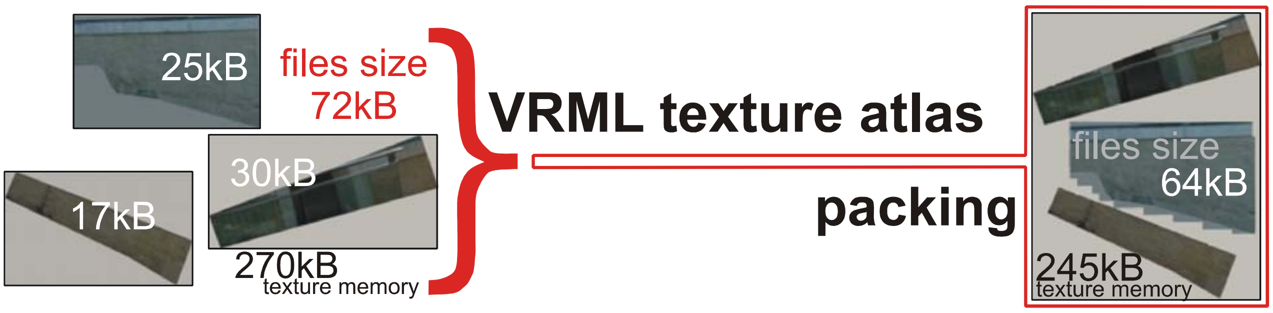
Heuristic approach to automatic texture atlas composition

David Sedláček[*]
Czech Technical University in Prague



Abstract

We present a solution for a variant of the 2D containment problem, to create a texture atlas. The texture atlas is an efficient color representation for a given VRML scene. It contains a composition of several texture images stored in one file. The containment problem is the question how to place a set of shapes into a "container shape" without overlapping, while minimizing the area of the container. In our case we place a set of texture images of arbitrary shape into a rectangle. We use a heuristic search algorithm based on Minkowski operators to solve the 2D translational containment problem. This particular problem is called packing problem and is known to be NP-complete. We present its practical implementation and show results measured on several test scenes.

Keywords: Minkowski operators, texture atlas, packing problem.

1 Introduction

Both number and size of files negatively influence the speed when large virtual worlds are transferred over the Internet. For a given virtual scene, we can efficiently decrease a number of texture files using a texture atlas. The texture atlas is a bitmap containing a set of input textures. When using a good packing method, the size is decreased, too. Saving a texture memory of graphic card is thus a next good reason for using texture atlases. Unfortunately a combination of texture atlas and mipmapping technique is impossible in general, but this specific issue is not the goal of this paper.

The basic idea of our algorithm is to extend successively an arbitrary-shaped container with incoming arbitrarily-shaped textures by attaching the textures on the outer border of the container. The goal is to keep the rectangular area of the resulting container as small as possible. The Minkowski operation is used to compute efficiently all positions the incoming polygon can be attached to using translations. Afterwards we choose that position where the container area is minimally expanded and the empty space between polygons is the smallest (with "placing operator").

We did not implemented rotations yet but we consider them as important improvement of the next version of the algorithm. However the angle of rotation different from multiples of $\pi/2$ brings degradation of the texture quality due to the necessary interpolation among pixels applied during the rotation of input raster image(s). The second problem with rotations arises in back-mapping of the texture map to the original 3D object. The texture mapping coordinates are from interval <0, 1> and if we rotate by a small angle the change is manifested in decimal places. We cannot be sure in number of decimal places VRML browsers work and if they recognize such "little" changes. From there reasons we expect useful rotations only by multiples of $\pi/2$.

This paper is organized as follows. Section 2 presents a related works, section 3 deals with VRML specific issues – image formats, and texture mapping. We define morphological operators and their properties in the section 4. We skip a formal description of the 2D translational containment problem and the state definition, since they are well described in [4]. Instead, we concentrate on the heuristic searching method and containment strategy in section 5. Finally, several examples and measurements are shown in section 6.

[*] sedlad1@fel.cvut.cz

2 Related Work

Texture atlases are widely used in computer games. Avatar texture atlas is the most known example, see fig. 1. Such texture atlas is usually explicitly defined - first the texture atlas is designed, then this bitmap is mapped on avatar body. A priori knowledge of the avatar model is required.

Automatic texture atlas generation for arbitrary, but single 3D object was described in [3].

In our approach, we process textures and scenes of arbitrary structure, having no explicit knowledge about their relationships.

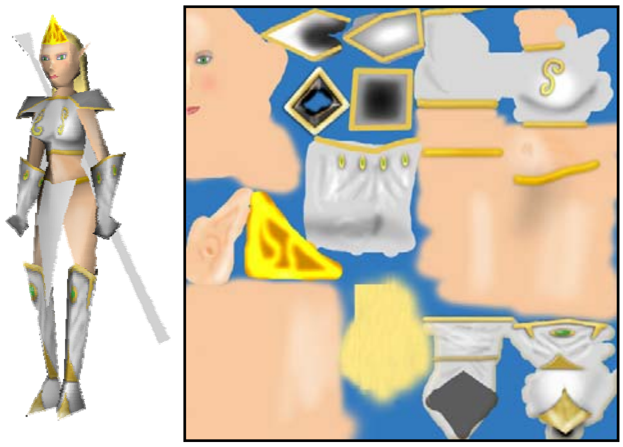


Fig. 1: An avatar and corresponding texture atlas (from http://www.allelves.ru/forest/)

Related research about *containment algorithms* can be subdivided into four categories [5]:

- The physics approach, which applies classical physics theory by adding potential energy to the shapes to place, that can be viewed as physical elements. However, solving these large equations take a large amount of time, what makes it useless when we need fast results.
- The Computational Geometry approach, which latest development in multi-polygon rotational case [1] can place convex *m-gon P* into convex *n-gon* container *Q* by solving $O(m^4 n^4)$ linear programs.
- The Operational Research approach, which can lead to practical results, but tend to become more complex when rotations are part of the equation.
- The Meta-Heuristic approach, which can obtain interesting results if the selected heuristic is appropriate, but fail to be applicable when the input space increases substantially.

The texture atlas packing problem is often solved with the meta-heuristic approach. For tens of texture images, it gives satisfying results in a very short time while other approaches tend to be much slower than the heuristic method. In the method proposed by Sander at al. [6], the bounding boxes of the polygons are packed. They first sort bounding boxes by area sizes, then rotate them to align the longer axis of the rectangle with the vertical direction. In order to decrease a height, they place the rectangles sequentially into rows alternating left-to-right and right-to-left order.

Another heuristics proposed by Lévy at al. [3] work directly with polygonal shapes. They initially sort polygons like Sander, but taken the inspiration at game Tetris, they let polygons fall down vertically and search for the best horizontal position on top of already positioned shapes.

These heuristics are fast but they are not able to cover unused empty space that can naturally appear inside already positioned polygons. For this reason we have decided to take inspiration at Minkowski operators. Minkowski operators make possible to find these unused places in a short time. Minkowski operators (especially difference) tend to be faster than normally known techniques for solving collisions - they do not inform us whether there is a collision, but they tell us where the collision never occurs.

3 Textures in VRML

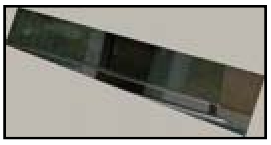
Three bitmap types are used in VRML scenes - GIF, JPG, and PNG. Due to different image format characteristics, we have to solve the containment problem for these formats independently. If using a combination of these formats in the final texture atlas, accuracy, transparency, and compression would be negatively influenced and complicated. For example, when converting GIF to PNG, we lose animation possibility. When GIF or PNG image would be converted to JPG format, the lossy compression would cause image degradation.

Since we want to keep the image quality of all original textures unchanged, we have to manage JPG images in a special way. The DCT transformation processes pixels arranged in blocks of 8 x 8 pixels. This block is called *superpixel*. If input textures are placed into the texture atlas close to each other, original superpixels should not overlap. That is why we process JPG images on the level of superpixels (see fig. 2c) instead of single pixels such as in the case of PNG files.

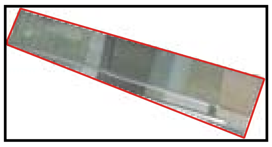
Input sets for our test scenes usually contained from 5 to 60 texture files (the maximum was almost 200 images). Figure 2 presents one typical example of one texture file in JPG format. It can be seen that one input image can contain pixels/texels that are not mapped to a 3D model. Those pixels can be avoided from further processing, thus achieving more efficient use of space in the final atlas.

After placing input images into the texture atlas, all relevant texture coordinates have to be recomputed in the source VRML file(s). Such remapping process includes only a simple arithmetic. A problem occurs when the original texture image is applied as a tile, i.e. the VRML source code maps its 2D texture coordinates out of the standard range <0, 1>. Although this is a regular VRML technique, it cannot be used in combination with the texture atlas, since newly computed texture coordinates are directed to another image data in this case. We seek for such a situation in the VRML code and avoid relevant image from the texture atlas creation. For this reason, our method can produce more than only one output image file for a specific image format.

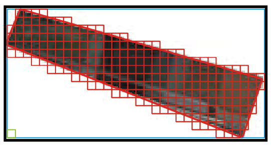
a) One JPG texture image from the input set.



b) Convex hull of image data really mapped to 3D surface. Remaining white space represents wasted area.



c) Polygon approximation with respect to DCT superpixels. Each small square size is 8*8 pixels.



d) Our final polygon representation. The square in left bottom corner is the reference point (0, 0).

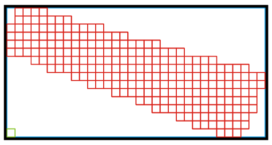


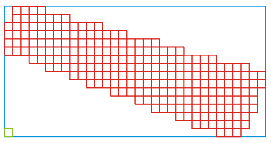
Fig. 2: Input JPG image and corresponding polygon representation

4 Theoretical Background

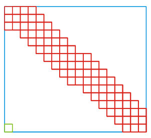
In this section we introduce the basic operators and their properties, especially Minkowski sum and difference operator. Operators help us to find non-overlapping positions for each polygon in the container. This section is based on the work of Marques at al. [4].

Let $A, B \subseteq Z^2$, $t \in Z^2$. Usually, $A, B \subseteq R^2$, $t \in R^2$.

In our approach, we consider *A, B,* and *t* as sets of discrete points.



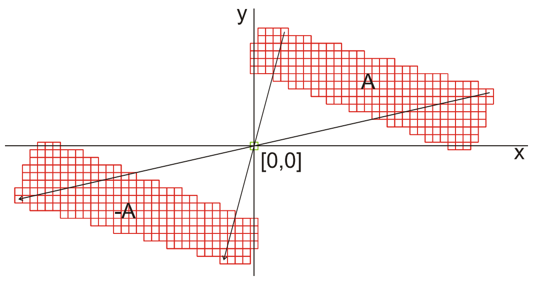
A



B

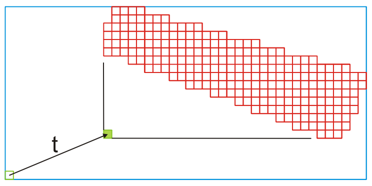
Morphological mirror of *A*, (-*A*), is

(1) $$(-A) = \{-a : a \in A\}$$



Translation *A* by *t*, *A*+*t*, is

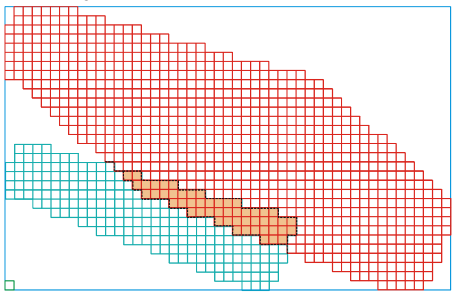
(2) $$A + t = \{a + t : a \in A\}$$



A+*t*

Minkowski sum *A* with respect to *B*, $A \oplus B$, is

(3) $$A \oplus B = \bigcup_{b \in B} A + b$$



$A \oplus B$

The Minkowski sum keeps four relation terms – *reflexivity, commutative law, associative law,* and *transitivity* [2]. The following relations define the Minkowski sum for sets and vectors.

(4) $$A \oplus \{t\} = A + t$$

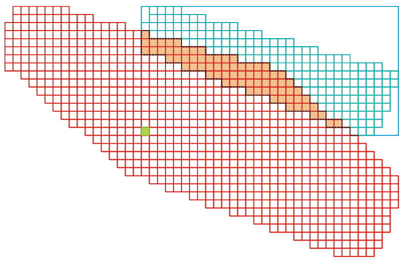
(5) $$A \oplus (B + t) = A \oplus B + t$$

The most useful property determines how translations of shapes are mapped from and into Minkowski sum sets. We have the following fundamental property.

(6) $$\left((B + t) \cap A \neq \emptyset\right) \Leftrightarrow \left(t \in A \oplus (-B)\right)$$

The Minkowski difference A with respect to B, $A \Theta B$, is defined as follows.

(7) $$A \Theta B = \overline{\overline{A} \oplus B}$$



Minkowski difference keeps the previous terms (*reflex., commut., assoc., transit.*) like that Minkowski sum.

The following property says that if a shape *B* positioned at point *t* is contained in *A*, then the point *t* belongs to the Minkowski difference of *A* with the mirror of *B*. By using this property we can get the set of all points *t* in which *B* fits into *A* just by computing the difference [4].

(8) $$\left((B+t) \subseteq A\right) \Leftrightarrow \left(t \in A \Theta (-B) \neq \varnothing\right)$$

5 Heuristic Searching

We start with an empty container *C*, with zero *x* and *y* size (C_x = 0, C_y = 0, *Area(C)* = *0*). Then we add the first polygon *A1* from the input set to the container *C*, thus obtaining a new status ($C + A1$, C_x = $A1_x$, C_y = $A1_y$). The following algorithm is applied when searching for the best position of the next polygon *A* from the input set to be added to the container *C*.

1. Compute the Minkowski difference M_d of *C* with respect to *A*, see in fig. 3.
   This gives us a set of translation vectors *t*, which applied on *A* causes overlapping *C* with *A*. We actually need the opposite of this.

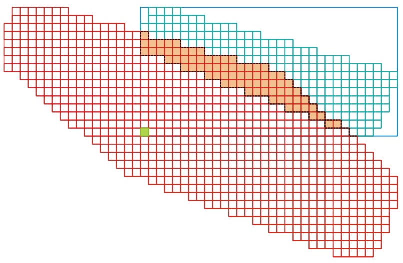


Fig. 3: Container (smaller polygon) shown together with computed Minkowski difference (bigger area). Second polygon for M. difference was polygon A at fig. 6. Those two areas overlapped. The one filled square in bigger area represents reference point (0,0).

2. Compute O_d as the outer border of the M_d area, i.e. find all 4-connected neighboring pixels of M_d (fig. 4.). Then O_d is a set of vectors *t*, which defines all translations of *A* to positions close to *C*, see fig. 5. Such translations are candidates for the final positioning of *A*.

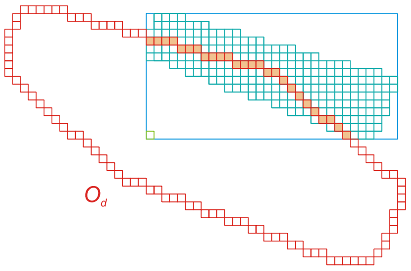


Fig. 4: Outer border of M_d area (O_d), in relation with container.

3. To find the best position of *A*, we evaluate every translation vector *t* (t_x,t_y) from O_d set using *containment strategy* that is explained in details below, in fig. 5.
   Polygon *A*, translated to its final position, is then added to the container *C*, sizes C_x, C_y are updated, and the algorithm is repeated until all input polygons are processed.

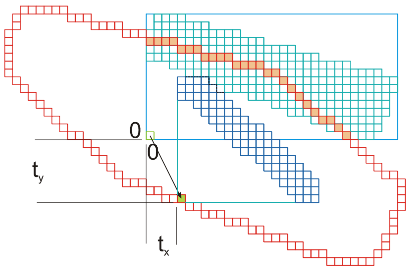


Fig. 5: Placing polygon A next to container C at one position from O_d. We can see the exact matching A to C.

Containment Strategy

When searching for the best translation vector *t* for polygon *A* in container *C*, our aim is to minimize the final container area, see fig. 6. Theoretically, the best *t* does not change that area at all.

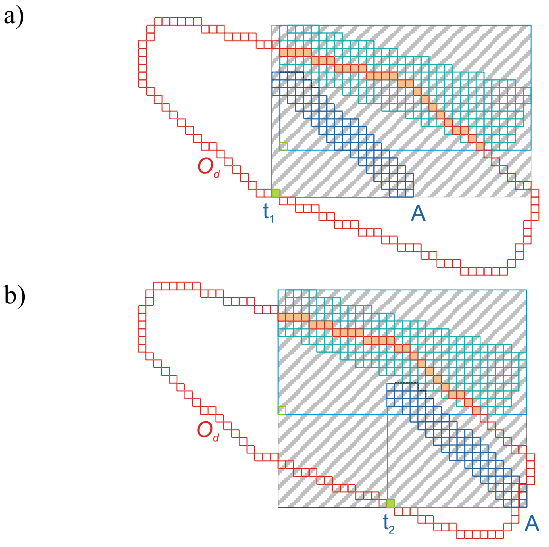


Fig. 6: Minimization of the container area (depicted as hashed rectangle). Two sample translation vectors, t1 (figure a) and t2 (figure b), were used for positioning the polygon A. Figure a) exhibits better results (smaller occupied area) than b).

Since more than one translation vector can satisfy the smallest container area condition, we have designed one additional criterion - *placing operator*. The operator concentrates on filling the empty space inside the container. Although this can generally enlarge the overall container area (see fig. 7), this strategy often leads to better utilized/covered internal space.

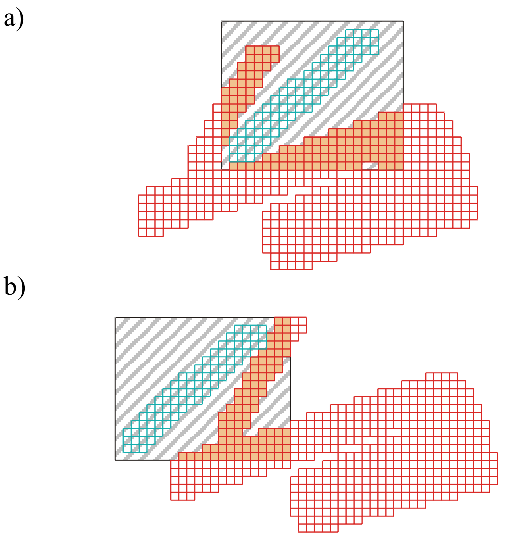


Fig. 7: The *placing operator* has been designed to maximize occupied neighborhood around the newly placed polygon *A*. Such a neighborhood is highlighted by the hatched rectangle; its occupancy by already placed polygons is depicted by solid pixels in red. The higher number of occupied pixels, the higher the placing operator value. Figure a) shows a placing operator with value 104 that is better than figure b) with value 69.

To combine the container area minimization approach with the placing operator application, we have designed the following empirical containment strategy procedure controlled by a single parameter *weight*:

1. For each vector *t*, compute *area* and *place* values. The *area* is the final container area size; the *place* is the placing operator value.
2. Find the minimal *area* value and store it as *minimal area*.
3. Remove all vectors *t* having *area* > *minimal area* + *weight*.
4. Among remaining vectors, find the one with the smallest *place* value and use that vector for positioning the polygon in the container.

The *weight* says how much we accept worse solutions in terms of the container area size. However we are not able to determine the best *weight* value. In our tests, the optimal *weight* value was within the range from 0 to 200 pixels (i.e. from 0 to 10% of current container area respectively). We observed certain dependencies on container area and placed polygon area. Our further experience is described in the following section.

Strategy Evaluation and Experiences

Based on measuring of various input sets, we have got the following observations.

The first polygon added to the container affects the final texture atlas the most. That is why we implemented several selection techniques for the first polygon together with sorting other input polygons. We have got good results when sorting polygons top to down by their area size. Unfortunately, we have found several sets where random polygon selection behaved better. We also tried to compare more configurations for one set of input polygons. We have generated up to 100 random sequences of input polygons for the same scene and then we compared the created texture atlases. The result of such comparisons is shown in Fig. 10.

If the created texture atlas area is similar to a square or rectangle with side ratio about 4:3 (see fig. 8), we generally get better results both from the perspective of file size and required texture memory. Here, the *weight* value helps to achieve such a square-like shape. Without using it, rectangles tend to be too long or high.

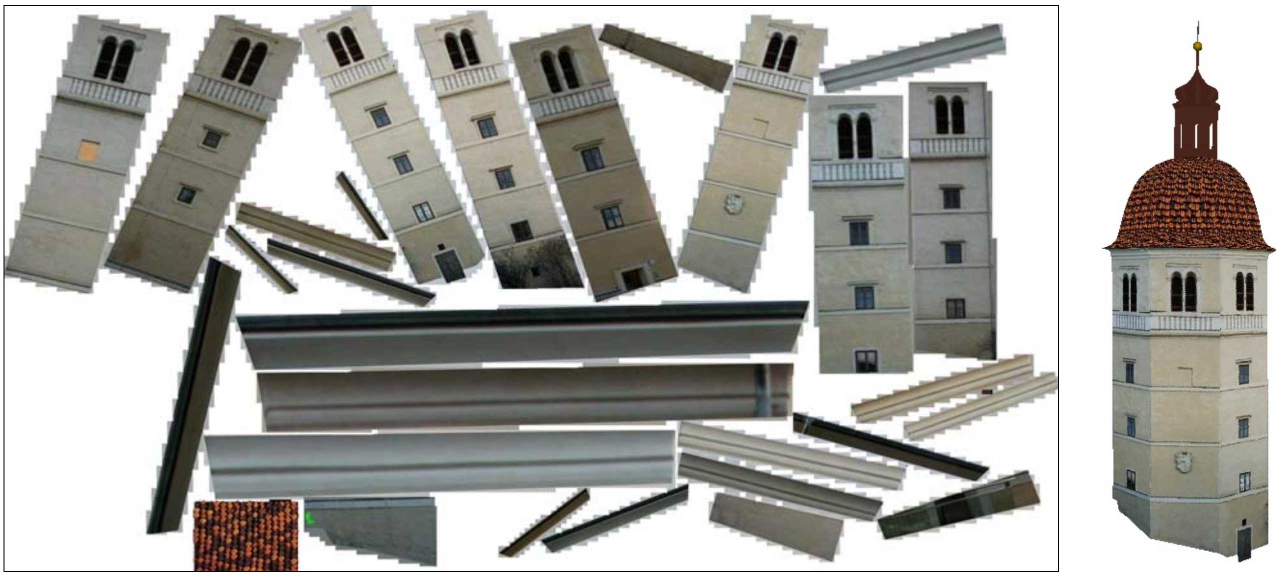


Fig. 8: Texture atlas (left) and 3D model (right). The size of the atlas is 1605 x 1014 pixels.

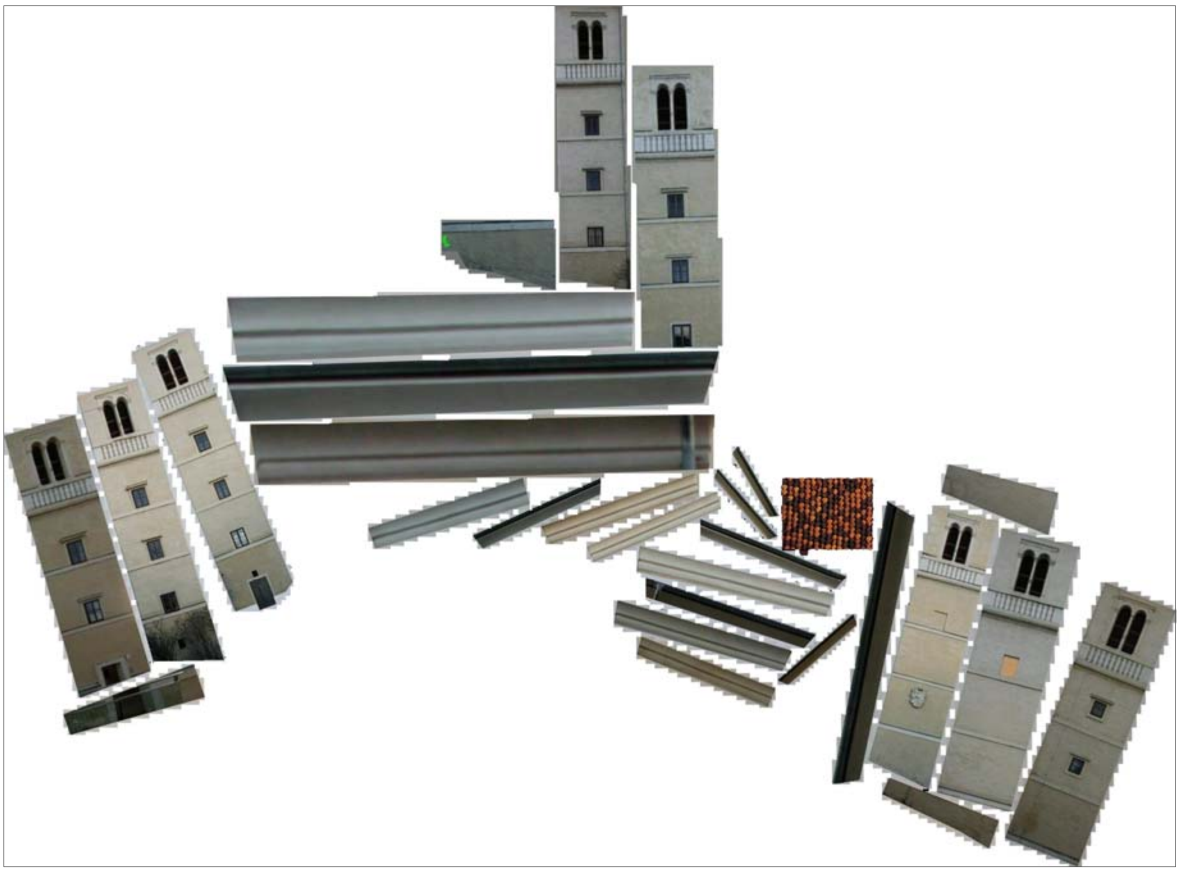


Fig. 9: Texture atlas generated using a placing operator with higher weight (10% of container area). The algorithm is trying to find the best position for every input texture using information from already placed polygons (the highest value of placing operator) but does not care about final rectangular container area size. We can see similarly shaped polygons placed near to each other.

Scene	*Number of bitmaps*		*Summed file size [kB]*		*Texture memory allocation [kB]*		*Transfer time[s]*		*File size reduction [%]*	*Transfer time reduction [%]*
	Input	Output	Input	Output	Input	Output	Input	Output		
Bell tower	30	1	182	168	5853	5013	20	15	8	25
Tower-b1	13	6	169	146	1399	1553	13	9	14	31
Tower-b2	15	7	79	64	1525	1110	8	7	18	12
Bridge-b3	15	5	76	39	665	700	6	4	48	33
Bridge with towers	42	17	278	236	3389	3163	23	17	15	26
Maribor plague	116	64	869	730	11565	12602	64	47	16	17
Town Hall	168	43	1077	1007	20220	21145	71	48	7	33
Maribor synagogue	43	16	1093	978	13776	16261	50	41	10	18
Turk well	66	1	289	166	4197	3662	15	8	43	47

Table 1: A comparison of test scenes without and with texture atlases

6 Results

The algorithms described in the previous section were implemented in Java. Figure 8 shows an example (Bell tower) consisting of 30 JPG input images. In this case, the file size of the final texture atlas was 19% less than the sum of all input file sizes. We also measured the time for transferring a 3D scene (wrl file + bitmaps) over Internet using phone line. We observed a drop from 20 s for the original data set to 15 s for the scene with the texture atlas.

The texture atlas in fig. 8 was computed in 16 seconds on Pentium 1,5 GHz. Since a short processing time was not the primary goal of this work, we consider this value as acceptable. More results are shown in Table 1.

Figure 9 shows the texture atlas for the same input set as for fig. 8, but with a setting that accepts worse container area, while emphasizing the placing operator.

For several models, the size of allocated texture memory is bigger than for the original input. This occurs when input texture polygons fully fill their rectangular image areas, thus composition of textures in the final atlas cannot cover any unused space and stack up empty space leaved by placing algorithm. In this case is more obvious how much empty space is left by the placing algorithm.

The quality of texture atlas can be expressed as decrease of final size (texture atlas size with respect to summed input files size) that affects the time needed for data transfer. It can be also expressed as reduction of necessary texture memory (similar to previous). Because our main intention is to transfer files faster over the Internet we focused at the first criterion.

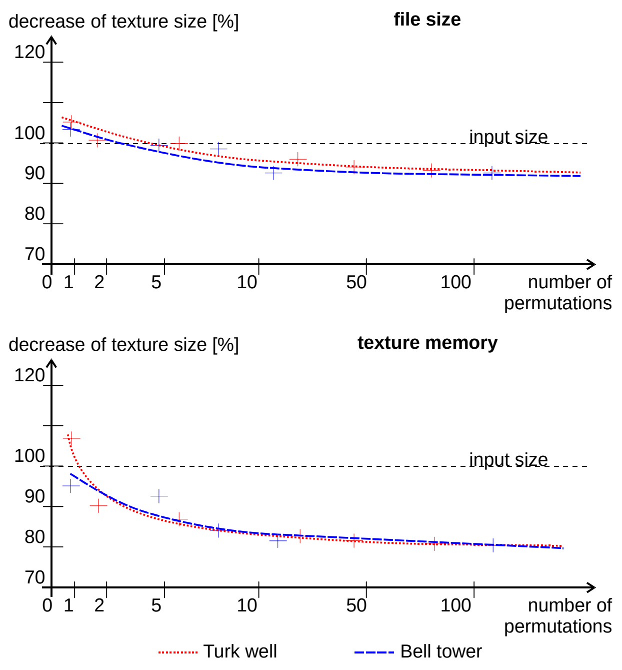


Fig. 10: Quality of texture atlas depending on the length of the iteration process in containment strategy.

Figure 10 shows how the texture atlas quality increases with the number of tested configurations/permutations of a given input texture set. We measured two different scenes. The results were very similar. After processing the first five configurations, the resulting atlas sizes became better than for the input files. Then the next progress brought slightly better improvements, although not very distinguishable. For example, a difference of the atlas quality between the 10th and the 113th configuration was only 2% in terms of texture memory and about 1% in terms of files size.

7 Conclusion and Future Work

We have implemented a method that combines a set of texture images into a texture atlas. In most cases, this technique decreases the overall file sizes. Both lower size and lower number of files have a positive effect on data transfer over Internet.

Algorithm works well for arbitrary-shaped textures, with normal size (big and small together) or small size textures. If there are more large textures than the small ones, or textures with rectangular shape, the placing algorithm leaves more empty space (it does not have any small textures to be placed to empty spaces).

The content of the resulting texture atlas is still far from the optimum. We do not have the best solution and from the first look at our texture atlases, it is obvious that some polygons should lie at another place. In order to minimize unused space, we need to improve our containment strategy. One possibility is to increase the number of iterations in the searching process. We are also going to extend our concept with a rotation operator. Since smooth rotation of a polygonal shape would cause troubles with texture coordinates remapping and image quality degradation, we want to apply only rotations by multiples of $\pi/2$.

Acknowledgments

I want to thank my professor, Jiří Žára, from Czech Technical University in Prague. I also thank VRVis Center in Graz, Technical University in Graz, and Maribor University of Technology for providing test data.

References

[1] T.M. Cavalier, R.B. Grinde . *A New Algorithm for the Two-Polygon Containment Problem*. Computers and Operations Research 24(3):231-251 ,1997.

[2] H. Cohn: *Advanced Number Theory*. Dover Publications, Inc., New York, 1980.

[3] B. Lévy, S. Petitjean, N. Ray, J. Maillot. *Least Squares Conformal Maps for Automatic Texture Atlas Generation*. In SIGGRAPH 02 Conf.Proc., pages 362-371, ACM Press, 2002.

[4] N. Marques, P. Capela, J. Bernardo: *Heuristic Reasoning for 2D Containment Problems*. In: WSCG'99, pages 180-185 (Volume I), Pilsen: University of West Bohemia, 1999.

[5] N. Marques, P. Capela, J. Bernardo: *Solving Multiple Layer Containment Problems Using Iterative Methods*. In: WSCG'00, pages 204-211 (Volume II). Pilsen: University of West Bohemia, 2000.

[6] P. Sander, J. Snyder, S. Gortler, H. Hoppe. *Texture mapping progressive meshes*. In SIGGRAPH 01 Conf. Proc., pages 409-416, ACM Press, 2001.