Conference Paper

Lightweight Semiautomatic City Engine Systems: A Specific Case of Urban Induction Theory

Robert Ward

Xian Jiaotong University, Department of Architecture, Suzhou, China

Abstract

Generative urban design is an emerging field which seeks to apply a procedural algorithm to grow urban forms parametrically. These tools are mainly used for cityscapes in the entertainment industry. Currently, there is active research to repurpose them as semantically meaningful tools for urban design. This paper reviews the two primary approaches to generative urban design: *CityEngine* and Urban Induction (UI). After consideration UI was chosen as the preferred approach. This paper is a commentary on and assumes a background knowledge of UI. A novel synthesis of UI and Form-Based Codes is proposed as a lightweight method of generation. This method is then tested on a 3,040 acre site in San Francisco. Results indicate while the method works conceptually in simple cases it should be expanded into a comprehensive toolset before it can be considered an open-source replacement for *CityEngine*.

Keywords: Urban Design, Parametric Design, GIS, Grasshopper, Urban Induction, City Engine.

Corresponding Author:
Robert Ward
robert.ward17@student.xjtlu.edu.cn

Received: 15 March 2019
Accepted: 25 May 2019
Published: 20 November 2019

Publishing services provided by Knowledge E

© Robert Ward. This article is distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use and redistribution provided that the original author and source are credited.

Selection and Peer-review under the responsibility of the Architecture across Boundaries Conference Committee.

1. Introduction

Generative Design is a computational design process (CDP) in which the architect instructs a computer to produce design by a set of rules and 'kit of parts', the generative grammar, as described by Alexander in his landmark paper [17]. Generative urban design is a CDP used to generate a city. Generative urban design is especially helpful in translating numerical data such as height, density, and Floor Area Ratio (FAR) into 3D building geometry. This translation of numerical input into geometry is a useful way to test design proposals, communicate between design disciplines, visualize for clients, and generate inputs for environmental analyses. At the core of any generative urban design tool, is a procedural algorithm which grows urban forms parametrically from rules. Typically these rules are defined in L-systems, Shape Grammars, or Sortal Grammars. Thus far these techniques have been used to make computer generated imagery (CGI) of cityscapes for films and video games. The result superficially looks like a city but

lacks the rigorous ontology of a real City Information Model (CIM). Current research is directed toward generating formal outputs which are compatible with CIM ontologies. This research addresses the core nature of design problems as open-ended problems which cannot be 'solved' by computation and logic [13], but rather advanced through iterative, recursive design moves of generation and evaluation of a design proposal. Algorithms can automate these design moves within a limited scope, but there is a point of diminishing returns where automation requires very complex algorithms for marginally useful results. Generally, the more computation is used the less room for creativity in the design. Considering the diminishing returns principle, the goal of generative urban design is the optimum efficiency of automation and freedom of creativity.

1.1. Literature Review

A review of the literature shows two primary techniques generative urban design. The first technique by Yonish & Muller uses parametric L-systems as explained in their seminal paper [1] at SIGGRAPH 2001. Their research led directly to the creation of *CityEngine* software. The second technique is Urban Induction (UI) theory as developed by Beirão & Duarte [2] in 2012. In addition to these two comprehensive theories are papers on specific topics. The central theme across reviewed specific topic papers was a recreation of partial GIS functionality inside *Grasshopper*, with complex and necessarily inefficient scripts. The option of simply using GIS itself was not considered in the critical reflection of the reviewed papers. Among the specific topic papers, Kim et al. [3] stands out for a novel proposal to create a computational implementation of New Urbanist Form-Based Code. Form-Based Codes are model planning codes in which zoning ordinances specifiy a form, not a function. As such, they parametrically describe a simple relationship between architecture and streetscape.

A review of available tools shows that much of the functionality of *City Engine* (2007) can be duplicated in *Grasshopper* (2014) as both have a visual programming interface (VPI) to build algorithms. Additionally, *CityEngine* lacks the technical realism necessary in urban design and is mostly used for geodesign [15, 16] and CGI applications [4]. For example, *CityEngine* is typically used to generate an entire city in greenfield conditions based on a raster underlay of population density. In practice, this scenario rarely occurs. It is more typical to develop urban infill in a portion of a city which is driven by context and connectivity. In professional practice, *CityEngine* has a smaller user base than *Grasshopper* because the barriers to entry are higher in terms of both money and time. *CityEngine* is a niche software which only works within the ecosystem of *arcMap* GIS

and uses a proprietary scripting language called 'CGA Shape Grammar'. Conversely, *Grasshopper* is a free VPI for *Rhinocerous3D* and uses the popular Python and C# scripting languages.

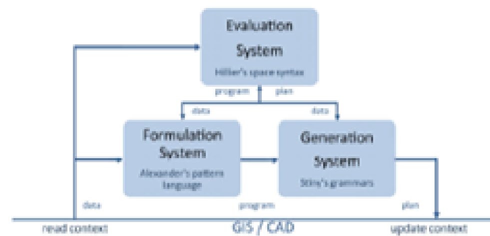

Figure 1: Urban Induction.

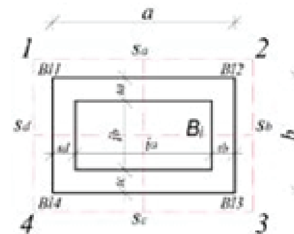

Figure 2: UIP_{30} *AddingUUnits*.

In contrast to *CityEngine* generating a cityscape with L-systems, UI theory aims to create semantically meaningful results that are well-matched to the rigorous city description ontologies developed by Lynch and Alexander [10, 11]. The theoretical, mathematical, and technical underpinnings of UI are described exhaustively by Beirão in [5]. UI mirrors the design moves made by urban design professionals as they move a design proposal from vague to specific. UI proceeds, through recurrent generation-analysis-generation loops (induction) of Urban Induction Patterns (UIP) as shown in Figure 1.

In UI theory, a UIP is a stepwise design move in the generation module consisting of a parametric geometry with attached tabular data, as shown in Figure 2. Each UIP depends on the preceding UIP for input and conversely becomes the input for the succeeding UIP. As UIPs layer on top of each other, the complexity of a city emerges generatively. An analysis of the UI generation module shows UIPs can be classified in two groups: UIPs upstream of UIP_{30} used for generation of circulation networks, which forms the urban grid dividing space, and UIPs downstream of UIP_{30} used for filling each grid cell (blocks) with forms (buildings). The moment of filling blocks occurs at UIP_{30}.

Beirão proposes a specific implementation of UIP generation, *CIty Maker,* and concludes that *CIty Maker* is best split across GIS, CAD, and VPI [5, 7]. This cross-platform

technique, called 'dis-integrative,' is well attested as a superior approach to computational design [6, 8]. The advantages to a dis-integrative over an integrated work environment are less scripting and more artistic control, while the disadvantage is more careful data management.

When a dis-integrative approach is critically applied to *CIty Maker* a classification by critical path emerges between UIPs upstream and downstream of UIP_{30}. The layout of a grid (... - UIP_{30}) is 'cheap' when done manually because it is simple geometry to draft with existing tools, but difficult to generate computationally. Conversely filling the grid (UIP_{30} - ...) is 'expensive' because it is complex geometry that has no existing tool, but it is a simple algorithm to script. Within the critical path of generation, UIP_{30} *AddingUUnits* occupies a unique position that can only be implemented efficiently by an algorithm by specific to the project, as discussed in section 3.1 of this paper.

1.2. Purpose

Based on the literature review, I propose a lightweight semi-automatic implementation of UIP generation. The purpose is to improve the usefulness and flexibility of *CIty Maker*. Here 'semi-automatic' means at certain design moves a UIP is more efficiently generated by computer and other design moves by a user with project-specific innovations. It is 'lightweight' in the sense it only addresses a limited scope of only those UIPs which lie on the critical path, see table 1. Thus, three levels of automation in UIP generation become available at each design move: total automation by computer as attempted in *CIty Maker* [1, 2], partial automation by computer, and manually by user constrained in a UIP framework. Ideally, all higher level design is done by the user and all lower level drafting is done by machine.

2. Methodology

Following the recommendations of [5–7], the lightweight semi-automatic implementation is dis-integrative requiring manual data transfer between CAD, GIS, and VPI at specific design moves. The softwares used are *Civil3D, Grasshopper, Infraworks*, and optionally *arcMap*. All UIPs upstream of UIP_{30} can be generated entirely with *Civil3D* because it is a CAD & GIS integrated environment. At the design move of UIP_{30} data in *Civil3D* must be round-tripped through the *Grasshopper* VPI for UIP generation. *Infraworks* is a CIM and visualization environment, which duplicates most of the functionality in *City Engine* but is tightly integrated with *Civil3D*. In particular,

Infraworks transforms a circulation network centerlines from *Civil3D* into 3D textured meshes through typical section sweeps and intersection rules. Additionally, it has a limited ability to apply procedural raster-based facades to building masses as in *City Engine*.

Figure 3: Location in San Francisco.

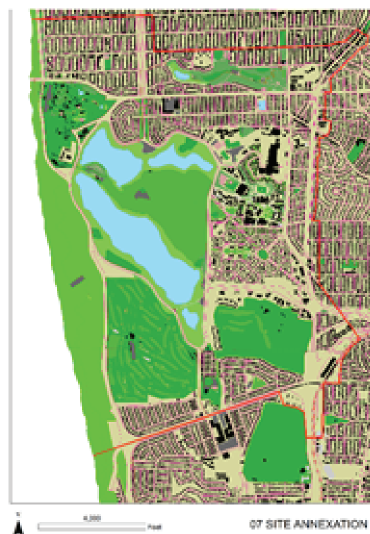

Figure 4: Site inside redline.

These tools were tested on a sparely built 3,040 acre (2,130ha) area of San Francisco. This site, shown in figures 3-4, is planned to be a LEED-ND development of 150,000 units to ease the severe housing shortage [14]. It is representative of infill and densification tasks typical in urban design.

Table 1: Urban Induction Patterns.

UIP	Name	Description
24	AddBlocktoCells	Defines a grid cell as an island
25	AdjustingBlockCells	Fixes overlapping
26	ClassifyUUnitCells	Creates sets of labels for block classification
27	DefineUUnit	Defines blocks by type
28	InititalUUnit	Creates initial labels to classify block types
29	AddUUnitByLabel	Replaces an island with a block type
30	AddingUUnits	Adds typical buildings to blocks by type

2.1. Setting out of Grids

Prior to UIP_{24} centerlines are generated by a variety of methods in CAD or GIS. At the design move of UIP_{24} centerlines are converted to UIP gridlines and grid cells. Here the blocks are classified using the New Urbanist Form-Based Code transect schema. The transect describes typical values for the height, setback, and inter-building spacing, parameters of a Form-Based Code. The transect is organized as seven zones of increasing urbanization from Natural Zone (T-1) to Urban Core Zone (T-6), see Figure 5 and 6 [9]. Classifying blocks by transect zone combines several UIP parameters controlling height, density, and typology into a single parameter, called 'Transect Zone,' that can be stored in the LAYER property of the block polygons. Using the LAYER property to encode 'Transect Zone' an efficient way to have multiple UIP parameters displayed both visually and textually in CAD.

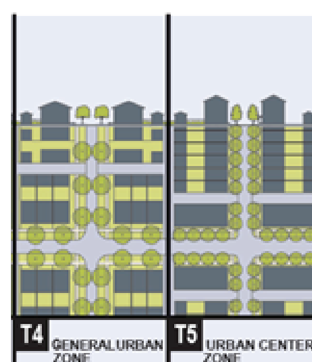

Figure 5: Transect Zones.

Option 1 *Civil3D*: Sites, Parcel, and Alignment Objects to automatically generate the grid lines, cells, and blocks geometry, see table 2. This geometry is exported as an SDF file then re-imported back into CAD as line and polygon geometry with attached tables.

Figure 6: Form-Based Code.

These tables and shapes can be modified as needed and then exported as an SQLite or DXF.

TABLE 2: Civil3D Grid Generation Commands.

Command	Description
CreateSite	Creates a named topology
CreateParcelFromObjects	Creates a Polygon Feature in the topology
CreateAlignmentEntities	Creates a Linear Feature in the topology
CreateParcelROW	Creates a Buffer of a Linear feature
ExportToSDF	Exports CAD topology to GIS topology
MapImport	Imports GIS data to CAD geometry with attached tables
ADEDefData	Opens attached table manager (add, edit, delete columns)
ADEAttachData	Attaches attribute tables to CAD geometry
MapExport	Exports CAD geometry as GIS geometry

Option 2 *Civil3D* + *GIS*: Centerlines are exported as a DXF to GIS where scripts will process them into geodatabase lines and polygon features then classify them based on the COLOR and WIDTH properties, and build an attribute tables populated with typical values by class. By using the *Model Builder* VPI in *arcMap* prototype scripts can be rapidly adjusted to specific projects. The ESRI file geodatabase which stores the results is then exported to an SQLite or DXF file.

The SQLite database format is preferred as the universal medium for data transfer because it is common between all three environments, open source, and stores polyline geometry as true arcs. However using SQLite requires a high level of skill to build and troubleshoot correctly; therefore, shapefile (loss of true arcs) or DXF (loss of attribute table) are lightweight alternative formats.

2.2. Filling of Blocks

This design move occurs at UIP_{30}. Whereas the setting out of grids is more efficiently done manually to achieve unique site-specific results, filling those grid cells with representative building masses is most efficiently done by an algorithm. By using VPI scripting a project-specific algorithm is accessible to most users in a lightweight way.

In this study a single *Grasshopper* script describes a single building typology which is parametrically adjustable to the block polygon, see table 3. In this study, two typologies were used: a courtyard block of two units, and a highrise with a podium. Following UI theory the algorithm for the highrise and podium was discretized into two scripts, one for each design move. The first script *CityEngHighrise.gph* sets out the podium and highrise footprints. The second script *CityEngHighriseExtrude.gph* extrudes heights by attractor system. All scripts employed randomization in the parameters to create a controlled variation and were constrained by daylighting depths.

Option 1: Transfer block polygon by DXF to *Grasshopper*. When using this method it is necessary to perform a basepoint shift, also known as affine transformation. The coordinate system in *Rhinocerouse3D* cannot support the large distance from the origin of georeferenced *Civil3D* files. The basepoint shift was based on the Lower Left (LL) corner of the bounding box as an integer number.

Option 2: Use the *slingshot!* plugin to read SQLite block polygons with attributes directly into *Grasshopper* VPI, or write arcpy script components for use in the *arcMap* VPI.

Table 3: Scripts.

Script	Description	Zone
CityEngCourtyardBlock.gph	Courtyard block with openings on two longest sides	T5
CityEngQuadBlocks.gph	Midrise Apartments with parking garage	T6-T5
CityEngHighrise.gph	Highrise and Podium footprint	T6
CityEngHighriseExtrude.gph	Highrise extrusion by transit station attractor	T6

Following the semi-automatic principle, the blocks were selected manually as input. This allows the same scripts to be applied block polygons in different transect zones or different scripts to be applied to block polygons in the same transect zones as needed by project goals. For example, the *CityEngCourtyardBlocks.gph* script was used with two different height range parameter values in transect zones T-5.1 and T-5.2. The approximate size of algorithm required to manage simple massing is shown in Figure 7.

Following the lightweight principle, the scripts generate two geometries: a mass by extrusion for use in CAD and the polygons of the mass's 'roof' for use in GIS. When the elevated 'roof' polygons are transferred to GIS, the ELEVATION property is stored as an attribute of the geometry. Transfer to GIS is done by moving the elevated polygons back to *Civil3D* through a reverse basepoint shift. Although *Civil3D* supports the exchange of complex mesh geometry, by restricting the exchange format to elevated polygons it is easier to exchange with *arcMap* or *Infraworks*.

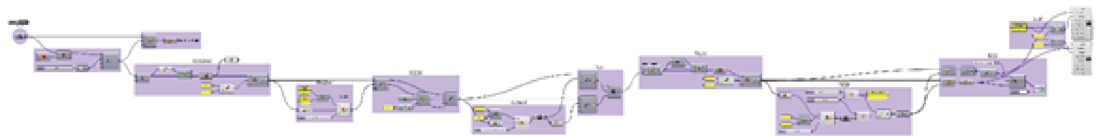

Figure 7: *CityEngCourtyardBlocks.gph* at UIP_{30}.

3. Results/discussion

The proposed Lightweight Semi-Automatic generative urban design tool was tested on an urban infill site in San Francisco. This tool is a specific implementation of the UI generation module from UIP_{24} through UIP_{30} which results in a 3D textured mesh representation of urban space inside a CIM environment, following the recommendations of [5–7] as shown in Figure 8 and 9. This tool innovates upon the *CIty Maker* UI generation module by analyzing a critical path of design moves as criteria to divide and distribute a limited scope of UIPs between GIS, CAD, VPI, and CIM softwares, to optimize for least scripting size. This dis-integrative implementation of UI allowed for UIP_{24} through UIP_{29} to be generated efficiently without scripting in *Civil3D*. Additional optimization was gained by merging the Form-Based Codes ontology into the UI ontology, which synthesizes multiple UIP parameters into a single UIP parameter called 'Transect Zone' stored in the LAYER property of block polygon geometry. This innovation renders unnecessary the plugins and scripting required to move attached data tables with geometry at the GIS to VPI transfer. Scripting was isolated to only the design move of block filling at UIP_{30}. Another innovation is the use of a CIM at the final design move, which merges circulation network (UIP_{24} - UIP_{29}) and building mass (UIP_{30}) data, and generates a textured 3D mesh in an integrated visualization environment.

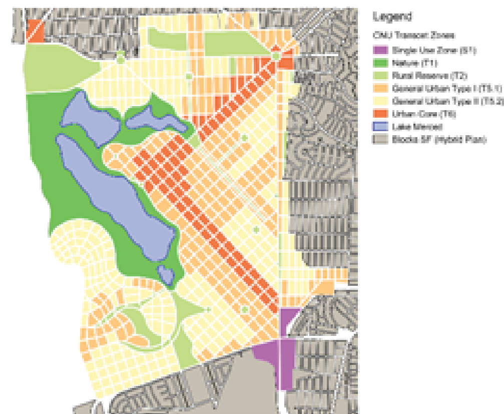

Figure 8: Input.

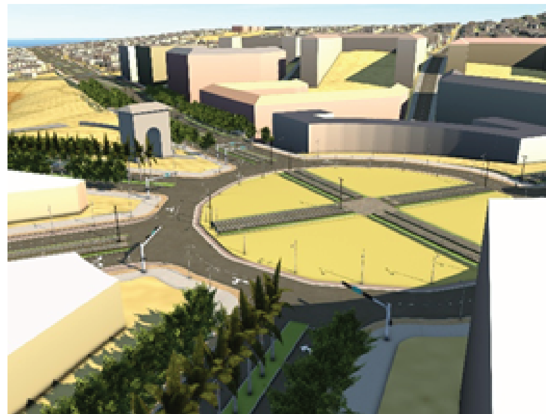

Figure 9: Output.

3.1. The critical point

The complexity of urban space first emerges at UIP_{30} when block polygons are packed with representative building masses. These masses are neither purely symbolic nor a true mass model but rather occupy a Level of Detail (LOD) in-between. Ideally, these building masses should be unique but also read *en masse* as coherent domains of a particular urban character specific to the project and site.

Generating a successful outcome for the block filling design move (UIP_{30}) is a unique problem, which can be solved by neither native CAD nor GIS functionality, nor can it be manually drafted efficiently. This unique combination of importance coupled with the difficulty in execution makes UIP_{30} the critical point of generative urban design. The only way to efficiently complete UIP_{30} is with scripting shape grammars [6]. By isolating the use of scripting to only UIP_{30} and using a VPI the gentlest possible learning curve is presented to the user, as shown by the blue curve in Figure 10, following the recommendations of [12]. User accessibility is further enhanced by the synthesis of UI

and Form-Based Code ontologies, which support the use of prototype script types for each transect zone typology.

During the development of *CityEngCourtyardBlocks.gph* it was discovered that placement of building masses was most efficiently done using the Right of Way (ROW) line as input for an inner offset toward the block interior with constraint by daylighting width. The results of this method yielded the most realistic results, in regards to constructability and code compliance, on irregular block geometry. This is might be because ROW lines underlie the framing of positive spatial void volumes, described in [10, 11] as the prerequisite to desirable streetscape design.

The courtyard block typology of *CityEngCourtyardBlocks.gph* is the simplest case of generation at UIP_{30} because the entire perimeter is used; however, in a high-rise or commercial street situation the complexity increases because a smaller domain of the total perimeter must be extracted to ensure that the building addresses the street in the correct orientation. During the development of *CityEngHighrise.gph* it was realized using street centerlines or transit stations as attractors in a classic attractor system was the most efficient method. Experiment showed best results with irregular block polygon geometry were obtained by the creation of a major axis between block polygon centroids and attractor points and also perpendicular to ROW lines as shown in Figure 11. This technique had the added benefit of introducing slight variation in shape after clipping.

3.2. Further development

The results indicate a Lightweight Semi-Automatic generation module of UI is conceptually possible, but it requires further research and development as a design tool. The ontology of UI and *CIty Maker* present an exciting opportunity to develop an open-source substitute for *CityEngine*.

Circulation network generation (UIP_{24}-UIP_{29}) can be improved by using visualLISP scripts to automate the conversion of SDF to SQLite. The use of the SQLite format as a medium of exchange needs further examination. In this study, SQLite databases generated outside of *Civil3D* failed to import correctly into *Civil3D*. Also replacing proprietary GIS *arcMap* with open-source GIS *QuantumGIS* should be studied.

Improvements at the critical point of $UIP_{30}$and downstream should include a comparison of *Grasshopper* versus *Dynamo* as the VPI. Using *Dynamo* as VPI would produce representative masses as BIM objects in *Revit*. When UIP_{30} is generated in a BIM environment a higher LOD beyond simple massing might be easier to automate.

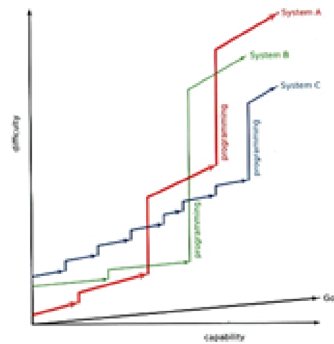

Figure 10: learning curves of scripting.

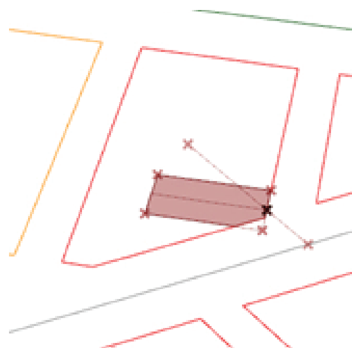

Figure 11: major axis generation.

Regardless of the VPI used, obtaining a higher LOD on the representative masses is a priority for improvement. This increase in LOD requires the parallel development of an efficient method to transfer 3D mesh from VPI back to GIS or CIM. The addition of a simple UI evaluation module using the Single Objective Optimization (SOO) or Multi-Objective Optimization (MOO) tools in *Grasshopper* could also be useful in higher creating LOD representative masses.

Acknowledgment

Thank you To Christiane Herr. Also thanks to all the reviewers who gave their valuable inputs to the manuscript and helped in completing the paper.

References

[1] Parish, Y. & Muller, P. (2001). Procedural Modeling of Cities, in *SIGGRAPH*. Los Angeles, CA: ACM.

[2] Duarte, J., Beirão, J., Montenegro, N., et al. (2012). City Induction: A Model for Formulating, Generating, and Evaluating Urban Designs *Communications in*

Computer and Information Science, pp. 79-104.

[3] Kim, J., Clayton, M., Yan, W. (2013), Parameterize Urban Design Codes with BIM and Object-Oriented Programming

[4] Schneider, C., Koltsova, A., Schmitt, G. (2011). Components for Parametric Urban Design in Grasshopper—From Street Network to Building Geometry.

[5] Beirão, J. (2012). CIty Maker—Designing Grammars for Urban Design, *Architecture and the Built Environment.* Vol.5, pp. 1-272.

[6] Derix, C. (2209). In-Between Architecture Computation, *International Journal of Architectural Computing*.

[7] Beirão, J., Arrobas, P., Duarte, J. (2012). Parametric Urban Design: Joining morphology and urban indicators in a single interactive model, *City Modelling* Vol. 1, pp. 167-176.

[8] Wortmannm, T., Tuncer, B. (2017). Differentiating parametric design: Digital workflows in contemporary architecture and construction, *Design Studies* Vol. 52. pp. 173-197.

[9] Duany Plater-Zyberk & Company. (2009). *SmartCode Version 9.2* Gaithersburg, MD: The Town Paper Publisher.

[10] Lynch, K. (1960). *Image of the City* Cambridge, MA: MIT Press.

[11] Alexander, C., Ishikawa, S., Silverstein, M. (1977). *A Pattern Language* New York, NY: Oxford University Press.

[12] Woodbury, R. (1977). *Elements of Parametric Design* New York, NY: Routledge.

[13] Horst, R., Webber, M. (1973). Dilemmas in a General Theory of Planning, *Policy Sciences* Vol. 4, pp. 155-169.

[14] Ferenstein, G. (2015). Here's What San Francisco Looks Like as an Affordable City, *Medium,* December, 11th.

[15] Tulloch, D. (2016). Working Toward a Taxonomy of Geodesign, *Transactions in GIS,* Vol. 21,28 pp. 635-646.

[16] Yanwen, L., Jiang H., Yuting, H., (2016). A Rule-based City Modeling Method For Supporting District Protective Planning, *Sustainable Cities and Society*, Vol. 28, pp. 277-286.

[17] Alexander, A. (1968). Systems Generating System, *Architectural Design*. Vol. 7, pp. 90-91