it tracks the right bottom points of the building facade. The need for this tool arises because some building facades have no windows on their left side, only on the right. Since the previous tool uses the corner window positions to determine the corners of the facade for efficient 2D projection, it cannot be used in such cases.

5. Tool: exportWindowProperties

This tool is utilized to collect and store the position, length, and width of all the windows on a given facade. The user selects the sketch files of all the windows present on the corresponding facade and clicks on this tool. The information regarding the position, length, and width of each window is then collectively stored in a JSON file for further use.

2.1.2. Maximizing the Solar Panels

This particular section describes the workbench's functionality to maximize the total number of possible solar panel modules that can be placed on the facade. Additionally, it also provides a total manufacturing cost estimate for this entire project. This entire functionality is achieved by pressing a single button on the workbench, as described in detail below.

6. Tool: maximize_solar_panel

This tool's main function is to maximize the number of solar panels that can be fit in a selected facade and also to provide a manufacturing cost estimate. It serves a variety of functions in order to achieve that. Firstly, **it projects the entire 3D facade into 2D**. For that, it calculates the slope and reference point of the facade by taking into consideration the four corner vertices and the left bottom point of the facade (obtained from previous tools). This procedure is implemented in Listing 4 as shown below:

Listing 4. A Python function which performs a geometric projection.

```

def get_slope_reference_point():

    height_wall = wall_dict["vertexes_anticlockwise"][3][2]
    list_x = [wall_dict["vertexes_anticlockwise"][0][0],
              wall_dict["vertexes_anticlockwise"][1][0],
              wall_dict["vertexes_anticlockwise"][2][0],
              wall_dict["vertexes_anticlockwise"][3][0]]

    list_y = [wall_dict["vertexes_anticlockwise"][0][1],
              wall_dict["vertexes_anticlockwise"][1][1],
              wall_dict["vertexes_anticlockwise"][2][1],
              wall_dict["vertexes_anticlockwise"][3][1]]


    arr_x = np.asarray(list_x)
    arr_y = np.asarray(list_y)

    for value in left_bottom_dict:
        if sketch in value["label"]:
            point = value["placement"]["origin"]
            i_x = (np.abs(arr_x - point[0])).argmin()
            i_y = (np.abs(arr_y - point[1])).argmin()

    if i_x == 0:
        if list_x[0]!= list_x[3]:
            wall_dict["vertexes_anticlockwise"][3] =
                    [list_x[i_x], list_y[i_x], height_wall]
    elif i_x == 1:
        if list_x[1]!=list_x[2]:
            wall_dict["vertexes_anticlockwise"][2] =
                    [list_x[i_x], list_y[i_x], height_wall]
    elif i_x == 2:
        if list_x[2]!=list_x[1]:
            wall_dict["vertexes_anticlockwise"][1] =
                    [list_x[i_x], list_y[i_x], 0.0]
    else:
        if list_x[3]!=list_x[0]:
            wall_dict["vertexes_anticlockwise"][0] =
                        [list_x[i_x], list_y[i_x], 0.0]

    p0 = np.array(wall_dict["vertexes_anticlockwise"][0])
    p1 = np.array(wall_dict["vertexes_anticlockwise"][1])

    dist = np.linalg.norm(p0 - p1)
    fac_width = dist * 1000
    wall_vertices_list=wall_dict["vertexes_anticlockwise"]
    first_wall_vertex_xy =
          [wall_dict["vertexes_anticlockwise"][0][0],
          wall_dict["vertexes_anticlockwise"][0][1]]
    reference_point_ = [list_x[i_x], list_y[i_x], 0]
    reference_point_xy = [reference_point_[0],
                            reference_point_[1]]

    point1 = wall_dict["vertexes_anticlockwise"][0]
    point2 = wall_dict["vertexes_anticlockwise"][1]
    point_x = [point1[0], point2[0]]
    point_y = [point1[1], point2[1]]

    slope, intercept = np.polyfit(point_x,point_y,1)
    origin = [0,0]
    change_point_value = [origin[0] - reference_point_[0],
                        origin[1] - reference_point_[1]]
    reference_point =
             [reference_point_[0]+change_point_value[0],
              reference_point_[1]+change_point_value[1]]

    return slope, change_point_value, reference_point,
           fac_width, wall_vertices_list
```

This tool is designed to extract essential information required for generating the 2D facade and converting it back into 3D after the placement of solar panel modules. It collects the reference point and slope information required for generating the 2D facade, and the 3D orientation of the facade is extracted from its left bottom point. This information is crucial for the placement of the optimized number of solar panel modules in 3D space. Additionally, the tool calculates the 2D positions of all the windows based on the provided data.

Subsequently, the floors within the facade are calculated using a proprietary algorithm as shown in Listing 5. This algorithm considers the list of windows along with their 2D positions and the height of the facade. The algorithm proceeds as follows:

1. Firstly, it calculates all the top and bottom 2D positions of each window.
2. Then, it separates all the bottom and top positions into two distinct lists.
3. Subsequently, the topmost corner positions of a window on a specific floor are computed, and the same applies to the lowermost corner positions.
4. The corresponding topmost corner points of each floor are compared against the bottom corner point. The floor's height is then determined as the average of those two values.

Listing 5. A Python function which returns a list of floors.

```


def get_floors(window_list, height):

    window_corner_list =
    get_top_bottom_points_for_each_window(window_list_all)

    top_corner_list=top_corner_as_list(window_corner_list)
    bottom_corner_list=
    bottom_corner_as_list(window_corner_list)

    max_top_corner_new =
    get_max_corner_points(top_corner_list)

    max_top_corner_new.sort()

    min_bottom_corner_new =
    get_min_corner_points(bottom_corner_list)

    min_bottom_corner_new.sort()

    if max_top_corner_new[0] > min_bottom_corner_new[0]:
        min_bottom_corner_new.pop(0)

    index_list_min = []
    index_list_max = []

    if len(max_top_corner_new)<len(min_bottom_corner_new):
        for index, value in enumerate(max_top_corner_new):
            if value > min_bottom_corner_new[index]:
                index_ =
                min_bottom_corner_new.
                index(min_bottom_corner_new[index])

                index_list_min.append(index_)
        min_bottom_corner_new =
        [i for j, i in enumerate(min_bottom_corner_new)
        if j not in index_list_min]

    elif len(max_top_corner_new)>len(min_bottom_corner_new):
        for index, value in enumerate
                        (min_bottom_corner_new):
            if value > max_top_corner_new[index]:
                index_ =  max_top_corner_new.
                index(max_top_corner_new[index])

                index_list_max.append(index_)

        max_top_corner_new =
        [i for j, i in enumerate(max_top_corner_new)
        if j not in index_list_max]

    else:
         min_bottom_corner_new = min_bottom_corner_new
         max_top_corner_new = max_top_corner_new

    if len(min_bottom_corner_new)>len(max_top_corner_new):
        min_bottom_corner_new.pop(-1)

    floor_list = []
    for (item1, item2) in zip(max_top_corner_new,
                              min_bottom_corner_new):
        floor_list.append((item1+item2) / 2)

    floor_list = np.array(floor_list)
    difflist_floor = np.diff(floor_list)
    difflistbool_floor =
        difflist_floor[difflist_floor<1600]

    index_list_ = []
    for item in difflistbool_floor:

        index= np.where(difflist_floor == item)
        for item2 in index:
            for item3 in item2:
                index_list_.append(int(item3))

    floor_list =
    [i for j, i in enumerate(floor_list)
    if j not in index_list_]
    floor_index_list = []

    for value in floor_list:
        if value < 1600:
            index = floor_list.index(value)
            floor_index_list.append(index)

    floor_list =
    [i for j, i in enumerate(floor_list)
    if j not in floor_index_list]
    floor_list.insert(0, 0.0) # hardcoded
    floor_list.insert(-1,height)
    floor_list.sort()


    return floor_list
```

Then, a text file is generated to provide comprehensive information about the permissible panel and module dimensions, window list within a facade, floors in the facade, as well as the height and width of the facade in 2D, all of which are generated from the workbench. This information is then utilized to implement the solar panel maximization algorithm, which is described in the next section.

2.2. *Solar Panel Optimization Algorithm*

The goal of this section is to explain the Solar Panel Optimization algorithm, implemented using Python. Let us first look at the nature of the problem at hand. Essentially, we are given the following data:

- A finite collection of solar panels that are both photovoltaic and collector.
- A building facade (this information also includes the positions of the various windows and balconies present on the facade.)
- A set of engineering requirements that a panel, if attached, must satisfy.

The goal (it should be immediately clear that we are dealing with 2-dimensional objects), then, is to place the panels on the facade such that the total area covered is maximized. It should be immediately clear, that this is a **Constrained Optimization** problem. The easiest way to generate this maximized placement is by brute-forcing: go through all possible combinations of panel placements and then pick the one that covers the maximum possible area. However, as the number of panels increases and the facades become more complicated, this brute-force approach becomes highly undesirable as it is inefficient and slow. The next best thing is a greedy approach.

2.3. *A Rough Overview of the Algorithm*

We will first employ a "divide and conquer" approach. Given a facade, we will break the optimization problem into smaller chunks.

- **STEP 1:** divide the entire facade into floors. A typical floor is shown in Figure 6.
- **STEP 2:** For each floor, we will use a "dynamic programming" approach: given a floor, we will begin an iteration that will start from the leftmost point of the floor and

end at the right end of the floor. At each stage of the iteration, we will try placing a panel of the largest area from the list of available panels, and progressively work towards the smaller ones.

- **STEP 3:** Once we have placed all the panels, we will create the vertical module divisions, that is, the separation that defines the length of each prefabricated module. The length limitation in ENSNARE project is 6 meters, due to the prefabrication processes of the industrial partners. Each module might contain several windows/panels. These module separations need to adhere to certain engineering requirements—there is an upper limit and a lower limit to the module separations.

Note that we also need to attach registration areas for the panels. These are the yellow rectangles in Figure 7. The goal of the registration areas is to host the pipes and cables of the solar panels. Also, if needed, to have the possibility to open and access from the exterior of the facade these pipes and cables, for instance, in case of emergency.

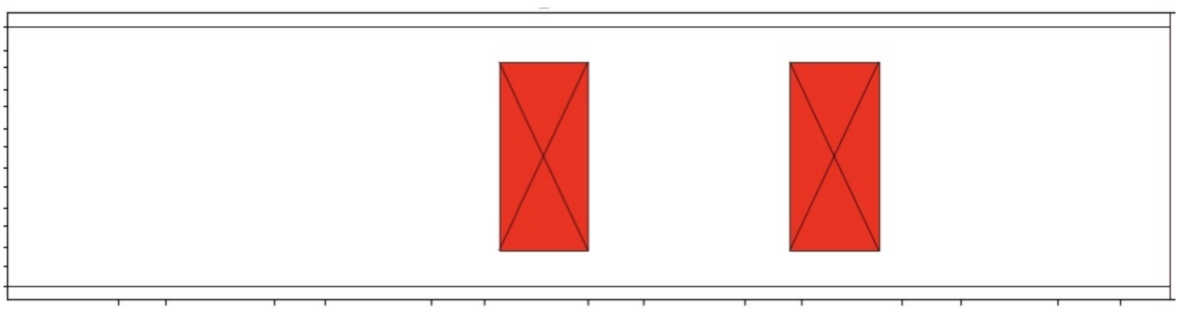

Figure 6. An example of a typical facade section from floor to floor. In this case, the floor has two windows (in red).

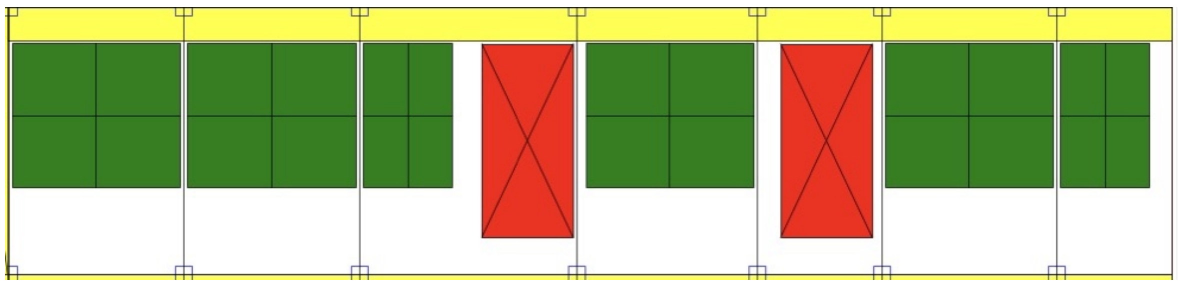

Figure 7. A typical facade section from floor to floor, in this case, with windows and the added prefabricated modules with the registration area in yellow, and the solar panels in green.

2.4. *Pythonic Implementation of Basic Objects*

For the purposes of this algorithm, we will be mostly dealing with objects such as windows, panels, registration areas, etc. All of these objects have a certain distinct physical property in common: they are all rectangular. That is, we need precisely four two-tuples to describe every such rectangular object. It is important to note that this rectangular nature is essential for the algorithm. Since the optimization problem is highly geometric in nature, we will be required to implement a wide array of useful geometric properties. We have implemented **RectangularObject** using dataclasses as described in Listing 6.

The various class properties will be used extensively throughout the code. We have omitted a few class methods in Listing 6 for the sake of brevity. One such class method is **intersects (self,cls)**, described in Listing 7, returns a boolean object to decide if two rectangular objects are intersecting or not:

Listing 6. The base class for all the rectangular objects that we will encounter in this project.

```python

@dataclass(kw_only= True)
class RectangularObject:

    object_type: str = field(init = False)
    object_coordinates: list

    @property
    def area(self) -> float:
        return (self.length)*(self.height)

    @property
    def right_end_point(self) -> float:
        return self.object_coordinates[3][0]

    @property
    def left_end_point(self) -> float:
        return self.object_coordinates[0][0]

    @property
    def top_end_point(self) -> float:
        return self.object_coordinates[1][1]

    @property
    def bottom_end_point(self) -> float:
        return self.object_coordinates[0][1]

    @property
    def length(self) -> float:
        return abs(self.object_coordinates[3][0] - self.object_coordinates
    [0][0])

    @property
    def height(self) -> float:
        return abs(self.object_coordinates[1][1] - self.object_coordinates
    [0][1])
```

Listing 7. Class method to decide if two rectangular objects are intersecting.

```python

#to check if two rectangular objects intersect
def intersects(self,cls) -> bool:
    A = self.left_end_point > cls.right_end_point or cls.left_end_point >
    self.right_end_point
    B = self.bottom_end_point > cls.top_end_point or cls.bottom_end_point >
    self.top_end_point
    if A or B:
        return False
    else:
        return True
```

Further, the use of **dataclasses** increases the performance of the code. Other rectangular objects, such as floors, are first of all child classes of the rectangular object class. They might have additional properties, as Listing 8 demonstrates:

Listing 8. Python implementation of a facade. A facade can consist of floors, panels, windows, etc.

```python

    @dataclass(kw_only= True)
    class Facade(RectangularObject):

        object_type = ENSNARE.FACADE
        has_pitched_roof:bool = False
        has_panels:bool = False
        has_module_divisions:bool = True
        floor_list = []
        pitched_roof:PitchedRoof = field(init = False)
```

As we will see in later sections, the algorithm will act on a facade object.

2.5. Specific Requirements of the ENSNARE Project

The various modules and panels have to adhere to certain engineering requirements which is essential to the manufacturing process, which is given by the manufacturing partners of the project:

- Sizes of the registration areas;
- The length of a given module must be at least the MIN_MODULE_LENGTH and at most MAX_MODULE_LENGTH;
- The sizes of the inner and outer profile areas;
- The panels cannot be of arbitrary size: there is a finite amount of panels of specific lengths and widths.

All this information is contained in the **requirements.py** file. A typical example is demonstrated in Listing 9 (note that the REQUIREMENTS class also inherits from float, since these are floating point numbers).

Listing 9. requirements.py, which contains the various manufcaturing requirements.

```

class REQUIREMENTS(float,Enum):
"engineering requirments for the ENSNARE project"

INNER_PROFILE = 100
OUTER_PROFILE = 40
STEP_SIZE = 10
MAX_MODULE_LENGTH = 3300
MIN_MODULE_LENGTH = 1500


AVAILABLE_PANEL_DATA = [
        (1066, 1756),
        (1756, 1086),
        (1551, 1756),
        (1756, 1551),
        (2036, 1756),
        (1756, 2056),
    ]

class REGISTRATION_AREA_SIZE(float,Enum):
    "sizes of the available registration areas"

    VERTICAL = 400
    HORIZONTAL = 400
```

The facade information, which might comprise floors, windows, and balconies, is provided in the form of a text file. This text file consists of a dictionary, which contains the coordinates of the above-mentioned objects. The general structure of the text file is shown in Listing 10:

Listing 10. A typical facade text file. We have provided a simplified version for the sake of brevity.

```

{
    "facade": {
        "length": 22_000,
        "height": 2600,
        "windows": [
            {
                "xy": (2_200, 400),
                "length": 2_000,
                "height": 1400
            },
            {
                "xy": (5_750, 400),
                "length": 1_950,
                "height": 1400
            } }
```

Before the main algorithm can do anything, we first need to translate this facade information into Pythonic objects. A helper function that converts this into a JSON object (which Python can understand) is implemented in Listing 11:

Listing 11. A helper function to convert the text file into a Python dictionary.

```

def read_from_textfile(file_name:str) -> dict:

    file = open(file_name, "r")
    text_data = file.read()
    dictionary = ast.literal_eval(text_data)
    return dictionary
```

Once this is performed, extracting the Pythonic objects from this is fairly straightforward. This is achieved via the function **return_facade_from_textfile**, which accepts the text file location as an argument and returns a **Facade** object. This is implemented in Listing 12:

Listing 12. Function which reads the facade text file and returns a Python Facade object.

```

def return_facade_from_textfile(text_file_location:str) -> Facade:

#storing the textfile as a python dictionary
facade_data_dict = read_from_textfile(file_name = text_file_location)

#creating the facade
facade_length = facade_data_dict["facade"]["length"]
facade_height = facade_data_dict["facade"]["height"]
facade_coordinates = np.array([(0,0),
                              (0,facade_height),
                              (facade_length,facade_height),
                              (facade_length,0)])

new_facade = Facade(object_coordinates=facade_coordinates)
```

2.6. The Main Algorithm I: Placing Panels

The function responsible for placing panels on a given floor is the **place_panel** function, which accepts a **Floor** object as an argument and returns an object of the same type. The **Floor** class, as described in Listing 13, is a child class of the **RectangularObject** class, with a few further attributes to store the list of windows and panels present on the floor.

Listing 13. Python implementation of a Floor class. This is a simplified version.

```

@dataclass(kw_only=True)
class Floor(RectangularObject):

    object_type = ENSNARE.FLOOR
    window_list:list[RectangularObject]
    floor_module_separation:list
    panel_list:list[RectangularObject]
    registration_area_side:RectangularObject
    registration_area_top:RectangularObject
```

The **place_panel** function places the panel starting from the left endpoint of the floor. The rate at which it moves along the right is given by the step-size parameter. For the sake of clarity, we have broken this up into several snippets as described in Listings 14–17.

Listing 14. A snippet of the panel placing function.

```
def place_panel(floor:Floor) -> Floor:
    #placing the panels with a step size given by the inner profile size
    current_distance = starting_point

    while current_distance < floor.right_end_point:

        #creating a temporary copy for each iteration
        temp_panel_list = [panel for panel in available_panels]
```

At each stage of the iteration, we try to place the panel of the maximum area—this is the greedy approach that we referred to earlier.

Listing 15. A snippet of the panel placing function.

```
#trying to fit all the panels until we run out
while len(temp_panel_list) != 0:
    #starting with the panel of maximum area
    panel = max(temp_panel_list, key=attrgetter('panel_area'))
```

For each panel, we try to see if it fits at the given position of the iteration: we check if it intersects any windows and panels. We also see if it sticks out of the floor.

Listing 16. A snippet of the panel placing function.

```
#instead of checking if the panel touches the windows
#we will check if the panel touches the window shell
modified_window_list = [window.inner_profile_shell for window in floor.
window_list]

#now check if this panel fits
panel_exceeds_floor = test_panel.bottom_end_point < (floor.
bottom_end_point+REQUIREMENTS.OUTER_PROFILE)
panel_touches_shell = any(shell.intersects(test_panel) for shell in
modified_window_list)
panel_exceeds_length = test_panel.right_end_point > (floor.
right_end_point-REQUIREMENTS.OUTER_PROFILE)
```

If it fits, we will add this panel to the list of panels associated with this floor. Otherwise, we discard this panel and see if there are any smaller panels available. If there are no smaller panels available, we move on to the next stage of the iteration.

Listing 17. A snippet of the panel placing function.

```
    #i.e panel does not fit
    if panel_exceeds_floor or panel_touches_shell or panel_exceeds_length
:
        #if this is the smallest panel and it is hitting a window
        if len(temp_panel_list) == 1 and panel_touches_shell:
            #finding the problematic window
            problematic_windows = find_intersecting_objects(test_panel,
modified_window_list)
            new_distance = max(shell.right_end_point for shell in
problematic_windows)
            current_distance = new_distance
            break

        temp_panel_list.remove(panel)
        continue

    #if it fits
    else:
        floor.panel_list.append(test_panel)
        current_distance = current_distance + REQUIREMENTS.OUTER_PROFILE+
test_panel.length
        break

current_distance = current_distance + step_size
```

2.7. The Main Algorithm II: Placing Modules

Once the panels have been placed, a function called the **create_module_seprations** tries to create the module separations: it creates the separations using an object called the **VerticalSeparator** which is implemented in Listing 18. This is simply a vertical line given by the top point, bottom point, and its distance from the y-axis.

Listing 18. A Vertical Separator.

```

    @dataclass
    class VerticalSeparator:

        top:float = 0
        bottom:float = 0
        distance:float = 0
```

The **create_module_seprations**, described in Listing 19, first collects the left/right endpoints of the windows and panels present on the floor.

Listing 19. A snippet of the create_vertical_separations function.

```

def create_module_separation(floor:Floor) -> Floor:
    #First, we collect all the rectangular objects on this floor
    #Then, we collect their right/left end points
    end_point_list = []
    for window in floor.window_list:
        end_point_list.append(window.right_end_point+REQUIREMENTS.
    OUTER_PROFILE)
        end_point_list.append(window.left_end_point-REQUIREMENTS.
    OUTER_PROFILE)
    for panel in floor.panel_list:
        end_point_list.append(panel.right_end_point+REQUIREMENTS.
    OUTER_PROFILE)
        end_point_list.append(panel.left_end_point-REQUIREMENTS.OUTER_PROFILE
    )
```

From this list, the function attempts to create pairings x, y (where x and y are members of this list) in such a way that the following three conditions are satisfied:

1. $x < y$
2. $|x - y| \leq$ MAX_MODULE_LENGTH
3. $|x - y| \geq$ MIN_MODULE_LENGTH

This is implemented in Listing 20 shown below:

Listing 20. A snippet of the create_vertical_separations function.

```

    #Main module separation algorithm
    start_point = end_point_list[0]
    module_separation.append(start_point)
    for current_point in end_point_list:
        current_length = current_point-start_point
        if current_length <= max_mod_length and current_length >=
    min_mod_length:
            module_separation.append(current_point)
            start_point = current_point

    floor.floor_module_separation = module_separation
    return floor

```

So far, the algorithm acts on each floor: it places the panels then it places the modules (i.e., the vertical separations). Once this is performed for each floor, we can combine them to obtain a panel placement on the entire facade.

2.8. Generating Outputs

The output of the solar panel maximization algorithm generates a visual plot of the entire facade. This is achieved using Matplotlib (see Figure 8).

It is also possible to generate a JSON file providing detailed information regarding the 2D coordinates of each module and panel in the facade. Currently, four different designs of prefabricated modules are being used in the project as shown in Figure 9.

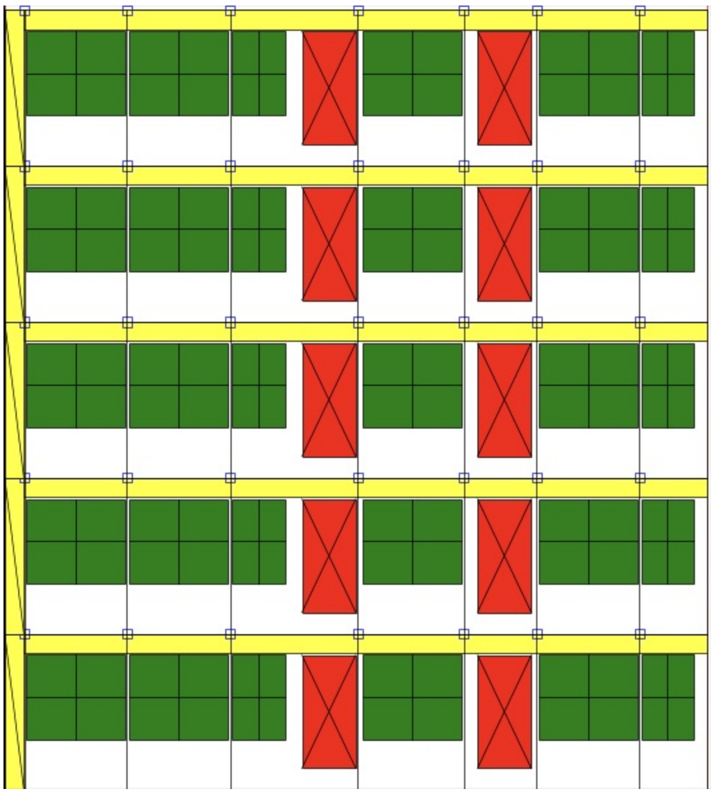

Figure 8. A typical facade with windows in red, and the layout of the added prefabricated modules with solar panels in green and the registration area in yellow. The tiny squares determine the location of the connectors or anchors.

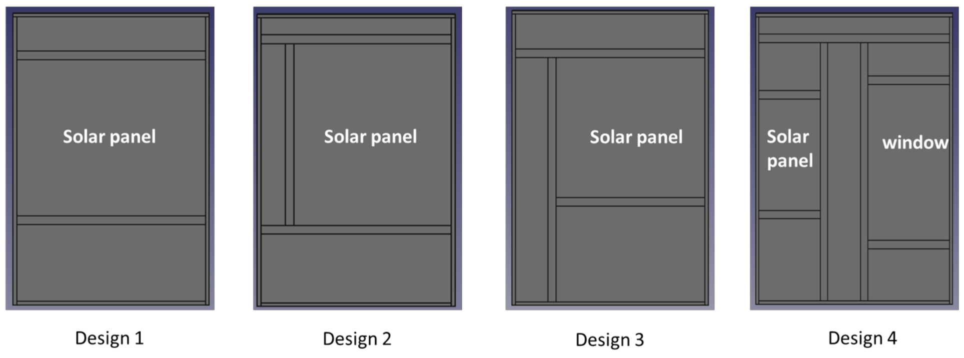

Figure 9. Four prefabricated modules were designed in FreeCAD.

Comparing the module and panel dimensions from the data generated by the algorithm, the most appropriate design (out of the four available designs) is chosen. These four designs are hard-coded using simply Python and FreeCAD tools. Now, the 3D projection algorithm is applied and the already existing 3D model in FreeCAD now gets updated by the solar panel modules as shown in Figure 10.

Furthermore, two Excel files are generated as well. One of them provides the connector positions corresponding to each module and window in the facade. The other file provides intricate details regarding the dimensions of each component of each module in the facade, which are essential for robotic assembly. These data can also provide a total manufacturing cost estimate for the project.

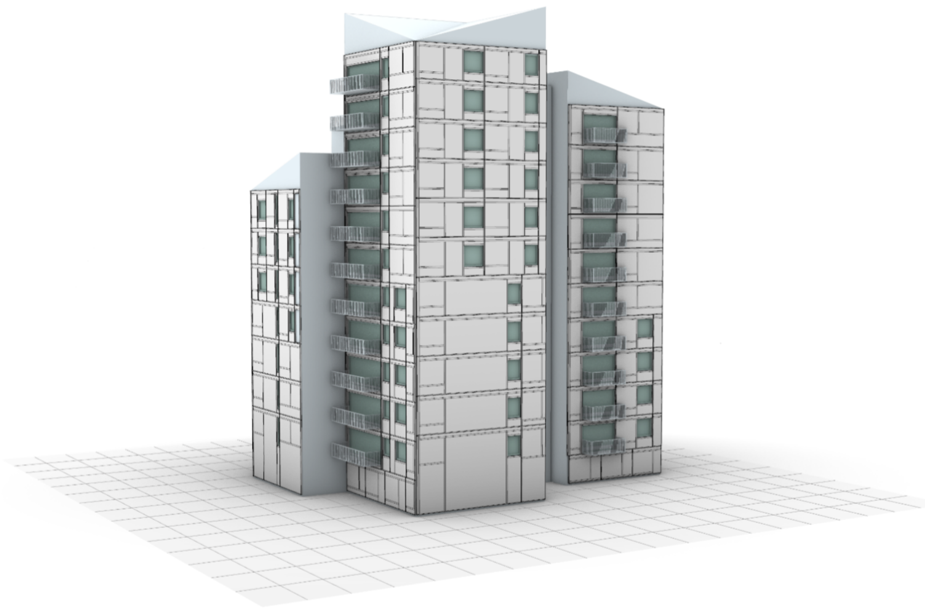

Figure 10. Output of the code described in this section RG1.2, that includes prefabricated layout definition including solar panels and registration areas in the building model of demo-building in Milan, Italy.

3. Addressing RG1.4 by Re-Configuring the Layout of the Prefabricated Modules

In the next step of the building renovation project, the engineering team visits the site and obtains accurate measurements of the building. The technique used in the ENSNARE project is based on using targets on the critical parts of the building, such as the location of the connectors, windows, and other building edges. Once the engineering team has measured the location of the targets (see Figure 11), the building model measurements need to be readjusted, and accordingly, the layout of the prefabricated modules too.

Adjusting the layout of prefabricated modules manually is time-consuming work. The Flowchart described in Figure 12 describes the automated steps to reduce this time.

Data pre-processing may be necessary depending on the structure and orientation of the data provided. The pre-processing steps taken may include generating a transformation matrix to multiply the provided data and transform it into a form compatible with the algorithm. Additionally, it is important to note that the connector positions are provided, rather than the actual window positions on the facade. As a result, it is necessary to determine the correct window dimensions and positions before applying the algorithm. This is achieved by subtracting or adding the connector offset length to the provided data in order to derive the required positions and dimensions of windows.

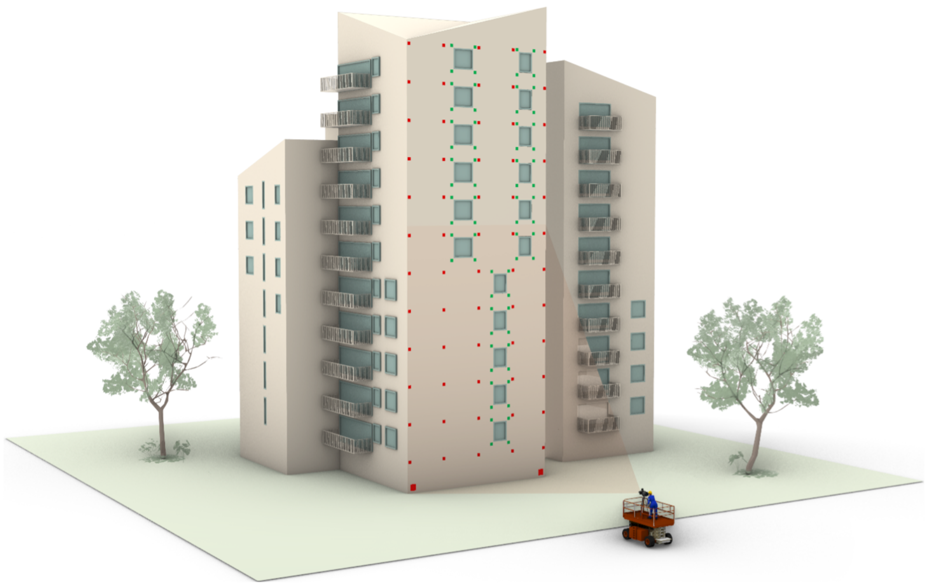

Figure 11. Capturing the real building images with AprilTags, which are located in the critical points of the facade. Example of the demo building in Milan.

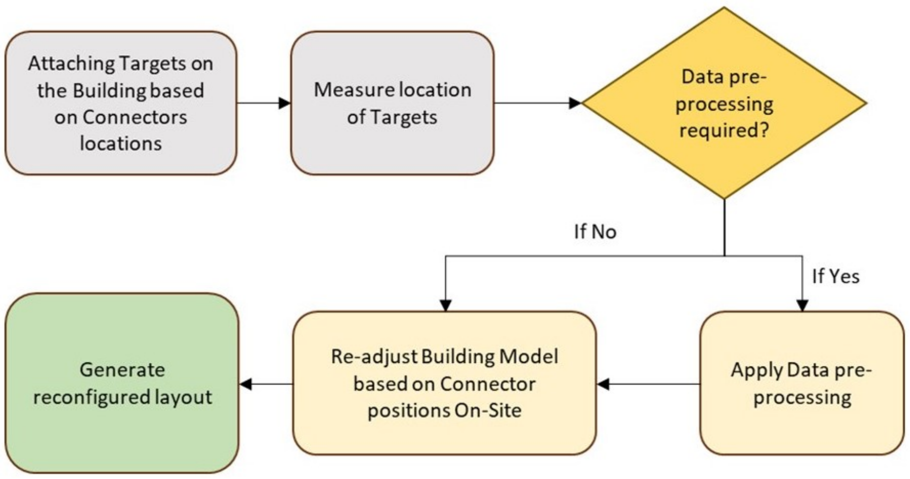
Attaching Targets on the Building based on Connectors locations
Measure location of Targets
Data pre-processing required?
If No
If Yes
Generate reconfigured layout
Re-adjust Building Model based on Connector positions On-Site
Apply Data pre-processing

Figure 12. Flow-chart of the solution RG1.4. The inputs are shown in gray, the outputs in green, and the steps required in yellow.

In order to generate the text file with correct information, the following functions were implemented:

1. First, the data provided needed to be converted into 2D because otherwise the algorithm cannot be applied. Hence, the slope of the facade is determined, along with its width, height, and floors from the provided data. This is achieved through Listing 21 shown below:

Listing 21. A Python function which determines the facade slope.

```

def get_slope_width(panel_list):
    reference_point_3d = [0.0, 0.0, 0.0]
    x_value_ref = 0.0
    y_value_ref = 0.0
    z_value_ref = 0.0
    y_value_list_floors = []
    y_value_start = 0.0
    for item in panel_list:
        y_value = item['y/m']
        if y_value > 0.0:
            y_value = 0.0
            item['y/m'] = 0.0
        if abs(y_value - 0.0) < 1.0:
            x_value_ref =  item['x/m']
            y_value_ref =  item['y/m']
            z_value_ref =  item['z/m']

        if y_value < y_value_start:
            y_value_start = y_value
            x_value_start = 0.0
            if abs(x_value_start - item['x/m']) < 1.0:
                top_most_point_x = item['x/m']
                top_most_point_y = item['y/m']
                top_most_point_z = item['z/m']
        y_value_list_floors.append(abs(item['y/m']*1000))

    point_1 = np.array(reference_point_3d)
    point_2 = np.array([x_value_ref,
                        y_value_ref,
                        z_value_ref])
    point_3 = np.array([top_most_point_x,
                        top_most_point_y,
                        top_most_point_z])

    dist_hor = np.linalg.norm(point_1 - point_2)
    fac_width = dist_hor * 1000

    dist_vert = np.linalg.norm(point_1 - point_3)
    fac_height = dist_vert * 1000

    point_x = [point_1[0], point_2[0]]
    point_y = [point_1[2], point_2[2]]
    slope, intercept = np.polyfit(point_x,point_y,1)

    groups = []
    for item in y_value_list_floors:
        if len(groups) == 0:
            groups.append([item])
            continue
        curr_group = groups[-1]
        if all(abs(item - x) < 10 for x in curr_group):
            curr_group.append(item)
        else:
            groups.append([item])

    floor_list = []
    for floor_level in groups:
        floor_list.append(mean(floor_level))

    floor_list[0] = 0.0
    floor_list[-1] = fac_height

    return slope, fac_width, fac_height, floor_list
```

2. Then, the window list is generated, as shown in Listing 22, from the provided data through the code shown below. Once this window and other important information pertaining to the facade width, height, and floors have been obtained, the text file is generated in order to apply the algorithm.

Listing 22. A Python function which generates a text file containing the list of windows.

```

def generate_window_list(window_info):
    windows_list = []
    for item in window_info:
        individual_window_dict = {}
        if item['y/m'] > 0.0:
            item['y/m'] = 0.0
```

```python

        perpendicular = abs(item['y/m'])
        point = [item['x/m'], item['z/m']]
        ref_point = [0.0, 0.0]
        x_coord = math.dist(point, ref_point)
        individual_window_dict["xy"] =
        (x_coord*1000, perpendicular*1000)
        windows_list.append(individual_window_dict)


    window_final_list = []
    for chunk in chunker_longest(windows_list, 4):
        chunk_list = list(chunk)

        # sort out the x and y values of the
        4 corner points of the window
        x_coords = []
        y_coords = []
        for item in chunk_list:
            x_coords.append(item['xy'][0])
            y_coords.append(item['xy'][1])
        x_coords.sort()
        y_coords.sort()

        # Take the window as the largest approximation
        of all the corner points
        if x_coords[0] >= x_coords[1]:
            LBP_x = x_coords[0]
            LTP_x = x_coords[0]
        else:
            LBP_x = x_coords[1]
            LTP_x = x_coords[1]

        if x_coords[2] >= x_coords[3]:
            RBP_x = x_coords[2]
            RTP_x = x_coords[2]
        else:
            RBP_x = x_coords[3]
            RTP_x = x_coords[3]

        if y_coords[0] >= y_coords[1]:
            LBP_y = y_coords[0]
            RBP_y = y_coords[0]
        else:
            LBP_y = y_coords[1]
            RBP_y = y_coords[1]

        if y_coords[2] >= y_coords[3]:
            LTP_y = y_coords[2]
            RTP_y = y_coords[2]
        else:
            LTP_y = y_coords[3]
            RTP_y = y_coords[3]

        # Calculating the corner points of the window
        Left_Bottom_Point = [LBP_x, LBP_y]
        Left_Top_Point = [LTP_x, LTP_y]
        Right_Bottom_Point = [RBP_x, RBP_y]
        Right_Top_Point = [RTP_x, RTP_y]

        # Creating dictionary for each window
        window_dict_single = {}
        window_dict_single['xy'] = Left_Bottom_Point
        window_dict_single['length'] =
          abs(Left_Bottom_Point[0] - Right_Bottom_Point[0])
        window_dict_single['height'] =
          abs(Left_Bottom_Point[1] - Left_Top_Point[1])

        window_final_list.append(window_dict_single)

    return window_final_list
```

It must be said that the prefabricated module layout is as detailed as for being used for manufacturing purposes, including profiles, enclosure boards, insulation and, of course, solar panels (see Figure 13).

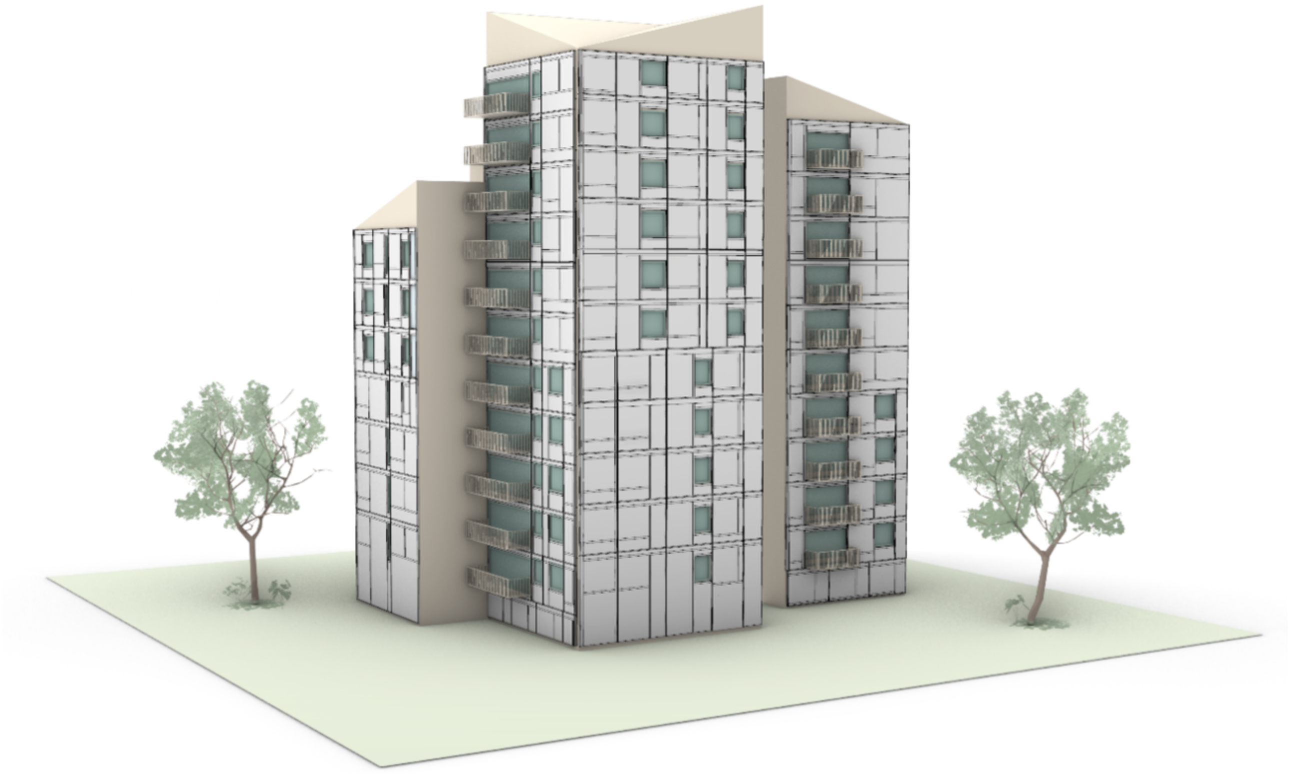

Figure 13. Building model and detailed prefabricated module layout of the demo-building in Milan.

4. Tests and Results of the Solutions

The main objective of the solutions was to reduce time. In order to test the efficiency of the solutions, up to 10 residential buildings have been used to test the aforementioned tools, as explained in Figure 14. Applying the algorithm takes a few minutes, depending on the complexity of the building.

As an example, a complex residential building like the demo building in Milan, which has approximately 3000 m^2 of building envelope was monitored. Defining the prefabricated layout manually can take up to 1020 h (0.34 $\times$ 3000) according to previous manual tests [23]. With the solutions described in this paper, it can take about 25 min in each of the steps (RG 1.2 and RG 1.4).

With this technique, the arduous work of defining the layout of prefabricated modules for facade renovation can be reduced to a minimum. By utilizing these tools, residential building owners, promoters, and engineers can make informed decisions regarding the installation of prefabricated modules with solar panels, thereby ensuring that the renovation project is both economically viable and energy-efficient.

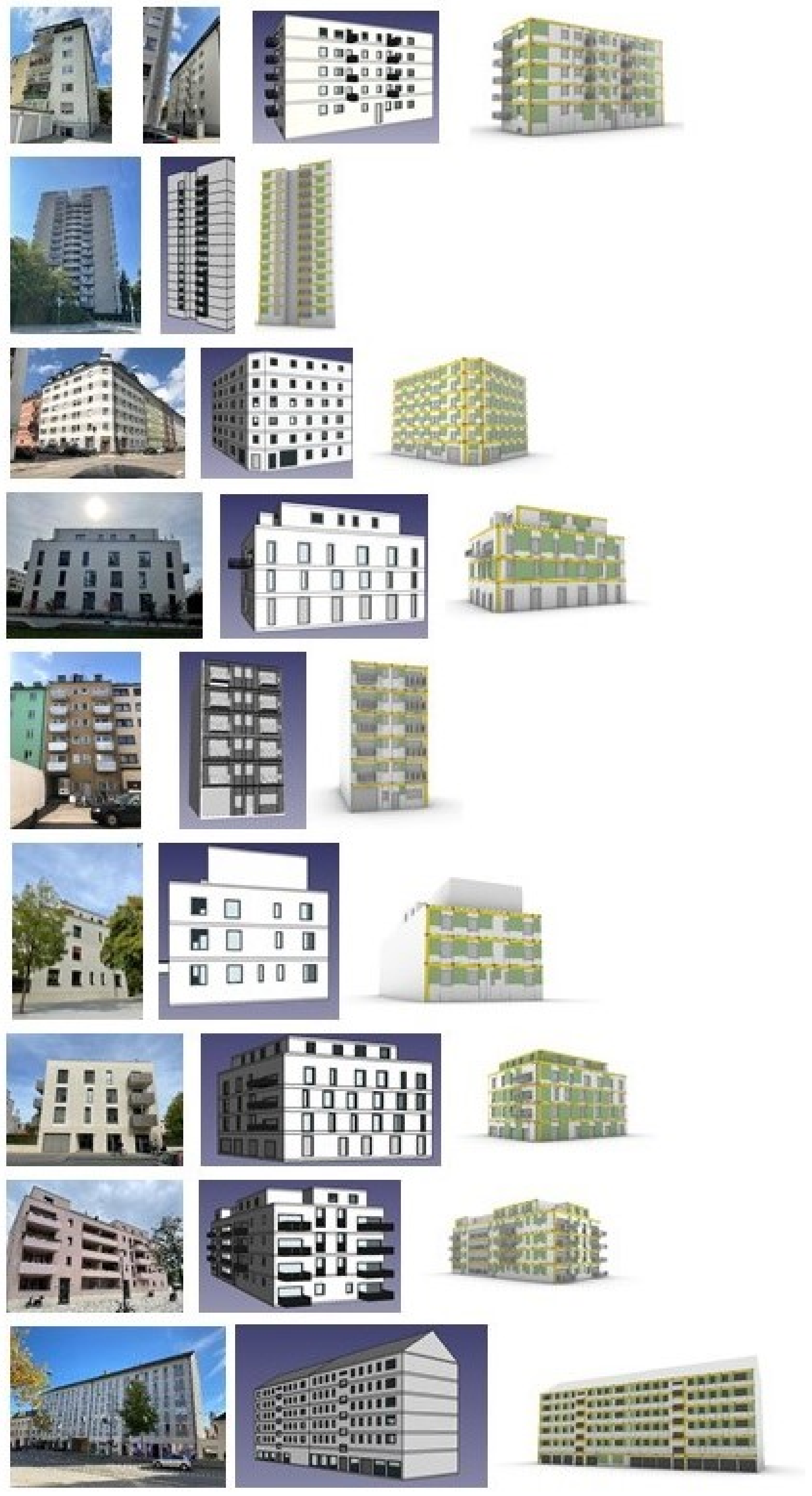

Figure 14. Residential buildings where the tools were applied. In the columns in the left and center, the initial inputs are shown, that is images of the building and the building models. In the column in the right, the building models with the prefabricated module layouts are shown, that is, the output of RG1.2 and RG1.4.

5. Conclusions and Further Developments

The results described in this paper have successfully addressed the research gaps **RG1.2** and **RG1.4**. The time consumed for generating the prefabricated facade module layout is minimal with a commercial and updated computer/processor.

The various tools and techniques introduced will help semi-automate the process of making buildings more energy efficient. Future research will include:

1. Improving the optimization algorithm. Currently, there are certain facades where the algorithm performs poorly. This can be circumvented by trying a different approach, instead of the greedy approach as discussed in this paper. It is possible to re-formulate this problem as an optimal packing problem [26,27].
2. As discussed before, the methods discussed in this paper have not been fully extended to include more diverse building types, such as buildings with pitched roofs. This is one possible avenue for further advancements.

As a final conclusion, it must be commented that the techniques described in this paper will be integrated with an online tool that will include not only the techniques in RG1.1, RG1.2, and RG1.3, but also energy calculations. All this will contribute to reducing the energy consumption of the residential building stock. However, it must be said that these automated procedures developed in RG1.2 and RG1.4 cannot exchange the experience and knowledge of human designers. Aesthetic criteria, for instance, is not considered yet. Therefore, the designs and layouts must always be supervised and, if needed, amended.

Author Contributions: Conceptualization, K.I.; methodology, K.I.; software, S.D., A.S. and K.I.; validation, K.I., S.D. and A.S.; resources, K.I. and T.B.; writing—original draft preparation, S.D. and K.I.; writing—review and editing, K.I. and A.S.; visualization, K.I.; supervision, C.H.; project administration, K.I. and T.B.; funding acquisition, K.I. and T.B. All authors have read and agreed to the published version of the manuscript.

Funding: The research presented in this paper was originally carried out in the context of the H2020 project ENSNARE. This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 958445.

Data Availability Statement: Data are contained within the article.

Acknowledgments: The authors would like to thank Agi Hidri for the assistance with the figures.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

RES	Renewable Energy Sources
CAD	Computer-Aided Design
OSM	Open Street Maps

References

1. European Commission. *Focus: Energy Efficiency in Buildings*; European Commission: Brussels, Belgium, 2020.
2. Garay Martinez, R.; Astudillo Larraz, J. Performance assessment of façade integrated glazed air solar thermal collectors. *Energy Procedia* **2017**, *115*, 353–360. [CrossRef]
3. Urbikain, M.K. Energy efficient solutions for retrofitting a residential multi-storey building with vacuum insulation panels and low-E windows in two European climates. *J. Clean. Prod.* **2020**, *269*, 121459. [CrossRef]
4. Hillebrand, G.; Arends, G.; Streblow, R.; Madlener, R.; Mueller, D. Development and design of a retrofit matrix for office buildings. *Energy Build.* **2014**, *70*, 516–522. [CrossRef]
5. Mateus, D.; Duarte, J. A grammar-based system for building envelope design to maximize PV performance. In Proceedings of the 10th Conference on Advanced Building Skins, Bern, Switzerland, 10–11 October 2016.
6. Mateus, D.; Duarte, J.P.; Romão, L. Energy-Based Design: A Digital Design System for the Design of Energy-Harvesting Building Envelopes. In Proceedings of the XXV International Conference of the Ibero-American Society of Digital Graphics (SIGraDi 2021), Online, 8–12 November 2021. [CrossRef]
7. Stiny, G. Introduction to shape and shape grammars. *Environ. Plan. Plan. Des.* **1980**, *7*, 343–351. [CrossRef]

8. Barco-Santa, A.; Vareilles, É.; Gaborit, P.; Aldanondo, M. Building renovation adopts mass customization: Configuring insulating envelopes. *J. Intell. Inf. Syst.* **2017**, *49*, 119–146. [CrossRef]
9. Tsai, I.; Kim, Y.; Seike, T. Decision-making consideration in energy-conservation retrofitting strategy for the opening of existing building in Taiwan. *AIJ J. Technol. Des.* **2017**, *55*, 963. [CrossRef]
10. Du, H.; Huang, P.; Jones, P. Modular facade retrofit with renewable energy technologies: The definition and current status in Europe, Energy and Buildings. *Energy Build.* **2019**, *205*, 109543. [CrossRef]
11. D'Oca, S.; Ferrante, A.; Ferrer, C.; Pernetti, R.; Gralka, A.; Sebastian, R.; Op't Veld, P. Technical, financial, and social barriers and challenges in deep building renovation: Integration of lessons learned from the H2020 cluster projects. *Buildings* **2018**, *8*, 174. [CrossRef]
12. ENSNARE Consortium. 2021. Available online: https://www.ensnare.eu (accessed on 29 September 2023).
13. *DIN 18202*; Toleranzen im Hochbau–Bauwerke. Deutsches Institut für Normung (DIN): Berlin, Germany, 2013.
14. Skibniewski, M.; Hendrickson, C. Analysis of Robotic Surface Finishing Work on Construction Site. *J. Constr. Eng. Manag.* **1988**, *114*, 53–68. [CrossRef]
15. Balaguer, C.; Abderrahim, M.; Carlos, B.; Mohamed, A. Trends in robotics and automation in construction. In *Robotics and Automation in Construction*; InTech: Rijeka, Croatia, 2008. [CrossRef]
16. Warszawski, A. Economic implications of robotics in building. *Build. Environ.* **1985**, *20*, 73–81. [CrossRef]
17. Hu, R.; Iturralde, K.; Linner, T.; Zhao, C.; Pan, W.; Pracucci, A.; Bock, T. A Simple Framework for the Cost–Benefit Analysis of Single-Task Construction Robots Based on a Case Study of a Cable-Driven Facade Installation Robo. *Buildings* **2021**, *11*, 8. [CrossRef]
18. Pan, M.; Linner, T.; Pan, W.; Cheng, H.; Bock, T. Structuring the context for construction robot development through integrated scenario approach. *Autom. Constr.* **2020**, *114*, 103174. [CrossRef]
19. Pan, M.; Linner, T.; Pan, W.; Cheng, H.; Bock, T. A framework of indicators for assessing construction automation and robotics in the sustainability context. *J. Clean. Prod.* **2018**, *182*, 82–95. [CrossRef]
20. Pan, M.; Linner, T.; Pan, W.; Cheng, H.; Bock, T. Influencing factors of the future utilisation of construction robots for buildings: A Hong Kong perspective. *J. Build. Eng.* **2020**, *330*, 101220. [CrossRef]
21. BERTIM, D2.5. Efficient Mass Manufacturing and Installation of Prefabricated Modules. 2016.
22. Hephaestus Consortium, Hephaestus–EU H2020 Project. 2017. Available online: https://www.hephaestusproject.eu (accessed on 29 September 2023).
23. Iturralde, K.; Gambao, E.; Bock, T. Compilation and assessment of automated façade renovation. In Proceedings of the 38th International Symposium on Automation and Robotics in Construction (ISARC), Dubai, United Arab Emirates, 2–4 November 2021. [CrossRef]
24. Iturralde, K.; Shen, J.; Bock, T. AprilTag detection for building measurement. In Proceedings of the 40th International Symposium on Automation and Robotics in Construction, Chennai, India, 3–9 July 2023; pp. 589–592. [CrossRef]
25. The FreeCAD Team, FreeCAD Your Own 3D Parametric Modeler, (n.d.). Available online: https://www.freecadweb.org (accessed on 29 September 2023).
26. Huang, E.; Korf, R.E. Optimal Rectangle Packing: An Absolute Placement Approach. *J. Artif. Intell. Res.* **2013**, *46*, 47–87. [CrossRef]
27. Barco Santa, A.; Fages, J.G.; Vareilles, E.; Aldanondo, M.; Gaborit, P. Open Packing for Facade-Layout Synthesis Under a General Purpose Solver. In Proceedings of the CP 2015—21st Interna- tional Conference on the Principles and Practice of Constraint Programming, Cork, Ireland, 31 August–4 September 2015; Volume 9255, pp. 508–523. [CrossRef]