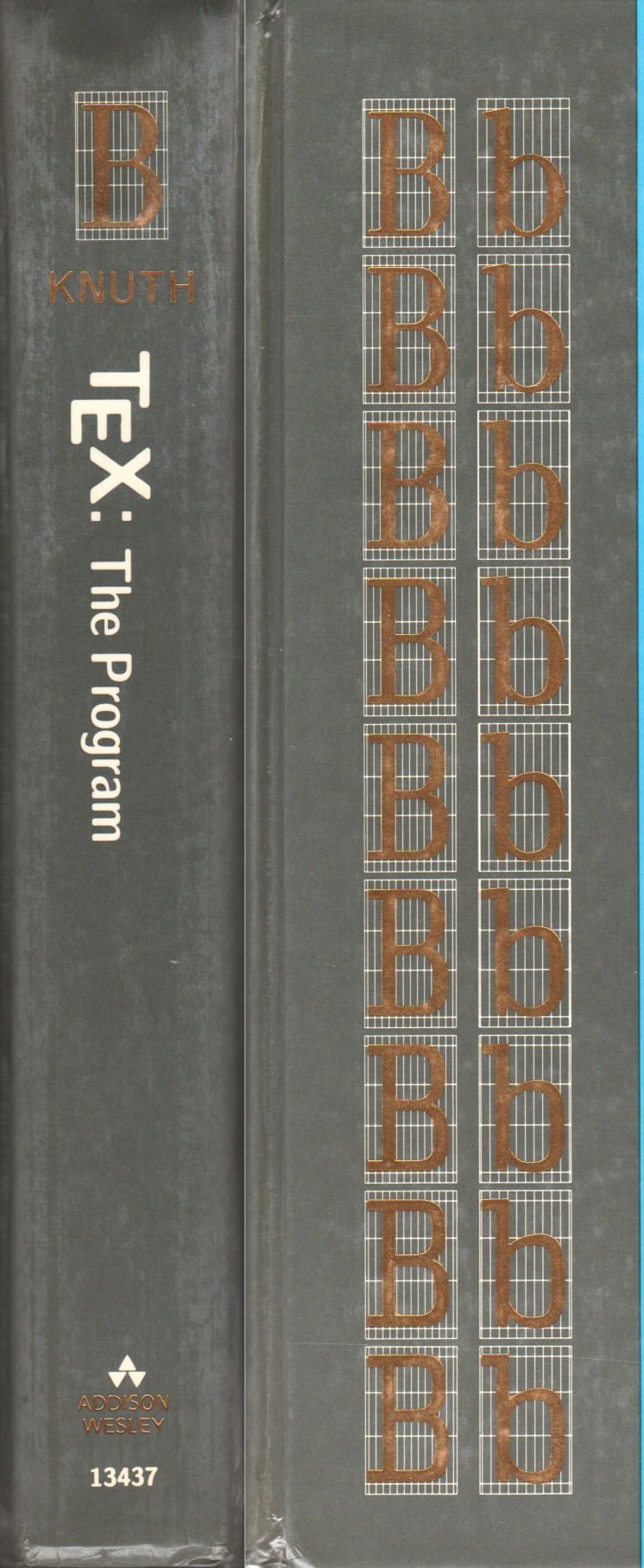
«Literate Programming»

Donald E. Knuth

```
⟨Emphatic declarations 1⟩;
  examples: array [vast] of small .. large; beauty; real;
⟨True confessions 10⟩;
for readers (human) do write (webs);
while programming = art do
    begin incr (pleasure); deer (bugs); incr (portability);
    incr (maintainability); incr (quality); incr (salary);
    end {happily ever after}
```

This code is used in theory and practice.



COMPUTERS & TYPESETTING

DONALD E. KNUTH

815. Since *line_break* is a rather lengthy procedure—sort of a small world unto itself—we must build it up little by little, somewhat more cautiously than we have done with the simpler procedures of TEX. Here is the general outline.

```
⟨ Declare subprocedures for line_break 826 ⟩
procedure line_break(final_widow_penalty : integer);
  label done, done1, done2, done3, done4, done5, continue;
  var ⟨ Local variables for line breaking 862 ⟩
  begin pack_begin_line ← mode_line;  { this is for over/underfull box messages }
  ⟨ Get ready to start line breaking 816 ⟩;
  ⟨ Find optimal breakpoints 863 ⟩;
  ⟨ Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and
      append them to the current vertical list 876 ⟩;
  ⟨ Clean up the memory by removing the break nodes 865 ⟩; pack_begin_line ← 0;
  end;
```

816. The first task is to move the list from *head* to *temp_head* and go into the enclosing semantic level. We also append the `\parfillskip` glue to the end of the paragraph, removing a space (or other glue node) if it was there, since spaces usually precede blank lines and instances of '`$$`'. The *par_fill_skip* is preceded by an infinite penalty, so it will never be considered as a potential breakpoint.

This code assumes that a *glue_node* and a *penalty_node* occupy the same number of *mem* words.

```
⟨ Get ready to start line breaking 816 ⟩ ≡
  link(temp_head) ← link(head);
  if is_char_node(tail) then tail_append(new_penalty(inf_penalty))
  else if type(tail) ≠ glue_node then tail_append(new_penalty(inf_penalty))
    else begin type(tail) ← penalty_node; delete_glue_ref(glue_ptr(tail));
      flush_node_list(leader_ptr(tail)); penalty(tail) ← inf_penalty;
      end;
  link(tail) ← new_param_glue(par_fill_skip_code); init_cur_lang ← prev_graf mod ´200000;
  init_l_hyf ← prev_graf div ´20000000; init_r_hyf ← (prev_graf div ´200000) mod ´100; pop_nest;
```

See also sections 827, 834, and 848.

This code is used in section 815.

817. When looking for optimal line breaks, TEX creates a "break node" for each break that is *feasible*, in the sense that there is a way to end a line at the given place without requiring any line to stretch more than a given tolerance. A break node is characterized by three things: the position of the break (which is a pointer to a *glue_node*, *math_node*, *penalty_node*, or *disc_node*); the ordinal number of the line that will follow this breakpoint; and the fitness classification of the line that has just ended, i.e., *tight_fit*, *decent_fit*, *loose_fit*, or *very_loose_fit*.

```
define tight_fit = 3  { fitness classification for lines shrinking 0.5 to 1.0 of their shrinkability }
define loose_fit = 1  { fitness classification for lines stretching 0.5 to 1.0 of their stretchability }
define very_loose_fit = 0  { fitness classification for lines stretching more than their stretchability }
define decent_fit = 2  { fitness classification for all other lines }
```

818. The algorithm essentially determines the best possible way to achieve each feasible combination of position, line, and fitness. Thus, it answers questions like, "What is the best way to break the opening part of the paragraph so that the fourth line is a tight line ending at such-and-

854. During the final pass, we dare not lose all active nodes, lest we lose touch with the line breaks already found. The code shown here makes sure that such a catastrophe does not happen, by permitting overfull boxes as a last resort. This particular part of TEX was a source of several subtle bugs before the correct program logic was finally discovered; readers who seek to "improve" TEX should therefore think thrice before daring to make any changes here.

```
⟨ Prepare to deactivate node r, and goto deactivate unless there is a reason to consider lines of text
      from r to cur_p 854 ⟩ ≡
  begin if final_pass ∧ (minimum_demerits = awful_bad) ∧ (link(r) = last_active) ∧ (prev_r = active)
        then artificial_demerits ← true   { set demerits zero, this break is forced }
  else if b > threshold then goto deactivate;
  node_r_stays_active ← false;
  end
```

This code is used in section 851.

855. When we get to this part of the code, the line from r to *cur_p* is feasible, its badness is b, and its fitness classification is *fit_class*. We don't want to make an active node for this break yet, but we will compute the total demerits and record them in the *minimal_demerits* array, if such a break is the current champion among all ways to get to *cur_p* in a given line-number class and fitness class.

```
⟨ Record a new feasible break 855 ⟩ ≡
  if artificial_demerits then d ← 0
  else ⟨ Compute the demerits, d, from r to cur_p 859 ⟩;
  stat if tracing_paragraphs > 0 then ⟨ Print a symbolic description of this feasible break 856 ⟩;
  tats
  d ← d + total_demerits(r);   { this is the minimum total demerits from the beginning to cur_p via r }
  if d ≤ minimal_demerits[fit_class] then
    begin minimal_demerits[fit_class] ← d; best_place[fit_class] ← break_node(r);
    best_pl_line[fit_class] ← l;
    if d < minimum_demerits then minimum_demerits ← d;
    end
```

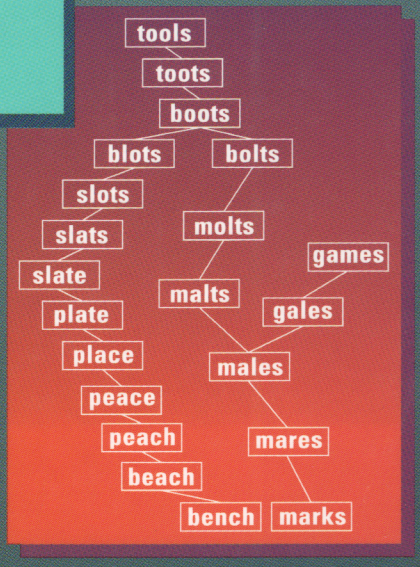
This code is used in section 851.

856.

```
⟨ Print a symbolic description of this feasible break 856 ⟩ ≡
  begin if printed_node ≠ cur_p then
    ⟨ Print the list between printed_node and cur_p, then set printed_node ← cur_p 857 ⟩;
  print_nl("@");
  if cur_p = null then print_esc("par")
  else if type(cur_p) ≠ glue_node then
      begin if type(cur_p) = penalty_node then print_esc("penalty")
      else if type(cur_p) = disc_node then print_esc("discretionary")
        else if type(cur_p) = kern_node then print_esc("kern")
          else print_esc("math");
      end;
  print("␣via␣@@");
  if break_node(r) = null then print_char("0")
  else print_int(serial(break_node(r)));
  print("␣b=");
```

The Stanford GraphBase

A Platform for Combinatorial Computing



DONALD E. KNUTH

17. So all we have to do is set up those magic tables. If uu is a uniform random integer between 0 and $2^{31}-1$, the index $k = uu \gg kk$ is a uniform random integer between 0 and $nn-1$, because of the relation between nn and kk. Once k is computed, the code above selects vertex k with probability $(p+1-(k \ll kk))/2^{31}$, where $p = magic\rightarrow prob$ and $magic$ is the kth element of the magic table; otherwise the code selects vertex $magic\rightarrow inx$. The trick is to set things up so that each vertex is selected with the proper overall probability.

Let's imagine that the given distribution vector has length nn, instead of n, by extending it if necessary with zeroes. Then the average entry among these nn integers is exactly $t = 2^{30}/nn$. If some entry, say entry i, exceeds t, there must be another entry that's less than t, say entry j. We can set the jth entry of the magic table so that its *prob* field selects vertex j with the correct probability, and so that its *inx* field equals i. Then we are selecting vertex i with a certain residual probability; so we subtract that residual from i's present probability, and repeat the process with vertex j eliminated. The average of the remaining entries is still t, so we can repeat this procedure until all remaining entries are exactly equal to t. The rest is easy.

During the calculation, we maintain two linked lists of $(prob, inx)$ pairs. The *hi* list contains entries with $prob > t$, and the *lo* list contains the rest. During this part of the computation we call these list elements 'nodes', and we use the field names *key* and *j* instead of *prob* and *inx*.

```
⟨ Private declarations 8 ⟩ +≡
  typedef struct node_struct {
    long key;      /* a numeric quantity */
    struct node_struct *link;      /* the next node on the list */
    long j;      /* a vertex number to be selected with probability key/2^30 */
  } node;
  static Area temp_nodes;      /* nodes will be allocated in this area */
  static node *base_node;      /* beginning of a block of nodes */
```

18.
```
⟨ Internal functions 18 ⟩ ≡
  static magic_entry *walker(n, nn, dist, g)
      long n;      /* length of dist vector */
      long nn;      /* 2^⌈lg n⌉ */
      register long *dist;      /* start of distribution table, which sums to 2^30 */
      Graph *g;      /* tables will be allocated for this graph's vertices */
  { magic_entry *table;      /* this will be the magic table we compute */
    long t;      /* average key value */
    node *hi = Λ, *lo = Λ;      /* nodes not yet included in magic table */
    register node *p, *q;      /* pointer variables for list manipulation */

    base_node = gb_typed_alloc(nn, node, temp_nodes);
    table = gb_typed_alloc(nn, magic_entry, g→aux_data);
    if (¬gb_trouble_code) {
      ⟨ Initialize the hi and lo lists 19 ⟩;
      while (hi) ⟨ Remove a lo element and match it with a hi element; deduct the residual
          probability from that hi element 20 ⟩;
      while (lo) ⟨ Remove a lo element of key value t 21 ⟩;
    }
    gb_free(temp_nodes);
    return table;      /* if gb_trouble_code is nonzero, the table is empty */
```

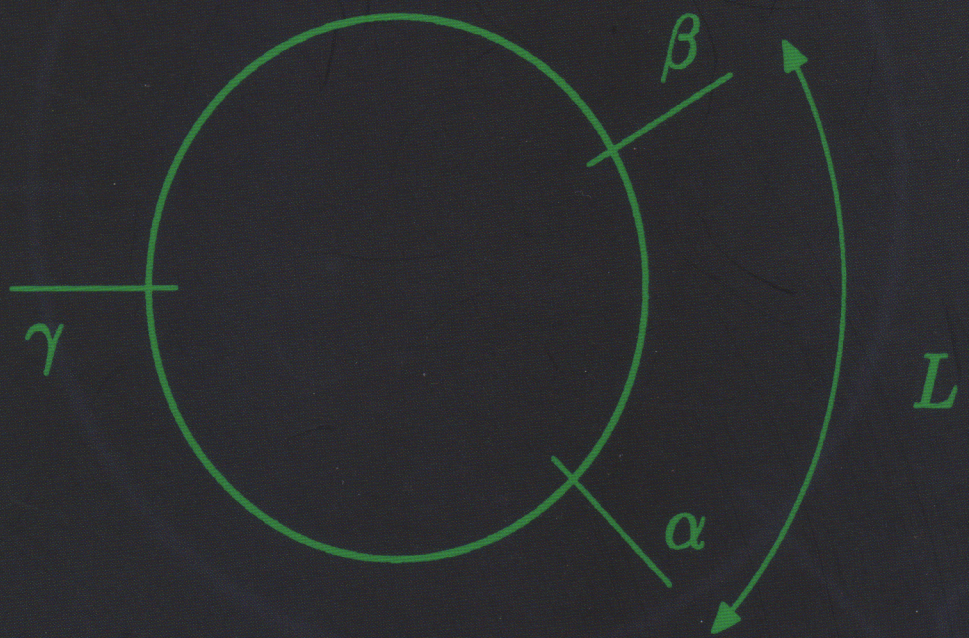
Donald E. Knuth

Tutorial

LNCS 1750

MMIXware

A RISC Computer for the Third Millennium



Springer

Division. Long division of an unsigned 128-bit integer by an unsigned 64-bit integer is, of course, one of the most challenging routines needed for `MMIX` arithmetic. The following program, based on Algorithm 4.3.1D of *Seminumerical Algorithms*, computes octabytes q and r such that $(2^{64}x + y) = qz + r$ and $0 \le r < z$, given octabytes x, y, and z, assuming that $x < z$. (If $x \ge z$, it simply sets $q = x$ and $r = y$.) The quotient q is returned by the subroutine; the remainder r is stored in *aux*.

```
⟨ Subroutines 5 ⟩ +≡
octa odiv ARGS((octa, octa, octa));
octa odiv(x, y, z)
    octa x, y, z;

  register int i, j, k, n, d;
  tetra u[8], v[4], q[4], mask, qhat, rhat, vh, vmh;
  register tetra t;
  octa acc;
  ⟨ Check that x < z; otherwise give trivial answer 14 ⟩;
  ⟨ Unpack the dividend and divisor to u and v 15 ⟩;
  ⟨ Determine the number of significant places n in the divisor v 16 ⟩;
  ⟨ Normalize the divisor 17 ⟩;
  for (j = 3; j ≥ 0; j--) ⟨ Determine the quotient digit q[j] 20 ⟩;
  ⟨ Unnormalize the remainder 18 ⟩;
  ⟨ Pack q and u to acc and aux 19 ⟩;
  return acc;
```

```
⟨ Check that x < z; otherwise give trivial answer 14 ⟩ ≡
if (x.h > z.h ∨ (x.h ≡ z.h ∧ x.l ≥ z.l)) {
  aux = y; return x;
}
```

This code is used in section 13.

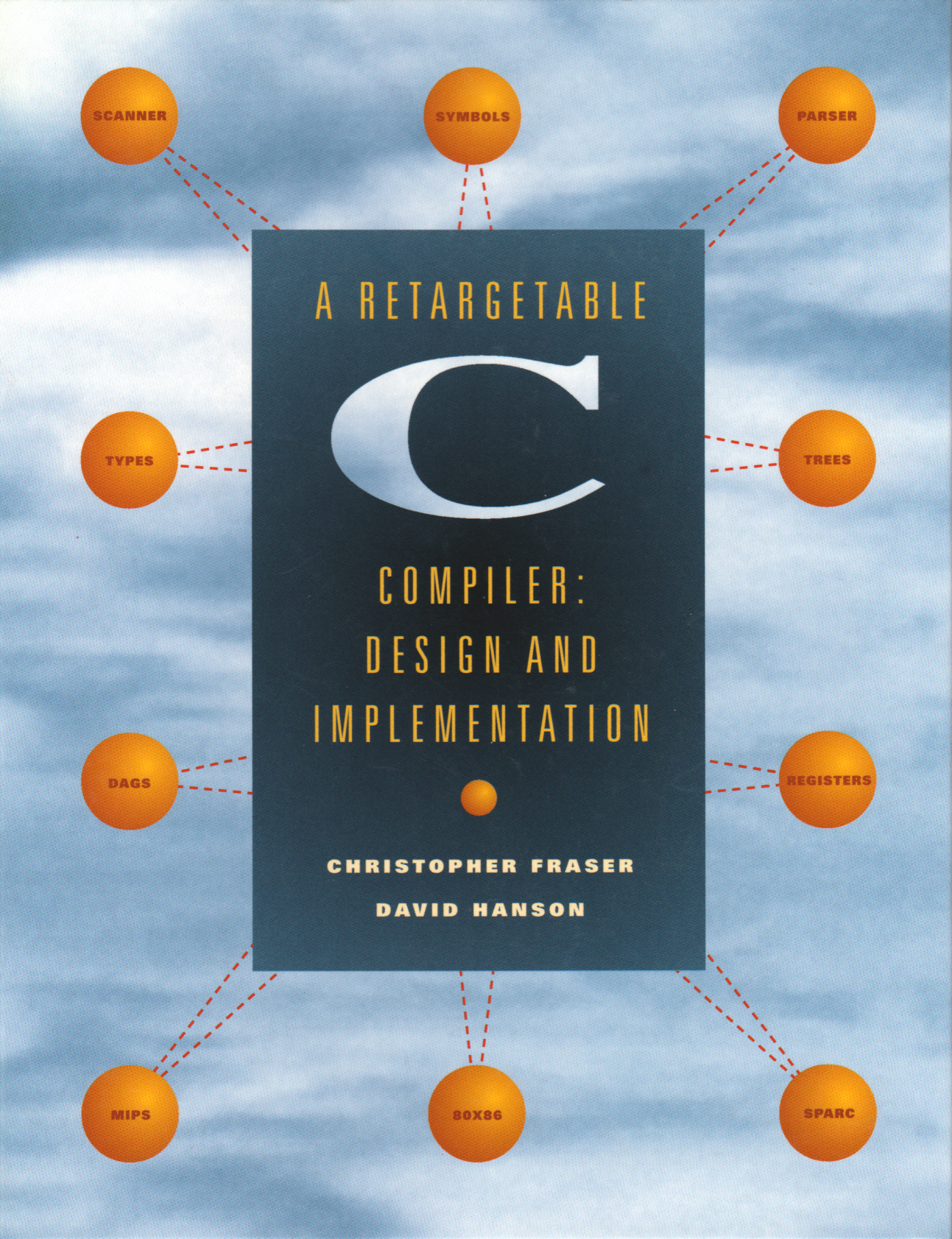
```
⟨ Unpack the dividend and divisor to u and v 15 ⟩ ≡
u[7] = x.h ≫ 16, u[6] = x.h & #ffff, u[5] = x.l ≫ 16, u[4] = x.l & #ffff;
u[3] = y.h ≫ 16, u[2] = y.h & #ffff, u[1] = y.l ≫ 16, u[0] = y.l & #ffff;
v[3] = z.h ≫ 16, v[2] = z.h & #ffff, v[1] = z.l ≫ 16, v[0] = z.l & #ffff;
```

This code is used in section 13.

```
⟨ Determine the number of significant places n in the divisor v 16 ⟩ ≡
for (n = 4; v[n − 1] ≡ 0; n--) ;
```

This code is used in section 13.

ARGS = macro ( ), §2.
octa = **struct**, §3.
overflow: **bool**, §9.
octa, §9.
ominus: **octa** ( ), §5.
sign_bit = macro, §4.
tetra, §3.
omult: **octa** ( ), §8.
tetra = **unsigned int**, §3.
tetra, §3.



A RETARGETABLE C COMPILER: DESIGN AND IMPLEMENTATION

CHRISTOPHER FRASER

DAVID HANSON

binary-expression uses the techniques described in Section 8.2 to handle all the binary operators, which have precedences between 4 and 13 inclusive (see Table 8.2).

Each of these functions parses the applicable expression, builds a tree to represent the expression, type-checks the tree, and returns it. Three arrays, each indexed by token code, guide the operation of these functions. `prec[t]`, mentioned in Section 8.2, gives the precedence of the operator denoted by token code `t`. `oper[t]` is the generic tree operator that corresponds to token `t`, and `optree[t]` points to a function that builds a tree for the operator denoted by `t`. For example, `prec['+']` is 12, `oper['+']` is `ADD`, and `optree['+']` is `addtree`, which, like most of the functions referred to by `optree` and like `optree` itself, is in `enode.c`. `prec` and `oper` are defined by including `token.h` and extracting its third and fourth columns:

⟨*tree.c data*⟩+≡ 150 169

```
static char prec[] = {
#define xx(a,b,c,d,e,f,g) c,
#define yy(a,b,c,d,e,f,g) c,
#include "token.h"
};
static int oper[] = {
#define xx(a,b,c,d,e,f,g) d,
#define yy(a,b,c,d,e,f,g) d,
#include "token.h"
};
```

`token.h` is described in Section 6.2.

Each function is derived using the rules described in Section 7.5. Code to build and check the trees is interleaved with the parsing code. The code for *expression* is typical and is also the simplest:

⟨*tree.c functions*⟩+≡ 150 156

```
Tree expr(tok) int tok; {
    static char stop[] = { IF, ID, '}', 0 };
    Tree p = expr1(0);

    while (t == ',') {
        Tree q;
        t = gettok();
        q = pointer(expr1(0));
        p = tree(RIGHT, q->type, root(value(p)), q);
    }
    ⟨test for correct termination 156⟩
    return p;
}
```

192 addtree
157 expr1
191 optree
174 pointer
149 RIGHT
109 token.h
150 tree
160 value

Martin Ruckert

Understanding MP3

- Syntax
- Semantics
- Mathematics
- Algorithms

vieweg

after the last valid byte that we may take out of the *buffer* and may possibly point even past the end of the buffer itself.

⟨ stream data 2 ⟩ +≡ (151)
unsigned char **start*;
unsigned char **finish*;

⟨ initialize *s* 25 ⟩ +≡ (152)
s→*start* = *s*→*buffer*;
s→*finish* = *s*→*buffer*;

Further, we want to know what is the current *buffer_position*, that is the position of the first byte in the buffer counted from the beginning of the stream starting with 0.

⟨ stream data 2 ⟩ +≡ (153)
int *buffer_position*;

The *buffer_position* makes it possible to convert relative positions inside the *buffer* into absolute byte positions inside the whole stream. For example, the position where the current frame starts is in the variable *frame*.

⟨ stream data 2 ⟩ +≡ (154)
unsigned char **frame*;

The value of *frame* can be converted to the *frame_position*, the absolute position of the frame in the stream.

⟨ derive further information 65 ⟩ +≡ (155)
s→*info*.*frame_position* = *s*→*buffer_position* + (*s*→*frame* − *s*→*buffer*);

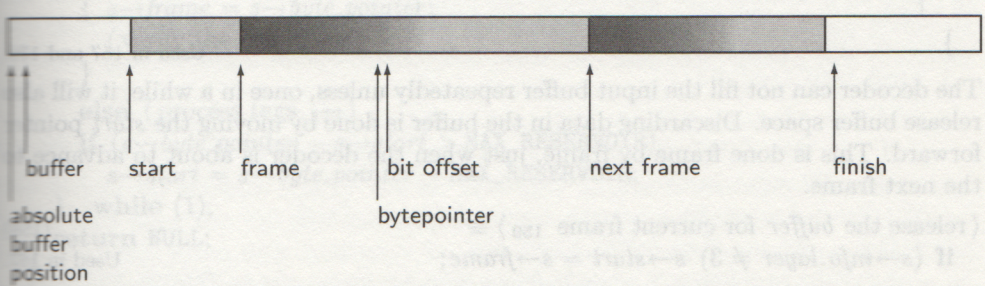


Fig. 19: Access to the input buffer

The *frame* variable was used already much earlier to ⟨ prepare the frame for decoding 62 ⟩. There, we had to

⟨ position the stream past the header 156 ⟩ ≡ (156)
s→*byte_pointer* = *s*→*frame* + HEADER_SIZE;
s→*bit_offset* = 0; Used in 62.

This is quite simple using *frame*.

But now it is time to see how the input data actually gets into the *buffer*. To fill the input *buffer*, we determine how many bytes are still in the *buffer* and move them

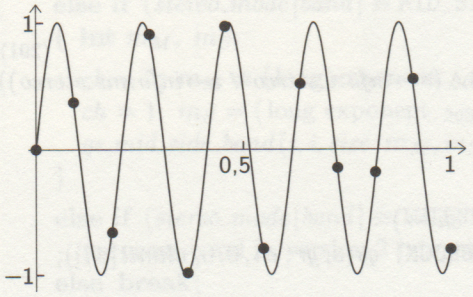


Fig. 55: High frequency output

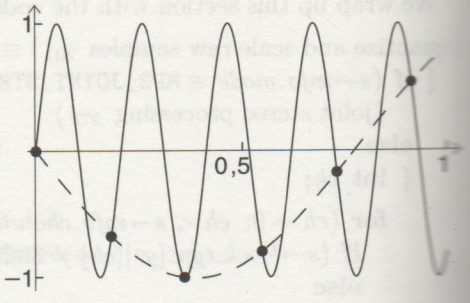


Fig. 56: Reduced sample rate and alias

The same effect can also be observed in the opposite direction: a low frequency having a higher frequency as its alias.

The aliasing effect, which is not caused by the limited quality of filters, but by the reduction of the sample rate, is usually not a problem. The misinterpretation of a frequency as its alias can only occur if the input of the frequency analysis contains both, the frequency and its alias.

This is the situation, we encounter, when each of the 32 subbands is split into 18 separate frequencies during layer III processing. As can be seen from Figure 27, the filters used for layer I subband filtering do overlap considerably. A frequency that belongs to the upper/lower half of one subband is also present in the output of the next/previous subband, and will be taken there for a higher/lower frequency, its alias.

Reducing this aliasing improves the coding efficiency of the MPEG encoder. A closer study of the situation reveals that each frequency and its alias frequency happen to be symmetric to the boundary between subbands. For example, within the first subband (frequencies 0 to 17) and the second subband (frequencies 18 to 35) the alias of frequency 17 will be frequency 18, 16 will pair up with 19, 15 with 21, and so on. In the encoder, the effect of aliasing is reduced, by computing from each frequency an estimate of its alias and compensating the appropriate alias frequency. This process is reversed in the decoder. The standard specifies the exact factors to be used to compensate for aliasing. So, for a frequency and its alias, we load the two values, separate them into the estimated original and the alias, using two constants c and d, compensate each value for the alias, and write the new values back.

```
⟨ apply alias reduction 292 ⟩ ≡
  ⟨ determine sblimit 293 ⟩
  if (s→block_type[gr][ch] ≠ SHORT_BLOCK ∨ s→mixed_block[gr][ch])
  { int k;
    if (s→block_type[gr][ch] ≡ SHORT_BLOCK ∧ s→mixed_block[gr][ch]) k = 1;
    else k = SUBBANDS − 1;
    ⟨ increase sblimit if needed 294 ⟩
    if (k > s→sblimit[ch] − 1) k = s→sblimit[ch] − 1;
```

```
for ( ; k > 0; k--)
{ int i;
  double *zHi = z[ch] + k * SUBFREQUENCIES;
  double *zLo = zHi - 1;
  for (i = 0; i < 8; i++, zLo--, zHi++)
  { double zHi, zLo, c, d;
    zLo = *zLo;
    zHi = *zHi;
    c = alias_coefficients[i].c;
    d = alias_coefficients[i].d;
    *zLo = zLo * c - zHi * d;
    *zHi = zHi * c + zLo * d;
  }
}
```

Used in 298.

We do not need to perform alias reduction for all subbands, but only for those subbands that are not zero. The number of subbands that contain non zero values is stored in the variable *sblimit* and is computed by rounding up *ulimit* to the next multiple of 18 and dividing by 18. Since the boundaries of layer III bands do not coincide with the subband boundaries, we complete the initialization for the last subband.

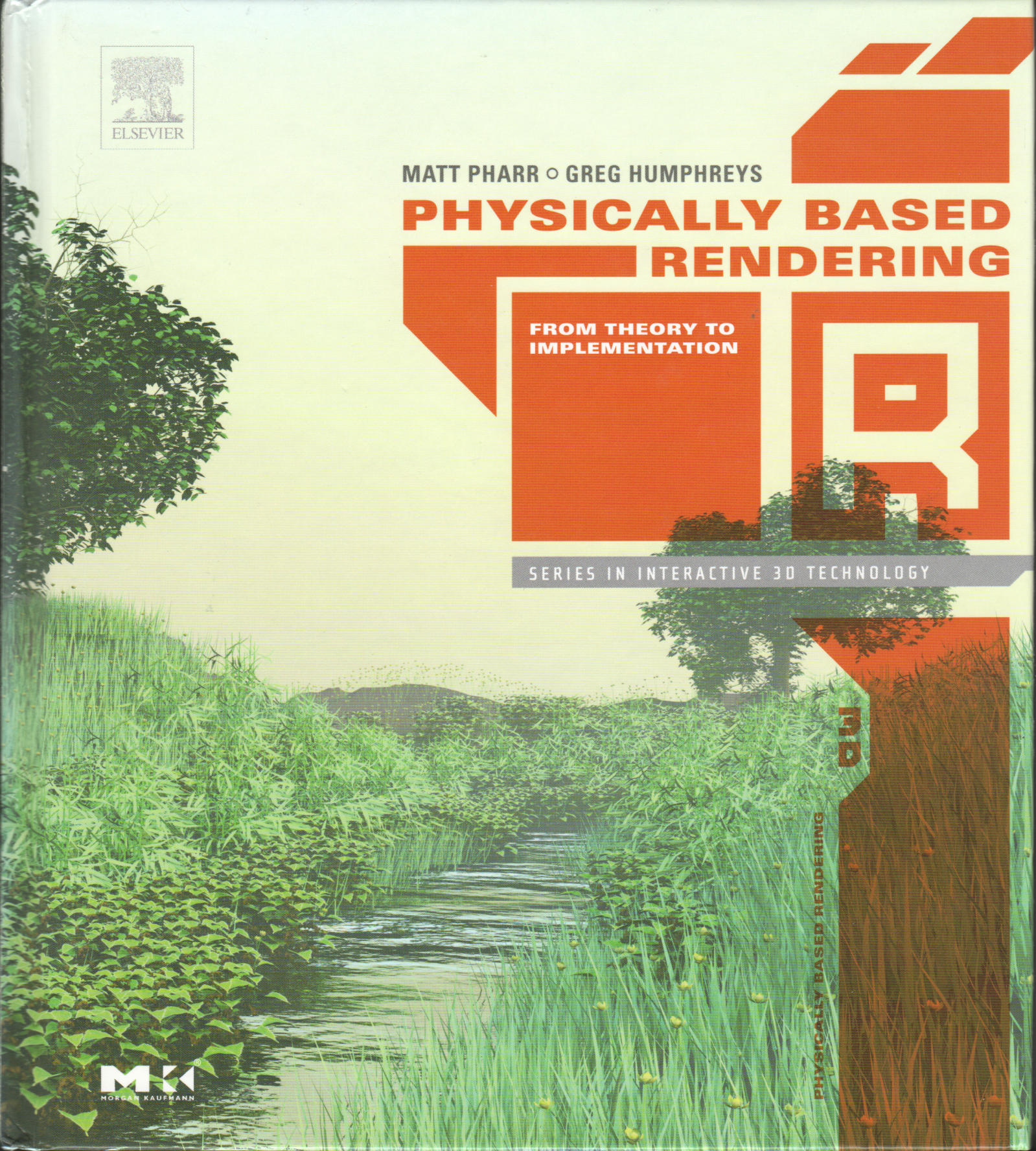
```
⟨ determine sblimit 293 ⟩ ≡                                    (293)
  s→sblimit[ch] = (ulimit[ch] + SUBFREQUENCIES - 1)/SUBFREQUENCIES;
  { int i;
    for (i = ulimit[ch]; i < s→sblimit[ch] * SUBFREQUENCIES; i++) z[ch][i] = 0.0;
  }
```

Used in 292.

Due to alias reduction, nonzero values may spread into the next subband. Therefore, after we perform alias reduction, we have to

```
⟨ increase sblimit if needed 294 ⟩ ≡                            (294)
  { if (s→sblimit[ch] < SUBBANDS ∧
        ulimit[ch] > s→sblimit[ch] * SUBFREQUENCIES + 1 - SUBFREQUENCIES/2)
    { int i;
      for (i = 0; i < SUBFREQUENCIES; i++)
        z[ch][s→sblimit[ch] * SUBFREQUENCIES + i] = 0.0;
      s→sblimit[ch]++;
    }
  }
```

Used in 292.


ELSEVIER
MATT PHARR ○ GREG HUMPHREYS
PHYSICALLY BASED RENDERING
FROM THEORY TO IMPLEMENTATION
SERIES IN INTERACTIVE 3D TECHNOLOGY
PBR
PHYSICALLY BASED RENDERING
MK
MORGAN KAUFMANN

With thanks for your invention of literate programming and many happy hours spent reading your programs (not to mention many lessons learned from TAOCP!)

Physically Based Rendering

FROM THEORY TO IMPLEMENTATION

MATT PHARR
NVIDIA

GREG HUMPHREYS
Department of Computer Science
University of Virginia

Don,

This just would not have been possible without your vision of programs as literature! I don't think we'll offer rewards for bugs until the second edition, though :)

- Greg

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MORGAN KAUFMANN PUBLISHERS

2013 Sci-Tech Awards: Matt Pharr, Greg Humphreys and Pat Hanrahan

Oscars

Subscribe 486,909

8,203 views

⟨*Sampler Method Definitions*⟩ ≡

```
Sampler::Sampler(int xstart, int xend, int ystart, int yend, int spp) {
    xPixelStart = xstart;
    xPixelEnd = xend;
    yPixelStart = ystart;
    yPixelEnd = yend;
    samplesPerPixel = spp;
}
```

The `Sampler` implementation should generate samples for pixels with x coordinates ranging from `xPixelStart` to `xPixelEnd-1`, inclusive, and analogously for y coordinates.

⟨*Sampler Public Data*⟩ ≡ 296

```
int xPixelStart, xPixelEnd, yPixelStart, yPixelEnd;
int samplesPerPixel;
```

`Samplers` must implement the `Sampler::GetNextSample()` method, which is a pure virtual function. The `Scene::Render()` method calls this function until it returns `false`; each time it returns `true`, it should fill in the `sample` that is passed in with values that specify the next sample to be taken. All of the dimensions of the sample values it generates have values in the range [0, 1], except for `imageX` and `imageY`, which are specified with respect to the image size in raster coordinates.

⟨*Sampler Interface*⟩ ≡ 296

```
virtual bool GetNextSample(Sample *sample) = 0;
```

In order to make it easy for the main rendering loop to figure out what percentage of the scene has been rendered based on the number of samples processed, the `Sampler::TotalSamples()` method returns the total number of samples that the `Sampler` is expected to return.[4]

⟨*Sampler Interface*⟩ +≡ 296

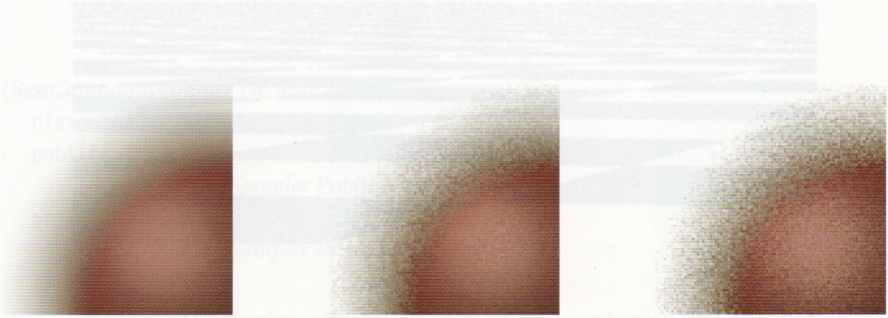
```
int TotalSamples() const {
    return samplesPerPixel *
           (xPixelEnd - xPixelStart) *
           (yPixelEnd - yPixelStart);
}
```

Sample 299
Sampler 296
Sampler::GetNextSample() 296
Sampler::samplesPerPixel 296
Sampler::TotalSamples() 296
Sampler::xPixelEnd 298
Sampler::xPixelStart 298
Sampler::yPixelEnd 298
Sampler::yPixelStart 298
Scene::Render() 24

7.2.1 SAMPLE REPRESENTATION AND ALLOCATION

The `Sample` structure is used by `Samplers` to store a single sample. A single `Sample` is allocated in the `Scene::Render()` method. For each camera ray to be generated, the

4 The low-discrepancy and best-candidate samplers, described later in the chapter, may actually return a few more or less samples than `TotalSamples()` reports. However, since computing the actual number can't be done quickly, and since an exact number is not really required, the expected number is returned here instead.



(a) (b) (c)

Figure 7.34: (a) Reference depth of field image, and images rendered with (b) the low-discrepancy and (c) best-candidate samplers. Here the low-discrepancy sampler is again the most effective.

shaped strata from an equal number of samples in each direction, so the `RoundSize()` method rounds up sample size requests so that they are an integer number squared.

⟨*BestCandidateSampler Public Methods*⟩ ≡ 344

```
int RoundSize(int size) const {
    int root = Ceil2Int(sqrtf((float)size - .5f));
    return root*root;
}
```

The `BestCandidateSampler::GetNextSample()` method has a similar basic approach to the other samplers in this chapter, except that the sample pattern sometimes extends beyond the image's boundaries due to the way it is tiled. These out-of-bounds samples must be ignored, which can lead to multiple tries in order to find an acceptable sample.

⟨*BestCandidateSampler Method Definitions*⟩ +≡

```
bool BestCandidateSampler::GetNextSample(Sample *sample) {
again:
    if (tableOffset == SAMPLE_TABLE_SIZE) {
        ⟨Advance to next best-candidate sample table position 347⟩
    }
    ⟨Compute raster sample from table 349⟩
    ⟨Check sample against crop window, goto again if outside 349⟩
    ⟨Compute integrator samples for best-candidate sample 350⟩
    ++tableOffset;
    return true;
}
```

If it has reached the end of the sample table, the sampler tries to move forward by `xTableCorner`. If this leaves the raster extent of the image, it moves ahead by `yTableCorner`, and if this takes y beyond the bottom of the image, it is finished.

BestCandidateSampler 344
BestCandidateSampler::GetNextSample() 345
BestCandidateSampler::tableOffset 344
Ceil2Int() 856
Sample 299
SAMPLE_TABLE_SIZE 335

⟨*BestCandidateSampler Private Data*⟩+≡ 344
```
float sampleOffsets[3];
```

Computing the raster space sample position from the positions in the table just requires some simple indexing and scaling. We don't use the Cranley-Patterson shifting technique on image samples because this would cause the sampling points at the borders between repeated instances of the table to have a poor distribution. Preserving good image distribution is more important than reducing correlation. The rest of the camera dimensions do use the shifting technique; the `WRAP` macro ensures that the result stays between zero and one.

⟨*Compute raster sample from table*⟩≡ 346
```
#define WRAP(x) ((x) > 1 ? ((x)-1) : (x))
sample->imageX = xTableCorner + tableWidth *
                 sampleTable[tableOffset][0];
sample->imageY = yTableCorner + tableWidth *
                 sampleTable[tableOffset][1];
sample->time  = WRAP(sampleOffsets[0] +
                     sampleTable[tableOffset][2]);
sample->lensU = WRAP(sampleOffsets[1] +
                     sampleTable[tableOffset][3]);
sample->lensV = WRAP(sampleOffsets[2] +
                     sampleTable[tableOffset][4]);
```

The sample table may spill off the edge of the image plane, so some of the generated samples may be outside the appropriate sample region. The sampler detects this case by checking the sample against the region of pixels to be sampled and generating a new sample if it's out of bounds.

⟨*Check sample against crop window, goto* `again` *if outside*⟩≡ 346
```
if (sample->imageX <  xPixelStart ||
    sample->imageX >= xPixelEnd   ||
    sample->imageY <  yPixelStart ||
    sample->imageY >= yPixelEnd) {
    ++tableOffset;
    goto again;
}
```

As explained previously, for integrator samples, the precomputed randomly scrambled low-discrepancy values are used if just one sample of this type is needed; otherwise a stratified pattern is used.

KS

1. Introduction. This is a "mock-up" of how the full power of literate programming might be useful to `R` programmers. It presents a simple (but interesting) function that computes the Kolmogorov–Smirnov statistic from empirical data. Then it shows some examples of that function in use.

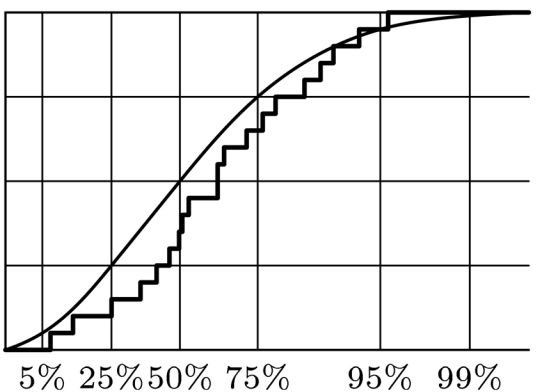
(The author apologizes for any awkward coding, due to the fact that this is actually his very first attempt to program in `R`.)

⟨ Define the function 5 ⟩
⟨ Test the function 9 ⟩

2. The Kolmogorov–Smirnov statistic. The *empirical distribution function* $F_n(x)$ of a set of samples $X_1, X_2, \ldots, X_n$ is defined to be

$$F_n(x) = \frac{\text{number of } X_1, X_2, \ldots, X_n \text{ that are} \le x}{n}.$$

For example, the jagged line shown here is an empirical distribution function taken from §3.3.1 of the book *Seminumerical Algorithms*:



3. The curved line in that illustration is another cumulative distribution function, $F(x)$. One way to measure goodness of fit, proposed by A. N. Kolmogorov in 1993 and modified by N. V. Smirnov in 1939, is to compute

$$K_n^+ = \sqrt{n} \sup_{-\infty<x<+\infty} \big(F_n(x) - F(x)\big);$$
$$K_n^- = \sqrt{n} \sup_{-\infty<x<+\infty} \big(F(x) - F_n(x)\big).$$

Here K_n^+ measures the greatest amount of deviation when F_n is greater than F, and K_n^- measures the maximum deviation when F_n is less than F. The normalizing factor $\sqrt{n}$ ensures that K_n^+ and K_n^- will converge to a limiting distribution as $n \to \infty$, if the X's are independent samples from F.

4. The given data comes from distribution F if and only if $F(X_j)$ has the uniform distribution between 0 and 1.

Hence the obvious way to compute K_n^+ and K_n^- is to start by sorting the X's so that $X_1 \le X_2 \le \cdots \le X_n$. Then we can compare $F(X_j)$ to the "ideal" value j/n.

However, the time needed for sorting is of order $n \log n$. We will use an improved method suggested by T. Gonzalez, S. Sahni, and W. R. Franta [*ACM Transactions on Mathematical Software* **3** (1977), 60–64], who noticed that *linear* time suffices if we place the unsorted samples into n bins.

Indeed, if we put Y_j into bin $\lfloor Y_j \rfloor$, the final statistics depend only on the smallest and largest elements in each of the n bins. Other elements in those bins don't contribute to the extremes that are measured by K_n^+ and K_n^-.

5. Coding details.

```
lit.ks.test=function(x,p) {
  ⟨ Return FALSE if y doesn't make sense 8 ⟩;
  ⟨ Insert y into bins, remembering extreme values 6 ⟩;
  ⟨ Find the overall extreme values lo, hi 7 ⟩;
  return (c(lo,hi)/sqrt(n))
}
```

This code is used in section 1.

6. The main idea is to keep track of $lb[j]$ and $ub[j]$, the smallest and largest element of bin $j-1$, as well as $count[j]$, the total number of elements in that bin, for $1 \le j \le n$.

(This is a situation where C programmers wish that `R` had 0-origin indexing.)

⟨ Insert y into bins, remembering extreme values 6 ⟩ ≡

```
lb=1:n
ub=count=array(0,dim=n)
for (j in 1:n) {
  yy=y[j]; k=yy%/%1+1
  if (yy<lb[k]) lb[k]=yy
  if (yy>ub[k]) ub[k]=yy
  count[k]=count[k]+1
}
```

This code is used in section 5.

7. Here's the key logic that makes it all work.

⟨ Find the overall extreme values *lo*, *hi* 7 ⟩ ≡

```
hi=lo=j=0
for (k in 1:n) {
  if (count[k]) {
    if (lb[k]-j>lo) lo=lb[k]-j
    j=j+count[k]
    if (j-ub[k]>hi) hi=j-ub[k]
  }
}
```

This code is used in section 5.

8. Of course a good subroutine intended for general use will check to see that bad parameters haven't been supplied. Otherwise the computer might hang up with subscripts out of range.

⟨ Return FALSE if y doesn't make sense 8 ⟩ ≡

```
OK=FALSE
if (sum(y<0)) message('That CDF has negative values!')
else if (sum(y>n)) message('That CDF has values > 1!')
else if (sum(y==n)) {
  for (j in 1:n)
    if (y[j]==n)
      message('x[',j,'] is too high: ',x[j])
} else OK=TRUE
if (!OK) return(FALSE)
```

This code is used in section 5.

9. A few unit tests. First let's make sure that those error messages are properly issued.

⟨ Test the function 9 ⟩ ≡

```
  lit.ks.test(-1, function(x) return(x))
That CDF has negative values!
[1] FALSE
  lit.ks.test(2, function(x) return(x))
That CDF has values > 1!
[1] FALSE
  lit.ks.test(c(.6,.23,pi,-.1,1), punif)
x[3] is too high: 3.14159265358979
x[5] is too high: 1
[1] FALSE
```

See also section 10.

This code is used in section 1.

10. And finally, we also want to obtain a valid result when we supply valid parameters.

(The built-in system function *ks.test* computes the slightly different statistics $D_n^- = K_n^-/\sqrt{n}$ and $D_n^+ = K_n^+/\sqrt{n}$.)

⟨ Test the function 9 ⟩ +≡

```
  x=c(.999,.21,.64,.87,.22)
  lit.ks.test(x,punif)/sqrt(5)
[1] 0.27 0.18
  ks.test(x,punif,alternative="less")
D^- = 0.27, p-value = 0.4134
alternative: the CDF lies below the null hypothesis
  ks.test(x,punif,alternative="greater")
D^+ = 0.18, p-value = 0.651
alternative: the CDF lies above the null hypothesis
```

11. Index.

⟨ Define the function 5 ⟩ Used in section 1.
⟨ Find the overall extreme values *lo*, *hi* 7 ⟩ Used in section 5.
⟨ Insert y into bins, remembering extreme values 6 ⟩ Used in section 5.
⟨ Return `FALSE` if y doesn't make sense 8 ⟩ Used in section 5.
⟨ Test the function 9, 10 ⟩ Used in section 1.