Raster Set Representations with Efficient JOIN Operator

Josef Pelikán

Charles University in Prague
Department of Software and Computer Science Education (KSVI)
Josef.Pelikan@mff.cuni.cz, http://sun3.ms.mff.cuni.cz/~pepca/

Abstract

Some 3D computer graphics algorithms are using front-to-back scene traversal (for example hidden-line removal algorithms or fast form-factor computation). In such methods each pixel has to be processed at most once. For this purpose various raster set representations can be used.

This paper gives a survey of selected data structures for raster set representation. Some new non-symmetrical methods useful for scanline processing of 2D objects are described. Efficiency of DIFFERENCE and UNION operators (and compound JOIN operator) in typical rendering environment is our key point of view. Quantitative results measured in different computer architectures are included.

1 Motivation

There are several computer graphics algorithms where compact and efficient raster set representation is wanted. We can point out methods using front-to-back traversals of 3D scene in the case that multiple processing of a single pixel cannot be accepted or shape algebras used in window systems and graphics libraries. Generally all methods with high pixel-processing cost are relevant.

Applications in hidden surface removal are described in [Foley90] and [Hrúz96], efficient form-factor computation based on UNION set operator can be found in [Nechvíle97]. A little different situation is also mentioned in [Steinhart91], where a coherency between adjacent scanlines plays an important role.

An immediate stimulation for this study was lack of any *practical* information about data structure efficiency in a typical computing/rendering environment. One can read about a number of smart data structures in classical literature (eg. [Preparata85] or [Samet89]) but often a theoretical complexity doesn't tell well about the real benefit of the method (example: almost every searching algorithm has $\mathcal{O}(\log_2 N)$ complexity but real gains of individual methods can vary a lot). Using only theoretical results, there is little chance to find the most suitable data representation for a given problem, especially in context of modern computer architectures with memory cache hierarchies and advanced CPU pipelines.

2 Basic concepts

In 2D raster graphics generally, *"set"* stands for a subset of some regular 2D lattice. In this text we are dealing with integer numbers neglecting other geometric properties of a lattice: term *"1D set"* is used for a subset of $\mathcal{Z}$, *"2D set"* denominates a subset of $\mathcal{Z}^2$.

Members of such sets are called *"pixels"*, processing units (graphical primitives) are called *"objects"*. Objects use to be compact and can be successfully surrounded by *"minimal bounding rectangles"*. In most cases objects are planar polygons, sometimes they are limited to triangles.

2.1 Set operators

Although the whole suite of *set operators* is utilized in some applications, we will focus our attention on operators used in front-to-back traversals. We consider situations where each pixel must not be processed more than once:

Representing an already-processed-region of pixels R we want to know which parts of a next graphical object A_i fall outside that region. So we need fast DIFFERENCE operator on 2D sets ($A_i \setminus R$).

After actual object processing we need the UNION set operator to update the region R ($R \Leftarrow R \cup A_i$).

The efficiency of these two operators appears to be critical for the whole algorithm (see results in section 6). Though we introduce a compound JOIN operator which combines the functionality of DIFFERENCE and UNION operators. At once it computes the difference $A_i \setminus R$ (pixels to be processed) and updates the region $R \Leftarrow R \cup A_i$. Formally we will write $JOIN(R, A_i)$.

2.2 RECTANGLE-TEST, SEGMENT-TEST

In order to eliminate as many objects as possible in early stages of processing, a coarse RECTANGLE-TEST is performed. For the given region R and the actual object A_i the test should return TRUE if and only if $B(A_i) \setminus R \neq \emptyset$, where $B(A_i)$ stands for the minimal bounding rectangle of A_i.

The third return value PAINT-ALL might be useful in some situations. It is returned when $R \cap B(A_i)$ is empty. In that case the whole object A_i can be processed without any tests.

Doing RECTANGLE-TEST is reasonable when it runs much faster than processing of typical non-contributing polygon and when expected number of trivially rejected polygons is high.

There are data structures with amortized complexity of $\mathcal{O}(N)$ for sequences of N tests (that means constant average time for an individual test).

We will often use an *separable* 2D set representation (which consists of an horizontal array of vertical 1D structures or an vertical array of horizontal ones). Then our RECTANGLE-TEST will be in form of n vertical SEGMENT-TESTs or m horizontal SEGMENT-TESTs.

2.3 Scanline systems

As we mentioned earlier, objects are planar polygons in most cases. *"Scanline algorithms"* are efficient methods for enumerating pixels inside a polygon [Foley90, Wallis90]. There are two main concepts:

1. maintaining an *"active edge list"* and generating segments scanline by scanline ("scanline sweep")
2. using a temporary storage for scan converted area and processing polygon edges one after another

In both cases one direction (mainly the horizontal one) is preferred because processing of line segments in this direction is very fast. The choice of *horizontal scanlines* is meaningful in context of real frame-buffer architectures and CPU instruction sets.

In this paper we concentrate on *non-symmetrical* 2D data structures (i.e. separable structures with different methods used for horizontal and vertical directions). First we will give a survey of *1D data structures* and then some *composite* 2D data structures are described.

The practical point of view plays an important role here, so many measurements on real implementations are included in last sections.

3 1D set representations

Raster set representations in 1D are used for representation of any subset $S \subseteq \mathcal{Z}$. Some methods can represent only finite subsets or have even stronger restrictions (for example existence of fixed finite domain: $S \subseteq T \subset \mathcal{Z}$ for $|T| < \infty$). Real subsets often consist of relatively small number of disjoint intervals.

In next sections K will stand for an upper bound of interval size and N for an actual number of represented intervals.

3.1 Bit array

Bit array (bit mask) is the most simple set representation with $\mathcal{O}(K)$ memory requirement and direct member access ($\mathcal{O}(1)$). Real implementations can use words of different bit-widths w (8, 16, 32 or 64 in practice). In many situations w plays an important role (example: implementation of $\min(S)$ – looking for the minimal member of the set S).

An straightforward implementation of $\mathsf{JOIN}(R, A)$ needs $\mathcal{O}(\lceil d(A)/w \rceil)$ operations where $d(A) = \max(A) - \min(A)$ is the *diameter* of the set A.

3.2 X-transition list

This data structure is described in [Wallis90], similar method is used in [Steinhart91]. We store information about points where the characteristic function $\chi(A)$ changes (i.e. endpoints of all intervals). These points are arranged in a sorted list, no other information is needed (the INSIDE-TEST is realized by the "odd-even" rule).

Formally we can write *generalized X-transition list* representing the set A as a ordered sequence

$$-\infty \leq x_1 < x_2 < \cdots < x_n \leq \infty, \text{ where } x_i \in \mathcal{Z}$$

while $x \in A \Leftrightarrow x_{2k+1} \leq x < x_{2k+2}$ for any $0 \leq k < n/2$.

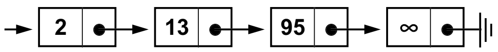

Figure 1: X-transition list for $\langle 2, 12) \cup \langle 95, \infty)$

The INSIDE-TEST can be done in $\mathcal{O}(2N)$ time, any binary set operation (including JOIN) takes $\mathcal{O}(2(N_1+N_2))$ time. A general set-operation algorithm can be formulated – its principle is similar to one of merge-sort.

3.3 Segment list

Segment list is slight modification of X-transition list. Every item of a segment list represents one disjoint segment of the set A. This method gives more compact data representation and faster JOIN operator implementation ($\mathcal{O}(N_1 + N_2)$). A little more complicated set-operation routine is drawback of this approach.

3.4 Segment search tree

To reduce complexity of the INSIDE-TEST, any type of search tree can be build above a segment list. Then segment list items become nodes of a search tree as well.

If we are updating a tree by the JOIN operator, only three elementar actions are used:

1. one of existing segments has to be expanded (to the left and/or to the right): no topological change of the search tree is done.
2. a new disjoint segment is added: classical *insert operation* in the search tree is performed.
3. two existing segments merge: the segment stored lower in the tree is deleted, the higher one is updated.

3.5 Simple segment tree

A data structure "segment tree" for 1D set storage is described in [Preparata85, Samet89]. Here we are using a simplified version of this data structure – only one set of disjoint intervals is stored.

A complete binary tree gives a hierarchical representation of a raster universe. Leaf nodes correspond to individual pixels, inner nodes represent intervals of length 2^k and finally the root node represents the whole universe.

Only one binary value is stored in every node – if it is TRUE, the whole segment is covered by the set A no matter of values stored in descendant nodes. If the bit has value FALSE, the segment is covered according to node descendants.

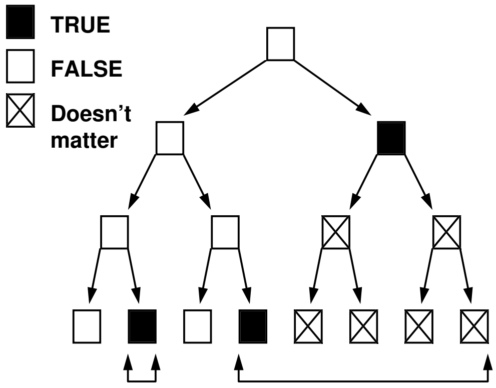

Figure 2: Simple segment tree representing the set $\{2\} \cup \langle 4, 8)$ from $\langle 1, 8)$ universe

The complexity of JOIN(R, S) operation is $\mathcal{O}(2 \log_2 |R|)$ for *single segment* S (only two tree paths at segment endpoints have to be traversed).

3.6 Bit pyramid

Bit pyramid is an alternative storage method of the simple segment tree: the complete binary tree is stored in a bit array B_i using heap-like organization. Bit node B_i has two descendants B_{2i} and B_{2i+1} and one ancestor $B_{\lfloor i/2 \rfloor}$ (only for $i > 1$).

The complexity of JOIN(R, S) operation is $\mathcal{O}(2 \log_2 d(S))$ for *single segment* S (only two *partial* tree paths have to be traversed).

3.7 Agent representation

An self-optimizing data structure for very fast SEGMENT-TESTs is based on idea from the well known factor set representation ([Aho74]). Every pixel has a pointer to some preceding pixel (*the agent*) in the same segment. The first pixel in a segment plays a role of a *master agent*, it points to itself (see Figure 3).

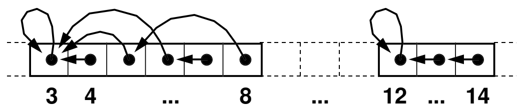

Figure 3: Agent representation of the set $\langle 3, 8) \cup \langle 12, 14)$

Taken any two pixels we can simply follow their agent-chains to determine if they belong to the same segment. If we get the same master agent, the pixels are lying in the same segment.

An improvement is applicable in 1D case – in SEGMENT-TEST we need not pass through both agent-chains. We will only pass the right one and stop in situation when our actual pointer jumps over left tested pixel. In that case we are sure that both pixels are lying in the same segment.

The factor set representation is often implemented in self-optimizing way: while passing through agent-chains we can re-assign pointers and thus make some agent-subchains shorter (see Figure 4).

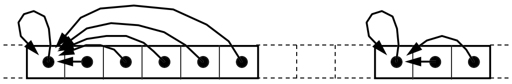

Figure 4: Totally optimized set $\langle 3, 8\rangle \cup \langle 12, 14\rangle$

Let's assume the agent-chain in the form

$$p_{x_0} \to p_{x_1} \to \cdots \to p_{x_k} \to A$$

where p_x is pointer from cell x, $p_x \to p_y$ is equivalent to $p_x \to y$ and A is the master agent ($p_A \to A$). We were testing three self-optimizing strategies:

First – optimize only the first pointer of the chain: $p_{x_0} \to A$

Shift – optimize the chain locally: $p_{x_0} = p_{x_{k-1}} = p_{x_k} \to A$, for other pointers: $p_{x_i} \to p_{x_{i+2}}$

All – optimize the whole chain: $p_{x_0} = p_{x_1} = \cdots = p_{x_k} \to A$

No major difference in performance of these strategies was perceived, so we decided to use only the simplest one – **First**.

JOIN(R, S) operation (Figure 5) can be performed in $\mathcal{O}(|S|)$ time. When JOIN operations are mixed with SEGMENT-TEST operations and number of later ones is much higher, there is a chance to become data structure with average $\mathcal{O}(\log^* K)$ SEGMENT-TEST time (see [Aho74]).

4 Composite 2D data structures

Although data storage methods used in front-to-back processing (e.g. frame buffer) use to be organized in horizontal scanlines, we decided to take

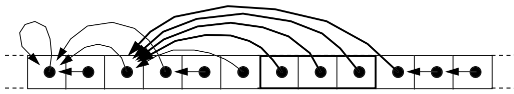

Figure 5: Unoptimized set $\langle 3, 14\rangle$ just after the JOIN operation

horizontal 1D data structures and put them together into a 2D data structure.

For example we can take number of horizontal *bit masks* to get 2D bit mask. Then we build a secondary (vertical) structure above each column of bit mask words. This secondary data structure is used for fast RECTANGLE-TESTs.

In fact we used only two types of horizontal data structures: bit masks and segment lists. Vertical data structures were chosen from the whole set of described methods.

Many other techniques have not been implemented yet. They are going to be tested in the future to get more complete results.

5 Test configurations

Four different computer architectures were used for our tests:

PC – PC equipped with processor: AMD 5x86 at 150MHz, RAM: 40MB, primary/secondary cache: 16KB/256KB, OS: Windows 95, compiler: MS Visual C++ 4.0

Sun – Sun SPARCclassic, microSPARC at 50MHz, 48MB, 6KB/0, Solaris 2.3, SPARCompiler 3.0.1

Sun5 – Sun SPARCstation 5 SX, SPARC, 32MB, Solaris 2.3, SPARCompiler 3.0.1

SGI – SGI Indy, R4600 at 133MHz, 64MB, 32KB/512KB, IRIX 5.3, SGI C++ 4.0

Each test run had other parameters: framebuffer resolution, type and attributes of a test scene, pixel processing method (processing goal), control algorithm and used data structure.

The *resolution* can be set to values between 512×512 and 8192×8192, in fact the maximal reasonable value was 4096×4096 (memory managers started to swap while computing higher resolutions).

Five main *scene types* were used:

Rnd*n* – randomly generated polygons with a given vertex number (n) and bounding rectangle (no self-intersection tests were done, so polygons could be non-simple: see Figure 6)

Eqn – randomly placed regular polygons of equal size and vertex number n (Figure 7)

Reg$n$ – randomly placed regular polygons with equal vertex number (n) but variable size

Sticks – randomly oriented long "sticks" with equal width, clipped to the window boundary (Figure 8)

Teapot$k$ – set of three "Utah teapots" with different positions and orientations (k defines the size of triangle mesh to approximate one Bèzier patch: see Figure 9)

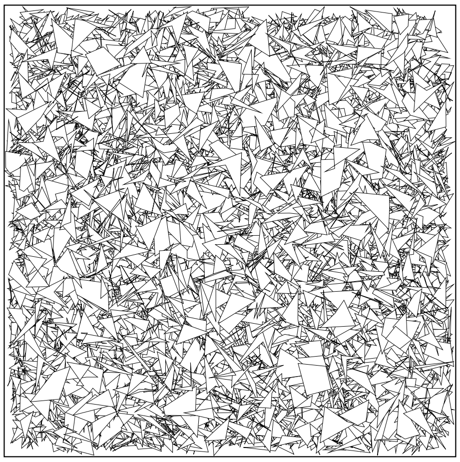

Figure 6: Test scene "Rnd4"

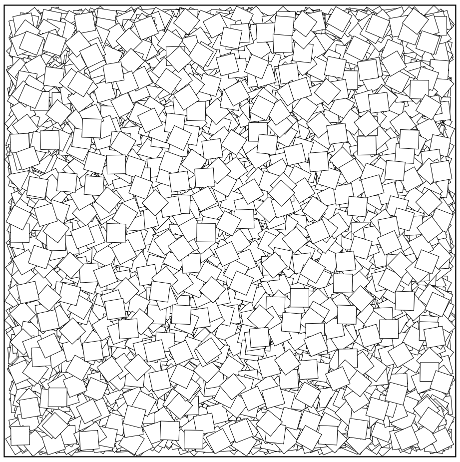

Figure 7: Test scene "Eq4"

Quantitative parameters of test scenes were: *the number of polygons* and maximum *polygon diameter*. So we will write *"Eq3-10k-100"* for scene arranged of 10000 equally sized triangles with the diameter of 100 pixels.

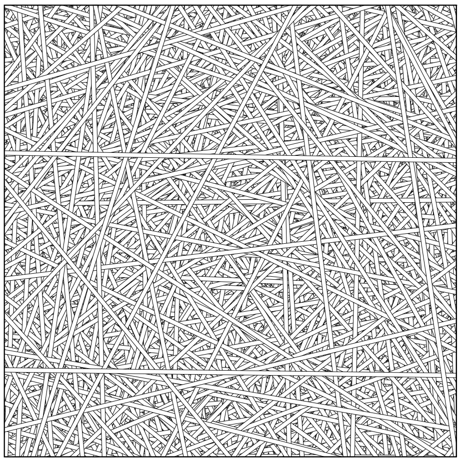

Figure 8: Test scene "Sticks"

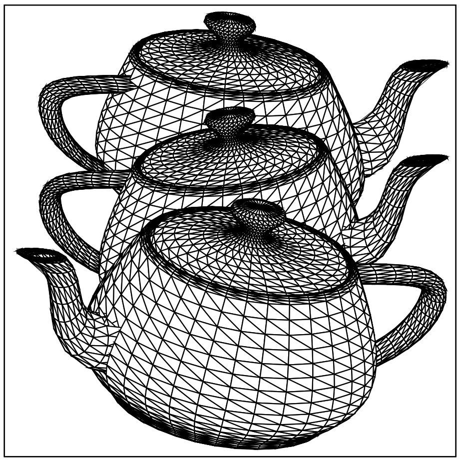

Figure 9: Test scene "Teapot8"

The previous two attributes have effect upon global *occlusion coefficient* O_* which can be defined as a ratio

$$O_* = \frac{(N_* - V_*)}{V_*}$$

where N_* is total number of processed items: polygons ($* = p$), h-lines ($* = h$) or pixels ($* = x$) and V_* is the number of (at least partially) visible ones. Value of the occlusion coefficient plays a role in algorithm efficiency – RECTANGLE-TESTs are more reasonable for higher O_* values.

Table 1 shows quantitative parameters of test scenes rendered in 2048×2048 resolution.

Selected algorithms and data structures described in previous sections were implemented and tested. Parameters of individual tests are described by shortcuts in form "X-Y" where X

Test scene	$N_p(O_p)$	$N_h(O_h)$	$N_x(O_x)$
Rnd4-15k-150	15k(1.2)	1.4m(4.5)	34m(7.8)
Eq4-10k-100	10k(2.2)	890k(5.6)	50m(11)
Eq16-8k-100	8k(2.9)	790k(6.5)	61m(14)
Reg4-12k-120	12k(1.6)	720k(3.5)	32m(7.1)
Sticks-3k-16	3k(1.4)	3.1m(9.4)	77m(17)
Teapot18	61k(1.2)	810k(1.5)	7.5m(2.0)

Table 1: Occlusion coefficients

denotes pixel processing procedure (goal) and Y stands for a control algorithm. Following goals were tested:

D – dummy (empty) routine to determine an overhead of the test program and polygon filling procedure

F – flat polygon shading in 8-bit color model

P – polygon fill using simple matrix-defined pattern and 8-bit color model

G – Gouraud polygon shading with dithering in multiple segment colormap (star topology). Color interpolation was done using integer arithmetic only

FF – form-factor computation on a single plane ([Cohen93, Sillion89]). Table of precomputer delta form-factor was used, only one quarter of a matrix was stored. 32-bit integer arithmetic was used

Control algorithms and data structures:

P – *painter's algorithm*: back-to-front scene traversal with repainting

Z* – visibility is determined using *Z-buffer algorithm*, "ZB" or "ZF" denote back-to-front or front-to-back processing order

B*n* – *bit-mask* organized in horizontal n-bit words ("*n*-bit-mask", *n* can be 8, 16 or 32). In this method and all following ones the *front-to-back* scene traversal was used

By – byte mask (this memory-consuming variant was implemented due to its simplicity but it appeared to be poor)

V*n* – *n*-bit-mask with additional data structure representing completely covered vertical segments. A vertical data structure was used for inexpensive RECTANGLE-TEST of processed polygon.

In this case *the length of a vertical segment* was stored for each column and y-coordinate.

A*n* – *n*-bit-mask: vertical covered segments in every column were represented by *"agent pointers"*

Y*n* – *n*-bit-mask: vertical covered segments in every column were represented by *segment lists*

SL – *segment list* for each scanline (y-coordinate)

SLT – *segment list* with RECTANGLE-TEST of every processed polygon

An example of the complete test description: *"PC/1024/Rnd4-20k-50/G-B32"* (PC architecture, 1024×1024 resolution, 20000 random quadrilaterals with 50×50 bounding rectangle, Gouraud shading, front-to-back traversal with bit-mask organized in 32-bit words, no RECTANGLE-TESTs).

6 Results

To examine performance of individual systems, *"Arch"* script addressing typical test tasks was created. As our results show, there is little difference in relative speed of individual test items – only Z-buffer tests are relatively slower on PC platform. Other tests were done in PC environment only – there is a good chance that global trends and observations are valid in all platforms.

The *Arch* test script consists of eleven items: D-P, F-P, F-ZF, G-P, G-B8, G-B32, G-A32, G-Y32, G-SLT, FF-ZF and FF-A32. Tables 2 and 3 show CPU times measured for ten test scenes "2048/Rnd4-15k-150" and twenty "2048/Teapot18".

	Architecture			
Test	**PC**	**Sun**	**Sun5**	**SGI**
D-P	20.9	98.8	44.3	19.9
F-P	37.8	148.4	62.3	38.8
F-ZF	122.6	231.1	118.3	75.9
G-P	95.4	264.8	140.7	79.2
G-B8	57.8	181.6	80.7	45.4
G-B32	54.2	159.2	73.9	42.5
G-A32	48.6	126.5	60.7	35.9
G-Y32	47.9	158.5	63.1	37.4
G-SLT	82.2	178.8	81.6	58.9
FF-ZF	139.5	263.5	141.0	85.5
FF-A32	55.6	131.6	65.1	35.4

Table 2: *Arch* test: 10× 2048/Rnd4-15k-150

Second test series show efficiency of JOIN operation in different *screen resolutions*. Ten random

Test	Architecture PC	Sun	Sun5	SGI
D-P	34.6	150.3	50.6	31.3
F-P	44.0	204.4	67.2	45.0
F-ZF	91.3	312.7	140.5	68.7
G-P	68.3	228.6	93.9	60.5
G-B8	68.6	223.4	87.1	58.5
G-B32	65.9	205.2	81.6	56.1
G-A32	57.4	160.4	66.9	45.5
G-Y32	54.5	179.3	66.4	45.6
G-SLT	49.7	163.7	66.6	42.7
FF-ZF	123.2	360.1	197.1	85.4
FF-A32	66.8	172.5	77.6	46.6

Table 3: *Arch* test: 20× 2048/Teapot18

test scenes *"Reg3"* were rendered using flat shading. Polygons (triangles) in that model had variable diameter: approximately from $X_{res}/100$ to $X_{res}/10$, average occlusion factors were $O_p \approx 1.3$ and $O_x \approx 7.1$.

The *resolution test* shows flexibility and resolution-independence of inspected data structures. The results are summarized in Table 4.

Test	Resolution 512	1024	2048	4096
F-P	4.1	8.9	24.4	77.7
F-B8	4.9	11.7	32.5	99.9
F-B16	5.0	11.7	32.0	96.5
F-B32	4.8	11.0	29.0	84.5
F-A8	4.3	10.4	29.8	94.2
F-A16	4.3	10.2	28.4	88.3
F-A32	4.3	9.5	26.3	79.4
F-Y8	4.2	10.0	28.8	95.5
F-Y16	4.2	9.8	27.2	86.4
F-Y32	4.3	**9.3**	**24.7**	**74.9**
F-SL	4.9	11.3	25.2	59.4
F-SLT	**4.0**	**9.6**	**21.8**	**52.3**
F-ZF	8.7	28.1	97.5	swap

Table 4: *Resolution* test: 10× Reg3

Segment list data structures (Yn and SL*) are more efficient in higher resolutions - they take benefit from a continuous topology and relatively high occlusion factors of 3D scenes. In practice, closer objects (faces) often merge into large regions which can occlude many farther objects. Large continuous areas are especially good for X-transition and segment lists because the $\mathcal{O}(N)$ searching term is suppressed.

The advantage of sparse region representations should even grow up when occlusion factor increases. To demonstrate this behavior we set up another test – scenes with different occlusion factors are compared. As in the "resolution" test we use Reg$n$ model with variable object diameter. Three groups of scenes were rendered, with average occlusion factors of 0.3, 1.2 and 4.0 respectively. The results are shown in Table 5 (note differences between the first line "F-P" and the other ones).

Test	O_p factor 0.3	1.2	4.0
F-P	10.5	26.4	66.1
F-B32	**20.4**	34.1	64.0
F-V32	36.4	79.0	99.2
F-A32	23.6	32.1	37.9
F-Y32	22.4	**30.7**	**35.5**
F-SL	**11.6**	32.5	53.8
F-SLT	11.8	**28.9**	**36.1**

Table 5: *Occlusion* test: 12× Reg4

The most important tests were done in comparing different methods and representations for fast JOIN operator in front-to-back scene traversals. The complete set of implemented algorithms was compared on several types of test scenes.

Here we present the results measured on four typical test scenes with 2048 × 2048 resolution:

1. **Rnd4-15k-150** scene model with average occlusion factor $O_p = 1.2$
2. **Reg4-12k-120** scene model with average occlusion factor $O_p = 1.6$
3. **Sticks-3k-16** scene model with average occlusion factor $O_p = 1.4$
4. **Teapot18** scene with occlusion factor $O_p = 1.2$

Average computing times of 10 test runs on the *PC architecture* are listed below in Table 6.

The simplest goal (flat shading) was used to minimize the processing overhead. The first line ("F-P" – no tests, back-to-front repainting) can be used for an overhead estimation.

Table 6 shows that the composite 2D data structures (agent pointers – "A32" or vertical segment lists – "Y32") are more efficient than the simple bit mask ("B32"). Another method with good results is *horizontal segment list* ("SLT") even if it has no secondary (vertical) structure.

The complete source code of the used programs is available at
`http://sun3.ms.mff.cuni.cz/~pepca/join/`

Test	Scene 1	2	3	4
F-P	37.7	21.2	37.7	22.0
F-By	86.2	54.0	90.1	36.5
F-B8	51.4	28.0	48.2	30.5
F-B16	51.1	27.7	47.0	31.0
F-B32	47.7	25.3	**43.6**	29.2
F-V32	69.4	54.4	44.7	30.5
F-A8	46.3	24.8	40.7	27.4
F-A16	44.5	23.9	38.1	26.5
F-A32	**42.0**	**22.2**	**35.5**	**24.8**
F-Y8	49.2	24.2	42.5	24.9
F-Y16	45.0	22.8	39.6	24.6
F-Y32	**41.4**	**21.0**	**35.5**	**23.5**
F-SL	45.4	22.6	51.1	26.8
F-SLT	**40.1**	**19.0**	44.7	**21.1**

Table 6: JOIN tests

together with original makefiles, test scripts and other result tables.

7 Conclusion

Many data structures and algorithms for 1D and 2D raster set representations are used. We were trying to give a survey of hybrid 2D structures, when scanline data structure used to be very efficient (bit mask) and a superior vertical structure makes RECTANGLE-TESTs as fast as possible.

We presented some experimental data and studied the real behaviour of the described algorithms. In the future we are going to extend our attention to other methods, especially to *"pure 2D ones"*. After that we will be able to give a complete comparison of raster set representation methods.

References

[Aho74] Aho A., Hopcroft J., Ullman J.: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974

[Cohen93] Cohen M., Wallace J.: *Radiosity and Realistic Image Synthesis*, Academic Press, Cambridge, MA, 1993

[Foley90] Foley J., van Dam A., Feiner S., Hughes J.: *Computer Graphics, Principles and Practice*, second edition, Addison-Wesley, Reading, MA, 1990

[Hrúz96] Hrúz T., Považan I., Gosiorovský R.: Hidden Line Problem Formulated as a Set Union Problem, In *WSCG'96 Proceedings*, Plzeň, 1996, pp. 246-256

[Nechvíle97] Nechvíle K.: The High Resolution Hemicube Algorithm, In *WSCG'97 Proceedings*, Plzeň, 1997, pp. 381-385

[Preparata85] Preparata F., Shamos M.: *Computational Geometry: An Introduction*, Springer-Verlag, New York, NY, 1985

[Recker90] Recker R., George D., Greenberg D.: Accelerating Techniques for Progressive Refinement Radiosity, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, vol. 24, No. 2 (March 1990), pp. 59-66

[Samet89] Samet H.: *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1989

[Sillion89] Sillion F., Puech C.: A General Two-pass Method Integrating Specular and Diffuse Reflection, *Computer Graphics (SIGGRAPH'89 Proceedings)*, vol. 23, No. 3 (July 1989), pp. 335-344

[Steinhart91] Steinhart J.: Scanline Coherent Shape Algebra, in *Graphics Gems II*, edited by J. Arvo, Academic Press, 1991, pp. 31-45

[Wallis90] Wallis B.: Fast Scan Conversion of Arbitrary Polygons, in *Graphics Gems I*, edited by A. Glassner, Academic Press, 1990, pp. 92-97