3D Regular Expressions - Searching Shapes IN Meshes

The development of an algorithm to identify recurring geometries

Gabriel Wurzer[1], Bob Martens[2], Katja Bühler[3]
[1,2]Vienna University of Technology, Austria, [3]VRVis Forschungs GmbH, Austria
[1,2]www.tuwien.ac.at, [3]www.vrvis.at
[1]gabriel.wurzer@tuwien.ac.at, [2]bob.martens@tuwien.ac.at, [3]buehler@vrvis.at

Abstract. *We have grown accustomed to performing elaborate queries on textual data, e.g. via online search engines, file system managers and word processors. In the past decade, retrieval methods that also work on non-textual data have become mainstream (e.g. face recognition software). Sadly, these developments have so far not caught on for data mining within geometrical data, e.g. 3D meshes generated in the course of architectural work. Specifically during data exchange, such a search functionality would be handy, as it often happens that geometry is exported but object identity is lost - think, for example, of generative geometry or exported BIM data. In this paper, we present an example of such a search functionality, based on angular search. Our method is inspired by regular expressions, a string matching technique commonly employed for matching substrings.*
Keywords. *Shape retrieval; angular search; sub-mesh; regular expressions.*

NAÏVE IDEA

Instead of an introduction, let us jump directly to the idea behind the search algorithm and see how it can be applied within the architectural workflow: Assume that we have imported a large mesh into a modeling environment, in which all information but the list of vertices and faces is lost. Such a situation can occur during data exchange, entailing two **major problems**:

1. there is *no object identity*, i.e. we have to manually select vertices and faces belonging to an object in order to work with it. This can be a challenging task, though, as geometry might overlap (see Figure 1a).
2. In cases where there is *more than one instance of the same geometry*, a manual approach is highly tedious. Furthermore, the modeling environment has to load identical geometry multiple times into memory, which may prohibit working smoothly with the mesh for lack of performance. What is needed is an approach that can replace instances of the same geometry by a reference to a single one.

Our **contribution** concentrates on solving the mentioned problems and additionally brings forward a "search and replace" functionality for 3D meshes. In more detail, we present an algorithm that

- can *find shapes IN meshes (i.e. sub-meshes)*, given a search pattern in the way of a set of paths (which we interpret as succession of angles);
- can *restore object identity,* thus making it possible to work with a possibly inaccessible collection of vertices and faces;

- can *replace* found geometry *by a reference to a single geometry container;*
- can *replace* found geometry *by a different geometry.*

Figure 1 gives two results obtained with the approach: In Figure 1a, object identity was restored from a previously inaccessible polygon soup. In Figure 2b, we have searched for the given outline and replaced each occurrence by a different geometry. The latter takes 19s on a 2.4GHz single-core processor (C++ implementation, mesh containing 12064 vertices), which is moderately fast. The pattern is given as own mesh, which is automatically compiled into a search description which our algorithm needs.

In the coming sections, we will describe exactly how the search is done and how the mentioned compilation proceeds (see "Background" and "Elaboration"). We further provide details on the studies conducted (see "Studies"), which have served as test-bed and ground for discussion concerning the future development of the tool. Before concluding, we also deliver details on the two implementations existing so far (see "Implementation").

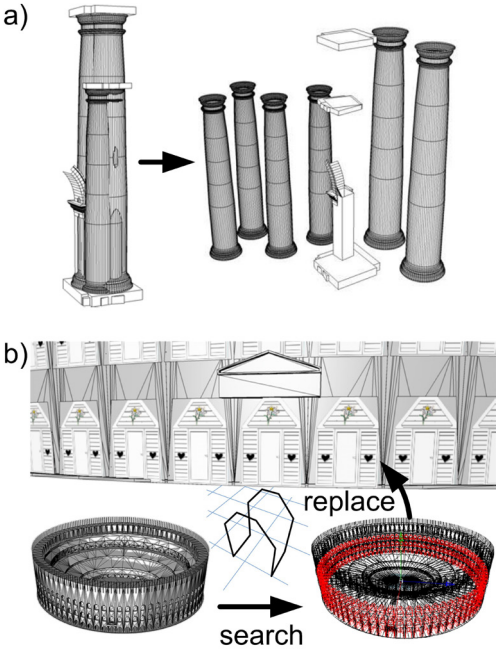

Figure 1
Searching and replacing in meshes. (a) Restoring object identity from a polygon soup. (b) Searching and replacing geometry within a mesh.

BACKGROUND

The approach is based on two underlying methods, *regular expressions* and *angular search*. Therefore, we first take a quick look at both, before elaborating the details of the presented method.

Regular expressions

A regular expression (regex) is a textual search technique that specifies a searched string by supplying a grammar of characters to match. We do not intend to give an elaborate introduction into this topic here, as this information is widely available and considered a standard technique in computer science. We instead forward the interested reader to (Forta 2004) and focus further explanations on the constructs that the search algorithm uses (also see Table 1):

- A string is a sequence of characters, a mesh a set of vertices connected by edges. We search for paths within that mesh, taking the sequence of angles between each pair of edges on that path as criterium.
- Meshes can be tesselated, meaning that each edge can be subdivided. The intermediate points do not contain significant information when we consider only angles as matching criterion. We thus adopt the regex repetition, in order to "jump over" points that lie in the same direction as previously encountered ones.
- The specified angular paths can self-intersect. In order to check that the same point is reached, we adopt the notion of back-references, i.e. the storage of a point that was reached so far for later equality comparison.

Throughout the paper, we will use a couple of termini found in regular expressions. To begin with, we use the word *automaton* to signify a list of matching criteria (*transitions*) that are evaluated sequentially. As example, take the following regular expression "ab". This specifies two transitions "a" and

Table 1
Regular expressions versus 3D regular expression.

Regex construct	matches e.g.	becomes in 3D regex algorithm
character sequence, e.g. abc	abc	angle sequence, e.g. 90° 10°
repetition (one or more times), e.g. a+	a, aa, aaa	match vertices in same direction
backreferences, e.g. (a|b)\1	aa,bb	match previously encountered point

Table 2
Transition types used in the automaton.

Transition	long name	meaning
FAIL	Failure Transition	ends execution, reporting failure
MATCH	Match Transition	ends execution, reporting a submesh
ANG	Angle Transition	at current point, find edge pairs having angle
CLO	Closure	find points in the same direction
BOR	Begin of Regex Group	begins a new regex group (for backreferencing)
EOR	End of Regex Group	ends a regex group, storing the curring point
REF	Backreference	references a previously matched point or point at a certain percentage of a visited edge
BAT	Begin-At	begins matching at a previously matched point

"b" which are put into a list ("a" "b"). Implicitly, two more transitions are added to the head and the tail of the list, namely MATCH and FAIL. Both signify a stop condition - in the first case, the algorithm has found the supplied string, in the second case, the algorithm has failed. The list of transitions thus becomes (FAIL, "a", "b", MATCH). An automaton has a *transition pointer*, placed initially on the *second* item of the list ("a"). Each transition type has its own way of matching. In the simple case mentioned, we have a character transition, which compares the current character in the string to the one specified. If both are equal, the transition pointer is advanced ("b"). This process is repeated until we finally encounter MATCH. In the case that the criteria specified in the current transition are not satisfied, an error flag is raised and the transition pointer is set to the preceding transition.

Angular search

Angular search is concerned with finding a sequence of angles in a given geometry. Examples of such algorithms are to be found in the automotive industry, in the form of a search tool for mechanical parts in a large CAD database (Berchtold and Kriegel 2004). However, in the concrete case mentioned, only section plans were taken into account, and the whole algorithm was limited to 2D retrieval. In contrast, our algorithm searches in 2D or 3D and additionally takes proportions into account (i.e. surplus to the angular search). Examples of other shape retrieval techniques, which use statistical data instead of angles, are the ShapeSifter tool that is based upon features such as surface area, volume etc. (Sung, Rea, Corney, Clark and Pritchard 2002) and the Princeton Shape Search Engine which can compare sketches to sections (Funkhouser, Min and Kazhdan 2003).

ELABORATION

Our search technique describes an angular path the transition types given in Table 2. The most important one is the angle transition (ANG), which tries to find an edge pairs having a given angle, extending from the current point. This is usually followed by a closure transition (CLO), which jumps over any intermediate points lying at the same angle (as mentioned, these do not contain significant information). As further transition types, we have begin and end of a regular expression group (BOR, EOR), backreference (REF) and begin-at (BAT). These are described in due course, using examples to help understanding. As in regular expressions, we also have FAIL and MATCH; the latter reports the points encountered during the whole matching process, i.e. the sub-mesh found.

We will now walk through the different possibilities for implementing a regular expression au-

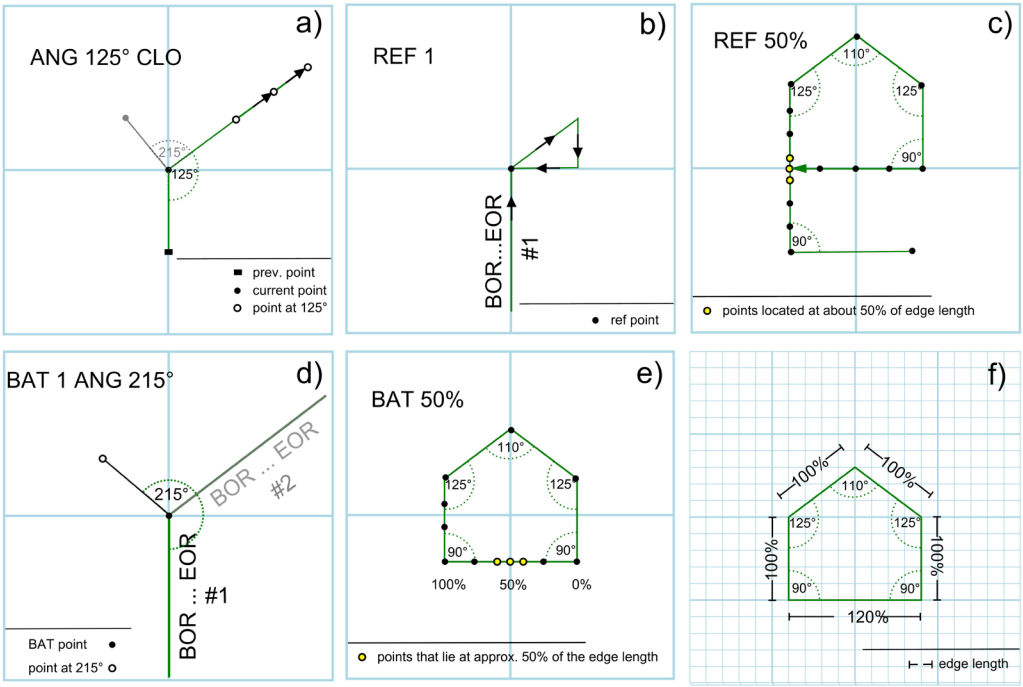

Figure 2
Transitions. (a) Angle and closure (b) backreference to point and (c) to edge, (d) branching at a point using the Begin-At transition, (e) branching 50% of an edge, counting from its start. (f) Relative edge lengths used to introduce proportions.

tomaton, starting with the 2D case and extending this to 3D. We also give details on the used compiler, which converts a search pattern (a mesh consisting of paths) into an automaton.

The 2D regex automaton

Two different modes are considered in the 2D case: If this is the first ANG to be matched, then an *inner angle* (0...360°) between edge pairs situated at the current point is found. In all other cases, the edge last taken is already fixed: The automaton then searches for a next edge at the correct angle with reference to the previous one. Depending on required strictness, angles are compared using a tolerance interval. The same applies to the matching of intermediate points. In the case no suitable angle is found, the algorithm gets to the previous transition and tries the next edge pair.

Figure 2 brings examples of such a 2D search: In Figure 2a, we search for an angle of 125 degrees formed by the current point (shown in the center), the previous point (shown as a tiny rectangle) and a possible next point (shown as circle). Two cases are distinguished: (Case 1) If there is yet no previous point (because we have just started), we try the combination of all neighbor points *twice* (neighbor 1 - current point - neighbor 2; neighbor 2 - current point - neighbor 1), since that establishes the marching order of the algorithm. (Case 2) In case that there is a previous point, as shown, the algorithm tries to continue along a non-visited neighbor which has the correct inner angle. Regardless of which of both cases the algorithm has dealt with, the marching direction has been fixed (shown by an arrow). The next transition, a closure, consumes all points of the mesh lying in that direction, which allows us to skip past points that contain no significant angular information.

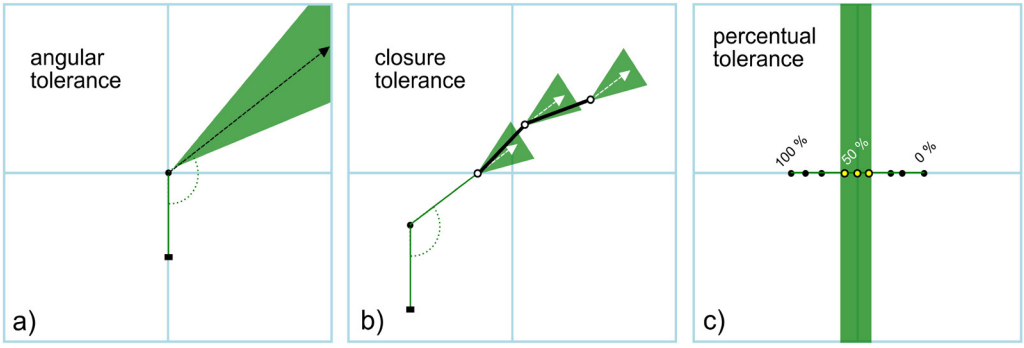

Figure 3
Tolerance values. (a) Angular tolerance at points, (b) closure tolerance for marching forward, (c) percentual tolerance for matching edges.

In Figure 2b, a backreference (REF) is shown where the path self-intersects. Two steps are needed to make such checks for self-intersection possible: The preceding transitions are enclosed in a begin-of-regex (BOR) and end-of-regex (EOR) group, in this case: the first regex group (#1). Internally, the automaton stores all points encountered when matching that group. The backreference REF 1 checks whether the current point corresponds to the last point encountered in regex group #1.

A variation of a backreference is given in Figure 2c: Here, REF 50% checks whether the current point corresponds to one of the matched points of a regex group situated within a certain percentage of the edge length. Thus, this specific flavor of REF checks for self-intersections with an edge.

Figure 2d shows a fork: The regex has so far matched regex group #1, arriving at the point shown in the center. It is now possible to begin at that point when matching, using the begin-at transition at group 1 (BAT 1). In the example shown, one might BAT 1 for matching the right path, before executing BAT 1 to find the left path.

One may also begin matching at a certain percentage of a previously encountered edge, as shown in Figure 2e: From the matched points of the given regex group, the algorithm selects the ones lying within a tolerance interval around the given percentage (BAT 50%) and tries to match onward from these.

Introducing proportions

So far, the regex algorithm is length-invariant, as it considers only angles. However, *relative length* does make a difference when looking at *proportions*: A square is not the same as a rectangle, for example. Thus, we introduce length as shown in Figure 2f: Initially, we memorize edge lengths relative to the length first edge within the search pattern (*reference length*). During matching, we can reject points if they do not lie at a certain distance of an edge start, in the following manner: The edge start is given as an ANG transition, the edge itself is matched via a following CLO. The next ANG represents the edge end (and, at the same time, the start of a new edge). When trying to match the latter angle, one would usually start at the last point matched by CLO and then go back point by point until we have either found the angle or there are none left. However, because the relative edge length is known, we can consider only points established by CLO that are situated at a certain distance of the edge length (expressed in percent of the *reference length*). This minor modification is all that is needed to include proportions.

Tolerance intervals

A point yet unaccounted for are the tolerance values which govern the strictness of the search algorithm. We have three such intervals, as given in Figure 3:

- Angular tolerance (Figure 3a) defines what deviations from a prescribed angle is acceptable.

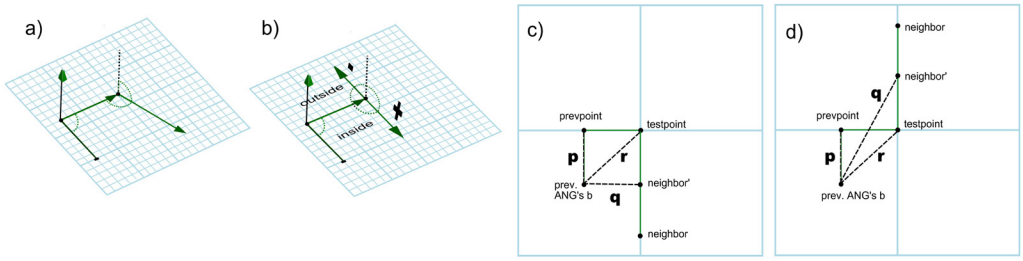

Figure 4
3D regex. We need an additional angle at each next point, (a) the angle between the previous normal and the next leg. (b) There is an ambiguity for successive 90° angles. Through comparison of lengths, we can find out whether we have an (c) inside or (d) outside winding.

- Closure tolerance (Figure 3b) specifies what deviations from the marching direction are to be accepted.
- Percentual tolerance (Figure 3c) states the interval around a percentage of an edge's length.

These three values are specified globally, for the time being. However, results obtained with the help of some basic test cases (see "Studies") show that this is a possible weakness of our algorithm, as we cannot easily adapt to sub-meshes that contain parts which require a more fine-grained, localized notion of tolerance. Thus, this part is likely to be extended in future implementations.

The 3D regex automaton

For the 3D case, the 2D regex algorithm is extended. In order to fix a next edge e_{i+1}, we need to take the last two edges e_i and e_{i-1} into account (refer to Figure 4a): The cross product e_i x e_{i-1} gives the normal vector n of the plane in which the last two edges lie. A suitable next edge is one that (1.) has the correct angle between e_i and e_{i+1} (same as in the 2D case) and additionally (2.) has the correct angle between n and e_{i+1}. Because of this, we need at least four points in the mesh to be searched.

An ambiguity arises for cases in which there is an ANG 90° following an ANG 90° (see Figure 4b), since it is not clear whether to march left ("outside") or right ("inside"). This case can be resolved through projection: Let (prev. ANG's b, prevpoint, testpoint, neighbor) be successive points, neighbor being the candidate for marching onward. Then, we have to fix three lengths p, q and r, as follows (refer to Figures 4c and d): p is the length (prev. ANG's b, prevpoint). An intermediate point neighbor' is a point situated at length p on the edge (testpoint, neighbor). The distance (prev. ANG's b, neighbor') is defined to be q. r is the length (prev. ANG's b, testpoint). If q is smaller than or equal to r, we can conclude that we have an "inside" winding (Figure 4c). In all other cases, we have an "outside" winding (Figure 4d). The winding criterion is added to the transition specification and compared at runtime with the mesh, for cases in which successions of 90° angles are present.

The regex compiler

The regular expression description is translated automatically from a search pattern made of paths (ordered edges) into a regex automaton. Briefly outlining the algorithm, we sequentially translate each edge pair into an ANG CLO (first pass). In that process, we ignore co-linear edges. However, these are needed later for checking intersections (second pass): (a) In case the end vertex of the forward edge was already visited, we generate a REF after ANG CLO. There are two distinct cases: If the visited vertex was co-linear (i.e. it was ignored during the first pass), we generate an edge reference (REF %). In all other cases, we surround the edge that leads to the point with BOR..EOR and generate a backreference to that regex group (REF #). (b) In case the start vertex of the forward edge was visited, we generate a BAT before ANG CLO in the previous fashion (co-linear: BAT %, else BOR…EOR BAT #).

STUDIES

Under this section, we examine the studies conducted with the regex algorithm in some detail. In

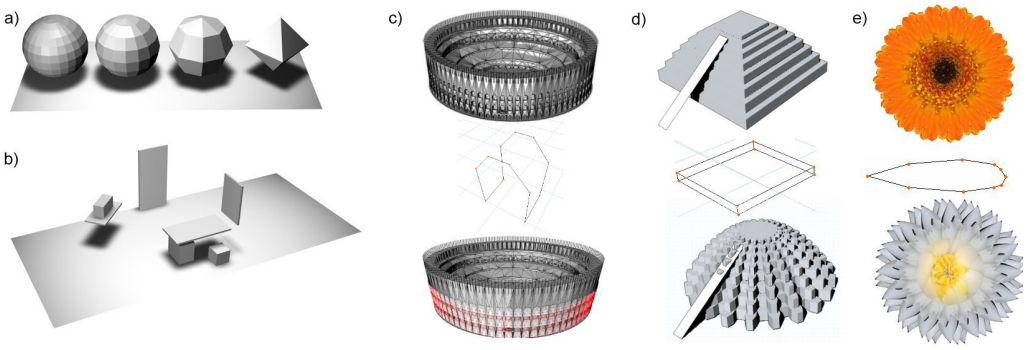

Figure 5
Basic Test Cases. (a) Sphere Matching (b) Proportion Check. Replacement of (c) Openings (d) Pyramid steps (e) Leafs.

all cases, we have applied a single regex pattern (acting as input) to a scene (also an input), producing an output in the form of a set of selected vertices and edges of the found sub-meshes. During evaluation, the number and type of matched sub-meshes (not all were "correct" in a visual sense, even though the angles and proportions matched) were compared to the type of regex used (ranging from "closely resembling" the searched sub-mesh to more perturbed versions). In a post-step, the replacement algorithm has been used on the selected point- and edge-sets or their connected components, typically placing and orienting an object such that it fit into the resulting bounding box. The latter is quite trivial to extend to arbitrary geometry that would be ill-suited for bounding-box placement, using specific rules stated in a scripted program of the modeling platform.

Basic test cases

During the development of the approach, we have used a set of basic test cases for assessing the algorithm:

- In the simplest case, we have looked at a set of spheres (Figure 5a) with increasing tesselation (4, 8, 16, 24 vertices as base). As result, we could show that a regex of k ANG transitions can only find objects with tesselation greater or equal to k (a 8-ANG regex will find the 8-, 16- and 24-sphere but not the 4-sphere).
- Proportions were tested on a scene resembling a room, simplified as cubes of different sizes (Figure 5b). We were able to find different geometries based on their proportions, however, it must be mentioned that we also had counterintuitive cases where geometry is so proportionally close that one regex intended for a specific type of furniture also returned a different geometry (not a false positive in the classical sense, though). We have furthermore perturbed the regex pattern, and checked that angular precision can comfortably cope with such errors.
- We tested the 3D "search and replace" algorithm using the colosseum mesh shown in Figure 5c. For every instance of the matched search pattern (shown red in the lower part of Figure 5c), we used the bounding box to locate the replacement. The orientation (heading, pitch, roll) of the replacement was concluded from the found points in comparison with the search pattern. Further tests for "search and replace" were also conducted with a pyramid (Figure 5d) and a gerbera, which was turned into a lily by replacing each leaf (Figure 5e).

Reconstruction of destroyed synagogues

We are currently testing the 3D regex algorithm on large-scale models in the context of virtual reconstruction of destroyed synagogues, mainly stemming of the period 1890-1910 (see Figure 6). Though hundreds of synagogues were in this era

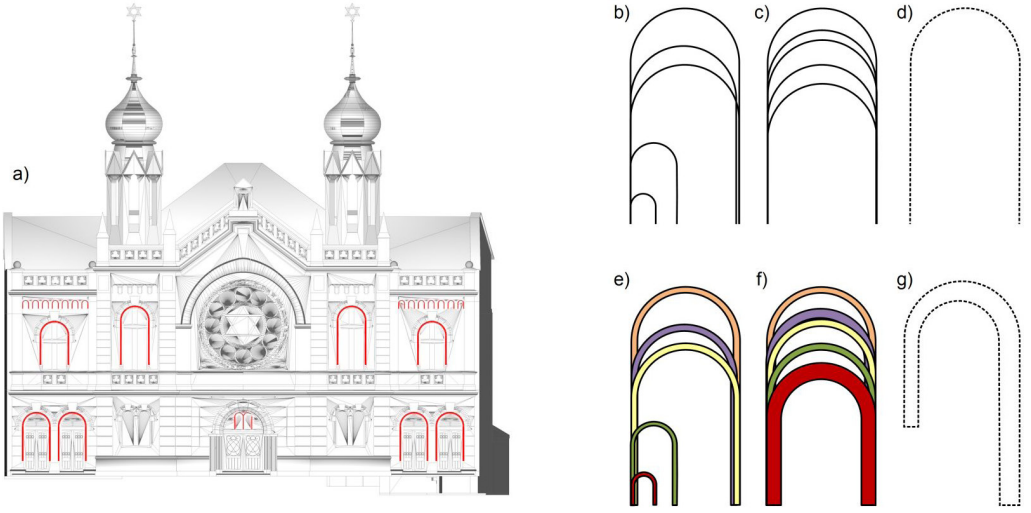

Figure 6
Matching "windows" in the synagogue case study. (a) Overview of matches, (b) arc - real size, (c) proportional size, (d) non-proportional size, (e) connected arc - real size, (f) proportional size, (g) non-proportional size.

built all over Europe, a significant portion has been destroyed in 1938. The option of re-erecting these sacral buildings is not on the agenda, particularly due to a missing usership locally. By way of a virtual reconstruction a certain degree of commemoration is facilitated. However, issues of incompleteness and missing bits and pieces of information play a central role in the process of reconstruction.

The collection of already realized 3D models serves as a knowledge base for ongoing reconstruction activities. However, the secure detection of already existing modeled elements - aiming at "re-use" - is cumbersome. First of all a subset of suitable entities has to be identified and eventual adaptation of the existing modeling properties might be considered. Instead of a laborious manual search through individual models, the goal is to be pointed in a straight-forward and structured way to (similar) building elements in the whole model collection. The 3D-model itself can be regarded as a structured database. Extracting information from a large set of models would enhance the orderly subsequent re-use of already modeled geometries. Any further building models added to the collection would presumably donate to the range of so far not recorded geometries.

To which kind of geometrical representations would this predominantly apply? In the past years the following sets of entities are for the building type "synagogue" of repetitive nature: furniture, bima and torah ark, ornaments, doors and windows, banisters, columns and ceiling beams, tower and dome elements. For our study, we have taken a first step in matching repetitive geometry, in the form of doors and windows (which we model as arcs, see Figure 6 for an overview), ornaments (Figure 7a), columns (Figure 7b) and ornamented windows (Figure 7c). We are far from finished with that undertaking, but the first results are already quite promising with respect to insights that would occur in a "real" project situation where geometry is to be searched.

To begin with, the mesh we are trying to search in can be considered a pathological case: Albeit being seemingly well-structured (see Figure 8a), a closer look reveals that it is composed of a multitude of overlapping components with little or no semantic interconnectedness. In the example in Figure 8b, we can see a door that is formed by a wall in the background (which is shared with another door to the left), a single arc in the foreground, and several ar-

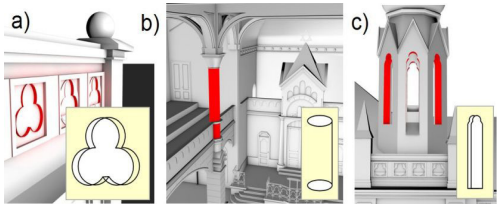

Figure 7
(a) Ornaments, (b) columns, (c) ornamented arcs.

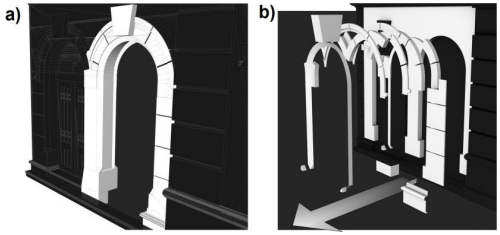

Figure 8
(a) Pathological door made of (b) arbitrary sub-meshes.

bitrarily connected arcs in the middle. Matching has proceeded only in the foreground arc, since that is at least connected. The truth is that, however, a designer would not know why the regex cannot match the whole door shown. What would be needed is an algorithm that can combine overlapping geometry in order to facilitate a "what you see is what you match"-sort of approach, which we have still not produced (future work).

Furthermore, there is a subtle influence by the regex chosen for arcs. Figures 6b and 6e show the real sizes of arcs that are matchable in the mesh. However, as the regex is size invariant, it actually "sees" only relative proportions as depicted in Figures 6c and 6f. Even if we increase percentual tolerance, we still get false positives, such as those above the main entrance (see Figure 6a). Optionally comparing the actual size of the pattern to the found instance may solve the problem, but this is again to be done as future work.

Choosing a regex without proportions reveals that the arcs all have the same radius (see Figures 6d and 6g). However, this is a bad choice, since that would also match every half-circle (height of the lower part close to zero). Without proportions, the situation is even worse in connected search patterns (Figure 6g), since we could get a deformed match which is hardly what we wanted in the first place. There is no way to circumvent this problem - matching without proportions just produces bad results (or, more precisely, results which are correct from an angular view but counterintuitive to us).

IMPLEMENTATION

At the moment, we have two different implementations: A plugin for the Maxon™ Cinema4D® modeling environment (written in C++), and a more academical implementation utilizing the NetLogo programming language. The first one was written in 2002 as part of the diploma thesis of the first author; the second one is a vanilla implementation that seeks to faithfully implement what was written in the thesis, in order to have some degree of quality control for this paper. NetLogo is rather slow with regards to performance, but its visualization capabilities and functional programming language make it an ideal test-bed for exploring further extensions of the approach.

Coming to performance, we disregard the NetLogo implementation (which is rather slow because of running on a Java Virtual Machine with added NetLogo stack on top). The C++ implementation is better - at average, 650 vertices per second per core (2.4 GHz 32-bit processor), although this largely depends on the mesh structure (good tesselation, connectivity/density), precision settings (more tolerance means more possibilities are tried, which slows down the algorithm) and the use of proportions (not using them generates more possibilities, again slowing down the program). For example, in the synagogue use case described earlier, the algorithm would find occurrences of the simple arc show under Figure 6c in either 6min 30s when the precision values were very strict (regex exactly resembling the sub-mesh, all precisions set to 1) or 1h 40min for a very loose setting (regex approximating the sub-mesh, angular and closure precision 30, percentual tolerance 10). This boils down to a performance of either 1500 (strict case) or 100 vertices (loose case) per second per core, with raises a variety of questions and analysis tasks for future work.

In that context, we also wish to note also that the overall efficiency is highly coupled to the search pattern (and thus: the compiler) used; a regex that discriminates early (i.e. sharp angles first, before coming to rounded forms) has a far better performance than one that considers discriminating factors as last step. Devising a better regex compiler and searching in an optimized mesh (overlapping edges and points merged) are clearly on our agenda. Also, future versions of the approach might lead away from the idea matching linearly in an automaton but recursively in the supplied search pattern (i.e. without compilation, but still utilizing the presented concepts).

CONCLUSION AND OUTLOOK

We have presented an algorithm that can search in meshes that have lost all information but their vertices and faces, based on regular expressions and angular search. The benefits of this are threefold: (1.) We can restore object identity, (2.) we can replace multiple instances of the found geometry by a reference to a single geometry container and (3.) we can replace found geometry by an alternative one.

Two case studies frame the presented approach: The "basic test cases", which we applied during development, and the ongoing "synagogue" test cases, which use a collection of models exported from CAD. As discussed, the first results with the latter domain have shown that the complexities associated with "real" data are not to be underestimated: The data is both huge (typically 350K vertices, 450K polygons) and of bad quality (overlapping geometry, unintelligible polygon groups forming connected components, bad tesselation). On top of this, the expectation regarding the growth of the model collection in the next coming years is expected to be substantial.

A meta-search will therefore become of even more importance, connected with a pre-step for automated simplification and cleaning of the mesh which lies on our future agenda. Also, building a library of patterns used for matching as well as a taxonomy that connects these would seem a useful extension that is yet too early to undertake, as we have to fix the foundations first. Other tasks that we would like to look into in the future are: data extraction from laser scan data and 3D fractal analysis of architecture based on "finding sub-meshes within sub-meshes".

ACKNOWLEDGEMENTS

This work is based on a diploma thesis (Wurzer, 2004) supervised by Katja Bühler (Vienna UT and VRVis Forschungs GmbH), Peter Ferschin and M. Eduard Gröller (Vienna UT). The synagogue model base is a results of a continuing effort in virtual reconstruction by Bob Martens (Vienna UT), Herbert Peter (Academy of Fine Arts Vienna), among many others participating in that effort.

REFERENCES

Forta, B 2004, *Sams Teach Yourself Regular Expressions in 10 Minutes*, Sams, Indianapolis.

Berchtold, S and Kriegel, H-P 1997, 'S3: Similarity Search in CAD Database Systems', *Proceedings of the SIGMOD Conference*, May 13-15, Tucson, USA, pp. 564-567.

Sung, R, Rea, H, Corney, JR, Clark, DER and Pritchard, J 2002, 'Shapesifter: A retrieval system for databases of 3D engineering data', *New Review of Information Networking*, 8 (1), pp. 33-53.

Funkhouser, T, Min, P and Kazhdan, M 2003, 'A Search Engine for 3D Models', *ACM Transactions on Graphics*, 22(1), pp. 83-105.

Peter, H 2001, *Die Entwicklung einer Systematik zur virtuellen Rekonstruktion von Wiener Synagogen*, Diploma Thesis (Vienna University of Techology).

Sung, R, Rea, H, Corney, JR, Clark, DER and Pritchard, J 2002, 'Shapesifter: A retrieval system for databases of 3D engineering data', New Review of Information Networking, 8 (1), pp. 33-53.

Wurzer, G 2004, *3D Regular Expressions: Searching IN Meshes*, Diploma Thesis (Vienna University of Techology).