www.itcon.org - Journal of Information Technology in Construction - ISSN 1874-4753

INTEGRATED GENERATIVE TECHNIQUE FOR INTERACTIVE DESIGN OF BRICKWORKS

PUBLISHED: August 2014 at http://www.itcon.org/2014/13
GUEST EDITORS: Bhzad Sidawi and Neveen Hamza

Kereshmeh Afsari, PhD Student
Georgia Institute of Technology, http://www.gatech.edu/
kafsari3@gatech.edu

Matthew E. Swarts, Research Faculty
Georgia Institute of Technology
matthew.swarts@coa.gatech.edu

T. Russell Gentry, Associate Professor
Georgia Institute of Technology
russell.gentry@coa.gatech.edu

SUMMARY: *Bricks have been used in the construction industry as a building medium for millennia. Distinct patterns of bricks depict the unique aesthetic intentions found in Roman, Gothic and Islamic architecture. In contemporary practice, the use of digital tools in design has enabled methodologies for creating new forms in architecture. CAD and BIM systems provide new opportunities for designers to create parametric objects for building form generation. In masonry design, there exists an inherent contradiction between traditional patterns in brick design, which are formal and prescribed, and the potential for new patterns generated using design scripting. In addition, current tools do not provide interactive techniques for the design of the brickwork patterns in a way that changes of the pattern can be managed parametrically while being mapped to the brick wall in real time. An interactive technique can help to inform and influence the design process, by providing constant feedback on the constructive aspects of the proposed brick pattern and its geometry. This research looks into the parametric techniques that can be applied to create different kinds of patterns on brick walls. It discusses a methodology for an interactive brickwork design within generative techniques. By integrating data between two computational platforms – the first based on image analysis and the second on parametric modeling, we demonstrate a methodology and application that can generate interactive arbitrary patterns and map it to the brick wall in real-time.*

KEYWORDS: *Brickwork, interactive design, generative design, brick patterns, UDP, Sketchpad.*

REFERENCE: *Kereshmeh Afsari, Matthew E. Swarts, T. Russell Gentry (2014). Integrated generative technique for interactive design of brickworks, Journal of Information Technology in Construction (ITcon), Vol. 19, pg. 225-247, http://www.itcon.org/2014/13*

1. INTRODUCTION

1.1. Background

Bricks are commonly used in the construction of buildings and are one of the oldest known building materials (Chamberlain, 2003) dating back to 7000BC. Bricks were used during the time of the Ancient Egyptians, the Romans, the Greeks, ancient Mesopotamians and the gothic period when they became very common in northern Europe. Each era distinguished and introduced specific characteristics of bricks in the buildings. Although during the renaissance and Baroque periods, exposed brick walls were often covered in plaster, in the middle-18th century they again became popular as an exposed aesthetic element in buildings (The Brick Directory, 2012).

Brick's aesthetic is represented through history in many architectural styles which can be attributed primarily to the visual patterns expressed on the façade of buildings (Sullivan & Horwitz-Bennett, 2008). These patterns work at multiple scales are often achieved by placing a single brick type into one of several possible positions within the overall coursework. For this reason, such patterning is an economical technique to add design complexity and interest to brick walls and is a common characteristic of many early buildings (Buck, 2003). Patterned brickwork was the favorite building technique in the eastern Islamic province originating from the ancient civilizations of this area. The use of brick in this region progressed from purely structural purposes towards a more decorative complexity expressed in a variety of brick bonds that created patterns of light and shade thus generating strong visual effects on the walls (Islamic Art, 2007). The architecture of Iran during the Seljuk period is a pivotal period in terms of style in using various methods of decorative brickworks (Panahi, 2012; Saoud, 2003). In contemporary architecture, there are some examples of brickwork design enabled by new technologies and robotic production that practice new methods of brick bonding. Some examples in this regard are the works by Gramazio and Kohler architects (Gramazio and Kohler, 2006) as well as Zwarts and Jansma architects (Zwarts and Jansma Architects, 2014).

Most patterns of brickwork adhere to a common module that facilitates the dimensioning of the brickwork – the dimensions of walls and openings are driven by relationships between the width, height and length of the bricks themselves: "relationships allow rowlocks and headers to tie adjacent wythes together and courses of brick in different orientations" (Brick Industry Association, 2009). In order to understand these relationships, manipulate them during design and integrate them with formal design objectives, the brick objects and patterns must be defined parametrically. In this way, we can select patterns and adjust geometries without the need to continuously remodel (Gentry, 2013) . To achieve this parametric linkage, we must formailze the secret rules of modules of bricks and their geometry of patterns and integrate them into parametric relations which can assist in the design process by reducing re-modeling efforts.

1.2. Parametric techniques in designing brick wall

Despite the long history of bricks in the built environment, there are currently limited possibilities that can be applied in the brickwork design. In this regard, digital technologies can assist in developing proper methodologies (Al-Haddad, Gentry and Cavieres, 2011). Brick walls consist of uniformly shaped and sized bricks that are laid in courses with mortar joints (Chamberlain, 2003). This feature makes the parametric design of the brick walls possible to provide designers with the ability to parametrically test formal concepts and also to manage continuous changes of the design. We believe that parametric models for brickworks, instantiated within digital tools should be developed. One challenge in this regard is to find a generic methodology and an appropriate set of parameters that can generate the required parametric behaviors (Sacks et al, 2004).

In Building Information Modeling (BIM) authoring tools, as with newly adopted parametric modeling techniques, components like walls, slabs and windows, are defined to represent an entire assembly (Goedert and Meadati, 2008). When these types of parametric components, for instance a wall, are used, then all the different components of the wall are combined and behave as a single entity. These tools are currently limited as they do not allow the selection of the components of the wall (e.g. bricks) independently, so there is little possibility to access the basic components like the brick and permute their geometry representations while preserving the linked definition of the whole assembly.

To embed design expertise, the basic unit that is required is a parametric component that holds internal geometric relationships and deals with external rules to represent the assembly of masonry components. If the design changes, these parametric relationships will allow constant application of changes automatically in order to organize the assembly within the coherent topology and geometry of its components. Thus, the components have to be modeled both by their appearance and by their semantic relationships within their specific domain

(Al-Haddad, Gentry and Cavieres, 2010). Therefore, the methodology needed for implementing patterns on brick walls should be a components-based parametric technique to generate the geometry of the whole assembly while providing access to each of the components. These parametric relations for generating brick patterns should maintain the geometric consistency of the components within the assembly.

1.3. Research aims and objectives

Most of the recent parametric design tools with BIM authoring capabilities define a wall as a single entity and cannot provide access to the geometry representation of the units of the wall so that the designer can create patterns/brick bonding. Current software design tools that can provide techniques to design a parametric brick wall such as BrickDesign plug-in for Rhino (ROB Technologies, 2012) have limited capabilities when it comes to the brick bonding and brickwork patterns. These current solutions provide means to map an image as the pattern on the brick wall but if a designer wants to manipulate the mapped image on the brick wall and changes the patterns, he should implement changes on the image separately and re-import it again into the design platform. As design progresses, the need for automating these iterations becomes important to reduce the time spent between applications. In contemporary architecture, the works by Gramazio and Kohler architects such as Gantenbein Vineyard- which has a contemporary brick facade delivered with robotic production method- (Gramazio and Kohler, 2006) as well as the works by Zwarts and Jansma architects on the design of brick surfaces- with a tool that translates a grayscale picture into brick bonds- (Zwarts and Jansma Architects, 2014) have demonstrated the power of automating this process but there are very few academic studies that document these new techniques for creating brickworks.

This research describes the technical issues and proposes a methodology for interactive design of patterns on brick walls. The physical requirements for preserving load bearing capacity in such complex design situations has been discussed in prior work (Al-Haddad et al., 2012). A complete discussion of the physical, structural or aesthetic judgments resulting from such patterning is beyond the scope of this paper – but the readers are pointed to the work of Moravánszky (Moravánszky, 2002).

1.4. Brickwork patterns

One key parameter for bricks is the position of the brick in the overall pattern. For six common positions, the position nomenclature is as follows: stretcher, header, rowlock, rowlock stretcher, sailor, and soldier positions (Sullivan and Horwitz-Bennett, 2008; Brick Industry Association, 2009). The relationship between and repeats between and among the six positions defined the overall bonding pattern. Six traditional patterns are shown in Figure 1.

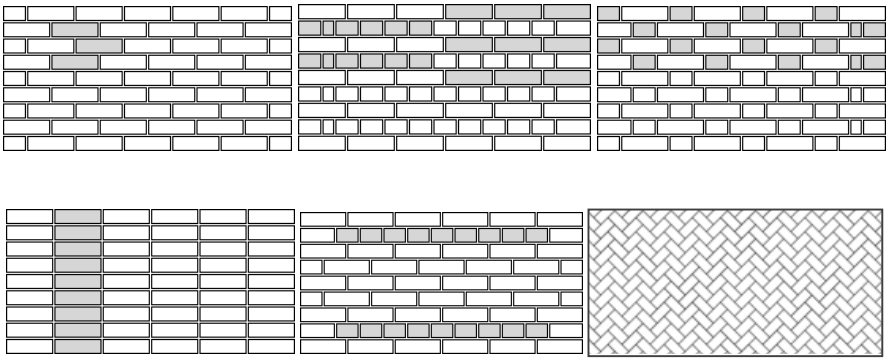

FIG. 1: Traditional pattern brick bonding (From left to right; Top: running, English, Flemish; Bottom: stack, common, Herringbone)

1.5. Computational methods for brickwork design

In addition to traditional brickwork patterns, there are other techniques for creating patterns on brick walls that have emerged over time. Designers are constantly seeking out alternative patterns (Sullivan and Horwitz-Bennett, 2008). In general, pattern may refer to the different arrangement of the brick texture or color which is exposed in the face. For that reason, many patterns may be possible to be produced by using the same bond. In addition to the feature of the individual brick, there are other methods that can produce distinctive patterns on brick walls. These methods can include the method of handling the mortar joint and the method of offsetting

specific bricks into or out of the neutral plane of the wall (The Brick Industry Association, 1999). Overall, the main factors in designing brickwork patterns that need to be considered in the computation methods are as below.

- Brick shape
- Brick texture
- Brick color
- Bond pattern
- Relative placement of bricks in relation to the bonding plane
- Types of mortar joints

The first five factors deal with the brick elements while the last factor is related to the mortar joint. In brickwork, a considerable amount of the surface of the wall is covered by the mortar joint and mortar joint can change the appearance of the wall within different profiles.

In this study, the proposed computational techniques look into the first five factors that are related to the block itself. We have reviewed a number of techniques for expressing a pattern of brickworks onto an otherwise uniform bonding plane. In the text below, we discuss a procedure to develop patterns using one of the three operations on individual bricks. These operations are described as "select", "corbel" and "yaw".

1.5.1. Select

For this technique, the designer should be able to select base modules in different colors and shapes as well as the different size of bricks produced by different manufacturers (Sullivan and Horwitz-Bennett, 2008). Upon selection of the base modules, the units can be combined to generate a chosen pattern either through mapping a traditional or other bonding pattern or by mapping a digital image onto the wall. Figure 2 indicates how different brickwork design can be achieved within one particular bonding pattern but by utilizing different colors and shapes of the bricks.

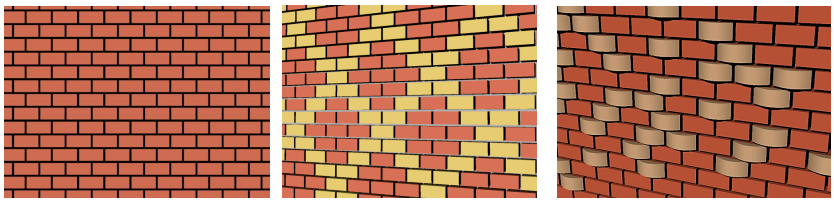

FIG. 2: Brick patterns by using one bonding pattern and different colors/shapes of bricks

1.5.2. Corbel

Corbel as a solid unit in the wall protruding from it, can create several patterns on brick walls by offsetting successive courses of bricks. This has traditionally been used in many historic buildings and it is also used in some examples of contemporary architecture. Figure 3 shows examples of this pattern in 3D models.

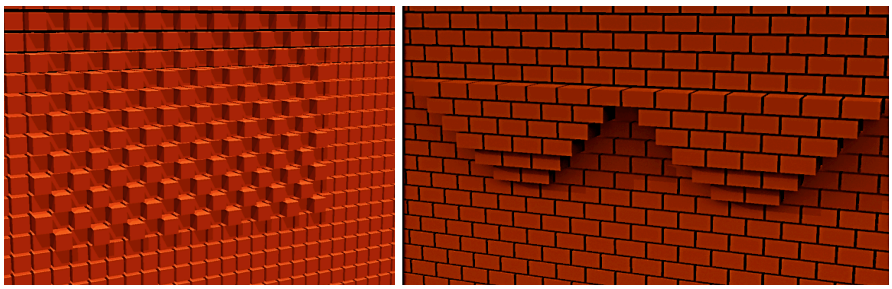

FIG. 3: Examples of corbel patterning

1.5.3. Yaw

Yaw is a rotation that is an aviation term for this particular rotation around one of the Euler axes (i.e. yaw axis) and in this context is a rotation around the vertical axis through a masonry unit (Figure 4). This specific method of brickwork can be found in contemporary buildings.

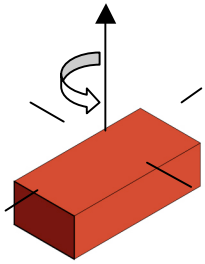

FIG. 4: Yaw rotation of a brick

Gantenbein Vineyard Façade designed by Gramazio & Kohler, is an example of this type of brickwork. The building is a large fermentation room for processing grapes and its façade resembles a big basket filled with grapes transferred the digital image data to the rotation of the individual bricks (Gramazio and Kohle, 2006). Another example is the Design Studio II building of the Southern Polytechnic State University (Gentry, 2013) designed by Cooper Carry Architects that has brick facades with rotating bricks (Figure 5).

FIG. 5: Façade of the Southern Polytechnic and State University (Gentry, 2013)

The case studies highlighted above are intended to motivate the generative algorithms presented below – for a more detailed description, of the projects, see the prior paper by Gentry (2013). In the text that follows, we present the methodology for automating the design process using the "yaw" technique of brickwork patterning.

1.6. Interactive design solution

Interactivity is known as a key component of the new media which is identified based on user control, responsiveness, real time interactions, connectedness, personalization/customization and playfulness. Machine-interactivity that happens between human and machine is related to the process of feedbacks. In other words, how a system can be controlled by reusing the result of its previous performance is a critical factor in an interactive environment (Dholakia et. all, 2000). In current software applications, there are limitations in designing a pattern and mapping it to a brick wall. Firstly, some tools such as BIM authoring tools, do not provide direct access to the geometry of each block within a brick wall. They treat an assembly, for instance a wall, as a single entity without dealing with the geometry of the elemental pieces that generate the assembly. Thus in general, they provide very limited capabilities for designing patterns on a block wall although there are ways to accomplish this. For example, in Revit, for changing the pattern of bricks, a separate wall needs to be created. This could be created as a wall hosted family that either cuts its host or adds elements to the face. Even though it is possible to create a brick wall that contains the geometry of the individual blocks, whether in BIM tools or other 3D modeling software programs, for mapping a pattern on the wall there are challenges. In fact, these two (i.e. design of the image of the pattern and design of the brick wall) occurs in two separate platforms. In order to map an image as a pattern on a 3D brick wall, a designer needs to go back and forth within two platforms. One platform deals with designing and editing the image for pattern. It is usually an image editing program that works as a sketch pad. The other platform, as a 3D modeling environment, deals with mapping the

image on a 3D wall as well as changing the intensity of the pattern or the dimensions and placements of the individual blocks. This process is illustrated in Figure 6.

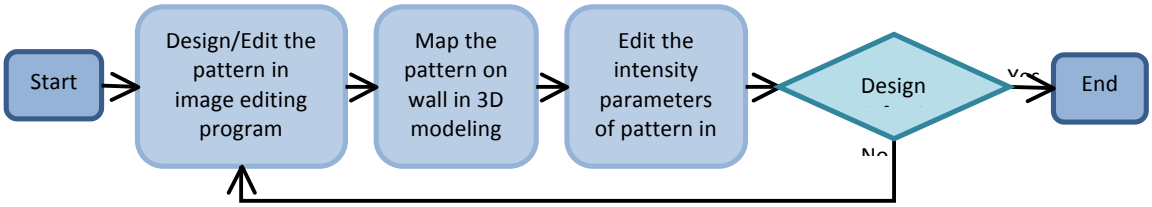

FIG. 6: The process of designing patterns on 3D brick walls in current tools

Architectural design is an iterative design process which is based on evaluating, analyzing and refining the design over time. Therefore, the process of using two separate platforms in designing the brickwork is an obstacle in implementing the design changes. Real-time interaction with both image and 3D assembly would help the designer to interactively create and edit brickworks. This can help in managing the design changes and would facilitate the decision-making process during conceptual design.This user interaction poses a composite structure that can automate the process of this particular design feedback loop. It indicates the need for a process that connects these two platforms (3D modeling environment and sketchpad tool) to let the system parse the input from the sketch pad within the 3D modeling tool in real-time. In this study we will show a technique for this integration.

2. STUDY APPROACH

The study has so far identified the gaps in the existing knowledge and introduced computational techniques for the development of brickwork design. The main contribution of this paper, discussed below, is a methodology for a computational solution to interactively create patterns on brick walls through the yaw rotation of the bricks. The computational technique and digital innovation discussed in this paper, integrates multiple platforms with different generative design capabilities in realizing the brickwork design. This technique can map an image to the brickwork within an interactive, iterative solution that can manage design changes and designer feedback in real-time. The brickwork sample is represented in Figure 7. A generative design technique is used that integrates the data between two computational platforms to achieve an interactive solution. The approach implemented initially as an experiment in design scripting course in the College of Architecture at Georgia Institute of Technology and developed further in this study.

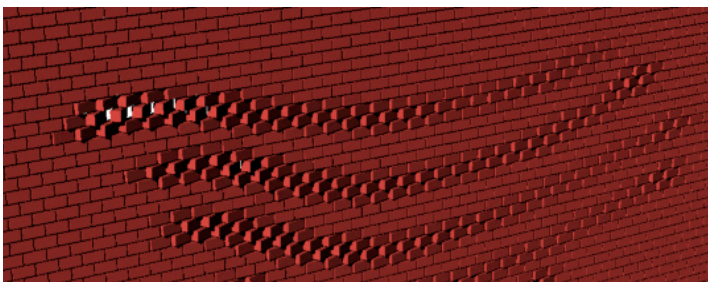

FIG. 7: Example result of the computational brickwork technique developed in this study

3. METHODOLOGY

This study introduces a methodology for designing brickwork that is based on "yaw" control of the brickwork pattern. It integrates data between two design platforms (Grasshopper plug-in for Rhino and Processing which is a Java-based programming language and integrated development environment). The methodology uses scripting techniques to achieve interactive design of brickworks in real-time. In order to demonstrate the tool, we need a platform to visualize 3D bricks where the position of each brick can be accessed individually. Each block needs to be rotated with a specific angle to contribute to the overall pattern. The foundation platform for the application is Rhinoceros and the Grasshopper plug-in. Also, a sketching tool is required where we can create images interactively to drive the desired pattern. Therefore, the visual programming language Processing is used to

generate the sketch-pad for free-form graphic input. Finally, an approach for data transfer is needed to map the sketch of the pattern (in Processing) to the brick wall in the 3D environment (in Rhino) to generate patterns on the wall in real-time. To facilitate this interaction, serial data transmission using the user datagram protocol (UDP) was selected. The method of data processing in this study is summarized in Figure 8. The flow of data in each step is described in what follows.

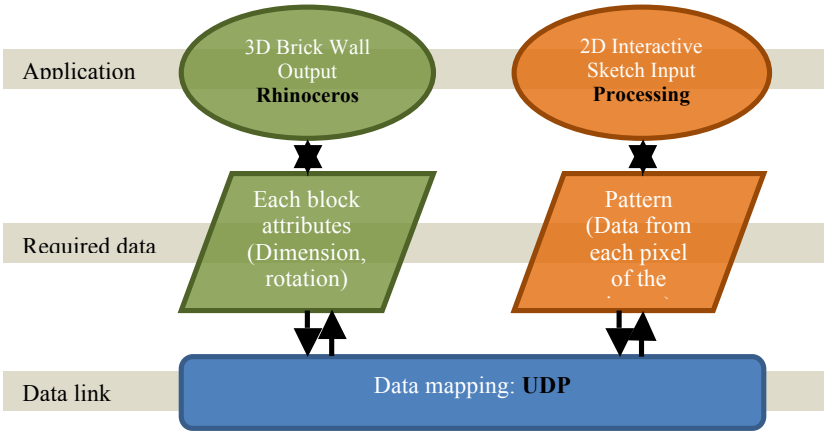

FIG. 8: Diagram of data exchange in this study

3.1. Parametric bricks

Firstly, we have implemented parametric behaviors of the components for the brick wall within a Grasshopper VB scripting component. This algorithm, shown in Figure 9, has six input variables, namely: brick width, brick height, brick depth, mortar thickness, number of rows and columns of the bricks in the overall wall height and wall width respectively. These parameters handle different design possibilities of the brick wall. For example changing the values for brick width, brick height and brick depth will result in different sizes of the blocks and will create different brick walls accordingly.

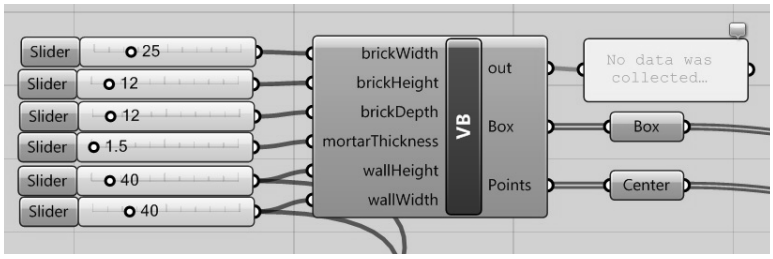

FIG. 9: VB component for parametric brick wall

Algorithm 1 creating parametric brick wall
1: **Create** and set a direction to +1
2: **For** each of the bricks in each row
3: **For** each of the columns
4: **Set** the direction to negative direction
5: **Compute** an x value with brick and mortar width
6: **Compute** a z value with brick and mortar height
7: **Create** a center point with x and z
8: **Create** a box using the center point and brick size
9: **Add** the box to a list of boxes
10:**Return** the list of boxes and centroids

FIG. 10: Algorithm for creating the parametric brick wall

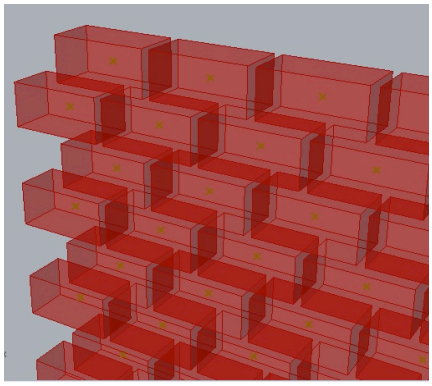

FIG. 11: Boxes (bricks) and their centroids as the output parameters of the VB component

This code (Figure 10) can generate wall components parametrically with two output parameters: "Box" as the geometry representation of bricks and "Center" as the series of centroids of each brick shown in Figure 11.

3.2. Sketching tool

After generating the parametric brick wall, we need to create a tool that can generate sketch patterns and images interactively. For this, we have chosen "Processing" which is an open source programming environment for rapid development of interactive prototypes that is based on the Java programming language (Reas, Fry and Maeda, 2007). There are several libraries developed specifically for the Processing environment for video, 3D geometry, simulation, audio, and data transfer.

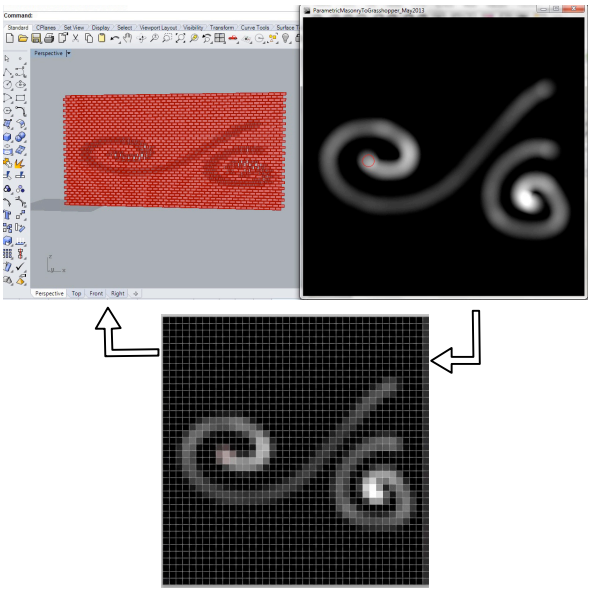

FIG. 12: Patterning example for a 40x40 brick wall, Top-right: Processing interactive sketch pad, Top-left: 3D model of brick wall in Rhino, Bottom: visualization of the pixel configuration for the image mapped to bricks.

A program in Processing was created to produce the sketch pad. In this sketch tool, images of any arbitrary sketch can be created and then the code maps each pixel of the image to greyscale. Accordingly, the value of each pixel in greyscale is calculated and mapped to the 3D visualization platform to be used as a rotation angle for each brick component of the wall. Thus, each pixel of this sketch tool represents a brick of the wall and based on the greyscale intensity (from white to black), it holds a value that defines the degrees each brick should rotate to contribute in generating the pattern of this image on the wall. Figure 12 depicts this notion. Furthermore, the Processing program can open a given digital image and map it as the initial pattern. The image can be edited to achieve the desired sketch which is subsequently mapped to the brick wall.

The "setup" function in the Processing code defines the base sketch pad setup, like its size. The "draw" function is called in each frame. The UDP object used within setup function and this specific data transfer protocol is discussed in section 3.3. In addition to these two fundamental functions, there are three major functions that can

handle the sketching in real-time by dragging the mouse cursor with a brush tool. This brush is visually represented by a red circle attached to the mouse cursor as shown in Figure 13. The three major functions are drawBrush, drawBrushIndicator and BuildMap functions.

The drawBrush function (Figure 14) manages drawing on the canvas through the movement of the mouse. The drawBrushIndicator creates the pointer for the brush. Moreover, three events for the mouse interactions are designed: pressed, dragged and moved. Also, the pointer's size can be changed by two functions increaseBrushRadius and decreaseBrushRadius during the sketching to cover bigger or smaller surfaces as required.

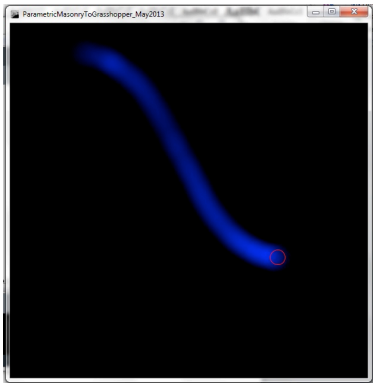

FIG. 13: Sketch tool canvas and brush created in Processing

Algorithm 2 drawBrush function

1: **Load** the pixels in the provided canvas

2: **Set** the brush center as mouse pos relative canvas

3: For each pixel in a square around the brush center

4: **If** the pixel is within the bounds of the canvas

5: **Calc** inverse square distance to brush center

6: **Div** inverse square by 32 and set as intensity

7: **Get** the current value of the pixel

8: **If** it is the left mouse button

9: **Add** intensity to current pixel, max of 255

10: **If** it is the right mouse button

11: **Subtract** intensity from curr pixel, min of 0

12: **Update** the pixels in the provided canvas

FIG. 14: drawBrush function

The BuildMap function (Figure 15) deals with creating the data from the sketch and its pixels by loading and updating rows and columns of pixels. The number of rows and columns are coordinated with the 3D model according to the number of bricks used within the width and height of the brick wall. The techniques of such data transfer are discussed in the next section.

Algorithm 3 BuildMap function

1: **Create** image with size of the program window

2: **Calculate** column pixel widths

3: **Calculate** row pixel widths

4: **Load** the new image

5: **For** each pixel

6: **Set** the pixel value to 0

7: **Close** the Image

8: **Draw** the Image to the program window

FIG. 15: BuildMap function

3.2.1. Normal mapping & heightmap

Normal mapping, a type of bump mapping, is a technique that models a complex surface in a way that lightens a 3D model with a low polygon count. It stores a surface normal in a texture map by specifying the surface normal at each texel. In the normal map, the RGB values of each texel serve as the x,y,z components of the normals from the normal map texture at that texel (Han et. al, 2007; Sloan, 2006). We utilize the base drawing canvas as a normal map, storing the x gradient in the red channel, the y gradient in the green channel, and the height map in the blue channel. The gradients are calculated using a simple 3x3 kernel matrix Sobel operator for calculating slopes in both the x direction and y direction independently. Figure 16 describes the algorithm. By storing the output in the red and green channels, we remove the need for a secondary image buffer when calculating the vertical and horizontal slope values from the height map in the blue channel.

Algorithm 4 calcImageGradient function

1: **Load** the pixels in the provided canvas

2: **For** each pixel in the canvas

3: **Compute** the x direction gradient from blue channel

4: **Set** the red channel to the x gradient

5: **Compute** the y direction gradient from blue channel

6: **Set** the green channel to the y gradient

7: **Close** the canvas object

FIG. 16: Calculating the image gradient

The user interaction with the program is, through the movement of the mouse as well as pressing specific letters on the keyboard to apply specific functions such as opening a file or sending data to Grasshopper. The keyPressed event is summarized in Table 1. For instance, when the sketch is ready, by pressing “c” key, color channels are summed to the blue channel and by pressing “s” key, the program sends the data to grasshopper.

TABLE 1: Keys and functions used in keyPressed event for defining keyboard interactions

Key	Function
'+' or '='	Increase the brush radius
'_' or '-'	Decrease the brush radius
'g' or **'G'**	Calculate the image gradient
'c' or **'C'**	Replicate the blue channel
's' or **'S'**	Send map to Grasshopper
'o' or **'O'**	Open an Image file

3.3. Data mapping

Having created the sketch in the Processing program, the data needs to be mapped to the 3D brick wall within Rhino and Grasshopper. For this purpose, UDP (User Datagram Protocol) is selected. UDP-based protocols are often used for transferring large amounts of data, especially when it is on the same computer. UDP does not have error checking, so there is no room for error in the data transmission. While it is easy to implement these protocols, they can provide better portability (Gu and Grossman, 2005). In Grasshopper, we used the gHowl plug-in that contains UDP components to facilitate communication with the Processing program. Then within Processing sketch, the UDP library "hypermedia.net" is imported to create UDP objects. So, in grasshopper, two sets of UDP components are used: first, for sending data to Processing such as the number of bricks in rows and columns (Figure 17) and second, for receiving data sent from Processing (Figure 18) that contains the greyscale values of each pixel to be used later to generate the rotation angles.

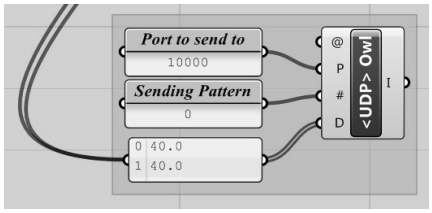

FIG. 17: UDP components in grasshopper to send data to processing

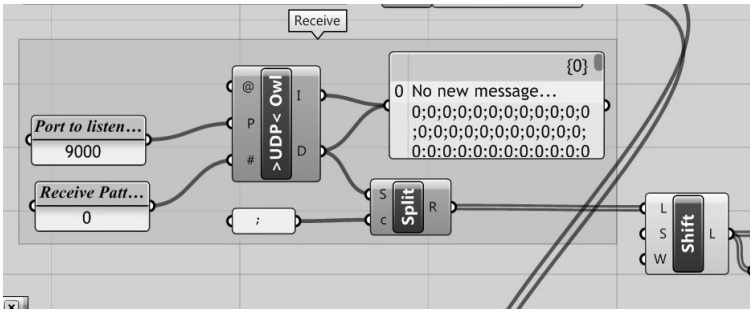

FIG. 18: UDP components in grasshopper to receive data from processing

On the other hand, in the Processing program, UDP objects are defined for both sending and receiving data. IP address and ports for incoming and outgoing data is specified. The ports are coordinated with the ports in the UDP components of grasshopper. It is important to select ports which are not commonly used by other software or services. Table 2 shows the UDP-related variables in the Processing.

Moreover, in Processing, two main functions are dealing with data mapping: 1) receive; for receiving data from Grasshopper. This gets and sets the number of columns and rows based on the number of bricks received from the Grasshopper. Figure 19 describes the code; 2) sendMapToGrasshopper function that sets a message containing the greyscale values calculated for each pixel and sends it to Grasshopper (Figure 20).

TABLE 2: Processing UDP-related variables: UDP objects, IP and ports

Item	Data type	name	Description
UDP objects	UDP	udps_outgoing	For sending data
	UDP	udpr_incoming	For receiving data
IP	String	ip	The remote IP address
Ports	int	port_outgoing_remote = 9000	the remote port to send to grasshopper
	int	port_outgoing_local = 6005	the local port from which data is sent
	int	port_incoming1_local = 10000	the local port to receive data from grasshopper

Algorithm 5 receive function

1:**Receive** an array of bytes from a port on an IP address

2:**Convert** the bytes to a string of characters

3:**Split** the characters by the endline marker

4:**Store** in an array of strings

5:**For** each string in the string array

6: **Convert** the string into a float value

7: **Store** value in a float array

8:**Check** for that the first 2 values in array exist

9: **Set** an update Boolean to false

10: **If** the column count is different than current

11: **Set** the update Boolean to true

12: **If** the row count is different than current

13: **Set** the update Boolean to true

14: **If** the update Boolean is true

15: **Call** the BuildMap function to rebuild image

FIG. 19: Function for receiving data from grasshopper (in Processing)

UDP can send "string" data. Thus in both Grasshopper and Processing, when sending the data which initially represents values as integer, it needs to be converted to string data type. Later, when the data is received, it again needs to be cast back to an integer data type so that it can be processed. While this is somewhat inefficient, it makes debugging significantly easier as the data is human readable. We found that for our purposes, the inefficiency was not an issue with the data transmission speed and processing; however improvements could easily be made if the data transmission needs are increased. Finally, for generating the pattern on the brick wall,

the values received in Grasshopper should be mapped to the angle of rotations for each brick. Figure 21 shows the grasshopper components being used to map the received values from the sketch and generate the appropriate angle to rotate each brick.

Algorithm 6 sendMapToGrasshopper function

1: **Create** an empty string to store the message

2: **Load** the bitmap

3: **For** each column

4: **For** each row, in reverse order

5: **Get** the value of the x-gradient channel

6: **Add** the value as a string to the message

7: **Send** the entire message to a port using UDP

8: **Close** the bitmap

FIG. 20: Function defining the data being sent to grasshopper

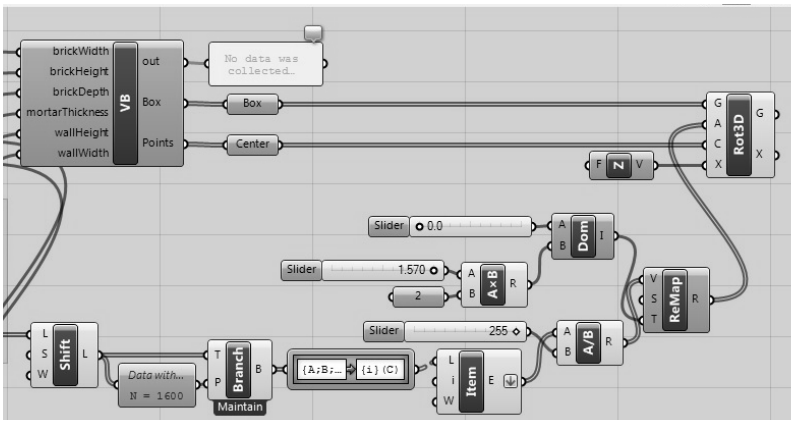

FIG. 21: Grasshopper components dealing with mapping data to the rotation of the bricks

4. RESULTS

This program produces an interface that integrates two programs, Rhino/Grasshopper with Processing (Figure 22). It uses a brush tool to create sketches on a sketch pad within processing and then map the image as patterns on brick wall in Rhino platform. Figure 23 shows the conceptual framework of the interface.

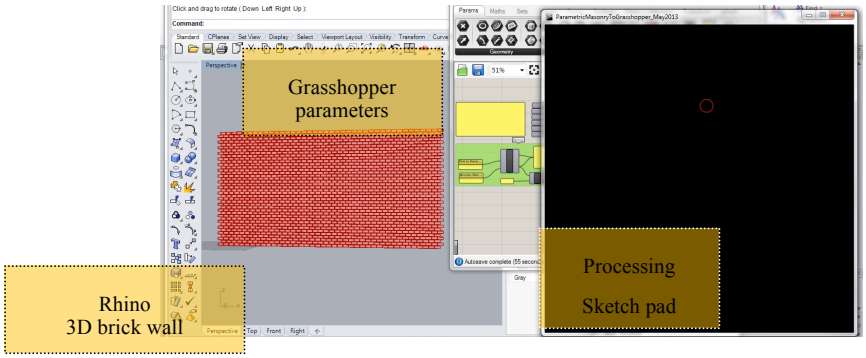

FIG. 22: Interface of the tool

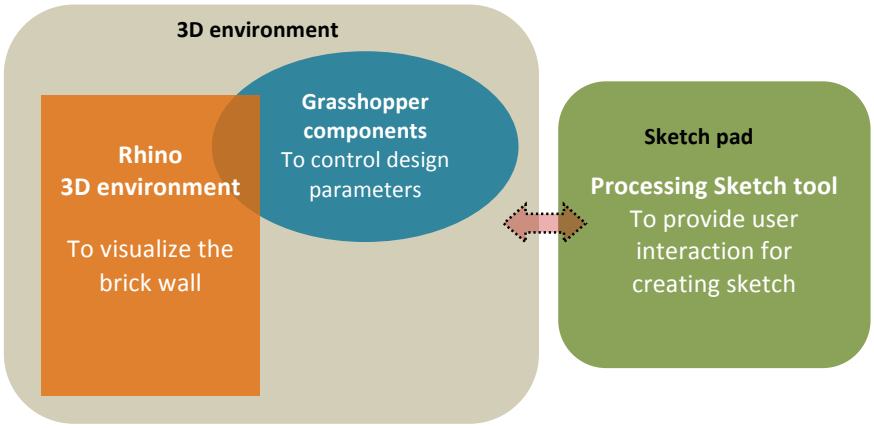

FIG. 23: Conceptual framework of the interface

This program integrates two capabilities: a) It can create a sketch from scratch; b) It can create sketches on a given image. For creating a sketch from scratch, an arbitrary sketch can be drawn by using the custom built brush on the sketch pad of Processing with mouse interaction, and then the image can be mapped to the Rhino environment through keyboard interactions to create the pattern on the 3D brick wall. Figure 24 shows an example.

For creating a sketch on a given image, the program can import an image file into the canvas through the "openImageFile" function. It uses JFileChooser java class which provides a simple mechanism for the user to choose a file from a filebrowser.

Upon opening an image file, the canvas will be updated. This part of code uses a function as "SetHeightMapWithNewImage" (Figure 25) that updates the canvas and its pixel configuration based on the dimensions of the imported image.

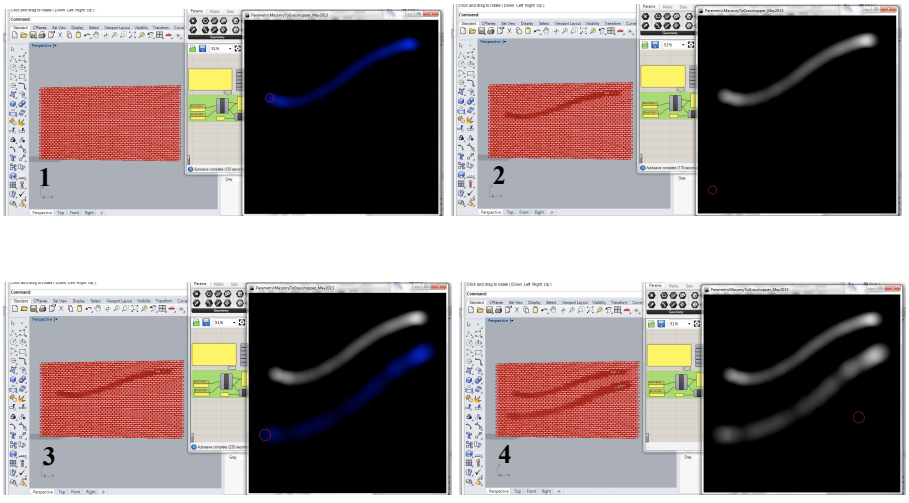

FIG. 24: An example of creating sketch on processing and mapping the image to 3D brick wall in Rhino (1: draw a sketch, 2: map the pattern on the brick wall, 3: edit the sketch, 4: map the edited image on the wall)

Algorithm 7 SetHeightMapWithNewImage function

1: **Create** new image with same size as input image

2: **For** each pixel

3: Copy input image pixel value to the new image

4: **Set** the size of the program window to the width and height of the new image

5: **Re-compute** the pixel width of each column

6: **Re-compute** the pixel height of each row

7: **Draw** the new image to the program window

FIG. 25: Code to handle updating the "heightmap"

Then the image can be manipulated by the brush tool to achieve the desired result and later can be mapped to the brick wall. Figure 26 shows how the program works on a given image to map the pattern on the brick wall. This program can map the pattern of any sketch on the brick wall apart from some of the details of the image. The pattern is in fact an abstract picture of the original image.

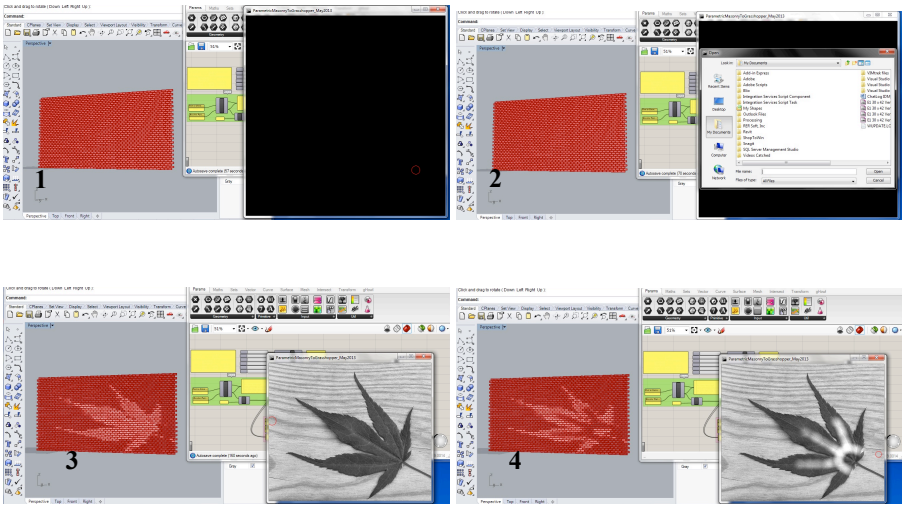

FIG. 26: An example of creating sketch based on a given image and mapping the image to 3D brick wall in Rhino (1: starting the program, 2: opening an image, 3: mapping the pattern of the image on the wall, 4: editing the image and mapping it on the wall)

5. LIMITATIONS AND FUTURE WORK

One limitation of this study is related to positioning and laying the bricks in a wall assembly. The implementation was limited to stretcher face bricks laid in running bond in a single wythe. Additional complexity will have to be handled to extend this work to multiple wythe, reinforced brick masonry, composite Concrete Masonry Units (CMUs) and brick and cavity walls (Chamberlain, 2003). Positioning and other laying possibilities of the bricks can be associated with the algorithm in future works. Moreover, according to Gentry (2013), computational development for early stage design of masonry should be linked to later-stage structural analysis and detailing. Therefore, future research can integrate these studies as well. Besides, further architectural studies regarding design constraints such as connections to the doors and windows as well as brick laying in the corners and lintels would be beneficial.

6. CONCLUSION

So far the digital tools that can create the design of patterns on the brick walls have been limited and do not support interactive design. Few tools have capabilities to represent a parametric brick wall encompassing the data attributes related to each masonry unit in the wall. We know that brickwork has a long history and that the use of patterning on the brick walls has been popular through the centuries. Hence, by improving brickwork techniques within digital tools, brickwork pattern design can be revitalized and applied effectively in contemporary architectural practice. This paper demonstrates one approach for the computational development of brickwork patterns using a digital technique.

This research categorized the computational techniques for creating contemporary patterns on brick walls as "corbel", "yaw" and "select" techniques. An experimental prototype was demonstrated for designing interactive brickwork patterns using the "yaw" technique. It showed that data integration between Rhino and Processing, through scripting techniques and the UDP can provide interactive solution for this purpose. Rhino and Grasshopper control the 3D bricks by managing the wall components. This platform is integrated with a sketch tool created in Processing for sketching patterns interactively. Then through UDP-based data exchange, the sketch is mapped to the brick wall in real-time. Thus the program can generate interactive parametric patterns on the brick wall based on sketches. These sketches can be created from scratch or by importing an image providing arbitrary patterns for brickwork design.

7. REFERENCES

Al-Haddad, T., Gentry, R., Cavieres, A., Carpo, M., Cho, J., Wagner, L. (2010). Representation + Fabrication: Connecting descriptions and artifacts in the digital age. Architectural Research Centers Consortium . Washington DC.

Al-Haddad, T., Gentry, T. R., & Cavieres, A. (2011). Digitally Augmented Masonry: Application of Digital Technologies to the Design and Constrcution of Unconventional Masonry Structures. The North American Masonry Conference (pp. 37-48). Masonry Society , Boulder, CO.

Bricks and Brass (2012). Bricks and Brickwork in the Period Home. Retrieved May 2013, from Bricks and Brass, http://www.bricksandbrass.co.uk/design_by_element/external_wall/bricks_and_brickwork_in_period_home.php

Buck, S. (2003). Seventeenth-Century Precendents in Brick Construction in England and Virginia. CWF.

Cavieres, A., Gentry, R., & Al-Haddad, T. (2011). Knowledge-based parametric tools for concrete masonry walls: Conceptual design and preliminary structural analysis. Automation in Construction.

Chamberlain, S. D. (2003). Exterior Brick Masonry Walls: Causes of and Solutions to Inevitable Deterioration. Architectural Technology by Hoffmann Architects.

Dholakia R., Zhao M., Dholakia N. and Fortin D. (2000), Interactivity and Revisits to Websites: A Theoretical Framework, Research Institute for Telecommunications and Information Marketing

Gentry, T. R. (2013). Digital tools for masonry design and construction. Georgia Tech.

Gramazio & Kohle (2006). Gantenbein Vineyard Facade. Retrieved Mar 2014, from Gramazio & Kohler Architecture and Urbanism: http://www.gramaziokohler.com/web/e/projekte/52.html

Goedert, J.D., and Meadati, P. (2008). Integration of construction process documentation into Building Information Modeling, Journal of Construction Engineering and Management, July 2008.

Gu, Y., Grossman, R. (2005) Optimizing UDP-based protocol implementations, in: Proceedings of the Third International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2005), Lyon, France, 3-4 February.

Han C., Sun B., Ramamoorthi R., Grinspun E. (2007), Frequency Domain Normal Map Filtering

Islamic Art (2007). Materials and Mediums. Retrieved Mar 2013, from Patterns in Islamic Art: http://www.patterninislamicart.com/background-notes/materials-and-mediums/

Moravánszky, A. (2002). Das Pathos des Mauerwerks, in Detail Jg. 42. No. 1/2 (Jan.-Feb. 2002),. S. 28-38

Panahi, A. (2012). Application of geometry in brick decoration of Islamic architecture of Iran In Seljuk period. Journal of American Science, 814- 821.

Reas, C., Fry, B., Maeda, J. (2007). Processing: A Programming Handbook for Visual Designers and Artists (1st ed.), The MIT Press.

ROB Technologies (2012), BrickDesign getting started, Retrieved Aug 2013, from http://brickdesign.rob-technologies.com/wordpress/wp-content/uploads/downloads/2012/03/120316_BrickDesignGettingStarted.pdf

Saoud, R. (2003). Muslim Architecture under Seljuk Patronage. Foundation for Science, Technology and Civilization.

Sacks, R., Eastman, C., Lee, G. (2004). Parametric 3D modeling in building constrcution with examples from precast concrete. Automation in constrcution, 291-312.

Sloan P. (2006) Normal Mapping for Precomputed Radiance Transfer, Microsoft Corporation

Sullivan, C. C., & Horwitz-Bennett, B. (2008). Building with Bricks. BUILDING DESIGN+CONSTRUCTION, 48-57.

The Brick Directory (2012). The History of Bricks and Brickmaking, http://www.brickdirectory.co.uk/html/brick_history.html

The Brick Industry Association (2009). Dimensioning and Estimating Brick Masonry. Reston, Virginia.

The Brick Industry Association (1999). Technical notes on brick constrcution. Reston, Virginia.

Zwarts and Jansma Architects (2014). Parametric Design for Brick Surfaces, http://www.zwarts.jansma.nl/page/2103/en

8. APPENDIX

The Processing code that is written in version 2.0b is presented below. The functions are associated with the algorithms 1 to 7.

```
import hypermedia.net.*; //the UDP library
import javax.swing.*;  //for graphics of the file chooser
import java.io.File; //for file objects

UDP udps_outgoing;
UDP udpr_incoming1;
String sentMessage;// the message to send
String ip = "127.0.0.1"; // the remote IP address
int port_outgoing_remote = 9000; // the remote port to send to grasshopper
int port_outgoing_local = 6005; // the local port from which data is sent
int port_incoming1_local = 10000; // the local port to receive data from grasshopper

int cols = 40;//initial wallWidth
int rows = 40;//initial wallHeight
int Rx;
int Ry;

int brushRadius = 25;
int brushRadiusSquared = 625;
int brushStrength = 100;
PImage heightmap;
boolean bDrawBrushIndicator = true;

void setup(){
  size(600, 600);
  background(0);
  smooth();
  noStroke();
  frame.setResizable(true);

  udps_outgoing = new UDP( this, port_outgoing_local ); //sending from this port
  udpr_incoming1 = new UDP( this, port_incoming1_local ); //receiving data(wallWidth) to this port
  udpr_incoming1.listen( true );

  BuildMap();
  //for the circle
  noFill();
  stroke(255, 0, 0); //red stroke
}
void draw(){
  redraw();
}
void BuildMap(){
  heightmap = createImage(width, height, RGB);
  Rx = width/cols;
  Ry = height/rows;
  heightmap.loadPixels();
  for (int i=0; i<heightmap.pixels.length; i++)
  {
    heightmap.pixels[i] = 0;
  }
  heightmap.updatePixels();
  image(heightmap, 0, 0);
```

```
}
void SetHeightMapWithNewImage(PImage i_image){
 heightmap = createImage(i_image.width,i_image.height, RGB);
 heightmap.loadPixels();
 i_image.loadPixels();
 for (int i=0; i<heightmap.pixels.length; i++)
 {
  heightmap.pixels[i]=i_image.pixels[i];
 }
 i_image.updatePixels();
 heightmap.updatePixels();
 frame.setSize(heightmap.width, heightmap.height);
 Rx = heightmap.width/cols;
 Ry = heightmap.height/rows;
 image(heightmap, 0, 0);
}
void drawBrushIndicator(){
 ellipse(mouseX, mouseY, brushRadius, brushRadius);
}
void mousePressed(){
 drawBrush(heightmap);
 if (bDrawBrushIndicator)
  drawBrushIndicator();
}
void mouseDragged(){
 drawBrush(heightmap);
 if (bDrawBrushIndicator)
  drawBrushIndicator();
}
void mouseMoved(){
 if (bDrawBrushIndicator){
  image(heightmap, 0, 0);
  drawBrushIndicator();
 }
}
void increaseBrushRadius(){
 brushRadius+=1;
 brushRadiusSquared = brushRadius*brushRadius;
}
void decreaseBrushRadius(){
 brushRadius-=1;
 brushRadiusSquared = brushRadius*brushRadius;
}
//Receiving Wall size
void receive( byte[] data1, String ipR1, int portR1 ) {
 String dataString = new String( data1 );
 String[] dataStrings = split(dataString, "\n");
 print( "received: ");
 for (int i=0;i<dataStrings.length;i++)
 {
  print( dataStrings[i] +",");
 }
 println(" from "+ipR1+" on port "+portR1+"." );

 int[] dataInts = new int[dataStrings.length];
 for (int i=0;i<dataStrings.length;i++){
  dataInts[i]=(int)Float.parseFloat(dataStrings[i]);
```

```
    println(dataInts[i]);}
  if (dataInts.length>=2){ //check that there are at least 2 values to use
    boolean doUpdate = false;
    //check if the cols is different
    if (cols != dataInts[0]){
      doUpdate = true;
      cols = dataInts[0];
    }
    if (rows != dataInts[1])
    { //check if the rows is different
      doUpdate = true;
      rows = dataInts[1];
    }
    if (doUpdate)
    { //update the output sent to Grasshopper, only if the rows or cols were changed
      BuildMap();
    }
  }
}
void sendMapToGrasshopper()
{
  sentMessage="";
  int value;
  for (int i=0;i<cols;i++) {
    for (int j=rows-1;j>=0;j--) {
      heightmap.loadPixels();
      value = (heightmap.pixels[i*Rx+j*Ry*heightmap.width] >> 16) & 0xFF; //red (x gradient)

      sentMessage = sentMessage + (str(value)+";");
      udps_outgoing.send( sentMessage, ip, port_outgoing_remote );// sends the message
      heightmap.updatePixels();
    }
  }
}
void keyPressed()
{
  if (key=='+'||key=='=')
    increaseBrushRadius();
  if (key=='_'||key=='-')
    decreaseBrushRadius();
  if (key=='i'||key=='I')
    bDrawBrushIndicator=!bDrawBrushIndicator;
  if (key=='g'||key=='G')
    calcImageGradient(heightmap);
  if (key=='c'||key=='C')
    replicateBlueChannel(heightmap);
  if (key=='s'||key=='S') //for 'send'
    sendMapToGrasshopper();
  if (key == 'h'||key=='H')
    heightmap.save ("heightmap.png");
  if (key=='b'||key=='B')
    calcRGBlur(heightmap);
  if (key=='o'||key=='O') //for open
    openImageFile(heightmap);
}
void drawBrush(PImage canvas){
  canvas.loadPixels();
```

```
  int brushPosition = mouseX+mouseY*canvas.width;
  int newX, newY;
  int intensity;
  for (int xi=-brushRadius;xi<=brushRadius;xi++){
   for (int yj=-brushRadius;yj<=brushRadius;yj++){
    newX = mouseX+xi;
    newY = mouseY+yj;
    if (newX<canvas.width){
     if (newX>=0){
      if (newY<canvas.height){
       if (newY>=0){
        brushPosition=newX+(newY)*canvas.width;
        int distanceSquared = xi*xi+yj*yj;
        if (distanceSquared<brushRadiusSquared){
         intensity = 8-(255*distanceSquared/brushRadiusSquared)/32;  //1-x^2 with some factor
         int current = (canvas.pixels[brushPosition] >> 0) & 0xFF;
         //current+=intensity;
         if (mouseButton == LEFT)
          current = constrain(current+intensity, 0, 255);
         if (mouseButton == RIGHT)
          current = constrain(current-intensity, 0, 255);
         canvas.pixels[brushPosition]=
          (((canvas.pixels[brushPosition]>>16)&0xFF)<<16) |
          (((canvas.pixels[brushPosition]>>8)&0xFF)<<8) |
          current;
        }//if less than brush radius
       }
      }
     }
    }
   }
  }
  canvas.updatePixels();
  image(canvas, 0, 0);
}
void replicateBlueChannel(PImage canvas)
{
  canvas.loadPixels();
  for (int xi = 0; xi < canvas.width; xi++) {
   for (int yj = 0; yj < canvas.height; yj++ ) {
    int currentPosition = xi+yj*canvas.width;
    int b = (canvas.pixels[currentPosition]>>0)&0xFF; //get the blue value only
    canvas.pixels[currentPosition] = b<<16 | b<<8 | b<<0;
   }
  }
  canvas.updatePixels();
}
void calcImageGradient(PImage canvas)
{
  canvas.loadPixels();
  for (int xi = 1; xi < canvas.width-1; xi++) {
   for (int yj = 1; yj < canvas.height-1; yj++ ) {

    int r = 0;
    int g = 0;
    int currentPosition = xi+yj*canvas.width;
    int b = (canvas.pixels[currentPosition]>>0)&0xFF; //get the blue value only
```

```
    {
     int[][] neighbors = {
      {
       (canvas.pixels[xi-1+(yj+1)*canvas.width]>>0)&0xFF, (canvas.pixels[xi+(yj+1)*canvas.width]>>0)&0xFF,
        (canvas.pixels[xi+1+(yj+1)*canvas.width]>>0)&0xFF
      }
      ,
      {
       (canvas.pixels[xi-1+(yj+0)*canvas.width]>>0)&0xFF, (canvas.pixels[xi+(yj+0)*canvas.width]>>0)&0xFF,
        (canvas.pixels[xi+1+(yj+0)*canvas.width]>>0)&0xFF
      }
      ,
      {
       (canvas.pixels[xi-1+(yj-1)*canvas.width]>>0)&0xFF, (canvas.pixels[xi+(yj-1)*canvas.width]>>0)&0xFF, (canvas.pixels[xi+1+(yj-
        1)*canvas.width]>>0)&0xFF
      }
      ,
     };

     r = constrain(neighbors[0][0]-neighbors[0][2]+2*neighbors[1][0]-2*neighbors[1][2]+neighbors[2][0]-neighbors[2][2], 0, 255);
     g = constrain(-neighbors[0][0]-2*neighbors[0][1]-neighbors[0][2]+neighbors[2][0]+2*neighbors[2][1]+neighbors[2][2], 0, 255);
    }

    canvas.pixels[currentPosition] = r<<16 | g<<8 | b<<0;
   }
  }
  canvas.updatePixels();
}

void calcRGBlur(PImage canvas)
{
  PImage temp = createImage(canvas.width, canvas.height, RGB);
  temp.copy(canvas, 0, 0, canvas.width, canvas.height, 0, 0, canvas.width, canvas.height);
  temp.loadPixels();
  canvas.loadPixels();
  for (int xi = 1; xi < canvas.width-1; xi++) {
   for (int yj = 1; yj < canvas.height-1; yj++ ) {

    int r = 0;
    int g = 0;
    int currentPosition = xi+yj*canvas.width;
    int b = (canvas.pixels[currentPosition]>>0)&0xFF; //get the blue value only
     int[][] neighbors_r = {
      {
       (temp.pixels[xi-1+(yj+1)*canvas.width]>>16)&0xFF, (temp.pixels[xi+(yj+1)*canvas.width]>>16)&0xFF,
        (temp.pixels[xi+1+(yj+1)*canvas.width]>>16)&0xFF
      }
      ,
      {
       (temp.pixels[xi-1+(yj+0)*canvas.width]>>16)&0xFF, (temp.pixels[xi+(yj+0)*canvas.width]>>16)&0xFF,
        (temp.pixels[xi+1+(yj+0)*canvas.width]>>16)&0xFF
      }
      ,
      {
       (temp.pixels[xi-1+(yj-1)*canvas.width]>>16)&0xFF, (temp.pixels[xi+(yj-1)*canvas.width]>>16)&0xFF, (temp.pixels[xi+1+(yj-
        1)*canvas.width]>>16)&0xFF
```

```
        }
        ,
       };
       int[][] neighbors_g = {
        {
         (temp.pixels[xi-1+(yj+1)*canvas.width]>>8)&0xFF, (temp.pixels[xi+(yj+1)*canvas.width]>>8)&0xFF,
          (temp.pixels[xi+1+(yj+1)*canvas.width]>>8)&0xFF
        }
        ,
        {
         (temp.pixels[xi-1+(yj+0)*canvas.width]>>8)&0xFF, (temp.pixels[xi+(yj+0)*canvas.width]>>8)&0xFF,
          (temp.pixels[xi+1+(yj+0)*canvas.width]>>8)&0xFF
        }
        ,
        {
         (temp.pixels[xi-1+(yj-1)*canvas.width]>>8)&0xFF, (temp.pixels[xi+(yj-1)*canvas.width]>>8)&0xFF, (temp.pixels[xi+1+(yj-
          1)*canvas.width]>>8)&0xFF
        }
        ,
       };
       r = constrain(
          (neighbors_r[0][0]+neighbors_r[0][1]+neighbors_r[0][2]+neighbors_r[1][0]+neighbors_r[1][1]+neighbors_r[1][2]+neighbors_r[2][
          0]+neighbors_r[2][1]+neighbors_r[2][2])/9, 0, 255);
       g = constrain(
          (neighbors_g[0][0]+neighbors_g[0][1]+neighbors_g[0][2]+neighbors_g[1][0]+neighbors_g[1][1]+neighbors_g[1][2]+neighbors_g[
          2][0]+neighbors_g[2][1]+neighbors_g[2][2])/9, 0, 255);
     }
     canvas.pixels[currentPosition] = r<<16 | g<<8 | b<<0;
    }
   }
   canvas.updatePixels();
   temp.updatePixels();
}
void openImageFile(PImage canvas){
  try {UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());}
  catch (Exception e) {e.printStackTrace();}
  final JFileChooser fc = new JFileChooser();
  int returnVal = fc.showOpenDialog(this);
  if (returnVal == JFileChooser.APPROVE_OPTION){
    File file = fc.getSelectedFile();
    if (file.getName().endsWith("jpg")){ // check if the file is an image
      PImage temp = loadImage(file.getPath());
      temp.filter(GRAY); //change the greyscale
      if (temp != null){
        SetHeightMapWithNewImage(temp);
      }
      else{println("maybe this file is not an image");}
    }
    else{ // just print the contents to the console
      String lines[] = loadStrings(file);
      for (int i = 0; i < lines.length; i++) {println(lines[i]);}
    }
  }
  else{
    println("Image open command cancelled by user.");
  }
}
```