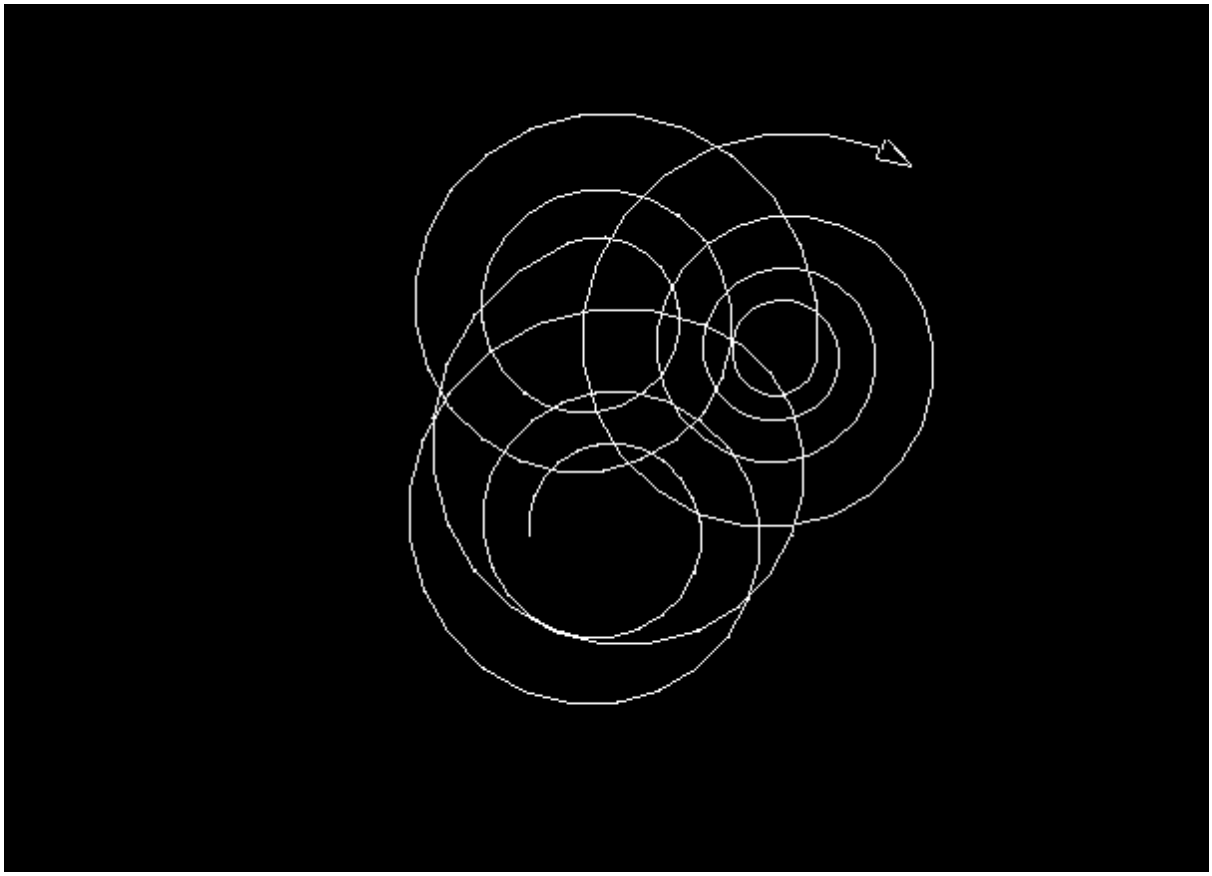
El zoo de computación

Bernat Romagosa i Carrasquer

Para algunos de nosotros, la aventura de programar comenzó de la mano de un personaje experto en geometría, testarudo pero infalible, que nos pedía un poco de imaginación a la hora de ver, en su forma triangular, una tortuga casi mágica que esperaba nuestras órdenes. Esta tortuga, protagonista del lenguaje de programación educativo LOGO, quería enseñarnos que la computación no era una disciplina relegada solo a unos pocos expertos, sino una herramienta con la que hasta los más pequeños podríamos aprender a construir formas y mundos virtuales sin la ayuda de un adulto y, además, sin miedo a equivocarnos.



Diversas espirales encadenadas en LOGO

Para entender lo revolucionario que fue este lenguaje hay que darse cuenta de que, cuando se lanzó su primera versión en 1967, solo algunas de las universidades más punteras contaban con ordenadores para la educación, y éstos eran colosales artilugios rellenos de cables y transistores. Hoy en día cuesta muy poco convencer a alguien de la importancia de enseñar a programar a todo el mundo, pero a mediados de los 60 no era fácil imaginar un mundo en el que hasta la persona más tecnófoba poseería varias computadoras, desde teléfonos móviles a ordenadores portátiles, pasando por la gran multitud de dispositivos programables con que interactuamos a diario.

LOGO fue tan pionero que tendrían que pasar veinte años desde su nacimiento -en el mejor de los casos- hasta que nuestros colegios empezaran a contar con aulas de informática. Por

cierto, para Papert, contar con un aula de ordenadores era un planteamiento tan absurdo como lo sería contar con un aula de lápices, y resulta curioso ver cómo hoy en día estas aulas están comenzando a desaparecer con la ubicuidad de los dispositivos portátiles, aunque éstos solamente se parezcan en forma a las máquinas completamente programables con que Papert soñó.

Para Papert, las computadoras debían acercarse más a los humanos, y no al revés. Es por este motivo que LOGO se diseñó ya desde el principio para ser dinámico, brindando resultados inmediatos a su programador sin necesidad de pasar por procesos intermedios. Cuando el usuario inicia el intérprete de LOGO, se encuentra con una tortuga que espera sus órdenes, y todo lo que se escribe en la línea de comandos tiene un efecto inmediato en el comportamiento del pequeño reptil triangular. Se programa en tiempo real, de la misma forma en que se habla y se escucha en tiempo real, sin necesidad de aplicar transformaciones a aquello que uno dice para que otro lo entienda.

Este dinamismo fue heredado de Lisp, un lenguaje dinámico nacido en 1958 que, con una especificación que cabe en media hoja de papel, consigue una versatilidad y una potencia computacional de la que muy pocos lenguajes modernos pueden presumir. Lisp fue la cuna de muchas de las grandes ideas de la ingeniería informática, como el tipado dinámico o las funciones de primer orden.

En cualquier caso, LOGO no sería el último de los lenguajes educativos, aunque sí sería el de referencia y el más utilizado mundialmente durante muchos años. Hasta hoy se ha implementado una incontable cantidad de nuevos lenguajes para la enseñanza de la programación, pero el caso que nos ocupa en este artículo es el del camino que LOGO abrió, que se ha ido ramificando a lo largo de todos estos años y ha dado frutos tan diversos que parece mentira que todos ellos partan de una misma raíz.

Una de estas primeras ramificaciones empezó a gestarse cuando un joven llamado Alan Kay visitó el laboratorio de Seymour Papert en el MIT. Kay vio en LOGO no solamente un lenguaje de programación, sino también una forma completamente distinta de entender la computación, en que los ordenadores estarían al servicio de todo el mundo, y serían fácilmente programables por cualquier persona, sin necesidad de una formación avanzada.

En Xerox PARC, un centro de investigación situado en Palo Alto, el equipo de Alan Kay partió de las revolucionarias ideas de Lisp, LOGO y otras maravillas tecnológicas de la época (como Simula, Sketchpad o los múltiples inventos futuristas de Douglas Engelbart) para emprender un camino del que todavía hoy en día no hemos recogido todos los frutos. Este equipo daría a luz al primer lenguaje orientado a objetos de la historia, Smalltalk, inventando además las interfaces gráficas a que estamos tan acostumbrados actualmente (menús, barras de desplazamiento, iconos, ventanas, etc).

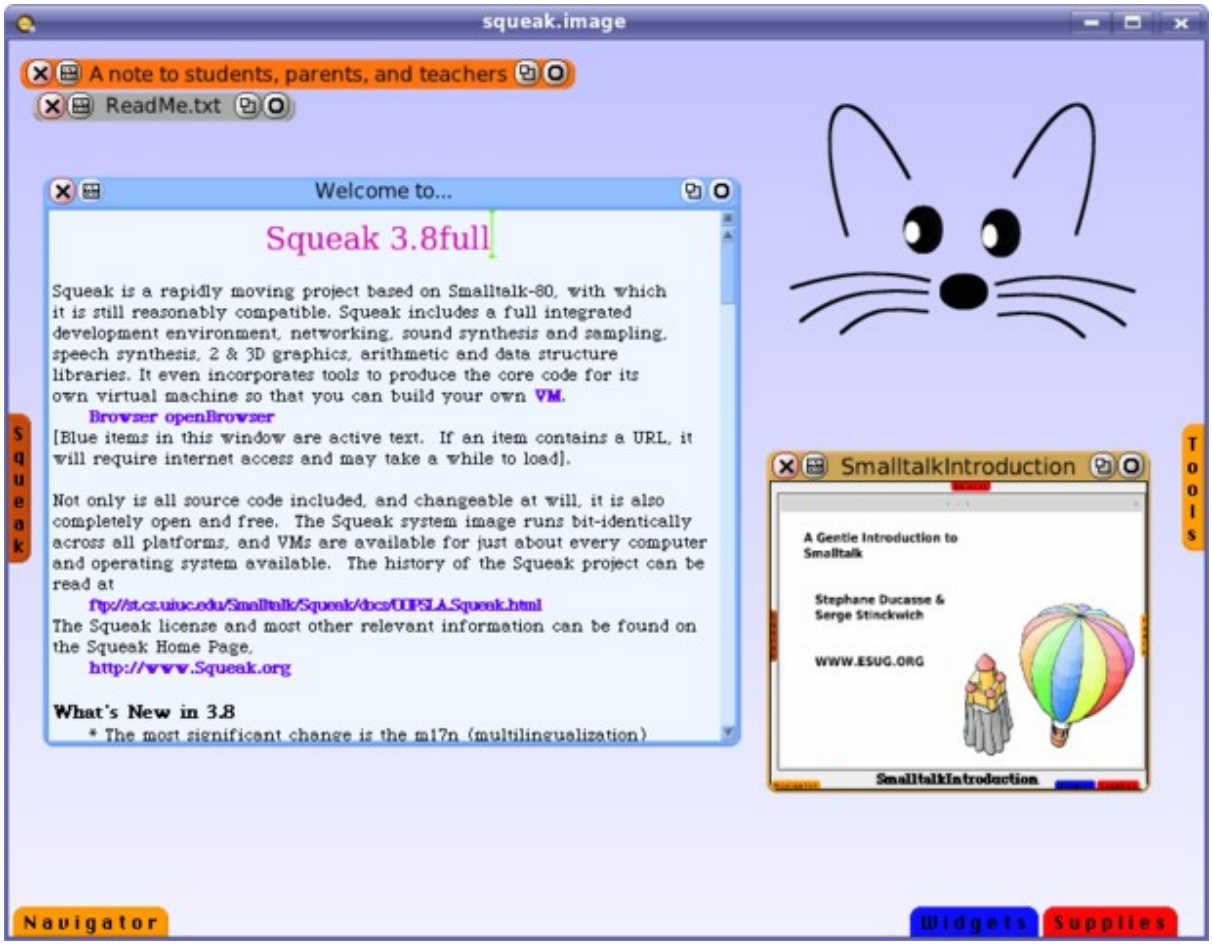
Alan Kay, mostrando su maqueta del DynaBook
[https://commons.wikimedia.org/wiki/File:Alan_Kay_and_the_prototype_of_Dynabook,_pt._5_(3010032738).jpg]

Las hazañas de PARC son tantas y tan interesantes que sorprende que la mayoría de los profesionales de la informática ni siquiera haya oído hablar de ellas. Probablemente, el equipo de Palo Alto estaba tan ocupado inventando el futuro que no tuvo tiempo de divulgar y recoger crédito por sus descubrimientos. En este laboratorio se desarrollaron avances tecnológicos tan importantes como los gráficos de mapa de bits, los procesadores de texto, el ethernet, o la impresión láser, invento con que Xerox revolucionó totalmente el mundo de la copistería. Alan Kay incluso llegó a concebir y esbozar el diseño de una pequeña computadora de bolsillo llamada DynaBook parecida a las tabletas actuales, aunque solo en su tamaño, y no en su programabilidad.

El Alto, un ordenador personal desarrollado en PARC, era una máquina totalmente programable en tiempo real. De la misma forma en que la tortuga de LOGO esperaba nuestras órdenes para dibujar formas en la pantalla, el Alto esperaba nuestras órdenes para modificar el comportamiento del ordenador dinámicamente. Lamentablemente, todos estos conceptos eran demasiado avanzados para su tiempo, y la indústria solamente entendió que los ordenadores tenían que ser capaces de mostrar iconos y hacer nuestros puestos de trabajo más productivos y versátiles con procesadores de texto e imagen pre-programados e imposibles de modificar.

Smalltalk, el lenguaje y entorno de programación que daba toda esta potencia computacional al Alto, siguió su camino y se mantuvo razonablemente popular hasta los años 80, cuando fue rápidamente desplazado por otros lenguajes con más recursos económicos y mayor capacidad de marketing.

Tuvieron que pasar muchos años hasta que Alan Kay, Dan Ingalls y Adele Goldberg, miembros del equipo original de Smalltalk, diseñaran en 1996 una versión modernizada de este lenguaje, capaz de generar música, programar mundos en tres dimensiones, procesar imágenes o ser el motor de páginas web dinámicas e interactivas. Al igual que su predecesor, este nuevo Smalltalk, bautizado con el nombre de Squeak y con un ratón como logotipo, nació con la intención de acercar la programación a todo el mundo, y hasta contaba con un sub-lenguaje de scripting para los más pequeños que heredaba muchas de sus ideas de LOGO.



El entorno de Squeak Smalltalk 3.8 [https://en.wikipedia.org/wiki/File:Squeak-x11.png]

Paralelamente, mientras Squeak todavía estaba en gestación, un estudiante de doctorado del MIT llamado Mitchel Resnik comenzó a trabajar en su tesis bajo la tutoría de dos gigantes de la computación. El primero de ellos, Hal Abelson, fue director de la primera implementación de LOGO en el Apple II, pero es mucho más conocido por ser el co-autor de *Structure and Interpretation of Computer Programs*, uno de los libros más influyentes del mundo de la programación. El segundo tutor de Resnik fue, precisamente, Seymour Papert.

Bajo la tutoría de estas dos leyendas, Resnik desarrolló una tesis brillante que culminó, en 2003, con el nacimiento del lenguaje de programación educativo Scratch. No es exagerado afirmar que Scratch marcó un punto de inflexión en la historia de la enseñanza de la computación, convirtiéndose en el nuevo LOGO de una época en que los ordenadores ya estaban en todos los hogares y escuelas, y para una sociedad que estaba ya preparada para aprender a programar.

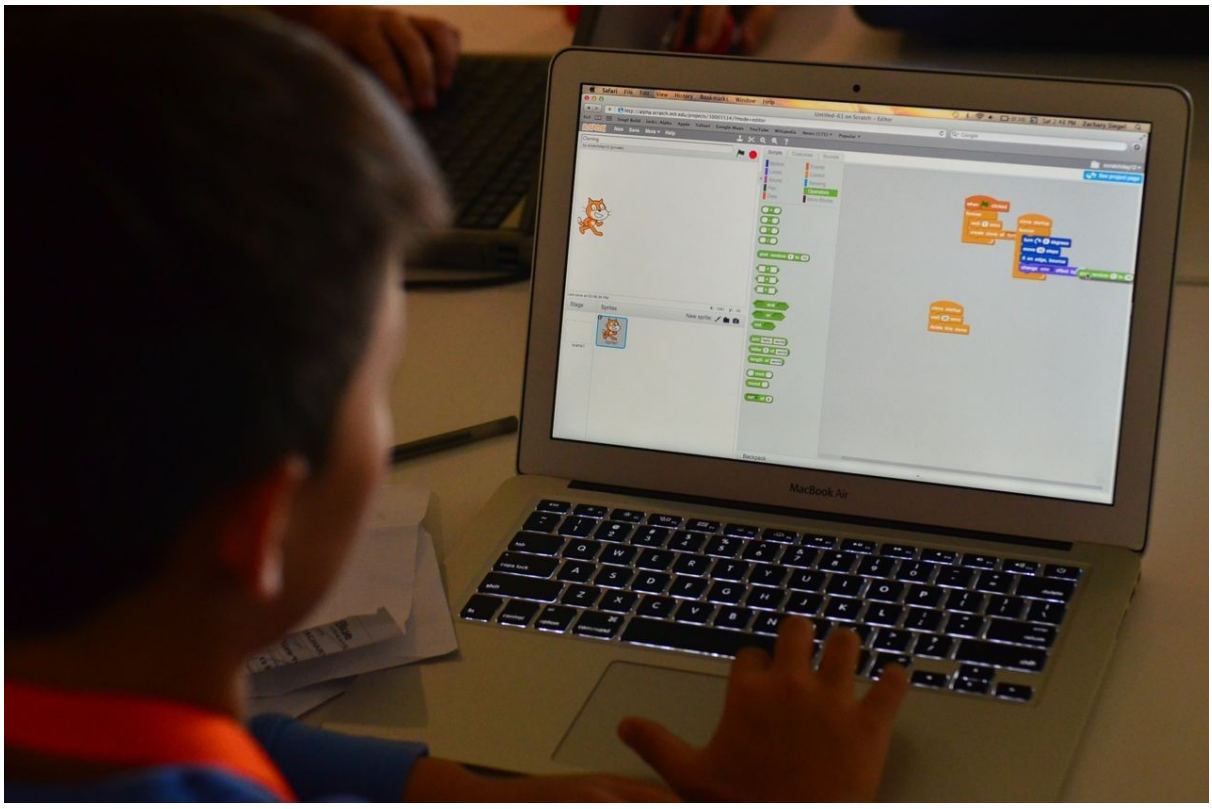
Incidentalmente, y no por casualidad, Scratch se implementó en Squeak, ese nuevo Smalltalk moderno que nació justo en el momento en que se le necesitaba, cerrando así el círculo y haciéndole un guiño a esa primera visita en que Kay conoció a Papert.

En Scratch, Resnik solventó algunas de las barreras de entrada más problemáticas de todo el resto de lenguajes de programación, educativos o no. En cualquier otro lenguaje hay que aprender una sintaxis, hay que memorizar un conjunto de instrucciones y hay que recordar algunas convenciones de código y estilo. En la mayoría de los lenguajes, además, habrá retahílas de código que, sin tener demasiado claro para qué sirven, siempre tendremos que acordarnos de escribir cuando empecemos a escribir un programa, definamos una función, declaremos unas variables o construyamos un nuevo objeto.

En definitiva, en todos los lenguajes existen componentes textuales que tenemos que aprender y recordar, y no podemos equivocarnos siquiera en un solo carácter ni obviar un solo signo de puntuación. Esto no parece haber molestado jamás a los programadores profesionales, pero es evidente que éste es un bache que frustra y desmotiva a la mayoría de los principiantes.

Scratch consigue eliminar todas estas barreras, porque no basa su código en texto sino que utiliza una metáfora gráfica de programación basada en bloques: piezas encajables inspiradas en los bloques de construcción que los más pequeños (y muchos adultos que no quieren crecer) utilizan para hacer realidad sus arquitecturas imaginarias.

Los bloques de Scratch se encajan entre sí y se anidan los unos dentro de los otros para conformar las diferentes estructuras clásicas de la programación imperativa. En Scratch tenemos objetos, condicionales, estructuras iterativas, listas, variables y un gran sinfín de otros conceptos de programación, todos ellos convertidos en pequeñas piezas encajables.



Programando en Scratch 2.0 [https://www.flickr.com/photos/scratchedteam/7243640120/]

Algunos autodenominados puristas de la programación no creen que los bloques sean metáforas válidas y afirman que, en realidad, Scratch es simplemente una representación gráfica del "código real", es decir, del texto. Nada más lejos de la realidad. Si se conoce un poco el funcionamiento interno de Scratch, se entenderá que el intérprete está leyendo los bloques gráficos directamente desde la zona de programación. En ningún momento existe una traducción de los programas construidos mediante bloques a algún otro lenguaje textual. De hecho, la programación por bloques es mucho más cercana al esqueleto intrínseco de los lenguajes (los árboles sintácticos) que la textual. En los lenguajes textuales, a menudo nos vemos obligados a utilizar recursos visuales, como la inserción de tabulaciones, espacios y saltos de línea, para explicitar y entender la estructura de nuestros programas. En Scratch, esta estructura está incrustada en el propio lenguaje.

Cabe recordar que antes de los lenguajes textuales los programas se perforaban en tarjetas, y antes de ello el código se cableaba directamente en circuitos. Hay muchas maneras de construir programas, y no es en absoluto necesario traducir un programa de una metáfora a otra para que la máquina la entienda y la pueda interpretar.

En cualquier caso, la metáfora de los bloques pareció funcionar de maravilla, y pronto Scratch contó con millones de usuarios alrededor de todo el mundo. Se puede decir que éste fue el lenguaje que configuró la primera comunidad internacional de programadores infantiles, que compartían código sin fronteras, mejoraban sus programas colaborativamente y aprendían tanto de los errores y los éxitos de los demás como de los propios.

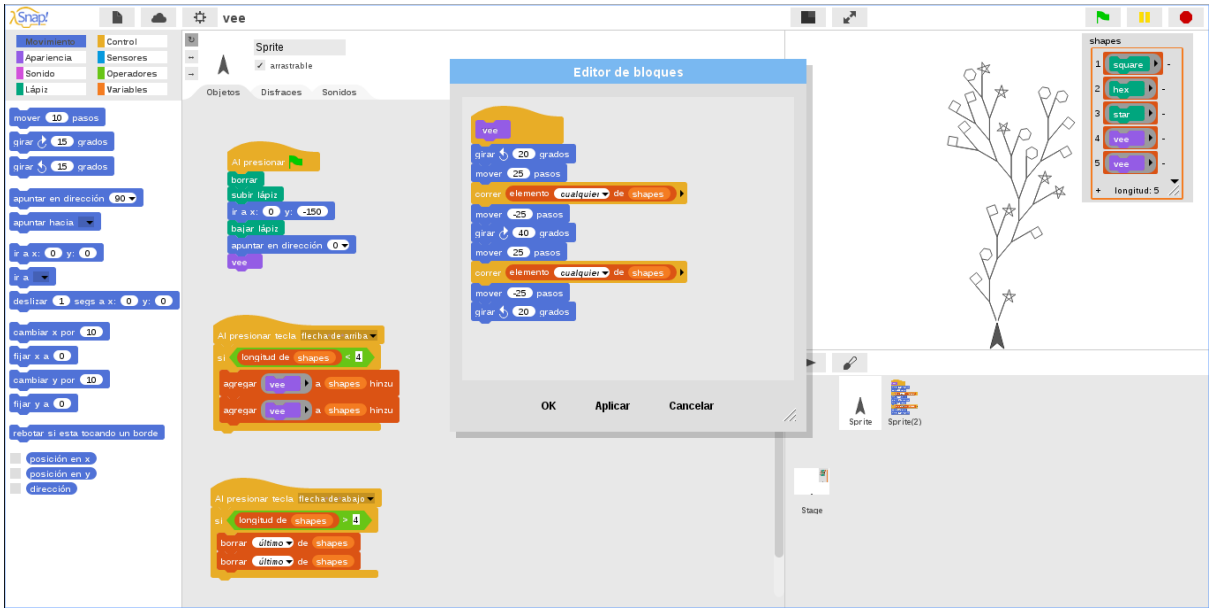
Mientras Scratch ganaba tracción, nuestra historia seguía su curso en la otra costa de Estados Unidos, en la Universidad de Berkeley, donde el doctor Brian Harvey estaba también

trabajando en el mundo de la enseñanza de la programación. De hecho, él y sus estudiantes ya habían implementado, en 1992, una de las versiones modernas de LOGO más utilizadas en todo el mundo. Cuando nació Scratch, Harvey vio un nuevo camino a explorar y se interesó por la metáfora de los bloques, pero echó en falta una serie de características y funcionalidades en el lenguaje del MIT que le impedían utilizarlo para enseñar computación a nivel avanzado.

En una conferencia sobre Scratch, Brian Harvey expuso estas limitaciones. En esa sesión estaba también Jens Mönig, un abogado en activo y ex-programador de Smalltalk (recordemos que Scratch estaba implementado en Squeak, una versión moderna de Smalltalk). Jens Mönig mostró su interés en implementar las funcionalidades que Harvey echaba en falta en el lenguaje. Ambos se pusieron manos a la obra poco después, desarrollando una extensión de Scratch llamada BYOB que tendría bastante repercusión por sus capacidades avanzadas.

Pero BYOB pronto empezó a verse en peligro cuando el equipo del MIT anunció que la siguiente versión de Scratch dejaría de funcionar sobre Squeak. Como el MIT no parecía interesado en liberar el código fuente de la nueva versión, Mönig y Harvey escogieron el difícil camino de implementar un clon de Scratch desde cero, que esta vez funcionaría directamente sobre la web.

Casi todos los grandes lenguajes de programación tienen dialectos, y Scratch no podía ser menos. Este primer dialecto de Scratch, que fue bautizado con el nombre de Snap*!*, permitiría ofrecer cursos de computación avanzados a estudiantes de primero de carrera de muchas universidades de Estados Unidos, tanto de carreras técnicas como de otras especialidades más alejadas de la programación.



Programa que demuestra recursividad, funciones anónimas, bloques personalizados y listas de primer orden en Snap!

Snap*!* ha resultado ser un lenguaje computacionalmente muy potente, y el mismo Brian Harvey dice que es, en realidad, un LISP disfrazado de Scratch. Además, su código fuente legible y bien estructurado ha propiciado la proliferación de multitud de modificaciones y

extensiones para finalidades diversas. Una de ellas, la que cierra esta historia, tuvo también su inicio (o quizás cabría decir su reinicio) en una conferencia.

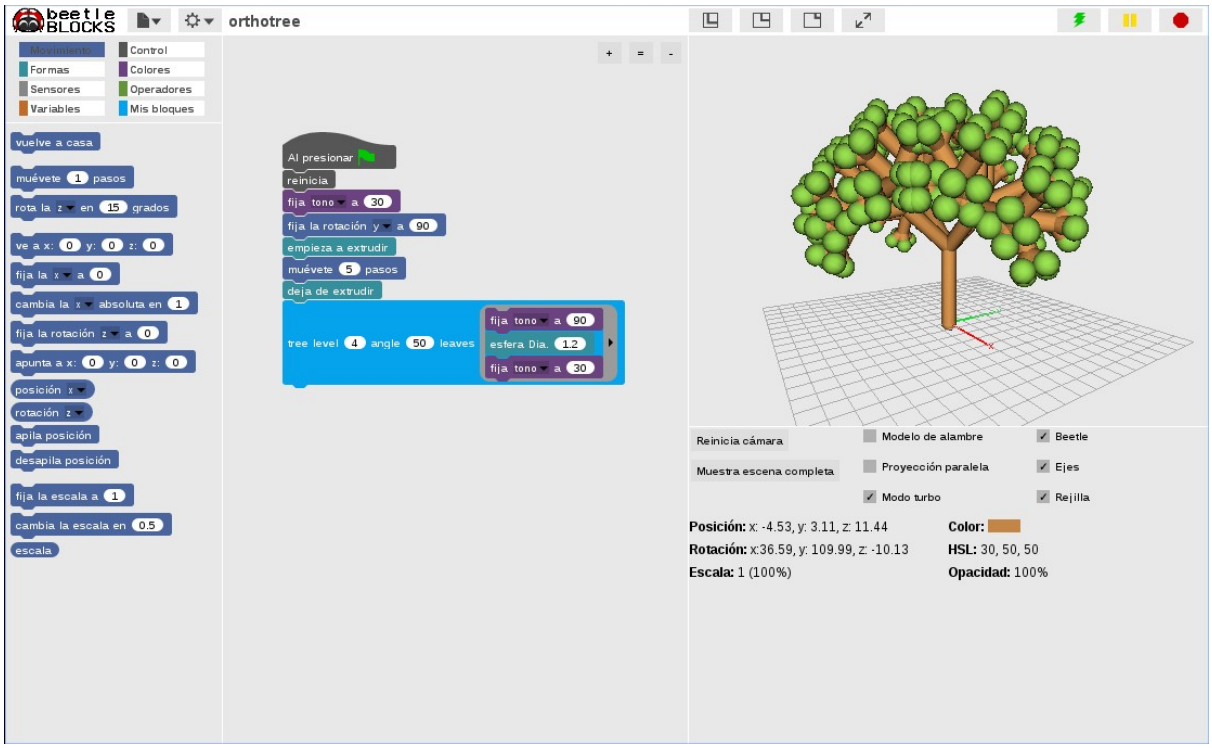
En verano de 2014, un par de miembros del Citilab de Cornellà de Llobregat asistimos a la conferencia bianual de Scratch que se organiza en el MIT Media Lab para presentar algunos de nuestros proyectos relacionados con Scratch. Entre ellos, una modificación de Snap*!* llamada Snap4Arduino que permite interactuar con placas Arduino, heredera directa de nuestra primera modificación de Scratch para el mismo propósito, S4A.

En esa conferencia, Eric Rosenbaum y Duks Koschitz propusieron a Jens Mönig la continuación de un proyecto que tenían en marcha, consistente en una extensión de Scratch para generar formas tridimensionales que podían exportarse para imprimir. Esta extensión estaba ya rozando sus límites, y consideraron que quizás Snap*!* podía ser la nueva base para resucitar el proyecto.

Jens Mönig, conociendo las modificaciones de Snap*!* y Scratch que habíamos desarrollado en el Citilab, nos invitó a esa reunión, donde Rosenbaum y Koschitz nos mostraron el prototipo de su extensión de Scratch, llamada BeetleBlocks. En la pantalla, un cono, que ellos aseguraban que representaba un escarabajo, se movía por el espacio tridimensional generando sólidos y extruyendo su trayectoria, formando construcciones maravillosas a partir de algoritmos. En BeetleBlocks, la tortuga de LOGO había ganado volúmen, y parecía poder unir, en cierta manera, los mundos del diseño, el making, la programación y el arte. Personas de distintas disciplinas podrían utilizar el lenguaje desde sus diferentes puntos de vista, y aprender, a veces sin saberlo, conceptos y habilidades de las otras disciplinas.

De esta reunión surgió el compromiso de iniciar la migración del código de BeetleBlocks a Snap*!*, lo cual, pasados unos meses, se formalizó en un acuerdo entre las cuatro partes. Así, se formó un equipo multidisciplinar en qué Duks Koschitz y Eric Rosenbaum liderarían el proyecto, Jens Mönig contribuiría al desarrollo en lo relativo a Snap*!*, y Bernat Romagosa, en nombre del Citilab, tomaría el rol de desarrollador del proyecto.

BeetleBlocks vio su primera versión pública en diciembre de 2014, y se ha estado utilizado activamente en una asignatura sobre diseño y computación que Duks Koschitz imparte en el Pratt Institute de Nueva York, lo cual ha permitido al equipo de desarrollo retroalimentarse de sus usuarios, añadiendo funcionalidades y retocando la interfaz gráfica para ajustarse a las peticiones y recomendaciones de los alumnos de Koschitz.



Árbol recursivo generado en BeetleBlocks

También en el Citilab hemos estado utilizando BeetleBlocks en diversos eventos de divulgación de la programación y actividades educativas. En nuestra primera experiencia con alumnos, enmarcada en una jornada de descubrimiento de nuevas profesiones para estudiantes de instituto, nos dimos cuenta de que lo que teníamos entre manos no era simplemente una herramienta para diseñar formas tridimensionales, sino también una puerta de entrada muy atractiva al mundo de la computación. Pocas veces se puede presumir de que un grupo de alumnos de 16 años, un viernes por la tarde y después de una larga jornada de presentaciones y actividades, te pidan alargar el taller un rato más y se lamenten por tener que abandonar el aula.

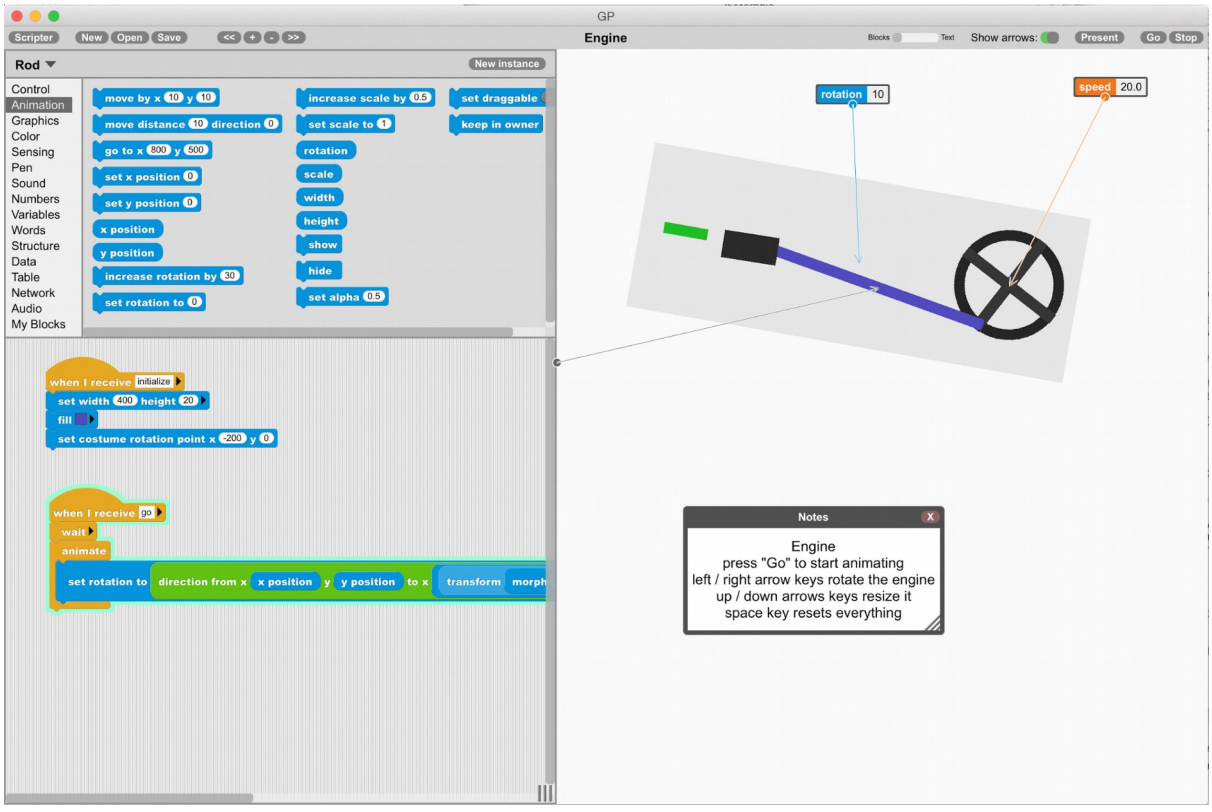
Otra de las experiencias que más nos han marcado ha sido la colaboración con Jo Milne, reconocida artista escocesa residente en Barcelona, que un día apareció en el Citilab con una peculiar estructura tridimensional sinterizada, inspirada en su particular aventura alrededor de la teoría de cuerdas y otras concepciones del universo. Con Milne, iniciamos un proyecto de colaboración entre arte y programación, plasmando las ideas de la artista en complicadas tiras de bloques encajados para poderlas luego parametrizar y modificar computacionalmente para convertirlas en obras interactivas.

Desde alumnos de instituto a estudiantes de diseño, pasando por artistas, BeetleBlocks está demostrando ser una herramienta versátil con que atraer a todos los públicos al mundo de la computación. Seymour Papert sabía, por Piaget, que el juego y la experimentación sin temor al fallo eran las claves para el autoaprendizaje, conformando herramientas tan potentes que todos los humanos aprendemos a hablar y a pensar justamente de este modo, sin necesidad de recibir clases de sintaxis o gramática. En LOGO, se aprendía a pensar computacionalmente bajo el pretexto de experimentar y jugar con una tortuga pixelada bajo nuestras órdenes. Rosenbaum y Koschitz han modernizado esa tortuga y le han dado el poder de moverse por el tercer eje del espacio, reduciendo además la posibilidad de error y la necesidad de memorizar

instrucciones con la adopción de la metáfora de la programación por bloques, y sin perder la idea principal del dinamismo y la inmediatez.

Pero el lento camino que la tortuga empezó a andar en los años 60 no termina con el gato de Scratch ni con el escarabajo de BeetleBlocks. Puede que el futuro nos brinde pronto una sorpresa de la mano de un equipo de investigación formado por John Maloney, programador de la máquina virtual de Squeak y miembro del equipo de desarrollo de Scratch, Yoshiki Ohshima, programador de Smalltalk involucrado en numerosos proyectos citados en este artículo, y Jens Mönig, de quien ya hemos hablado anteriormente.

Este grupo, que no por casualidad se encuentra bajo la tutela del mismo Alan Kay, se ha propuesto demostrar que, en efecto, la programación por bloques es una metáfora válida y no tiene por qué quedar relegada al ámbito educativo. Tomando prestadas las grandes ideas de Lisp, Smalltalk, Self, LOGO y Scratch, este equipo está desarrollando un nuevo lenguaje de programación por bloques con el que podremos implementar cualquier tipo de aplicación imaginable, tanto para teléfonos móviles como para servidores, ya sea para el mundo de la web o para tratar con bases de datos, e igualmente válido para un estudiante de secundaria que para un doctor en matemáticas. Haciendo honor a su intención de servir para cualquier propósito, han bautizado el nuevo lenguaje con el nombre de GP, acrónimo de General Purpose (propósito general) o, como suelen bromear, Guinea Pig (conejillo de índias), por si lo primero no acaba de funcionar.



Simulación de un émbolo en GP

En mayo de 2015, tuvimos el privilegio de que Jens Mönig nos visitara en el Citilab para ofrecernos una deslumbrante demostración en vivo de este nuevo lenguaje. Es muy

aventurado predecir el futuro, pero nos gustaría no equivocarnos al anunciar que, posiblemente, la tortuga de LOGO haya encontrado finalmente su forma definitiva.

La tortuga, lenta pero segura, abrió el camino de la programación dinámica educativa. El ratón de Squeak pavimentó el camino para, curiosamente, cederle paso a un gato. Aprovechándose de las enseñanzas de la tortuga y el gato, un escarabajo nos ha enseñado a programar en tres dimensiones. Si la idea de Papert era experimentar para aprender, ¿no es una inocente cobaya el animal perfecto para completar este zoo de la computación?

BIBLIOGRAFIA

Paulo Blikstein - Seymour Papert's Legacy: Thinking About Learning, and Learning About Thinking [https://tltl.stanford.edu/content/seymour-papert-s-legacy-thinking-about-learning-and-learning-about-thinking]
Cyntia Solomon - Logo Things [https://logothings.wikispaces.com/]
Wikipedia [http://wikipedia.org]
Edward B. Fiske - Interview: Seymour Papert on Computers [http://www.nytimes.com/1985/07/02/science/education-interview-seymour-papert-on-computers.html]