CEIG - Spanish Computer Graphics Conference (2016)
Alejandro García-Alonso and Belen Masia (Editors)

Tools for Structural Analysis and Optimization of Procedural Masonry Buildings

Josep Lluis Fita, Gonzalo Besuievsky and Gustavo Patow

ViRVIG, University of Girona, Spain

Abstract

We present a set of off-the-shelf tools that will enable structural simulations and optimization into procedural modeled masonry buildings, as historical buildings like cathedrals or churches. For instance, with our tools we are capable of easily knowing whether the roof of a given masonry structure is sound, if it falls down, and, in the later case, even which brick of this structure has moved. For this we integrate a set of custom tools into the available Houdini platform [Sid12], together with the freely available Bullet *engine and a set of* Python *scripts to quickly and efficiently simulate masonry structures.*

Categories and Subject Descriptors (according to ACM CCS): Procedural Modeling [Computer Graphics]: Virtual Historical Buildings—Masonry Structures

1. Introduction

Modeling urban environments is becoming increasingly popular in computer graphics research for applications like urban planning, video-games or GPS-based navigation tools, to name just a few. However, generating a convincing urban environment or building requires considerable manual work, unless more advanced techniques are used by designers such as geometric modeling and procedural modeling.

However, they lack one important feature for the description of general buildings and masonry structures in particular: their lack of any structural analysis. So, in recent years, some researchers have found some ways to explore methodologies that combine visual results with structural analysis, especially on masonry structures. However, all of them rely on custom approaches, which require a considerable coding effort as a structural analysis implementation requires quite involved calculations.

For this reason we have focused our analysis on this kind of methodologies, where the aim is to know how these techniques influence in the creation of a realistic virtual environment. Thus, this paper presents a methodology for using off-the-shelf tools for the structural analysis of masonry buildings, like ancient masonry structures and buildings, especially in those buildings built on the Romanesque and Gothic period, such as churches or cathedrals. We also present the algorithms and scripts needed to simulate such an ancient structure. Specifically, our main contributions are

- the introduction of a pipeline for the structural analysis of masonry buildings, allowing any interested person to perform such analysis without complex implementation efforts and without requiring costly acquisitions.
- our pipeline is completely based on off-the-shelf, freely available tools
- combined with an optimization algorithm, it allows to find valid values for constructive parameters of a masonry building.

2. Previous Work

In this section, we describe those procedural modeling techniques that are suitable for modeling ancient buildings of archaeological value, mainly focusing on the structural masonry buildings. We also review the most popular tools used for content creation.

For procedural modeling, Muller et al. [MWH*06] presented an initial proposal intended for building generation based on the concept of shape grammars. Starting from an initial axiom, these grammars iteratively replaced labeled geometry parts by new parts resulting after the execution of a *rule* specifying the basic operation to be applied. This initial approach was later extended by many authors [KK12, MWA*12], but in almost all cases only considering the shape and not the building structure.

Whiting and co-authors [WOD09] introduced one of the first works to describe a method based on masonry structures such as cathedrals, stone bridges and churches for procedural modeling. Their method allows the user or designer to create a set of rules for the creation of an ancient building, through the introduction of user-defined parameters. Once the user has created the set of rules, the method automatically searches a stable configuration for the building shape according to physical constraints with the resolution of inverse static problems through quadratic equations of equilibrium. Later on, [WSW*12] presented an extension, introducing a new set of methods that integrate the study of a building soundness

DOI: 10.2312/ceig.20161322

through the integration of architecture design and structural analysis. Once the geometry is loaded, the aim of the method is to create a new stable structure through the analysis of the stability gradient, according to constraints previously introduced by the user, such as horizontal direction, vertical direction and block thickness.

Panozzo and co-workers [PBSH13], introduced an algorithm that automatically generates a 3D masonry structure from an input shape (e.g., a NURBS surface) in such a way that it is a self-supporting structure without mortar, and its shape is as close as possible to the one given as input, but constructive with masonry blocks. More recently, Deuss and colleagues [DPW*14], based on the work by Panozzo et al., introduced a new algorithm that processes all kind of generated masonry models. Their method is based on the detection of structures called quasi-arches, which represent subsets of the original input masonry structure that can be built independently of any other part of the same roof. Once the roof is partitioned in these subsets, the algorithm provides detailed instructions to build these through blocks, holding them with hooks and chains. The final step is to fill the empty regions among the quasi-arches with blocks.

Given the above mentioned arguments, we have decided to take one masonry structure generated by [PBSH13], and tested it on a virtual environment using an off-the-shelf numerical physics simulator, with the aim of improving the knowledge about this type of virtual shapes and the performance of our algorithm in their simulation with standard and freely available tools.

3. The Algorithm

This section presents the tools we developed for the calculation of the configuration of the masonry structures necessary for supporting the weight of a given vault. In Figure 1 we can see the schema of our algorithm.

3.1. Modeling masonry buildings

As we can see on Figure 1, we start our procedure with the creation of a masonry building structure, like a cathedral or church. In our case, we have modeled it with procedural modeling techniques, specially the walls, and introduced configurable parameters to allow the optimization process described next. On the other hand, the roof has been imported from the publicly available results provided by Panozzo et al. [PBSH13], and added to our masonry structure. See Figure 2.

3.2. Parameter variations

In our algorithm we distinguish between two kind of parameters: intrinsic and constructive parameters. The first ones refer to all parameters related to the simulation and the numerical method used. These include the density for the bricks, $2691 Kg/m^3$; friction, set to 0.7 corresponding to a non-polished granite surface; mortar strength, called "*glue*" in our system and set to 10 which represents a small value, as masonry buildings barely used any mortar to hold the bricks together because of the precision used to cut the stones; and rotational stiffness, which represents the resistance of the bricks to rotate in place, and that was given a value of $7000 Nm/rad$. In general, intrinsic parameters are set as constraints, being left outside the optimization process, but there is no reason they could be included, if wanted, in the list of variables to find a value for. The constructive parameters refer to any geometric parameter, like height or thickness of the walls, the brick sizes, the size of the windows, and actually any parameter that could be defined in the input geometric parameterized model (in our case, created with procedural techniques) [WOD09].

Our algorithm starts by letting the user select a few constructive parameters for some specified parts of the building (e.g., the wall thickness or the number of buttresses), previously configuring a maximum and minimum threshold for each one. In case of boolean parameters (e.g., existence or absence of certain elements), the system tries without these elements, and if a solution is not found, then it tries again with the opposite situation. All numerical parameters are then initialized to their maximum values. Then, in an iterative brute-force procedure, the algorithm varies the selected parameter determining at each step whether the structure falls down during the simulation or not, see Section 3.3. In case the structure works correctly and stays in place, our algorithm decreases the selected parameters, always checking they have not reached the lower threshold limitation. After this optimization step, the algorithm simply outputs the results and provides analytic of the performance of the optimized resulting building. Please, refer to the scheme in Figure 1. It is important to state that this brute-force optimization could be replaced by any suitable optimization method, from gradient descent methods to stimulated annealing [PTVF07].

3.3. Structural Stability

At first, our algorithm takes the initial vertical position (the y-axis in our implementation) of the bounding box of the entire vault, and saves this initial value. At each iteration of the optimization process already described, a physical simulation is performed using the standard *Bullet* library. When this simulation has finished, the algorithm takes again the vertical position (Y coordinate) of the vault bounding box center, and analyses this with Equation 1.

$$(InitialRoofPosition - FinalRoofPosition) < Threshold \quad (1)$$

If the result of the difference between *InitialRoofPosition* and *FinalRoofPosition* is greater than the threshold, our algorithm understands that the vault is *not* supported by the walls. If it does, the algorithm repeats the same operation, but this time repeating the computation for each brick, which has a higher computational cost, and verifies the movement of the bricks with Equation 2.

$$\underset{allBricks}{Max}\,(InitialBrickPosition_i - FinalBrickPosition_i) < Threshold \quad (2)$$

Where *InitialRoofBrickPosition* and *FinalRoofBrickPosition* are respectively the vertical positions of the bounding box centers of the initial and final position of a given brick. If they are greater than the user-defined threshold, then the algorithm assumes that the brick has moved along the vertical axis.

It is important to notice that, even if the vault (or the entire building) may hold its global position, there could be locality instabilities, resulting in cracks in the structure, which in our system are detected by the movement at the brick level. So, actually, our sys-

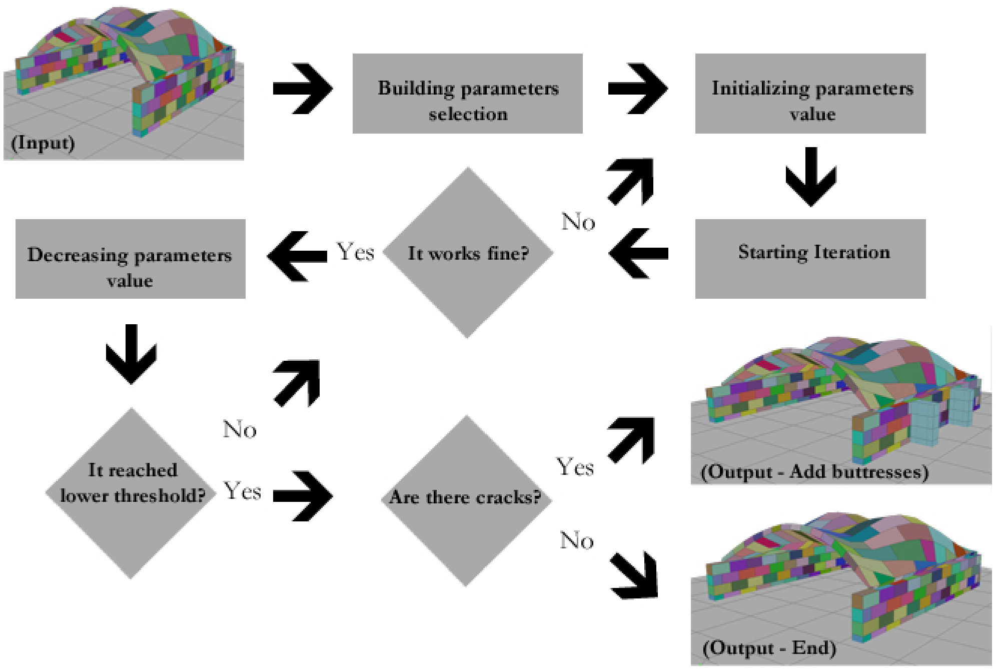
(Input)
Building parameters selection
Initializing parameters value
Starting Iteration
No
It works fine?
Yes
Decreasing parameters value
No
It reached lower threshold?
Yes
Are there cracks?
Yes
(Output - Add buttresses)
No
(Output - End)

Figure 1: *Scheme of our pipeline.*

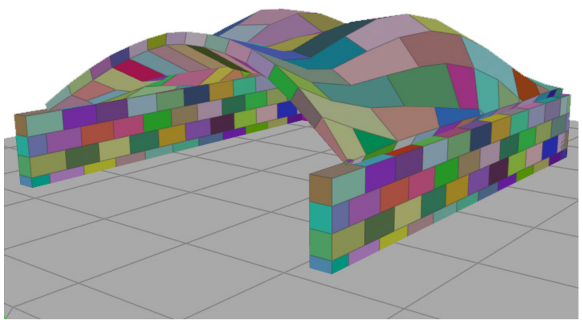

Figure 2: *The masonry structure with the roof taken from [PBSH13].*

tem is not only capable of detecting severe structural instabilities, but also to detect small movements.

4. Application Example: Walls and buttress

As we said above, our system can easily accommodate a whole range of stability analysis, from global one to a brick-level assessment. This, of course, may include some external stresses that might threaten the whole building stability. As we compute displacements at the brick level, we traverse the walls or the roofs by visiting all bricks in order to detect some cracks. In case that the detection is positive, the algorithm automatically adds buttresses and automatically calculates the most suitable number of buttresses for each wall, in a similar way as done before.

The first thing that the algorithm does is to clean any additional structure (e.g., buttresses) that might have been created previously. Then, it starts by accommodating one buttress at each wall, and iteratively proceeds by increasing the number of buttresses. The calculation of the buttresses positions follows the expression in Equation 3.

$$InitialPosition = (SpaceAmongButtresses + WallPosition) \tag{3}$$

Where *SpaceAmongButtresses* and *WallPosition*, are constants that our algorithm calculates from the longitudes of the walls through the expresion in Equation 4.

$$LongitudeRow = ((NumberOfPoints \times SizeOfBrick) + HalfBrick) \tag{4}$$

Where the parameter *NumberOfPoints* refers to the number of bricks in a wall row. The *SizeOfBrick* refers to the brick longitude, taking into account that each row of bricks has, at the end, a *HalfBrick*. The last parameter refers to the size of a half brick, which usually is computed as $SizeOfBrick/2$, but can differ for aesthetic reasons in a real building.

Once the algorithm has obtained the *LongitudeRow* parameter, then it calculates the distance among buttresses through the following equation:

$$DistanceAmongButtresses = \frac{LongitudeRow}{(\sum NumberOfButtresses) + 1} \tag{5}$$

Our algorithm finally repeats the procedure for computing the

InitialPosition (See Equation 3) used for the first buttress, repeating it for each wall. It uses the user-defined constant *DistanceAmongButtress* in order to obtain the initial situation for the next buttresses, in a way such that the set of buttresses are centered with respect to the wall endings. To create the buttresses the algorithm follows the same procedure as the one explained for the creation of the walls, with the difference that it uses necessary parameters such as *Orientation* are taken automatically (i.e., perpendicular) from the respective wall. The algorithm keeps track of the overall constructive parameters for the stability of the building, taking default values for the intrinsic parameters like the ones to setup the global structural simulation, the stone density or the strength of the mortar needed between the bricks.

Our algorithm iterates over the buttresses thickness the same way as before, detecting failures and cracks in their structure. If the result is positive for each wall, it repeats automatically the steps described above until the cracks have been eliminated. As a result the outcome can be that some walls require a smaller number of buttresses than others for their stabilization. See Figure 3.

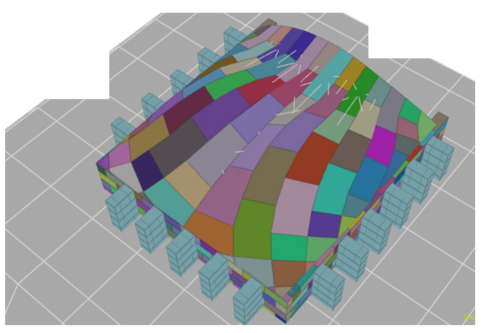

Figure 3: *The walls with their respective buttress.*

4.1. Results

In this section we discuss about the results that we have obtained with our prototype implementation. For our implementation, we have chosen freely available commercial software *Houdini* from SideFX [Sid12], the open source *Bullet* solver (incorporated by default in Houdini, along with other solvers) and *Python* scripts.

We have designed two kind of tests for our algorithm. For the first one, we tested the structure with a small value of stiffness for the building bricks (around $7000 Nm/rad$) and three different minimum thickness thresholds for the walls, *Maximum* (2m), *Medium* (1m) and *Minimum* (0.5m). This resulted in the addition of 6 buttresses in all walls except for the back one, which required 7 buttresses. For the second, we tested the structure with the same features but, in this case, with a larger value of the stiffness between the bricks (this time we used the value of $70000 Nm/rad$). We observed that the results depend heavily on the stiffness values and the amount of mortar (i.e., glue) added between the bricks. For a small value of the stiffness, the building is stable only for the thickest configuration, but for a large value of the stiffness the configuration is stable even for the thinnest wall, without requiring the introduction of additional supporting structures like buttresses. The reason is that, with the *Minimum Friction* value for the vault, the *Rotational Stiffness* has to provide the stabilization setting for the structure. If both parameters are set to low values (around $7000 Nm/rad$), the structure is unstable unless buttresses are added. In this case the maximum tolerable net force on the walls is smaller than the *Maximum Friction* value for the vault, resulting in an overall stability setting. See figure 4.

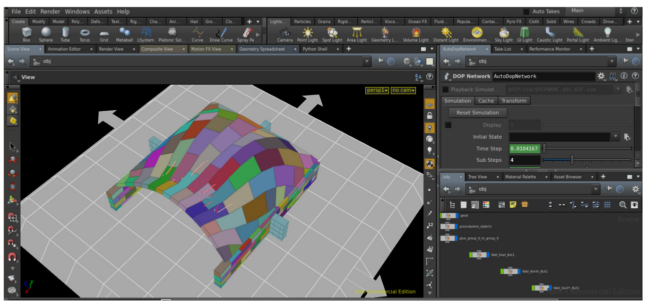

Figure 4: *User interface of our application.*

With respect to timing, to perform a complete optimization our unoptimized prototype needs around 10 minutes to find the optimal values, even using our brute-force optimization method. Thus, it is expected that, with a more appropriate optimization algorithm (e.g., conjugate gradients) and with an optimized implementations, this time could be much reduced, in our experience up to the order of a few seconds. This would make this simulation feasible for near-interactive editing operations.

5. Conclusions

After an initial review of the state-of-the-art literature on techniques involving soundness analysis, we found that only a handful of papers deal with this complex, but crucial aspect of modeling, all of them relying on complex custom implementations. Given the importance of this kind of analysis in architecture, and the increasing use of procedural techniques, we decided to develop a pipeline based on off-the-shelf tools for the structural analysis of masonry buildings. As part of our ongoing experimental study, we have tested the algorithm over masonry structures that combine the main elements found in a typical ancient masonry building: walls, buttresses and a vault. To test our pipeline with different settings, we created procedural walls and buttresses, but used a pre-computed roof with proven structural stability [PBSH13], and played with its constituent parameters. We have learned about the complex interplay of the different parameters affecting the simulation, in particular through the relationship among wall thickness, friction and rotational stiffness among the bricks; and structural elements like buttresses. This is important not only for entertainment purposes, but also for art historians, curators, conservators and other specialists related to ancient masonry buildings. As future work we plan to go a step forward and use our tools to simulate the complex construction process of a masonry building like an ancient cathedral.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. This work was partially funded by the TIN2014-52211-C2-2-R project from Ministerio de Economía y Competitividad, Spain.

References

[DPW*14] Deuss M., Panozzo D., Whiting E., Liu Y., Block P., Sorkine-Hornung O., Pauly M.: Assembling self-supporting structures. *ACM Trans. Graph. 33*, 6 (Nov. 2014), 214:1–214:10. URL: http://doi.acm.org/10.1145/2661229.2661266, doi:10.1145/2661229.2661266. 2

[KK12] Krecklau L., Kobbelt L.: Interactive modeling by procedural high-level primitives. *Computers & Graphics*, 0 (2012), –. 1

[MWA*12] Musialski P., Wonka P., Aliaga D. G., Wimmer M., van Gool L., Purgathofer W.: A survey of urban reconstruction. In *EUROGRAPHICS 2012 State of the Art Reports* (May 2012), EG STARs, Eurographics Association, pp. 1–28. URL: http://www.cg.tuwien.ac.at/research/publications/2012/musialski-2012-sur/. 1

[MWH*06] Müller P., Wonka P., Haegler S., Ulmer A., Van Gool L.: Procedural modeling of buildings. *ACM Trans. Graph. 25*, 3 (2006), 614–623. 1

[PBSH13] Panozzo D., Block P., Sorkine-Hornung O.: Designing unreinforced masonry models. *ACM Trans. Graph. 32*, 4 (July 2013), 91:1–91:12. URL: http://doi.acm.org/10.1145/2461912.2461958, doi:10.1145/2461912.2461958. 2, 3, 4

[PTVF07] Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P.: *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3 ed. Cambridge University Press, New York, NY, USA, 2007. 2

[Sid12] SideFX: Houdini 12, 2012. http://www.sidefx.com. 1, 4

[WOD09] Whiting E., Ochsendorf J., Durand F.: Procedural modeling of structurally-sound masonry buildings. *ACM Trans. Graph. 28* (December 2009), 112:1–112:9. 1, 2

[WSW*12] Whiting E., Shin H., Wang R., Ochsendorf J., Durand F.: Structural optimization of 3d masonry buildings. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 159:1–159:11. 1