By-example Synthesis of Architectural Textures
supplemental material

Sylvain Lefebvre[1,3] Samuel Hornus[2,3]

[1]REVES / INRIA Sophia Antipolis [2]GEOMETRICA / INRIA Sophia Antipolis

Anass Lasram[3]

[3]ALICE / INRIA Nancy

This document contains some material to supplement the article

S. Lefebvre, S. Hornus ans A,. Lasram. *By-example Synthesis of Architectural Textures.* In ACM Transactions on Graphics 29(4), 2010. (Proceedings of the annual ACM SIGGRAPH conference.)

This additional material first shows more synthesis results in Section 5 and give further details and illustrates the way our synthesised textures are filtered in the GPU fragment shader in Section 1

While the main paper gives a complete description of our system for synthesizing architectural textures, there is also a number of small algorithmic improvements that build on the special nature of our problem and provide a faster implementation. We describe them here, since they are not specifically related to the core of our synthesis system. They deal with Dijkstra's algorithm (Section 2, page 1), the integration of a histogram computation into Dijkstra's algorithm's inner loop (Section 3, page 3) and the fast construction of the graph G (Section 4, page 4).

Many notations that are used here are defined in the main paper and not redefined here. Please refer to the paper for their definition.

1 Filtering

As described in the paper, textures are rendered by transforming texture coordinates to locations in the source image. This simple transformation leads to visible seams around the cuts when using hardware filtering. In fact texels on each side of a cut are not contiguous in the source image. To circumvent the problem, we sample two additional texels having the same coordinate as the current texel but belonging to the previous and the next strip. The three resulting texels are blended depending on the MIP-map level and the distance between the current texel and the two borders of the current strip. Figure 1 highlights the approach.

Figure 2 reports the performance and limitations of the proposed filtering approach. The approach provide very good results for the first levels of detail and enable correct anisotropy. The coarsest MIP-map levels however, show some discontinuities due to multiple strips within the filter width. In practice, these discontinuities are not perceived because the textured surface is far enough from the viewer.

2 A specialized implementation for Dijkstra's algorithm

Basics We compute shortest paths that start from some node $n^\star = (c^\star, z_0)$ in G. The graph nodes are processed in order of increasing distance from $n^\star$. (The distance from $n^\star$ to node n is the cost of the shortest path connecting them, as defined in the main paper). To do so, we store the unprocessed nodes in a priority

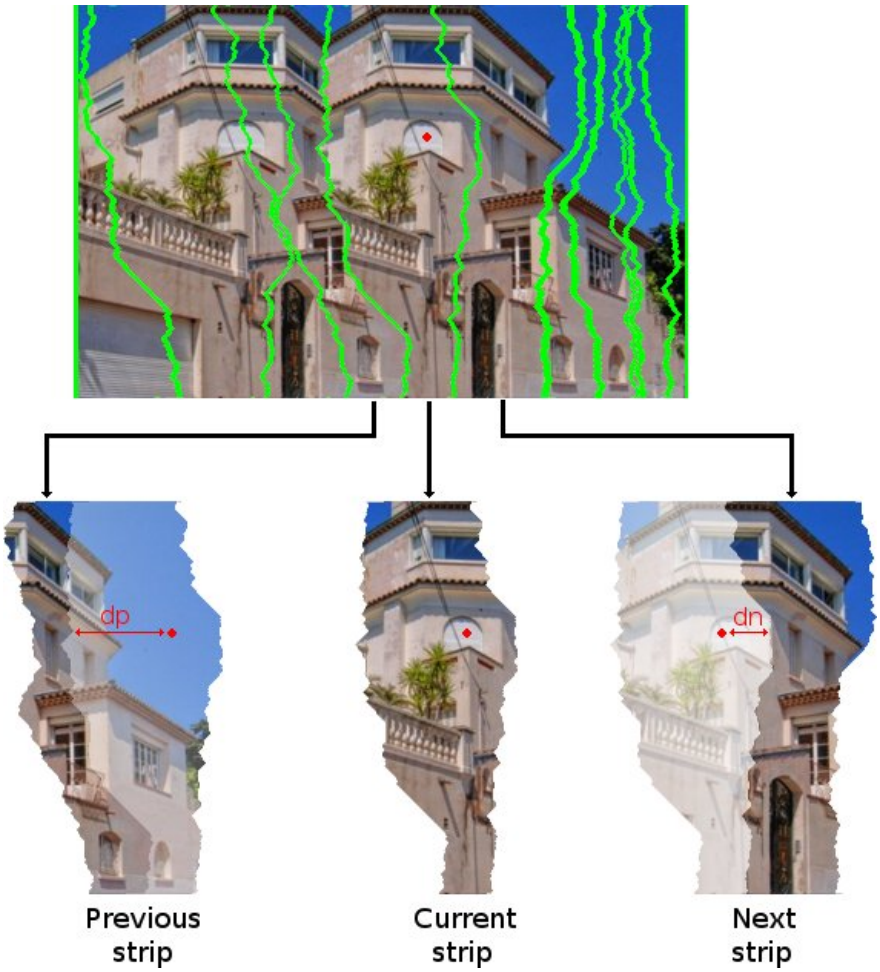


Figure 1: Filtering, the red dot texel in the current strip is blended to the two superimposed texels in the previous and next strip. The blending factor depends on the MIP-map level and the distance dp and dn.

queue. When a node n is processed, each outgoing edge is examined and the distance from $n^\star$ to the adjacent node n' is updated if needed (in which case the position of n' in the priority queue is updated). The process is repeated until the target node $c^\dagger$ gets processed.

Stopping and restarting the search When the target node is reached, Dijkstra's `while` loop is stopped, but the (probably) non-empty priority-queue is kept, so that we can restart the computation from where we started if a new target node is asked for, that has not been yet processed.

Recall that for a desired width $\mathcal{T}$ we need only consider a finite subgraph $G' = (\mathcal{C} \times [c^\star_{min} - c^\star_{max} .. \mathcal{T}], E_\prec \sqcup E_\parallel)$. Accordingly, we allocate just enough memory for $|\mathcal{C}| \times (\mathcal{T} - c^\star_{min} + c^\star_{max} + 1)$ nodes.

When processing node n, if one of its adjacent node $n' = (c', z')$ falls outside the allocated space ($z' > \mathcal{T}$), we add node n in a separate pool $\mathfrak{P}$ of nodes to be re-processed later should the target width be increased.

Assume, thus, that we need to reach a second, further node $n^\dagger = (c^\dagger, \mathcal{T}')$ with $\mathcal{T}' > \mathcal{T}$. First, we reallocate enough memory and copy the processed nodes in the new location. Second, we insert the relevant nodes from the pool $\mathfrak{P}$ into the priority queue prior to restarting Dijkstra's inner loop; they are the nodes (c, z) with $z \leq \mathcal{T}'$. In that way, no node is missed and the shortest-paths tree computation proceeds correctly, in an interruptible fashion.

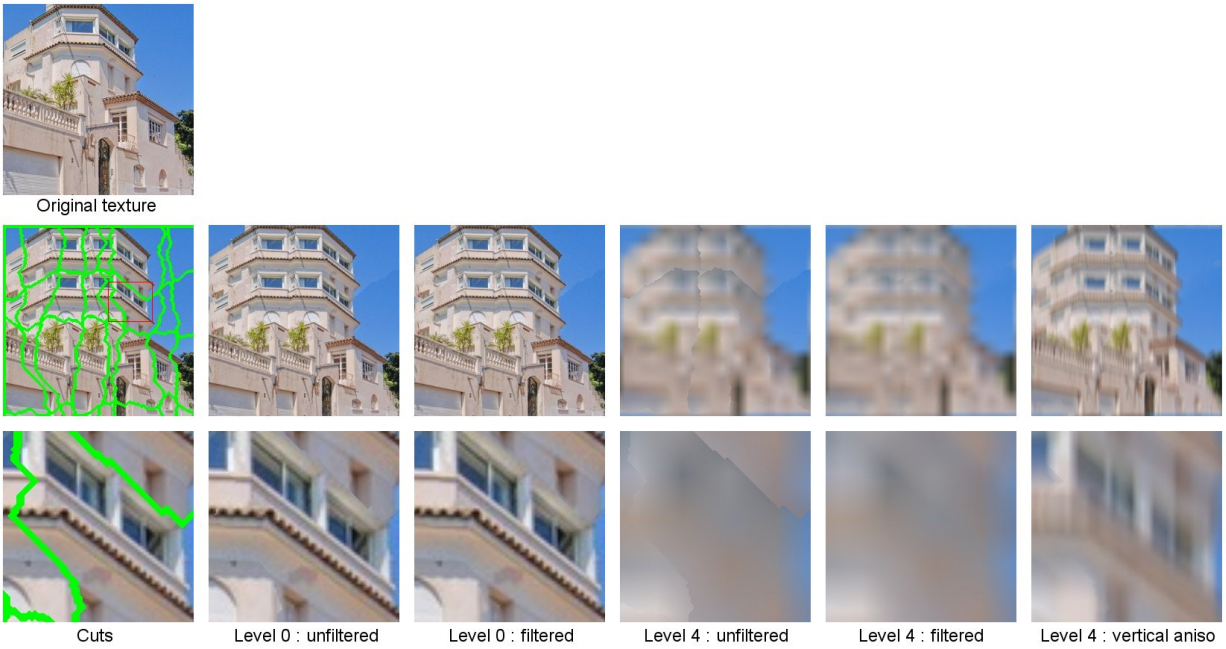


Figure 2: Filtering performance. *Top:* Original texture, *Middle:* Enlarged texture with different level of detail and different filtering mode, *Bottom:* Zoom-in of the marked red rectangle.

Unraveling the strips We describe a simple optimization to accelerate the inner loop. Let n be the node being processed: the shortest path from $n^\star$ to n is thus known and the distance from $n^\star$ to n is the smallest among all the nodes that haven't been processed yet. If we follow a **growth**-edge leading to node n' (we unravel a strip that starts at n), observe that the distance of $n^\star$ to n' is the same as that of n, since growth-edges do not increase the cost of a path. We deduce that, if n' has not been processed yet, we have just found the shortest-path to n'. Generalizing, we follow all possible growth-edges from n, recursively. The recursion is faster than update operations in the priority queue, which saves a significant amount of time. This corresponds to unraveling all the strips that starts at n.

In this recursive exploration, nodes at the other end of a **start**-edge are examined and possibly inserted in the priority queue (but we do not recurse on it). Processing the nodes in the queue resumes after the end of the recursive exploration of the strips started by n. Note that the recursive exploration always ends since a strip cannot grow indefinitely.

3 Integration of the the histogram-based limitation on repetitions

The "strip-unraveling" recursive process described above turns out to be crucial for our very efficient implementation of the histogram computation.

Limiting repetition by forbidding edges in Dijkstra's inner loop An assembly of strips from the source image is a sequence of cuts $p = c_0, c'_0, c_1, ..., c'_{k-1}, c_k, c'_k$ where c'_i is parallel to c_{i+1}. We make the approximation that for each strip $c_i \to c'_i$, the source columns with index in $I_i = \{c_{imin} + 1, \ldots, c'_{imin}\}$ are copied into the result. The disjoint union of the sets I_i is a multiset. Let C^j be the index of the column of the source image that appears as the j-th column in the result image. For window size ξ and threshold R, we must ensure that for all Δ, no index appears more that R times in the multiset (or histogram) $H_\Delta = \{C^j | \Delta \leq j < \Delta + \xi\}$.

We maintain the histogram H_Δ as Δ ranges from 0 to $\mathcal{T}-\xi$ with unit increments. To efficiently maintain the maximum value $\mathcal{M}$ in the histogram (the maximum multiplicity in the multiset), we also maintain a histogram of the histogram: when the multiplicity of $\mathcal{M}$ goes down to zero, we can then quickly find the new maximum, by decreasing the value of the maximum until we find a non-zero multiplicity in the "histogram of the histogram". Whenever $\mathcal{M}$ gets larger than R, the path is declared bad and ignored.

When processing a node in Dijkstra's algorithm, we first construct the histogram H_{Δ_0} with a value of Δ_0 so that the window of width ξ is at the end of the corresponding result image. When a growth-edge is examined, the window is translated by increments of one pixel until the end of the new result image. If, in the process, $\mathcal{M}$ goes above R, the edge is not followed. Such a process is slow. We describe below how to accelerate it.

Doing it faster Recall that when examining a node n, we recursively explore all the strips that starts at n instead of relying on Dijkstra's priority queue. As we explore the strips from n, only cut c'_k defined above changes (the current shortest-path from c_0 to c_k doesn't change). Thus, only the set I_k changes and simply grows to the right.

The trick is simple: before starting the recursive exploration, we "proactively" update the histogram by growing the set I_k as far to the right as possible (this is limited by the size of the source image). This gives us a maximum value of Δ, called Δ_n, beyond which the threshold R is exceeded, and below which everything is fine. Then during the recursive exploration, instead of slowly updating the histogram, we simply have to check whether the current Δ is lower that or equal to Δ_n. This is a very fast test.

When a new node n' is popped from the priority queue, we rebuild the corresponding path (from $c^\star$ to n') and histogram, and repeat the procedure prior to the next recursive strips exploration.

This optimization speeds up the shortest-path computation by more than an order of magnitude. For small window-width ξ we could store the current histogram directly in each node in the priority-queue to further accelerate the search. We haven't tested such an option as such small values of ξ do not get us rid of visually unpleasant repetitions.

4 Fast construction of the graph G

We examine the following task: Given a set of cuts $\mathcal{C}$, compute $\mathfrak{F}(c) = \mathrm{Front}(\mathrm{Succ}(\mathcal{C}, c))$ for each cut $c \in \mathcal{C}$ as fast as possible. Here is a simple algorithm to do so:

```
find_front(CutSet C, Cut c) {
    CutSet F = empty set;
    for( all c' in C ) {
        if( c does not precedes c' ) continue;
        bool is_front = true;
        for( all f in F ) {
            if( c' precedes f ) remove f from F;
            if( f precedes c' ) is_front = false;
        }
        if( is_front ) insert c' in F;
    }
}
```

This algorithm has, at best, quadratic complexity. And $|\mathfrak{F}(c)|$ doesn't necessarily have constant size, which makes the algorithm impractical.

Assume now that the cuts in $\mathcal{C}$ are ordered as $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ in such a way that for all $i < j$, the cut c_j does not precedes c_i ($c_j \not\prec c_i$, so either $c_i \prec c_j$ or c_i and c_j cross each other). Then the algorithm can be simplified a little to compute $\mathfrak{F}(c_i)$:

```
find_front(CutSet C, CutIndex i) {
```

```
    CutSet F = empty set;
    for( j = i+1 to n ) {
        bool is_front = true;
        for( all f in F ) {
            if( f precedes c_j ) is_front = false;
        }
        if( is_front ) insert c_j in F;
    }
}
```

This is the algorithm that we use, with a first step that prune a big part of the set of candidates $\{c_j | j \in [i+1..n]\}$ prior to examine them. We now explain how to obtain such an ordering of $\mathcal{C}$ and how to prune the candidate set. Again, we gain more than an order of magnitude for computing $\mathfrak{F}(c)$ compared to the naive algorithm.

QuickCutSort and the Cut-Tree Our QuickCutSort algorithm is inspired by quicksort for ordering $\mathcal{C}$. The algorithm recursively partitions the array of cuts into three consecutive subarrays. A pivot cut p is chosen randomly. The left subarray L_p contains cuts that precedes ($\prec$) the pivot. The middle subarray X_p contains cuts that cross the pivot. The right subarray R_p contains cuts that the pivot precedes. Each subarray is recursively ordered. The partition is easily done in one pass over the array.

The Cut-Tree is a tree in which each node represents an interval of consecutive cuts in the set $\mathcal{C}$ ordered as above, and also contains a pivot cut (contained in the interval) and three children. The tree simply reflects the execution of the QuickCutSort algorithm above, which makes it a kind of search tree on the poset of cuts $\mathcal{C}$.

When we compute $\mathfrak{F}(c_i)$, instead of checking all the cuts in $\{c_j | j \in [i+1..n]\}$, we use the tree $\mathcal{T}$ to compute a much smaller set of candidates, as an ordered subsequence of $\mathcal{C}$. The construction of the tree take additional time and space, but overall the graph G is constructed roughly twice faster.

Let c be the cut for which we want to compute $\mathfrak{F}(c)$. We start at the root of the tree $\mathcal{T}$ and recurse. Let p be the pivot at the current node of $\mathcal{T}$.

- If $c \prec p$, then we recurse in L_p, we append p to the candidate set and we recurse in X_p. We effectively omit R_p from consideration although their appear after c in the ordering given by QuickCutSort, because a cut in R_p can not belong to $\mathfrak{F}(c)$.
- If c crosses p, then we recurse in X_p and we recurse in R_p.
- If $p \prec c$ or $p = c$, then we recurse in R_p.

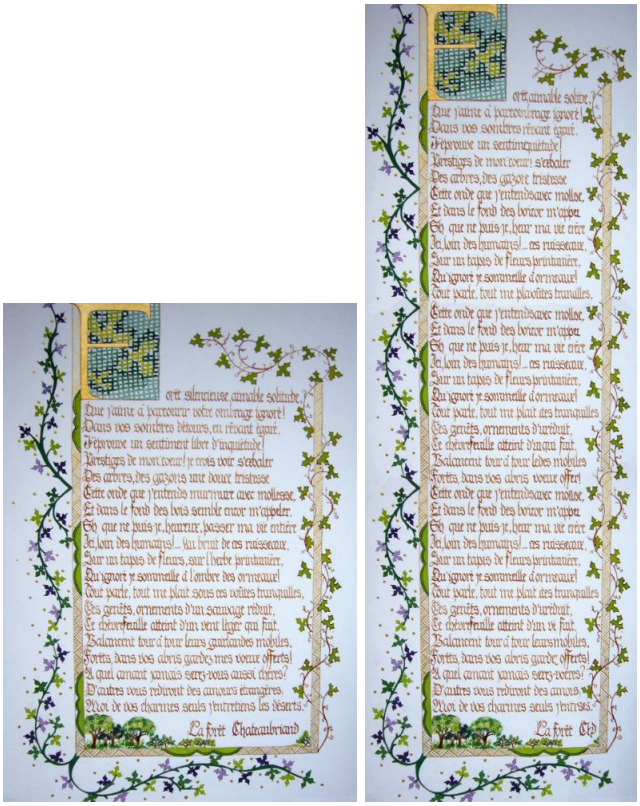
This simple tree traversal gathers all the cuts in $\mathfrak{F}(c)$ and a little more, so the same pseudocode as above is used to remove the false candidates.

5 Additional results

We here provide more synthesis examples.

Mandala The original (left) has size 579x580. Image courtesy of Denise Schwab. No control is used in the following results. The histogram parameters are $R = 2$, $\xi = 100$.

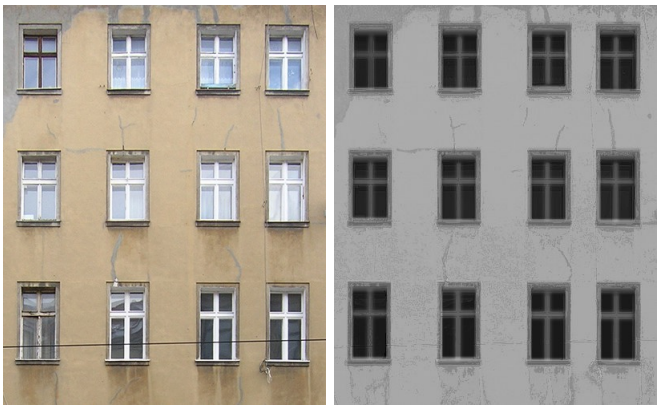
Chateaubriand The original (left) has size 443x625. Image courtesy of Claire Royer. No control is used in the following result. The histogram parameters are $R = 2$, $\xi = 100$.

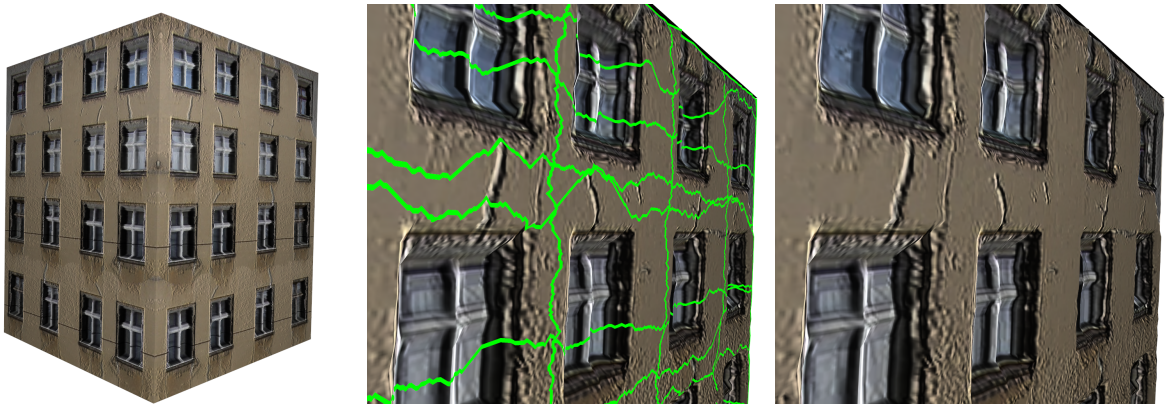


Industrial The original (left) has size 512x367. Image courtesy of www.cgtextures.com. No control is used in the following result. The histogram parameters are $R = 2$, $\xi = 400$.

Video-game Door The original (left) has size 218x340. Image courtesy of www.cgtextures.com. No control is used in the following result. The histogram parameters are $R = 2$, $\xi = 100$.

Parallax occlusion mapping Up to now, only RGB colors were considered for synthesis which is not sufficient in applications like parallax-mapping or displacement-mapping. The picture below shows a parallax occlusion mapping effect using synthesis from feature vectors containing color and relief information. Image courtesy of www.cgtextures.com.





Some facades We show various syntheses from facades pictures (starting next page). Images courtesy of [TSKP10].

References

[TSKP10] O. Teboul, L. Simon, P. Koutsourakis, and N. Paragios. Segmentation of building facades using procedural shape priors. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2010.

Source image: 

Three results with variety. $\xi = 100$, $R = 1$, $D = 70$:

A lower building. $\xi = 300$, $R = 1$:

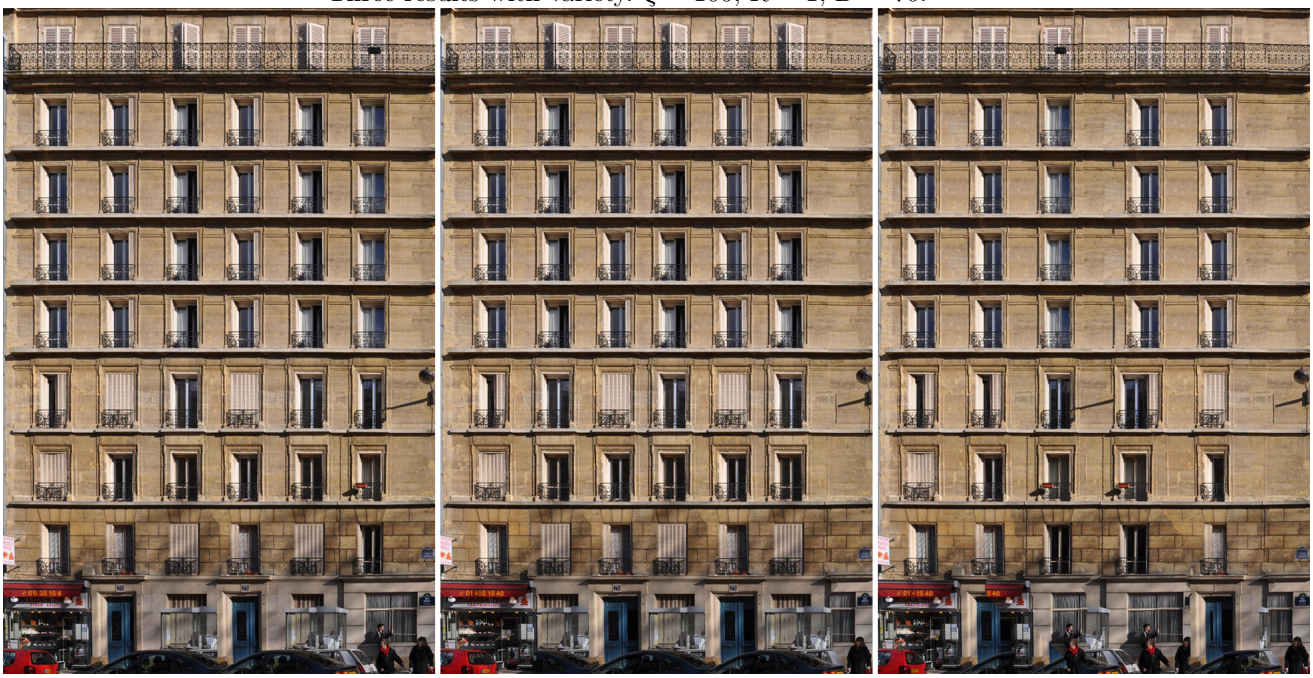
Source image: 

Three results with variety. $\xi = 100$, $R = 1$, $D = 70$:

A lower building. $\xi = 300$, $R = 1$:

Source image: 

Three results with variety. $\xi = 100$, $R = 1$, $D = 70$:



A lower building. $\xi = 300$, $R = 1$:

Source image: 

Three results with variety. $\xi = 100$, $R = 1$, $D = 70$:

A lower building. $\xi = 300$, $R = 1$:

Source image: 

Three results with variety. $\xi = 100$, $R = 1$, $D = 70$:

A lower building. $\xi = 300$, $R = 1$:

Source image: 

Three results with variety. $\xi = 100$, $R = 1$, $D = 70$:

A lower building. $\xi = 300$, $R = 1$: