The Child-Engineering of Arithmetic in ToonTalk

Ken Kahn
Animated Programs and the Institute of Education, University of London
3 Green View, The Green
Theydon Bois, Essex, CM16 7JD, England, UK
kenkahn@toontalk.com

A short version of this paper to appear in the *Proceedings of the Interaction Design and Children Conference*, June 2004

ABSTRACT
Providing a child-appropriate interface to an arithmetic package with arbitrarily large numbers and exact fractions is surprisingly challenging. We discuss solutions to problems ranging from how to present fractions such as 1/3 to how to deal with numbers with tens of thousands of digits to how to deal with irrational numbers. As with other objects in ToonTalk®, we strive to make the enhanced numbers work in a concrete and playful manner.

Keywords
Exact arithmetic, programming languages for children, ToonTalk

INTRODUCTION TO THE PROBLEM

In the programming language ToonTalk [1,2] children enter an animated world full of objects and tools that the children pick up and use in a game-like manner. These objects range from robots that can be trained, to birds that can deliver messages, to boxes for storing things. This paper focuses exclusively on number pads in ToonTalk.

Until recently, arithmetic in ToonTalk was limited to integers that are internally represented with 32 bits or less. This severely limited the extent to which child could explore numbers within ToonTalk. As a programming language where children primarily create games the limitations of numbers rarely arose. The built-in coordinate system of ToonTalk considers the screen as a 1000x1000 surface so there was no need for fractions when programming games or animations. If the result of a calculation was a number greater than 2^{31} (approximately two billion) then an animated character appeared and explained that the number would be "too large for the computer" (a half truth).

But what about the child who wants to explore mathematics? If a child divided 1 by 2 he or she got 0 as a result (since all results were coerced to integers). If this child trained a robot to repeatedly double a number then why should this robot stop when doubling 2,000,000,000?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IDC 2004, June 1-3, 2004, College Park, Maryland, USA

© 2004 ACM 1-58113-791-5/04/0006…$5.00.

In September 2002, the large research project WebLabs [3] was funded by the European Commission. WebLabs project members in six countries are providing learning materials and building components in ToonTalk to enable children to build scientific models and explore mathematics. Children participating in the project explore topics ranging from convergence and divergence of infinite sequences to Newtonian mechanics to cardinality of infinite sets. None of these activities would have been feasible without enhancing ToonTalk's arithmetic.

ARE FLOATING POINT NUMBERS A SOLUTION?

The simplest way to provide support for fractions and very large numbers is to introduce floating point numbers to ToonTalk. This is the approach taken by the vast majority of programming languages. If the 64-bit IEEE standard floating point numbers are used then the numbers could range between $\sim 10^{-323.3}$ and $\sim 10^{308.3}$ with 52 bits of accuracy. Surely this is enough?

Approximate but high accuracy numbers are fine for scientific uses but what about mathematical exploration? Floating point numbers are poorly suited for this. Fundamental arithmetical properties such as the associativity and commutativity of multiplication are no longer guaranteed due to the possibility of round off errors. For example, one of the WebLabs activities is to explore the convergence of the series 1/2+1/4+1/8+… With floating point numbers, the sequence of partial sums prematurely converges to 1 after 55 iterations. Another example is the exploration of very large numbers. With floating point numbers you can't empirically verify answers to questions like how many zeros are at the end of the factorial of 1000.

BIG NUMBERS AND EXACT RATIONAL NUMBERS

The alternative to floating point numbers that we chose is to use a multiple precision exact arithmetic package [4]. With such a package the size of integers is only constrained by the available memory in the system. Computing with integers with millions of digits is feasible. Fractions are represented as two integers: the numerator and the denominator. There is no round off error when adding, subtracting, multiplying, or dividing. Approximations only arise if roots or trigonometric functions are used since they produce irrational numbers. In these cases, ToonTalk falls back upon floating point numbers.

INTERFACE CHALLENGES WITH EXACT ARITHMETIC

The biggest challenge in enhancing ToonTalk's numbers was how to display them. Consider the result of dividing 3 by 2. Should it be 1 1/2, 3/2, or 1.5? And how should the result of dividing 1 by 3 be portrayed? And how should we deal with very large numbers such as 10000! (factorial of 10,000) which has over 35,000 digits?

The Challenge of Very Large Numbers

ToonTalk numbers are depicted as light green pads that can be picked up and manipulated by either the hand of the programmer's avatar or by robots. If a number is too large to see all at once the virtual camera clips it.

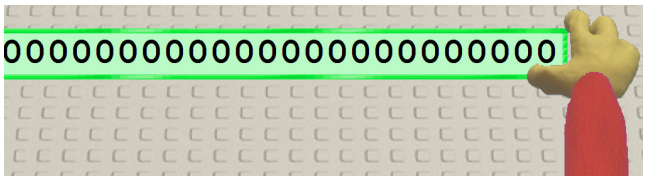

Figure 1 – Screen shot holding the end of 10000!

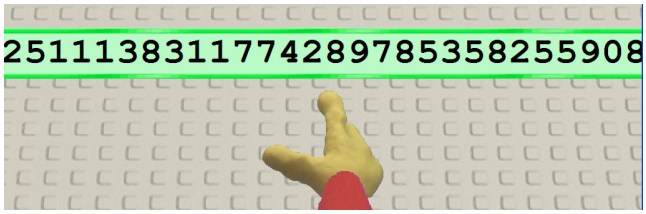

Figure 2 – Screen shot pointing to the middle of 10000!

Large numbers can be viewed by moving the camera (it follows your hand) but for very large numbers you don't get a sense of how big it really is.

Rather than view the number on the floor one can take it outside and view it while standing or flying a helicopter.

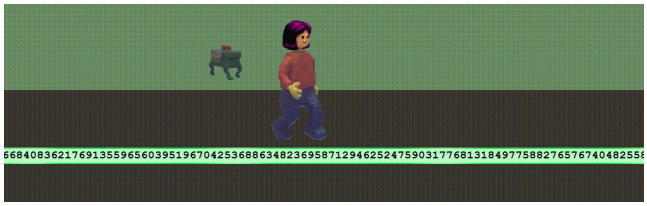

Figure 3 – Walking the length of 10000!

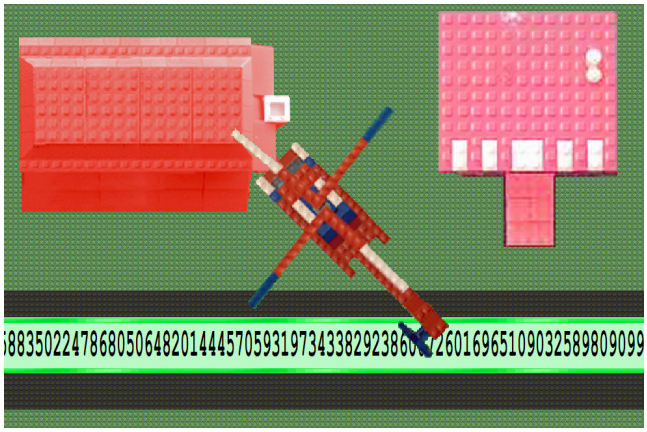

Figure 4 – Flying low over 10000!

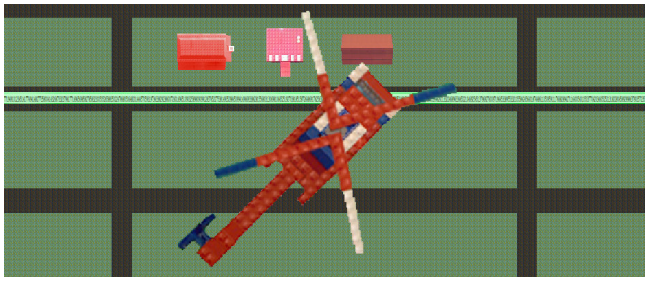

Figure 5 – Flying high over 10000!

When a number pad is contained in something else then ToonTalk isn't free to clip the number to just the portion visible. And for large numbers the font size would be much smaller than a pixel in order to display the whole number. Only in this context is only a portion of the number displayed followed by "…". A smaller font is used as well to indicate the ellipsis.

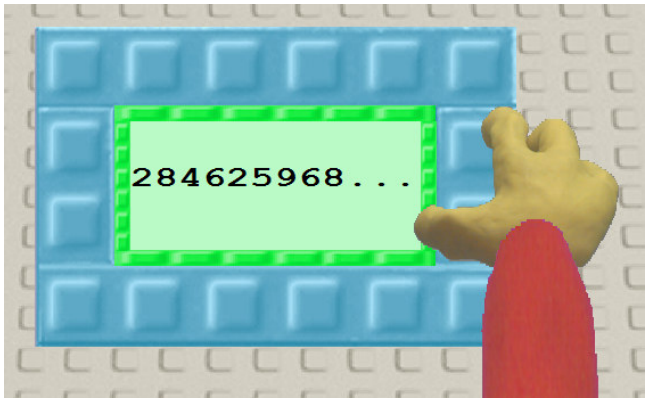

Figure 6 –10000! in a box

The Challenge of Displaying Fractions

Unfortunately there is no single dominant format for displaying the result of dividing 3 by 2. The three alternatives are depicted below.

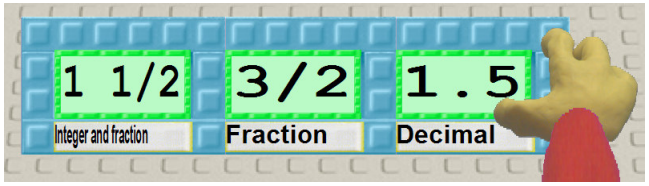

Figure 7 – Three presentations of 3 divided by 2

The *Decimal* format is well-suited for visual comparison (consider whether 1.4 is less than 1.5 vs. whether 7/5 is less than 3/2) but not every rational number can be represented as a finite decimal expansion. The *Integer and proper Fraction* format is general and is easier to visually compare than the *Fraction* format (compare 1 2/5 with 1 1/2). But it is more complex than the general *Fraction* format.

The first solution to this problem was to add all these formats to ToonTalk. ToonTalk arithmetic is expressed by dropping number pads (possibly with an arithmetic operator) on top of other number pads. When a number or a mathematical operation is dropped on a number they are

combined by a small mouse with large red hammer. This follows the general ToonTalk rule that when two things are combined, it is the one underneath that determines the type (or format in this case) of the result.

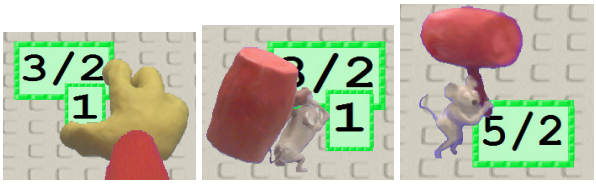

Figure 8 – Three screen shots of 1 added to 3/2

The *Decimal* format has the additional problem of how to display repeating decimals such as 1/3 as .33333… Truncation is not a viable solution since it could introduce two numbers that look the same but are really different (in the decimal places not displayed). This would break a general principal in ToonTalk that two things that look the same should be treated the same by the matching mechanism that robots use. Ellipsis is a poor solution since it introduces ambiguity. Is .03… 1/30 (i.e. .03333…) or 1/33 (i.e. .030303…)? The vinculum (the bar over the repeating digits) avoids this ambiguity and is the standard solution to this. One problem with $.\overline{03}$ is that it is not easy to cut and paste with other applications since there is no standard character encoding for it. (Some countries such as Portugal use parentheses to indicate the repeating portion. There are no encoding problems with .(03).) Another problem is that the repeating part can be very long. 1/7 has 6 repeating digits. The reciprocal of an integer near a billion may need a billion digits to display the entire repeating portion.

Consequently, the *Decimal* format was split into two formats both of which use decimals when the expansion is finite and otherwise use either the *Integer and proper Fraction* format or the *Fraction* format.

Initial user testing with enhancement showed it was an adequate solution but somewhat complex and confusing. Some users confused the different presentation styles for the same number with the notion of different types of numbers. Most users, however, just used the default format number that was in the user's toolbox.

We greatly improved the situation by inventing a new number format that we call the *Shrinking Digits* format. The idea is to display the fractional part as a decimal that uses a decreasingly smaller font size. Visually this indicates that there is more to the number – you just can't see it since it is too small.

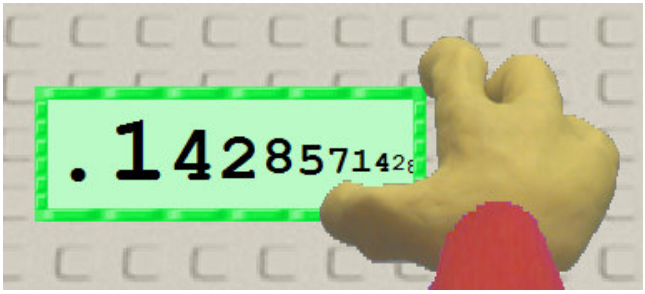

Figure 9 – 1/7 in Shrinking Digit format

Number pads, like everything else in ToonTalk, can have their size changed directly by using a bicycle pump or by putting the number in a box. When more room is available more digits can be seen. Simply by pumping up the number pad, one can see more and more digits. Frequently the repeating segment of digits is easy to see. Note that this format does not suffer from the problem that two numbers look the same in practice since it is easy to expand the numbers to see how they differ.

Figure 10 – Expanding 1/7 with the bicycle pump

The *Shrinking Digits* format has been the default number format in the recent beta versions of ToonTalk and it has been very successful.

Numbers in other bases

ToonTalk numbers can be displayed in any base between 2 and 36. The base is 10 unless changed explicitly by typing '#' which separates the base and the number. When two numbers are combined, the result has the same base as the number underneath. Fractions with the *Shrinking Digits* format are currently only supported in base 10.

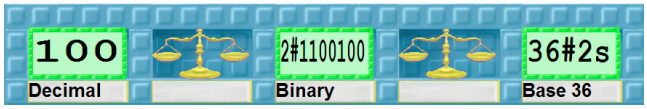

Figure 11 – 100 in 3 bases (scales indicate equality)

There is pedagogic value in seeing multiple presentations (i.e. syntaxes) for the same semantic object (the value of the number). This is true both for fractions and for numbers in different bases. Each format has its own strengths and weaknesses.

A rather extreme use of bases has been explored by one of the WebLabs partners. They have experimented with using base 36 for encryption. A text message (without spaces or punctuation) is a base-36 number that can be encrypted by applying a mathematical operation (e.g., multiplying by n) and decrypted by applying the inverse operation (dividing by n).

Performance issues

Today's computers execute most mathematical operations in nanoseconds. But those speeds only apply to 32-bit or 64-bit numbers. A child who trains a robot to repeatedly double a number can quickly generate numbers with millions of bits. Operations on such big numbers can take many seconds. Frequent operations like this can make ToonTalk very jerky and hard to use. And this problem can arise in subtle ways due to the nature of exact fractions. If a child is exploring the convergence of a series such as 1/4+1/16+1/64+… and lets their computation run for thousands of iterations they'll have a number that is nearly 1/3 but internally is the quotient of two very large numbers.

The best solution to this problem that we've explored is to automatically offer trouble shooting advice the first time a user creates a very large number (currently defined as more than 5000 digits). The advice web page explains why ToonTalk is slowing down and becoming jerky. It even advises that if the user really wants to explore extremely large numbers then they'll speed things up significantly if they change to a base that is a power of 2 (e.g. octal or hexadecimal). This is because converting a million bit number to decimal is much more expensive than adding or multiplying such numbers.

Ironically the fact that computations slow down as the size of the numbers grows alleviates another problem with exact arithmetic: memory usage. 10 to the 10 to the 10^{th}, for example, requires 10 billion bytes and yet can be expressed with a few keystrokes. Typically computations slow down more and more rather than use up too much memory.

Another hidden cost of exact arithmetic is the size of the files needed to save ToonTalk programs. If very big numbers are involved these files can become quite large consequently taking a long time to create and load.

Irrational numbers

If a child uses only addition, subtraction, multiplication, and division then exact rational arithmetic can handle every computation. But if they use roots, trigonometric operations, or logarithms then the results can be irrational. The square root of 2, for example, can not be expressed as a ratio of two integers, as Euclid discovered 2300 years ago.

A very ambitious solution to this problem is to represent these values symbolically (perhaps as Taylor expansions) and compute additional digits as needed. (This is essentially the same as Turing's notion of computable numbers [5].) This would fit well with the *Shrinking Digits* format. This would be a huge implementation effort so instead ToonTalk falls back on floating point numbers as approximations. To indicate that the result is an approximation the digits are displayed in gray rather than black. Because these approximations have all the problems discussed above they are avoided when feasible. For example, the square root of 9/4 is exactly 3/2, while the square root of 3/2 is approximately 1.224744871391589. Fortunately irrational numbers only result from applying "advanced" operations and less mathematically advanced children will typically not encounter them.

Java and big decimals

When a child wants to publish something they have built in ToonTalk for those without ToonTalk, they can click on a button that converts what they are holding into a Java applet. Their program can then be run by practically any web browser. The version of Java that runs in all browsers does not support exact arithmetic. It does, however, support big integers and "big decimals". A big decimal is approximation but, unlike floating point numbers, the degree of accuracy can be set to a relatively high number. The Java applets can, for example, run accurately with 1000 digits after the decimal point.

PRELIMINARY USAGE RESULTS

While the exact arithmetic enhancements to ToonTalk are in beta at the time of writing this (January 2004), they have been tested by hundreds of children. We have received no reports of any confusion resulting from the usage of the *Shrinking Digits* format. We have seen many children enthusiastically explore very large numbers by placing them outside and walking and flying over them to get a sense of how large they are.

FURTHER ENHANCEMENTS

Several improvements are possible. While exploring large numbers it is great that children can get a sense of how large these numbers are by walking and flying over them, there are times when a scientific notation might be better. E.g. a googol in scientific notation would be displayed as 10^{100} or 10e100 rather than 1 followed by 100 zeroes. This format should not introduce approximations. So its utility is limited to integers that end with a large number of zeros and to some floating point numbers that are already approximations.

Trigonometric and logarithm functions sometimes have values that are rational numbers. In these cases, ToonTalk could avoid floating point approximations. Also these operations currently signal an error when applied to numbers that are outside of the range of floating point numbers (between $\sim 10^{-323.3}$ and $\sim 10^{308.3}$).

The *Shrinking Digits* format is currently limited to decimal expansions. Other bases might be useful.

RELATED WORK

Many programming languages for professionals such as Java, Python, Scheme, and Lisp support large integers and exact fractions. However, these languages leave the presentation of such numbers under programmer control. This is very flexible but is too complex for a system for children.

Squeak is a programming system used by both professionals and children [6]. It has support for large integers and exact fractions. Etoys, the part of Squeak used by children,

simply prints a limited number of decimal digits and does nothing special with large integers.

CONCLUSIONS

We want to empower children to explore arithmetic without the limits imposed by 32 or 64 bit integers and floating point numbers. New interface challenges arise when numbers are very large or have no finite decimal expansion. In the context of ToonTalk, we have provided new ways for children to hold in their hand and manipulate numbers such as 1/3, a googol, or the 1000^{th} Fibonacci number.

ACKNOWLEDGMENTS

The entire ToonTalk and WebLabs communities have been very helpful in providing feedback and testing of these enhancements to ToonTalk's numbers. I am grateful to Gordon Simpson, Leonel Morgado, and Mary Dalrymple for their comments on this paper.

REFERENCES

1. Kahn K., "ToonTalk - An Animated Programming Environment for Children", *Journal of Visual Languages and Computing*, June 1996.
2. ToonTalk Web Site, www.toontalk.com
3. WebLabs Web Site, www.weblabs.eu.com
4. GNU Multiple Precision Library, www.gnu.org/software/gmp/gmp.html
5. Turing, A., On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, Series 2, 42 (1936), pp 230-265.
6. Squeak Web Site, www.squeak.org