The New C Standard

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

information defined here

This is a sentence from the C99 standard, the number on the inside margin is the sentence number and this wording has been ~~deleted~~added by the response to a DR interpretation. 0

Commentary

This is some insightfull commentary on the above sentence. We might also say something relating to this issue in another sentence (see arrow pointing out in the margin). another sentence

The term *blah* is delimited with single-quotes to highlight the fact that it is defined as a term.

Rationale

This is a quote from the Rationale document produced by the C Committee to put a thoughtfull spin on the wording in the standard.

C90

This section deals with the C90 version of the standard. Specifically, how it differs from the C99 version of the above sentence. These sections only appear if there is a semantic difference (in some cases the words may have changed slightly, leaving the meaning unchanged).

DR #987

This is the text of a DR (defect report) submitted to the ISO C Standard committee.

Response

The committees response to this DR is that this question is worth repeating at this point in the book.

This is where we point out what the difference is, and what the developer might do, if anything, abou it.

C++

1.1p1

This is a sentence from the C++ standard specifying behavior that is different from the above C99 sentence. The 1.1p1 in the outside margin is the clause and paragraph number of this quote in the C++ Standard.

This is where we point out what the difference is, and what the developer might do, if anything, abou it. You believed the hype that the two language are compatible? Get real!

Other Languages

Developers are unlikely to spend their entire professional life using a single language. This section sometimes gives a brief comparison between the C way of doing things and other languages.

Comment received during balloting

We vote against the adoption of the proposed new COBOL standard because we have lost some of our source code and don't know whether the requirements in the proposed new standard would invalidate this source.

Common Implementations

processors 17 bit

Discussion of how implementations handle the above sentence. For instance, only processors with 17 bit integers can implement this requirement fully. GCC has extensions to support 16 bit processors in this area (the text in the outside margin is pushed towards the outside of the page, indicating that this is where a particular issue is discussed; the text appearing in a smaller point size is a reference to material appearing elsewhere {the number is the C sentence number}). translated invalid program

The New C Standard

This is a quote from the document referenced in the outside sidebar.

Coding Guidelines

General musings on how developers use constructs associated with the above sentence. Some of these sections recommend that a particular form of the construct described in the above sentence not be used.

1 Do it this way and save money.

DEV A possible deviation from the guideline, for a described special case.

REV Something to look out for during a code review. Perhaps a issue that requires a trade off among different issues, or that cannot be automated.

Example

An example, in source code of the above sentence.

The examples in this book are generally intended to illustrate some corner of the language. As a general rule it is considered good practice for authors to give examples that readers should follow. Unless stated otherwise, the examples in this book always break this rule.

```
struct {float mem;} main(void)
{
int blah;  /* The /* form of commenting describes the C behavior   */
           // The // form of commenting describes the C++ behavior
}
```

Usage

A graph or table showing the number of occurrences the constructs specified by the above C sentence occur in source code.

token syntax
preprocessing token syntax
lexical element

763

token:
keyword
identifier
constant
string-literal
punctuator
preprocessing-token:
header-name
identifier
pp-number
character-constant
string-literal
punctuator
each non-white-space character that cannot be one of the above

Commentary

Tokens (preprocessor or otherwise) are the atoms from which the substance of programs are built. Preprocessor tokens are created in translation phase 3 and are turned into tokens in translation phase 7. [] translation phase 3 [] translation phase 7

All characters in the basic source character set can be used to form some kind of preprocessor token that is defined by the standard. When creating preprocessing tokens the first non-white-space character is sufficient, in all but one case, to determine the kind of preprocessing token being created. [] punctuator syntax [] header name syntax

The two characters, double-quote and single-quote, must occur with their respective matching character if they are to form a defined preprocessor-token. The special case of them occurring singly, which matches against "*non-white-space character that cannot be one of the above*," is dealt with below. The other cases that match against *non-white-space character that cannot be one of the above* involve characters that are outside of the basic source character set. A program containing such extended characters need not result in a constraint violation, provided the implementation supports such characters. For instance, they could be stringized prior to translation phase 7, or they could be part of a preprocessor-token sequence being skipped as part of a conditional compilation directive. [] character ' or " matches [] basic source character set [] preprocessing token shall have lexical form [] # operator

The header-name preprocessing token is context-dependent and only occurs in the **`#include`** preprocessing directive. It never occurs after translation phase 4. [] #include h-char-sequence

C90

The *non-terminal* operator was included as both a *token* and *preprocessing-token* in the C90 Standard. Tokens that were operators in C90 have been added to the list of punctuators in C99.

C++

C++ maintains the C90 distinction between operators and punctuators. C++ also classifies what C calls a constant as a literal, a string-literal as a literal and a C character-constant is known as a character-literal.

Other Languages

Few other language definitions include a preprocessor. The PL/1 preprocessor includes syntax that supports some statements having the same syntax as the language to be executed during translation.

Some languages (e.g., PL/1) do not distinguish between keywords and identifiers. The context in which a name occurs is used to select the usage to which it is put. Other languages (e.g., Algol 60) use, conceptually, one character set for keywords and another for other tokens. In practice only one character set is available. In books and papers the convention of printing keywords in bold was adopted. A variety of conventions were used for writing Algol keywords in source, including using an underline facility in the character encodings, using matching single-quote characters, or simply all uppercase letters.

Common Implementations

The handling of "*each non-white-space character that cannot be one of the above*" varies between implementations. In most cases an occurrence of such a preprocessing token leads to a syntax or constraint violation.

Coding Guidelines

Most developers are not aware of that preprocessing-tokens exist. They think in terms of a single classification of tokens—the token. The distinction only really becomes noticeable when preprocessing-tokens that are not also tokens occur in the source. This can occur for pp-number and the "*each non-white-space character that cannot be one of the above*" and is discussed elsewhere. There does not appear to be a worthwhile benefit in educating developers about preprocessing-tokens. [] pp-number syntax [763] preprocessing token syntax

Summary

The following two sections provide background on those low-level visual processing operations that might be applicable to source code. The first section covers what is sometimes called *early vision*. This phase of vision is carried out without apparent effort. The possibilities for organizing the visual appearance of source code to enable it to be visually processed with no apparent effort are discussed. At this level the impact of individual characters is not considered, only the outline image formed by sequences (either vertically or horizontally) of characters. The second section covers eye movement in reading. This deals with the processing of individual, horizontal sequences of characters. To some extent the form of these sequences is under the control of the developer. Identifiers (whose spelling is under developer-control) and space characters make up a large percentage of the characters on a line.

reading practice letter detection 763

The early vision factors that appear applicable to C source code are proximity, edge detection, and distinguishing features. The factors affecting eye movement in reading are practice related. More frequently encountered words are processed more quickly and knowledge of the frequency of letter sequences is used to decide where to move the eyes next.

The discussion assumes a 2-D visual representation; although 3-D visualization systems have been developed[35] they are still in their infancy.

vision early

coding guidelines introduction

1 Early vision

One of the methods used by these coding guidelines to achieve their stated purpose is to make recommendations that reduce the cognitive load needed to read source code. This section provides an overview of some of the operations the human visual system can perform without requiring any apparent effort. The experimental evidence[29] suggests that the reason these operations do not create a cognitive load is that they occur before the visual stimulus is processed by the human cognitive system. The operations occur in what is known as *early vision*. Knowledge of these operations may be of use in deciding how to organize the visible appearance of token sequences (source code layout).

source character set

The display source code uses a subset of the possible visual operations that occur in nature. It is nonmoving, two-dimensional, generally uses only two colors, items are fully visible (they are not overlayed), and edges are rarely curved. The texture of the display is provided by the 96 characters of the source character set (in many cases a limited number of additional characters are also used). Given that human vision is tuned to extract important information from natural scenes, it is to be expected that many optimized visual processes will not be applicable to viewing source code.

vision preattentive

1.1 Preattentive processing

Some inputs to the human visual system appear to *pop-out* from their surroundings. Preattentive processing, so called because it occurs before conscious attention, is automatic and apparently effortless. The examples in Figure 1 show some examples of features that *pop-out* at the reader.

Preattentive processing is independent of the number of distractors; a search for the feature takes the same amount of time whether it occurs with one, five, ten, or more other distractors. However, the disadvantage is that it is only effective when the features being searched for are relatively rare. When a display contains many different, distinct features (the mixed category in Figure 1), the *pop-out* effect does not occur. The processing abilities demonstrated in Figure 1 are not generally applicable to C source code for a number of reasons.

basic source character set

- C source code is represented using a fixed set of characters. Opportunities for introducing *graphical effects* into source code are limited. The only, universally available technique for controlling the visual appearance of source code is white space.
- While there are circumstances when a developer might want to attend to a particular identifier, or declaration, in general there are no constructs that need to *pop-out* to all readers of the source. Program development environments may highlight (using different colors) searched for constructs, dependencies between constructs, or alternative representations (for instance, call graphs), but these are temporary requirements that change over short periods of time, as the developer attempts to comprehend source code.

gestalt principles

1.2 Gestalt principles

Founded in 1912 the Gestalt school of psychology proposed what has become known as the *Gestalt laws of perception* (*gestalt* means *pattern* in German); they are also known as the *laws of perceptual organization*.

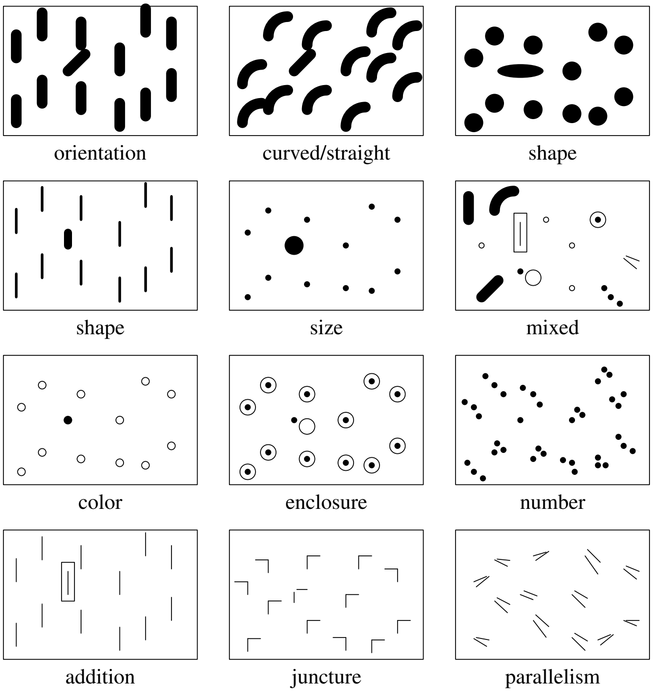

Figure 1: Examples of features that may be preattentively processed (parallel lines and the junction of two lines are the odd ones out). Based on Ware.[40]

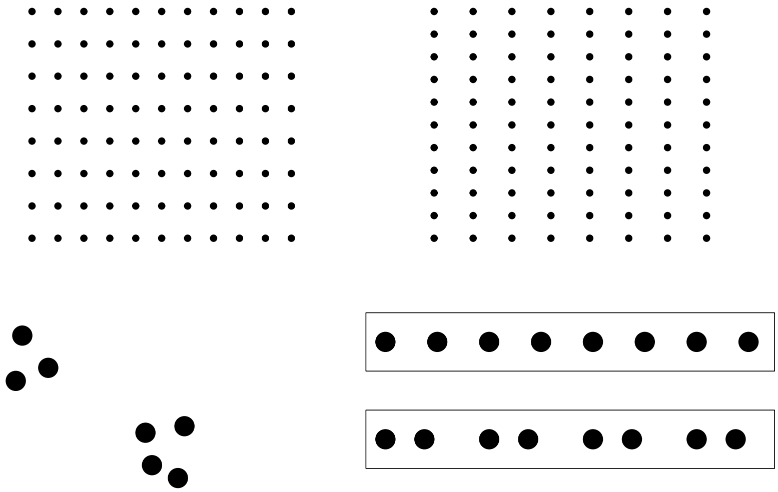

Figure 2: Proximity—the horizontal distance between the dots in the upper left image is less than the vertical distance, causing them to be perceptually grouped into lines (the relative distances are reversed in the upper right image).

The underlying idea is that the whole is different than the sum of its parts. These so-called *laws* do not have the rigour expected of a scientific law, and really ought to be called by some other term (e.g., principle). The following are some of the more commonly occurring principles

- **Proximity**: Elements that are close together are perceptually grouped together (Figure 2).
- **Similarity**: Elements that share a common attribute can be perceptually grouped together (Figure 3).

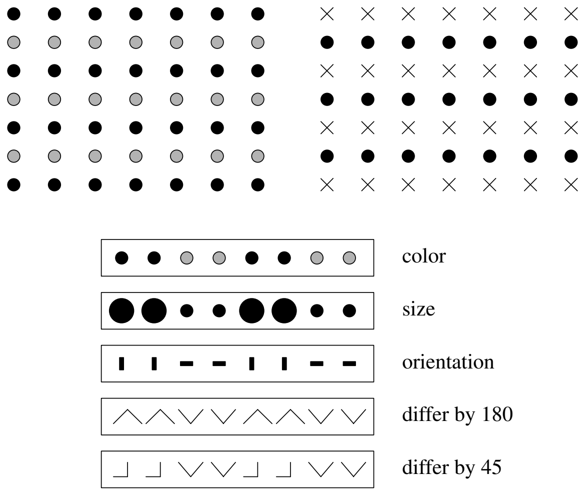

Figure 3: Similarity—a variety of dimensions along which visual items can differ sufficiently to cause them to be perceived as being distinct; rotating two line segments by 180° does not create as big a perceived difference as rotating them by 45°.

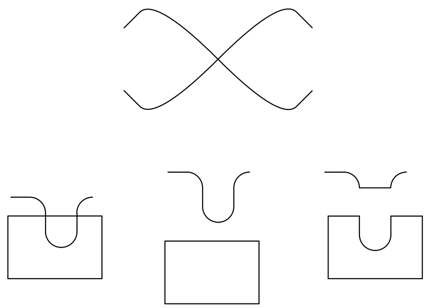

Figure 4: Continuity—upper image is perceived as two curved lines; the lower left image is perceived as a curved line overlapping a rectangle rather than an angular line overlapping a rectangle having a piece missing (lower right image).

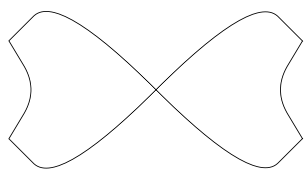

Figure 5: Closure—when the two perceived lines in the upper image of Figure 4 are joined at their end, the perception changes to one of two cone-shaped objects.

continuation gestalt principle of

- **Continuity**, also known as **Good continuation**: Lines and edges that can be seen as smooth and continuous are perceptually grouped together (Figure 4).
- **Closure**: Elements that form a closed figure are perceptually grouped together (Figure 5).
- **Symmetry**: Treating two, mirror image lines as though they form the outline of an object (see Figure 6). This effect can also occur for parallel lines.
- **Other** principles include grouping by connectedness, grouping by common region, and synchrony.[24]

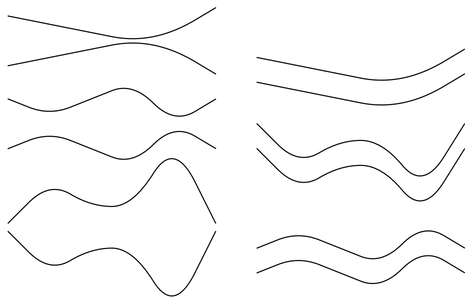

Figure 6: Symmetry and parallelism—where the direction taken by one line follows the same pattern of behavior as another line.

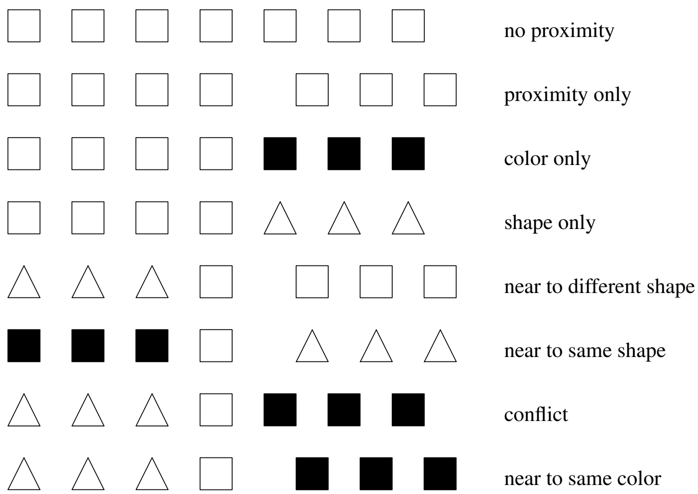

Figure 7: Conflict between proximity, color, and shape. Based on Quinlan.[30]

The organization of visual grouping of elements in a display, using these principles, is a common human trait. However, when the elements in a display contain instances of more than one of these perceptual organization principles, people differ in their implicit selection of principle used. A study by Quinlan and Wilton[30] found that 50% of subjects grouped the elements in Figure 7 by proximity and 50% by similarity. They proposed the following, rather incomplete, algorithm for deciding how to group elements:

1. Proximity is used to initially group elements.
2. If there is a within-group attribute mismatch, but a match between groups, people select between either a proximity or a similarity grouping (near to different shape in Figure 7).
3. If there is a within-group and between-group attribute mismatch, then proximity is ignored. Grouping is then often based on color rather than shape (near to same color and near to same shape in Figure 7).

Recent work by Kubovy and Gepshtein[17] has tried to formulate an equation for predicting the grouping of rows of dots. Will the grouping be based on proximity or similarity? They found a logarithmic relationship between dot distance and brightness that is a good predictor of which grouping will be used.

The symbols available to developers writing C source provide some degree of flexibility in the control of its visual appearance. The appearance is also affected by parameters outside of the developers' control—for

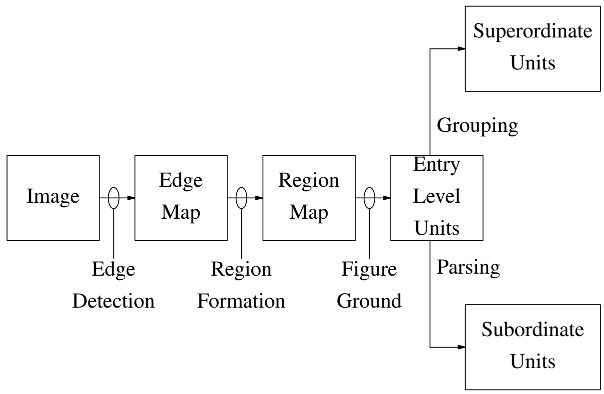

Figure 8: A flowchart of Palmer and Rock's[24] theory of perceptual organization.

instance, line and intercharacter spacing. While developers may attempt to delineate sections of source using white space and comments, the visual impact of the results do not usually match what is immediately apparent in the examples of the Gestalt principles given above. While instances of these principles may be used in laying out particular sequences of code, there is no obvious way of using them to create generalized layout rules. The alleged benefits of particular source layout schemes invariably depend on practice (a cost). The Gestalt principles are preprogrammed (no cost). These coding guidelines cannot perform a cost/benefit analysis of the various code layout rules because your author knows of no studies, using experienced developers, investigating this topic.

reading practice

Edge detection

1.3 Edge detection

The striate cortex is the visual receiving area of the brain. Neurons within this area respond selectively to the orientation of edges, the direction of stimulus movement, color along several directions, and other visual stimuli. In Palmer and Rock's[24] theory of perceptual organization, edge detection is the first operation performed on the signal that appears as input to the human visual system. After edges are detected, regions are formed, and then figureprinciples operate to form entry-level units (Figure 8).

C source is read from left to right, top to bottom. It is common practice to precede the first non-white-space character on a sequence of lines to start at the same horizontal position. This usage has been found to reduce the effort needed to visually process lines of code that share something in common; for instance, statement indentation is usually used to indicate block nesting.

Edge detection would appear to be an operation that people can perform with no apparent effort. An edge can also be used to speed up the search for an item if it occurs along an edge. In the following sequences of declarations, less effort is required to find a particular identifier in the second two blocks of declarations. In the first block the reader first has to scan a sequence of tokens to locate the identifier being declared. In the other two blocks the locations of the identifiers are readily apparent. Use of edges is only part of the analysis that needs to be carried out when deciding what layout is likely to minimize cognitive effort. These analyses are given for various constructs elsewhere.

statement visual layout
declaration visual layout

```
/* first block */
int glob;
unsigned long a_var;
const signed char ch;
volatile int clock_val;
void *free_mem;
void *mem_free;

/* second block */
int                glob;
unsigned long      a_var;
const signed char  ch;
volatile int       clock_val;
void *             free_mem;
void               *mem_free;

/* third block */
              int  glob;
    unsigned long  a_var;
const signed char  ch;
     volatile int  clock_val;
           void *  free_mem;
             void  *mem_free;
```

Searching is only one of the visual operations performed on source. Systematic line-by-line, token-by-token reading is another. [] reading kinds of The extent to which the potentially large quantities of white space introduced to create edges increases the effort required for systematic reading is unknown. For instance, the second block (previous code example) maintains the edge at the start of the lines at which systematic reading would start, but at the cost of requiring a large saccade to the identifier. [763] Saccade The third block only requires a small saccade to the identifier, but there is no edge to aid in the location of the start of a line.

1.4 Reading practice

reading practice

A study by Kolers and Perkins[16] offers some insight into the power of extended practice. [] expertise In this study subjects were asked to read pages of text written in various ways; pages contained, normal, reversed, inverted, or mirrored text.

```
Expectations can also mislead us; the unexpected is always hard to
perceive clearly.  Sometimes we fail to recognize an object because we
saw tI  .eb ot serad eh sa yzal sa si nam yreve taht dias ecno nosremE
si tI  .ekam ot detcepxe eb thgim natiruP dnalgnE weN a ekatsim fo dnik eht
These are but a few of the reasons for believing that a
person cannot be conscious of all his mental processes.
Many other reasons can be
                         Several years ago a professor who teaches psychology at a
                  large university had to ask his assistant, a young man of great
                                                                      intelligence
```

The time taken for subjects to read a page of text in a particular orientation was measured. The more pages subjects read, the faster they became. This is an example of the power law of learning. [] power law of learning A year later Kolers[15] measured the performance of the same subjects, as they read more pages. Performance improved with practice, but this time the subjects had past experience and their performance started out better and improved more quickly (Figure 9). These results are similar to those obtained in the letter-detection task. [763] letter detection

Just as people can learn to read text written in various ways, developers can learn to read source code laid out in various ways. The important issue is not developers' performance with a source code layout they have extensive experience reading, but their performance on a layout they have little experience reading. [] reading practice For instance, how quickly can they achieve a reading performance comparable to that achieved with a familiar

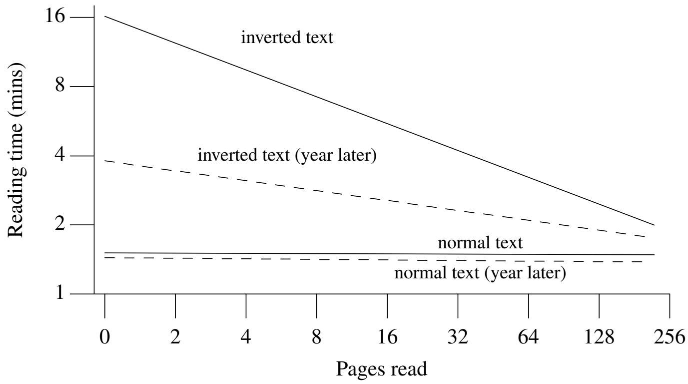

Figure 9: Results for the same six subjects in two tests more than a year apart. Based on Kolers.[15]

layout (based on reading and error rate). The ideal source code layout is one that can be quickly learned and has a low error rate (compared with other layouts).

Unfortunately there are no studies, using experienced developers, that compare the effects of different source code layout on reading performance. Becoming an experienced developer can be said to involve learning to read source that has been laid out in a number of different ways. The visually based guidelines in this book do not attempt to achieve an optimum layout, rather they attempt to steer developers away from layouts that are likely to be have high error rates.

Many developers believe that the layout used for their own source code is optimal for reading by themselves, and others. It may be true that the layout used is optimal for the developer who uses it, but the reason for this is likely to be practice-based rather than any intrinsic visual properties of the source layout. Other issues associated with visual code layout are discussed in more detail elsewhere.

declaration visual layout
statement visual layout

distinguishing features

1.5 Distinguishing features

A number of studies have found that people are more likely to notice the presence of a distinguishing feature than the absence of a distinguishing feature. This characteristic affects performance when searching for an item when it occurs among visually similar items. It can also affect reading performance—for instance, substituting an *e* for a *c* is more likely to be noticed than substituting a *c* for an *e*.

A study by Treisman and Souther[37] found that visual searches were performed in parallel when the target included a unique feature (search time was not affected by the number of background items), and searches were serial when the target had a unique feature missing (search time was proportional to the number of background items). These results were consistent with Treisman and Gelade's[38] feature-integration theory.

What is a unique feature? Treisman and Souther investigated this issue by having subjects search for circles that differed in the presence or absence of a gap (Figure 10). The results showed that subjects were able to locate a circle containing a gap, in the presence of complete circles, in parallel. However, searching for a complete circle, in the presence of circles with gaps, was carried out serially. In this case the gap was the unique feature. Performance also depended on the proportion of the circle taken up by the gap.

As discussed in previous subsections, C source code is made up of a fixed number of different characters. This restricts the opportunities for organizing source to take advantage of the search asymmetry of preattentive processing. It is important to remember the preattentive nature of parallel searching; for instance, comments are sometimes used to signal the presence of some construct. Reading the contents of these comments would require attention. It is only their visual presence that can be a distinguishing feature from the point of view of preattentive processing. The same consideration applies to any organizational layout using space characters. It is the visual appearance, not the semantic content that is important.

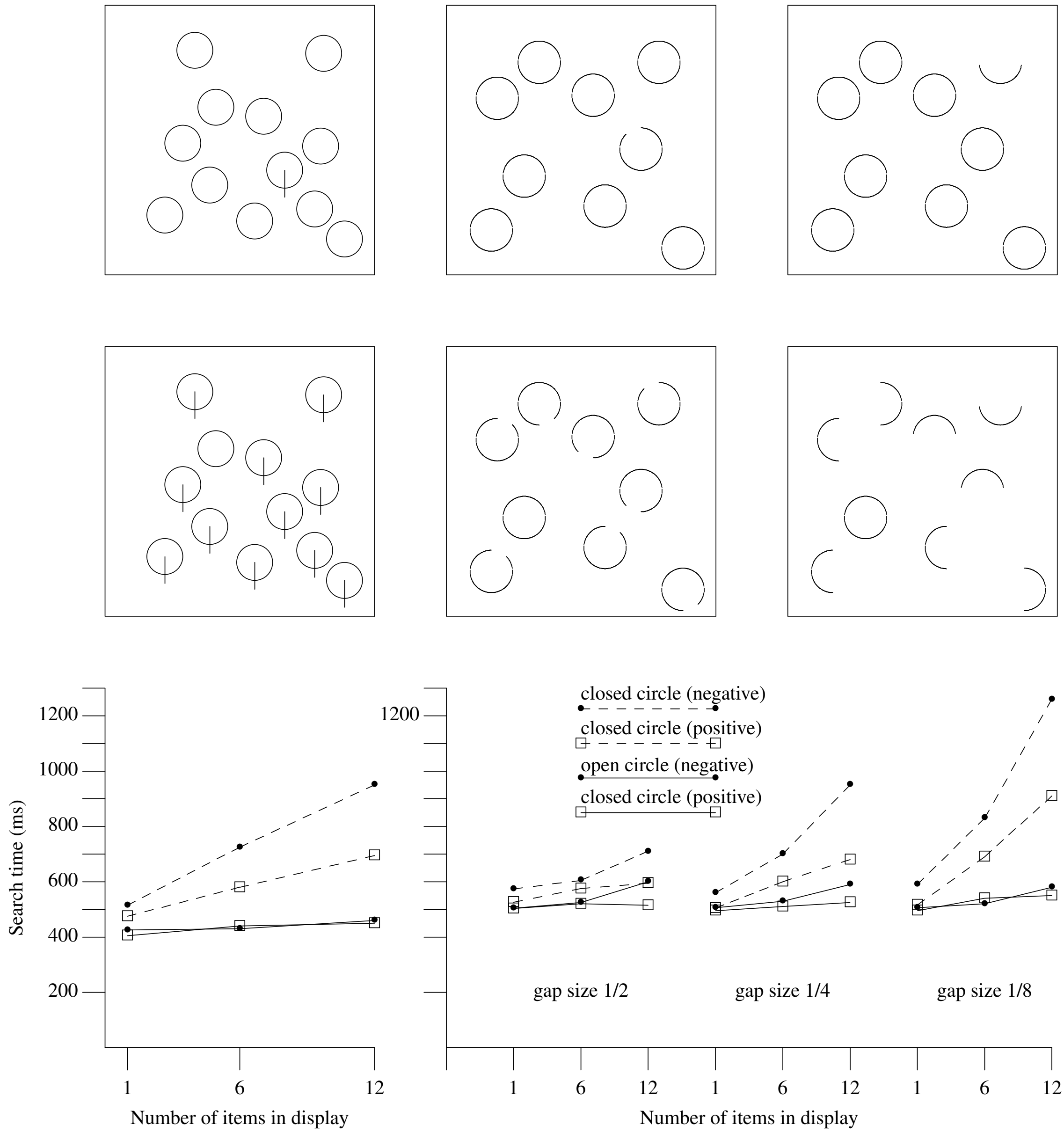

Figure 10: Examples of unique items among visually similar items. Those at the top include an item that has a distinguishing feature (a vertical line or a gap); those underneath them include an item that is missing this distinguishing feature. Graphs represent time taken to locate unique items (positive if it is present, negative when it is not present) when placed among different numbers of visibly similar distractors. Based on displays used in the study by Treisman and Sother.[37]

1.6 Visual capacity limits

A number of studies have looked at the capacity limits of visual processing.[12,39] Source code is visually static, that is it does not move under the influence of external factors (such as the output of a dynamic trace of an executing program might). These coding guidelines make the assumption that the developer-capacity bottleneck occurs at the semantic level, not the visual processing stage.

Reading eye movement

2 Reading (eye movement)

While C source code is defined in terms of a sequence of ordered lines containing an ordered sequence of characters, it is rarely read that way by developers. There is no generally accepted theory for how developers read source code, at the token level, so the following discussion is necessarily broad and lacking in detail. Are there any organizational principles of developers' visual input that can be also be used as visual organizational principles for C source code?

Developers talk of reading source code; however, reading C source code differs from reading human language prose in many significant ways, including:

- It is possible, even necessary, to create new words (identifiers). The properties associated with these words are decided on by the author of the code. These words might only be used within small regions of text (their scope); their meaning (type) and spelling are also under the control of the original developer.
- Although C syntax specifies a left-to-right reading order (which is invariably presented in lines that read from the top, down), developers sometimes find it easier to comprehend statements using either a right to left reading, or even by starting at some subcomponent and working out (to the left and right) or lines reading from the bottom, up.
- Source code is not designed to be a spoken language; it is rare for more than short snippets to be verbalized. Without listeners, developers have not needed to learn to live (write code) within the constraints imposed by realtime communication between capacity-limited parties.
- The C syntax is not locally ambiguous. It is always possible to deduce the syntactic context, in C, using a lookahead of a single word[1] (the visible source may be ambiguous through the use of the preprocessor, but such usage is rare and strongly recommended against). This statement is not true in C++ where it is possible to write source that requires looking ahead an indefinite number of words to disambiguate a localized context.
- In any context a word has a single meaning. For instance, it is not necessary to know the meaning (after preprocessing) of `a`, `b` and `c`, to comprehend `a=b+c`. This statement is not true in computer languages that support overloading, for instance C++ and Java.
- Source code denotes operations on an abstract machine. Individually the operations have no external meaning, but sequences of these operations can be interpreted as having an equivalence to a model of some external *real-world* construct. For instance, the expression `a=b+c` specifies the abstract machine operations of adding `b` to `c` and storing the resulting value in `a`; its interpretation (as part of a larger sequence of operations) might be *move on to the next line of output*. It is this semantic mapping that creates cognitive load while reading source code. When reading prose the cognitive load is created by the need to disambiguate word meaning and deduce a parse using known (English or otherwise) syntax.

Reading and writing is a human invention, which until recently few people could perform. Consequently, human visual processing has not faced evolutionary pressure to be good at reading.

[1] There is one exception—for the token sequence `void func (a, b, c, d, e, f, g)`. It is not known whether `func` is a declaration of a prototype or a function definition until the token after the closing parenthesis is seen.

Roadside joggers endure sweat, pain and angry drivers in the name of
1 2 3 4 5 6 7 8
286 221 246 277 256 233 216 188

fitness. A healthy body may seem reward enough for most people. However,
9 10 12 13 11 14 15 16 17 18 19
301 177196 175 244 302 112 177 266 188 199

for all those who question the payoff, some recent research on physical
21 20 22 23 24 25 26 27

activity and creativity has provided some surprisingly good news. Regular
29 28 30 31 32 33 34 35 36 37
20166 201 188 203 220 217 288 21275

Figure 11: A passage of text with fixation position (dot under word), fixation sequence number, and fixation duration (in milliseconds) included. From Reichle, Pollatsek, Fisher, and Rayner[32] (timings on the third line are missing in the original).

While there are many differences between reading code and prose, almost no research has been done on reading code and a great deal of research has been done on reading prose. The models of reading that have been built, based on the results of prose-related research, provide a starting point for creating a list of issues that need to be considered in building an accurate model of code reading. The following discussion is based on papers by Rayner[31] and Reichle, Rayner, and Pollatsek.[33]

During reading, a person's eyes make short rapid movements. These movements are called *saccades* and take 20ms to 50 ms to complete. No visual information is extracted during these saccades and readers are not consciously aware that they occur. A saccade typically moves the eyes forward 6 to 9 characters. Between saccades the eyes are stationary, typically for 200ms to 250ms (a study of consumer eye movements[26] while comparing multiple brands found a fixation duration of 354ms when subjects were under high time pressure and 431ms when under low time pressure). These stationary time periods are called *fixations*. Reading can be compared to watching a film. In both cases a stationary image is available for information extraction before it is replaced by another (4–5 times a second in one case, 50–60 times a second in the other). However, in the case of a film the updating of individual images is handled by the projector while the film's director decides what to look at next; but during reading a person needs to decide what to look at next and move the eyes to that location. Saccade

The same reader can show considerable variation in performing these actions. Saccades might move the eyes by one character, or 15 to 20 characters (the duration of a saccade is influenced by the distance covered, measured in degrees). Fixations can be shorter than 100ms or longer than 400ms (they can also vary between languages[22]). The content of the fixated text has a strong effect on reader performance.

The eyes do not always move forward during reading—10% to 15% of saccades move the eyes back to previous parts of the text. These backward movements, called *regressions*, are caused by problems with linguistic processing (e.g., incorrect syntactic analysis of a sentence) and oculomotor error (for instance, the eyes overshot their intended target). [763] garden path sentence

Saccades are necessary because the eyes' field of view is limited. Light entering the eyes falls on the retina, where it hits light-sensitive cells. These cells are not uniformly distributed, but are more densely packed in the center of the retina. This distribution of light sensitive cells divides the visual field (on the retina) into three regions: foveal (the central 2°s, measured from the front of the eye looking toward the retina), parafoveal (extending out to 5°s), and peripheral (everything else). Letters become increasingly difficult to identify as their angular distance from the center of the fovea increases. visual field

A reader has to perform two processes during the fixation period: (1) identify the word (or sequence of letters forming a partial word) whose image falls within the foveal and (2) plan the next saccade (when to make it and where to move the eyes). Reading performance is speed limited by the need to plan and perform saccades. If the need to saccade is removed by presenting words at the same place on a display, there is a threefold speed increase in reading aloud and a twofold speed increase in silent reading. The time needed to plan and perform a saccade is approximately 180ms to 200ms (known as the *saccade latency*), which means that the decision to make a saccade occurs within the first 100ms of a fixation. How does a reader make a good saccade decision in such a short period of time?

The contents of the parafoveal region are partially processed during reading. The parafoveal region increases a reader's perceptual span. When reading words written using alphabetic characters (e.g., English or German), the perceptual span extends from 3 to 4 characters on the left of fixation to 14 to 15 letters to the right of fixation. This asymmetry in the perceptual span is a result of the direction of reading, attending to letters likely to occur next being of greater value. Readers of Hebrew (which is read right to left) have a perceptual span that has opposite asymmetry (in bilingual Hebrew/English readers the direction of the asymmetry depends on the language being read, showing the importance of attention during reading[28]).

The process of reading has attracted a large number of studies. The following general points have been found to hold:

- The perceptual span does not extend below the line being read. Readers' attention is focused on the line currently being read.
- The size of the perceptual span is fairly constant for similar alphabetic orthographies (graphical representation of letters).
- The characteristics of the writing system affect the asymmetry of the perceptual span and its width. For instance, the span can be smaller for Hebrew than English (Hebrew words can be written without the vowels, requiring greater effort to decode and plan the next saccade). It is also much smaller for writing systems that use ideographs, such as Japanese (approximately 6 characters to the right) and Chinese.
- The perceptual span is not hardwired, but is attention-based. The span can become smaller when the fixated words are difficult to process. Also readers obtain more information in the direction of reading when the upcoming word is highly predictable (based on the preceding text).
- Orthographic and phonological processing of a word can begin prior to the word being fixated.
- Words that can be identified in the parafovea do not have to be fixated and can be skipped. Predictable words are skipped more than unpredictable words, and short function words (like *the*) are skipped more than content words.

The processes that control eye movement have to decide where (to fixate next) and when (to move the eyes). These processes sometimes overlap and are made somewhat independently (see Figure 11).

Where to fixate next. Decisions about where to fixate next seem to be determined largely by low-level visual cues in the text, as follows.

- Saccade length is influenced by the length of both the fixated word and the word to the right of fixation.
- When readers do not have information about where the spaces are between upcoming words, saccade length decreases and reading rate slows considerably.
- Although there is some variability in where the eyes land on a word, readers tend to make their first fixation about halfway between the beginning and the middle of a word.
- While contextual constraints influence skipping (highly predictable words are skipped more than unpredictable words), contextual constraints have little influence on where the eyes land in a word (however, recent research[18] has found some semantic-context effects influence eye landing sites).

- The landing position on a word is strongly affected by the launch site (the previous landing position). As the launch site moves further from the target word, the distribution of landing positions shifts to the left and becomes more variable.

When to move the eyes. The ease or difficulty associated with processing a word influences when the eyes move, as follows.

- There is a spillover effect associated with fixating a low-frequency word; fixation time on the next word increases.
- Although the duration of the first fixation on a word is influenced by the frequency of that word, the duration of the previous fixation (which was not on that word) is not.
- High-frequency words are skipped more than low-frequency words, particularly when they are short and the reader has fixated close to the beginning of the word.
- Highly predictable (based on the preceding context) words are fixated for less time than words that are not so predictable. The strongest effects of predictability on fixation time are not usually as large as the strongest frequency effects. Word predictability also has a strong effect on word skipping.

2.1 Models of reading

It is generally believed that eye movements follow visual attention. This section discusses some of the models of eye movements that have been proposed and provides some of the background theory needed to answer questions concerning optimal layout of source code. An accurate, general-purpose model of eye movement would enable the performance of various code layout strategies to be tested. Unfortunately, no such model is available. This book uses features from three models, which look as if they may have some applicability to how developers read source. For a comparison of the different models, see Reichle, Rayner and Pollatsek.[33]

2.1.1 Mr. Chips

Mr. Chips

Mr. Chips[20] is an ideal-observer model of reading (it is also the name of a computer program implemented in C) which attempts to calculate the distance, measured in characters, of the next saccade. (It does not attempt to answer the question of when the saccade will occur.) It is an idealized model of reading in that it optimally combines three sources of information (it also includes a noise component, representing imperfections in oculomotor control):

1. Visual data obtained by sampling the text through a *retina*, which has three regions mimicing the behavior of those in the human eye.
2. Lexical knowledge, consisting of a list of words and their relative frequencies (English is used in the published study).
3. Motor knowledge, consisting of statistical information on the accuracy of the saccades made.

Mr. Chips uses a single optimization principle—entropy minimization. All available information is used to select a saccade distance that minimizes the uncertainty about the current word in the visual field (ties are broken by picking the largest distance). Executing the Mr. Chips program shows it performing regressive saccades, word skips, and selecting viewing positions in words, similar to human performance.

Mr. Chips is not intended to be a model of how humans read, but to establish the pattern of performance when available information is used optimally. It is not proposed that readers perform entropy calculations when planning saccades. There are simpler algorithms using a small set of heuristics that perform close to the entropy minimization ideal (see Figure 12).

The eyes' handling of visual data and the accuracy of their movement control are physical characteristics. The lexical knowledge is a characteristic of the environment in which the reader grew up. A person

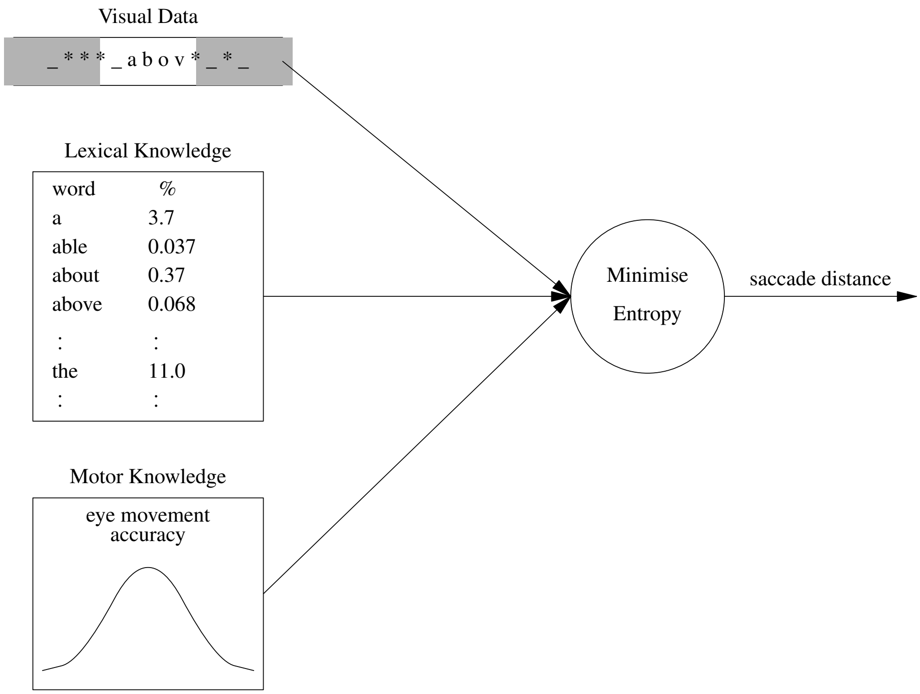

Figure 12: Mr. Chips schematic based on.[20] The shaded region in the visual data is the parafoveal; in this region individual letters (indicated by stars) can only be distinguished from spaces (indicated by underscores).

has little control over the natural language words they hear and see, and how often they occur. Source code declarations create new words that can then occur in subsequent parts of the source being worked on by an individual. The characteristics of these words will be added to developers' existing word knowledge. Whether particular, code-specific letter sequences will be encountered sufficiently often to have any measurable impact on the lexicon a person has built up over many years is not known (see Figure).

E-Z Reader model

2.1.2 The E-Z Reader model

The E-Z Reader model of eye movement control in reading is described by Reichle, Pollatsek, Fisher, and Rayner.[32] It aims to give an account of how cognitive and lexical processes influence the eye movements of skilled readers. Within this framework it is the most comprehensive model of reading available. An important issue ignored by this model is higher order processing. (The following section describes a model that attempts to address cognitive issues.) For instance, in the sentence "*Since Jay always jogs a mile seems like a short distance.*" readers experience a disruption that is unrelated to the form or meaning of the individual words. The reader has been led down a syntactic *garden path*; initially parsing the sentence so that "*a mile*" is the object of "*jogs*" before realizing that "*a mile*" is the subject of "*seems*". Also it does not attempt to model the precise location of fixations.

garden path sentence

The aspect of this model that is applicable to reading source code is the performance dependency, of various components to the frequency of the word being processed (refer to Figure 11). The *familiarity check* is a quick assessment or whether word identification is imminent, while *completion of lexical access* corresponds to a later stage when a word's identity has been determined.

EMMA

2.1.3 EMMA

EMMA[34] is a domain-independent model that relates higher-level cognitive processes and attention shifts with lower-level eye movement behavior. EMMA is based on many of the ideas in the E-Z model and uses ACT-R[1] to model cognitive processes. EMMA is not specific to reading and has been applied to equation-

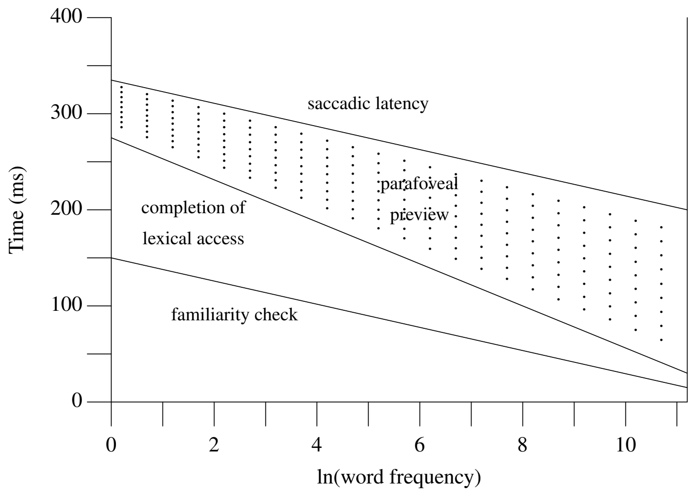

Figure 13: How preview benefit is affected by word frequency. The bottom line denotes the time needed to complete the familiarity check, the middle line the completion of lexical access, and the top line when the execution of the eye movement triggered by the familiarity check occurs. Based on Reichle, Pollatsek, Fisher, and Rayner.[32]

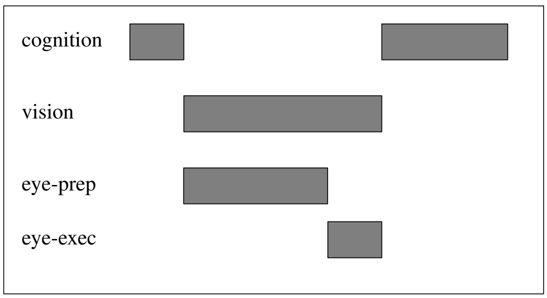

Figure 14: Example case of EMMA's control flow. From Salvucci.[34]

solving and visual search.

The *spotlight* metaphor of visual attention, used by EMMA, selects a single region of the visual field for processing. Shifting attention to a new visual object requires that it be encoded into an internal representation. The time, T_{enc}, needed to perform this encoding is:

$$T_{enc} = K(-\log f_i)e^{k\theta_i} \tag{1}$$

where f_i represents the frequency of the visual object being encoded (a value between 0.0 and 1.0), θ_i is its visual angle from the center of the current eye position, and K and k are constants.

The important components of this model, for these coding guidelines, are the logarithmic dependency on word frequency and the exponential decay based on the angle subtended by the word from the center of vision.

2.2 Individual word reading (English, French, and more?)

word reading individual

When presented with a single word, or the first word in a sentence, studies have found that readers tend to pick an eye fixation point near the center of that word. The so-called *preferred viewing location* is often toward the left of center. It had been assumed that selecting such a position enabled the reader to maximize

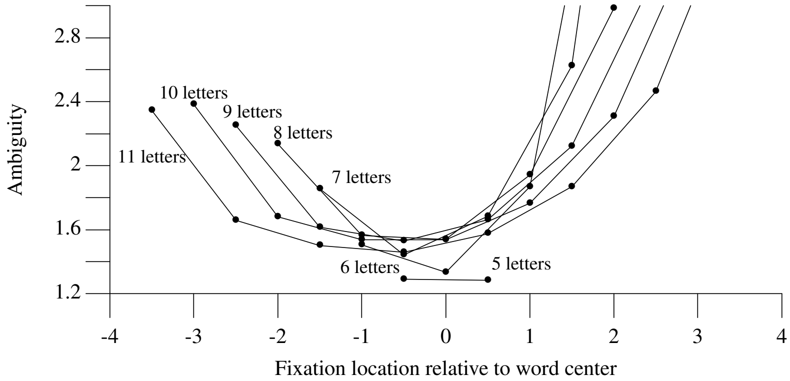

Figure 15: The ambiguity for patterns defined by the first and last letter and an interior letter pair as a function of the position of the first letter of the pair. Plots are for different word lengths using 65,000 words from CLAWS[19] (as used by the `aspell` tool). The fixation position is taken to be midway between the interior letter pair.

information about the letters in the word (fixating near the center would enable a reader to get the most information out of their eyes available field of view). Clark and O'Regan[5] performed a statistical analysis of several English word corpuses and a French word corpus. They assumed the reader would know the first and last letters of a word, and would have a choice of looking anywhere within a word to obtain information on more letters. (It was assumed that reliable information on two more letters would be obtained.)

Knowing only a few of the letters of a word can create ambiguity because there is more than one, human language, word containing those letters at a given position. For instance, some of the words matched by *s*at****d* include *scattered*, *spattered*, and *stationed*. The results (Figure 15) show that word ambiguity is minimized by selecting two letters near the middle of the word. Clark and O'Regan do not give any explanation for why English and French words should have this property, but they do suggest that experienced readers of these two languages make use of this information in selecting the optimal viewing position within a word.

word frequency

There are a number of experimental results that cannot be explained by an eye viewing position theory based only on word ambiguity minimization. For instance, the word frequency effect shows that high-frequency words are more easily recognized than low-frequency words. The ambiguity data shows the opposite effect. While there must be other reading processes at work, Clark and O'Regan propose that ambiguity minimization is a strong contributor to the optimal viewing position.

The need to read individual identifiers in source code occurs in a number of situations. Developers may scan down a list of identifiers looking for (1) the declaration of a particular identifier (where it is likely to be the last sequence of letters on a line) or (2) a modification of a particular identifier (where it is likely to be the first non-space character on a line).

If developers have learned that looking at the middle of a word maximizes their information gain when reading English text, it is likely this behavior will be transferred to reading source code. Identifiers in source code are rarely existing, human language, words. The extent to which experienced developers learn to modify their eye movements (if any modification is necessary) when reading source code is unknown. If we assume there is no significant change in eye movement behavior on encountering identifiers in source code, it can be used to predict the immediate information available to a developer on first seeing an identifier. Knowing this information makes it possible to select identifier spellings to minimize ambiguity with respect to other identifiers declared in the same program. This issue is discussed elsewhere.

identifier syntax

Calculating the ambiguity for different positions within C source code identifiers shows (see Figure 16) that the ambiguity is minimized near the center of the identifier and rises rapidly toward the end. However, there is a much smaller increase in ambiguity, compared to English words, moving toward the beginning of the identifier. Why English speakers and developers (the source code used for these measurements is

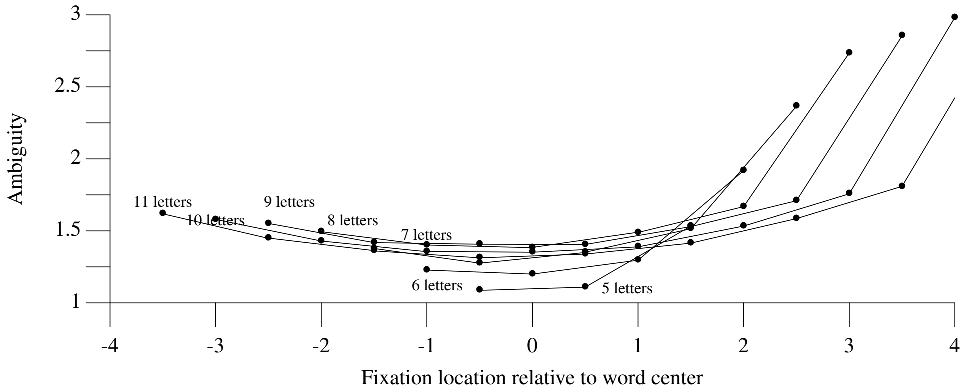

Figure 16: The ambiguity of source code identifiers, which can include digits as well as alphabetic characters. Plots are for different identifier lengths. The 344,000 identifiers in the visible form of the .c files were used.

likely to be predominantly written by English speakers but not necessarily native speakers) should create words/identifiers with this ambiguity minimization property is not known.

If native-English speakers do use four letters worth of information to guide identifier lookup, will they be misled by their knowledge of English words? Of the 344,000 unique identifiers (143,109 containing between 5 and 11 characters) in the .c files, only 1,549 corresponded to words in the CLAWS list of 65,000 words. The letter pattern counts showed the words containing a total of 303,518 patterns, to which the list of identifiers added an additional 1,576,532 patterns. The identifiers contained letters that matched against 166,574 word patterns (15,752 for center pair) and matched against 608,471 patterns that were unique to identifiers (49,524 for center pair).

These results show that more than 80% of letter patterns appearing in identifiers do not appear in English words. Also, identifier letter patterns are three times more likely to match against a pattern that is unique to identifiers than a pattern that occurs in an English word. In most cases developers will not be misled into thinking of an English word because four-letter patterns in identifiers do not frequently occur in English words.

2.3 White space between words

words
white space
between

The use of white space between tokens in source code is a controversial subject. The use of white space is said to affect *readability*, however that might be measured. The different reasons a developer has for reading source code, and the likely strategies adopted are discussed elsewhere.

[] reading
kinds of

Is the cost of ownership of source code that contains a space character, where permitted, between every identifier and operator/punctuator[2] less than or greater than the cost of ownership of source code that does not contain such space characters? This subsection discusses the issues; however, it fails to reach a definitive conclusion.

Readers of English take it for granted that a space appears between every word in a line of text. This was not always the case. Spaces between words started to be used during the time of Charlemagne (742–814); however, as late as the seventeenth century there was still some irregularity in how spaces were used to separate letters.[3] The spread of Latin to those less familiar with it, and the availability of books (through the introduction of the printing press) to people less skilled in reading, created a user-interface problem. Spaces between words simplified life for occasional users of Latin and improved the user friendliness of books for intermittent readers. The written form of some languages do not insert spaces between words (e.g., Japanese and Thai), while other written forms add some spaces but also merge sequences of words to form a single long word (e.g., German and Dutch). Algorithms for automating the process of separating

[2] In some cases a space character is required between tokens; for instance, the character sequence `const int i` would be treated as a single identifier if the space characters were not included.

words in unspaced text is an active research topic.[23]

ReadersofEnglishdonotneedspacesbetweenwords.Isitsimplylackofpracticethatreducesreadingrate?Thest udy-byKolers[16]showedwhatcouldbeachievedwithpractice.Readersofsourcecodedonotneedspaceseither(inafewco ntextsthesyntaxrequiresthem)`a=b+c`. The difference between English prose and source code is that identifier words are always separated by other words (operators and punctuation) represented by characters that cannot occur in an identifier.

A study by Epelboim, Booth, Ashkenazy, Taleghani, and Steinmans[11] did not just simply remove the spaces from between words, they also added a variety of different characters between the words (shaded boxes, digits, lowercase Greek letters, or lower case Latin letters). Subjects were not given any significant training on reading the different kinds of material.

Epelboim[11]

The following filler-placements were used (examples with digit fillers are shown in parentheses):

1. *Normal: Normal text (this is an example);*
2. *Begin: A filler at the beginning of each word, spaces preserved (1this 3is 7an 2example);*
3. *End: A filler after the end of each word, spaces preserved (this1 is3 an7 example2);*
4. *Surround: Fillers surrounding each word, spaces preserved (9this1 4is3 6an7 8example2);*
5. *Fill-1: A filler filling each space (9this2is5an8example4);*
6. *Fill-2: Two fillers filling each space (42this54is89an72example39);*
7. *Unspaced: Spaces removed, no fillers (thisisanexample).*

Table 1: Mean percent differences, compared to normal, in reading times (silent or aloud); the values in parenthesis are the range of differences. From Epelboim.[11]

Filler type	Surround	Fill–1	Fill–2	Unspaced
Shaded boxes (aloud)	4 (1–12)	—	3 (-2–9)	44 (25–60)
Digits (aloud)	26 (15–40)	26 (10–42)	—	42 (19–64)
Digits (silent)	40 (32–55)	41 (32–58)	—	52 (45–63)
Greek letters (aloud)	33 (20–47)	36 (23–45)	46 (33–57)	44 (32–53)
Latin letters (aloud)	55 (44–70)	—	74 (58–84)	43 (13–58)
Latin letters (silent)	66 (51–75)	75 (68–81)	—	45 (33–60)

Epelboim et al. interpreted their results as showing that fillers slowed reading because they interfered with the recognition of words, not because they obscured word-length information (some models of reading propose that word length is used by low-level visual processing to calculate the distance of the next saccade). They concluded that word-length information obtained by a low-level visual process that detects spaces, if used at all, was only one of many sources of information used to calculate efficient reading saccade distances.

Digits are sometimes used in source code identifiers as part of the identifier. These results suggest that digits appearing within an identifier could disrupt the reading of its complete name (assuming that the digits separated two familiar letter sequences). The performance difference when Greek letters were used as separators was not as bad as for Latin letters, but worse than digits. The reason might relate to the relative unfamiliarity of Greek letters, or their greater visual similarity to Latin letters (the letters α , δ, θ, μ, π, σ, τ, and ϕ were used). The following are several problems with applying the results of this study to reading source code.

- Although subjects were tested for their comprehension of the contents of the text (the results of anybody scoring less than 75% were excluded, the mean for those included was 89.4%), they were not tested for correctly reading the filler characters. In the case of source code the operators and punctuators between words contribute to the semantics of what is being read; usually it is necessary to pay attention to them.

- Many of the character sequences (a single character for those most commonly seen) used to represent C operators and punctuators are rarely seen in prose. There contribution to the entropy measure used to calculate saccade distances is unknown. For experienced developers the more commonly seen character sequences, which are also short, may eventually start to exhibit high-frequency word characteristics (i.e., being skipped if they appear in the parafoveal).
- Subjects were untrained. To what extent would training bring their performance up to a level comparable to the unfilled case?

A study by Kohsom and Gobet[14] used native Thai speakers, who also spoke English, as subjects (they all had an undergraduate degree and were currently studying at the University of Pittsburgh). The written form of Thai does not insert spaces between words, although it does use them to delimit sentences. In the study the time taken to read a paragraph, and the number of errors made was measured. The paragraph was in Thai or English with and without spaces (both cases) between words. The results showed no significant performance differences between reading spaced or unspaced Thai, but there was a large performance difference between reading spaced and unspaced English.

This study leaves open the possibility that subjects were displaying a learned performance. While the Thai subjects were obviously experienced readers of unspaced text in their own language, they were not experienced readers of Thai containing spaces. The Thai subjects will have had significantly more experience reading English text containing spaces than not containing spaces. The performance of subjects was not as good for spaced English, their second language, as it was for Thai. Whether this difference was caused by subjects' different levels of practice in reading these languages, or factors specific to the language is not known. The results showed that adding spaces when subjects had learned to read without them did not have any effect on performance. Removing spaces when subjects had learned to read with them had a significant effect on performance.

Further studies are needed before it is possible to answer the following questions:

- Would inserting a space between words and operators/punctuators reduce the source reading error rate? For instance, in `a=b*c` the `*` operator could be mistaken for the `+` operator (the higher-frequency case) or `&` operator (the lower frequency case).
- Would inserting a space between words and operators/punctuators reduce the source reading rate? For instance, in `d=e[f]` the proximity of the `[` operator to the word `e` might provide immediate semantic information (the word denotes an array) without the need for another saccade.
- What impact does adding characters to a source line have on the average source reading rate and corresponding error rate (caused by the consequential need to add line breaks in some places)?
- Are the glyphs used for some characters sufficiently distinctive that inserting space characters around them has no measurable impact?
- Do some characters occur sufficiently frequently that experienced developers can distinguish them with the same facility in spaced and unspaced contexts? [763] letter detection

The results of the prose-reading studies discussed here would suggest that high performance is achieved through training, not the use of spaces between words. Given that developers are likely to spend a significant amount of time being trained on (reading) existing source code, the spacing characteristics of this source would appear to be the guide to follow.

2.4 Other visual and reading issues

grouping spatial location
word visual recognition

There are several issues of importance to reading source code that are not covered here. Some are covered elsewhere; for instance, visual grouping by spatial location and visual recognition of identifiers. The question of whether developers should work from paper or a screen crops up from time to time. This topic is outside of the scope of these coding guidelines (see Dillon[10] for a review).

font

Choice of display font is something that many developers are completely oblivious to. The use of Roman, rather than Helvetica (or serif vs. sans serif), is often claimed to increase reading speed and comprehension. A study by Lange, Esterhuizen, and Beatty[8] showed that young school children (having little experience with either font) did not exhibit performance differences when either of these fonts was used. This study showed there were no intrinsic advantages to the use of either font. Whether people experience preferential exposure to one kind of font, which leads to a performance improvement through a practice effect, is not known. The issues involved in selecting fonts are covered very well in a report detailing "*Font Requirements for Next Generation Air Traffic Management Systems*".[4] For a discussion of how font characteristics affect readers of different ages, see Connolly.[7]

letter detection

A study by Pelli, Burns, Farell, and Moore[25] showed that 2,000 to 4,000 trials were all that was needed for novice readers to reach the same level of efficiency as fluent readers in the letter-detection task. They tested subjects aged 3 to 68 with a range of different (and invented) alphabets (including Hebrew, Devanagari, Arabic, and English). Even fifty years of reading experience, over a billion letters, did not improve the efficiency of letter detection. The measure of efficiency used was human performance compared to an ideal observer. They also found this measure of efficiency was inversely proportional to letter perimetric complexity (defined as, inside and outside perimeter squared, divided by *ink* area).

A number of source code editors highlight (often by using different colors) certain character sequences (e.g., keywords). The intended purpose of this highlighting is to improve program readability. Some source formatting tools go a stage further and highlight complete constructs (e.g., comments or function headers). A study by Gellenbeck[13] suggested that while such highlighting may increase the prominence of the construct to which it applies; it does so at the expense of other constructs.

A book by Baecker and Marcus[2] is frequently quoted in studies of source code layout. Their aim was to base the layout used on the principles of good typography (the program source code as book metaphor is used). While they proposed some innovative source visualization ideas, they seem to have been a hostage to some arbitrary typography design decisions in places. For instance, the relative frequent change of font, and the large amount of white space between identifiers and their type declaration, requires deliberate effort to align identifiers with their corresponding type declaration. While the final printed results look superficially attractive to a casual reader, they do not appear, at least to your author, to offer any advantages to developers who regularly work with source code.

3 Kinds of reading

reading kinds of

The way in which source code is read will be influenced by the reasons for reading it. A reader has to balance goals (e.g., obtaining accurate information) with the available resources (e.g., time, cognitive resources such as prior experience, and support tools such as editor search commands).

Foraging theory[36] attempts to explain how the behavioral adaptations of an organism (i.e., its lifestyle) are affected by the environment in which it has to perform and the constraints under which it has to operate. Pirolli and Card[27] applied this theory to deduce the possible set of strategies people might apply when searching for information. The underlying assumption of this theory is that: faced with information-foraging tasks and the opportunity to learn and practice, cognitive strategies will evolve to maximize information gain per unit cost.

Almost no research has been done on the different information-gathering strategies (e.g., reading techniques) of software developers. These coding guidelines assume that developers will adopt many of the strategies they use for reading prose text. A review by O'Hara[21] listed four different prose reading techniques:

O'Hara[21]

> ***Receptive Reading**. With this type of reading the reader receives a continuous piece of text in a manner which can be considered as approximating listening behavior. Comprehension of the text requires some portion of the already read text to be held in working memory to allow integration of meaning with the currently being read text.*
>
> ***Reflective Reading**. This type of reading involves interruptions by moments of reflective thought about the contents of the text.*
>
> ***Skim Reading**. This is a rapid reading style which can be used for establishing a rough idea of the text. This is useful in instances where the reader needs to decide whether the text will be useful to read or to decide which parts to read.*
>
> ***Scanning**. This is related to skimming but refers more specifically to searching the text to see whether a particular piece of information is present or to locate a piece of information known to be in the text.*

Deimel and Naveda[9] provide a teachers' guide to program reading. The topic of visual search for identifiers is discussed in more detail elsewhere.

[] identifier visual search

Readers do not always match up pairs of **if**/**else** tokens by tracing through the visible source. The source code indentation is often used to perform the matching, readers assuming that **if**/**else** tokens at the same indentation level are a matching pair. Incorrectly indented source code can lead to readers making mistakes.

```
void f(int i)
{
if (i > 8)
   if (i < 20)
      i++;
else
   i--;
}
```

Example

```
#define mkstr(x) #x

char *p = mkstr(@); /* implementation supports the @ extended character */
```

For those wanting to teach code reading skills, Deimel and Naveda[9] offers an instructors guide (the examples use Ada). Studying those cases where the requirement is to minimize readability[6] can also be useful.

Usage

Table 2 shows the relative frequency of the different kinds of tokens in a source file. Adding the percentages for preceded by space and first on line (or followed by space and last on line) does not yield 100% because of other characters occurring in those positions. Some tokens occur frequently, but contribute a small percentage of the characters in the visible source (e.g., punctuators). Identifier tokens contribute more than 40% of the characters in the .c files, but only represent 28.5% of the tokens in those files.

A more detailed analysis of spacing between individual punctuators is given elsewhere.

[] punctuator syntax

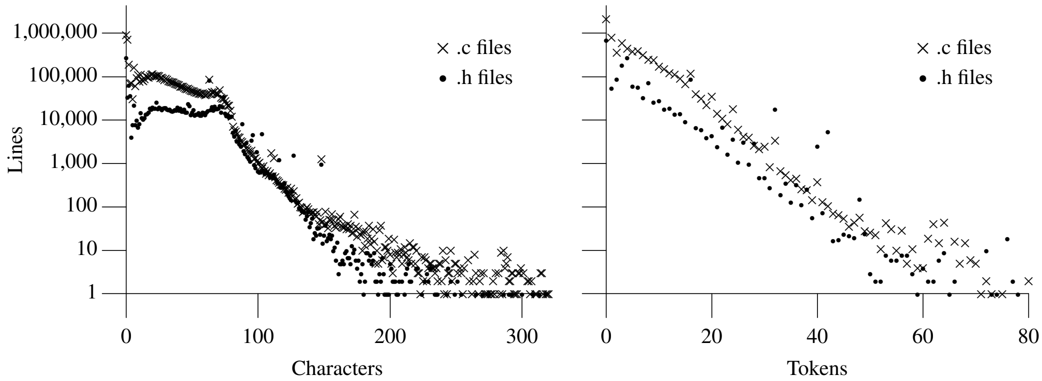

Figure 17: Number of non-white-space characters and tokens on a physical line; based on the visible form of the .c and .h files.

Table 2: Percentage of kinds of tokens in the visible form of the .c and .h files (value in parenthesis is the percentage of all non-white-space characters contained in those tokens), percentage occurrences (for .c files only) of token kind where it was preceded/followed by a space character, or started/finished a visible line. Comments are not tokens, but they are the only other construct that can contain non-white-space characters. The start of a preprocessing directive contains two tokens, but these are generally treated by developers, as being a single entity.

Token	% of tokens in .c files	% of tokens in .h files	% preceded by space	% followed by space	%first token on line	%last token on line
punctuator	53.5 (11.4)	48.1 (7.5)	27.5	29.7	3.7	25.3
identifier	29.8 (43.4)	20.8 (30.6)	54.9	27.6	1.4	1.2
constant	6.9 (3.8)	21.6 (15.3)	70.3	4.4	0.1	1.6
keyword	6.9 (5.8)	5.4 (4.2)	79.9	82.5	10.3	3.6
comment	1.9 (31.0)	3.4 (40.3)	53.4	2.2	41.2	97.4
string-literal	1.0 (4.6)	0.8 (2.2)	59.9	5.7	0.7	8.0
pp-directive	0.9 (1.1)	4.9 (4.4)	4.7	78.4	0.0	18.2
header-name	0.0 (0.0)	0.0 (0.0)	–	–	–	–

References

At the end of each entry, the pages on which that entry is cited are listed in parentheses.

[1] J. R. Anderson and C. Libiere. *The Atomic Components of Thought*. Lawrence Erlbaum Associates, 1998.

[2] R. Baecker and A. Marcus. *Human Factors and Typography for More Readable Programs*. Addison–Wesley, Reading, MA, USA, 1990.

[3] D. J. Boorstin. *The Discoverers*. Phoenix Press, 1983.

[4] S. Broadbent. Font requirements for next generation air traffic management systems. Technical Report HRS/HSP-006-REP-01, European Organisation for the Safety of Air Navigation, 2000.

[5] J. J. Clark and J. K. O'Regan. Word ambiguity and the optimal viewing position in reading. *Vision Research*, 39(4):843–857, 1998.

[6] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report Technical Report 148, Department of Computer Science, University of Auckland, 1997.

[7] G. K. Connolly. Legibility and readability of small print: Effects of font, observer age and spatial vision. Thesis (m.s.), University of Calgary, Alberta, Canada, Feb. 1998.

[8] R. W. De Lange, H. L. Esterhuizen, and D. Beatty. Performance differences between times and helvetica in a reading task. *Electronic Publishing*, 6(3):241–248, Sept. 1993.

[9] L. E. Deimel and J. F. Naveda. Reading computer programs: Instructor's guide and exercises. Technical Report CMU/SEI-90-EM-3 ADA228026, Software Engineering Institute (Carnegie Mellon University), 1990.

[10] A. Dillon. Reading from paper versus screens: a critical review of the empirical literature. *Ergonomics*, 35(10):1297–1326, 1992.

[11] J. Epelboim, J. R. Booth, R. Ashkenazy, A. Taleghani, and R. M. Steinmans. Fillers and spaces in text: The importance of word recognition during reading. *Vision Research*, 37(20):2899–2914, 1997.

[12] D. L. Fisher. Central capacity limits in consistent mapping, visual search tasks: Four channels or more? *Cognitive Psychology*, 16:449–484, 1984.

[13] E. M. Gellenbeck and C. R. Cook. Does signaling help professional programmers read and understand computer programs? In *Empirical Studies of Programmers: Fourth Workshop*, Papers, pages 82–98, 1991.

[14] C. Kohsom and F. Gobet. Adding spaces to Thai and English: Effects on reading. In *Proceedings of the* 19^{th} *Annual Meeting of Cognitive Science Society*, page ???, 1997.

[15] P. A. Kolers. Reading A year later. *Journal of Experimental Psychology: Human Learning and Memory*, 2(3):554–565, 1976.

[16] P. A. Kolers and D. N. Perkins. Spatial and ordinal components of form perception and literacy. *Cognitive Psychology*, 7:228–267, 1975.

[17] M. Kubovy and S. Gepshtein. Gestalt: From phenomena to laws. In K. L. Boyer and S. Sarkar, editors, *Perceptual Organization for Artificial Vision Systems*, chapter 5, pages 41–71. Kluwer Academic Publishers, Boston, 2000.

[18] F. Lavigne, F. Vitu, and G. d'Ydewalle. The influence of semantic context on initial eye landing sites in words. *Acta Psychologica*, 104:191–214, 2000.

[19] G. Leech, R. Garside, and M. Bryant. CLAWS4: The tagging of the British national corpus. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING 94)*, pages 622–628, Apr. 1994.

[20] G. E. Legge, T. S. Klitz, and B. S. Tjan. Mr. Chips: An ideal-observer model of reading. *Psychological Review*, 104(3):524–553, 1997.

[21] K. O'Hara. Towards a typology of reading goals. Technical Report Technical Report EPC-1996-107, Rank Xerox Research Centre, 1996.

[22] N. Osaka. Eye fixation and saccade during kana and kanji text reading: Comparison of English and Japanese text processing. *Bulletin of the Psychonomic Society*, 27(6):548–550, 1989.

[23] D. D. Palmer. A trainable rule-based algorithm for word segmentation. In P. R. Cohen and W. Wahlster, editors, *Proceedings of the Thirty-Fifth Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, pages 321–328. Association for Computational Linguistics, 1997.

[24] S. E. Palmer. *Vision Science: Photons to Phenomenology*. The MIT Press, 1999.

[25] D. G. Pelli, C. W. Burns, B. Farell, and D. C. Moore. Identifying letters. *Vision Research*, ???(???):???, 2002.

[26] R. Pieters and L. Warlop. Visual attention during brand choice: The impact of time pressure and task motivation. *International Journal of Research in Marketing*, 16:1–16, 1999.

[27] P. Pirolli and S. K. Card. Information foraging. *Psychological Review*, 106(4):643–675, 1999.

[28] A. Pollatsek, S. Bolozky, A. D. Well, and K. Rayner. Asymmetries in the perceptual span for Israeli readers. *Brain and Language*, 14:174–180, 1981.

[29] Z. Pylyshyn. Is vision continuous with cognition? the case for cognitive impenetrability of visual perception. *Behavioral and Brain Sciences*, 22(3):341–423, 1999.

[30] P. T. Quinlan and R. N. Wilton. Grouping by proximity or similarity? competition between gestalt principles in vision. *Perception*, 27:417–430, 1998.

[31] K. Rayner. Eye movements in reading and information processing: 20 years of research. *Psychology Bulletin*, 124(3):372–422, 1998.

[32] E. D. Reichle, A. Pollatsek, D. L. Fisher, and K. Rayner. Towards a model of eye movement control in reading. *Psychological Review*, 105(1):125–157, 1998.

[33] E. D. Reichle, K. Rayner, and A. Pollatsek. Comparing the E-Z reader model to other models of eye movement control in reading. *Behavioral and Brain Sciences*, ???(???):???, 2003.

[34] D. D. Salvucci. An integrated model of eye movements and visual encoding. *Cognitive Systems Research*, 1(4):201–220, 2001.

[35] M. L. Staples and J. M. Bieman. 3-D visualization of software structure. In M. Zelkowitz, editor, *Advances in Computers, Volume 49*, pages 96–143. Academic Press, Apr. 1999.

[36] D. W. Stephens and J. R. Krebs. *Foraging Theory*. Princeton University Press, 1986.

[37] A. Treisman and J. Souther. Search asymmetry: A diagnostic for preattentive processing of separable features. *Journal of Experimental Psychology: General*, 114(3):285–310, 1985.

[38] A. M. Treisman and G. Gelade. A feature-integration theory of attention. *Cognitive Psychology*, 12:97–136, 1980.

[39] P. Verghese and D. G. Pelli. The information capacity of visual attention. *Vision Research*, 32(5):983–995, 1992.

[40] C. Ware. *Information Visualization Perception for Design*. Morgan Kaufmann Publishers, 2000.