Asynchronous fixed-grid scanning with dynamic codes

Russ Beckley and Brian Roark
Center for Spoken Language Understanding, Oregon Health & Science University
{beckleyr,roark}@cslu.ogi.edu

Abstract

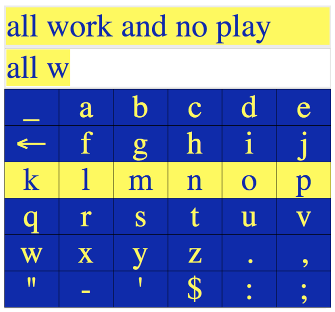
In this paper, we examine several methods for including dynamic, contextually-sensitive binary codes within indirect selection typing methods using a grid with fixed symbol positions. Using Huffman codes derived from a character n-gram model, we investigate both synchronous (fixed latency highlighting) and asynchronous (self-paced using long versus short press) scanning. Additionally, we look at methods that allow for scanning past a target and returning to it versus methods that remove unselected items from consideration. Finally, we investigate a novel method for displaying the binary codes for each symbol to the user, rather than using cell highlighting, as the means for identifying the required input sequence for the target symbol. We demonstrate that dynamic coding methods for fixed position grids can be tailored for very diverse user requirements.

1 Introduction

For many years, a key focus in Augmentative and Alternative Communication (AAC) has been providing text processing capabilities to those for whom direct selection of symbols on a keyboard (virtual or otherwise) is not a viable option. In lieu of direct selection, a binary (yes/no) response can be given through any number of switches, including buttons or pads that are pressed with hand, head, or foot, eyeblink detectors, or other switches that can leverage whatever reliable movement is available. These indirect selection methods typically involve systematically scanning through options and eliciting the binary yes/no response at each step of scanning. For example, row/column scanning is a very common approach for indirect selection. Auto row/column scanning on a square grid, such as that shown in Figure 1, will highlight each row in turn for some fixed duration (dwell time); if the binary switch is triggered before the dwell time expires, the row is selected; otherwise the next row is highlighted. Once a row is selected, cells in this row are then individually highlighted in turn, until one is selected, which identifies the intended character.

This sort of indirect selection method amounts to assigning a binary code to every symbol in the grid. If triggering the switch (e.g., pressing a button or blinking) is taken as a 'yes' or 1, then its absence is taken as a 'no' or 0. In such a way, every letter in the grid has a binary code based on the scanning strategy. For example, in Figure 1, the letter 'n' is in the third row and fourth column; if row scanning starts at the top, it takes two 'no's and a 'yes' to select the correct row; and then three 'no's and a 'yes' to select the correct column. This translates to a binary code of '0010001'.

In the preceding example, the codes for all symbols are determined by their position in the alpha-ordered grid. However, faster input can be achieved by assigning shorter codes to likely symbols. For example, imagine a user has just typed 'perso' and is ready to type the next letter. In this context, the letter 'n' is quite likely in English, hence if a very short code is assigned to that letter (e.g., '01'), then the user requires only two actions (a 'no' and a 'yes') to produce the letter, rather than the 7 actions re-



all work and no play
all w

_	a	b	c	d	e
←	f	g	h	i	j
k	l	m	n	o	p
q	r	s	t	u	v
w	x	y	z	.	,
"	-	'	$	:	;

Figure 1: Spelling grid in rough alpha order.

quired by the row/column code given above. There are methods for assigning codes that minimize the expected code length for a given probability model (Huffman, 1952). The quality of the probability model used for deriving codes can make a large difference in the code length and hence in the efficiency of the input method. When the model can accurately assign probabilities to symbols, the shortest binary codes can be assigned to the likeliest symbols, which thus require the fewest inputs (either yes or no) from the user. The best probabilistic models will take into account what has already been typed to assign probability to each symbol. The probabilities are contextually dependent, and therefore so are the optimal binary code assignments. This was illustrated in the 'person' example provided earlier. To provide another example, the probability of the letter 'u' is not particularly high overall in English (less than 0.02), but if the previously typed symbol is 'q', its probability is very high. Thus, in many contexts, there are other letters that should get the shortest code, but in that particular context, following 'q', 'u' is very likely, hence it should receive the shortest code.

Common scanning methods, however, present a problem when trying to leverage contextually sensitive language models for efficient scanning. In particular, methods of scanning that rely on highlighting contiguous regions – such as widely used row/column scanning – define their codes in terms of location in the grid, e.g., upper left-hand corner requires fewer keystrokes to select than lower right-hand corner using row/column scanning. To improve the coding in such an approach requires moving characters to short-code regions of the grid. In other words, with row/column scanning methods, the symbol needing the shortest code must move into the upper left-hand corner of the grid. Yet the cognitive overhead of dealing with frequent grid reorganization is typically thought to outweigh any speedup that is achieved through more efficient coding (Baletsa et al., 1976; Lesher et al., 1998). If one assumes a fixed grid, i.e., no dynamic reorganization of the symbols, then row/column scanning can gain efficiency by placing frequent characters in the upper left-hand corner, but cannot use contextually informed models. This is akin to Morse code, which assigns fixed codes to symbols based on overall frequency, without considering context.

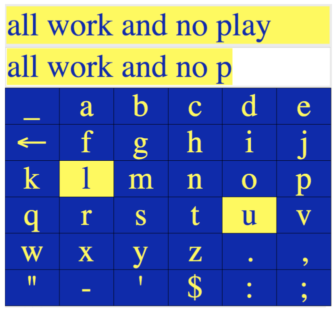


Figure 2: Scanning of non-contiguous sets of cells

Roark et al. (2010) presented a new approach which dropped the requirement of contiguous highlighting, thus allowing the use of variable codes on a fixed grid. For example, consider the grid in Figure 2, where two symbols in different rows and columns are jointly highlighted. This approach, which we will term "Huffman scanning", allowed the binary codes to be optimized using Huffman coding methods (see Section 2.2) with respect to contextually sensitive language models without dynamic reorganization of the grid. The method resulted in typing speedups over conventional row/column scanning.

One downside to the variable scanning that results from Huffman scanning is that users cannot anticipate their target symbol's binary code in any given context. In row/column scanning, the binary code of each symbol is immediately obvious from its location in the grid, hence users can anticipate when they will need to trigger the switch. In Huffman scanning, users must continuously monitor and react when their target cells light up. The time required to allow for this motor reaction means that scan rates are typically slower than in row/column scanning; and stress levels – due to the demands of immediate response to highlighting – higher.

Huffman scanning is not the only way to allow variable coding on a fixed grid. In this paper, we investigate alternatives to Huffman scanning that also allow for efficient coding on a fixed grid. The three alternative methods that we investigate are *asynchronous* methods, i.e., all of the scanning is self-paced; there is no scan rate that must be matched by the user. Rather than 'yes' being a button press and 'no' a timeout, these approaches, like Morse code, differentiate between short and long presses[1]. There are several benefits of this sort of asynchronous ap-

[1] Alternatively, two switches can be used.

proach: individuals who struggle with the timing requirements of auto, step or directed scanning can proceed without having to synchronize their movements to the interface; individuals can interrupt their communication – e.g., for side talk – for an arbitrary amount of time and come back to it in exactly the same state; and it reduces the stress of constantly monitoring the scanning sequence and reacting to it within the time limits of the interface.

The last of our alternative methods is a novel approach that displays the code for each symbol at once as a series of dots and dashes underneath the symbol – as used in Morse code – rather than using cell highlighting to prompt the user as in the other conditions. Unlike Morse code, these codes are derived using Huffman coding based on n-gram language models, thus change with every context. Since they are displayed for the user, no code memorization is required. This novel interface differs from Huffman scanning in several ways, so we also present intermediate methods that differ in only one or another dimension, so that we can assess the impact of each characteristic.

Our results show that displaying entire codes at once for asynchronous scanning was a popular and effective method for indirect selection, despite the fact that it shared certain dis-preferred characteristics with the least popular of our methods. This points the way to future work investigating methods to combine the preferred characteristics from our set of alternatives into a yet more effective interface.

2 Background and Related Work

2.1 Indirect selection

Some of the key issues influencing the work in this paper have already been mentioned above, such as the tradeoffs between fixed versus dynamic grids. For a full presentation of the range of indirect selection methods commonly in use, we refer the readers to Beukelman and Mirenda (1998). But in this section we will highlight several key distinctions of particular relevance to this work.

As mentioned in the previous section, indirect selection strategies allow users to select target symbols through a sequence of simpler operations, typically a yes/no indication. This is achieved by scanning through options displayed in the user interface. Beukelman and Mirenda (1998) mention circular scanning (around a circular interface), linear scanning (one at a time), and group-item scanning (e.g., row/column scanning to find the desired cell). Another variable in scanning is the speed of scanning – e.g., how long does the highlighting linger on the options before advancing. Finally, there are differences in selection control strategy. Beukelman and Mirenda (1998) mention automatic scanning, where highlighted options are selected by activating a switch, and advance automatically if the switch is not activated within the specified dwell time; step scanning, where highlighted options are selected when the switch is *not* activated within the specified dwell time, and advance only if the switch is activated; and directed scanning, where the highlighting moves while the switch is activated and selection occurs when the switch is *released*. In all of these methods, synchrony with the scan rate of the interface is paramount.

Speech and language pathologists working with AAC users must assess the specific capabilities of the individual to determine their best interface option. For example, an individual who has difficulty precisely timing short duration switch activation but can hold a switch more easily might do better with directed scanning.

Morse code, with its dots and dashes, is also an indirect selection method that has been used in AAC, but it is far less common than the above mentioned approaches due to the overhead of memorizing the codes. Once learned, however, this approach can be an effective communication strategy, as discussed with specific examples in Beukelman and Mirenda (1998). Often the codes are entered with switches that allow for easy entry of both dots and dashes, e.g., using two switches, one for dot and one for dash. In this study, we have one condition that is similar to Morse code in using dots and dashes, but without requiring code memorization[2]. The interface used for the experiments identifies dots and dashes with short and long keypresses.

[2]Thanks to a reviewer for pointing out that DynaVox Series 5 displays dynamically-assigned codes for non-letter buttons in their Morse code interface, much as we do for the entire symbol set. In contrast to our approach, their codes are not assigned using probabilistic models, rather to contrast with the standard Morse codes, which are used for the letters. Further, the cursor that we use to identify position within the code (see Section 3.5) is not used in the Dynavox interface.

2.2 Binary codes

In indirect selection, the series of actions required to select a given character is determined by the binary code. As mentioned in Section 1, row/column scanning assigns binary codes based on location within the grid. Ordering the symbols so that frequent characters are located in the upper left-hand corner of the grid will provide those frequent characters with short codes with a row/column scanning approach, though not the minimal possible binary codes. Given a probability distribution over symbols, there are known algorithms for building a binary code that has the minimum expected bits according to the distribution, i.e., codes will be optimally short (Huffman, 1952). The quality of the codes, however, depends on the quality of the probability model, i.e., whether the model fits the actual distribution in that context.

Roark et al. (2010) presented a scanning approach for a fixed grid that used Huffman codes derived from n-gram language models (see Section 2.3). The approach leveraged better probability models to achieve shorter code lengths, and achieved an overall speedup over row/column scanning for the 10 subjects in the trial, despite the method being closely tied to reaction time. The method requires monitoring of the target cell in the grid and reaction when it is highlighted, since the pattern of highlighting is not predictable from symbol position in the grid, unlike row/column scanning.

2.3 Language modeling

Language models assign probabilities to strings in the language being modeled, which has broad utility for many tasks in speech and language processing. The most common language modeling approach is the n-gram model, which estimates probabilities of strings as the product of the conditional probability of each symbol given previous symbols in the string, under a Markov assumption. That is, for a string $S = s_1 \dots s_n$ of n symbols, a k+1-gram model is defined as

$$\begin{aligned} \mathrm{P}(S) &= \mathrm{P}(s_1) \prod_{i=2}^{n} \mathrm{P}(s_i \mid s_1 \dots s_{i-1}) \\ &\approx \mathrm{P}(s_1) \prod_{i=2}^{n} \mathrm{P}(s_i \mid s_{i-k} \dots s_{i-1}) \end{aligned}$$

where the approximation is made by imposing the Markov assumption. Note that the probability of the first symbol s_1 is typically conditioned on the fact that it is first in the string. Each of the conditional probabilities in such a model is a multinomial distribution over the symbols in a vocabulary Σ, and the models are typically regularized (or smoothed) to avoid assigning zero probability to strings in Σ^*. See Chen and Goodman (1998) for an excellent overview of modeling and regularization methods.

For the current application, the conditional probability $\mathrm{P}(s_i \mid s_{i-k} \dots s_{i-1})$ can be used to assign probabilities to all possible next symbols, and these probabilities can be used to assign Huffman codes. For example, if the user has typed 'the perso' and is preparing to type the next letter, we estimate $\mathrm{P}(\text{ n} \mid \text{ t h e \_ p e r s o })$ as well as $\mathrm{P}(\text{ m} \mid \text{ t h e \_ p e r s o })$ and every other possible next symbol, from a large corpus. Note that smoothing methods mentioned above ensure that every symbol receives non-zero probability mass. Also note that the space character (represented above as '_') is a symbol in the model, hence the models take into account context across word boundaries. Given these estimated probabilities, known algorithms for assigning Huffman codes are used to assign short codes to the most likely next symbols, in a way that minimizes expected code length.

3 Methods

Since this paper aims to compare new methods with Huffman scanning presented in Roark et al. (2010), we follow that paper in many key respects, including training data, test protocol, and evaluation measures. For all trials we use a 6×6 grid, as shown in Figures 1 and 2, which includes the 26 characters in the English alphabet, 8 punctuation characters (comma, period, double quote, single quote, dash, dollar sign, colon and semi-colon), a white space delimiter (denoted with underscore) and a delete symbol (denoted with ←). Unlike Roark et al. (2010), our grid is in rough alphabetic order rather than in frequency order. In that paper, they compared Huffman scanning with row/column scanning, which would have been put at a disadvantage with alphabetic order, since frequent characters would have received longer codes than they do in a frequency ordered grid. In this paper, however, all of the approaches are using Huffman codes and scanning of possibly non-contiguous subsets of characters, so the code efficiency does not depend on location in the grid. Thus for ease of visual scanning, we chose in this study to use alphabetic ordering.

3.1 Language models and binary codes

We follow Roark et al. (2010) and build character-based smoothed 8-gram language models from a normalized 42M character subset of the English gigaword corpus and the CMU pronunciation dictionary. This latter lexicon is used to increase coverage of words that are unobserved in the corpus, and is included in training as one observation per word in the lexicon. Smoothing is performed with a generalized version of Witten-Bell smoothing (Witten and Bell, 1991) as presented in Carpenter (2005). Text normalization and smoothing parameterizations were as presented in Roark et al. (2010). Probability of the delete symbol $\leftarrow$ was taken to be 0.05 in all trials (the same as the probability of an error, see Section 3.2), and all other probabilities derived from the trained n-gram language model.

3.2 Huffman scanning

Our first scanning condition replicates the Huffman scanning from Roark et al. (2010), with two differences. First, as stated above, we use an alphabetic ordering of the grid as shown in Figure 2, in place of their frequency ordered grid. Second, rather than calibrating the scan rate of each individual, we fixed the scan rate at 600 ms across all subjects.

One key aspect of their method is dealing with errors of omission and commission, i.e., what happens when a subject misses their target symbol. In standard row/column scanning, rows are highlighted starting from the top of the grid, incrementing downwards one row at a time. If no row has been selected after iterating through all rows, the scanning begins again at the top. In such a way, if the subject mistakenly neglects to select their intended row, they can just wait until it is highlighted again. Similarly, if the wrong row is selected, there is usually a mechanism whereby the columns are scanned for some number of iterations, at which point row scanning resumes. The upshot of this is that users can make an error and still manage to select their intended symbol after the scanning system returns to it.

Roark et al. (2010) present a method for allowing the same kind of robustness to error in Huffman scanning, by recomputing the Huffman code after every bit. If the probability that the bit was correct is p, then the probability that it was incorrect is $1-p$. In Huffman scanning, a subset is highlighted and the user indicates yes or no – yes, the target symbol is in the set; or no, the target symbol is not in the set. If the answer is 'yes' and the set includes exactly one symbol, it is typed. Otherwise, for all symbols in the selected set (highlighted symbols if 'yes'; non-highlighted if 'no'), their probabilities are multiplied by p (the probability of being correct), while the probabilities of the other set of symbols are multiplied by $1-p$. The probabilities are then re-normalized and a new Huffman code is generated, the first bit of which drives which symbols are highlighted at the next step. In such a way, even if the target symbol is in the highlighted set when it is not selected (or vice versa), it is not eliminated from consideration; rather its probability is diminished (by multiplying by $1-p$, which in this paper is set to 0.05) and scanning continues. Eventually the symbol will be highlighted again, much as is the case in row/column scanning. We also use this method within the Huffman scanning condition reported in this paper.

3.3 Asynchronous scanning

Our second condition replaces the scan rate of 600 ms from the Huffman scanning approach outlined in Section 3.2 with an asynchronous approach that does not rely upon a scan rate. The grid and scanning method remain identical, but instead of switch versus no switch, we use short switch (rapid release) versus long switch (slower release). This is similar to the dot/dash distinction in Morse code. For this paper, we used a threshold of 200 ms to distinguish a short versus a long switch, i.e., if the button press is released within 200 ms it is short; otherwise long. Since Huffman scanning already has switch activation as 'yes', this could be thought of as having the long press replace no-press in the interface.

With this change, the scanning does not automatically advance to the next set, but waits indefinitely for the user to enter the next bit of the code. The same method for dealing with errors as with Huffman scanning is employed in this condition, i.e., re-

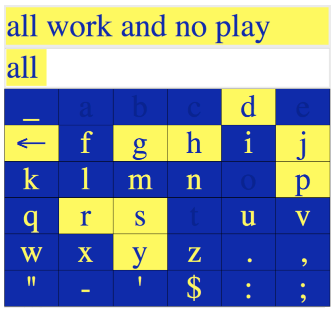


Figure 3: Scanning of non-contiguous sets of cells, with symbols that have been eliminated from consideration deemphasized (a, b, c, e, o, t)

computing the Huffman code after every bit and taking into account the probability of the bit being in error. One might see this as a self-paced version of Huffman scanning.

One benefit of this approach is that it does not require the user to synchronize their movements to a particular scan rate of the interface. One potential downside for some users is that it does require more active keypresses than auto scanning. In auto scanning, only the '1' bits of the code require switch activation; the '0' bits are produced passively by waiting for the dwell time to expire. In contrast, all bits in the asynchronous approaches require one of two kinds of switch activation.

3.4 Not returning to non-selected symbols

Our third condition is just like the second except it does not recompute the Huffman codes after every bit, changing the way in which user errors are handled. At the start of the string or immediately after a letter has been typed, the Huffman codes are calculated in exactly the same way as the previous two conditions, based on the n-gram language model given the history of what has been typed so far. However, after each bit is entered for the current symbol, rather than multiplying by p and $1-p$ as detailed in Section 3.2, symbols that have not been selected are eliminated from consideration and will not be highlighted again, i.e., will not be returned to for subsequent selection. For example, in Figure 3 we see that there is a set of highlighted characters, but also a set of characters that have been eliminated from consideration and are deemphasized in the interface to indicate that they can no longer be selected (specifically: a, b, c, e, o and t). Those are symbols that were not selected in previous steps of the scanning, and are no longer available to be typed in this position. If the user makes a mistake in the input, eliminating the actual target symbol, the only way to fix it is to type another symbol, delete it, and retype the intended symbol.

This condition is included in the study because recalculation of codes after every bit becomes problematic when the codes are explicitly displayed (the next condition). By including these results, we can tease apart the impact of not recalculating codes after every bit versus the impact of displaying codes in the next condition. Later, in the discussion, we will return to this characteristic of the interface and discuss some alternatives that may allow for different error recovery strategies.

This change to the interface has a couple of implications. First, the optimal codes are slightly shorter than with the previous Huffman scanning methods, since no probability mass is reserved for errors. In other words, the perfect user that never makes a mistake would be able to type somewhat faster with this method, which is not surprising, since reserving probability for returning to something that was rejected is of no utility if no mistakes are ever made. The experimental results presented later in the paper will show explicitly how much shorter the codes are for our particular test set. Second, it is possible to type a symbol without ever actively selecting it, if all other symbols in the grid have been eliminated. For example, if there are two symbols left and the system highlights one symbol, which is rejected, then the other symbol is typed. This contrasts with the previous methods that only type when a single character set is actively selected.

3.5 Displaying codes

Our final condition also does not recompute codes after every bit, but in addition does away with highlighting of cells as the mechanism for scanning, and instead displays dots and dashes directly beneath each letter in the fixed grid. For example, Figure 4 shows the dots and dashes required for each letter directly below that letter in the grid, and Figure 5 shows a portion of that grid magnified for easier detailed viewing. Each code includes the dots and dashes required to input that symbol, plus a cursor '|' that indicates how much of the code has already

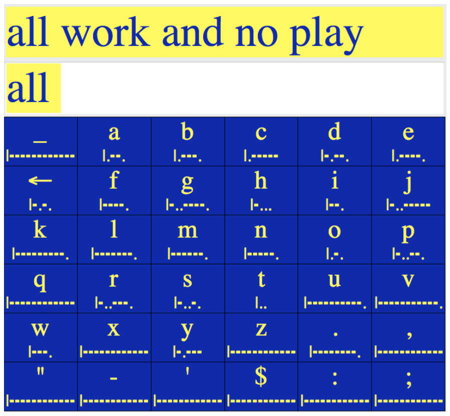


Figure 4: Scanning of non-contiguous sets of cells, displaying dots and dashes rather than highlighting

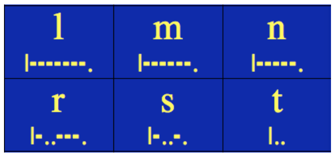


Figure 5: A magnification of part of the above grid

been entered. For example, to type the letter ‘s’ using the code in Figure 5 , one must input: long, short, short, long, short.

Since these codes are displayed, there is no memorization required to input the target symbol. Like row/column scanning, once the target symbol has been found in the grid, the input sequence is known in entirety by the user, which can facilitate planning of sequences of actions rather than simply reacting to updates in the interface. The cursor helps the user know where they are in the code, which can be helpful for long codes. Figure 6 shows a magnification of the interface when there are only two options remaining – a dot selects ‘l’ and a dash selects ‘u’.

4 Experiments

We recruited 10 native English speaking subjects between the ages of 26 and 50 years, who are not users of scanning interfaces for typing and have typical motor function. Following Roark et al. (2010), we use the phrase set from MacKenzie and Soukoreff (2003) to measure typing performance, and the same five strings from that set were used as evaluation strings in this study as in Roark et al. (2010). Practice strings were randomly selected from the rest of the phrase set. Subjects used an Ablenet Jellybean® button as the binary switch. The error rate parameter was fixed at 5% error rate.

The task in all conditions was to type the presented phrase exactly as it is presented. Symbols that are typed in error – as shown in Figure 7 – must be repaired by selecting the delete symbol (←) to delete the incorrect symbol, followed by the correct symbol. The reported times and bits take into account the extra work required to repair errors.

We tested subjects under four conditions. All four conditions made use of 8-gram character language models and Huffman coding, as described in Section 3.1, and an alpha-ordered grid. The first condition is a replication of the Huffman scanning condition from Roark et al. (2010), with the difference in scan rate (600ms versus mean 475ms in their paper) and the grid layout. This is an auto scan approach, where the highlighting advances at the end of the dwell time, as described in Section 3.2. The second condition is asynchronous scanning, i.e., replacing the dwell time with a long button press as described in Section 3.3, but otherwise identical to condition 1. The third condition was also asynchronous, but did not recompute the binary code after every bit, so that there is no return to characters eliminated from consideration, as described in Section 3.4, but otherwise identical to condition 2. Finally, the fourth condition

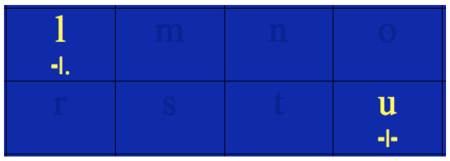


Figure 6: Cursor shows how much of code has been entered

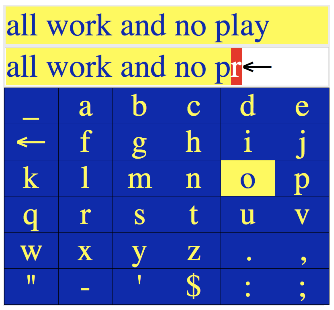


Figure 7: After an incorrect symbol is typed, it must be deleted and the correct symbol typed in its place

Scanning condition		Speed (cpm) mean (std)	Bits per character mean (std)	Bits per character opt.	Error rate mean (std)	Long code rate mean (std)
1.Huffman synchronous	Roark et al. (2010)	23.4 (3.7)	4.3 (1.1)	2.6	4.1 (2.2)	19.3 (14.2)
	This paper	25.5 (3.2)	3.3 (0.4)	2.6	1.8 (1.1)	7.3 (4.1)
2. Huffman asynchronous		20.0 (3.7)	3.1 (0.2)	2.6	3.1 (2.5)	3.8 (1.2)
3. Huffman asynch, no return		17.2 (3.2)	3.1 (0.3)	2.4	7.7 (2.7)	0 (0)
4. Huffman asynch, display codes		18.7 (3.9)	3.0 (0.3)	2.4	6.9 (2.5)	0 (0)

Table 1: Typing results for 10 users on 5 test strings (total 31 words, 145 characters) under 4 conditions.

displays the codes for each character as described in Section 3.5, without highlighting, but is otherwise identical to condition 3.

Subjects were given a brief demo of the four conditions by an author, then proceeded to a practice phase. Practice phrases were given in each of the four conditions, until subjects reached sufficient proficiency in the method to type a phrase with fewer than 10% errors. After the practice phases in all four conditions were completed, the test phases commenced. The ordering of the conditions in the test phase was random. Subjects again practiced in a condition until they typed a phrase with fewer than 10% errors, and then were presented with the five test strings in that condition. After completion of the test phase for a condition, they were prompted to fill out a short survey about the condition.

Table 1 presents means and standard deviations across our subjects for characters per minute, bits per character, error rate and what Roark et al. (2010) termed "long code rate", i.e., percentage of symbols that were correctly selected after being scanned past. For condition 1, we also present the result for the same condition reported in Roark et al. (2010). Comparing the first two rows of that table, we can see that our subjects typed slightly faster than those reported in Roark et al. (2010) in condition 1, with fewer bits per character, mainly due to lower error rates and less scanning past targets. This can be attributed to either the slower scanning speed or the alphabetic ordering of the grid (or both). In any case, even with the slower scan rate, the overall speed is faster in this condition than what was reported in that paper.

The other three conditions are novel to this paper. Moving from synchronous to asynchronous (with long press) but leaving everything else the same

Survey Question	Huffman synch	Huffman asynch	No return	Display codes
Fatigued	2.1	3.2	3.4	2.5
Stressed	1.9	2.2	2.9	2.0
Liked it	3.8	3.0	2.3	3.5
Frustrated	1.9	2.8	4.0	2.4

Table 2: Mean Likert scores to survey questions (5 = a lot; 1 = not at all)

(condition 2) leads to slower typing speed but fewer bits per character. The error rate is higher than in the synchronous condition 1, but there is less scanning past the target symbol. In discussion with subjects, the higher error rate might be attributed to losing track of which button press (short or long) goes with highlighting, or also to intended short presses being registered by the system as long.

The final two conditions allow no return to characters once they have been scanned past, hence the "long code rates" go to zero, and the error rates increase. Note that the optimal bits per character are slightly better than in the other trials, as mentioned in Section 3.4, yet the subject bits per character stay mostly the same as with condition 2. Typing speed is slower in these two conditions, though slightly higher when the codes are displayed versus the use of highlighting.

In Table 2 we present the mean Likert scores from the survey. The four statements that subjects assessed were:

1. I was fatigued by the end of the trial
2. I was stressed by the end of the trial
3. I liked this trial
4. I was frustrated by this trial

The scores were: 1 (not at all); 2 (a little); 3 (not sure); 4 (somewhat) and 5 (a lot).

The results in Table 2 show high frustration and stress with condition 3, and much lower fatigue, stress and frustration (hence higher 'liking') for condition 4, where the codes are displayed. Overall, there seemed to be a preference for Huffman synchronous, followed by displaying the codes.

5 Discussion

There are several take-away lessons from this experiment. First, the frustration and slowdown that result from the increased error rates in condition 3 make this a dispreferred solution, even though disallowing returning to symbols that have been ruled out in scanning reduced the bits per character (optimal and in practice). Yet in order to display a stable code in condition 4 (which was popular), recalculation of codes after every bit (as is done in the first two conditions) is not an option. To make condition 4 more effective, some effective means for allowing scanning to return to symbols that have been scanned past must be devised.

Second, asynchronous scanning does seem to be a viable alternative to auto scanning, which may be of utility for certain AAC users. Such an approach may be well suited to individuals using two switches for asynchronous row/column scanning. Other users may find the increased level of switch activation required for scanning in these conditions too demanding. One statistic not shown in Table 1 is number of keypresses required. In condition 1, some of the "bits" required to type the character are produced by *not* pressing the button. In the other three conditions, all "bits" result from either a short or long press, so the button is pressed for every bit. In condition 1, the mean number of key presses per character was 1.5, which is approximately half of the total button presses required per character in the other methods.

Future directions include investigations into methods that combine some of the strengths of the various approaches. In particular, we are interested in methods that allow for the direct display of codes for either synchronous or asynchronous scanning, but which also allow for scanning past and return to target characters that were mistakenly not selected. The benefit of displaying codes – allowing for anticipation and planning in scanning – are quite high, and this paper has not exhausted the exploration of such approaches. Among the alternatives being considered are: requiring all codes to have a short press (confirmation) bit as the last bit of the code; having a "reset" symbol or gesture; and recalculating codes after some number of bits, greater than one. Each of these methods would somewhat increase the optimal bits per character, but may result in superior user performance. Finally, we intend to include active AAC users in subsequent studies of these methods.

References

G. Baletsa, R. Foulds, and W. Crochetiere. 1976. Design parameters of an intelligent communication device. In *Proceedings of the 29th Annual Conference on Engineering in Medicine and Biology*, page 371.

D. Beukelman and P. Mirenda. 1998. *Augmentative and Alternative Communication: Management of Severe Communication Disorders in Children and Adults*. Paul H. Brookes, Baltimore, MD, second edition.

B. Carpenter. 2005. Scaling high-order character language models to gigabytes. In *Proceedings of the ACL Workshop on Software*, pages 86–99.

Stanley Chen and Joshua Goodman. 1998. An empirical study of smoothing techniques for language modeling. Technical Report, TR-10-98, Harvard University.

D.A. Huffman. 1952. A method for the construction of minimum redundancy codes. In *Proceedings of the IRE*, volume 40(9), pages 1098–1101.

G.W. Lesher, B.J. Moulton, and D.J. Higginbotham. 1998. Techniques for augmenting scanning communication. *Augmentative and Alternative Communication*, 14:81–101.

I.S. MacKenzie and R.W. Soukoreff. 2003. Phrase sets for evaluating text entry techniques. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*, pages 754–755.

B. Roark, J. de Villiers, C. Gibbons, and M. Fried-Oken. 2010. Scanning methods and language modeling for binary switch typing. In *Proceedings of the NAACL-HLT Workshop on Speech and Language Processing for Assistive Technologies (SLPAT)*, pages 28–36.

I.H. Witten and T.C. Bell. 1991. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094.