Implementing VisiCalc

Preface

I'm writing this in preparation for the Computer History Museum's *The Origins and Impact of VisiCalc* panel on April 8th 2003. This is basically a draft and I hope to do some more editing as time permits and you should expect many typos until then. I'm also going to continue to edit and change this as I remember details.

This version incorporates corrections and suggestions from readers. Normally I would just post the update but for the sake of purity I'm going to leave V1 available, at least for now and will continue to repost this as I make corrections and improvements. For those interested in the changes (if I remember them aside from typos and miscalculating 3+5*4)

- Added a reference for RPN
- Added screen shots from the reference card at http://www.bricklin.com/history.
- Added comments about memory and third party enhancements.
- Added a note from John Doty 2003-05-23

I will post incremental updates and try to catch up with a summary of what has changed in this section.

Introduction

This is my long-delayed attempt at writing about my experience in writing VisiCalc and the many design decisions that we made along the way. But even after nearly a quarter century I remember many of the details though maybe my memories have evolved. The process of writing down this experience is already evoking many memories and, unless proven otherwise, I'll assume that they are memories of real events but others may view it differently and I will try to correct the more creative aspects of my memory.

Even simple decisions were only simple in context. They were all intertwined and I will try to reduce the confusion by separating aspects of implementation, design and business.

For more details on the history of VisiCalc and even a version that still runs on the IBM PC, see Dan Bricklin's VisiCalc History pages.

Getting Started

I started to program VisiCalc in November 1978 and we shipped the first production copy in October 1979. The idea itself had been percolating in Dan's head long before and we had discussed various approaches over the year before I started to program it. At one point we considered implementing the program on the DEC PDT (Programmable Data Terminal) which was a small computer that fit inside a VT (Video Terminal) -100 which was a character-based computer terminal. It would have been expensive and aimed at high level meetings. We were lucky that we didn't make a deal with DEC.

As Dan described the product I envisioned a group of people sitting around a table with small devices pointing at a screen. Each had the ability to draw on the shared screen with graphics and formulas. The formulas would be recalculated as needed. This seemed reasonable give the technology of the day such as the Spatial Data Management System developed at MIT Architecture Machine Group, the predecessor of the MIT Media Lab.

The big breakthrough was when Dan put together a simple version in Integer Basic on the Apple][. It had a grid of rows and columns. While the use of a character grid with rows and columns seems uninteresting compared with a shared graphics screen it was the key to making the product usable because it gave people a framework to work with. It wasn't necessary to describe the equations since they were easily and implicitly defined by their position on the grid. We also dispensed with a pointing device since game paddle for the machine wasn't up to the task.

The Apple's screen was 40 columns and 25 rows. This was a small area and it was easy to move around using the arrow keys on the keyboard. Since everything could be done using the keyboard, proficient users would work very quickly.

Dan was attending Harvard Business School so asked me to help him by writing the program. I had already been programming on the 6502 and also did a project to convert a program from the TRS-80 to the Apple][simply to become proficient. I made a listing of the TRS-80 program by using my SX-70 Polaroid camera to take a picture of each page and then worked with this listing as I rewrote the code for the Apple.

In November of 1978 I started to prototype VisiCalc. We eventually shipped that prototype.

Background

By the time we created VisiCalc Dan and I had been working professional and academically for well over a decade. I started programming in Junior High School in 1963 and had been creating online services professionally since 1966. Those of us working in online systems in those days would have full responsibility how the program was used, not just the implementation details. While there were corporate projects that had a whole raft of people breaking the design and implementation project into small steps, many of us worked from ideas and then adjusted the programs as we gained experience. Working with online systems we deployed the program by simply giving others access and could quickly evolve the program as we learned how it was actually used. The process worked best when we were also users.

I did both commercial software at Interactive Data and was a student at MIT where I worked on the Multics project. Multics was a very influential project in that its goal was to make computing accessible to nontechnical users. It was also managed as an open source project within the development group. There were very few computer projects in those days so any large project would attract the best people available yet there was remarkable little reluctance to share with others and trade code.

We were able to treat the large mainframe computers as personal computers. We focused on making the program and the experience rather than the limitations of the smaller systems. The smaller less expensive systems were also valuable in that they allowed for more interaction with the user. The early systems were run in what was called full-duplex or echo-plex. When you typed a character on the keyboard of a teletype nothing printed. The computer system would normally send the character you pressed back to the teletype so you would think that it acted like a typewriter but that was only an illusion since there was no intrinsic relationship.

When we started to use screens instead of teletype we had the freedom to paint the screen in two dimensions. There were interactive editors for teletypes--today's VI is a descendent of QED on the SDS-940 which did just this. You were always editing the previous line. At MIT Richard Stallman added a redisplay capability to the Teco editor which allowed others to create sets of macros. One set was called Emacs (Editors Macros) become most popular and eventually was treated as the native editor. Later I implemented a version of Emacs for use at Software Arts and we traded it to Prime Computer for a disk drive (they used to be very expensive). The interactivity of Emacs provided a good example of the independence of what you did with the keyboard and what you saw on the screen.

Dan did a commercial word processor, WPS-8 at DEC that paid careful attention to making the users feel comfortable with editing on this new device by making it simple and familiar.

I also worked with a company, ECD that produced a 6502-based computer. It gave me a lot of experience with the 6502. One of the programmers, John Doty, created a useful assembly language for the machine that included macros to eliminate the need to use jump (or goto) instructions. It ran on Multics and we'd download the code to the 6502.

In addition to the commercial time sharing experience and the experience with Multics at MIT and the experience using the more interactive systems at the AI Lab, languages like Lisp and +AIU- well there was a lot of experience. I also founded the Student Information Processing Board as a way to make this computing bounty available to others students.

Design Principles

VisiCalc was a product, not a program. Decisions were made with the product in mind and, to the extent possible the programming was towards this end. In practice it was more complicated as we were designing against the limitations of the personal computers, price point and, most important, what the user could understand.

The goal was to give the user a conceptual model which was unsurprising -- it was called the principle of least surprise. We were illusionists synthesizing an experience. Our model was the spreadsheet -- a simple paper grid that would be laid out on a table. The paper grid provided an organizing metaphor for a working with series of numbers. While the spreadsheet is organized we also had the back-of-envelope model which treated any surface as a scratch pad for working out ideas. Since we were used to working with powerful computers without worry about the clock running, we already had the experience of focusing on the user’s needs rather than the computer’s needs.

The ability for Dan and I to work as a team was crucial. While he could've written the program, the fact that he wasn't gave him the freedom to focus on what the program should do rather than how to do it. I could appreciate his reasons and would eventually accept that I had to change code that I had labored over. We were able to find ways to take advantage of the limited space available for the program in deciding what features to include or not include.

The original version put the entry area at the bottom of the screen. By playing with this simple prototype Dan found that it was better to put the entry area at the top of the screen and I made the change to the evolving program.

In addition to prototyping, Dan put together a reference card for users. If we couldn't figure out how to explain a feature on the reference card we would change the program. The original method for copying formulas was too complicated so we just changed the design rather than try to explain it.

The Apple][

In 1978 the Apple][was viewed as a game machine. In fact, it was intended to be a hobbyist game machine. It had up to 64KB (that's kilo bytes) or 65336 8 bit bytes, or 2^16 compared with today's PC's which now have 2^29 (512 Megabytes) or 8000 (ok, 8192) times as much memory. We had no hard drive. Apple had cornered the market for floppy drives but they weren't universal so we supported the cassette tape player as a storage device but, fortunately, few users even know about it.

There was no way to start or stop the tape drive. We had to leave gaps in the data on the tape to allow for processing of each chunk of data before we got the next one.

Our goal was to fit in a 16KB machine but eventually we required 32KB. That included a 1KB buffer for the screen and more memory for the needs of the system. We also needed to implement a file system for the floppies as well as the firmware to support the drive. Steve Wozniak did a very clever and lean design and took advantage of the 6502 processor to control the disk drive as well as for computing. I had to figure out how he did this by reverse engineering it since we need to adjust the code for our needs. We also wanted to avoid being beholden to Apple's license which gave them the ability to revoke permission. More on that and the challenges later.

The Apple][was very fast compared with using our fingers? It was capable of doing maybe a million instructions per second but each one was very simple, a small portion of a single instruction in today's computers. Performance was just almost as important as space.

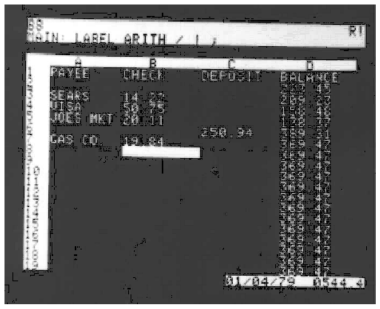
There were many many little design decisions. In order to keep things a bit organized I will group them in categories though many overlapped:

The User Experience

These are the design decisions visible to the user or, often, the lack of visibility was essential.

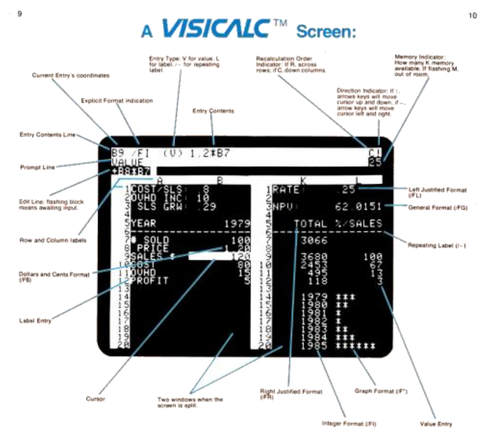
The Layout of the Screen

I started programming VisiCalc late November 1978 and by January we were able to demonstrate simple applications such as the one on the right. By then the screen was already looking like the production version (as in the example on the left taken from the first reference card).



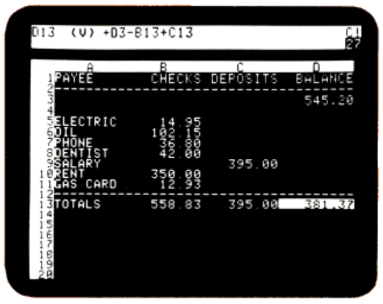
We had already arrived at the essential elements of the layout.

For more details you can look at the annotated screen on the back of the reference card.


9
10
A VISICALC™ Screen:
Entry Type: V for value, L for label, /-- for repeating label.
Recalculation Order Indicator: If R, across rows; if C, down columns.
Memory Indicator: How many K memory available. If flashing M, out of room.
Current Entry's coordinates
Explicit Format indication
Entry Contents
Direction Indicator: If !, arrows keys will move cursor up and down; if --, arrow keys will move cursor left and right.
Entry Contents Line
Prompt Line
Edit Line: flashing block means awaiting input.
Row and Column labels
Dollars and Cents Format (/F$)
Label Entry
Left Justified Format (/FL)
General Format (/FG)
Repeating Label (/--)
Cursor
Two windows when the screen is split.
Right Justified Format (/FR)
Integer Format (/FI)
Graph Format (/F*)
Value Entry

```
B9 /FI  (V) 1.2*B7                          C!
VALUE                                       25
+B8*B7
          A          B              K          L
 1COST/SLS:  .8            1RATE:        .25
 2OVHD INC:  10            2
 3 SLS GRW:  .29           3NPV:     62.0151
 4                         4
 5YEAR            1979     5     TOTAL %/SALES
 6-------------------      6------------------
 7# SOLD           100     7     3066
 8 PRICE          1.20     8
 9SALES $          120     9     3680      100
10COST              80    10     2453       67
11OVHD              15    11      495       13
12PROFIT             5    12      118        3
13                        13
14                        14     1979 ***
15                        15     1980 **
16                        16     1981 *
17                        17     1982 *
18                        18     1983 **
19                        19     1984 ***
20                        20     1985 ******
```

- The inverted L framed the grid though we didn't show actually grid lines since the screen was small.
- The status information was shown at the top:
  - B8 showed where the position of the cursor though you normally just thought of it as "there".
  - The R showed that we were doing row-order calculation.
  - The “!” meant that that two arrow keys went up and down.
  - The amount of memory available was later added to the status line. We had to be careful to make this number match the number shown in the manual since users didn't necessarily distinguish accidental properties from intrinsic properties so we need to be careful about even something that seemed obvious. By the time we shipped it was increasingly common for machines to have 48KB of memory which seemed a lot.
  - The black area showed formulas as they were typed. We also showed the typing in the cell itself.
  - The date and time at the bottom showed the version of the program -- we didn't have a clock so this wasn't part of the actual program.
  - Each row had a number. In this example it is left justified but we made sure it was right justified for the product. We departed from common notation by numbering the rows and using letters for the columns. This was because we had only 63 columns and 255 rows. These numbers were chosen to limit the number of bits we had to use. There had to be enough columns for a full year of production planning plus some extra. The large number of arrows allowed for it to be used for multiple sets of data such as a payroll table.



```
D13  (V) +D3-B13+C13                        C!
                                            27
         A           B         C          D
 1PAYEE          CHECKS DEPOSITS   BALANCE
 2-------------------------------------------
 3                                  545.20
 4
 5ELECTRIC        14.95
 6OIL            102.15
 7PHONE           36.80
 8DENTIST         42.00
 9SALARY                  395.00
10RENT           350.00
11GAS CARD        12.93
12-------------------------------------------
13TOTALS         558.83   395.00    381.37
14
15
16
17
18
19
20
```

We originally planned to let people rename the rows and columns with labels and implemented the feature. Eventually we decided that we needed to assure stable reference names so prevented people form moving into the zeroth row or column. Instead we implemented the ability to split the screen vertically or horizontally.

Remember that the screen was small so we couldn't fit that much on the screen so a single split was enough.

The columns were all the same width within a window. This avoided a level of indirection in reference the screen. This was a mistake since it would've added insignificant overhead and the lack of what came be called variable column with become a competitive disadvantage.

We made sure that there was a border between numbers in each column to assure readability. If the last column didn't fit then we reformatted for what was visible rather than clipping it as is typical in today's windowing systems. Clipping would let the user see "100" instead of "1000".

We did allow the two windows to have different widths which gave some flexibility. Movement in the two windows could be synchronized or they could be viewed independently. A single cell could be displayed in the two windows.

The windows could also show different global formats including the underlying formulas and a simple character graphics mode that showed the contents of the cell as the corresponding number of asterisks. This was a primitive graphic of plotting capability.

We did toy with the idea of using the split screen bitmap capability of the Apple][to show a live graph at the bottom of the screen but it would've added too much code.

The screen would automatically redisplay as values were changed though the user could turn it off for manual recalculations when automatic calculations were confusing. The “!” would recalculate.

Note that it was always "recalculate" -- the first calculation was just an unimportant special case.

Since displaying the spreadsheet was relatively slow we implemented scrolling as a special case by copying text from one part of the screen to the other. It took a few hundred bytes--a major investment in code, but we felt it was necessary to give a good feel. Thus we were surprised that 1-2-3 didn't do this smooth scrolling. Apparently Jon Sachs ran into some problems and it wasn't worth delaying their shipment for that feature.

Keyboard Usage and Interacting with VisiCalc

Before discussing keyboards, it's worth noting that back in 1979 people viewed the keyboard as an impediment to using computers. After all, only secretaries could type and the rest of us need to be able to talk to the computer. Hence the decades spent on trying to get computers to understand speech. It turns out that most people could type (at least those who used spreadsheets) since it was a basic skill necessary for getting through college. In fact, speech is a very problematic way to interact with a spreadsheet. In fact, the spreadsheet itself is used as a communications vehicle rather than speech.

The Apple][had a simple keyboard that only had upper case letters and only two arrow keys. There were

neither interrupts nor a clock. If the user typed a character before the keyboard input buffer was emptied, it would be lost.

Electric Pencil was an early word processor for the Apple][and it would lose characters if the user typed too fast. To avoid this problem in VisiCalc I polled the keyboard in the middle of potentially long loop--keyboard checks were strewn throughout the code.

The characters would be stored in the input buffer. Since the user would type ahead there was the opposite danger -- overtyping or running ahead. It is normally to press the arrow key until the cursor was in the right place. By the time the user reached the correct cell there would be a lot of extra characters in the input buffer. To prevent skidding we ignored these extra characters. Thus we preserved type ahead but not too much. I doubt if any but the geekiest users were even aware that there was an issue let alone a solution. This is the kind of design detail that makes a program feel good even if you don't know why.

Since there were only two arrow keys we used the space bar to toggle between vertical and horizontal motion and showed the current mode with the -- or ! in the upper right hand corner of the screen. The use of the space bar in conjunction with the arrow keys quickly became internalized to the point that users may not have noticed they were toggling the arrows.

Since I've mentioned the arrow keys I'll get a little ahead to note that the arrow keys worked very differently when entering a formula or label. If you pressed the arrow key when you needed to point to a cell you see the position in line with the formula and as soon as you typed the next character, such as a +-, the cell would be committed and you could continue to edit the formula. But if you were in an operator position and pressed the arrow, it would enter the formula into the cell and move the focus to the new cell. Again, few users were probably aware that these were very different function because the right thing "just happened" at the right place.

This was part of the larger goal of giving the user the illusion of infinite choice and freedom at very point even though only a very small number of choices were allowed. In practice only a few choices made sense in that context. Thus in the context of pointing to a cell, the arrows naturally pointed rather than terminated the formula.

We used the same principle to avoid error messages. One motivation was very simple -- error messages took up a lot of space. Instead we showed the formula as it was interpreted. If what you typed didn't make sense, it just didn't do the wrong thing.

In order to keep this illusion, we had to distinguish between cell names (A1) and functions such as SUM. We used the "@" as a prefix for functions. That seemed acceptable and apparently users didn't have a problem with it.

A1 wasn't really meant to be a name as much as short hand for *that cell*. This is why we didn't use a grid notation like R1C1.

This was also one reason we didn't allow people to give the cells themselves names. The bigger reason was that it wasn't necessary and the most proficient users, those who would most value such a feature, seemed to be very well served without them. But we did consider allowing the use of labels instead of cell names but, given the limits of the Apple][, it never became an issue.

We also need to distinguish formulas from labels and for that we required a formula start with a number or an operator such as +- or the special @. One could use a " to force interpretation as a lab

There were a small number of commands in VisiCalc and we used the / as the "command key". Remember that there were no function keys. The legacy of the / lasted long after VisiCalc and people used to expect / to be the command key on the IBM PC even for word processors. I got complaints when I implemented Lotus express and required a function key for commands. Today the F10 has become standard for that role.

The / itself was chosen because it seemed obvious to me and was otherwise available. But it was also a good choice for Dan whose fingers just happened to be a little crooked and were predisposed to reach that key.

The commands themselves were meant to mnemonic but we only showed the letters since the full names would've been part of the unimplemented help system. The goal was to have an interactive help system that allowed you to see the full names of commands and the keyboard options at any point but we estimated it would have taken 2000 bytes to implemented an interactive help system and that was an unaffordable luxury.

Overall though VisiCalc was designed for the casual user the proficient user was well-rewarded by having an interface which didn't require one to move one's hands off the keyboard or even look at the screen to see where a mouse pointer wound up. The arrows were reliable ways to move one's position (well, as long as one didn't scroll very far).

The Apple][had a reset key and in the first versions there was no way to prevent the user from accidentally pressing reset and losing all the work. This was simply unacceptable for a production product so we including a short command sequence that could be typed into the Apple][monitor to continue VisiCalc. Since we didn't know where VisiCalc interrupted we couldn't assume it was safe to continue and only allowed the user to save the spreadsheet at that point.

Files and I/O

The Apple][handled IO view add-in boards. If you wanted to print to a printer in slot 6 you would say PR#6. if that slot happened to contain a disk drive that same operation, however, would boot from the drive. In order to avoid such problems and do the right thing for each device VisiCalc had a table of signature bytes for each board so that we could avoid doing something like rebooting by mistake. We derived the bytes by examining the boards and, in effect, doing our own plug and play. Thus you could print and VisiCalc would find the printer automatically.

Since we didn't want to be beholden to Apple, I had to reverse engineer the low level I/O operations for the disk drives and implement a compatible file system. The first beta copies had a bug -- I didn't reserve the bitmap for the file system so after a few files the file system would get corrupted. Those people who were careful and wrote their files onto two floppies were not spared -- both would get corrupted at the same time.

Since we were handling the low level I/O operations we could also implement a scheme to discourage copying the floppy. I also added an extra touch by having VisiCalc write over itself after booting on the assumption that a user would test the copy and, unlike the product disk, it wouldn't be protected. Unfortunately, the write protect tab was not reliable on those drives so we would also overwrite the original copy.

The copy protection scheme did make the normal Apple][disk copy program fail. Later this expertise allowed us to ship a single disk that could handle the original floppies with 13 sectors per track and those with 16. It would even remember which way it booted and then format new disks with the same number of sectors.

Eventually the copy protection become too much of an impediment and we dropped it.

As I mentioned we also supported cassette drives in the initial version. When we saved the spreadsheet we made sure the first operation allocated the entire sheet since that could be a long operation and then read it back from the lower right back. This technique also sped up loading from disk.

We saved the spreadsheet in a format that allowed us to use the keyboard input processor to read the file. There were undocumented commands that allowed us to set the initial value of a cell and control the loading. They would also work from the keyboard.

Calculations and Formulas

At its heart, VisiCalc is about numbers. One of the early decisions we made was to use decimal arithmetic so that the errors would be the same one that an accountant would see using a decimal calculator. In retrospect this was a bad decision because people turn out to not care and it made calculations much slower than they would have been in binary.

We did want to have enough precision to handle large numbers for both scientific calculations and in the unrealistic case it would be used to calculate the United States budget. Of course, as it turned out, that was one of the real applications.

Since the formulas did depend on each other the order of (re)calculation made a difference. The first idea was to follow the dependency chains but this would have involved keeping pointers and that would take up memory. We realized that normal spreadsheets were simple and could be calculated in either row or column order and errors would usually become obvious right away. Later spreadsheets touted "natural order" as a major feature but for the Apple][I think we made the right tradeoff.

The functions or, as they were called, the @functions each had a story. Some, like **@sum** seem simple enough but we did have to deal with ranges. **@average** skipped over empty cells and **@count** could be used to find the count of nonempty cells.

For **@npv** (net present value) we decided on a formula which was different from that used in COBOL (a programming language). The COBOL committee was later asked to be compatible with VisiCalc though I don't think they made the change.

One of the early applications for VisiCalc was my 1979 tax form. I created **@lookup** for that purpose.

The transcendental functions like **@sin** were going to be a problem so we decided to omit them in the initial version but, unfortunately, in this review of VisiCalc, Carl Helmers praised that aspect of VisiCalc and we felt obliged to implement them. This was a real pain because I had to find books on such functions and how to compute them for the appropriate precision and range of values and all this had to be done in very little space. It took a week or two but eventually we did them. At this point Dan was available and joined in the programming.

While I could usually cobble together adequate routines to do what was needed I found myself doing a bad job in converting numbers to external representation. Late in coding I found some cases that didn't convert properly and produced results such as "+-0". I patched around it by looking for those cases in the screen buffer itself and fixing it. It worked well enough so that I could move on to other problems.

One design decision was to not do precedence in the formulas. If you typed 3+5*4 you got 32 instead of 23. We reasoned simple calculator users expected each operation to take place as it was typed. In hindsight this was a mistake—people expect precedence and the sequential operators on a simple calculator were not viewed in terms of the whole calculation as written. Internally I had been self-taught on how to do parsing of formulas (I keep trying to not type equation since mathematicians talk about equations since they are normally balancing them). In 1966 I was working on FFL (First Financial Language), a story in its own right but for another essay, and had a vague sense of how to do it but with some advice I figured out how to do a simple stack-based implementation. For VisiCalc I tried to do a very compact version of this and it would have been just as easy to implement precedence.

For those interested in the details of handling formulas ... Internally the parser works by pushing operands and operands on the stack and executing each operation when it was forced by a lower precedence operation. Thus for 3+4*5+2 you push the 3 on the operand stack and the + on the operator stack (at least, that's what I think—unless I actually run the code I presume anything I write is buggy), then push the 4 and then the * and the 5. So we have 3,4,5 and +,*. The + causes the previous * and + to execute, producing 3,20 and +. Next we do the first plus to get 23 on the stack, push the 2 and then the end of the equation causes remaining operations to be performed. This is left to right execution with precedence. All we do to not do precedence is to treat the * and the + as being the same priority. ()'s are used to force ordering.

I used two stacks in this example, if we didn't have to deal with reordering, I could write the formula as 3,4,5,*,+,2,+. This stack notation is known as reverse polish notation or RPN (Invented by Jon Lukasiewicz). HP calculators handled this natively and it was very natural once you got used to it. In fact, many of us liked it better than the standard notation since one didn't have to keep the entire context in mind and there is no need for ()'s. But it wasn't a sufficient improvement to get over the unfamiliarity.

Programming Decisions

This section is for geeks so I won't try to translate all of the terms.

OK, so what's going on behind the curtain? Faced with a 16KB target, that included enough space to actually hold a useful spreadsheet, I went into severe design mode. I normally don't worry about the size of my code since it takes a lot of work and normally doesn't make a difference and there is a real risk of getting locked into premature design decisions.

For VisiCalc I had no choice. It was made more difficult by not knowing much about the program since no one had used it yet. Dan's ability to work on the prototype gave us some clues about where we were headed. I started to mock up the program by writing the initializations code, SSINIT (SpreadSheet Init) so that we had a framework for displaying the sheet. Now all I had to do was fill in the stuff underneath.

One guiding principle was to always have functioning code. It was the scaffolding and all I needed to do was flesh it out. Or not. Since the program held together omitting a feature was a choice and it gave us flexibility.

I was lucky in that basic architecture was viable. Well, after programming for 15 years I did have some idea of how to write such a program so it was more than luck. In fact, I did have to do some reworking as we went along but I also left stubs such as the reality of row and column 0 for the labels even though we didn't allow users to move there. It allowed them to be handled normally by most of the code.

One of the earliest issues was representation -- how do I represent the formulas in memory (and later, on disk). I was still spending a little time at Interactive Data at that point and designing the layout was the kind of productive doodling I needed to stay awake in a training class.

The basic approach was to allocate memory into fixed chunks so that we wouldn't have a problem with the kind of breakage that occurs with irregular allocation. Deallocating a cell freed up 100% of its storage. Thus a given spreadsheet would take up the same amount of space no matter how it was created. I presumed that the spreadsheet would normally be compact and in the upper left (low number rows and cells) so used a vector of rows vectors. The chunks were also called cells so I had to be careful about terminology to avoid confusion. Internally the term "cell" always meant storage cell. These cells were allocated from one direction and the vectors from the other. When they collided the program reorganized the storage. It had to do this in place since there was no room left at that point -- after all that's why we had to do the reorganization.

The actual representation was variable length with each element prefixed by a varying length type indicator. In order to avoid having most code parse the formula the last by was marked $ff (or 0xff in today's representation). It turned out that valid cell references at the edges of the sheet looked like this and created some interesting bugs.

The program was tuned for the Apple]['s 6502 processor. It processes 8 bits at a time and has up to 64KB bytes. The program was tuned to this processor

- Arrays of 16 bit values were split into two 8 bit arrays so that the value could be incremented or decremented in a single operation in a loop.
- Loops tended to go from the high to low value since this saved a byte in each loop

The assembler had macros so that instead of directly coding to the machines conditional instructions I could use an "aif/aelse/aendif" set in order to assure that the structure of the code was maintained. There was a special "calret" (call/return) operator that was used for calling a subroutine at the end of a sequence of code. It generated an efficient jump instruction but the reader could assume that the code continued and returned at that point instead of wondering about an unstructured transfer.

Though there was a very strong emphasis on efficiency there was even a strong emphasis on readability. Anything that might surprise someone reading the code was carefully documented. My assumption was that I would forget those key points myself. It also helped others who would read the code but the primary audience was me in the fog of coding.

The spreadsheet array itself was designed for efficient processing.

- The spreadsheet itself had a guard row so that there was no need to constantly check against the bounds but sometimes an operation, such as replicate would skip over this guard and would simply wrap back. It might have produced strange results but no damage.
- The insert operation was really a move and it fail if the last row was occupied. The bias was towards assuming that everything was in the upper left.
- Though I knew how to constructed shared structures the formulas were copied into each cell. Attempting to share the representation would have added complexity and we would have had to hide this from the user in order to make irregularities in the spreadsheet normal. Programs that emphasized regularity failed because they did time series well and everything else poorly.

Memory and Patching

Memory was clearly limited hence the care to save every byte of code. The original goal was to fit the program and the operating system and the screen into 16KB but the code grew to 20KB so we required a 32KB machine but by the time we shipped 48KB was common. Shortly before we shipped Apple sent us their expanded memory card. It was created in order to be able to run UCSD Pascal. I quickly hacked the code to support the extra memory but only used 12KB in order to avoid having to remap addresses in order to use the remaining memory on the board. I didn't want to risk introducing subtle errors at the last minute.

Later other companies introduced memory boards and patched the code to reference the additional memory. I found it amazing that people went through so much trouble to figure out the code and make changes. I was so concerned about efficiency that I over-engineered the program for performance. It turned out that there was plenty of slack that allowed one to add extra code for extended memory but even more surprising was that people figured it out without listings. The code itself was very readable, at least if you had the source and realized that all those jumps were "call/return" sequences rather than random changes in the path of execution. Because of legal issues I wasn't able to keep a copy of such disassemblies but I'm very interested in compare them side by side. When I get a chance I want to scan in a listing for those curious.

There were also companies that extended the Apple][with video cards that could display 80 columns. Here too they created their own patches. I'm curious about how the techniques they used and how reliable these changes were. An interesting question which I didn't think of was whether you could combine patches if you had a memory board with an extended video card. Probably not.

We could've made VisiCalc more adaptable but it was difficult to make it a priority and the strained relationships with our publisher (Personal Software which later renamed itself VisiCorp after its premier product) made it difficult for us to work directly with such third parties.

Tools and Environment

We started programming by using the tools from ECD on Multics. I worked at night when the computer time cost $1/hour. Honeywell also took advantage of the low fee to use the machine at night to develop the Ada language for the military but those developers worked during their day from France.

Once we formed the company we decided to buy our own computer. Prime was trying to follow in Multics' path and seemed like a reasonable choice. It had a PL/1 compiler which made it easy for me to port some of my work. The first project was to implement a simple editor and then an assembler and other development tools. On the side I evolved an editor that was originally supposed to be a line editor (QED) into a screen editor (Emacs). Seth Steinberg who had worked at MIT Architecture Machine Group added a lisp-like language and Emacs became a very useful tool. We even programmed an email system within the editor. It also allowed us to create tools to assist in formatting the code and other housekeeping.

Later we developed our own language and tools to make it easier to code and to write for multiple platforms but that's a separate topics. For now I'm focusing on the early days of VisiCalc. Once we had grown we used more advanced tools such as an in-circuit emulator which allowed us to examine code as it executed. It proved itself invaluable when I found that the reason my code was failing was that the memory ship was defective and the values changed on their own! Sometimes it is the hardware!

Later Enhancements Versions

The Apple][version was the key version of VisiCalc. Before we shipped we added the ability to run demonstration programs and eventually evolved this into a macro capability for advanced VisiCalc.

Before we shipped we started to grow the company and moved from my attic to share office with John Strayhorn's Renaissance Computing and hired Steve Lawrence who I had worked with at ECD.

After the Apple][we created versions for the Commodore Pet and the Atari 800 since both use the same processor. Brad Templeton (currently chairmen of the EFF) helped us with the port to the Commodore Pet.

We hired Seth Steinberg to convert the code to the Z80 and he did a very faithful port. He was very skilled and recognized the goal was to keep the code base aligned rather than trying to show off his own skills.

And then, well, that's another story.

On October 17th we finally shipped the prototype, now dubbed the production version, of VisiCalc.

Notes

John Doty's comments

John Doty sent a note about this piece and cc'ed me.

Interesting. Bob remembers me as a "programmer", but my real job was production troubleshooter. I was a grad student, and I was broke, so I had called my friend Jerry Roberts about a job at his company, ECD. He hired me to work on production of their 6502-based MicroMind.

One of the problems I identified was the lack of a good preship diagnostic, so I wrote one. It was the kind of assembly code I hate: lots of loops and conditionals. Keeping the branch targets straight was giving me a headache. I asked one of the programmers, Dan Franklin, to add stacks to the macro expander. I then used them to implement if, do, repeat, continue, etc. This made my job a lot easier.

Bob implies that I wrote the assembler itself. I did not: it already existed when I arrived. I don't remember who wrote it (Spencer Love perhaps?).

The programmers quickly adopted my macros. Bob and most of the others were MIT EECS students and graduates, steeped in the PL/I mainframe culture of that department at that time. Assembly language on a micro was culture shock: I suppose anything that made the programming process more familiar was welcome.

Of course, I had learned assembly language on the 1130. But that's another story...

I should note that I had actually done a lot of assembly language programming over the years but always try to use the most effective tool I could find. Even when programming in assembler on the IBM 360 I tried to keep my code structured because I knew that if I wasn't very careful to keep my code very simple I wouldn't be able to understand it myself. No wonder people were surprised to find out I could write very compact code since I only did it when there was a real need. Sometimes it's even necessary to build a piece of hardware to accomplish the task.

Later in the email exchange John added

I should also note that there's another spreadsheet connection in this story. The diagnostic I wrote for ECD incorporated some ideas I'd gotten from Jon Sachs a couple of years earlier when we were struggling with a flaky Nova computer.

This connection with Jonathon Sachs is just one of a number of ways we crossed paths one-off. I hope someone can take the initiative to trace down the paths of people and their ideas -- you'll find a very dense fabric of interconnections. This was especially true when there was a small academic community doing research but is still true today within the various computing communities.