PrismArch

Deliverable No D2.2

Integration-ready version of AI algorithms to traverse the parametric solution space

Project Title:	PrismArch - Virtual reality aided design blending cross-disciplinary aspects of architecture in a multi-simulation environment
Contract No:	952002 - PrismArch
Instrument:	Innovation Action
Thematic Priority:	H2020 ICT-55-2020
Start of project:	1 November 2020
Duration:	24 months

Due date of deliverable:	October 31st, 2021
Actual submission date:	November 4th, 2021
Version:	1.1
Main Authors:	Antonios Liapis, Konstantinos Sfikas, Georgios N. Yannakakis

Project funded by the European Community under the H2020 Programme for Research and Innovation.

Deliverable title	Integration-ready version of AI algorithms to traverse the parametric solution space
Deliverable number	D2.2
Deliverable version	1.1
Contractual date of delivery	31 October, 2021
Actual date of delivery	4th November, 2021
Deliverable filename	PrismArch_D2.2_v1.1.pdf
Type of deliverable	Software
Dissemination level	Public
Number of pages	36
Workpackage	WP2
Task(s)	T2.2
Partner responsible	UoM
Author(s)	Konstantinos Sfikas (UoM), Antonios Liapis (UoM), Georgios N. Yannakakis (UoM)
Editor	Konstantinos Sfikas (UoM)
Reviewer(s)	Spiros Nikolopoulos (CERTH), George Adamopoulos (AKT), Dimitrios Ververidis (CERTH)

Abstract	This document describes the integration-ready version of AI algorithms to traverse the parametric solution space. The software deliverable includes the generative algorithms which explore the parametric space guided by constraints, objectives and diversity measures defined in D2.1. The algorithms can generate solutions from scratch, or adapt the user's own

	designs, using artificial evolution, constrained optimization, and quality-diversity search.
Keywords	Artificial Intelligence, Parametric Design, Possibility Space, Design Intelligence, Spatial Analytics, Fitness Functions, Constraints, Evolutionary Algorithms, Diversity Measures

Deliverable history

Version	Date	Reason	Revised by
0.1	21/12/2020	Table of Contents	CERTH, AKT
0.2	1/10/2021	Beta version	UoM
0.3	15/10/2021	Elaborated version incorporating the core aspects of the deliverable	UoM
0.4	22/10/2021	Version ready for internal review	UoM
1.0	29/10/2021	Internal review and comments	CERTH, AKT
1.1	04/11/2021	Final revised version, ready for external review	UoM

List of abbreviations and Acronyms

Abbreviation	Meaning
UoM	University of Malta (UNIVERSITA TA MALTA)
CERTH	Center for Research & Technology Hellas (ETHNIKO KENTRO EREVNAS KAI TECHNOLOGIKIS ANAPTYXIS)
AKT	AKT II Limited
ToC	Table of Contents
DoA	Description of Action
WP	Work Package
IP	Intellectual Property
NDA	Non-Discolsure Agreement
AEC	Architecture, Engineering and Construction
VR	Virtual Reality
AR	Augmented Reality
BIM	Building Information Modelling
CAD/CAM	Computer-Aided Design & Computer-Aided Manufacturing
MAP-Elites	Multidimensional Archive of Phenotypic Elites [4]
QD	Quality-Diversity
EA	Evolutionary Algorithm
EC	Evolutionary Computation

Executive Summary

Deliverable D2.2 describes the algorithmic library for design-driven exploration of the space, following design principles and preliminary research detailed in D2.1 [18]. D2.2 has been written with the feedback of AEC partners in dedicated workshops, including those described in D2.1 [18].

Chapter 1 (Introduction) includes the description of work of this deliverable as well as its relation to other deliverables. Furthermore it explains how the specific approach of this deliverable has been shaped, and will continue to be shaped, in cooperation with the AEC industry partners, so as to facilitate the different perspectives, as well as the designers' subjective criteria.

Chapter 2 (Designer-Driven Exploration of the Parametric Solution Space) is a high-level description of the modes of interaction between the end-user (the designer) and the Quality-Diversity-driven algorithmic design-assistance.

Chapter 3 (Implementation) describes the computational framework that has been developed in order to support the back-end of the aforementioned modes of interaction. The framework includes a large number of algorithms, ranging from simple Evolutionary Strategies to various "flavors" of Quality-Diversity algorithms, thus providing a lot of flexibility for the upcoming integration steps. Furthermore, the framework includes a large number of benchmarks: These include: 1) An example of constrained, numerical optimization, 2) Evolvable Binary Tables, 3) Evolvable Topologies and 4) Evolvable Geometry with Topological and Other Constraints.

A large amount of effort has been devoted to ensuring the extensibility of the computational framework. A modular architecture has been deployed which allows all available algorithms to be applicable to all the benchmarks and also enables the easy and fast extension of the framework's functionality, or the definition of new benchmarks.

Table of Contents

1. INTRODUCTION

This section describes the overall process and considerations that led to the formation of this deliverable. These include the prescribed Description of Work, the ongoing development of other relevant deliverables, the ongoing discussions with the AEC partners, the focused workshops on Quality Diversity and Designer modeling that took place in June 2021 and the AEC partners' specific feedback through the relevant questionnaires.

1.1 Description of Work and relevant deliverables

As prescribed in the DoA and, more specifically, the descriptions of T2.2 and D2.2, this software deliverable *"includes the generative algorithms which explore the parametric space guided by constraints, objectives and diversity measures defined in D2.1. The algorithms can generate designs from scratch, or adapt existing designs using artificial evolution, constrained optimization and quality-diversity search."* Since the algorithms' operation is general-purpose, and not tied to any specific problem, the computational framework of deliverable includes several benchmarks appropriate for benchmarking their performance in different settings, ranging from numerical optimization and simplistic design problems to design problems that approach a realistic degree of complexity.

The developed computational framework is based on the ground-work that was implemented in D2.1 [18], including the extended literature review and the valuable feedback from the AEC partners. The intended interaction, detailed in Section 2, is aligned with AEC requirements as described in D1.1 [19] and D1.2 [20] and accounts for cognitive requirements of VR-aided environments detailed in D3.1 [17]. The currently implemented algorithms and benchmarks provide a large flexibility for the up-coming step of integrating the algorithms in an interactive setting, in the VR-space of the PrismArch platform as part of D5.2 [20]. Furthermore, the modular architecture of the developed framework (especially the template-based and extendable evaluation methods detailed in Section 3.3) facilitates its future extension and the generation and exploitation of Designer Modeling methods that can enhance its operation, as part of D2.3 [20].

1.2. Workshops with AEC partners, surveys and general feedback

While the deliverable document D2.1 [18] ("Initial version of parametric design space") was being developed, a large number of discussions with the AEC partners took place, aiming to clarify the role that Quality Diversity (QD) algorithms could, or should, play in the PrismArch platform, both conceptually and technically. In the same period, a series of workshops was organized by the University of Malta's development team, whose aim was to clarify the capabilities of QD algorithms and their potential applications in PrismArch. We presented examples of how QD has been applied to design problems, as well as examples where the designers' agency is retained and works in cooperation with QD. Finally, we showcased how the interaction between the designers and the QD systems can be used as the basis for applying Designer Modeling in the near future.

After the workshops took place, we asked the AEC partners to take part in a survey (through a relevant questionnaire) that would help us to analyze the different perspectives and take all points of view into consideration. The AEC partners' answers, the relevant discussions, as well as the direct feedback that they gave on D2.1 [18] have helped us in shaping an approach that respects different perspectives that exist in the industry and also exploits as best as possible the designers' expertise and existing methodologies. The partners' specific feedback to the aforementioned questionnaire has been embedded in the deliverable document D2.1 [18], for future reference.

The main outcomes of the workshops, questionnaires and discussions can be summarized in the following shared vision: The design-assistance system, which is based on Quality Diversity algorithms should be formed as a system that helps the designers to explore the design-space more efficiently, by providing them with design examples that satisfy the specified design constraints, while being diverse along a number of dimensions that are selected by the designer. At the same time, the system's operation should be guided by the designer's intentions, preferences and overall judgement. This way, the proposed use of QD forms a dynamic cooperation between the designer and the AI, resulting in a novel form of design experience.

Furthermore, as part of Task T2.3 and during the development of D2.3 [20], the data-traces of the designers' activity will be used for Designer Modeling, i.e. "the implementation of machine learning and statistical models for the purpose of identifying patterns in the needs and goals of different types of designers, and also for learning the preferences of individual designers. These models will be used to adjust, filter, and discern when to show AI-generated suggestions (T2.2) to designers in order to maximize their impact to the task at hand."

2. DESIGNER-DRIVEN EXPLORATION OF THE PARAMETRIC SOLUTION SPACE

This section is a technical view of the algorithmic selection that underlies the interaction that is described in the previous section.

2.1 Intended Interaction

As explained in the previous section, the designers' agency is an important element of the design process. The proposed and implemented methods attempt to enhance this agency via interactive forms of Quality Diversity which can be further enhanced in the near future, through Designer Modeling. Using the current implementation, two different modes of interaction can be implemented in the integration process: The first one, referred to as "Direct Interaction" is one where the designer has direct access to all of the search algorithm's parameters. The second one, referred to as "Indirect Interaction", is one where the algorithm's operation is indirectly affected through the designer's agency which is expressed in the form of sequential selections (preferences) from a set of provided solutions. Any of these two modes of interaction can be further enhanced, in the near future, by merging it with methods of Designer Modeling. Given the discussions with the AEC partners, the indirect mode of interaction seems to be the preferred one. However, for the sake of completeness, the following sections cover both cases (direct / indirect), as well as a preview of the potential model-based extension.

2.1.1 Direct Interaction

In the direct mode of interaction, the designer is placed in a position where they have full control of the QD algorithm's parameters. This mode of operation exposes the largest amount of control to the designer, effectively placing them in a position where they can analyze the solution space of the problem at hand from a very broad point of view. The types of the options that are exposed to the designer in this mode of operation include, but are not limited to, the following: 1) The ability to select the behavioral characterizations that define the diversity of the population of solutions. 2) The ability to select a fitness function that biases the algorithm's search towards a specific quality. 3) The ability to provide the algorithm with an initial set of solutions of their own preference (commonly referred to as a "seed"), thus manually assisting the algorithm in finding optimal solutions. Needless to say that in this mode of interaction the designer should have at least a high-level understanding of Evolutionary Computation and Quality Diversity and a basic understanding of the exposed parameters' effect on the algorithm's operation. Should those conditions be met, the designers may engage in an interactive and exploratory relation with the algorithm, where they can observe and analyze the newly generated knowledge about the design-space.

Exposing the algorithm's parameters to the user, via a proper interface, is a design problem in itself. The main element of the required UI for this approach is an overview of the current "population" of solutions that the QD algorithm has managed to generate, so far. The MAP-Elites[4] algorithm, apart from being efficient, is also convenient in this case. Following its internal functionality, the population of solutions can be presented in the form of a grid where the coordinates of an individual correspond to a pair of values of the selected behavioral dimensions. The visual representation of the every generated design may be in the form of 2D images or 3D models, depending on the computational cost or the designer's preference. When evolving geometrical structures relevant to architectural or engineering problems, the ideal mode of presentation would probably be to lay out the generated 3D models on a horizontal surface in the 3D space and allow the designer to scale up the results so that they can observe the details of a single solution. Since the operation of the MAP-Elites algorithm (similar to any other stochastic-search-algorithm) can be computationally expensive, it is essential that the computational process is running on a separate thread, so as not to freeze the application’s operation. During the algorithm’s operation, the set of generated solutions that are presented to the designer can be updated at specified generation or time-intervals. The algorithm’s parameters can be exposed to the user through a control panel that can be separate from the “presentation table”. The numerical parameters of the algorithm (such as the number of generations or the number of subdivisions per behavioral dimension) can be exposed in the form of sliders. The remaining properties (such as the selection of a fitness function and the behavioral dimensions) can be exposed in the form of drop-down lists. FInally, as soon as the designer decides to stop the evolutionary process, they should be able to select the generated solutions that they prefer, store them for later re-use, or convert them to a commonly used drawing file type (as shown in Figure 3), such as an .obj file, a Rhino 3D model or a Revit BIM file, so that they can further refine the solution using other methodologies.

2.1.2 Indirect Interaction

In the Indirect Interaction mode, the algorithm’s functionality remains - to a large degree - hidden from the designer, sparing them the implementation details and algorithmic parameters, and letting them focus on the characteristics of the generated designs. The designer, in this case, affects the algorithm’s operation by expressing their preference and making selections among sets of provided, diverse solutions.

The interaction between the designer and the algorithm, in this case, is continuous and based on a sequential decision making process that the designer is involved in. Initially, the algorithm generates a diverse set of solutions, based only on the problem definition and a number of behavioral characterizations. The number of generated solutions should be relatively small, as the designer must be able to compare them and select the ones they prefer based, in part, on their own, subjective criteria. The selected solutions form the basis for the next set of generated solutions, in an iterative fashion. More precisely, the algorithm will generate a set of diverse solutions, using the designer’s selections as the initial population (seed).

The User Interface for this approach can take many possible forms. A simple approach would be to present only the current population of solutions to the designer and completely replace it with a new set, as soon as the designer confirms their choice. Alternatively, the user interface could visualize the complete interactive process (as shown in Figure 1), in the form of a directed graph, within the 3D space of the PrismArch platform. One of the benefits of the latter is that the experience of interacting with the QD algorithms would be more in-line with the general spirit of the PrismArch platform (some aspects of which are illustrated in Figure 3), which - among other things - aims to exploit the benefits of a VR-immersed experience to the best degree possible. Furthermore, representing the interaction in the form of a 3D graph would also facilitate the ability to revert to previous designs and explore alternative routes, thus searching the design space with more flexibility.

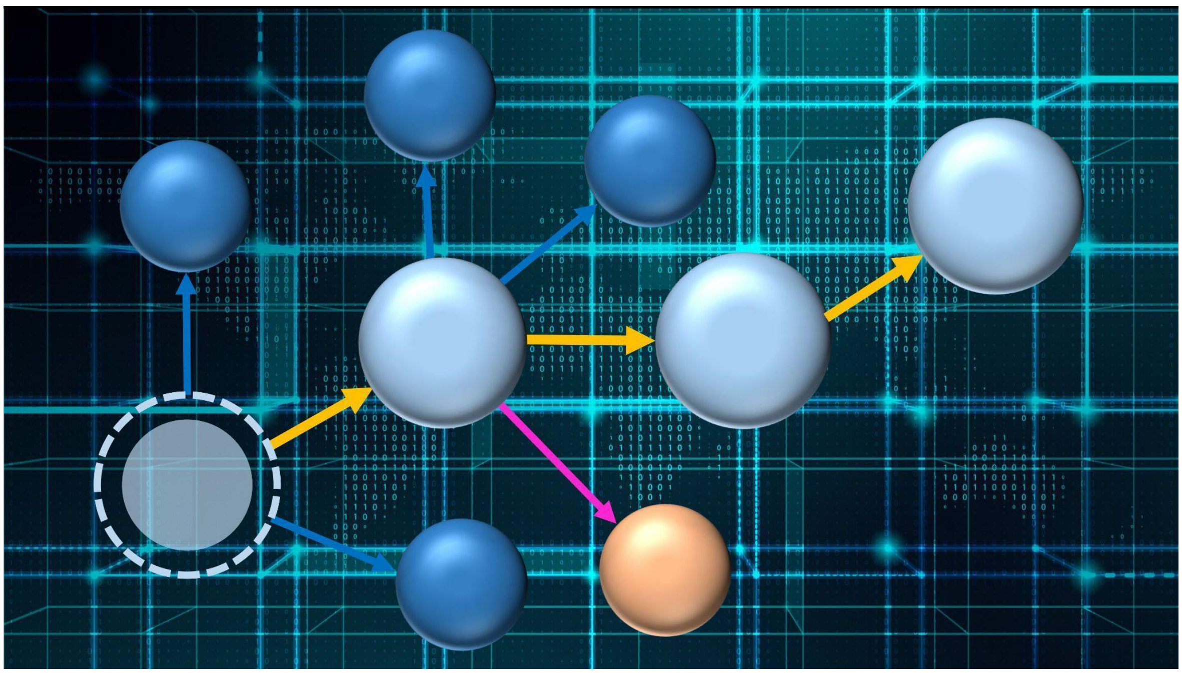


Figure 1: A potential view of the Indirect Interaction between the designer and the QD algorithm. Starting from a problem specification (dashed circle on the bottom-left), the system initially generates a small population of diverse solutions (represented abstractly as spheres in this diagram). From them on, the designer's choices guide the evolutionary process towards their preferred direction, while the QD algorithm keeps diversifying the solutions, providing them with interesting alternatives along the way.

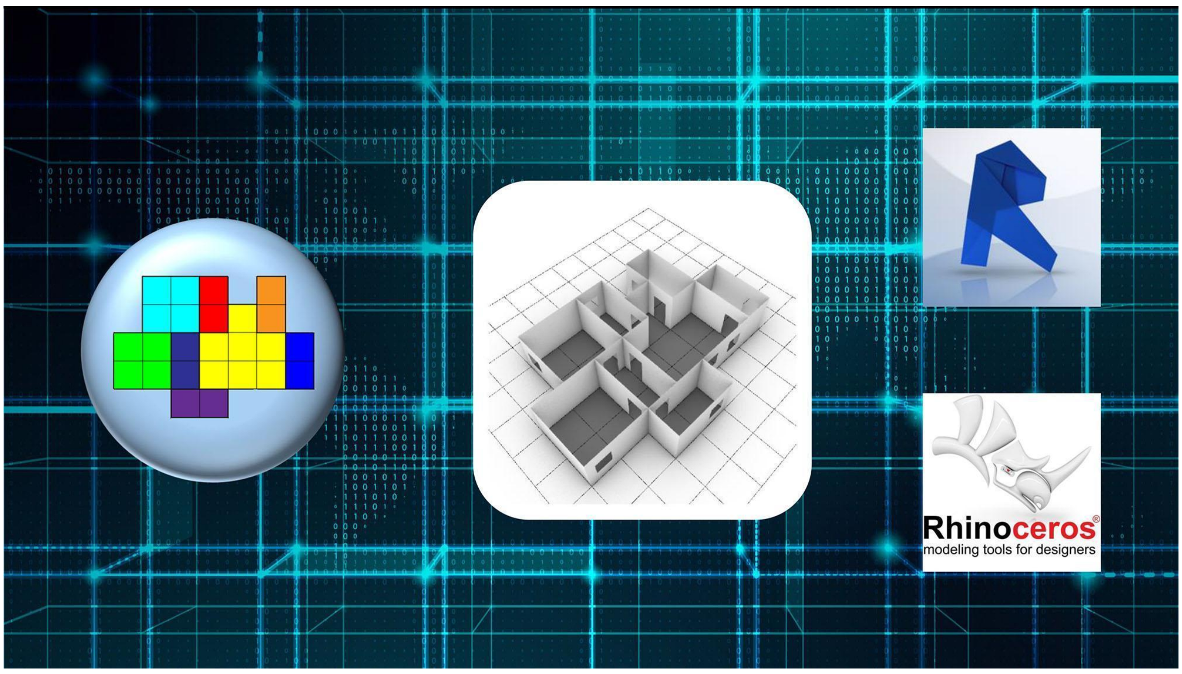


Figure 2: Illustration showcasing that the generated architectural designs can be visualized in 2D or 3D modes, as well as converted to editable files for the design software that PrismArch incorporates.

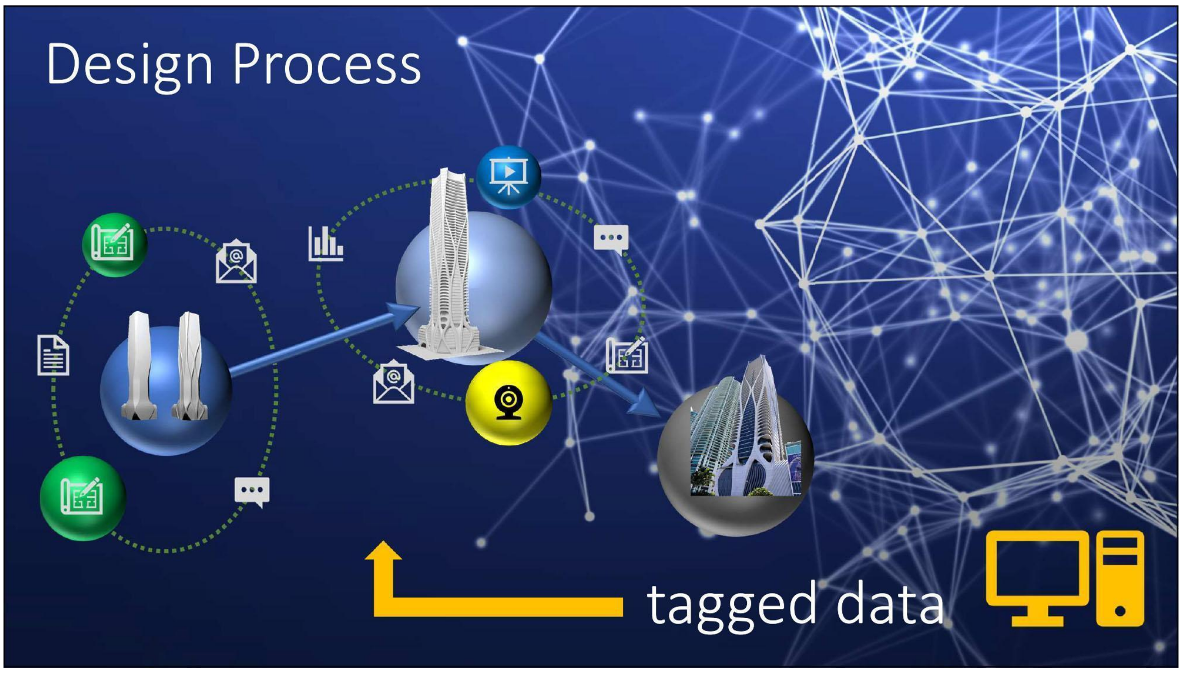


Figure 3: A potential way of visualizing the design process of a project, from a historical perspective. The node on the left represents the concept-phase, and the right-most node represents the materialized project. The interaction between the designers and the QD algorithms (shown in Figure 1) could be part of the concept phase, when the largest part of abstract design-exploration usually occurs.

2.1.3 Model-Based Interaction

The interaction between the designer and the algorithm, especially in the indirect mode, is a process that is heavily based on the designer's initiative. The continuous agency of the designer in the evolutionary process ensures that the generated results tend towards their preferred "direction" within the space of potential solutions to the problem at hand. However, the fact remains that the algorithm generates diversity across a number of predefined Behavioral Characterizations (BCs), while the designer is making choices based on their own, subjective criteria, whose relation to the BCs, or to other characteristics of the solutions is unknown and cannot be defined a-priori in a computable manner.

One of the possible ways of applying designer-modeling in the context of PrismArch is to bridge this gap between the QD algorithms' operation and the designer's initiative. More precisely, the designer's preferences, captured through a large number of discrete choices, can be used to generate models (for example in the form of Neural Networks) that capture some parts of their subjective criteria of selection or preference via machine learning methods. Once such models have been generated, they can be incorporated back into the interactive evolution processes, enriching the dimensions of diversity with the ones that are more aligned with the subjective criteria of human designers.

The concrete specification and actual implementation of this methodology is going to be developed and reported in the context of deliverable D2.3[20].

3. IMPLEMENTATION

3.1. Technical Details about the developed Framework

The source-code of the developed framework can be found on PrismArch's gitlab repository (https://gitlab.com/prismarch-h2020/ai-generative-algorithms)[1]. The framework has been developed in the C# programming language (.Net Framework 4.8, language version: v7.3), using Microsoft's Visual Studio 2019 IDE. The solution is divided in a set of interrelated projects, each of which addresses a clear and specific goal. This subdivision has been applied for reasons of code-clarity and maintainability. Furthermore, the included projects are of three types: 1) Libraries that address specific computational problems of the project, 2) Unit Test projects that perform automated tests for critical aspects of the developed code and 3) Console Applications that are used to run experiments, to test the overall behavior and performance of the developed algorithms and benchmarks, or to showcase the usability of specific parts of the framework. The following list presents all the developed libraries and other types of projects that are included in the framework.

- **Libraries:**
    - **Algorithms:**
      A library that contains the definition of a number of algorithms. Their description can be found in section 3.2.
    - **Benchmarks:**
      A library that contains the concrete implementations of all the developed benchmarks, including the definition of their data-structure, generation methods, mutation methods, evaluation methods etc. so that any of them can be addressed by any of the available algorithms. The Constrained Numerical Optimization benchmark is kept in a separate library, due to its relatively large size and complexity. The description of all the implemented benchmarks can be found in section 3.4.
    - **Constrained_Optimization_CEC2006:**
      A library that contains the definition of all the Constrained Numerical Optimization problems that are found on the relevant benchmark [16].
    - **EC__Abstract_Classes:**
      A library that contains the abstract classes that any specific benchmark must implement, in order to be addressable by any of the provided algorithms. The description of these abstract classes can be found in section 3.3.
    - **Common_Tools:**
      A general-purpose library that includes a large number of language extensions and utilities. This library is being developed as the project progresses, incorporating parts of reusable code in an organized manner.
- **External Libraries:**
    - **Triangle:**
      A geometric utilities library that provides computational methods for Voronoi tessellations and Delaunay triangulations of point-clouds, in 2D. The library is licenced under the MIT license.
    - **Newtonsoft Json:**
      A Json serialization and deserialization library that is installed on some of the

[1] Access to PrismArch's gitlab repositories is restricted and requires authentication by the PrismArch team.

framework's projects, installed through Visual Studio in the form of a Nuget package.
    - **SkiaSharp:**
      A 2D-Visualization library that is used mainly for debugging and testing purposes, across various parts of the project, installed through Visual Studio in the form of a Nuget package.
- **Unit Tests:**
    - **UTEST__Use_Cases:**
      Unit tests for the "Use_Cases" project.
    - **UTEST__Constrained_Optimization_CEC2006:**
      Unit tests for the Constrained_Optimization_CEC2006 project.
    - **UTEST__Common_Tools:**
      Unit tests for the Common_Tools project.
- **Console Applications:**
    - **TEST__Evolvable_Binary_Tables:**
      A console application project that showcases the application of various algorithms to the problem of evolving Binary Tables.
    - **TEST__Evolvable_Topologies:**
      A console application project that showcases the application of various algorithms to the problem of evolving Undirected Graphs.
    - **TEST__Evolvable_Geometries:**
      A console application project that showcases the application of various algorithms to the problem of evolving Geometries with Topological and Other Constraints.
    - **EXP__Constrained_Optimization_CEC2006:**
      A set of computational experiments that test the performance of the Constrained Monte Carlo Elites algorithm (which can be found in the Algorithms project) on the included Constrained Numerical Optimization problems (which can be found in the Constrained_Optimization_CEC2006 project).

3.2 Algorithms

The implemented Framework offers a variety of evolutionary algorithms. Its main focus lies on Quality-Diversity algorithms, but it also includes a classic Genetic Algorithm, as well as other, relatively simple stochastic search and optimization methods. The reason for including those extra algorithms is that they can be used as baselines in potential future tests or experiments.

The following list describes the implemented algorithms that are included in the developed computational framework.

- **(1+1) evolutionary strategy:**
  A single-objective evolutionary approach that operates on a "population" of one individual.
- **(1+1) constrained evolutionary strategy:**
  A variation of the (1+1) algorithm which can also be applied to constrained problems.
- **Genetic Algorithm:**
  A classic, single-objective, population-based optimization algorithm which uses

crossover and mutation to evolve the population and selects individuals in a fitness-proportionate manner.

- **MAP-Elites:**
  The MAP-Elites algorithm *"provides a holistic view of how high-performing solutions are distributed throughout a search space. It creates a map of high-performing solutions at each point in a space defined by dimensions of variation that a user gets to choose. This Multi-dimensional Archive of Phenotypic Elites (MAP-Elites) algorithm illuminates search spaces, allowing researchers to understand how interesting attributes of solutions combine to affect performance, either positively or, equally of interest, negatively. For example, a drug company may wish to understand how performance changes as the size of molecules and their cost-to-produce vary"* [4].
  which is one of the main approaches on Quality-Diversity optimization
- **Monte-Carlo-Elites:**
  The Monte-Carlo-Elites algorithm *"extends the functionality of MAP-Elites, by treating the selection of parents as a multi-armed bandit problem. Using variations of the upper-confidence bound to select parents from under-explored but potentially rewarding areas of the search space can accelerate the discovery of new regions as well as improve its archive's total quality"* [7].
- **Constrained MAP-Elites:**
  A variation of the MAP-Elites algorithm that can be applied to constrained optimization problems. The inclusion of this variation in the framework is crucial, as design-problems are, more often than not, evaluated based on some hard constraints. The implementation is a custom-developed variant that is inspired by similar approaches [14, 15, 5, 6], but slightly modified to better suit the available benchmarks.
- **Constrained Monte-Carlo-Elites:**
  An extended version of Constrained MAP-Elites that also applies parent-selection optimization.

All algorithms have been implemented as abstract methods and, as such, they can be applied / tested on any of the provided benchmarks.

3.3 Prerequisite abstract classes

In order for any specific problem (benchmark) to be addressable by one of the available evolutionary algorithms, a set of abstract classes must be implemented. Namely, one must define the state representation (genotype), the mutation method(s), the evaluation methods (including fitness functions and behavioral characterizations) and the constraint evaluation methods. The abstract classes for all of these elements can be found in the "EA_Shared_Elements" project.

The following list explains all of those classes in some detail.

- **Data Structure:**
  A "Data Structure" is the base-class for any specific solution representation. Every one of the provided benchmarks concretely implements this class, defining its internal data-representation, a number of low-level operations, as well as a number of exposed properties and methods that can give the developer a quick access to many types of information about the object.
- **Generation Method:**
  A method that can generate random solutions to the problem at hand. Its

implementation depends on the selected state representation for the problem. For example, if the individual's genome is represented as a table of boolean values, then the generation method may simply assign random values to the cells of this table.

- **Crossover_Method:**
  A crossover method generates a new individual (offspring) by combining the characteristics of two other individuals (parents). A typical implementation of a crossover method is the so-called "Uniform Crossover", where the offspring copies half the genome of one parent and half from the other, via random selection.
- **Mutation_Method:**
  A mutation method alters the state of an individual in a stochastic manner, by a relatively small degree.
- **Evaluation_Method:**
  An evaluation method is a method that measures a phenotypic or behavioral characteristic of an individual and returns a number that represents its quantity. Evaluation Methods can be used as fitness functions, thus driving the optimization process, or as Behavioral Characterizations, thus defining the dimensions of diversity between solutions. The purpose of the evaluation method can be marked by setting the "goal" variable to "NONE" for behavioral characterizations and to "MINIMIZATION" or "MAXIMIZATION" for fitness functions.
- **Constraint_Evaluation_Method:**
  A method that calculates whether an individual satisfies a specific constraint.

3.4 Benchmarks

The developed library contains a number of available benchmarks that showcase the algorithms' general functionality and applicability. These include a numeric optimization test-bed (3.4.2), two simple design-related problems (3.4.2, 3.4.3), as well a more complex design problem that is actually relevant to the design-assistance goals of PrismArch (3.4.4).

3.4.1 Constrained Numerical Optimization

Numerical optimization is very often used as a benchmark for evolutionary computation methods. Since constrained problems are of large importance in the context of the developed framework, we included a standard benchmark [16] for constrained numerical optimization that can be used to evaluate the baseline performance of our algorithms.

3.4.2 Evolvable Binary Tables

Two-dimensional data-tables are abstract data-structures with no inherent meaning. Their ease of definition, evaluation and manipulation, however, especially in the context of evolutionary computation, has led to them being used to represent a large number of problems, ranging from game-levels to topological optimization of structural elements (and many more). This benchmark focuses on the simplest case of such tables, whose values are binary (true / false). Although it is unlikely that this benchmark will be directly used to represent architectural solutions in the context of PrismArch, it is still useful to include it to the developed computational framework as a high-performance, baseline problem that also facilitates a visual form of evaluating the algorithms' operation.

Data_Structure:

- **DS__Binary_Table_2D:**
  This data structure represents a 2D table of binary values. The user needs to specify

the table's width and height during initialization. The data structure's internal data representation is based on a BitArray[2], in order to minimize the memory footprint of each individual and facilitate the existence of large numbers of individuals in memory.

Generation Methods:

- **GM__Random_Cell_Values:**
  A stochastic generation method that initializes a binary table by randomly assigning a true or false value to each cell. The probability of active cells can be set by the user to any value between 0.0 and 1.0, which will result in approximately 0% to 100% of the cells being active, respectively.

Crossover Methods:

- **CM__Uniform_Crossover:**
  A crossover method that iterates through the genome's length and selects a gene from one of the two parents, at random, with equal probability.

Mutation Methods:

- **GM__Flip_Random_Cells:**
  A stochastic mutation method that mutates a binary table by "flipping" (i.e. converting from true to false and vice versa) the values of a random set of cells. The method iterates over the individual's genes and selects them for mutation based on a user-defined mutation rate (i.e. mutation probability for each gene).

Constraint Evaluation Methods:

- **CEM__Active_Cells_Connected:**
  Requires that all active cells are connected.
- **CEM__Inactive_Cells_Connected:**
  Requires that all inactive cells are connected.
- **CEM__Minimum_Num_Active_Cells:**
  Requires a minimum number of active cells. The minimum number is manually set by the user.
- **CEM__Minimum_Num_Inactive_Cells:**
  Requires a minimum number of inactive cells. The minimum number is manually set by the user.
- **CEM__Maximum_Num_Active_Cells:**
  Requires a maximum number of active cells. The maximum number is manually set by the user.
- **CEM__Maximum_Num_Inactive_Cells:**
  Requires a maximum number of inactive cells. The maximum number is manually set by the user.

Evaluation Methods:

- **EM__Symmetry_LR:**
  The degree of horizontal (Left - Right) symmetry of a binary table.
- **EM__Symmetry_UD:**
  The degree of vertical (Up - Down) symmetry of a binary table.

[2] A BitArray is a data type provided by the c# language which represents an array of bits. In contrast to an array or list of boolean values, the BitArray optimizes memory usage so that a single binary value takes up only one bit of memory (instead of one byte, which is the case for boolean variables).

- **EM__Percent_Active_Cells:**
  The percentage (0...1) of active cells (i.e. cells whose value is "true") of a binary table.
- **EM__Percent_Inactive_Cells:**
  The percentage (0...1) of inactive cells of a binary table.

3.4.3 Evolvable Topology

This benchmark is an implementation of evolvable topologies. It focuses specifically on the example of unidirectional graphs and provides a broad set of operations and evaluation methods, allowing for the problem to be addressed by any of the provided algorithms. Despite the fact that plain, undirected graphs are quite abstract structures, they can easily be expanded with additional properties and be used as integral parts of a large number of design-related data-structures. The next benchmark is one example of this kind.

Data Structure:

- **DS__Undirected_Graph:**
  Undirected graphs can be perceived as a list of existing vertices and a list of "edges" that connect some of those vertices to each other. From a purely technical perspective, undirected graphs can be represented in many different ways. The choice of one over the other usually depends on what types of operations they will be subject to, as well as what types of properties will be necessary to calculate. In our implementation, we have chosen a data representation that facilitates a large number of operations, including the addition and removal of vertices and edges.

Generation Methods:

- **GM__Graph_Generator:**
  A stochastic generation method that generates an undirected graph. The user has to define the minimum and maximum number of nodes, as well as the probability of active edges. The method does not ensure that the generated graph will have any specific characteristics such as, for example, being connected or planar.

Mutation Methods:

- **MM__Add_New_Vertex__Connected:**
  Adds a new node to the graph and connects it to one of the existing vertices.
- **MM__Add_New_Vertex__Disconnected:**
  Adds a new, disconnected node to the graph.
- **MM__Add_New_Vertices__Connected:**
  Adds a number of new vertices to the graph, and ensures that they are directly or indirectly connected to some of the preexisting vertice. The minimum and maximum possible number of additional vertices are defined by the user.
- **MM__Add_New_Vertices__Disconnected:**
  Adds a number of new, disconnected vertices to the graph. The minimum and maximum possible number of additional vertices are defined by the user.
- **MM__Remove_Random_Edge:**
  Randomly selects one of the existing edges and removes it.
- **MM__Remove_Random_Edges:**
  Iterates over the list of available edges and removes them based on a user-specified probability.

- **MM__Remove_Random_Vertex:**
  Randomly selects a single existing node and removes it. The method also makes sure that any existing edges that contain this node are also removed.
- **MM__Remove_Random_Vertices:**
  Iterates over the list of existing vertices and removes them with a user-defined probability.

Constraint Evaluation Methods

Depending on the designer's preferences, a number of constraints can be imposed on the generated graphs. Our implementation includes the following ones that are based on a set of typical characterizations of graphs that are either true or false.

- **CEM__Is_Planar:**
  Requires that the graph is planar, i.e. it can be drawn on a plane without intersecting edges.
- **CEM__Is_Not_Planar:**
  Requires that the graph is not planar.
- **CEM__Is_Cyclic:**
  Requires that the graph is cyclic.
- **CEM__Is_Not_Cyclic:**
  Requires that the graph is not cyclic.
- **CEM__Is_Connected:**
  Is satisfied when the graph is connected, i.e. when all vertices are directly or indirectly connected to each other.
- **CEM__Is_Not_Connected:**
  Is satisfied when the graph is not connected.
- **CEM__Is_Fully_Connected:**
  Is satisfied when all vertices are directly connected to each other.
- **CEM__Is_Not_Fully_Connected:**
  Is satisfied when the graph is not fully connected.

Evaluation Methods

Our implementation contains a number of evaluation methods that can be used as either fitness functions or behavioral characterizations of an undirected graph.

- **EM__Order**:
  A graph's order is simply the number of vertices it includes.
- **EM__Percent_Isolated_Vertices**:
- Returns the percentage of vertices of the graph that are "isolated", i.e. they are not connected to any other vertex.
- **EM__Percent_Leaf_Vertices**:
  Returns the percentage of vertices of the graph that are connected to only a single other vertex.
- **EM__Degree__Average__Normalized**:
  Returns the normalized average degree of the graph. That is, the average number of connections per vertex. Normalization is applied by taking into account the maximum possible connections per vertex.
- **EM__Degree__Standard_Deviation__Normalized**:
  The standard deviation of the normalized degree per vertex. This metric represents how uniformly the edges are distributed in the graph.

3.4.4 Evolvable Geometry with Topological and Other Constraints

Despite the fact that graphs can capture a set of important properties of an architectural design, the description which they offer is by no means complete. As a matter of fact, there are an infinite number of geometric solutions that can be reduced to the same topological abstraction. This benchmark focuses on exactly this aspect of architectural design: the discovery of geometric solutions that satisfy a set of predefined topological (and other) constraints.

Data Structure:

The data structure of this benchmark consists of a number of discrete parts that work together. Namely, 1) The Problem Specification, 2) A tessellation of a 2D plane and 3) A utilization of the plane's cells. Both the tessellation of the 2D plane and the utilization of the cells are evolvable and considered as part of the parametric solution space, given a specific Problem Specification. The following subsections describe these parts in more detail:

Problem Specification:

The problem specification is a custom-designed data-structure that attempts to capture the necessary topological and other characteristics that could be a starting point for an architectural design. At its current implementation, the problem specification describes: 1) A number of space units, including their characteristic name and color, 2) The desired connections between those space units and 3) Their desired area (in square meters) per space unit. The desired connectivity between the space units is an especially important aspect of the problem specification, as it defines the topological constraints that the generated geometry should satisfy. The constraints that the problem specification defines are only a starting point and can be extended in the near future, in coordination with the AEC partners.

Spatial Tessellation

The spatial tessellation is a specific subdivision of a 2D plane into discrete cells. This subdivision generates a number of discrete "cells" which can be assigned to any of the prescribed space units, or left empty. The subdivision of the plane is itself evolvable and can be directly or indirectly evaluated. A direct evaluation can be based on the properties of the tessellation itself, while an indirect evaluation can be based on the potential utilizations that it enables.

Space Units' assignment

Given a specific Problem Specification and a Spatial Tessellation, the remaining information to define a specific design is to assign certain cells to certain prescribed space units. This approach (overall) treats the problem of designing 2D layouts in a way that is properly manageable by the available computational methods. The prescribed constraints, as well as any other user-defined constraints other characteristics can easily be evaluated.

Generation Methods:

- **GM__Random_Space_Unit_Assignment:**
  This method assigns random cells to random space units. The result has a very low probability of satisfying any of the assigned constraints.
- **GM__Select_Place_Expand_Repeat:**
  This method has a high probability in generating a solution that satisfies the

prescribed connectivity and size constraints, as they are prescribed in the problem specification. It operates by placing the prescribed rooms iteratively, while attempting to satisfy the connectivity constraints while doing so.

Mutation Methods:

Mutation methods for this benchmark have been organized in three categories. The first two of them, referred to as “Destruction Methods” and “Repair Methods” are designed in order to work in cooperation. In essence, the proposed approach is to use composite mutation methods that first “destroy” a given solution in a strategic manner and then repair it, thus generating a new solution with a high probability of being feasible, at the cost of some computational overhead. The third category, referred to as “Generic Mutation Methods” consists of purely stochastic mutation methods that do not specifically attempt to generate feasible solutions. The latter operate with the minimum possible computational overhead.

Destruction Methods:

Destruction methods are special mutation methods whose purpose is to strategically destroy the order of a specific individual, so as to promote the discovery of new solutions. They are not meant to be used on their own, but rather paired with some repair method.

- **MM__Delete_Random_Space_Units:**
  This method iterates over all the existing space units and selects some of them, based on an assigned probability. The selected space units are deleted, i.e. all the cells that are assigned to them are cleared.
- **MM__Increase_Cells_Of_Random_Space_Unit:**
  This mutation method selects an existing space unit, at random. Then it increases this space unit’s cells, by up to a maximum assigned number of cells. The space unit’s expansion is performed by using only adjacent, free cells. If there are no more adjacent free cells, the process may stop prematurely.
- **MM__Safe_Strip_Down_Random_Space_Unit**:
  This mutation method selects an existing space unit, at random. Then it removes from it as many cells as possible, without breaking this space unit’s own connectivity and without breaking its prescribed connectivity with other space units.
- **MM__Spatial_Tessellation__Global_Noise:**
  A mutation method that applies various degrees of noise to all parts of the spatial tessellation. The method iterates over the points that define the spatial tessellation and mutates their coordinates by a small amount, defined by the method’s mutation rate. This method is classified as a “destruction method” because there is a chance that it may disconnect some of the existing space units.
- **MM__Spatial_Tessellation__Local_Noise:**
  A mutation method that applies various degrees of noise to a subset of the spatial tessellation’s cells. The method iterates over the points that define the spatial tessellation and selects some of them, based on the method’s mutation chance. For the selected points, the method mutates their coordinates by a small amount, defined by the method’s mutation rate. This method is classified as a “destruction method” because there is a chance that it may disconnect some of the existing space units.

Repair Methods:

The following set of repair methods are available. Each one of them attempts to repair a specific aspect of the solution, bringing it closer to a feasible state. They can be applied in isolation or as a sequential operation which includes all of them.

- **MM__Repair_Space_Units_Existence:**
  A method that places all the missing, prescribed space units in the design, while attempting to satisfy as many connectivity constraints as possible.
- **MM__Repair_Connections__Shortest_Path:**
  This method attempts to repair the prescribed connections between space units by using the shortest possible "path" (i.e. sequence of cells) that connects them.
- **MM__Repair_Space_Unit_Area__Increase_Decrease_Threshold:**
  This method attempts to repair the area of all the existing space units, so that it is closer to their prescribed area, by either adding or removing cells.
- **MM__Repair_Space_Unit_Coherence__Keep_Largest_Fragment:**
  This method attempts to repair the coherence of any fragmented rooms by keeping their largest fragment and deleting the rest.

Generic Mutation Methods:

- **MM__Assign_Random_Space_Units_To_Random_Cells:**
  This method iterates over all the cells of the tessellated plane and selects some of them, at random, based on the assigned selection probability. The selected cells are associated with one of the prescribed space units, at random.

Constraint Evaluation Methods:

- **CEM__Prescribed_Connections__Exist:**
  A constraint evaluation method that is satisfied only when all the prescribed connections exist. More precisely, when both of the following are true: 1) All the prescribed space units exist, and 2) The existing rooms are adjacent to the ones that they are supposed to.
- **CEM__Prescribed_Space_Units__Are_Coherent:**
  A constraint evaluation method that is satisfied when both of the following are true: 1) All the prescribed rooms exist, and 2) All the prescribed rooms are coherent, i.e. they exist "in one piece" in the solution.
- **CEM__Prescribed_Space_Units__Areas_Are_Correct:**
  A constraint evaluation method that is satisfied when both of the following are true: 1) All the prescribed rooms exist, and 2) The area of each room is within a specific error margin.
- **CEM__Prescribed_Space_Units__Exist:**
  A constraint evaluation method that is satisfied when all the prescribed space units exist in the solution.

Evaluation Methods:

The implemented evaluation methods for this benchmark have been organized in two discrete categories. The first one includes constraint-related evaluation methods. Those methods attempt to capture the "gradient" of satisfying the available constraints, so as to facilitate the discovery of feasible geometries. The second category includes a set of abstract, geometrical evaluation methods that can be used to differentiate feasible or infeasible solutions in regard to their geometric complexity and other relevant characteristics.

Constraints-Related Evaluation Methods:

- **EM__Prescribed_Space_Units__Existence_Percentage:**
  An evaluation method that measures the percentage of: the number of existing space units over the number of prescribed space units.
- **EM__Existing_Space_Units__Coherence_Percentage:**
  An evaluation method that measures the degree to which the existing space units are coherent.
- **EM__Prescribed_Space_Units__Coherence_Percentage:**
  An evaluation method that measures the degree to which the prescribed space units are existing and coherent.
- **EM__Prescribed_Space_Units__Coherence_Percentage__Fine_Grained:**
  An evaluation method that measures the degree to which the prescribed space units are existing and coherent, but also takes into account the degree to which a space unit may be non-coherent.
- **EM__Prescribed_Connections__Existence_Percentage:**
  An evaluation method that measures the percentage of: the number of prescribed connections that exist over the total number of prescribed connections.
- **EM__Existing_Space_Units__Area_Score:**
  An evaluation method that calculates a granular score that represents the proximity of the area of the existing space units to their prescribed areas.
- **EM__Prescribed_Space_Units__Area_Score:**
  An evaluation method that calculates a granular score that represents the proximity of the area of the existing space units to their prescribed areas. It also takes into account the prescribed, but non-existing space units.

Geometric Evaluation Methods:

The following list of evaluation methods capture various abstract, geometrical aspects that can be used for the differentiation between feasible or infeasible solutions. This

- **EM__Existing_Space_Units__Perimeter_Minimization_Score:**
  Calculates a score that is based on the perimeter of existing rooms. Higher values of this score indicate that the perimeter is closer to its minimum possible value, given the space units' current area.
- **EM__Prescribed_Space_Units__Perimeter_Minimization_Score:**
  Calculates a score that is based on the perimeter of existing rooms. Higher values of this score indicate that the perimeter is closer to its minimum possible value, given the space units' current area. It also takes into account any missing, prescribed space units, penalizing the score accordingly.
- **EM__Total_Perimeter_Minimization_Score:**
  Calculates a score that is based on the perimeter of the whole solution. Higher values of this score indicate that the perimeter is closer to its minimum possible value, given the solution's total area.
- **EM__Orthogonal_Lines__Maximization_Score:**
  Calculates how close the generated space units are to being "orthogonal", i.e. either horizontal or vertical.
- **EM__Orthogonal_Lines__Maximization_Score__Weighted:**
  Calculates how close the generated space units are to being "orthogonal", i.e. either

horizontal or vertical. The score is calculated as a weighted average, giving higher weights to longer lines.

- **EM__Square_Angles_Between_Consequent_Lines__Maximization_Score:**
  Calculates how close the angles of all consequent lines are to being square angles (90 degrees).
- **EM__Straight_Or_Square_Angles_Between_Consequent_Lines__Maximization_Score:**
  Calculates how close the angles of all consequent lines are to being square (90 degrees) or straight (180 degrees) angles.

3.5. Extensibility

This section presents the ways in which the available algorithmic functionality can be extended. This includes the application of the same algorithms to new types of problems, as well as the extension of specific algorithmic modules in the context of the already considered types of problems.

During the development of this computational framework, a fair amount of planning has been devoted to facilitating its extensibility. The main method that was followed in achieving this goal was to disassociate the algorithmic operation from the specifics of each benchmark. In order to do so, the algorithms' operation is based on a set of abstract classes (as described in section 3.3). These abstract classes must be concretely implemented in the context of a specific benchmark. This technique allows for the application of the same algorithm to many different problems, as well as the testing of algorithmic variations on the same problem.

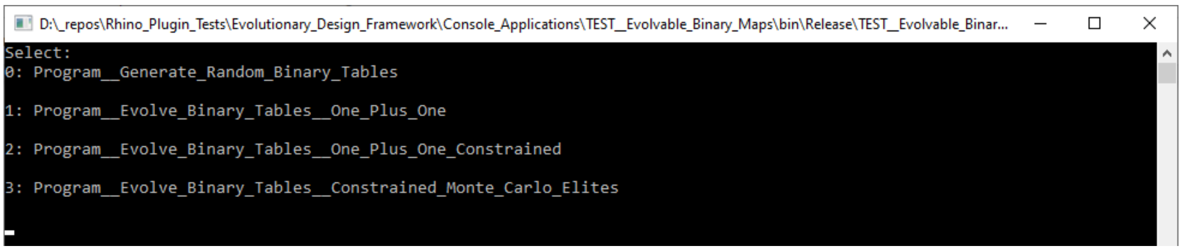
Given the ground-work that has been laid out, it is very easy to extend any aspect of the project, including 1) Appling new algorithms (or algorithmic variations) on the existing sets of problems, 2) Defining new problems to be addressed through the same algorithms and 3) Extending the algorithmic parameters (such as the generation, mutation and evaluation methods). The best guide in how any of these aspects can be achieved is the project itself, which includes four different algorithms, four benchmarks and a large number of interchangeable methods that can work together in any conceivable combination.

3.6. Executable Demos

In order to present the computational framework's functionality, a number of executable demos have been developed. These simple programs showcase how the modular architecture of the developed framework comes together in addressing specific problems. Their source-code can be found in the "Executable_Demos" sub-folder of the framework.

3.6.1 Evolvable Binary Tables

This demo showcases the application of the available algorithms and methods on the problem of evolving 2D Binary Tables. The relevant source-code can be found in the "DEMO__Evolvable_Binary_Tables" Visual Studio project. As soon as the code is executed, a console application will appear (as shown in the screenshot below), presenting the user with four options. The user can select any of them, by typing its corresponding number (0 - 3).



```
D:\_repos\Rhino_Plugin_Tests\Evolutionary_Design_Framework\Console_Applications\TEST__Evolvable_Binary_Maps\bin\Release\TEST__Evolvable_Binar...
Select:
0: Program__Generate_Random_Binary_Tables

1: Program__Evolve_Binary_Tables__One_Plus_One

2: Program__Evolve_Binary_Tables__One_Plus_One_Constrained

3: Program__Evolve_Binary_Tables__Constrained_Monte_Carlo_Elites
```

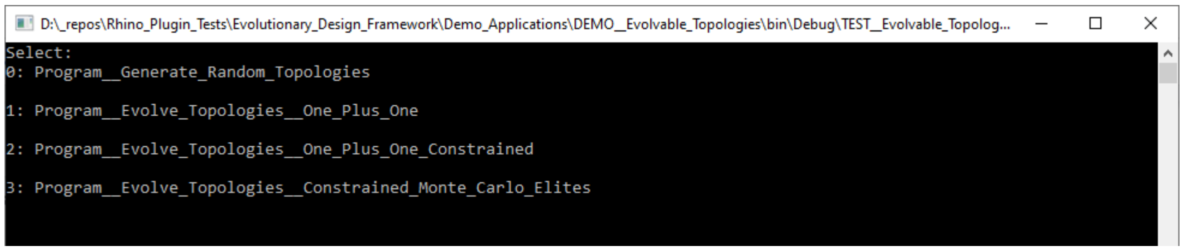
Figure 1. Screenshot of the console-menu-options for the "Evolvable Binary Tables" Demo.

The following list explains the purpose and functionality of each option:

1) **"Program__Generate_Random_Binary_Tables"**
   This program showcases how a predefined stochastic generation method can be used to randomly generate a set of binary tables.
   The generated artifacts are automatically saved in a uniquely named folder, in the same directory as the executable, in the form of images and serialized data files.
2) **"Program__Evolve_Binary_Tables__One_Plus_One"**
   This program showcases how the simple evolutionary strategy (1+1) can be used to evolve binary tables towards the optimization of a single objective.
   The generated artifacts are automatically saved in a uniquely named folder, in the same directory as the executable, in the form of images and serialized data files.
   A fitness report is also stored in the form of a csv file.
3) **Program__Evolve_Binary_Tables__One_Plus_One_Constrained**
   This program showcases how a simple evolutionary strategy (1+1)-constrained can be used to evolve binary tables towards the optimization of a single objective, while also fulfilling the selected constraint(s).
   The generated artifacts are automatically saved in a uniquely named folder, in the same directory as the executable, in the form of images and serialized data files.
   A fitness report is also stored in the form of a csv file.
4) **Program__Evolve_Binary_Tables__Constrained_Monte_Carlo_Elites**
   This program showcases how the Constrained Monte Carlo Elites algorithm can be used to evolve a set of binary tables while illuminating the design space of a set of behavioral dimensions, optimizing a single objective and fulfilling the selected constraint(s).
   The algorithm's state is automatically saved in a uniquely named folder, in the same directory as the executable, in the form of images and serialized data files.

3.6.2. Evolvable Topologies

This demo showcases the application of the available algorithms and methods on the problem of evolving Undirected Graphs. The relevant source-code can be found in the "DEMO__Evolvable_Topologies" Visual Studio project. As soon as the code is executed, a console application will appear (as shown in the screenshot below), presenting the user with four options. The user can select any of them, by typing its corresponding number (0 - 3).



```
D:\_repos\Rhino_Plugin_Tests\Evolutionary_Design_Framework\Demo_Applications\DEMO__Evolvable_Topologies\bin\Debug\TEST__Evolvable_Topolog...
Select:
0: Program__Generate_Random_Topologies

1: Program__Evolve_Topologies__One_Plus_One

2: Program__Evolve_Topologies__One_Plus_One_Constrained

3: Program__Evolve_Topologies__Constrained_Monte_Carlo_Elites
```

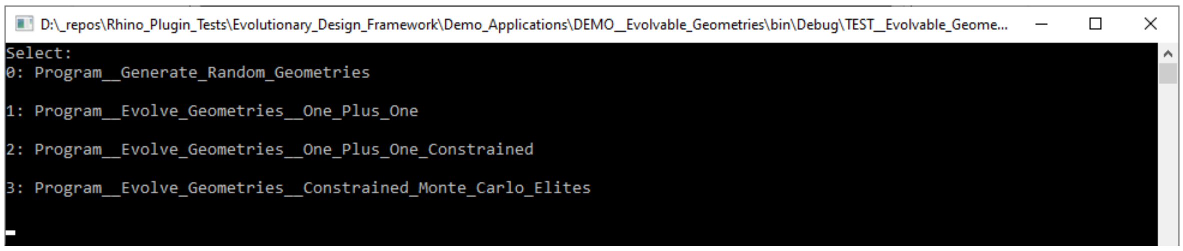
Figure 2. Screenshot of the console-menu-options for the “Evolvable Topologies” Demo.

The following list explains the purpose and functionality of each option:

1) **Program__Generate_Random_Topologies:**
   This program showcases how a predefined stochastic generation method can be used to randomly generate a set of undirected graphs.
2) **Program__Evolve_Topologies__One_Plus_One:**
   This program showcases how the simple evolutionary strategy (1+1) can be used to evolve undirected graphs towards the optimization of a single objective.
3) **Program__Evolve_Topologies__One_Plus_One_Constrained:**
   This program showcases how the simple evolutionary strategy (1+1)-constrained can be used to evolve undirected graphs towards the optimization of a single objective, while also fulfilling the selected constraint(s).
4) **Program__Evolve_Topologies__Constrained_Monte_Carlo_Elites:**
   This program showcases how the Constrained Monte Carlo Elites algorithm can be used to evolve a set of undirected graphs while illuminating the design space of a set of behavioral dimensions, optimizing a single objective and fulfilling the selected constraint(s).

3.6.3. Evolvable Geometries With Topological And Other Constraints

This demo showcases the application of the available algorithms and methods on the problem of evolving geometries with topological and other constraints. The relevant source-code can be found in the “DEMO__Evolvable_Geometries” Visual Studio project. As soon as the code is executed, a console application will appear (as shown in the screenshot below), presenting the user with four options. The user can select any of them, by typing its corresponding number (0 - 3).



```
D:\_repos\Rhino_Plugin_Tests\Evolutionary_Design_Framework\Demo_Applications\DEMO__Evolvable_Geometries\bin\Debug\TEST__Evolvable_Geome...
Select:
0: Program__Generate_Random_Geometries

1: Program__Evolve_Geometries__One_Plus_One

2: Program__Evolve_Geometries__One_Plus_One_Constrained

3: Program__Evolve_Geometries__Constrained_Monte_Carlo_Elites
```

Figure 3. Screenshot of the console-menu-options for the “Evolvable Geometries” Demo.

The following list explains the purpose and functionality of each option:

1) **Program__Generate_Random_Geometries:**
   This program showcases how a predefined stochastic generation method can be used to randomly generate a set of geometric solutions.

2) **Program__Evolve_Geometries__One_Plus_One:**
   This program showcases how the simple evolutionary strategy (1+1) can be used to evolve geometric solutions towards the optimization of a single objective.
3) **Program__Evolve_Geometries__One_Plus_One_Constrained:**
   This program showcases how the simple evolutionary strategy (1+1)-constrained can be used to evolve geometric solutions towards the optimization of a single objective, while also fulfilling the selected constraint(s).
4) **Program__Evolve_Geometries__Constrained_Monte_Carlo_Elites:**
   This program showcases how the Constrained Monte Carlo Elites algorithm can be used to evolve a set of geometric solutions while illuminating the design space of a set of behavioral dimensions, optimizing a single objective and fulfilling the selected constraint(s).

4. CONCLUSIONS AND PLANNED EXTENSIONS

As this document already clarified, the development of the Quality-Diversity-driven computational framework is at a mature stage. It includes a large number of algorithmic variations and benchmarks and is based on a modular and extensible architecture, all of which will be beneficial in the up-coming integration and further development steps.

The next big step is the integration of the developed framework into PrismArch's platform, which will form a VR-based environment for interactive design. The following section (4.1) describes a detailed example of frontend-backend integration, following the Indirect mode of interaction (section 2.2), while section 4.2 describes how this integration will lead towards the step of Designer Modeling.

4.1. Example of use of the algorithms once integrated into the PrismArch front-end

This section is an example of integration between the Quality-Diversity driven computational framework and the PrismArch platform's graph-based UI. This example takes into account the working prototype of a Graph-Based UI which has already been implemented by CERTH. This prototype is completely aligned with the necessary high-level features that are necessary to support the front-end of an Indirect interaction (see sec. 2.2), including the dynamic visualization of a graph and its meta-information in 3D space, and the user's interaction with specific nodes of this graph. More details about this UI will be reported in D5.2.

4.1.1. Functional Description

The functional description of the integration example is split into two parts: the preparation phase, where the designer must select a set of parameters, and the interaction phase, where the designer is involved in an interactive process of exploration of the parametric design space. The following two subsections describe these phases in detail.

Preparation

The following list describes the preparation steps in detail, in reference to Figure 4 which visualizes the steps in the form of a diagram.

1) The designer selects a number of Behavioral Dimensions (measurable Features) and a fitness function. The selected features will be used to establish a mode of diversifying solutions, while the fitness function will define an underlying objective for the QD algorithm.
2) The user's selected settings are received at the backend and an initialization of the QD algorithm takes place. The behavioral space is subdivided into discrete cells, awaiting the insertion of diverse designs.
3) The designer provides an initial design, which will be used as the starting point (seed) in the QD algorithm's search process.
4) The initial solution is evaluated across the selected behavioral dimensions and placed in its corresponding location in the archive.
5) The actual operation of the QD algorithm takes place. The algorithm generates variations of the initial design and gradually builds a database of diverse and fit design solutions that span across the behavioral space.

6) The initial design is presented to the user as the root-node of their decision-tree, in the graph-based UI, and the process of interaction can begin.

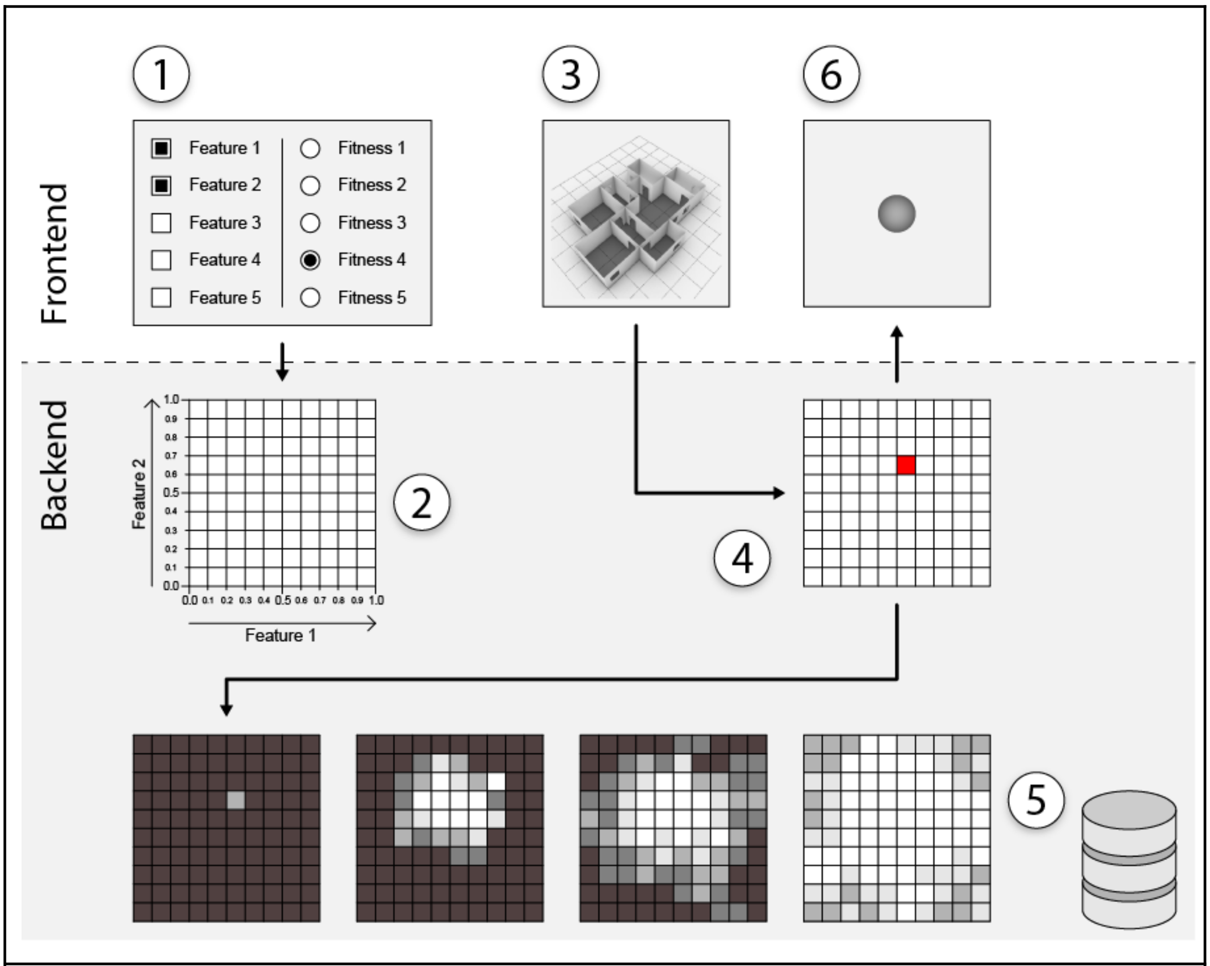


Figure 4: Preparatory steps of the interactive design process.

Interaction

As soon as the preparatory process has taken place, the interaction between the designer and the Quality-Diversity driven computational framework can begin. The process of interaction is visualized in Figure 5, in the form of a diagram, while the following list describes the steps in detail.

1) The system finds in the database a set of solutions that are relatively similar to the initial design, but different from each other and sends this set of solutions to the front-end.
2) The front-end visualizes the new options and presents them as offspring of the root node, so that the designer can make their initial selection out of the variations.
3) The designer selects one of the available designs, based on their own subjective criteria and the measurable information that the system may also expose. The designer's selection is sent to the backend in the form of a selection event.
4) The system treats the designer's selection as a reference node and, similar to (1), finds a new set of solutions in the database that are similar to the reference design but different from each other. It then sends those new designs to the front-end.

5) Similar to (2), the front-end visualizes the new design options and presents them as offspring of the previously selected node, so that the designer can make a new selection.

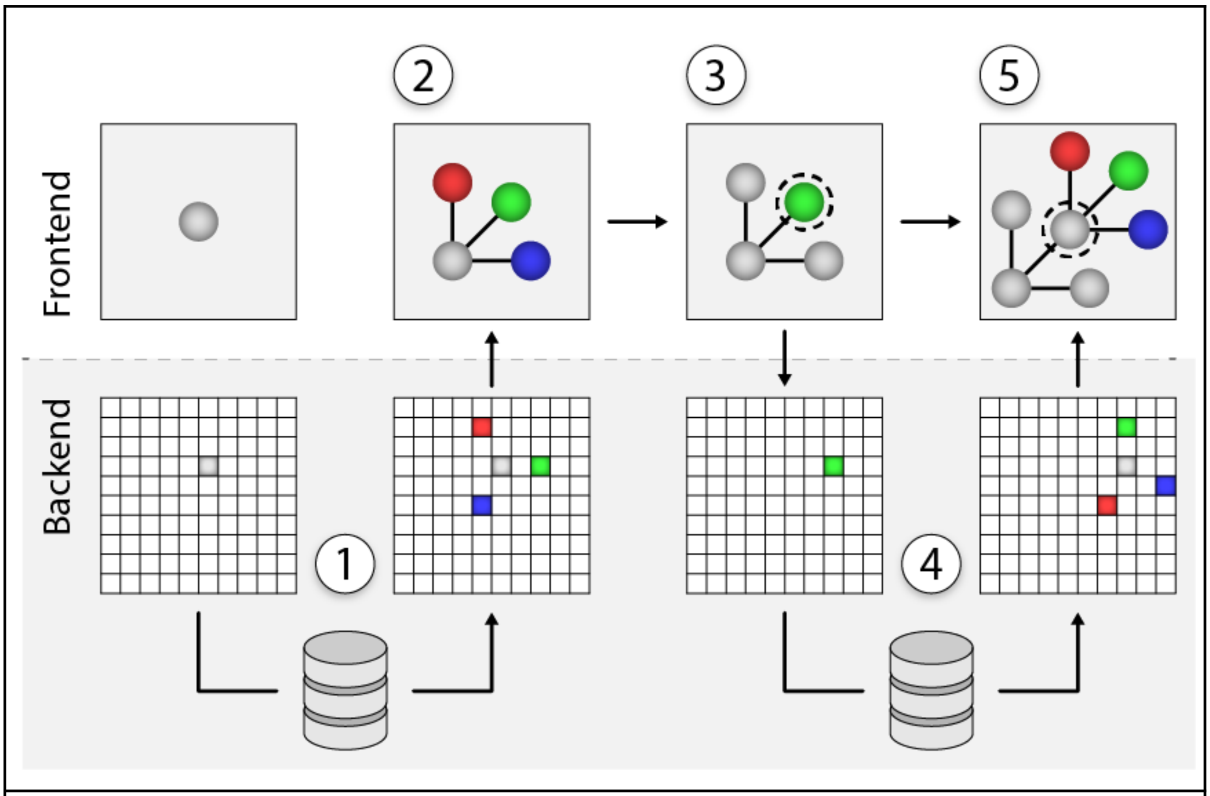


Figure 5. Visualization of the interactive process, between the designer and the Quality-Diversity-driven design-assistance tool, through the graph-based UI.

4.1.2. Technical information

The algorithmic framework of this deliverable will eventually be fully embedded in the PrismArch platform which is being developed in the Unreal Engine. Despite the fact that development in Unreal is typically based on C++, the cross-language integration will be supported by a very useful plugin called “UnrealCLR”[3]. In order for the integration to be implemented, an intermediate layer of interconnection between the front-end (the graph-based UI) and the backend (the QD-based algorithmic framework) will be developed by the UoM. One of the main goals of this intermediate layer is that it be incorporated into Unreal’s Blueprints Visual Scripting system, so as to facilitate the further UI development as much as possible. This blueprints-node will be able to receive simple events (i.e. commands in the form of strings) that describe the Designer’s selections and return new data (i.e. new designs to be included in the graph-based UI). In order for this type of integration to work, the abstract representation of the designer’s decision-tree will be mainly managed in the back-end, including all of the additions and deletions of nodes. Updating the graph’s visualization at the front-end can then be based on a shared data-structure that will be

[3] The source code for the UnrealCLR plugin can be found at https://github.com/nxrighthere/UnrealCLR. The plugin is very actively developed, being at a mature stage, and has furthermore been officially recognized and supported by Epic Games (the company that develops the Unreal Engine.

developed in close collaboration with CERTH, in order to support the existing UI as best as possible.

4.2. Towards Designer Modeling

As soon as the interactive design environment is in place, two critical aspects of future development can take place: The first one is to gather feedback from the end-users (in the context of D6.3) and take it into account in forming a revised version of the parametric design space and the QD algorithms' functionality. The second aspect is that the interactive design environment can be used to gather data (traces) of interactions between the designers and the algorithms and use them to train Designer Models that can then be incorporated into the framework itself, enhancing the algorithmic operation. Both aspects, as well as their results, will be reported in D2.3 [20]. Moreover, all AI components will be deployed and tested as part of the PrismArch solution and findings reported in the evaluation efforts of WP6 (mainly in D6.3).

5. REFERENCES

[1] Lehman, Joel, and Kenneth O. Stanley. "Abandoning objectives: Evolution through the search for novelty alone." Evolutionary computation 19.2 (2011): 189-223.

[2] Lehman, Joel, and Kenneth O. Stanley. "Evolving a diversity of virtual creatures through novelty search and local competition." Proceedings of the 13th annual conference on Genetic and evolutionary computation. 2011.

[3] Mouret, Jean-Baptiste. "Novelty-based multiobjectivization." New horizons in evolutionary robotics. Springer, Berlin, Heidelberg, 2011. 139-154.

[4] Mouret, J. B., & Clune, J. (2015). Illuminating search spaces by mapping elites. arXiv preprint arXiv:1504.04909.

[5] Alvarez, Alberto, et al. "Empowering quality diversity in dungeon design with interactive constrained map-elites." 2019 IEEE Conference on Games (CoG). IEEE, 2019.

[6] Alvarez, Alberto, et al. "Interactive constrained map-elites: Analysis and evaluation of the expressiveness of the feature dimensions." IEEE Transactions on Games (2020).

[7] Sfikas, Konstantinos, Antonios Liapis, and Georgios N. Yannakakis. "Monte Carlo elites: quality-diversity selection as a multi-armed bandit problem." arXiv preprint arXiv:2104.08781 (2021).

[8] Kimbrough, Steven Orla, et al. "Introducing a feasible-infeasible two-population (fi-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch." European Journal of Operational Research 190 (2005).

[9] Migkotzidis, Panagiotis, and Antonios Liapis. "SuSketch: Surrogate Models of Gameplay as a Design Assistant." IEEE Transactions on Games (2021).

[10] Liapis, Antonios, Georgios N. Yannakakis, and Julian Togelius. "Designer modeling for sentient sketchbook." 2014 IEEE Conference on Computational Intelligence and Games. IEEE, 2014.

[11] Liapis, Antonios, Georgios N. Yannakakis, and Julian Togelius. "Sentient sketchbook: computer-assisted game level authoring." (2013).

[12] Liapis, Antonios, Georgios N. Yannakakis, and Julian Togelius. "Designer modeling for sentient sketchbook." 2014 IEEE Conference on Computational Intelligence and Games. IEEE, 2014.

[13] Smith, Gillian, Jim Whitehead, and Michael Mateas. "Tanagra: A mixed-initiative level design tool." Proceedings of the Fifth International Conference on the Foundations of Digital Games. 2010.

[14] Fioravanzo, Stefano, and Giovanni Iacca. "Evaluating map-elites on constrained optimization problems." arXiv preprint arXiv:1902.00703 (2019).

[15] Khalifa, Ahmed, et al. "Talakat: Bullet hell generation through constrained map-elites." Proceedings of The Genetic and Evolutionary Computation Conference. 2018.

[16] Liang, J. J., et al. "Problem definitions and evaluation criteria for the CEC 2006 special session on constrained real-parameter optimization." Journal of Applied Mechanics 41.8 (2006): 8-31.

[17] Martin Brösamle, Panos Mavros, Christoph Hölscher, Spiros Nikolopoulos, & Risa Tadauchi. (2021). D3.1 Report on cognitive issues in VR-aided design environments. Zenodo. https://doi.org/10.5281/zenodo.5095109

[18] Antonios Liapis, Konstantinos Sfikas, Theodore Galanos, Georgios N. Yannakakis, Helmut Kinzler, Daria Zolotareva, Risa Tadauchi, Aleksandra Mnich-Spraiter, Jeg Dudley, Edoardo Tibuzzi, Arun Selvaraj, Dinos Ipiotis, & Oussama Yousfi. (2021). D2.1 Initial version of parametric design space. Zenodo. https://doi.org/10.5281/zenodo.5095066

[19] Daria Zolotareva, Risa Tadauchi, Helmut Kinzler, Arun Selvaraj, Oussama Yousfi, Edoardo Tibuzzi, Jeg Dudley, & Dimitrios Ververidis. (2021). D1.1 Report on current limitations of AEC software tools, leading to user and functional requirements of PrismArch. Zenodo. https://doi.org/10.5281/zenodo.5095028

[20] List of PrismArch deliverables: https://prismarch-h2020.eu/deliverables/