OCCAM'S RAZOR

ISSN: 1998-0537

Porque lo más sencillo es lo más probable

Número 6 · Invierno 2011

REALIDAD AUMENTADA
Interfaces Tangibles con ARToolkit

librerías
Leyendo Imágenes con un As en la Manga

misiones espaciales
Primeros Pasos con SPICE

historia
Homenaje Póstumo a la tv Analógica

SMOS
La Misión del Agua

imagen cortesía ESA

NO TE CORTES CON LA NAVAJA DE OCCAM

SUMARIO

Esta revista ha sido realizada con:

blender GIMP Emacs Vim LaTeX

Occam's Razor

Número 6. Invierno 2011

Dirección:
David Martínez Oliveira

Colaboradores:
Guillermo Buendicha, Roger Oliva Balagué, Manuel Martín-Neira, Jose Luis Vázquez García, Fernando Martín Rodríguez, Laura I. Rodríguez González, Er Imaginarius y el inefable Tamariz de la Perdiz

Maquetación y Grafismo
Laura I. Rodríguez González

Publicidad
Occam's Razor Direct
occams-razor@uvigo.es

Impresión
Por ahora tu mismo...Si te apetece

Consulta la página 50 para las excepciones a esta licencia

Editorial

Un Gran paso para Occam's...

by The Occam's Razor Team

Es otra vez esa época del año en la que Occam's Razor llega a vuestros discos duros intentando contaros algo interesante y que quizás no conozcáis todavía. Bien, aquí esta el numero 6 de Occam's Razor.

Como reza el título de esta editorial, este número supone "Un gran paso para esta revista". Y el contexto espacial no es gratuito. Es un gran paso porque, con este número, finalmente hemos conseguido movernos en la dirección que siempre hemos deseado, más contenidos científicos.
En este número encontraréis dos excelentes artículos de profesionales de la Agencia Espacial Europea, concretamente del Centro de Astronomía Espacial Europeo (ESAC) ubicado a las afueras de Madrid así como del Centro Espacial Europeo de Tecnología e Investigación (ESTEC) en los Países bajos.
En el primero de ellos Guillermo, Roger y Manuel nos describen con todo lujo de detalles la misión SMOS en la que España ha jugado un papel muy importante y bueno... mejor leed el artículo.
En el segundo, Jose Luis nos presenta SPICE, desarrollado por la NASA para el cálculo de geometría de misiones. Un excelente artículo que aúna de forma amena las operaciones espaciales y los ordenadores, a la vez que nos explica algunos, aparentemente sencillos, pero que se complican cuando salimos de nuestro querido planeta.
La sección de Historia regresa con un interesante artículo sobre la TV de la mano del siempre ameno Fernando Martín. No nos olvidamos de los temas más técnicos relacionados con los ordenadores a través de dos artículos relacionados con la imagen.
En el primero de ellos os hablamos de una interesante y ligera solución para leer imágenes en vuestros programas. El segundo nos introduce en el mundo de la Realidad Aumentada a través de ARToolkit.
Por último en la "La Cacharrería" os presentamos Teensy. Un microcontrolador con el que crear dispositivos de entrada, como por ejemplo teclados, es muy fácil.

Esperamos que este número os resulte interesante y desde estas líneas, la redacción de Occam's Razor os desea todo lo mejor para el próximo 2012!

The Occam's Razor Team

La Comunidad de Occam

Lista de Correo, Vídeos y más.

por The Occam's Razor Team

NUESTRA LISTA DE CORREO

La lista es totalmente abierta así que cualquiera puede leer las discusiones sin necesidad de suscribirse.

Aquí tenéis alguno de los temas que hemos tratado en la lista.

Hardware Abierto

Un tema de candente actualidad que seguimos modestamente en la lista, informando de nuevas iniciativas de hardware abierto como la tarjeta Snowboard, el reciente BeagleBone o el anuncio de la Convención de Hardware Libre, Electrónica y Robótica (OSHWCon 2011) celebrada en Septiembre.

También os hablamos de un interesante reportaje sobre Arduino que apareció poco después de publicar nuestro último número

Aquí tenéis los enlaces.

Más sobre SnowBoard...

Más sobre BeagleBone...

Más sobre OSHWCon 2011...

Más sobre Arduino, El documental...

Construye tu propia red de telefonía móvil

Un interesante proyecto para construir pequeñas redes de telefonía móvil, allí donde las grandes compañía no quieren llegar.

Mas...

El nombre del programa fantasma

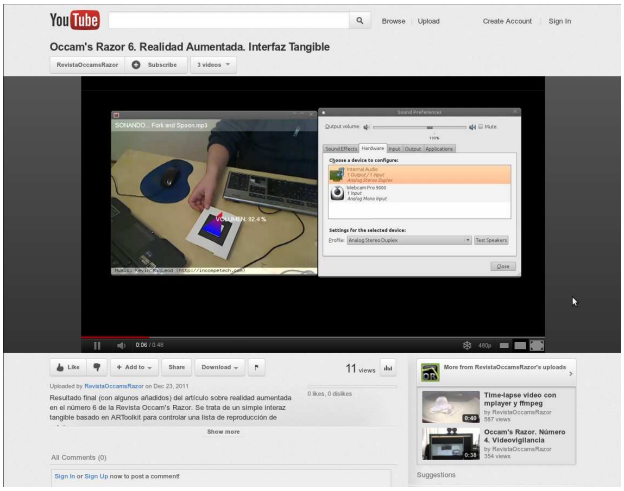
Nuestro amigo Carlos encontró un problema con el programa fantasma del que os hablamos en el último número. Todos los detalles en la lista de Occam's Razor.

Mas..

Los miembros de la lista reciben la revista antes que nadie y además tienen la posibilidad de ofrecer sus comentarios antes de que se publique oficialmente. Además, estamos preparando nuevas actividades relacionadas con la revista.

NUESTROS VIDEOS

En nuestro canal de YouTube podéis encontrar un par de vídeos sobre algunos de los artículos en este número. El primero es nuestra sencilla aplicación de Realidad Aumentada en vivo.



No os perdáis este video de los autores del artículo ("Interface Tangible Ex-Nihilo"). Que ha aparecido recientemente en Hack a Day

El segundo vídeo que hemos añadido ilustra nuestro proyecto de la cacharrería de este número.

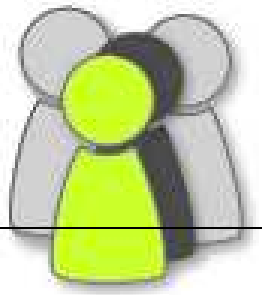
Si habéis producido algún video a partir de los artículos de Occam's Razor nos encantaría conocerlos y añadirlos como favoritos en nuestro canal o como video respuestas si están relacionados con alguno de los videos ya disponibles. ■

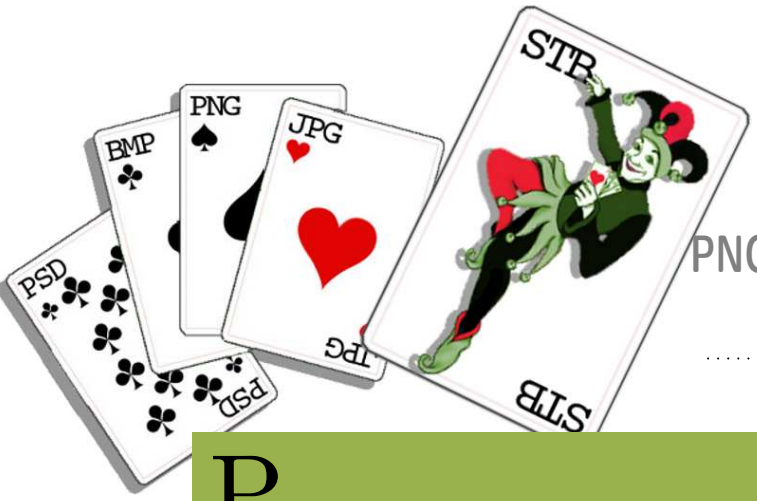
UNIROS A LA COMUNIDAD!

Nos vemos en nuestra lista de correo y canal de youtube.

Grupo Occam's Razor
Canal YouTube

OS ESPERAMOS!!!!





Leyendo Imágenes con Libertad

PNGs. JPGs o PSDs con una línea de código

por Er Imaginarius

Por una vez, y sin que sirva de precedente, no vamos a hablar de una librería en esta sección, aunque generar una librería a partir de `stb_image` sería trivial. `stb_image` es un fichero fuente C que nos proporciona funciones para leer en memoria los formatos de imágenes más populares con una sola línea de código.

`stb_image` es un conjunto de funciones escritas en el lenguaje C por Sean Barrett quién las liberó en el dominio público. En su página web encontraréis algunas otras funciones que también os pueden resultar interesante (descompresión Ogg Vorbis o *raterización* de fuentes TrueType).

Su uso es muy sencillo, como veremos enseguida, y nos permite añadir soporte para la lectura de imágenes en los formatos JPEG, PNG, BMP, TGA, PSD, HDR a nuestros programas, muy fácilmente. Es ideal si queremos evitar que nuestra aplicación tenga dependencias con otras librerías o si estamos pensando en generar un ejecutable estático para algún sistema *embebido*, ya que todo el código está contenido en un fichero .C. Sin más, aquí tenéis un ejemplo de como usar esta *ratilla*.

```c
#include "stb_image.h"

int main(int ac, char **a)
{
  int           w, h, c = 3;
  unsigned char *img;

  img = stbi_load (a[1], &w, &h, &c, 3);
}
```

Este sencillo programa, lee la imagen que le pasamos como primer parámetro en memoria, devolviéndonos un puntero a los datos. Además, nos devuelve el ancho y alto de la imagen, así como el número de componentes de color. El último parámetro nos permite indicar cuantas componentes de color queremos en nuestra imagen en memoria, es decir, le podemos pedir a la función que convierta la imagen por nosotros.

A modo de ejemplo, si al programa anterior le pasamos como primer parámetro una imagen con transparencia, el valor de `c` devuelto por la función será 4 (rojo, verde, azul y alpha), pero la imagen en memoria solo tendrá 3 componentes (rojo, verde y azul).

Para obtener el ejecutable, un sencillo:

```
gcc -g -o image image.c stb_image.c -lm
```

es suficiente. Observad que necesitamos incluir la librería matemática ya que `stb_image` utiliza algunas funciones como `pow` internamente.

Con `stb_image` podremos leer imágenes en nuestros programas de forma muy sencilla

Leer un jpeg en una línea de código está bastante bien, pero supongo que muchos de vosotros os preguntaréis como utilizar los datos que nos devuelve `stbi_load`.

PROCESADO DE IMÁGENES

Sí, el artículo habría quedado un poquillo corto de esta manera :), así que vamos a escribir un sencillo programa para modificar el contraste de una imagen de forma que podáis ver como acceder a los datos en memoria de la imagen.

El siguiente ejemplo, implementa la función de modificación de contraste de GIMP. Si no os lo creéis, podéis comprobarlo muy fácilmente.

Aquí está el programa

```c
#include <math.h>
#include "stb_image.h"

int main (int argc, char *argv[])
{
  int           i, w, h, c, size, _tf[256];
  unsigned char *img, *p;
  float         f, con;

  /* Carga imagen */
  p = img = stbi_load (argv[1], &w, &h, &c, 3);
  c = 3;
  size = w * h * c;

  /* Calcula funcion de transferencia */
  f = (tan (((con = 0.3) + 1) * M_PI/4));
  for (i = 0; i < 256; i++)
    {
      _tf[i] = (int)(i - 127) * (f) + 127.0;
      if (_tf[i] > 255) _tf[i] = 255;
      if (_tf[i] < 0) _tf[i] = 0;
    }
  /* Aplica funcion de transferencia */
  for (i = 0; i < size; i++, p++) *p=_tf[*p];
  stbi_write_bmp (argv[2], w, h, c, img);

  return 0;
}
```

Este sencillo programa toma dos parámetros, la imagen a la que queremos modificar el contraste, y el nombre del fichero en el que queremos grabar los resultados.

Veamos en detalle que es lo que estamos haciendo.

FUNCIONES DE TRANSFERENCIA

Modificar el contraste de una imagen, es una de esas operaciones de histograma. La particularidad de estas operaciones es que se pueden implementar de forma muy sencilla con lo que se llama una función de transferencia. En nuestro caso, una tabla que nos permite cambiar un determinado valor por otro que nosotros queramos.

Podemos grabar BMPs y TGAs con una línea más

En el ejemplo anterior, la función de transferencia es el vector `tf`, que contiene una entrada por cada posible valor de cada una de las componentes de la imagen. Como estamos trabajando con 8 bits por componente, la matriz debe ser de 256 valores.

Ahora solo necesitamos introducir en esa tabla el valor de salida que queramos para cada posible valor de las componentes de nuestra imagen y utilizar el bucle `for` al final del programa para aplicar la tabla. Observad que en nuestro ejemplo estamos aplicando la misma función a las tres componentes de la imagen, pero podríamos aplicar una función distinta a cada una de ellas. Eso dependerá de lo que queramos hacer.

Utilizando esta técnica podemos implementar un montón de ajustes de imagen: brillo, contraste, corrección gamma, inverso, posterización, monocromo, etc... Solo necesitamos poner los valores apropiados en el vector `_tf`.

VOLCANDO EL RESULTADO

Como ya os habréis dado cuenta `stb_image` también nos proporciona alguna función para grabar imágenes. En nuestro ejemplo estamos utilizando la función `stbi_write_bmp` para generar una imagen en formato `.BMP`. El otro formato que podríamos utilizar es `.TGA`.

AJUSTES FINOS

`stb_image` nos proporciona un montón de ajustes finos, para controlar que partes del módulo queremos utilizar y que partes no. A continuación os comentamos sus efectos.

`STBI_NO_WRITE`.

Si definimos este valor en nuestro programa (pasándole `-DSTBI_NO_WRITE` al compilador), eliminaremos el código para grabar imágenes (unos 7Kb). Pensad por ejemplo en un juego, en el que necesitáis cargar un montón de imágenes, pero no hace falta producir ninguna.

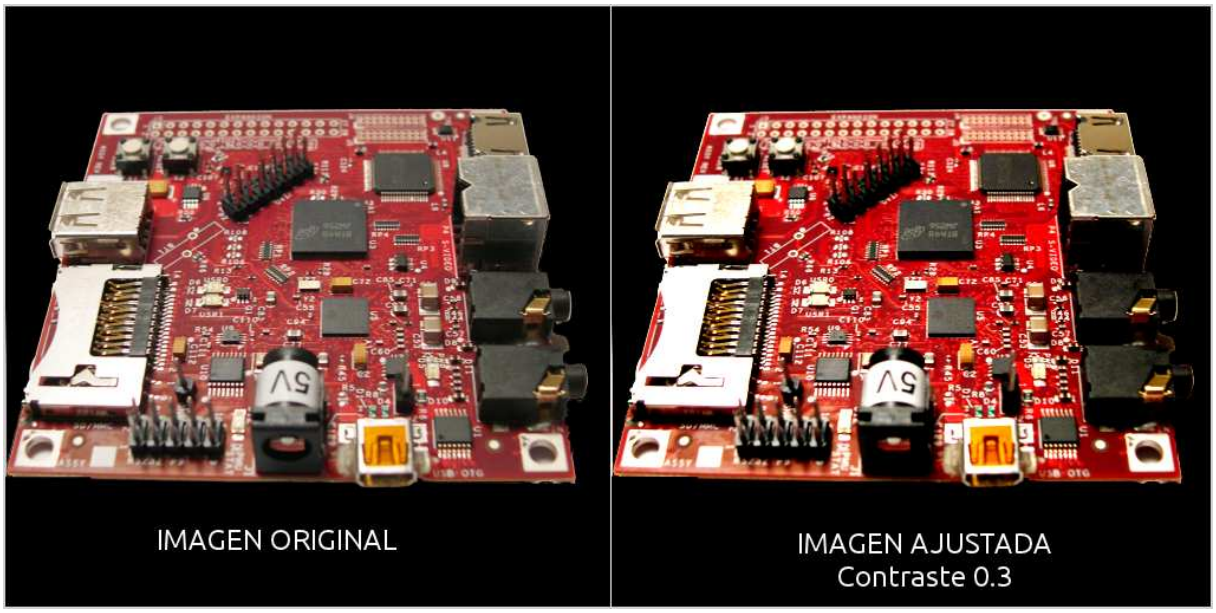
`STBI_NO_STDIO`.

Definiendo este valor durante la compilación de nuestra aplicación eliminaremos el código para leer imágenes de ficheros. Eso elimina la dependencia de `stdio.h` que añade unos cuantos Kbytes a nuestra aplicación.

En este caso, todavía disponemos de una función `stbi_load_from_memory`. ¿Cuando nos puede interesar utilizar esta opción?. En general cuando obtengamos nuestros datos desde cualquier otra cosa que no sea un fichero (una conexión de red, una webcam que produce imágenes JPEG, etc...)

HASTA LA PRÓXIMA

Esto es todo. Como siempre esperamos vuestros experimentos con `stb_image` y saber de los nuevos usos que se os ocurran. Podéis enviarnos vuestros resultados a nuestra lista de correo. Hay un montón de cosas interesantes que hacer con imágenes ahí fuera ;) ■



SMOS
La misión del Agua

Guillermo Buendicha
Ingeniero de Operaciones de MIRAS,
ESA–ESAC, Madrid, España

Roger Oliva Balagué
Coordinador de Calibración de MIRAS,
ESA–ESAC, Madrid, España

Manuel Martín–Neira
Ingeniero Principal de MIRAS
ESA–ESTEC, Noordwijk, Holanda

El presente artículo versa sobre la misión SMOS de la ESA, centrándose en sus objetivos científicos y tecnológicos, su evolución histórica, una breve descripción de los primeros dos años de vida, y finalmente una serie de retos que esperan en los años por venir. No pretende ser una descripción exhaustiva (se proporcionan referencias para el lector que esté interesado en un mayor nivel de detalle), sino más bien se dirige al público formado pero sin experiencia previa en el sector espacial.

IMAGEN Cortesía ESA

CONTEXTO PARA LA MISIÓN. EARTH EXPLORERS, LIVING PLANET

La misión SMOS (Soil Moisture and Ocean Salinity, o Humedad del Suelo y Salinidad de los Océanos) es uno de los llamados Earth Explorers, que son componentes integrantes del programa Living Planet de la Agencia Espacial Europea (ESA). La ESA mantiene actividades englobadas en el área de Observación de la Tierra desde 1977. A finales de la década de los 90, se inició el programa "Planeta Viviente", con una vertiente científica y de investigación más que de explotación de datos. El objetivo era comprender nuestro planeta como un sistema cambiante, modelar sus procesos y profundizar en la comprensión del cambio climático global.

El componente espacial de este programa (complementado con un componente terreno para la diseminación de los datos) son los seis primeros exploradores, familia de misiones que actualmente se está aumentando con 2 nuevas propuestas en estudio. Estos eran Cryosat, misión dedicada al análisis del espesor de las capas de hielo, destinada a ser la primera misión en volar, aunque un fallo en el lanzador en el 2005 hizo que se reiniciase como Cryosat II (finalmente lanzado en el 2010); GOCE, la misión que busca definir el mapa gravitatorio terrestre, lanzada en Marzo del 2009; SMOS, la misión del agua, que detallaremos más adelante y también volando desde finales del 2009; SWARM,

Programa *Living Planet* de la ESA (Cortesía ESA)

una constelación de 3 satélites para medir mejor el campo magnético terrestre, que se lanzará a lo largo del 2012; ADM Aeolus, pendiente de lanzamiento y dedicada al análisis de los vientos, y Earth-Care, que también será lanzada en los próximos años y dedicado al estudio de las nubes y la concentración de aerosoles en las capas altas de la atmósfera.

Todas estas misiones se complementarán con los Sentinels (Centinelas), familia de satélites que implementarán el programa GMES (conjunto entre la ESA y la Unión Europea) y con misiones desarrolladas en cooperación entre la ESA y Eumetsat. Como se puede ver, la observación y comprensión de nuestro planeta no es tarea sencilla, y requiere de un esfuerzo multidisciplinar.

HISTORIA

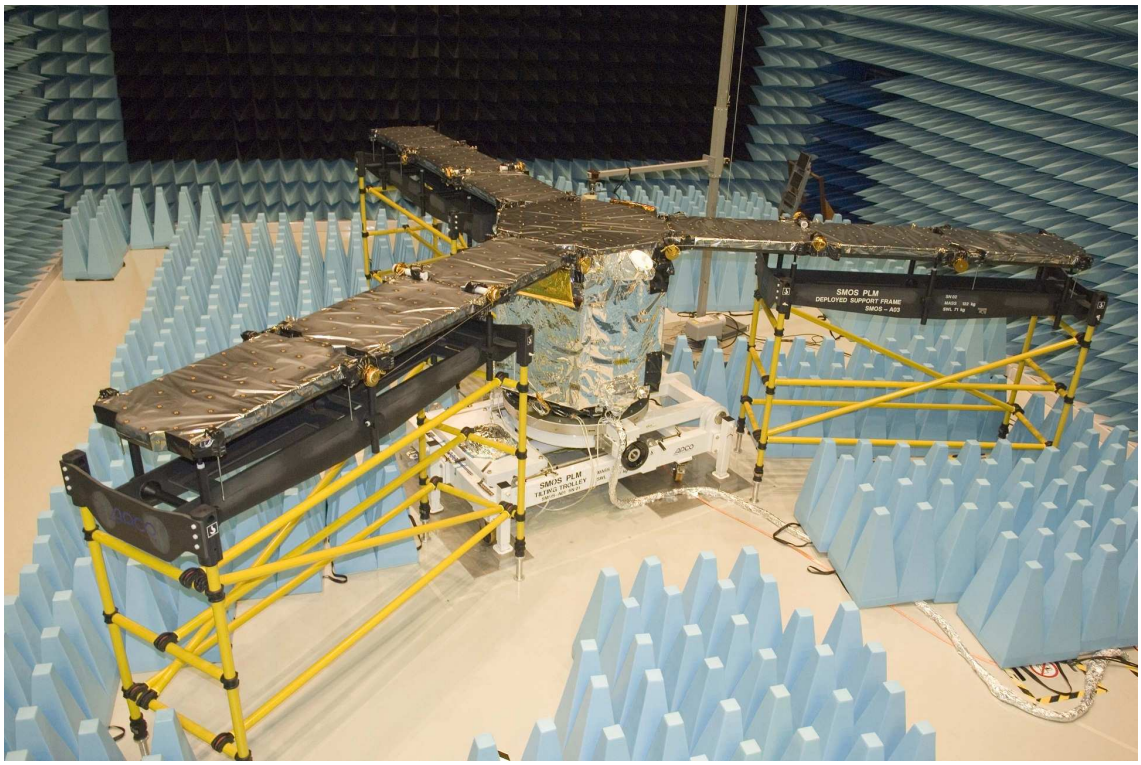
Respondiendo al concurso de propuestas para el programa Living Planet, se presentó en 1998 la misión SMOS, por parte del Insituto Francés CESBIO con Yann Kerr como investigador principal (PI) y el Instituto español de Ciencias del Mar con Jordi Font como co-investigador principal. La misión fue seleccionada por la ESA en el 99. Detrás de esto se halla el hecho de que tanto España como Francia fueron países dentro de la ESA altamente interesados en la suscripción a este programa en concreto.

Para dar una referencia de las escalas de tiempo en el desarrollo de una misión espacial, la fase A o de definición concreta de la misión se inició en el año 2000, con una fecha de lanzamiento previsto entonces para el 2007. Finalmente, como veremos, SMOS se lanzó a finales del 2009, llevando así a un ciclo de casi 10 años desde el comienzo del desarrollo de la misión una vez aprobada hasta su fase de explotación. Deberíamos añadir además un período anterior de estudios, desarrollo de la tecnología necesaria y experimentos previos, que comenzó ya en 1992. Este ciclo, de unos 18 años desde su concepción a su lanzamiento, es razonable y hasta ineluctable para una misión novedosa de tamaño medio como SMOS, pudiendo muchas veces ampliarse en el caso de misiones de mayor envergadura.

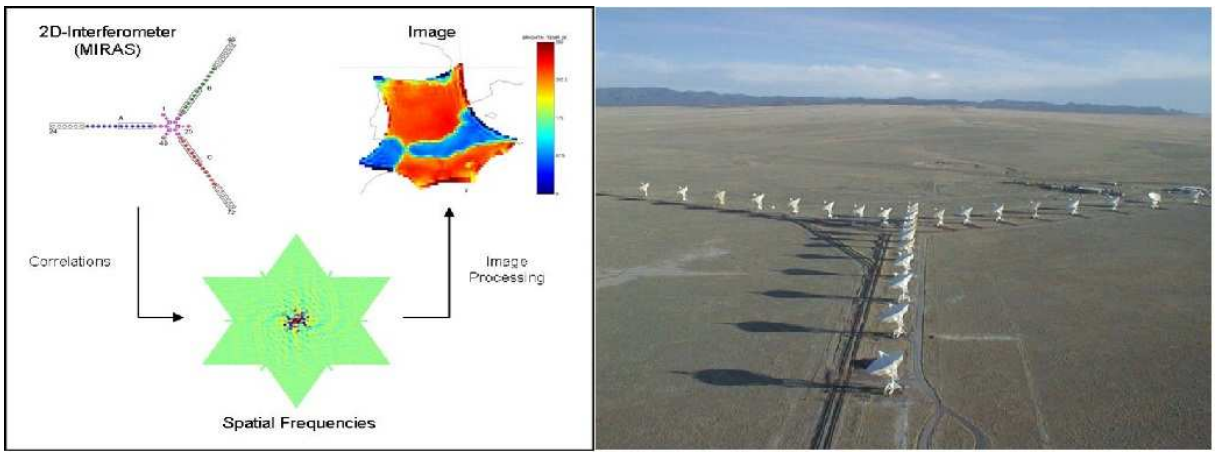
PROGRAMA INDUSTRIAL

Como se ha indicado, tanto Francia como España demostraron gran interés en el programa SMOS. A la hora de definir la contribución y el retorno industrial para estos países (siendo el retorno industrial una de las bases del funcionamiento de la ESA), Francia optó por el desarrollo de la plataforma del satélite, y por asumir las operaciones de la misma, y España escogió la construcción del instrumento así como el control y operaciones del mismo y el centro de procesado de datos científicos. Ambos países desarrollaron también, al margen de la ESA, centros de procesado científico de más alto nivel para la explotación a fondo de los datos.

La plataforma de SMOS, basada en el bus estándar PROTEUS, fue construida por Thales Alenia Space en Cannes, y el centro de operaciones se englobó en el centro del CNES francés en Toulouse. Esto facilitó el poder reutilizar conocimiento adquirido previamente, al haber operado CNES otras misiones basadas en PROTEUS.



Integración del instrumento MIRAS en SMOS (cortesía ESA)



Interferometría (cortesía ESA)

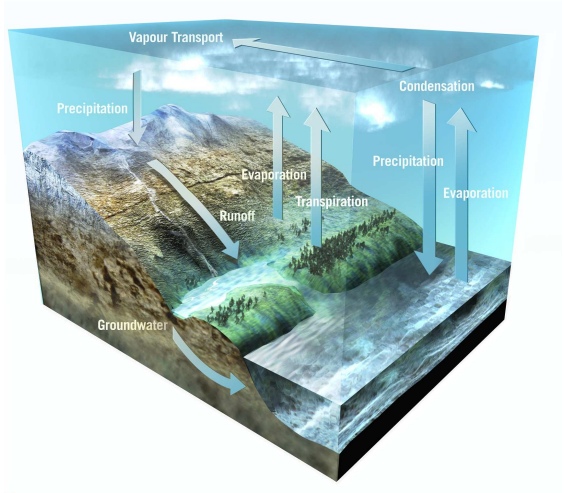
El instrumento de SMOS, MIRAS, fue desarrollado por la empresa EADS-CASA en Madrid, como principal contratista al frente de un amplio consorcio de empresas españolas y europeas entre las que se puede mencionar la participación de MIER, RYMSA, YLINEN, CONTRAVES, ASTRIUM-Munich, LABEN, Austrian Aeorspace, SENER y TECNOLOGICA.

El centro de operaciones y de procesado de datos del instrumento fue desarrollado por un consorcio liderado por la empresa INDRA, en el que participaron GMV y Deimos entre otras empresas.

La ESA asumió el papel de coordinador global del proyecto, gestionando las distintas fases del programa, y buscando el lanzador para el satélite, que se contrató finalmente con EUROCKOT, empresa germano rusa que utiliza el cohete Rockot desde la base de Plesetsk en Rusia.

LÓGICA DE FUNCIONAMIENTO

El objetivo de SMOS se puede resumir en la adquisición a nivel global y con periodicidad de dos variables claves para comprender el ciclo del agua: la salinidad superficial de los océanos, y la humedad retenida en los primeros centímetros del suelo.



Ciclo del agua (Cortesía ESA).

La propuesta científica propuso la implementación de un radiómetro que captase la emisión superficial en banda L (1.4 Ghz), debido a que esa banda es una banda protegida para la investigación científica, y además a que de la citada emisión se pueden inferir las dos variables buscadas.

El objetivo de la misión, sobre todo en su componente de humedad superficial, exigía un tiempo de revisita (tiempo que tarda SMOS en volver a medir un mismo punto en la superficie terrestre) de 3 días en el ecuador, y una resolución espacial o tamaño de pixel de 30 a 50 Kms. Eso exige a la vez un campo de vista amplio (aproximadamente a 1000 Kms) y un tamaño de pixel relativamente pequeño para el mismo, asumiendo una órbita síncrona solar a unos 760 Kms de altitud.

SMOS utiliza un radiómetro que capta la emisión superficial de banda L

La solución encontrada, frente a una única antena parabólica con barrido mecánico –cuyo tamaño no permitía cumplir con los requisitos de la misión ni encajaba en lanzadores disponibles– fue usar la interferometría desde el espacio. La idea consistía en componer una antena de apertura sintética con un array en Y griega de 69 detectores distribuidos en tres brazos a 120 grados y un Hub central. Esto permitía obtener una antena equivalente de unos 8 metros de diámetro, que podía plegarse gracias a sus segmentos articulados a la hora del lanzamiento, y posteriormente desplegarse en la configuración final de vuelo.

SMOS utiliza por tanto receptores pasivos, con capacidad de polarización tanto vertical como horizontal, y utiliza los principios interferométricos para obtener cada 1.2 segundos una imagen de la escena basada en la correlación de las distintas señales recibidas en cada uno de sus receptores.

Esta técnica es pionera por cuanto que la interferometría radiométrica nunca se había probado desde el espacio para observar la Tierra. La ESA planteaba un desafío tecnológico y científico importante: hacer radio-astronomía de la tierra. Para ello fue necesario una revisión de las ecuaciones clásicas de interferometría utilizadas en Radioastronomía, lo que fue en gran parte posible gracias a la definición y análisis de las mismas que realizó el Dr. Ignasi Corbella (UPC) reflejados en la ecuación que lleva su nombre.

Esta selección con una antena sintética impone así mismo retos que son nuevos para la ESA, como son la selección de 69 receptores con características extremadamente similares, y sobre todo, el que a la hora de computar sus correlaciones sea necesario una transmisión de información entre ellos que obliga a utilizar una red distribuida de fibra óptica, siendo SMOS la primera misión ESA que lo implementa.

Así mismo, SMOS necesita una red interna de calibración para poder evaluar el adecuado funcionamiento de los receptores.

SMOS es un satélite de tamaño medio, de 650 Kgs de masa de los cuales 350 corresponden al instrumento MIRAS. Algunos de los principales parámetros de la misión pueden verse en la siguiente tabla:

Misión	SMOS
Lanzamiento	02-Nov-09
Duración	Mínimo años
Instrumento	Microwave Imaging Radiometer using Aperture Synthesis - MIRAS
Concepto de Instrumento	Passive microwave 2D-interferometer
Frecuencia	L-band (21 cm-1.4 GHz)
Número de receptores	69
Espaciado entre receptores	0.875 lambda = 18.37 cm
Polarización	H & V (polarimetric mode optional)
Resolución espacial	35 km en el centro del campo de visión
Ángulo de inclinación	32.5 grados
Resolución radiométrica	0.8 - 2.2 K
Rango angular	0-55 grados
Resolución temporal	Revisita cada días en el Ecuador
Producción de datos	89 kbps H & V pol.
Masa	Total 658 kg en lanzamiento, incluyendo plataforma 275 kg, carga útil 355 kg, fuel 28 kg
Órbita	Síncrona solar, sobre el terminador, cuasi circular con altitud de 758 km. Hora solar del nodo ascendente a alas 06.00 hrs.
Lanzador	Rockot, KM-Breeze upper stage
Plataforma	Proteus (1 m cube)
Potencia	Hasta to 1065 W (511 W para MIRAS; batería de 78 AH Li-ion.
Centro de Operaciones del Satélite	CNES,Toulouse, France
Enlace en banda S (TTC)	4 kbps uplink, 722 kbps downlink
Centro de control de instrumento y procesado de datos	ESAC, Villafranca, Spain
Enlace en banda X ™	16.8 Mbps

Tabla de datos de misión.

ÓRBITA DE SMOS

La órbita seleccionada para SMOS, clásica en las misiones de observación de la tierra, es una órbita cuasi polar (es decir, que el satélite orbita la Tierra pasando de polo a polo) con una altitud de 760 Kms, lo que marca un periodo orbital de 100 minutos. La órbita no es estrictamente polar, sino que el plano de la misma tiene una inclinación de unos 8 grados respecto al eje terrestre, lo que permite que el plano de la órbita de SMOS se mantenga siempre perpendicular al vector del Sol. Esto es muy adecuado, ya que los paneles solares mantienen una orientación constante y un rendimiento óptimo.

La altitud de la órbita define el compromiso entre cubrir haz amplio de terreno (mayor cuanto más alta la órbita) y una precisión de pixel ajustada (mejor cuanto más baja la órbita). Al tiempo, se ha seleccionado de forma que el paso ascendente por el Ecuador (conocido como nodo ascendente) ocurre siempre a las 06:00 horas solares locales del punto de paso, por lo que las condiciones en el suelo son siempre las mismas. Esto es esencial para una evaluación consistente de los datos.

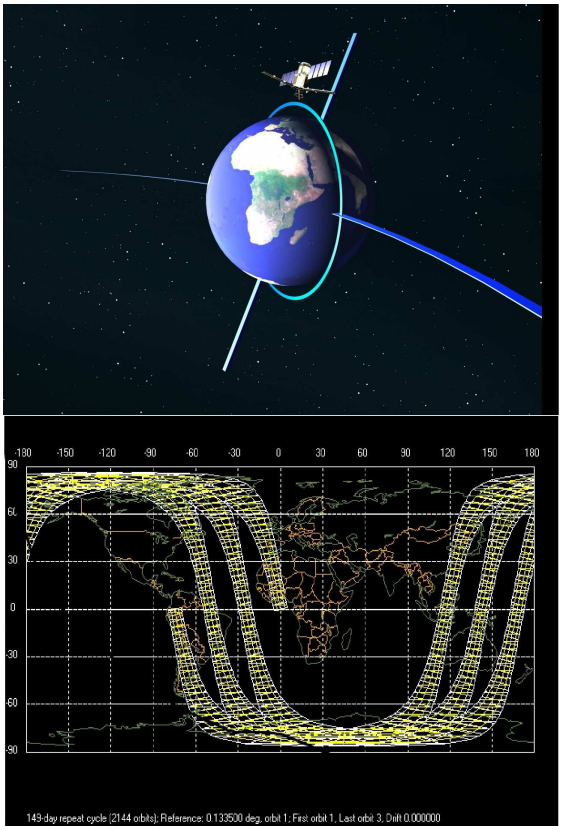


Gráfico de la órbita (Cortesía ESA).

Como ya se ha mencionado, MIRAS es un instrumento de gran tamaño, más de 350 kgs de peso y una envergadura de unos 8 metros de diámetro con los brazos desplegados. Consta de una unidad central (Hub) de forma hexagonal, que alberga unidades de electrónica (ordenador central, memoria de almacenamiento, correlador, transmisor en banda X), y de una serie de segmentos (12) que se configuran en 3 brazos en Y griega, con 4 segmentos cada uno, uno en el hub y 3 desplegados en segmentos exteriores. Los 69 receptores o LICEF de MIRAS se ubican en esos segmentos (6 en cada uno), habiendo 3 unidades especiales que tienen un doble receptor en ellas, llamadas NIR (Noise Injection Radiometer).

Unas unidades distribuidas (CMN's) en cada segmento se encargan de proporcionar la señal de un oscilador local para los LICEF de su segmento, de recolectar las señales de los mismos para el correlador, y de ejecutar los comandos y recolectar telemetría de estado.

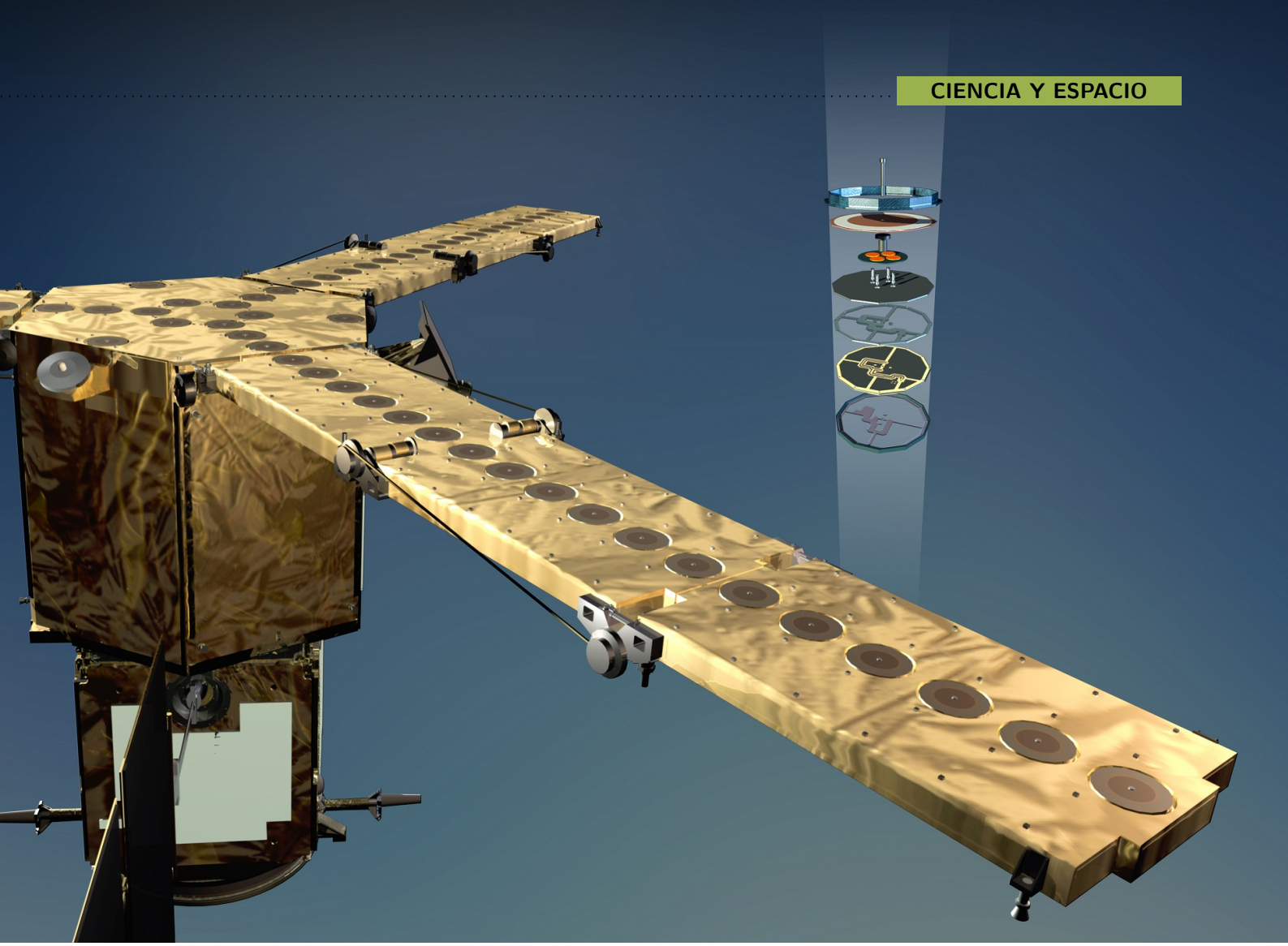


Imagen del Instrumento. Gráfico LICEF (Cortesía ESA)

Una red de calibración se encarga de alimentar, cuando es requerido, cada receptor con ruido correlado o no correlado, para medir la estabilidad y caracterizar el instrumento.

LANZAMIENTO, LEOP Y COMISIONADO

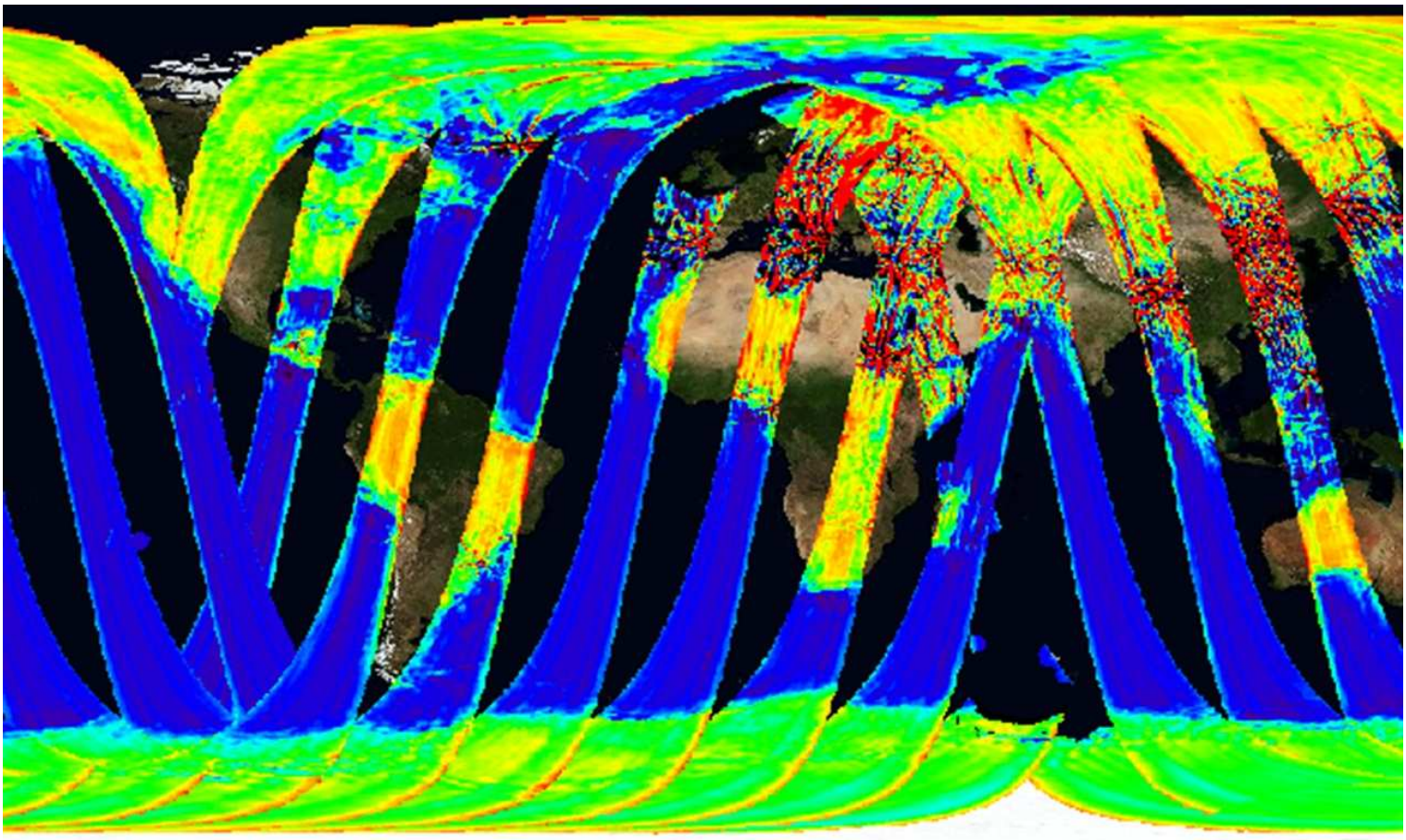
SMOS fue lanzado por un cohete Rockot (adaptado de antiguos misiles intercontinentales rusos) el 2 de Noviembre del 2009, a la 1:50 UTC desde el cosmódromo de Plesetsk, en Rusia. Rockot es un sistema de lanzamiento que incluye una última etapa, Fregat, que permite una inserción bastante precisa en la órbita, minimizando así el consumo de combustible cargado en el satélite. El lanzamiento fue todo un éxito, a las pocas horas de alcanzar la primera órbita SMOS pudo desplegar los paneles solares y también "abrir" los brazos del instrumento MIRAS, siendo ésta una de las maniobras más críticas para la misión.

Las primeras dos semanas de vida se dedicaron a ajustar la órbita de SMOS, y a realizar la comprobación de funcionamiento de la plataforma, lo que se llama el comisionado de la misma. En este periodo se verifican todos los subsistemas, y se alcanza la estabilidad térmica del satélite. En esta fase el instrumento MIRAS permaneció apagado, con un control térmico basado en sensores y resistencias eléctricas proporcionados por la plataforma.

Finalmente, el 17 de Noviembre el instrumento fue encendido, pasándose a un control térmico activo interno que mantiene los subsistemas estables en torno a 22 grados centígrados. La tarde de ese mismo día los primeros datos fueron adquiridos por la estación de recepción en ESAC, obteniéndose lo que se viene en llamar la "primera luz".

Durante las dos primeras semanas se ajusto la órbita y se comprobaron todos los sistemas de SMOS.

Siguió una fase de aproximadamente 2 semanas donde se realizaron unos chequeos para verificar el adecuado funcionamiento del instrumento, de los datos generados y de la capacidad de adquisición y procesado por el segmento de tierra, culminada con éxito.



Primera Luz SMOS (cortesía ESA)

Posteriormente, una fase de unos dos meses en los que se analizó más en detalle el funcionamiento científico del instrumento, incluyendo alguna modificación de la configuración y redundancia seleccionada en el lanzamiento, y una prueba de funcionamiento a distinto nivel térmico.

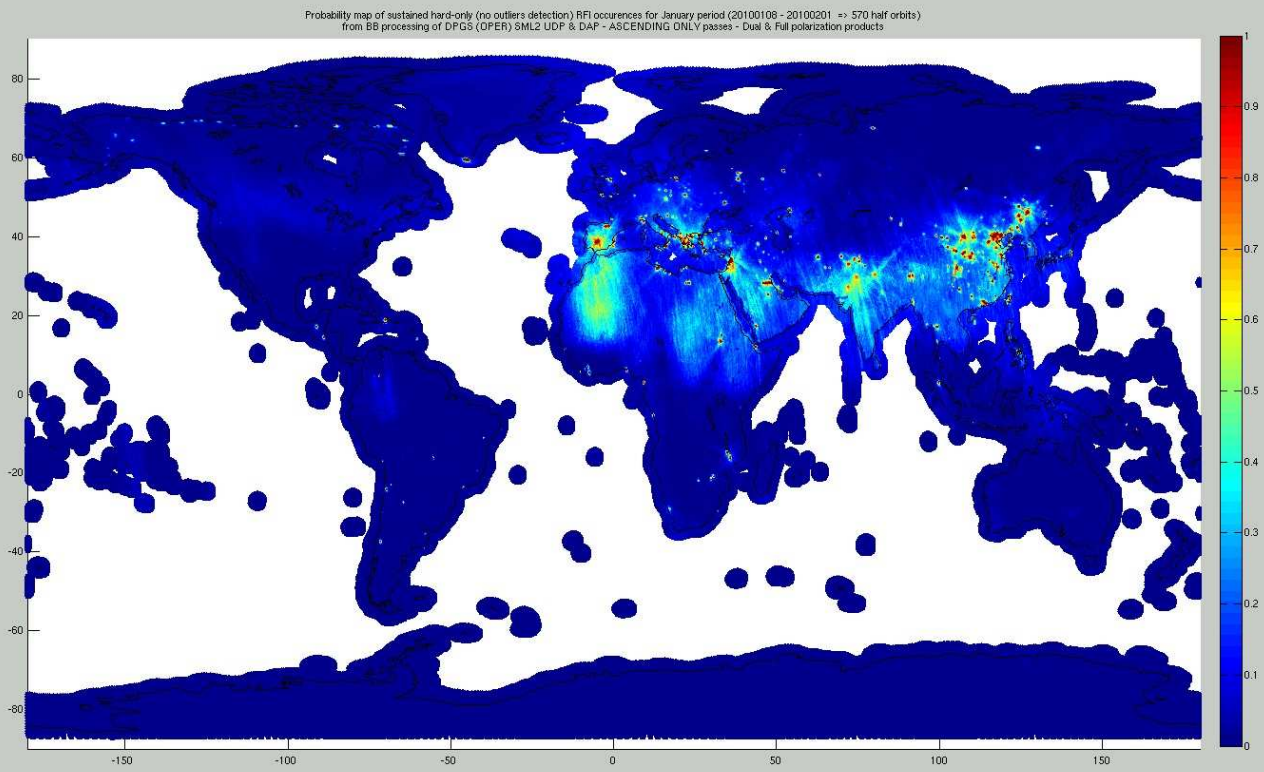
Finalmente, se concluyó la fase de comisionado del instrumento con 4 meses de operaciones donde se alternaron distintos escenarios y modos de MIRAS (con o sin medición de polarización cruzada) y un estudio de las frecuencias y secuencias óptimas de calibración basado en las medidas en órbita. También se analizaron y caracterizaron distintos parámetros del sistema (precisión geométrica, caracterización de antenas) y se compaginaron las observaciones de MIRAS con distintas campañas en tierra para validar las medidas del instrumento. En Mayo del 2010 se inició la fase de explotación regular, con una configuración fija para la misión, y con un plan de calibración estable.

La fase de comisionado de SMOS, prácticamente 6 meses, ha sido más larga de lo que es habitual en otras misiones de corte científico, debido principalmente al carácter absolutamente innovador del concepto de radiometría interferométrica desde el espacio, pero al mismo tiempo ha sido necesaria y esencial para que la fase nominal de la misión pueda proporcionar el máximo de datos científicos con la mejor calidad posible.

La fase de comisionado desveló lo que están siendo los grandes retos de SMOS a la hora de cumplir sus objetivos científicos. Posiblemente el mayor de ellos sea que la banda L en la que recibe MIRAS, teóricamente protegida por la regulación internacional del espacio radio-eléctrico, se encuentra en realidad contaminada por emisiones de sistemas defectuosos o ilegales. Los datos de SMOS descubrieron un gran número de estas emisiones espurias humanas en la banda protegida, que se daban en general por todo el planeta pero especialmente numerosas en Europa, Oriente Medio, y sudeste asiático.

Esas emisiones enmascaran la señal natural que observa SMOS dado que las emisiones humanas están muy por encima de los niveles naturales de radiación. Además, debido a las características del instrumento, una sola antena interferente puede afectar a SMOS aún cuando esté a miles de kilómetros del mismo.

Por otro lado, se detectaron unidades a bordo que se demostraron sensibles a la radiación (la órbita de SMOS atraviesa las zonas polares, donde la concentración de partículas cargadas es mayor, pero también la Anomalía del Atlántico Sur, donde los cinturones de radiación terrestres están más cercanos a la superficie) y a la iluminación solar. La configuración óptima frente a esos efectos fue seleccionada para garantizar el mejor retorno científico; hoy en día siguen presentes, pero su impacto en el retorno científico es mínimo.



Mapa de probabilidad de interferencia, (cortesía CESBIO)

SEGMENTO DE TIERRA Y OPERACIONES DE LA MISIÓN

El segmento de tierra de SMOS está compuesto por un segmento de control de plataforma, SOGS, en CNES, Toulouse, que se encarga de enviar todos los comandos (plataforma e instrumento) al satélite a través de una red de estaciones de Telecomando y Telemetría en banda S, así como de recabar la telemetría de la plataforma y una copia de la telemetría del instrumento, que posteriormente es enviada al Centro Europeo de Astronomía Espacial de la ESA en España, ESAC, cerca de Madrid. Desde el CNES se mantiene la actitud o apuntamiento del satélite, la órbita dentro de los parámetros requeridos, y se calculan posibles riesgos de colisión con restos de basura espacial. Al mismo tiempo, se garantizan la seguridad de la plataforma y del instrumento en casos de anomalías que puedan poner en riesgo la integridad de la misión.

En ESAC, se lleva a cabo el control y operaciones del instrumento, incluyendo la planificación semanal de las operaciones y calibraciones que ha de realizar, generando los comandos para MIRAS que se enviarán a través de CNES, y analizando los datos de telemetría de SMOS que se reciben tanto a través de CNES como del enlace en banda X con el instrumento.

En ESAC, así mismo, se reciben los datos científicos (recogidos por dos estaciones de seguimiento terreno, en ESAC y en SVALBARD, Noruega), y se realiza el procesado de datos hasta generar los productos de nivel 2 de salinidad y humedad superficial. En paralelo, ESAC tiene la capacidad de realizar campañas de reprocesado de datos de SMOS, tarea que comparte con el centro de la ESA en Kiruna, Suecia.

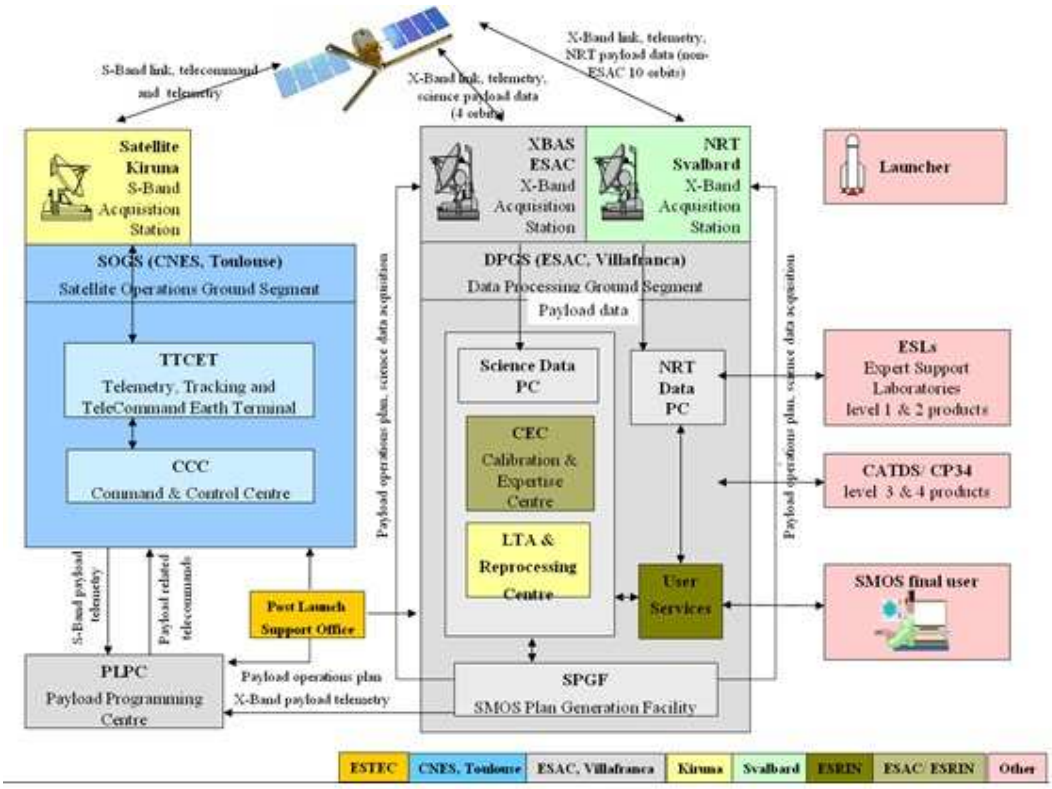
Por último, ESAC también alberga el grupo de calidad, que garantiza que el nivel de la data de SMOS es adecuado para su diseminación a la comunidad de usuarios.

ESAC es un centro clave en la misión SMOS.

ESRIN, el centro de la ESA en Italia dedicado a Observación de la Tierra, ejerce de centro de gestión del programa, y alberga el segmento de diseminación de datos a usuarios así como el de centro de asistencia a los mismos.

El centro tecnológico de la ESA en Holanda, ESTEC, alberga la oficina encargada de mantener el adecuado nivel de soporte industrial tras el lanzamiento para garantizar el máximo nivel de funcionamiento de la misión.

Y finalmente, un amplio número de centros expertos se encargan de colaborar para definir el mejor procesado posible para los datos de SMOS.



Segmento de tierra de la misión SMOS (Cortesía CNES-ESA)

FASE DE RUTINA, OPERACIONES Y PROCESADO DE LA SEÑAL

SMOS es una misión de escaneo periódico de la tierra. Esto quiere decir que en principio, mantiene una posición fija (con una inclinación de unos 32 grados hacia delante respecto a un apuntamiento vertical, perpendicular a la superficie de la Tierra) y recolecta datos aprovechando que la órbita polar le permite cubrir el planeta cada 3 días. Su modo de observación es fijo, siendo el que ubica los distintos receptores o LICEFs en polaridades distintas a lo largo del tiempo de integración de 1.2 segundos, obteniendo así una señal que obtiene los 4 parámetros de Stokes.

Entendida así, las operaciones de la misión pudieran parecer sencillas. Dos operaciones se entremezclan con este escenario rutinario:

La fase de rutina entremezcla dos operaciones: Transmisión de datos y calibraciones periódicas.

Una es la necesidad de trasmitir a tierra los datos adquiridos por SMOS. La misión contiene un requisito de datos en tiempo cuasi real (que pide que datos procesados hasta un determinado nivel estén disponibles para los usuarios en menos de tres horas desde la toma de la imagen), que impone que SMOS vuelque en cada órbita de 100 minutos los datos adquiridos en ella. Dos estaciones de adquisición, en la isla de Svalvard, Noruega (en una latitud alta, y por tanto con visibilidad de todas las órbitas de SMOS) y otra en ESAC (dedicada en exclusiva a SMOS) garantizan esto. Hay que comandar al instrumento para que cuando entre en zona de visibilidad por la estación de tierra vuelque los datos almacenados en la memoria a bordo, y actualice sus punteros.

La segunda es la necesidad de realizar actividades de calibración periódicas de forma que los datos de SMOS sean estables y precisos. Las distintas secuencias de calibración garantizan la estabilidad de todos los componentes del instrumento y regulan la sensibilidad de estos con variaciones temporales y térmicas. Una calibración periódica (cada 10 minutos) corrige las variaciones observadas debido a las variaciones térmicas en órbita, controlando la estabilidad de las diferencias de fase entre los osciladores locales asociados a cada pareja de receptores. La calibración de cada uno de los receptores se consigue con dos secuencias de calibración que verifican la señal enviada por unos diodos internos de referencia, una larga (cada dos meses) y otra mucho más corta (cada semana) que proporcionan información de los desvíos que haya sufrido cualquier receptor. El nivel absoluto de la imagen se contrasta cada dos semanas con la única señal externa de referencia para esta misión: las radiación cósmica de microondas, una reminiscencia del Big Bang que se observa girando el instrumento hacia el espacio exterior.

Asimismo, esta señal cósmica es aprovechada cada semestre para corregir cualquier desviación que hayan sufrido las antenas a bordo. Estas dos últimas secuencias de calibración requieren un giro del instrumento de forma que MIRAS observe al espacio a 180 grados respecto a la Tierra. Estas actividades han de coordinarse y convertirse en comandos de MIRAS y peticiones para el centro de control de la plataforma en CNES.

El instrumento MIRAS a bordo de SMOS tiene una capacidad de almacenamiento de 20Gb.

Los datos adquirida por MIRAS (tanto la de estatus del instrumento e información relevante de sus subsistemas, como la de las correlaciones entre receptores) es almacenada en la memoria del instrumento (con capacidad para 20 Gb, y posibilidad de almacenar más de un día de datos sin sobrescribirse), y transmitida a tierra. Constituye los datos crudos o "raw".

Éstos son recibidos en el centro de procesado de datos (DPGS en ESAC), y convertida primeramente en productos ordenados por semiórbitas, al tiempo que clasificados en distintos tipos de producto en función de los datos contenidos (correlaciones, calibración, estatus de instrumento, etc.). Es lo que se llama procesado de nivel 0.

Seguidamente, se corrige usando los datos de las distintas calibraciones y luego se recompone la imagen de SMOS aplicando una técnica similar a una transformada inversa de Fourier. Esa imagen, obtenida en el plano de antena, se proyecta a nivel de tierra, incluyendo entre otras cosas la variación de la señal debida a la rotación de Faraday debida a la presencia de carga eléctrica en la ionosfera terrestre. Es el procesado de nivel 1, a menudo escindido en distintos subniveles para identificar en qué punto del procesado nos hallamos. Al final de este nivel, obtendremos mapas de Temperatura de Brillo (emisividad en banda L).

Finalmente, dos procesados diferenciados se encargaran de convertir la temperatura de brillo en salinidad o en datos de humedad. Para ello, es preciso una segregación geográfica (tierra-mar), y luego aplicar distintos algoritmos para derivar la variable geofísica de la medida. Sin entrar en detalles, pero a título de ejemplo, la información de la salinidad del mar depende del conocimiento también de la temperatura física superficial, y sobre todo de la información de la rugosidad del mismo, debida al oleaje. Eso hace que se precisen datos complementarios a SMOS para estos procesados. Para la humedad superficial es necesario entre otros el conocimiento del índice vegetativo o cantidad de biomasa de la zona en cuestión, siendo esto algo con variabilidad estacional, así como información de la rugosidad del suelo.

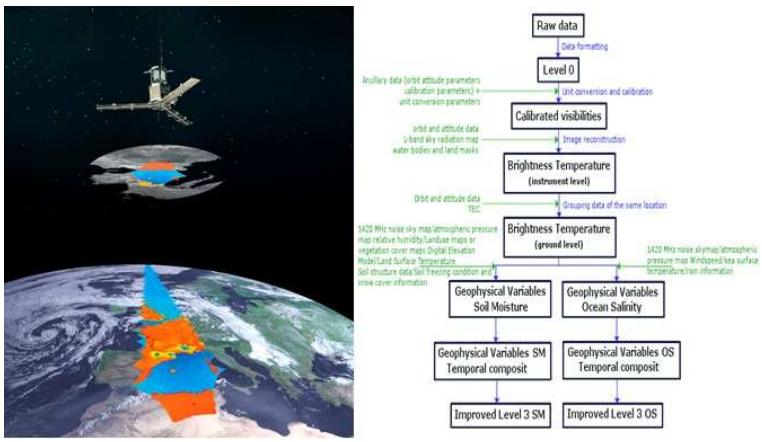
Los mapas globales de salinidad y de humedad constituyen los dos productos de nivel 2 que representan el final de la cadena operacional de procesado.

Todos los productos generados pasan una serie de controles de calidad para evitar que problemas o errores del procesador en tierra o el instrumento originen que datos falsos sean diseminados.

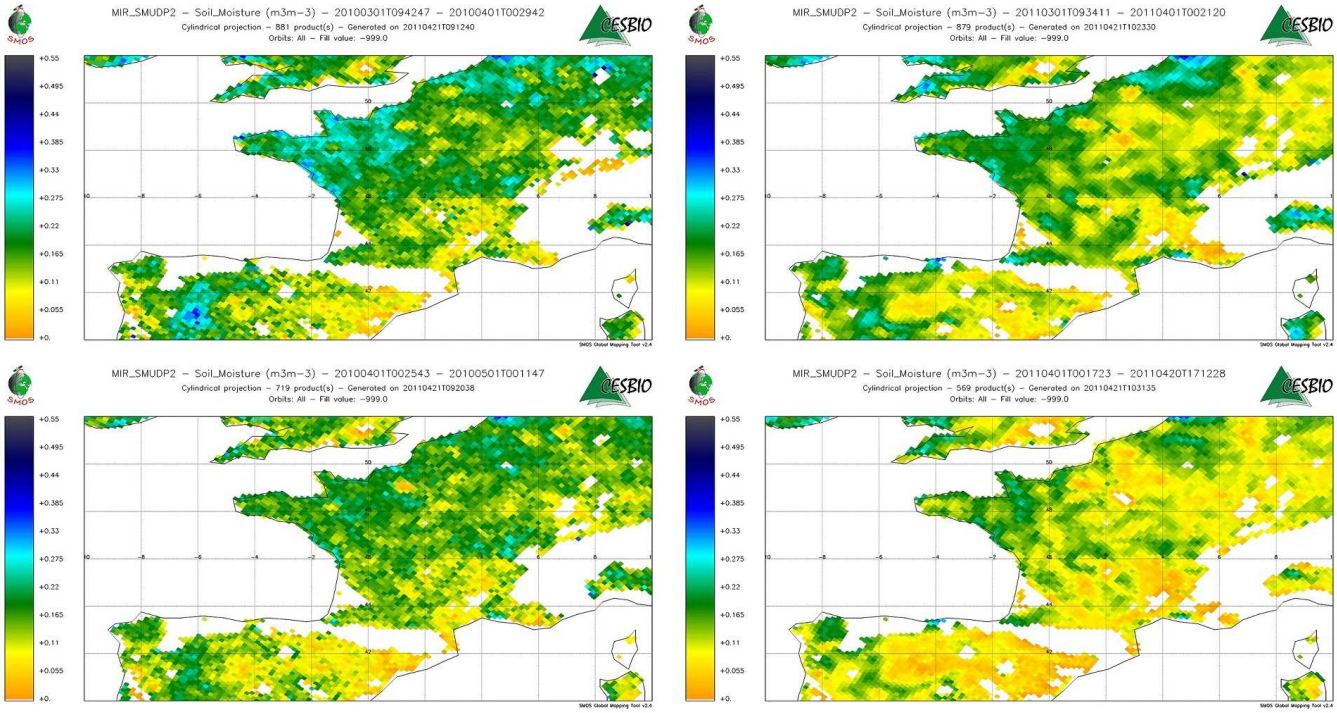
Finalmente, los datos son entregados a un grupo de suscriptores (son usuarios que han requerido y han sido autorizados a recibir directamente los datos) y a un archivo de misión que se encarga de gestionar cualquier otra petición que se recibiera.

Los datos de SMOS son gratuitos y abiertos a cualquier persona interesada en su análisis o explotación. El portal de observación de la tierra (http://earth.esa.int) gestiona las peticiones o acceso de los datos.

Procesados a niveles superiores (nivel 3, donde los datos de SMOS se analizan y se yuxtaponen temporalmente, o nivel 4 donde se complementan con datos de otros sensores espaciales) son realizados por los usuarios finales, pero también por los centros nacionales español (CP34) y francés (CATDS).



Procesado de señal (Cortesía ESA)



Comparativa de humedad en Europa 2010/2011 (Cortesía CESBIO)

Como se comentó, un requisito paralelo al científico solicita un componente de datos en tiempo cuasi real. Para ello, partiendo de los datos en nivel 0 se realiza un procesado rápido de nivel 1 (diferente al de la cadena nominal), y esos datos son enviados a un servicio de distribución para usuarios típicamente meteorológicos que utilizan las medidas de SMOS para mejorar las previsiones meteorológicas a corto y medio plazo.

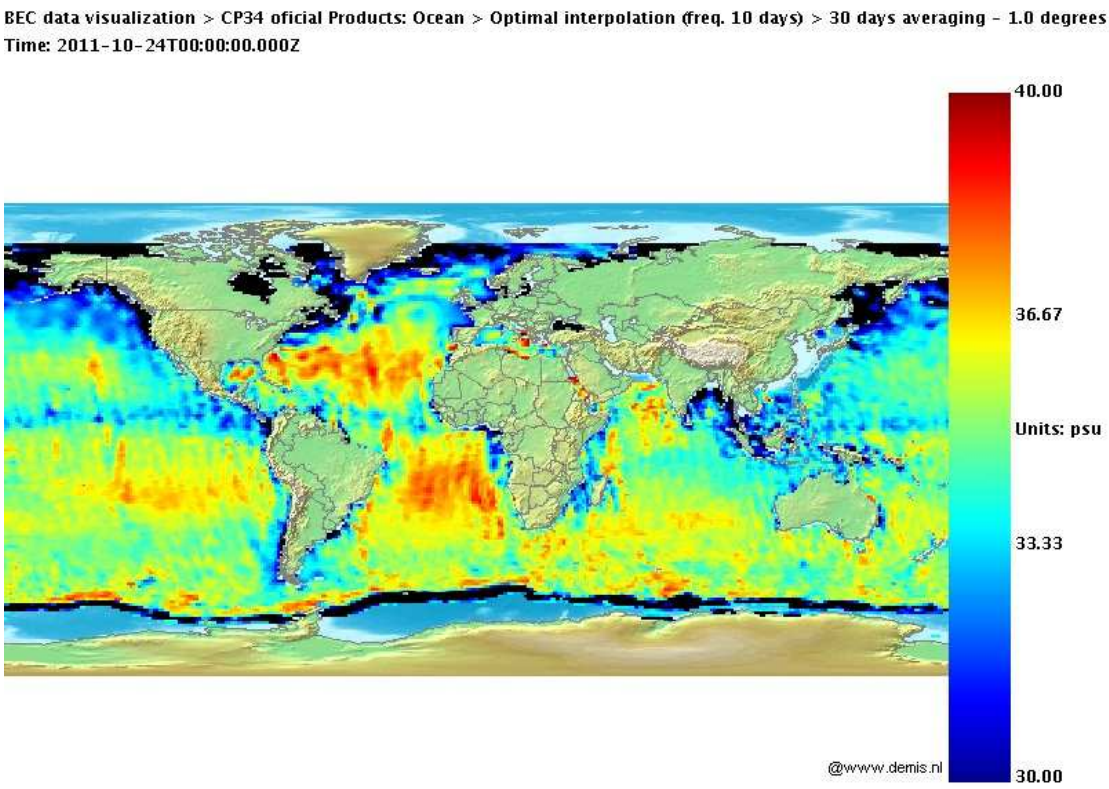
El procesado en tierra aquí descrito está continuamente siendo evaluado por un grupo de usuarios expertos, que proponen y sugieren mejoras. Estas se consolidan anualmente en una actualización de la cadena de procesado, y una campaña de reprocesado de todos los datos acumulados desde el comienzo de la misión se lleva a cabo en procesadores con alta capacidad (con velocidades que llegan a 12 veces la del procesador operacional), de forma que los datos se vuelven a consolidar con la última actualización.

RESULTADOS CIENTÍFICOS

Los objetivos científicos de SMOS, después de dos años de misión en vuelo, están convergiendo con los requisitos iniciales de la misión. Hay que destacar una vez más que los requisitos son ambiciosos, y que SMOS es un concepto absolutamente innovador, por lo que la consecución de los requisitos iniciales se espera tras varios años de mejoras en la comprensión del instrumento y campañas en tierra para respaldar sus datos.

Respecto a la humedad superficial, las mediciones de SMOS comparadas con campañas en tierra todavía muestran un sesgo inferior. No obstante, sitios de calibración como Dome-C en la Antártida o en zonas no contaminadas por interferencias, como en Australia, prueban que se puede obtener una buena correlación, mejorando sobre todo la comprensión de humedad a gran escala y su variabilidad temporal e incluso anual. SMOS se ha demostrado hasta la fecha como una herramienta esencial para la comprensión de la acumulación de agua a nivel mundial, analizando fenómenos como las inundaciones (cuando la saturación del suelo causa escorrentía más que asimilación), y al tiempo, la ingestión de productos de SMOS en los modelos de predicción meteorológica está mejorándolos.

SMOS ha sido la primera misión que ha podido realizar un mapa con una elevada precisión espacial y alta sensibilidad de la Salinidad oceánica. Fenómenos como los deshielos de casquetes polares y su contribución a las corrientes termo-halinas, o evaporación-precipitación que determinan eventos como el Niño o la Niña, o comprensión de los grandes balances salinos entre océanos (trasvase Atlántico-Pacífico) no habían podido ser monitorizados con antelación, y ahora se hacen visibles a la comunidad científica. Con ello, se está abriendo una puerta a múltiples formas de investigación antes no existentes. Además, en el área de salinidad, la misión Acquarius, lanzada en 2011, complementará y ampliará los datos de SMOS.



Mapa de Salinidad (Cortesía CESBIO)

RETOS FUTUROS DE SMOS

SMOS ha entrado ya en su tercer año de misión, lo que representa su objetivo inicial para poder demostrar la validez de su tecnología. No obstante, la extensión de la misión está siendo analizada para garantizar más años de datos de SMOS. Los principales retos que enfrenta SMOS de cara a los futuros años son:

Eliminación de interferencias (RFI's). En estos últimos dos años, se ha llevado a cabo un intenso trabajo por parte de la ESA y las autoridades nacionales de distintos países para identificar y apagar cuando ello es posible las fuentes de interferencia. Por otro lado, técnicas de filtrado de la señal han logrado mitigar parcialmente su impacto. A pesar de ello, un gran número de fuentes (y algunas nuevas) permanecen visibles, y queda todavía margen para mejorar las técnicas de filtrado. Como ejemplo se está probando una técnica de supresión de interferencias que permite mejorar sustancialmente la recuperación de salinidad en el Mar Mediterráneo o en el Mar Negro, contaminados por emisiones radio-eléctricas en la banda protegida.

Calibración. Las principales áreas para la mejora de la comprensión del funcionamiento de MIRAS pasan por refinar el conocimiento de la estabilidad del instrumento a corto y largo plazo, la mejora en la definición de un modelo de antena, avanzar en las correcciones de fuentes intensas como el sol o las RFI's.

Campañas en tierra. La caracterización de áreas clave en tierra y la correlación de sus datos con SMOS permitirán referenciar y calibrar mejor sus datos. El lanzamiento de la misión Aquarius (NASA-Argentina) en Junio del 2011 proporcionará también otras referencias (especialmente en salinidad) con las que SMOS podrá ser evaluado.

Uso del tiempo cuasi real. Los datos de SMOS en tiempo cuasi real están siendo usados en modelos de predicción meteorológica a corto y medio plazo, pero todavía a título experimental para evaluar el mejor algoritmo para incorporarlos. En los próximos años cabe esperar que estos modelos se refinen y se usen más ampliamente.

Procesado y reprocesado. Como se ha indicado, el procesado de los datos de SMOS no es un algoritmo estático y estable, sino abierto a continuas mejoras en gran medida derivadas de una mayor comunidad científica usando los datos. En particular, será necesario mejorar la conversión de la temperatura de brillo en productos de humedad y salinidad a través de mejoras en los procesadores de nivel 2. ■

REFERENCIAS

Página SMOS ESA, http://earth.esa.int/smosyhttp://www.esa.int/SMOS
Página SMOS CNES, http://smsc.cnes.fr/SMOS/
Página SMOS CESBIO, http://www.cesbio.ups-tlse.fr/SMOS_blog/
Página SMOS BEC-ICM, http://www.smos-bec.icm.csic.es/
Edición especial SMOS, Transactions on Geoscience and Remote Sensing (TGARS) 2008.
Publicaciones técnicas sobre SMOS hasta el 2009: http://esamultimedia.esa.int/docs/SMOS_publications.pdf
SMOS en el boletín de la ESA:
http://esamultimedia.esa.int/multimedia/publications/ESA-Bulletin-137/pageflip.html

Realidad Aumentada

Introdúcete en la Realidad Aumentada con ARToolkit

por Er ARturo Grandes

La realidad aumentada (augmented reality), se puede ver como la disciplina complementaria a la realidad virtual. Mientras en esta última se pretende crear un entorno lo más realista posible de forma sintética (en la memoria de un ordenador) la realidad aumentada busca *aumentar* la realidad con información adicional.

Las aplicaciones de realidad aumentada se han popularizado en los últimos años gracias a multitud de avances tecnológicos que han permitido llevar a la práctica ideas que llevaban flotando en las cabezas de los científicos e ingenieros durante varios años.

Curiosamente, los principales avances en este área se han centrado en intentar aumentar visualmente nuestra percepción de la realidad y, a día de hoy, el término *Realidad Aumentada* se identifica única y exclusivamente con este concepto.

Dicho esto, para el que escribe estas líneas, el concepto de *Realidad Aumentada* va mucho más allá y no puede más que considerar esta definición una metonimia en la que se ha tomado la parte por el todo. El concepto *Realidad Aumentada* no implica en sí mismo una dependencia directa con nuestro sentido de la vista. Literalmente sugiere aumentar la realidad o, más precisamente, la realidad como nosotros la percibimos. ¿Y cómo percibimos la realidad?, pues a través de nuestros sentidos.

La Realidad Aumentada nos permite añadir información adicional a la realidad que percibimos

La vista es sólo uno de nuestros cinco sentidos (sí cinco, aunque muchas veces nos olvidamos de que están ahí), y los seres humanos percibimos la realidad a través de nuestros sentidos. La *Realidad Aumentada*, en su sentido más general, puede ser vista como el conjunto de tecnologías que permiten *aumentar* nuestros sentidos, y por lo tanto la realidad que percibimos a través de ellos.

A lo largo de este texto nos referiremos a la *Realidad Aumentada* por su acrónimo en inglés, `AR`. Las razón fundamental es que así es como se refiere la mayoría del mundo a este tipo de tecnologías. La segunda razón es para evitar que el artículo se relacione con la mitología egipcia :).

HARDWARE PRET-A-PORTÉ

De igual forma que sucede con cualquier tecnología relacionada con el ser humano, el software no es suficiente y junto a este nos encontramos toda una serie de sistemas hardware necesarios para poder implementar la tecnología en cuestión. La *Realidad Aumentada* no es una excepción, y sin la existencia de ciertas soluciones hardware, el concepto en si mismo carece de sentido.

Wearable Computing es una de las tecnologías clave en un sistema de Realidad Aumentada

La principal tecnología hardware relacionada con la AR es lo que se denomina *Computación de llevar*. Sí bueno, la traducción es un poco *sui-generis*. El término técnico comúnmente utilizado es *wearable computing*, algo así como ordenadores para vestir, aunque esta última traducción, además de muy larga tampoco es que se ajuste al perfectamente al concepto en general.

De cualquier forma, la idea es muy sencilla. Si queremos extender nuestras capacidades humanas, deberíamos de poder llevar encima los dispositivos que utilicemos para ello de una forma lo menos intrusiva posible.

Esto se aplica a todos los elementos del sistema, no solo a los ordenadores, si bien, estos son los más problemáticos. De poco sirve poder extender nuestra percepción de la realidad si nuestro mundo se reduce a un laboratorio al que estamos encadenados a través de multitud de cables.

Así, esta necesidad de *llevar puesto* el hardware impone en general dos fuertes restricciones. Por una parte restricciones físicas; los cacharros tienen que ser lo más ligeros y pequeños posible. Por otra parte autonomía, lo que implica que tenemos que funcionar con baterías y que estas deben durar lo máximo posible.

Conseguir esto ha sido un gran problema durante mucho tiempo. En los últimos años, la proliferación de los teléfonos inteligentes ha proporcionado un impresionante empujón a esta línea de investigación.

Hoy en día disponemos de potentes *ordenadores de bolsillo* que podemos utilizar durante días y llevar en uno de los bolsillos de nuestra chaqueta.

Como suele ocurrir, el concepto de *Wearable Computing* ha llegado, pero de una forma un poco diferente a la que esperaban los pioneros de este campo...

EXTENSIÓN SENSORIAL

El ordenador en nuestro bolsillo es el que va a *aumentar la realidad*, pero para ello necesita poder percibirla de alguna forma. No sólo eso, el ordenador también necesita poder comunicar esa *realidad aumentada* al ser humano, a través de algún dispositivo que pueda comunicarse con ambos; el ordenador y la computadora.

Hasta que los interfaces neuronales se desarrollen lo suficiente, la interacción entre máquinas y seres humanos se tiene que realizar a nivel sensorial, es decir, a través de los sentidos.

Vista, Oído y Tacto son los sentidos en los que más se ha investigado hasta la fecha.

Utilizamos pantallas en las que los ordenadores pueden mostrar información que nosotros entendemos. Utilizamos altavoces para que los ordenadores puedan reproducir información que nosotros podamos oír. Utilizamos dispositivos hápticos para que los ordenadores puedan proporcionarnos información que nosotros podamos sentir. Utilizamos micrófonos para que los ordenadores puedan percibir nuestras palabras, etc...

Así que, necesitamos algunas cosas en torno a nuestro ordenador portátil para ser capaces de aumentar la realidad. Por ejemplo, un sistema AR visual necesita como mínimo una pantalla y algún sensor con el que averiguar donde se encuentra el usuario o algún elemento de interés. En el primer caso podríamos utilizar, por ejemplo, un GPS y unos acelerómetros. En el segundo caso una cámara de vídeo sería una opción válida. De la misma forma micrófonos, auriculares y tecnologías afines tales como, cancelación de ruido, reconocimiento y síntesis de voz son necesarias para poder aumentar nuestro sentido del oído.

DISPOSITIVOS HÁPTICOS. EL TACTO

El siguiente sentido en el que los investigadores han invertido muchos de sus esfuerzos es el tacto. Bajo el nombre genérico de dispositivos hápticos, se agrupan dos tipos fundamentales de tecnología. Los sistemas capaces de producir vibraciones y los sistemas capaces de producir realimentación de fuerza. Estos dispositivos se han desarrollado sobre todo en el ámbito de la realidad virtual, pero su aplicación a la realidad aumentada no tardara en hacerse patente.

La gama de dispositivos hápticos disponibles es bastante amplia. Desde un sencillo (o no tanto, según se mire) guante capaz de producir pequeñas vibraciones en la yema de los dedos para proporcionar la sensación de que estamos agarrando algo con una determinada textura, hasta complejos exoesqueletos capaces de aumentar nuestras capacidades físicas de forma artificial.

Como hemos dicho, estos sistemas son más comunes en el ámbito de la realidad virtual, donde no existe nada real y ser capaces de transmitir la sensación de tocar algo incrementa el realismo de la simulación enormemente.

GUSTO, OLFATO, SENTIDO COMÚN

Y hasta aquí hemos llegado. El olfato y el gusto son los grandes discriminados de la investigación sensorial. Existen investigaciones y resultados muy prometedores, sin embargo, a día de hoy, distan mucho de ser portables o de poder funcionar con una pequeña batería.

Finalmente, está el más importante de todos los sentidos. El sentido común. Sí, no es tontería, al final, las cosas tienen que tener sentido y muchas veces, por "güay" que parezca alguna solución puede que no resulte práctica en absoluto o que una obsesión por mimetizar el comportamiento humano nos lleve a soluciones poco viables, sobre todo cuando se intentan abordar antes de que llegue su momento.

Ojo, que no estamos hablando de investigación, sino de soluciones reales, que puedan ser utilizadas en el mundo real. Ahí el sentido común es fundamental.

APLICACIONES AR VISUALES

Bueno, tras esta breve introducción y reivindicación del espíritu más general de la *Realidad Aumentada*, vamos a empezar a centrarnos para poder empezar a crear nuestras aplicaciones de realidad aumentada "mú rápido".

En la actualidad, el término Realidad Aumentada se utiliza exclusivamente en su vertiente visual.

Como dijimos más arriba, el término *Realidad Aumentada* ha visto acotado su significado y hoy en día cuando escuchemos a alguien hablar de *Realidad Aumentada* se estará refiriendo a sistemas AR visuales. Estos sistemas son los más sencillos de entender, estamos viendo el resultado directamente, y los que más se han desarrollado en los últimos tiempos. Por esa última razón es relativamente sencillo desarrollar una aplicación AR muy fácilmente, utilizando herramientas disponibles en la actualidad.

Antes de meternos en harina un comentario más. Existen básicamente dos tipos de aplicaciones AR visuales.

A saber, las que ofrecen información al usuario en función de su posición y orientación, y las que reconocen algún patrón o forma y proporcionan información asociada al mismo. Las primeras se han hecho muy populares en los nuevos teléfonos inteligentes. Estas *pequeñas bestias*, además de un ordenador portatil, nos proporcionan un GPS, acelerómetros, cámara y una pantalla. Todo lo que necesitamos con su batería y con un formato portable.

Las segundas también empiezan a estar disponibles en estos dispositivos pero han tenido su origen en los ordenadores de sobremesa y los laboratorios, ya que la posición del usuario no es tan crítica. En este caso, solamente es necesario disponer de una cámara. El ordenador captura las imágenes de la cámara, las procesa en busca de un determinado patrón (este proceso puede ser bastante complejo... ya veremos más adelante), calcula la posición y orientación de dicho patrón con respecto a la cámara, y utiliza esa información para mostrar algún tipo de información asociado al patrón.

Si asociamos la posición de la cámara a la posición del usuario, es decir, nos pegamos la cámara a la cabeza, estaremos calculando la posición y orientación del usuario, de la misma forma que lo hacemos con el GPS, pero con unas restricciones diferentes.

Seguro que habéis visto este tipo de aplicaciones en más de una ocasión, pero sino, no os preocupéis, ya que estas son las aplicaciones de las que os vamos a hablar en este artículo.

ARTOOLKIT

Para desarrollar nuestra aplicación de *Realidad Aumentada mú rápido* vamos a utilizar la librería **ARToolkit**.

ARToolkit fue inicialmente desarrollado por los *Laboratorios de Tecnología Interacción Humana* en la Universidad de Washington (HIT Lab) y de la Universidad de Nueva Zelanda (HIT Lab NZ).

Inicialmente se distribuyó como software libre con licencia GPL, pero a partir de la version 2.72.1, liberada en 2007, el proyecto libre se cerró y se creó una compañía para continuar con el desarrollo del sistema desde una óptica más comercial. Así que, la versión con la que vamos a trabajar nosotros es precisamente esa, la última versión libre de la librería.

De cualquier forma, esta librería marcó un hito en el mundo de la *Realidad Aumentada* y sirvió de punto de partida para otras muchos clones que han aparecido durante estos últimos años. Hablaremos de ellos un poco más tarde.

INSTALANDO ARTOOLKIT

Lo primero que necesitamos hacer es instalar ARToolkit. Para ello descargamos las fuentes desde su página web. ARToolkit nos permite utilizar modelos VRML en nuestra aplicación a través de la librería *OpenVRML*. El problema es que la última versión libre de ARToolkit está preparada para utilizar OpenVRML versión 0.16.

En el momento de escribir este artículo, la última versión disponible de OpenVRML es la 0.18.X. El problema es que la version 0.16 no compila en las distribuciones modernas, y la versión 0.18 no funciona con ARToolkit. No desesperemos, lo bueno del software libre es que siempre hay opciones.

Tenemos dos posibilidades. O modificamos ARToolkit para que funcione con la última versión de OpenVRML, o hacemos que la versión 0.16 de VRML compile en nuestro sistema. Nosotros hemos optado por esta última opción y generado un simple *patch* para recompilar la versión 0.16. Podéis encontrar el parche en el código fuente que acompaña a este artículo, disponible desde nuestra página web.

ARToolkit nos permitirá iniciarnos en la AR Mú Rápido

Como algunos os habréis dado ya cuenta, existe una tercera opción. Simplemente no utilizar VRML. Afortunadamente el soporte VRML es opcional en la librería y podemos obviarlo si no lo necesitamos en nuestra aplicación. La aplicación que vamos a desarrollar en este artículo realmente no necesita OpenVRML, pero en el artículo queremos contaros como utilizar todo el potencial de la librería.

Una vez dicho esto, vamos a compilar ARToolkit.

CONFIGURANDO LA LIBRERÍA

ARToolkit proporciona algunas opciones que debemos seleccionar antes de compilarla. Esta configuración la realiza el script `Configure`, el cual nos hará una serie de preguntas sobre nuestras preferencias.

Lo primero que tendremos que elegir es el driver de captura de vídeo. El script nos proporciona cinco opciones:

```
Select a video capture driver.
  1: Video4Linux
  2: Video4Linux+JPEG Decompression (EyeToy)
  3: Digital Video Camcoder through IEEE 1394 (DV Format)
  4: Digital Video Camera through IEEE 1394 (VGA NONCOMPRESSED
     Image Format)
  5: GStreamer Media Framework
Enter :
```

Nuestra elección dependerá de la cámara que tengamos disponible. En caso de duda probad con `GStreamer` ya que esta opción proporciona distintas posibilidades para adquirir video. Por ejemplo, si vuestra cámara utiliza v4l2, como los dispositivos uvc bastante comunes estos días, la primera opción no funcionará, pero `Gstreamer` sí.

En el peor de los casos, `Gstreamer` nos permitirá utilizar como fuente un vídeo pregrabado almacenado en un fichero, que no es que sea lo mejor, pero es algo con lo que trabajar.

Tras la configuración del sistema de captura de vídeo, el script nos hará dos preguntas más:

```
Do you want to create debug symbols? (y or n)
Enter : n
Build gsub libraries with texture rectangle support? (y or n)
GL_NV_texture_rectangle is supported on most NVidia graphics cards
and on ATi Radeon and better graphics cards
Enter : y
```

PROBANDO... PROBANDO... 1,2,3

A la primera pregunta nosotros hemos respondido "no". A no ser que pretendáis depurar ARToolkit, esta opción no debería ser necesaria. La segunda dependerá de la tarjeta gráfica que utilicemos. Como nos informa el script, para tarjetas nVidia y ATI podemos responder "yes".

En nuestro caso estamos utilizando una tarjeta Intel y hemos respondido "y". De hecho, con nuestra configuración `GStreamer` y tarjeta Intel, responder "no" ha esta pregunta hace que la librería se estrelle al probar los ejemplos.

ARToolkit nos ofrece distintas opciones para la captura de vídeo, durante su configuración

Para saber si la extensión está disponible en vuestro sistema, podéis ejecutar el siguiente comando:

```
$ glxinfo | grep texture_rectangle
```

Ya solo falta hacer "make".

Ahora estamos en condiciones de probar el resultado de nuestro trabajo. En el directorio `bin` encontraremos diferentes programas de ejemplo que nos van a permitir comprobar si todo está en orden.

El primero que probaremos es `videoTest`. Este programa simplemente muestra el vídeo capturado por la cámara. En principio debería funcionar sin más, pero si tenemos varias cámaras instaladas o estamos utilizando uno de los drivers que necesitan configuraciones especiales, como es el caso de `GStreamer`, necesitamos decirle a ARToolkit como queremos que capture las imágenes.

Para ello utilizaremos la variable de entorno `ARTOOLKIT_CONFIG`. En la documentación de ARToolkit podéis encontrar información detallada de las opciones disponibles para los diferentes *drivers*. Como estamos utilizando `GStreamer`, nuestros parámetros serían estos:

```
$ export ARTOOLKIT_CONFIG="v4l2src device=/dev/video1 ! \
        video/x-raw-yuv,width=640,height=480 ! \
        ffmpegcolorspace ! \
        capsfilter caps=video/x-raw-rgb,bpp=24 ! \
        identity name=artoolkit ! fakesink"
```

¿Qué significa todo eso?. Bien, de hecho ese rollo de ahí arriba es una configuración de *pipes* de `GSTreamer`.

La primera parte de la configuración es la fuente de video, en nuestro caso estamos utilizando una cámara con driver V4L2. Además estamos utilizando la segunda cámara del sistema (`/dev/video1`). Nuestro ordenador de test es un portátil con una webcam integrada. Lamentablemente la cámara integrada es muy mala, y estamos utilizando una segunda cámara USB para estas pruebas.

La siguiente línea configura el formato de la fuente de vídeo, vamos, de nuestra cámara. En este caso estamos utilizando un formato YUV para los datos y queremos una resolución VGA estándar.

A continuación pasamos los datos de la cámara por el filtro `ffmpegcolorspace` que nos permite transformar los datos en formato YUV en datos RGB, que son los que le gustan a OpenGL. Para forzar la conversión al formato que necesitamos utilizamos un elemento `capsfilter` a continuación con el que limitar el formato de los datos..

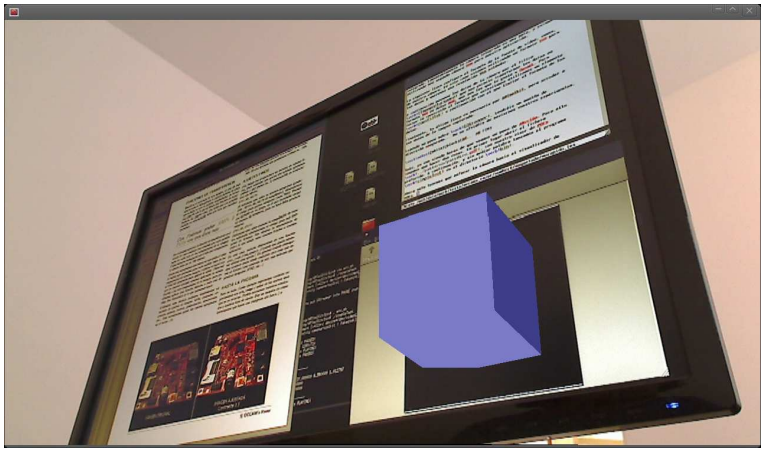
Finalmente, la última línea es necesaria por ARToolkit, para acceder a los datos de la imagen capturada.

Leyendo un poco sobre `GStreamer`, tendréis un montón de diversión asegurada... No os olvidéis de enviarnos vuestros experimentos.

AR... AR FIN

Vale, ya van siendo horas de que veamos un poco de ARcción. Para ello tenemos que hacer dos cosas. En primer lugar abrir el fichero `../patterns/pattHiro.pdf`, con nuestro visor de PDFs preferido. A continuación, y sin mediar palabra lanzamos el programa `simpleTest` en el directorio `bin`.

Ahora solo tenemos que enfocar la cámara hacia el visualizador de PDFs, y un bonito cubo azul aparecerá sobre el PDF. Como habréis comprobado el patrón aparece también en esta página, así que si estáis leyendo el artículo mientras hacéis estas pruebas podéis saltaros el primer paso.



Aplicación de Ejemplo SimpleTest Funcionando!

Ahora que todo está funcionando perfectamente, si habéis recompilado OpenVRML en el directorio `bin` podréis encontrar una aplicación `simpleVRML` que permite asociar ficheros VRML a diferentes patrones.

En el fichero `bin/Data/object_data_vrml` encontraréis el mapping entre patrones y modelos VRML utilizado por `simpleVRML`. Añadir vuestros propios modelos es una cuestión de editar una línea de texto :)

NUESTRA PRIMERA APLICACIÓN AR

Los ejemplos que vienen con ARToolkit, ya nos permiten hacer algunas cosillas, pero para desarrollar nuestra propia aplicación AR, necesitamos tener más control. Afortunadamente para nosotros, la librería es muy fácil de usar como veremos a continuación.

Vamos a comenzar con una versión simplificada de `simpleTest` que nos va a permitir introducir los principales elementos de este tipo de aplicaciones y las funciones que proporciona ARToolkit para facilitarnos la vida.

Comencemos con la función `main` la primera que se ejecuta.

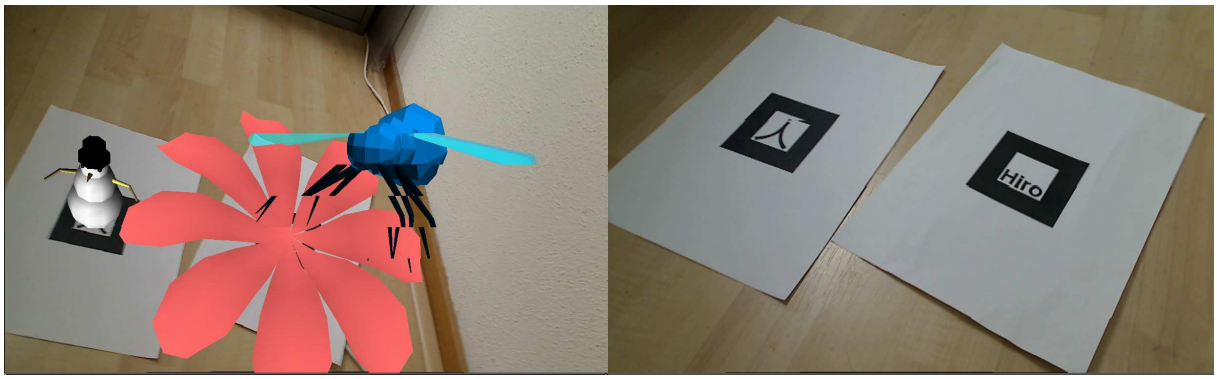
```
int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  init();

  arVideoCapStart();
  argMainLoop( NULL, NULL, mainLoop );

  return (0);
}
```

Lo primero que vemos es que nuestro ejemplo, al igual que todos los otros ejemplos distribuidos con ARToolkit utiliza GLUT para sus gráficos 3D. Como veremos en seguida, es muy sencillo utilizar cualquier otro método para inicializar OpenGL, glx, SDL, lo que más rabia os dé.

A continuación llamamos a la función `init` encargada de inicializar ARToolkit. En seguida veremos que es lo que hace. Tras ello inicia la captura de vídeo con la cámara que hayamos configurado e inicial el bucle principal de la aplicación en el que ejecutaremos la función `mainLoop` por cada paso del bucle.

INICIALIZANDO ARTOOLKIT

Como os adelantábamos unas líneas más arriba, la función `init` es la encargada de inicializar ARToolkit. ¿Y en qué consiste esa inicialización?. Veámoslo utilizando el código de la función.



Izquierda: simpleVRML — Derecha: Patrones Hiro y Kanji impresos

La función de inicialización necesita algunos parámetros. En nuestro sencillo ARminimo, los hemos declarado como variables globales que os mostramos a continuación.

```
char     *vconf = "";

int      xsize, ysize;
char     *cparam_name = DDIR "/camera_para.dat";
ARParam  cparam;
char     *patt_name = DDIR "/patt.hiro";
int      patt_id;
```

Lo primero que nos encontramos es la cadena `vconf` que utilizaremos para la configuración del sistema de vídeo. De hecho, esta cadena va a estar vacía, lo que significa que ARToolkit debe utilizar el contenido de la variable `ARTOOLKIT_CONFIG` para configurar el sistema de captura de imágenes.

Las variables `xsize` e `ysize` las utilizaremos para obtener el tamaño de la imagen que estamos capturando. En un momento veremos como los utilizamos.

El siguiente parámetro es `cparam_name`. Este parámetro es el nombre de un fichero que contiene los parámetros de configuración de la cámara. El que estamos utilizando nosotros viene con ARToolkit, y en general debería funcionar. En caso contrarío sería necesario producir uno nuevo para nuestra cámara. La documentación de ARToolkit nos explica como hacerlo.

La constante `DDIR` la pasamos desde el `Makefile`, con la opción `-D`. Si tenéis vuestro fichero de cámara en un lugar bien conocido, podéis sustituir esta constante por ese directorio.

Los parámetros de cámara que leemos del fichero anterior los almacenaremos en la variable `cparam`.

Finalmente, necesitaremos dos variables más. La primera contiene el nombre del fichero que contiene el patrón que queremos utilizar en nuestra aplicación. En nuestro caso se trata del patrón Hiro. Una vez cargado en memoria ese patrón, nos referiremos a él utilizando un identificador entero que almacenamos en la variable `patt_id`.

Tras declarar todas estas variables vamos a ver como las utilizaremos para inicializar ARToolkit. El código de la función `init` se muestra a continuación:

```
static void init( void )
{
  ARParam  wparam;

  if (arVideoOpen (vconf ) < 0 ) exit(0);
  if (arVideoInqSize (&xsize, &ysize) < 0 )
      exit(0);

  if (arParamLoad(cparam_name, 1, &wparam) < 0 )
  {
    printf("Camera parameter load error !!\n");
    exit(0);
  }

  arParamChangeSize (&wparam, xsize, ysize,
                     &cparam );
  arInitCparam( &cparam );
  arParamDisp( &cparam );

  if( (patt_id=arLoadPatt(patt_name)) < 0 ) {
    printf("pattern load error !!\n");
    exit(0);
  }

  argInit( &cparam, 1.0, 0, 0, 0, 0 )
}
```

Lo primero que hacemos es inicializar el sistema de captura de vídeo, terminando la aplicación si se produce cualquier error. Como ya os comentamos, en nuestro caso `vconf` es una cadena vacía ya que nosotros estamos utilizando el método de la variable de entorno para esto.

A continuación, le preguntamos al sistema de vídeo cual es el tamaño de las imágenes que vamos a capturar. Estos valores los utilizaremos en un momento para terminar de inicializar la librería.

Es el momento de leer el fichero de configuración de cámara, lo que conseguimos con una rápida llamada a `arParamLoad` con la que, de paso, almacenamos estos parámetros en una variable local.

Para finalizar la inicialización, actualizamos nuestros parámetros leídos del fichero, con el tamaño de la imagen capturada, almacenando el resultado en la variable `cparam` que contendrá nuestra configuración final.

La variable `cparam` se utiliza entonces para inicializar ARToolkit para nuestra cámara de acuerdo al tamaño de las imágenes que vamos a capturar. A modo informativo, mostramos estos parámetros utilizando la función `arParamDisp`.

PATRONES

Todavía nos falta un paso más para inicializar el sistema. Cargar los patrones que queremos utilizar en nuestra aplicación. En nuestro ejemplo vamos a utilizar un solo patrón que cargaremos con la función `arLoadPatt`. Esta función nos devolverá un número con el que poder identificar este patrón en nuestro programa.

Si nuestra aplicación necesitara más patrones,. este es el momento de cargarlos.

Con esto, ya solo nos queda llamar a la función `argInit` para que ARToolkit quede completamente inicializada.El segundo parámetro de la función es un factor de escala. El valor 1.0 significa que la imagen que mostraremos en la pantalla de nuestro ordenador tendrá el mismo tamaño que la imagen capturada. Si le diéramos el valor 2.0, la imagen que mostraríamos en la pantalla sería el doble de la que estamos capturando.

El resto de parámetros probablemente no los necesitéis, pero si tenéis mucha curiosidad echadle un ojo al fichero `include/AR/gsub.h` en las fuentes de ARToolkit.

BUCLE PRINCIPAL

Ahora que todo está listo para funcionar, tenemos que hacerlo funcionar. Y esto lo conseguimos con la función `mainLoop` que introdujimos al principio de nuestro `ARminimo`.

Esta función se ejecuta por cada ciclo del bucle principal, típicamente por cada cuadro de la escena 3D que dibujamos en la pantalla. Aquí está nuestra `mainLoop`:

```
static void mainLoop(void)
{
  ARUint8         *i;
  ARMarkerInfo    *marker_info;
  int             marker_num;

  if ((i =(ARUint8*)arVideoGetImage()) == NULL)
  {
    arUtilSleep(2);
    return;
  }

  argDrawMode2D();
  argDispImage (i, 0,0 );

  if (arDetectMarker (i, thresh, &marker_info,
                      &marker_num) < 0 ) {
    cleanup();
    exit(0);
  }

  arVideoCapNext();

  procesa_markers (marker_info, marker_num);

  argSwapBuffers();
}
```

Pues bien, lo primero que hacemos es leer una imagen de la cámara, si es que hay alguna disponible. En caso contrario, esperamos un poquito e iniciamos otro ciclo.

Una vez que tenemos la imagen en nuestro vector `i`, la dibujamos inmediatamente en pantalla. Para ello, primero activamos el modo 2D utilizando `argDrawMode2D` y luego mostramos la imagen con `argDispImage`.

Lo siguiente que podemos hacer, una vez tenemos una imagen de la cámara, es detectar nuestros markers o patrones en la imagen, y esto sí que es muy sencillo, basta con llamar a la función `arDetectMarker`. La función recibe como parámetro la imagen que hemos capturado, seguido de un umbral de detección, un puntero a un vector de elementos del tipo `ARMarkerInfo`, y un entero en el que nos devolverá el número de markers detectados en la imagen.

En nuestro ejemplo, al igual que en los distintos ejemplos que podemos encontrar en le paquete ARToolkit, el umbral se fija normalmente a un valor de 100. Este parámetro permite ajustar el proceso de conversión de la imagen de color a monocromo (blanco o negro). Más o menos, los pixels por encima de este valor serán blancos y los que estén por debajo serán negros... simplificando el proceso mucho. Los más curiosos podéis echarle un ojo a este artículo.

En este punto ya hemos terminado con la imagen y podemos olvidarnos de ella. Es el momento de decirle al sistema de video que empiece a capturar la siguiente imagen mientras nosotros terminamos algunas cosillas.

Esas cosillas las vamos a hacer en la función `procesa_markers`, tras la cual actualizaremos la pantalla con una llamada a `argSwapBuffer`.

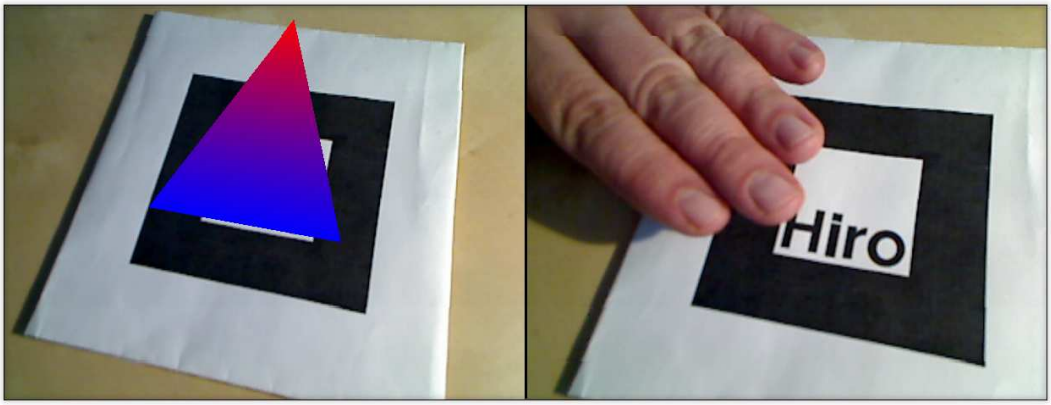
BOTONES

Ya solo nos quedan dos funciones por comentar. Una que analiza los patrones que hemos encontrado en la imagen, y otra que pinta algo en la pantalla basándose en los resultados de las etapas anteriores.

Para ilustrar estos dos pasos, vamos a comenzar con el ejemplo más sencillo posible: Un botón. Como se suele decir *y para muestra un botón* :).

Nuestro botón lo implementaremos tal que así. Supondremos que un patrón será normalmente visible, de forma que cuando no lo detectemos, supondremos que hay algo bloqueando la imagen y por tanto diremos que el botón ha sido pulsado.

Más sencillo no se puede, sin embargo, este simple ejemplo nos va ha permitir introducir todos los conceptos necesarios para que podáis desarrollar vuestras propias aplicaciones de realidad aumentada.



Izquierda: Patrón Visible (Botón Normal) — Derecha: Patrones Ocluído (Botón Pulsado)

PROCESANDO MARKERS

Es momento de añadir la lógica de nuestro botón. La función `procesa_markers` es la encargada de ello. Aquí la tenéis.

```
void
procesa_markers (ARMarkerInfo *mi, int num)
{
  int               j, k;

  /* check for object visibility */
  k = -1;
  for( j = 0; j < num; j++ ) {
    if( patt_id == mi[j].id ) {
      if( k == -1 ) k = j;
      else if( mi[k].cf < mi[j].cf ) k = j;
    }
  }

  if( k == -1 )  {
      if (boton) {
      printf ("Botton Pulsado\n");
      boton = 0;
    }
      return;
    }

  boton = 1;

  arGetTransMat(&mi[k], patt_center,
                        patt_width,
                        patt_trans);

  draw();
}
```

Si recordáis, los parámetros que recibe esta función son la lista de patrones detectados en la imagen capturada por la cámara así como su número. Así que lo que vamos a hacer es buscar entre los patrones detectados nuestro patrón de referencia.

Obtener la posición de los markers es muy fácilcon ARToolkit.

En este caso el bucle no es realmente necesario, ya que solo tenemos un patrón (hiro). Sin embargo lo hemos dejado por conveniencia para que os resulte más sencillo hacer experimentos con distintos patrones.

Tras el bucle, la variable `k` valdrá -1 si no se ha encontrado nuestro patrón (`patt_id`)o su índice en la tabla si el patrón aparece en la imagen.

Con esta información implementamos la lógica de nuestro botón. Si el patrón a desaparecido (k = -1) y hace nada estaba allí (boton = 1) entonces es que hemos pulsado el botón. Si el patrón aparece en la imagen, entonces el botón está disponible y actualizamos la variable botón.

Finalmente, le pedimos a ARToolkit que nos de la información asociada al patrón k (nuestro patrón de interés), tanto para representarla en 2D (centro y ancho) o en 3D (matriz de transformación). Estos valores los almacenamos variables globales por simplicidad.

Una vez que tenemos nuestra matriz, ya solo nos queda pintar algo en la posición del patrón que acabamos de localizar.

PINTANDO NUESTRO BOTÓN

Nuestro botón tomará la forma de un triángulo con uno de sus vértices coloreados. Exactamente, el triángulo que aparece en la figura de la página anterior.

Veamos como pintar esta figura con la información contenida en la variable `patt_trans`, que no es otra cosa que la posición y orientación de la cámara.

Esta es la función que dibuja el triángulo.

```
static void draw( void )
{
  double    gl_para[16];

  argDrawMode3D();
  argDraw3dCamera( 0, 0 );

  argConvGlpara(patt_trans, gl_para);
  glMatrixMode(GL_MODELVIEW);
  glLoadMatrixd( gl_para );

  glTranslatef( 0.0, 0.0, 25.0 );

  glBegin(GL_TRIANGLES);
  glColor3f(0,0,1);
  glVertex3f(-30,-30,0);

  glColor3f(0,0,1);
  glVertex3f(30,-30,0);

  glColor3f(1.0,0,0);
  glVertex3f(0,50,0);
  glEnd();
}
```

Lo realmente importante de esta función son las cinco primeras líneas. El resto de la función es código OpenGL estándar para dibujar un triángulo. De hecho podemos sustituir este triángulo por cualquier otra cosa; un modelo VRML utilizando OpenVRML, o figuras más complejas con luces, iluminación e incluso shaders.

Veamos en detalle la parte de ARToolkit. La primera función se encarga de activar el modo dibujo en 3D. Recordad que lo primero que hacíamos tras capturar la imagen era activar el modo 2D para dibujar esa imagen en el fondo. A efectos prácticos, lo que todo esto significa es que estamos cambiando de una proyección ortográfica para 2D a una proyección en perspectiva para 3D. Las funciones `argDrawMode3D` y `argDrawMode2D` realizan este cambia pero asegurándonos que cuando pintemos algo en 3D su posición corresponderá con el marker en la imagen 2D.

Ahora solo tenemos que transformar la matriz que nos ha devuelto ARToolkit al formato que espera OpenGL.

La matriz que nos devuelve ARToolkit es realmente la posición y orientación de la cámara así que si asignamos esta matriz a la matriz MODELVIEW de OpenGL, estaremos posicionando la cámara virtual en la posición correcta.

Eso es lo que hace la llamada a `glLoadMatrixd`.

Observad que hemos añadido una traslación en el eje z para que nuestro triángulo aparezca por encima del marker y justo en el mismo plano.

Un último comentario sobre la función de dibujo. Normalmente, y así lo veréis en la mayoría de los ejemplos que vienen con ARToolkit, justo después de activar el modo de dibujo en 3D, tendréis que restaurar el estado OpenGL que necesitéis para vuestra aplicación. Cosas como activar el z-buffer o la iluminación, asignar materiales, etc.

Nosotros hemos eliminado todo eso para mantener los listado pequeños, pero si pretendéis hacer algo más complicado que nuestro simple botón echadle un ojo a los ejemplos de ARToolkit y a esta página.

INTERFACES TANGIBLES

Como nuestra demo del botón es un poco escasa, vamos a introducir un par de conceptos más y *Aumentar* nuestra demo para que haga algo útil. Para ello vamos a introducir el concepto de *Interfaces Tangibles*

Los interfaces tangibles son un tipo especial de sistemas AR en los que la interacción con el mundo real y los objetos virtuales se restringe, generalmente, a una superficie en lugar de permitir una interacción libre en el espacio tridimensional. Esta restricción facilita la interacción con el sistema utilizando objetos reales.

Estos sistemas suelen utilizar un superficie multitoque (Multi-Touch Screen) sobre la que se manipulan objetos reales con algún tipo de identificación y comportamiento. Si bien ese es el más común de los escenarios, es posible construir este tipo de sistemas con nuestro modesto ARToolkit.

Para ilustrar lo que acabamos de comentar vamos a modificar nuestro simple botón para convertirlo en un interfaz tangible con el que modificar el volumen de nuestro ordenador :).

En los interfaces tangibles la interacción entre mundo real y virtual se restringe a las 2D.

En este sencillo ejemplo vamos a suponer que el sistema de audio de nuestro ordenador está controlado con *PulseAudio*. Simplemente necesitamos elegir un interfaz y este parece ser el más común últimamente.

La modificación del volumen la haremos utilizando la función estándar `system` de forma que modificar el programa para que funcione con otras configuraciones debería ser muy sencillo.

Así que todo lo que necesitamos saber es como decirle a *PulseAudio* que queremos modificar el volumen del sistema. Afortunadamente para nosotors (bueno, realmente lo hemos hecho a propósito :), *PulseAudio* incluye una conveniente herramienta de línea de comandos llamada `pacmd` que nos va a salvar el día. El comando que nos interesa es este:

```
pacmd set-sink-volume 0 100
```

La función system nos permite conectar con el control de volumen de PulseAudio.

Básicamente le estamos diciendo a pulse audio que ponga el volumen del *sink* 0 al valor 100.

CONTROL DE VOLUMEN TANGIBLE

Con todos estos elementos solo necesitamos modificar nuestra función `procesa_markers` añadiendo el código que se muestra a continuación justo antes de la llamada a la función `draw`.

```
a = atan2 (patt_trans[2][0],
           patt_trans[2][1]);

if (a < 0) a += (2 * M_PI);

int val = round ( (a) / (2 * M_PI) * 65535.0);

char c[1024];
snprintf (c, 1024, "pacmd set-sink-volume 0 %d",
                   val);
system (cmd);
```

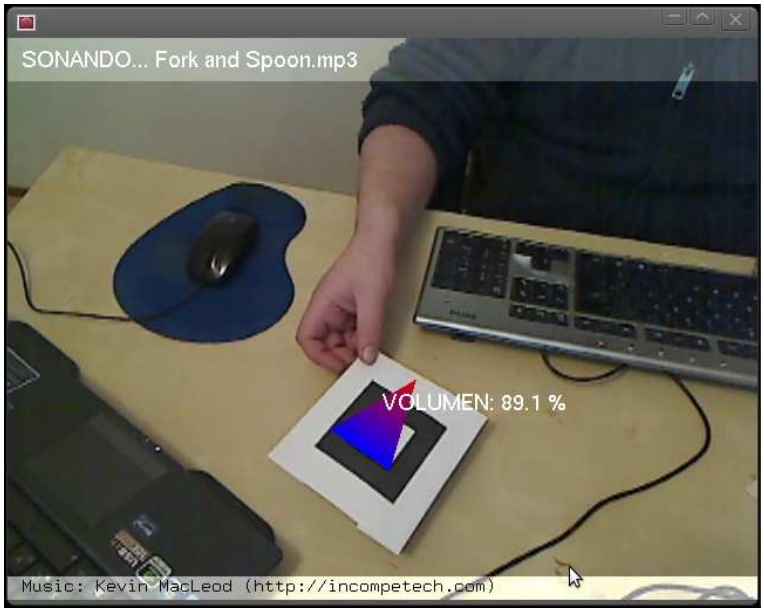
Lo que estamos haciendo es calcular la rotación del patrón con respecto al eje Z a partir de la matriz de transformación que nos devuelve ARToolkit. Luego convertimos el resultado en un valor lineal entre 0 y 65535, que es el rango de valores que espera `pacmd`. Una llamada a `system` y ya estamos listos

Solo nos falta un pequeño toque en la función `draw`. Suponiendo que el valor `a` que calculamos en el pedazo de código anterior es una variable global, podemos modificar el color del último vértice de nuestro triángulo para nos de una idea de como de alto está nuestro volumen.

```
glColor3f(fabs(a) / (2 * M_PI),0,0);
```

Con esto terminamos nuestra breve introducción a la realidad aumentada con ARToolkit. Hay varias funciones que no hemos comentado, pero no os queremos quitar toda la diversión. Las bases os las hemos contado, y la diferencia entre nuestro sencillo interfaz tangible y cualquier otra aplicación AR está en vuestras manos.

Por ejemplo, nosotros hemos ampliado el ejemplo que describimos en este artículo para improvisar un reproductor MP3 con controles *wireless* :), conectando nuestro botón con mplayer a través de su modo esclavo. Visitad nuestro canal de YouTube o los recursos al final del artículo para verlo en acción.



Resultado Final de nuestro interfaz tangible para mplayer

SIFT, SURF Y COMPAÑÍA

No podemos terminar este artículo sin daros algunas indicaciones sobre como funcionan los sistemas de realidad aumentada más actuales que ya no necesitan *markers* o patrones como los que hemos utilizado en este ejemplo.

Realmente no es que no utilicen patrones, sino que ahora, los patrones son imágenes arbitrarias y por lo tanto más fácil de integrar en nuestras aplicaciones.

En general todos esto sistemas se basan en algún algoritmo de extracción de rasgos de imagen (*image features* en inglés), algo así como descubrir los puntos interesantes de las imágenes y asignarles un vector de valores que luego puede ser utilizado para reconocer tanto la imagen original como su posición y orientación.

Existen varios de estos algoritmos, pero quizás el más popular, por sus buenos resultados es SIFT (*Scale Invariant Feature Transform* o Transformación de rasgos invariante con la escala). El algoritmo fue publicado por David Lowe allá por 1999, pero lo que realmente revolucionó el mundo de la realidad aumentada fue otro algoritmo, conocido como SURF (*Speeded Up Robust Feature* Rasgos Robustos Acelerados).

Conceptualmente ambos algoritmos son similares y ambos se basan en la extracción de *features* de la imagen, pero SURF fue diseñado para funcionar muy rápido y eso era fundamental en las aplicaciones de *Realidad Aumentada*.

Lo interesante de estos algoritmos es lo robustos que resultan frente a variaciones de escala, rotación e iluminación de las imágenes (dentro de unos márgenes), lo que es fundamental en aplicaciones que se no se utilizarán en entornos controlados.

Existen otras muchas técnicas y probablemente aparecerán más. Las páginas de Wikipedia sobre detección de rasgos describen buena parte de ellos para aquellos de vosotros que queráis profundizar en el tema.

RECAPITULACIÓN

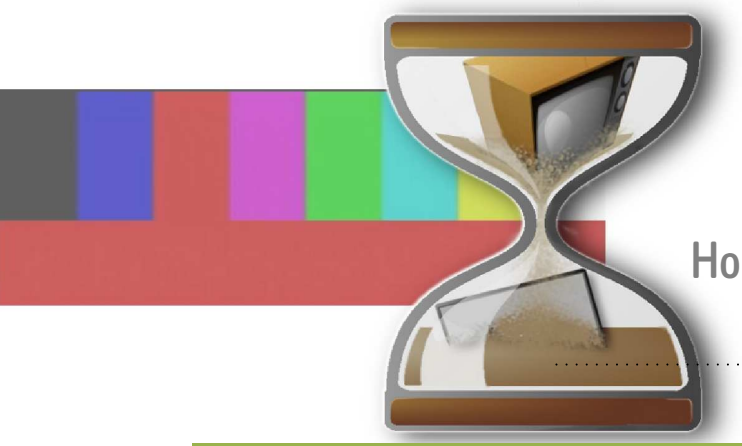
Aunque ARToolkit no es la librería de realidad aumentada más avanzada que nos podemos encontrar en 2012, sigue siendo un excelente punto de entrada en este apasionante mundo y un trampolín hacia soluciones más complejas.

Aún con sus limitaciones, ARToolkit permite crear muchas e interesantes aplicaciones AR y sobre todo, recordad que la tecnología no es lo más importante. Lo importante es la idea. ■

RECURSOS

Página Oficial de ARToolkit
http://www.hitl.washington.edu/artoolkit/
http://www.hitl.washington.edu/artoolkit/documentation/vision.htm
http://www.hitl.washington.edu/artoolkit/documentation/devstartup.htm

Páginas Wikipedia de Referencia
http://es.wikipedia.org/wiki/Realidad_aumentada
http://en.wikipedia.org/wiki/Brain-computer_interface
http://en.wikipedia.org/wiki/Scale-invariant_feature_transform
http://en.wikipedia.org/wiki/SURF



Historia de la Televisión

Homenaje Póstumo a la televisión analógica

por Fernando Martín Rodríguez

Cuando David (el director de esta revista) me preguntó: *"¿Qué hacemos esta vez para la sección de historia?"*, la verdad es que me sorprendió. Ya sabía que me había convertido en historiador tecnológico aficionado (que me perdonen los historiadores de verdad), pero: ¿Hemos llegado a consolidar una sección fija en la revista? Posiblemente, la respuesta es sí ya que en cuatro números publicados hay tres artículos de esta temática. La serie sobre biometría tuvo otros tres artículos, éste es el cuarto de la "sección de historia".

Y... ¿Sobre qué escribir? En aquel momento era inminente el "apagón analógico" del 3 de abril de 2010, por eso la decisión tomada fue ésta: escribir sobre la historia de la televisión (sí, a veces, los artículos tienen "un poco" de retardo). Tal vez, hasta podríamos poner un subtítulo: "Homenaje póstumo a la televisión analógica".

EL SUEÑO IMPOSIBLE

He "plagiado" el título de esta sección de un texto del profesor (y compañero) Xulio Fernández Hermida. Con él se refiere a que los primeros expertos en comunicaciones quisieron llegar más allá del telégrafo (primer sistema de telecomunicación de la historia) y ser capaces de "llevar la realidad que ocurre en un lugar a otro lugar alejado".

El teléfono (año 1876) ya supuso un avance en ese sentido, porque permite comunicarse a las personas de la forma más habitual para ellas (hablando). Por eso fue (y sigue siendo) el sistema de telecomunicación más utilizado de la historia.

Pero faltaba algo diferente. También había interés por ser capaz de transmitir ciertos contenidos: teatro, música, deporte, otros espectáculos... La radio nació en 1920 (aunque ya hubo experimentos desde 1906). Se había cumplido parte del sueño: un espectáculo producido en un teatro podía ser enviado a muchas personas en sus casas. ¿Qué faltaba? Todos lo sabemos: las personas recibimos la mayor parte de la información por dos sentidos: la vista y el oído. Se había conseguido "acercar los oídos de los usuarios al espectáculo" pero "faltaban los ojos".

El sueño se empezó a completar en 1897 cuando Carl Ferdinand Braun inventó el osciloscopio (1897) y con él el tubo de rayos catódicos. Este invento junto al descubrimiento de las propiedades electroópticas del selenio (su resistencia varía con la luz que recibe) permitió al ruso Vladimir Zworykin desarrollar el iconoscopio, el primer tubo de cámara práctico (año 1923). El tubo de cámara fue la primera tecnología de captura de imagen en movimiento. Se basaba en un tubo de rayos catódicos y una resistencia variable con la luz. No fue sustituido hasta la década de los noventa, cuando aparecieron los sensores de imagen de estado sólido (CCD's).

La tecnología de los tubos de rayos catódicos también se empleó en receptores de televisión y monitores de ordenador (tubos de imagen) hasta finales de los noventa, en esa época la tecnología de cristales líquidos LCD alcanzó la calidad suficiente para ser una opción viable. También en ese momento aparecieron los primeros monitores de plasma.

El considerado por todos como padre de la televisión es John Logie Baird que en 1927 logró transmitir una señal a una distancia de 438 millas entre Londres y Glasgow, para ello introdujo una señal analógica en un cable telefónico interurbano.

En 1937 estaba en el mercado en Inglaterra un receptor de TV de 405 líneas. La empresa que lo fabricaba era la Marconi-EMI (fundada por el conocido inventor Guillermo Marconi). A partir de aquel momento el sueño se había cumplido, siempre parcialmente, porque ver un espectáculo por televisión nunca será lo mismo que estar allí. Por supuesto, este nuevo servicio experimentó muchas mejoras y cambios y eso será lo que comentaremos en el resto del artículo.

LA SEÑAL DE TV BLANCO Y NEGRO

La señal original de televisión dependía mucho del sistema concebido para generarla: el tubo de cámara. Un tubo de cámara, como vimos en el apartado anterior, es un caso particular de tubo de rayos catódicos. Esto es: una cámara de vacío donde una gran diferencia de potencial provoca una descarga: un haz o flujo de electrones a gran velocidad (esta "corriente" circulando por el vacío es lo que recibe el nombre de "rayo catódico"). En los tubos de televisión (tubo de cámara y tubo de imagen) se utilizaban bobinas (bobinas de deflexión) para crear campos magnéticos capaces de dirigir el haz al punto deseado (nótese que existen electrodos de "enfoque" que consiguen que el haz sea estrecho y directivo). En el tubo de imagen se obliga al haz a recorrer toda la superficie disponible por líneas.

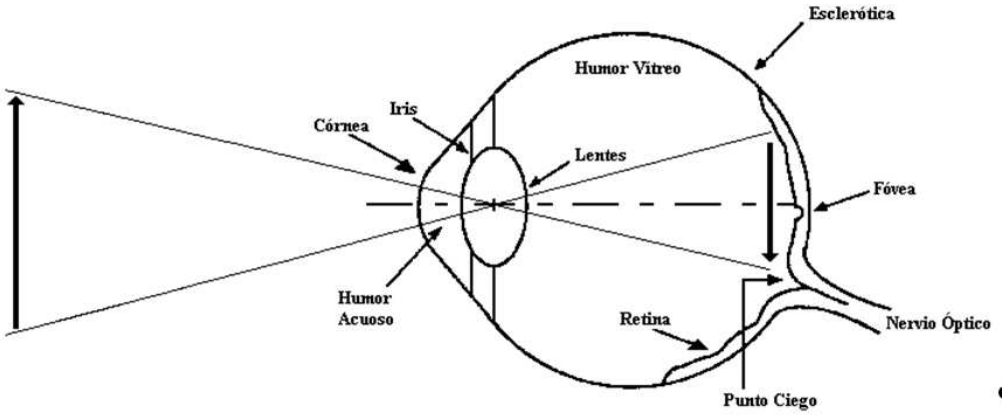


Figura 2 - Funcionamiento del ojo.

Con eso se pretende obtener el brillo de cada punto de la imagen. Si os fijáis en el esquema de la figura 1, el circuito se "cierra" atravesando un conductor "fotorresistivo" (hecho de selenio, material cuya resistencia cambia con la luz incidente). El tubo de cámara funcionaba así: en cada instante, el haz llega a un punto y "cierra" el circuito a través de ese punto (y la resistencia depende del brillo que llega a ese punto). Al igual que el ojo humano y que las cámaras de fotografía, un sistema óptico se encarga de formar una imagen sobre una superficie fotosensible (la capa de selenio).

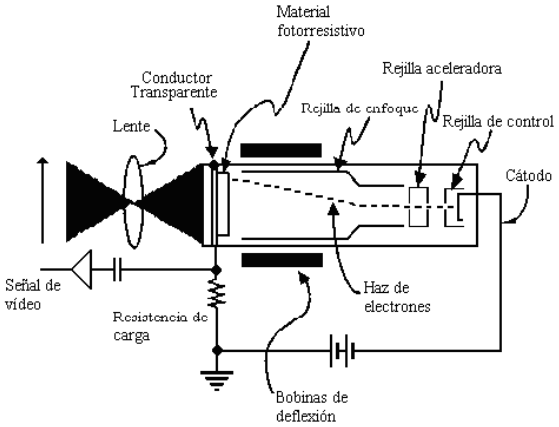


Figura 1 - Esquema del tubo de cámara.

Dado su origen tenemos una señal "barrida por líneas", es decir: la señal va a ser una forma de onda analógica proporcional al brillo de la imagen y que la recorre por líneas. Nótese que hablamos de brillo (sensación de luminosidad de la luz, que aumenta con la energía transportada por la misma) pero no de color, en cada punto tendremos un color gris del brillo adecuado; el brillo mínimo es el color negro y el máximo es el blanco.

En la figura 3, podemos ver que los tramos correspondientes a las diferentes líneas van separados por mínimos de tensión. Esos pulsos reciben el nombre de sincronismos horizontales y ocupaban el 25 % del margen dinámico (margen dinámico de una señal normalizada: 1 V). Nótese que se eligió un valor muy grande para proteger los sincronismos (en un tiempo en que la electrónica no era muy avanzada). Durante el tiempo que duran estos sincronismos (tiempo no activo de una línea) el haz del tubo receptor vuelve a su posición horizontal inicial (en esos momentos el haz de electrones está cortado, este tiempo también se llama tiempo de borrado o tiempo de retorno horizontal).

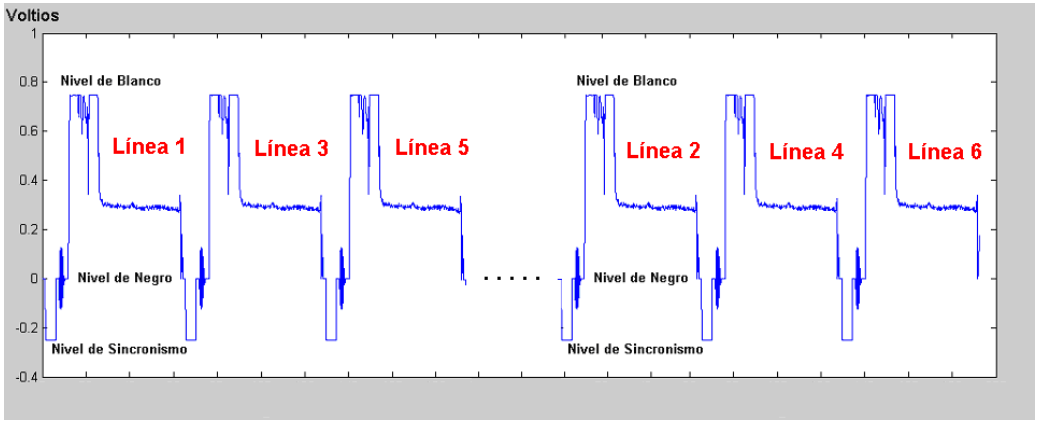


Figura 3 - Forma de onda de la señal de blanco y negro.

Durante el resto del tiempo (tiempo activo) la tensión indica el brillo o luminancia de la imagen. Los valores grandes de tensión provocarán un haz de electrones grande y un brillo grande (cercano al blanco), los valores pequeños de tensión provocarán haces débiles y puntos oscuros (cercanos al negro).

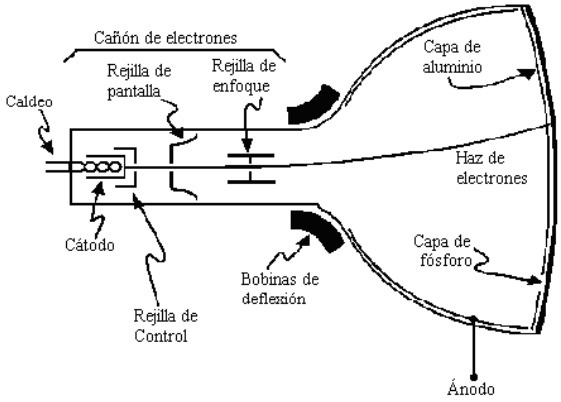


Figura 4 - Tubo de imagen (monitor blanco y negro con TRC). El haz de electrones cierra un circuito con la capa de aluminio (ánodo) y la energía del impacto es absorbida por el fósforo que emite luz de intensidad proporcional a dicha energía.

Además de los sincronismos de línea (u horizontales) es necesario "señalizar" donde empieza y acaba cada imagen. Para eso se han introducido una serie de líneas que no llevan información de imagen (líneas no activas). Varias de esas líneas tienen perfiles de tensión específicos que le sirven al televisor para identificarlas y así saber que ha acabado una imagen y va a empezar la siguiente. A estas líneas se les llama: líneas de sincronismo vertical.

Las líneas de sincronismo vertical son líneas no activas porque no tienen imagen (aunque son necesarias para la sincronización de la señal). Además de estas líneas no activas incluyeron muchas más: en la TV europea, de un total de 625 líneas por imagen se tienen 576 activas y 49 no activas. El sincronismo vertical ocupa 18 líneas. El exceso de líneas no activas (49-18 = 31 líneas) se pensó para dejar espacio para usos futuros. En las primeras emisiones esas líneas estaban vacías (la señal era nula). Después, tuvieron varios usos: teletexto (señal digital), señales para el control de calidad (por ejemplo, se pueden introducir sinusoides de diferentes frecuencias y se ve en qué estado se reciben, con lo que se tiene una idea de la respuesta en frecuencia de un canal) e incluso señales para controlar acceso a servicios de pago (sistemas de acceso condicional analógico como nagravision utilizado por Canal+ analógico).

Por último, en la figura 3 podemos ver que las líneas aparecen numeradas de una forma "extraña" (primero las impares y después las pares). Eso se debe a que la tasa temporal de imágenes entonces era relativamente baja: 25 imágenes por segundo en Europa y 30 imágenes por segundo en Estados Unidos. Este número proporciona sensación de movimiento pero es insuficiente para evitar el "flicker" o parpadeo de la imagen en pantalla. Para evitar este problema se recurrió al "barrido entrelazado": las líneas no van ordenadas de manera natural (1,2,3,4...) sino divididas en dos subcuadros o campos: el primero formado por las impares (1,3,5...) y el segundo por las pares (2,4,6...). Como consecuencia del entrelazado cada imagen supone dos recorridos verticales de pantalla. Como consecuencia, el recorrido del haz de un tubo de cámara también era entrelazado al igual que el de un tubo de imagen.

La tasa de imágenes de la televisión se eligió igual a la mitad de la frecuencia de la señal de corriente alterna (la corriente alterna oscila con 50 Hz en Europa y 60 Hz en Estados Unidos). Esta elección fue necesaria para eliminar ciertas interferencias debidas a la red eléctrica. Además de eso, en Europa se eligió un sistema con 625 líneas por imagen (576 activas) y en Estados Unidos el sistema se diseñó con 525 líneas (480 activas).

LA TV COLOR: NTSC, PAL Y SECAM

Entre los años 40 y 50 la televisión en blanco y negro llegó a casi todo el mundo. Por ejemplo, en España se empezó a emitir en 1956 (aunque se venían realizando pruebas desde 1952). El deseo de incorporar el color era natural entonces, y más con la experiencia del cine, que había logrado incorporar la tecnología del color en 1935.

En EEUU fue la cadena de TV CBS la que inició la investigación fundando el comité NTSC (Nacional Televisión System Committee). Dicho comité data de 1930 y pretendía encontrar una "extensión" a la señal blanco y negro que permitiera emitir en color pero que, además, la señal fuese compatible con los televisores blanco y negro existentes (esto es: que la señal se pudiese visualizar, en blanco y negro, pero correctamente sin "interferencias del color"). El sistema NTSC no fue aprobado por la FCC (Comisión Federal de Comunicaciones) hasta la década de los 50. El sistema NTSC se adoptó en Estados Unidos, gran parte de Latinoamérica y algunos países de extremo oriente: Japón, Filipinas...

En la historia de la TV en color, debemos mencionar los trabajos realizados por el ingeniero mexicano Guillermo González Camarena que patentó un sistema de color en 1940. Posteriormente, colaboró con investigadores norteamericanos en 1950. Fruto de su trabajo se desarrolló el sistema "bicolor simplificado" con el que se emitió TV en color en México entre 1962 y 1968. González Camarena murió en accidente de tráfico en 1965. Su falta hizo que el sistema bicolor dejara de desarrollarse y que México decidiera pasarse al sistema NTSC (había además una gran presión para adoptar un sistema "compatible con el resto del mundo" ante el gran evento de la olimpiada de Ciudad de México, 1968).

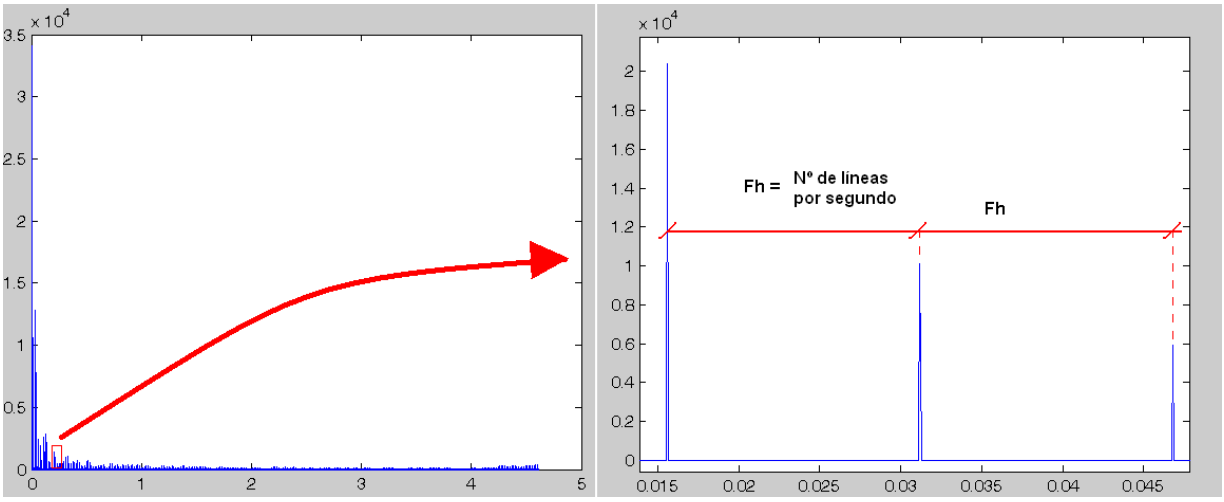


Figura 5 - El espectro de la señal B/N ocupa 5 MHz de ancho de banda, pero si aumentamos la resolución vemos que no es un espectro continuo sino que se trata de lóbulos separados.

En Europa se desarrolló el sistema PAL (Phase Alternate Line), que realmente es una mejora del sistema NTSC, inventada por el investigador Walter Bruch en los laboratorios de Telefunken (Hannover, Alemania) en 1963. El PAL ha sido el sistema de TV analógica más extendido en el mundo. No sin variantes, fue adoptado en la mayor parte de Europa (excepto Francia y Europa del este), Brasil, Argentina, Paraguay, Uruguay, gran parte de África (excepto países de habla francesa), Australia, China, la India, Turquía y países árabes. En España se utilizó en TV terrena, hasta el pasado 3 de abril de 2010 (aunque sigue en uso en algunas plataformas de TV por cable). Aun sigue en uso en muchos países que no han completado la transición a TV digital. Es el sistema de TV con más usuarios de la historia y, seguramente, todavía es el más visto del mundo.

El último sistema analógico del que hablaremos es el SECAM (Sequentiel Couleur Avec Mémoire) desarrollado en Francia por Henri de France trabajando en la empresa Thomson (Boulogne, Francia). Realmente, es un sistema más antiguo que el PAL que se adoptó en Francia y países de su influencia: Marruecos, Túnez, Camerún... También fue adoptado por la antigua Unión Soviética y por otros países del este de Europa.

Todos los sistemas de color analógicos se basan en expresar la señal en color como una señal en blanco y negro (B/N) a la que se suma una "componente de color". Así se logra que los receptores blanco y negro reciban "su señal de siempre" y no se preocupen por la parte de color; el receptor en color deberá diseñarse para detectar la parte de color. Una de las especificaciones de diseño fue que el ancho de banda total de la señal en color fuese el mismo de la antigua señal B/N, así se aprovecharían las antenas emisoras y toda la infraestructura dedicada a llevar la señal a los centros emisores (red terrestre de distribución). Esta restricción obliga a encontrar un "hueco" o zona sin información (o sin mucha información) en la señal B/N y, además, implica que el color realmente va a ser una señal interferente que puede provocar errores (sumar ruido a la señal de brillo). Esa interferencia del color sobre la luminancia, llamada cross-color, ha existido siempre en la TV analógica. También hay una interferencia de la señal de brillo (o luminancia) sobre el color: cross-luminancia.

Una imagen en color está formada por tres imágenes o señales: R, G y B que miden el "contenido" de cada punto luminoso en tres colores básicos o primarios: rojo, verde y azul.

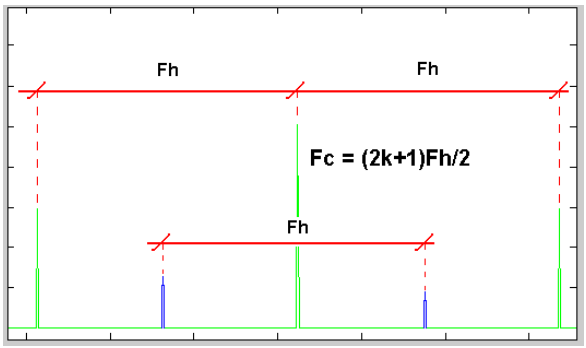


Figura 6 - La señal de color (verde) se coloca en la parte de frecuencia alta donde la luminancia (verde) ya es débil. Además, se modula de tal forma que los máximos no coincidan.

En el sistema NTSC se decidió utilizar una conversión de las señales originales R, G y B según las siguientes fórmulas:

$Y = 0{,}299R + 0{,}587G + 0{,}114B$
$I = 0{,}595R - 0{,}274G - 0{,}321B$
$Q = 0{,}211R - 0{,}522G + 0{,}311B$

Donde la señal Y corresponde con una señal B/N equivalente a la imagen en color que tenemos. Las componentes I y Q son la parte de color y serán añadidas a la señal B/N (señal principal).

La señal de TV en color reduce la calidad de las componentes de color

Fijaos que si tenemos $R = G = B$ (que es lo que ocurre con una señal gris, esto es: una imagen que ya es blanco y negro como una película antigua) las componentes de color se anulas ($I = Q = 0$). A las señales I y Q se les aplicará una reducción de calidad (en analógico un filtrado paso-bajo) y se modularán en una frecuencia de 3.58 MHz (subportadora de color) antes de ser sumadas a la señal principal.

Fijaos que se usa la idea de que la parte de color es menos importante y se puede reducir su calidad (esa es una idea que se sigue utilizando en los sistemas digitales). La subportadora de color se ha elegido de forma que la "interferencia" se sume en la parte menos significativa de la señal de luminancia. Además el valor de esa frecuencia se eligió de forma que los máximos en el espectro de color (una vez modulados) no coincidieran con los de luminancia "interferencia no destructiva".

La TV PAL utiliza la misma idea del sistema NTSC (modular el color con una sub-portadora auxiliar). Las diferencias son las siguientes:

- PAL tiene 25 imágenes por segundo y 625 líneas por imagen (frente a 30 imágenes por segundo con 525 líneas cada una en NTSC). Desde siempre, la TV europea ha tenido más resolución pero menos imágenes por segundo que la norteamericana.
- PAL utiliza otras fórmulas para la transformación del color:

  $Y = 0{,}299R + 0{,}587G + 0{,}114B$
  $U = 0{,}49(B - Y)$
  $V = 0{,}88(R - Y)$

  A veces, a estas componentes se les llama Y, Cb, Cr y cumplen la misma propiedad de las YIQ; si $R = G = B$ (color gris), las componentes de color U y V se anulan.
- La portadora de color es de 4.43 MHz. La mayor resolución implica que se debe subir más en frecuencia para llegar a la zona de componentes débiles de luminancia.
- La característica más definitoria del PAL (la que le da su nombre: Phase Alternate Line) es la alternancia en fase de la señal de color. Esto, básicamente consiste en que la señal de color va modulada con una fase en una línea y con fase opuesta en la siguiente. Los receptores PAL reciben ambas fases y promedian las señales demoduladas. Con esto se consigue anular los errores de color que aparecían, a veces, en el NTSC.

Por último, el sistema SECAM se basa en enviar alternativamente las dos componentes de color: en una línea se envía U (C_b) y V (C_r) va en la siguiente. En este sistema, el color se modula en FM: al tener una sola componente por línea se puede utilizar un sistema de modulación más robusto pero con más gasto de ancho de banda.

Algunos ingenieros dicen que el SECAM es un sistema inferior ya que la resolución en las componentes de color se reduce a la mitad y los receptores tienen que almacenar la línea anterior del componente que falta para "repetirla". Yo creo (y digo creo porque no me he puesto a hacer simulaciones para demostrarlo) que la pérdida de resolución en el color no causa errores importantes (de hecho, también se hace en los sistema digitales) y, sin embargo, la mayor robustez de la modulación FM hace a SECAM un sistema superior. Como la cuestión no parece resuelta (he encontrado opiniones en ambos sentidos pero ninguna parece definitiva, ni siquiera mayoritaria), es casi seguro que la diferencia de calidad entre SECAM y PAL era muy leve. No puedo afirmarlo categóricamente porque no he podido compararlos "en persona".

Los principales sistemas de TV en color analógicos son NTSC, PAL y SECAM

Una cuestión polémica es por qué Francia adoptó el SECAM. Era un sistema de diseño más antiguo (y de origen nacional) pero se adoptó para difusión cuando el PAL ya dominaba en Europa. Se dice que pudo influir cierto proteccionismo: adoptando un sistema diferente se protege a la industria nacional de la llegada de equipos extranjeros (que acabaron llegando ya que a partir de los 90 los diseños de televisores utilizaban chips decodificadores de vídeo capaces de entender PAL, NTSC y SECAM).

Algunos ingenieros norteamericanos bromeaban definiendo SECAM como "System Essentially Contrary to the American Method".

El sistema NTSC (EEUU 1953) utiliza una codificación del color en cuadratura en la que la señal I va modulada por $cos(w_0t)$ y la Q por $sen(w_0t)$. En este tipo de codificaciones el vector de color (o el número complejo I+jQ) cumplen la siguiente regla:

- Su módulo, $s = \sqrt{U^2 + V^2}$ es proporcional a la saturación o "pureza" del color. Un color de alta saturación tiene un porcentaje de blanco bajo. Mientras que si S es pequeña, el color tiene mucha parte de blanco.
- Su fase, $\varphi = tg^{-1}(\frac{V}{U})$ es proporcional al tinte o matiz del color. El tinte es un ángulo que indica de qué color (rojo, naranja ...) es el punto en cuestión. El origen ($\varphi = 0$) se suele poner en el rojo y existe una figura llamada "vectorscopio" que nos indica la distribución de los colores.

Figura a1 - Vectorscopio, el tinte varía con el ángulo, la saturación crece al alejarse del centro.

El problema del sistema NTSC es que la acumulación de amplificadores en el camino de la señal puede generar un error constante en la fase de la señal en cuadratura. Eso genera un error constante en el tinte y hace que las imágenes salgan con colores falsos. Los televisores norteamericanos tienen un mando para ajustar el tinte (H de "hue", en inglés).
Los diseñadores del PAL (1963) se propusieron evitar este problema. La solución fue alternar la fase de la componente en cuadratura. En la línea N se envía el color según la fórmula: $Ucos(w_0t) - Vsen(w_0t)$ y en la línea N+1 se le cambia el signo a V: $Ucos(w_0t) + Vsen(w_0t)$. Eso dibujado en forma de vector podemos verlo en la figura siguiente (en la mitad de las líneas enviamos el tinte cambiado de signo). Como el diseñador del receptor ya sabe esta característica, lo que hace es cambiarle el signo a la componente V decodificada en las líneas en que se le cambió el signo. Además, se hace el promediado de U y V entre el valor de la línea actual y el valor correspondiente a la línea anterior (para ello los televisores PAL incorporan un circuito que retarda la señal 64 μseg, tiempo de línea de este sistema). Ese promediado es capaz de compensar errores de tinte (figura 3.8) al precio de introducir errores mínimos ya que en vez de recibir el valor auténtico, se está promediando el color con el de la línea anterior (dos píxeles más arriba debido al entrelazado), todo funcionará bien si la imagen es suave, suposición que suele ser cierta).

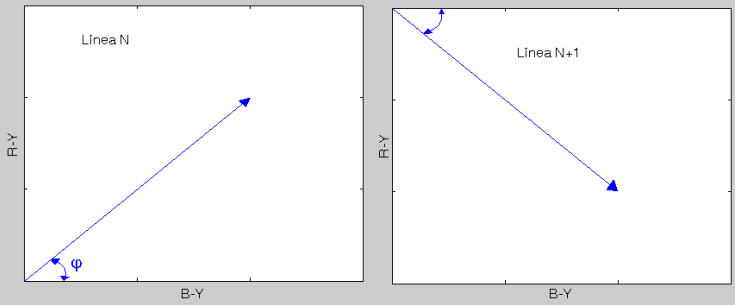


Figura a2 - Alternancia de fase en el sistema PAL (vista en polares: tinte, saturación).

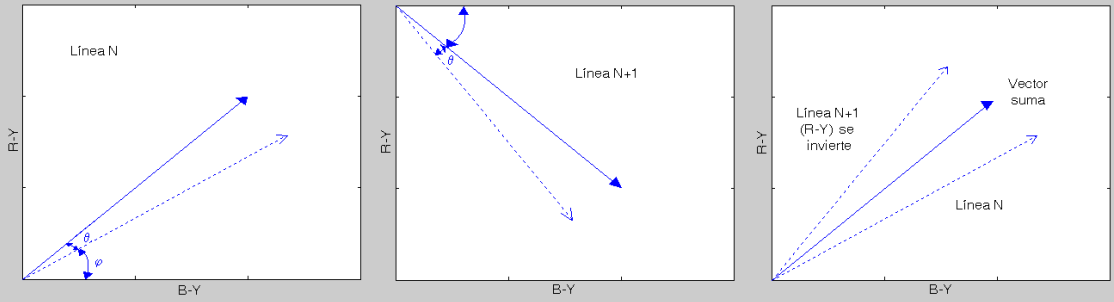


Figura a3 - Corrección de un error de tiente α (negativo).

INTENTOS DE MEJORAR LA TELEVISIÓN ANALÓGICA

Una vez consolidados los sistemas analógicos de televisión en color la tecnología de receptores evolucionó mucho. Por ejemplo, algunas mejoras importantes fueron:

- El mando a distancia (finales de los 70)... que obligó a centralizar en una CPU el control de todos los elementos variables del equipo y supuso un cambio radical en el diseño de los receptores.
- El tubo de imagen "trinitron" que posibilitaba pantallas con una curvatura mínima y mayor brillo. Sony retuvo la patente durante el límite legal de 20 años y en el momento de su caducidad (1996) surgieron muchas imitaciones: flattron, diamondtron.
- Relacionado con lo anterior, empezaron a aparecer pantallas cada vez mayores de 28 pulgadas y más (una pantalla se suele medir por el tamaño físico de su diagonal, por herencia anglosajona se mide en pulgadas, 1 pulgada = 2.54 cm).

Este último avance hizo pensar que la resolución de los sistemas analógicos era insuficiente. Hablamos de 576 líneas activas en PAL y SECAM, 480 en NTSC (todos ellos utilizando una relación de aspecto, ancho/alto = 4/3). Se definió alta definición como aquellos sistemas capaces de representar más de mil líneas activas en pantalla.

Los primeros intentos de crear sistemas de alta definición fueron:

- MAC (Multiplexed Analog Components): se llegó a estandarizar como sistema para el envío de TV terrestre de alta definición. Debería haber empezado a utilizarse en 1995, algo que nunca ocurrió. El sistema se basa en multiplexar en tiempo las diversas informaciones necesarias: brillo, color, audio... Aunque se trata de concatenar señales analógicas, tanto el emisor como el receptor deberán digitalizar la señal para poder ensamblar y desensamblar la señal MAC. MAC se llegó a utilizar en TV satélite pero nunca tuvo mucho éxito. Tal vez el fracaso se debió al elevado precio de los receptores (en aquel momento un sistema que hiciese procesado de señal digital a gran velocidad estaba en el límite de la tecnología) y por el hecho de que no se ganaría nada utilizando conversión a PAL para reutilizar televisores antiguos (haciendo esto en TDT se gana calidad pero convertir a PAL la señal MAC es perder la mitad de la resolución).
- MUSE (Multiple sub-Nyquist Sampling Encoding): fue un sistema desarrollado en Japón en los años 80 y fue el primer sistema de alta definición con éxito (sólo en Japón). Igual que MAC es una técnica que combina procesos digitales con otros analógicos. Nunca se consiguió importar a Europa. La principal limitación es el enorme costo en ancho de banda que supone.

Paralelamente al interés por la alta definición, nació el concepto de formatos panorámicos. La idea es que la relación de aspecto clásica (4/3) es muy baja: la pantalla es casi cuadrada y el usuario mantiene los ojos fijos en la pantalla sin apenas moverlos. Desde siempre, el cine ha sido una experiencia más agradable porque la pantalla más ancha permite que la mirada del observador se desplace por la misma. Se definió como formato panorámico aquel que tiene una relación de aspecto al menos aproximadamente igual a $16/9 = (4/3)^2$.

El primer formato comercial capaz de enviar televisión en formato panorámico fue el PALplus. Un televisor PAL estándar representará una imagen panorámica con barras negras arriba y abajo: las 432 líneas centrales tendrán imagen, la franja negra superior estará formada por (576-432)/2 = 72 líneas, la franja negra inferior estará formada por otras tantas. Las líneas negras, realmente contienen una señal modulada en alta frecuencia, invisible para un televisor PAL. Un televisor PALplus era capaz de decodificar esa señal oculta (señal de ayuda vertical) y utilizar esa información para generar 576 líneas con formato 16/9 (con mayor ancho que las convencionales). Hubo muchas pruebas de PALplus en diferentes televisiones europeas en los años 90 (en España se probó en autonómicas) pero ninguna tuvo un éxito destacable. Actualmente, ningún operador utiliza PALplus.

LA TECNOLOGÍA DE IMAGEN Y SU EVOLUCIÓN

Este artículo está dedicado a los avances en la señal de televisión, sus formatos y su tecnología de emisión/recepción. Paralelamente a la señal también ha evolucionado la tecnología de imagen, esto es: los dispositivos de captura (cámaras) y representación (monitores). Este apartado se ha incluido para conocer un poco de esta evolución.

Hasta los años 80, tanto cámaras como monitores se basaban en tubos de rayos catódicos (tubos de vacío donde se provocaban descargas de electrones, "tubo de cámara" para las cámaras y "tubo de imagen" para los monitores). Los tubos de rayos catódicos eran pesados (porque debían soportar las presión atmosférica sobre una cámara de vacío), grandes (necesitaban mucho fondo para dar recorrido a los haces de electrones), difíciles de fabricar y de gran consumo (porque necesitan enormes tensiones para generar los haces de electrones).

Los avances vinieron de la mano de aplicar una "filosofía digital" (o discreta) creando matrices de elementos sensores para las cámaras y otras de elementos luminosos para los monitores.

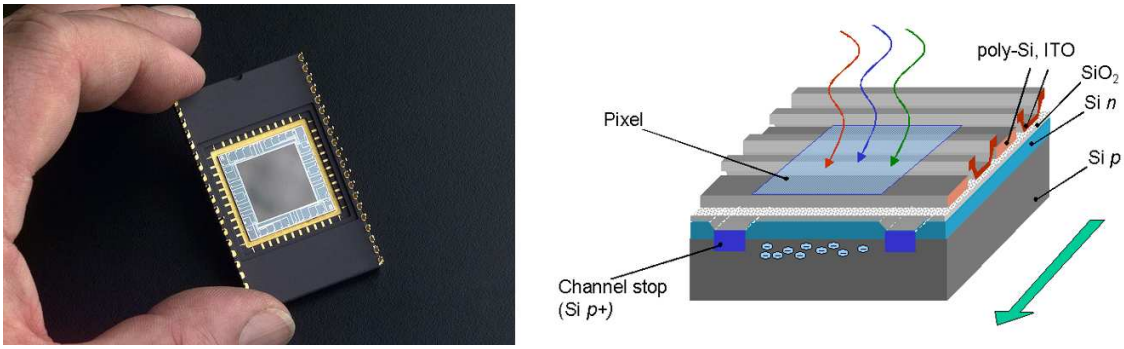


Figura 7 - Sensor CCD.

Las nuevas tecnologías de imagen son los sensores CCD y los sensores CMOS.

Los sensores CCD (Charged Coupled Devices), son transistores de efecto de campo sensibles a la luz. En lenguaje "cuántico" puede decirse que cada fotón (unidad de energía luminosa) recibido produce un par electrón-hueco (una unidad elemental de carga en la capacidad que el transistor genera entre puerta y tierra). Hasta aquí sabemos como convertir la luz en una señal (captación de imagen) pero otro problema es conseguir leer esa imagen. Los CCD's se construyen en estructuras espaciales (1D o lineales para escáneres o fotocopiadoras, 2D o matriciales para cámaras), además, su estructura permite que la carga acumulada se pueda desplazar muy rápidamente de un transistor a otro. Esa es la propiedad que se aprovecha para leer los CCD's. Recordad que toda cámara imita al ojo, donde un sistema óptico genera una imagen sobre una superficie sensible a la luz (retina). Los CCD's se convirtieron en la primera retina electrónica y por tanto mucho más eficiente en tamaño y consumo que un tubo de cámara.

Los sensores CMOS se basan en el mismo principio de funcionamiento pero no utilizan desplazamientos de carga para la lectura. Se pueden leer píxel a píxel y permiten integrar más funciones en el chip. Durante mucho tiempo fueron una alternativa barata pero de inferior calidad comparados con los CCD's. Los primeros sensores CMOS no podían competir con los CCD's en número de píxeles por unidad de área, por eso se utilizaban en cámaras sencillas y baratas (webcams). En los últimos años la tecnología CMOS ha mejorado mucho llegando a ser comparable a los sensores CCD.

En cuanto a los monitores debemos destacar dos nuevas tecnologías: la tecnología de cristal líquido o LCD y la tecnología de plasma.

Tecnologías de cristal líquido (LCD) se basan en la propiedad de ciertos materiales de filtrar la luz de manera diferente según el campo eléctrico que reciben. Una luz inicial es polarizada por un filtro polarizador (polarizador pasivo: ángulo de polarización fijo), después se atraviesa el material (polarizador activo: que puede cambiar o no el ángulo de polarización según la tensión recibida) y se encuentra, por último, otro filtro polarizador (un polarizador pasivo con un ángulo diferente al inicial). La luz atravesará o no el segundo filtro dependiendo de si el material ha girado la polarización inicial o no. El material consiste en un líquido que contiene moléculas orgánicas en suspensión. Los primeros display's LCD eran sistemas pequeños para calculadoras y relojes. El material se excitaba a través de electrodos individuales. Para monitores se han impuesto matrices activas que se basan en transistores TFT (Thin Film Transistor) que permiten aplicar tensiones individualizadas para gobernar cada píxel.

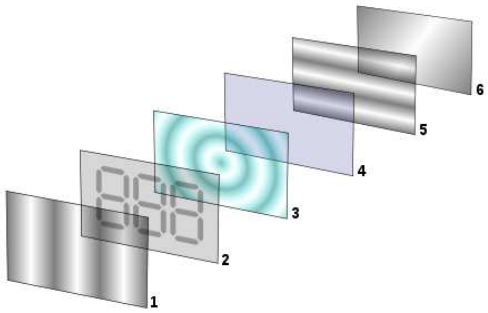


Figura 8 - Monitor TFT

Los monitores de plasma se basan en la propiedad de ciertos gases (como Argón, Xenón y Neón) de ionizarse y emitir luz ultravioleta al ser sometidos a tensiones fuertes (es el mismo principio de los tubos fluorescentes). Se crea una matriz de celdas de gas que son controladas por electrodos. El gas emite luz ultravioleta que es captada por granos de fósforo parecidos a los de los monitores de tubo. El efecto es el mismo: al absorber la energía de la luz ultravioleta, el fósforo brilla (emite luz visible). Esta tecnología produce un tamaño de píxel mucho mayor que en los monitores LCD (no se ha conseguido bajar de 0.3 mm, las imágenes son de menor resolución: menor número de píxeles por unidad de longitud). Por eso, se usa en pantallas grandes que van a ser observadas a distancia y no en monitores de ordenador. Otros problemas son: la pérdida de brillo por desgaste del fósforo y el mayor consumo.

Como ventajas: no se ven afectados por campos electromagnéticos externos (aunque, a cambio, producen mucha radiación) y son muy rápidos. Por último, destacar que un monitor de plasma no es capaz de crear puntos de intensidad intermedia, todos los brillos son "todo o nada". Para lograr intensidades se hace que los elementos conmuten a una frecuencia invisible para el ojo y el ciclo de trabajo (tiempo activo/tiempo total) determina el brillo.

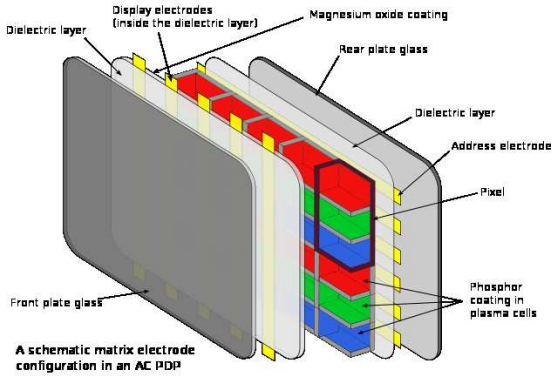


Figura 9 - Monitor de Plasma

MPEG Y DVB: TELEVISION DIGITAL

Y bien... hemos llegado a la actualidad. De esta parte hablaremos poco... en parte porque "todavía" no es historia y también porque ya llevo escrito mucho y me estoy cansando :).

Hacia el año 1986 se consiguió reunir al "Joint Photographic Experts Group" más conocido como JPEG. Como adivinaréis este grupo de expertos definieron el conocido estándar de codificación de imagen fija, aunque tardaron hasta los primeros 90 (sí, sí, JPEG es más que un formato de fichero, es un estándar que contiene varios algoritmos de compresión llamados "modos de JPEG").

El algoritmo básico JPEG (JFIF) consiste, simplificando, en los siguientes pasos:

- Si la imagen es en escala de grises, sólo tendremos luminancia. Si es en color la convertiremos al formato YUV con lo que tendremos tres subimágenes (planos). Los planos U y V pueden ser sub-muestreados, esto es: se eliminan algunas muestras. Hay varios formatos. El más habitual es el que reduce U y V a imágenes de la mitad de altura y la mitad de anchura.
- Cada plano se divide en bloques (o matrices) de tamaño 8x8. Para cada bloque se calcula la transformada discreta del coseno (DCT: Discrete Cosine Transform). Esta transformada (de propiedades similares a la de Fourier) es capaz de concentrar la mayor parte de la información en la parte de baja frecuencia.
- Los coeficientes (valores de la transformada) son cuantificados, esto es: se reduce su calidad de acuerdo con las tablas de cuantificación calculadas por A. B. Watson en 1993. Dichas tablas indican hasta donde se puede degradar cada coeficiente según su posición (en frecuencia) sin que haya degradación apreciable en la calidad que percibe el usuario final. Esta cuantificación, a grandes rasgos, va a reducir el margen dinámico de cada coeficiente (y por tanto el número de bits necesarios para su representación) y, sobre todo, va a convertir en cero muchos coeficientes de alta frecuencia.
- La cuantificación (que es la parte que introduce una pérdida de calidad inapreciable) no es más que una preparación para aplicar un algoritmo de codificación variable (se puede usar el código de Huffman o alguna variante). El reducido margen dinámico de los coeficientes y, sobre todo, la gran cantidad de ceros en la parte de alta frecuencia, hacen que la compresión (sin pérdidas) sea muy grande.

JPEG es por tanto un algoritmo de compresión con pérdidas (invisibles según el cálculo de Watson) que permite reducir el tamaño binario de una imagen digital dividiéndolo por factores de hasta 20. Por sus buenos resultados y porque es fácil de implementar se ha convertido en el estándar de facto en compresión de imagen fija.

Y lo que os estaréis preguntando es: ¿Por qué hablar de JPEG en un artículo sobre televisión? Simplemente porque fue el primer paso en el desarrollo de estándares de codificación de vídeo. Pensad que JPEG prácticamente resuelve el problema de aprovechar al máximo la redundancia espacial (parecido de un píxel con sus vecinos) para comprimir una imagen.

Poco después de JPEG, empezaron a aparecer estándares de codificación de vídeo:

M-JPEG (Motion JPEG): es la idea más simple, cada fotograma (imagen individual) se codifica en formato JPEG. Al no aprovechar la redundancia temporal (parecidos entre fotogramas) se trata de un método ineficiente (baja compresión). A pesar de ello, MJPEG y algunas variantes (DV-25) se siguen utilizando en dispositivos como videocámaras. AL tratarse de un método simple es adecuado para sistemas con poca potencia de cálculo y, además, al trabajar con imágenes independientes es un formato adecuado para poder editar el vídeo grabado que después se convertirá a un formato más eficiente.

H.261: pensado para videoconferencia es el primer estándar que introduce el concepto de codificación híbrida (transformada DCT para la redundancia espacial y codificación predictiva para la redundancia temporal). H.261 codifica la primera imagen con un método prácticamente igual al estándar JPEG. Después, se hace una "predicción" de cómo será la siguiente imagen y se codifica JPEG la diferencia entre la imagen que realmente hay y su predicción.

El sistema continúa hasta el infinito construyendo predicciones y "corrigiéndolas"... Sólo se envían las correcciones, las predicciones son calculadas por igual por emisor y receptor. H.261 mejoró mucho cuando se comenzaron a aplicar métodos de detección de movimiento que permitieron hacer mejores predicciones (aunque esto obliga a enviar al receptor los movimientos detectados en forma de "vectores de movimiento").

MPEG (Motion Pictures Experts Group): creado a partir de H.261 para permitir otras aplicaciones además de la videoconferencia. H.261 no se puede utilizar para almacenar vídeos y menos para emitir televisión porque toda la secuencia depende de emitir y recibir correctamente la imagen inicial (la que se codifica sin predicciones). Para superar ese problema se decidió dividir el vídeo en "grupos de imágenes" (GOP's: Groups of Pictures) donde la primera se codifica sin predicciones y las siguientes a partir de ella. Además, en MPEG se añadió la posibilidad de codificar algunas imágenes de forma "bidireccional", esto es: teniendo en cuenta el parecido con las anteriores y con las siguientes.

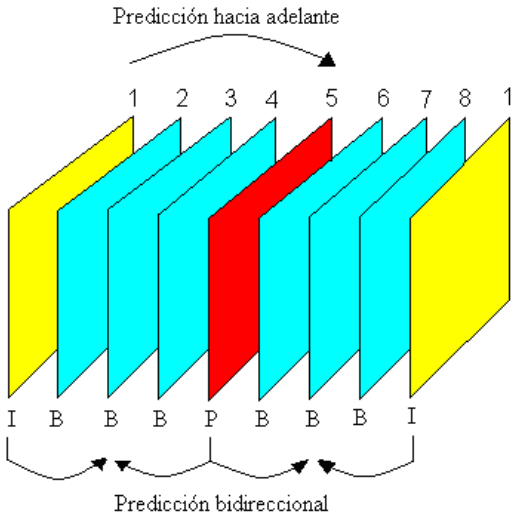


Figura 10 - Grupo de imágenes MPEG. Tipos de imágenes: I (codificación independiente), P (predicción hacia adelante), B (predicción bidireccional).

El siguiente paso al MPEG fue estandarizarlo para su empleo en televisión digital. El estándar DVB (Digital Video Broadcasting) consiste en una serie de normas que definen cómo enviar y recibir vídeo. Se empezaron distinguiendo tres tipos de DVB: DVB-S (televisión digital por satélite), DVB-C (televisión digital por cable) y DVB-T (televisión digital terrestre). Cada una de estas variedades utiliza un tipo de modulación digital diferente para adaptarse a cada tipo de canal de transmisión. Como novedades recientes podemos destacar:

- El uso de MPEG versión 4 para canales de alta definición (los canales convencionales comenzaron con MPEG-2 pero para alta definición se requiere una mayor compresión y MPEG-4 presenta aproximadamente el doble de eficiencia).
- La aparición de nuevas variedades de difusión como el uso de Internet como medio de transmisión. Realmente, IPTV (televisión digital por Internet) empezó siendo un sistema independiente del DVB pero posteriormente el consorcio se ha unido a la iniciativa.

CONCLUSIONES

Como conclusión me gustaría decir que "el sueño imposible" que comentábamos al comienzo sigue despertando tanto interés como al principio. Quizá su mayor encanto es que "siempre será imposible" ya que por mucho que se avance nunca lograremos que la experiencia sea igual a presenciar un evento en directo (tal vez deba ser así porque de lo contrario asistir en directo no tendría interés). Debido a esta conclusión está claro que la evolución de la televisión no terminará nunca. Actualmente el interés en el mundo audiovisual está volcado en los sistemas de visualización 3D. Estos sistemas aprovechan la binocularidad (visión con dos ojos) para crear sensación de profundidad (entregando una señal diferente a cada ojo). Ya se han probado emisiones de televisión en 3D (por ejemplo, durante el mundial de fútbol de 2010) pero todavía no hay un estándar establecido para ese tipo de emisiones. Sin duda, en breve lo habrá. ■

EN LA PRÁCTICA

Introducción a SPICE

Cálculo de Geometría para Misiones Planetarias

por Jose Luis Vázquez García

Todos hemos visto alguna vez las impresionantes imágenes de Marte que obtienen las sondas espaciales que orbitan el planeta. Nos dicen que son de tal o cual cráter, o del polo norte, o que corresponde a unas determinadas longitud y latitud en la superficie de Marte. Probablemente alguna vez te has preguntado cómo, una vez que la sonda envía la imagen a tierra, se calcula esa información. SPICE juega un papel muy importante en el proceso.

SPICE es una libreria de software (el *toolkit*) y una serie de formatos de datos que pueden ayudar a un científico a usar datos auxiliares para planificar las observaciones de un vehículo espacial, o para analizar los datos científicos obtenidos por esas observaciones. SPICE también es útil para científicos e ingenieros en la planificación de futuras misiones. En este contexto, *datos auxiliares* significa geometría de misión, y conversión entre diferentes sistemas de tiempo.

> *Por geometría de misión se entiende el cálculo de los parámetros geométricos necesarios para controlar un satélite y obtener datos de él. Por ejemplo, su posición y orientación, los parámetros que definen su órbita alrededor de un cuerpo, etc.*

SPICE fue desarrollado y es mantenido por el **Navigation and Ancillary Information Facility** (NAIF) del **Jet Propulsion Laboratory** para la NASA. Es software libre en el sentido de que no está sujeto a restricciones a su exportación bajo las leyes de Estados Unidos, y se distribuye a través de la página web de NAIF sin coste alguno. En el momento de la publicación de este artículo, SPICE está disponible en Fortran, C, IDL y Matlab para diferentes plataformas, y NAIF está trabajando en una versión Java del toolkit.

La página oficial de NAIF, http://naif.jpl.nasa.gov ofrece toda la información, los toolkits y la documentación necesaria para entender SPICE y trabajar con él.

SPICE puede responder a preguntas del tipo:

- ¿Cuál será la posición de Rosetta vista desde la Tierra en un determinado momento?
- Tengo una imagen de Marte tomada por MGS, y conozco el valor del reloj del satélite en el momento en que fue tomada. ¿En qué instante fue tomada la imagen?
- Quiero usar imágenes de Mars Express para buscar rovers en la superficie de Marte. ¿Es la resolución de la cámara suficiente?

En este artículo aprenderemos a instalar la librería (el toolkit) para C, y veremos unos ejemplos sencillos que nos ayudarán a coger soltura con SPICE.

La ESA y la NASA utilizan SPICE para los cálculos en algunas de sus misiones

INSTALANDO SPICE

Ya que todos los ejemplos que vamos a ver están en C, mostraremos aquí como instalar el toolkit para Linux/C (CSPICE). Ve a la página: http://naif.jpl.nasa.gov/naif/toolkit_C.html y descarga el toolkit adecuado para tu ordenador, y descomprímelo en el directorio en donde lo quieres instalar. Una vez descomprimido, ve al directorio cspice, y ejecuta el script *makeall.csh* (csh tiene que estar instalado en el sistema). Si bien el toolkit ya viene precompilado, siempre es una buena idea ejecutar el script para descartar posibles problemas con tu equipo.

```
[jvazquez@cork: cspice] $ ls
cspice.tar.Z
[jvazquez@cork: cspice] $ uncompress cpice.tar.Z
[jvazquez@cork: cspice] $ tar xf cpice.tar
[jvazquez@cork: cspice] $ ls
cspice/ cspice.tar
[jvazquez@cork: cspice] $ cd cspice
[jvazquez@cork: cspice] $ ls
data/ doc/ etc/ exe/ include/ lib/ makeall.csh* src/
[jvazquez@cork: cspice] $
```

Pasos Necesarios para instalar SPICE. Una vez descomprimido, ejecuta el script *makeall.csh* para compilarlo

Una vez compilado el toolkit, podrás ver varios directorios. De momento nos interesan *include*, donde se encuentran los ficheros de cabecera, y *lib*, donde está la biblioteca de SPICE.

LOS KERNELS DE SPICE

El toolkit es una parte de SPICE, la otra son los ficheros con datos que SPICE usa para trabajar.

Estos ficheros se llaman kernels, y almacenan posición y orientación para los cuerpos del sistema solar (satélites artificiales incluidos), información de tiempo, y parámetros que describen los instrumentos a bordo de un satélite.

Los kernels pueden ser ficheros binarios o de texto, y cada kernel sólo almacena un tipo de información; es decir, un kernel no almacena, por ejemplo, información sobre la posición de un cuerpo y sobre conversiones entre sistemas de tiempo (aunque esto es la mayoría de las veces más una convención que una restricción técnica).

Para usar SPICE en tu aplicación, primero tienes que cargar los kernels que necesitas. No es un paso trivial, muchas veces es más difícil averiguar cuáles son los kernels que hay que cargar que la aplicación en sí. No hay una regla para esto, si no que es la experiencia la que te dirá que kernels necesitas y cuales no.

¿Dónde encontrar los kernels? Para kernels genéricos y misiones de la NASA, el primer sitio en el que deberás buscar es

ftp://naif.jpl.nasa.gov/pub/naif

Para misones de la ESA, el 'almacén' oficial es

ftp://ssols01.esac.esa.int/pub/data/SPICE

Además, puedes acceder a este sitio a través de

su interfaz web.

Como puedes ver, este inferfaz proporciona mucha información sobre los kernels, sin necesidad de entender demasiado sobre SPICE.

En el siguiente artículo profundizaremos un poco más en los distintos tipos de kernels que hay, y cómo obtener información básica sobre ellos.

PRIMER PROGRAMA SPICE

Vamos a crear un programa muy simple, para comprobar que SPICE se ha instalado y funciona. Simplemente, imprimiremos la razón de conversión de grados a radianes, y de radianes a grados, que en SPICE vienen dados por dos funciones, `rpd_c` y `dpr_c`.

PROGRAMA 1: Probando SPICE

```
#include <stdio.h>

/* Cabecera de SPICE */
#include <SpiceUsr.h>

int main() {

  float radianes_por_grado = rpd_c();
  float grados_por_radian = dpr_c();

  printf( "1 grado  = %f radianes\n",
          radianes_por_grado );
  printf( "1 radian = %f grados\n",
          grados_por_radian );

  return 0;
}
```

Si has echado un vistazo al directorio *include*, habrás visto un montón de ficheros de cabecera. Sin embargo, el fichero *SpiceUsr.h* incluye todos los ficheros necesarios, por lo que es la única cabecera que necesitamos incluir en nuestros programas.

Para compilar el programa, debes decir a gcc donde encontrar las cabeceras de SPICE y la librería. SPICE tiene la particularidad de que la librería (cspice.a) se enlaza como un fichero objeto más, por lo que simplemente la tienes que añadir a los parámetros de entrada de gcc, en lugar de usar el flag *-l* del compilador. En mi ordenador, el comando para compilar este programa sería:

```
gcc -o program1 program1.c /opt/SPICE/cspice/lib/cspice.a \
  -I/opt/SPICE/cspice/include/
```

Compilando y ejecutando el programa 1.

```
[jvazquez@cork: programs] $  gcc -o program1 program1.c \
    /opt/SPICE/cspice/lib/cspice.a \
    -I/opt/SPICE/cspice/include/
[jvazquez@cork: programs] $ ls
program1* program1.c
[jvazquez@cork: programs] $ ./program1
1 grado = 0.017453 radianes
1 radian = 57.295780 grados
```

CALCULANDO POSICIONES

Ahora veremos un ejemplo un poco más complejo que combina el trabajo con tiempos y el cálculo de posiciones. Como esto requiere un conocimiento detallado de los sistemas de tiempo y de referencia espacial que usa SPICE, no te preocupes si no entiendes todo a la primera; paciencia.

Para simplificar imprimir vectores, creamos la libreria *print_ tools*, que puedes ver en el listado siguiente. Primero el fichero de cabecera *print_ tools.h*.

PROGRAMA 2: Librería para imprimir vectores

```
#ifndef PRINT_TOOLS_H
#define PRINT_TOOLS_H

#include <stdio.h>

#include <SpiceUsr.h>

void printVector( FILE *out, SpiceDouble *vector,
                  unsigned long size );

#endif
```

Y este es el fichero *print_ tools.c*

```
#include "print_tools.h"

void printVector( FILE *out,
                  SpiceDouble *vector,
                  unsigned long size ) {

  unsigned long i;

  fprintf( out, "[ " );

  for (i = 0; i < size - 1; i++ )
    fprintf(out, "%f, ", vector[i]);

  fprintf(out, "%f ]", vector[size - 1]);

}
```

Ahora, vamos a calcular la posición de la luna vista desde la tierra, en el sistema de referencia *J2000*, el 1 de junio de 2010, a las 12 del mediodía.

PROGRAMA 3: Calculando la posición de la Luna (1)

```
#include <stdio.h>

#include <SpiceUsr.h>

#include "print_tools.h"

int main() {

    SpiceDouble posicion[ 3 ];
    SpiceDouble lt;
    SpiceDouble et;

    /*
     * Cargamos los kernels que necesitamos
         * para nuestros cálculos
     */
    furnsh_c( "NAIF0009.TLS" );
    furnsh_c( "DE405.BSP" );

    /* Convertimos tiempo de formato UTC a ET */
    utc2et_c( "2010-06-01T12:00:00", &et );

    /* Calculamos la posicion de la luna */
    spkpos_c( "moon", et, "j2000", "none",
                  "earth", posicion, &lt );

    printf( "Posicion de la luna: " );
    printVector( stdout, posicion, 3 );
    printf( " Km\n" );

    return 0;

}
```

Vemos que las variables que declaramos son de tipo SpiceDouble; SPICE, como muchas otras librerías, utiliza sus propios tipos que permiten que los programas puedan ser compilados en diferentes plataformas sin que haya que cambiar el código. SpiceDouble está definido como el tipo de datos de punto flotante que ocupa el doble de memoria que un tipo entero; dependiendo de tu sistema operativo, será double o long double.

SPICE nos permite utilizar distintos Sistemas de Referencia fácilmente

El siguiente paso es cargar los kernels que necesitaremos.

DE405.BSP es el kernel que almacena información sobre efemérides de los principales cuerpos del sistema solar, y *NAIF0009.TLS* almacena información que permite convertir de UTC a ET, y viceversa. Puedes encontrar los kernels aquí y aquí (consultad el cuadro al final para obtener las URL completas si los enlaces no funcionan).

Ahora, ya que todos los cálculos que dependen del tiempo en SPICE se realizan en ET, tenemos que pasar de UTC a ET, con la función *utc2et_c*.

> ***ET, Ephemeris Time**. El tiempo de un evento en ET se mide como segundos que han pasado desde el 1 de enero de 2000 al mediodía.*

Fíjate que el formato en que tenemos que especificar UTC es

```
YYYY-MM-DDThh:mm:ss
```

El siguiente paso es calcular la posición de la luna, con la función *spkpos_c*. Los parámetros que recibe son:

1. *target*: es el cuerpo del que queremos calcular la posición.
2. *et*: instante para el que queremos calcular la posición.
3. *frame*: sistema de referencia en el que queremos la solución. En nuestro caso es J2000.
4. *corrections*: dejaremos esto para más adelante. De momento, siempre sera *none*.
5. *observer*: es el cuerpo desde el que queremos calcular la posición. En este caso es *earth*, porque queremos calcular la posición de la luna vista desde la tierra.
6. *position*: es un vector que nos da la posición que queremos calcular, en kilómetros.
7. *light time*: esto lo dejaremos también para más adelante.

Si examinamos el código, veremos alguna característica más de SPICE:

- Todas las funciones de SPICE tienen el sufijo *_c*.
- Muchas de las funciones de SPICE no devuelven valores. En su lugar, devuelven resultados en punteros que se le pasan como argumentos. Por ejemplo, la función *spkpos_c* devuelve dos valores, la posición de un cuerpo y otro parámetro (*light time*), cuyas direcciones se pasan como parámetros a la funcion (fíjate que el parámetro *posicion* se pasa directamente, en lugar de su posición, porque ya es un puntero).

Si compilamos el programa y lo ejecutamos (no olvides incluir en la compilación el fichero *print_tools*), obtendremos el siguiente resultado de la figura.

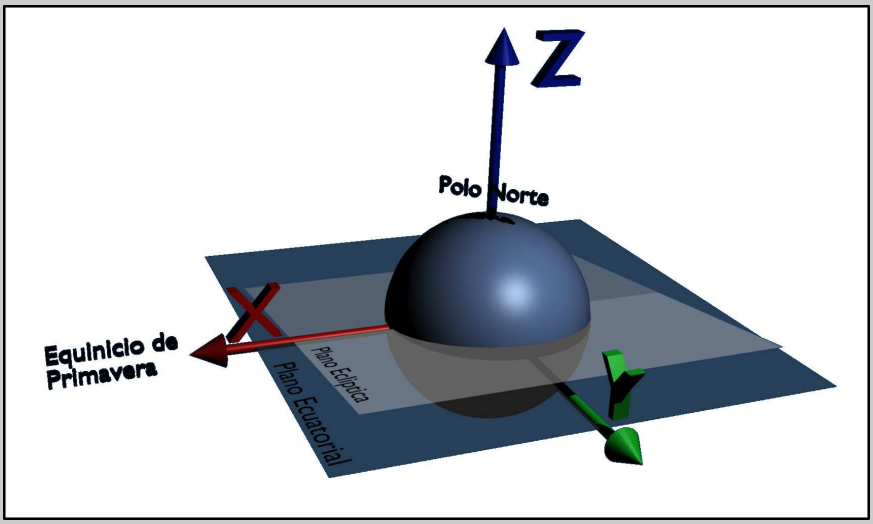
```
[jvazquez@cork: programs] $ ./pos_j2000
Posición de la luna:
[220242.595980, -312947.666645, -121185.983058] Km
[jvazquez@cork: programs] $
```

El problema que tiene este programa es que nos devuelve la posición de la luna en un sistema de referencia que no es muy útil para nosotros. El sistema J2000 está definido de forma que, si colocamos el origen de coordenadas en el centro de la tierra, el eje Z es el eje de rotación de la tierra, y el eje X es tal que el sol pasa por él en equinocio de primavera.

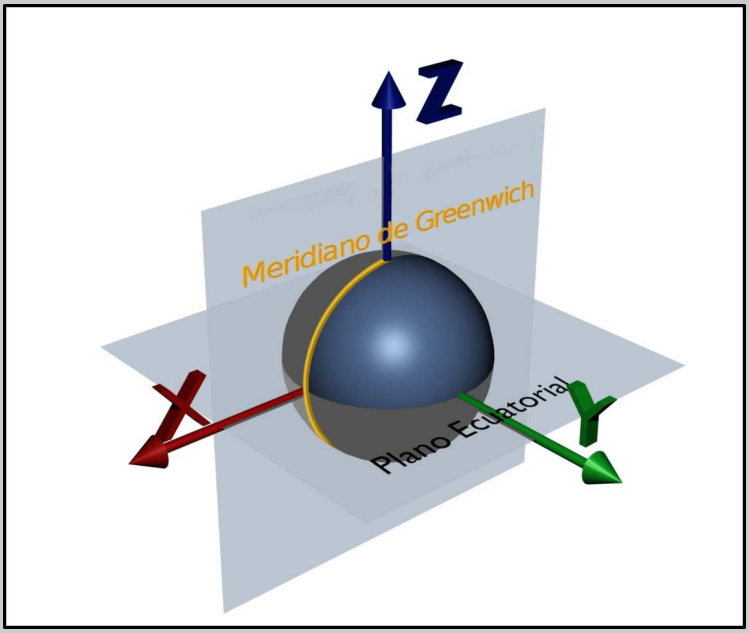
SISTEMAS DE REFERENCIA

En física existen dos tipos fundamentales de sistemas de referencia, inerciales y no inerciales.
En pocas palabras, en un sistema de referencia inercial se pueden aplicar las leyes de Newton, y en uno no inercial, no. Un sistema de referencia que está quieto o se mueve a velocidad constante (es decir, que no acelera) con respecto al fondo de estrellas fijas se considera inercial.
El sistema inercial fundamental en SPICE es el sistema J2000 (también lo puedes encontrar en la literatura como EME2000), y que se define de la siguiente forma:

- El centro del sistema es el centro de la Tierra.
- El eje Z del sistema coincide con el eje de rotación de la Tierra. Por lo tanto, el plano XY del sistema coincide con el plano que contiene el ecuador de la Tierra.
- El eje X se define de forma que el Sol lo cruza (atravesando el plano del ecuador de sur a norte) en el equinocio de verano.
- El eje Y completa el sistema de forma que $X \times Y = Z$.



En otras palabras, el eje X de J2000 es la intersección del plano que contiene al ecuador de la tierra con el plano de la eclíptica, que es el plano que contiene la trayectoria que sigue el sol alrededor de la tierra.
Otro sistema importante es el sistema IAU para la Tierra. Es un sistema que rota con la Tierra, por lo que no es un sistema inercial. Se define de forma que el eje de rotación de la Tierra es el eje Z del sistema, y el eje X está contenido en el plano del ecuador y cruza la superficie de la Tierra en el meridiano de Greenwich. El eje Y completa el sistema de forma que $X \times Y = Z$. Este sistema se llama en SPICE IAU_EARTH; también existe un IAU_MARS, IAU_VENUS, ...



Vamos a calcular la posición ahora en un sistema mucho más interesante para nosotros, el sistema *IAU* para la tierra, que está definido de forma que el eje Z es el eje de rotación de la tierra, pero el eje X cruza el meridiano cero. Por tanto, este sistema gira con la tierra

En el siguiente programa podemos ver que simplemente necesitamos cargar un nuevo kernel (que define el sistema que vamos a usar), y decirle a SPICE que nos dé la posición en el nuevo sistema. El kernel lo puedes encontrar aquí. *(Consulta la tabla de recursos al final del artículo si tu enlace no funciona)*

PROGRAMA 4: Calculando la posición de la luna

```
#include <stdio.h>
#include <SpiceUsr.h>
#include "print_tools.h"

int main() {

  SpiceDouble posicion[ 3 ];
  SpiceDouble lt;
  SpiceDouble et;
  SpiceDouble r, lon, lat;

  /*
  * Cargamos los kernels que necesitamos para
  * nuestros calculos
  */
  furnsh_c( "NAIF0009.TLS" );
  furnsh_c( "DE405.BSP" );
  furnsh_c( "PCK00009.TPC" );

  /* Convertimos tiempo de formato UTC a ET */
  utc2et_c( "2010-06-01T12:00:00", &et );

  /* Calculamos la posicion de la luna */
  spkpos_c( "moon", et, "iau_earth", "none",
            "earth", posicion, &lt );

  printf( "Posicion de la luna: " );
  printVector( stdout, posicion, 3 );
  printf( " Km\n" );

  /*
  * Calculamos la posicion en coordenadas
  * esfericas
  */
  reclat_c( posicion, &r, &lon, &lat );

  /* Longitud de radianes a grados*/
  lon *= dpr_c();
  /* Latitud de radianes a grados*/
  lat *= dpr_c();

  printf( "Distancia: %f Km\n", r );
  printf( "Longitud:  %.2f grados\n", lon );
  printf( "Latitud:   %.2f grados\n", lat );

  return 0;

}
```

Cambiar nuestro Sistema de Referencia es muy fácil. Solo tenemos que decirle a SPICE en que sistema queremos las soluciones

Al final del programa, podemos ver por qué este nuevo sistema de referencia es útil. Debido a cómo está definido, si convertimos un vector dado en este sistema a coordenadas esféricas (ρ, θ, ϕ), θ y ϕ se corresponden a la longitud y latitud del punto en que el vector cruza la superficie de la tierra. Por tanto, en nuestro ejemplo podemos calcular el punto de la tierra que está "debajo" de la luna.

La función *reclat_c* simplemente convierte un vector en sus coordenadas esféricas, aunque las dos coordenadas angulares son devueltas en radianes, por lo que hay que convertirlas a grados; esto es muy fácil en SPICE, simplemente tienes que multiplicar la cantidad por *dpr_c()*.

SPICE nos permite calcular la posición de cualquier cuerpo para el que exista un kernel

SPICE no sólo calcula posiciones de cuerpos naturales del sistema solar, sino de cualquier cuerpo para el que haya un kernel con efemérides. En este ejemplo, calcularemos la posición de Rosetta visto desde la Tierra en un instante de tiempo dado.

PROGRAMA 5: Calculando la posición de Rosetta

```
#include <stdio.h>

#include <SpiceUsr.h>

#include "print_tools.h"

int main() {

  SpiceDouble posicion[ 3 ];
  SpiceChar utc[ 50 ];
  SpiceDouble lt;
  SpiceDouble et;

  /*
  * Cargamos los kernels que necesitamos
  * para nuestros calculos
  */
  furnsh_c( "NAIF0009.TLS" );
  furnsh_c( "DE405.BSP" );
  furnsh_c( "PCK00009.TPC" );
  furnsh_c( "ORHR_____________00122.BSP" );

  /* Leemos el tiempo de la consola */
  printf( "Introduce el tiempo en formato "
  "YYYY-MM-DDThh:mm:ss: " );
  scanf( "%s", utc );

  /* Convertimos tiempo de formato UTC a ET */
  utc2et_c( utc, &et );

  /* Calculamos la posicion de Rosetta */
  spkpos_c( "rosetta", et, "iau_earth", "none",
  "earth", posicion, &lt );

  printf( "Posicion de Rosetta: " );
  printVector( stdout, posicion, 3 );
  printf( " Km\n" );

  return 0;

}
```

Como ves, la única diferencia con el programa anterior es que necesitamos cargar el kernel con efemérides de Rosetta

EN LA SIGUIENTE ENTREGA

Aprenderemos algo más sobre los kernels, usaremos la base de datos de efemérides de la nasa para hacer nuestros propios kernels de posición, y aprenderemos a definir un sistema de referencia referido a nuestra localización en la tierra. ■

MIDIENDO EL TIEMPO EN SPICE

No es tan fácil como puede parecer medir el tiempo. Según la teoría de la relatividad, depende del observador, por lo que el tiempo medido en Marte puede no ser el mismo que el medido en Venus o en la Tierra. Además, los sistemas que se usan en la vida diaria para medir el tiempo (en años, meses, dias, horas,...) no son los más adecuados para hacer operaciones.
En SPICE normalmente se trabaja con dos sistemas de tiempo fundamentales, UTC y ET.
La definición de ET (Ephemeris Time) es muy complicada. ET es la variable temporal en las ecuaciones diferenciales que describen el movimiento de los planetas en el sistema solar. Puede aparecer en dos formas, TDB (Barycentric Dynamical Time) y TDT (Terrestrial Dynamical Time). TDB describe el tiempo medido desde el baricentro (el centro de gravedad) del sistema solar, y TDT el tiempo medido en la Tierra. La diferencia entre ellos, aunque pequeña, cambia con el tiempo, en una época del año TDB va más rápido que TDT, y en otra es al revés. Por lo tanto, la diferencia entre TDB y TDT nunca es mayor que una determinada cantidad muy pequeña (del orden de una milésima de segundo).
SPICE toma TDB como ET, y se expresa como el número de segundos que han transcurrido desde un instante determinado (la época J2000, que es el uno de enero de 2000 a las 11 horas 58 minutos y 55.816 segundos). Este es el sistema de tiempo que usa SPICE para todos sus cálculos.
UTC, por otro lado, no es un sistema de medida del tiempo propiamente dicho, sino una forma de asignar un nombre a cada segundo ET. UTC intenta mantenerse en sincronía con la rotación de la Tierra, de forma que el 1 de enero de cualquier año el Sol esté en la misma posición. Si todos los años tuvieran la misma duración en segundos, esto no se cumpliría, puesto que la velocidad de rotación de la Tierra varía con el tiempo (típicamente se va ralentizando). El IERS (International Earth Rotation Service) se encarga de mantener la sincronía, que se hace intercalando un "leapsecond" (segundo intercalar), normalmente a finales de junio o diciembre, si es necesario. Este leapsecond puede ser positivo o negativo, dependiendo de si se añade o se sustrae de la cuenta de tiempo.
Por ejemplo, si la rotación de la Tierra se ralentiza, el efecto es que el año se alarga. Para corregir, se añade un leapsecond al final de un año, y la cuenta de segundos se correspondería con los siguientes momentos UTC:

```
... DECEMBER 31 23:59:57
... DECEMBER 31 23:59:58
... DECEMBER 31 23:59:59
... DECEMBER 31 23:59:60
... JANUARY 1 00:00:00
...
```

, en lugar de la cuenta normal:

```
... DECEMBER 31 23:59:57
... DECEMBER 31 23:59:58
... DECEMBER 31 23:59:59
... JANUARY 1 00:00:00
...
```

Si la rotación de la Tierra se acelerase (no se ha dado nunca desde que el sistema de leapseconds se introdujo en 1972), el leapsecond se sustrae (un leapsecond negativo), y la cuenta sería por el contrario:

```
... DECEMBER 31 23:59:57
... DECEMBER 31 23:59:58
... JANUARY 1 00:00:00
...
```

SPICE proporciona varias funciones para convertir entre diferentes sistemas e tiempo. Por ejemplo, *utc2et_c* permite convertir de UTC a ET, y *et2utc_c* de ET a UTC. Toda la información que maneja SPICE está dada en ET, por lo que estas operaciones son muy comunes, para traducir el tiempo entre el sistema intuitivo que entendemos las personas y el sistema más matemático que usa SPICE. El hecho de que haya que llevar la cuenta de leapseconds que han ocurrido históricamente para poder hacer la conversión es la razón de que tengamos que cargar en nuestro programa un "leapseconds kernel", que almacena precisamente esa información. Sólo hay un kernel de leapseconds oficial que se actualiza cada vez que aparece un nuevo leapsecond; a día de hoy, el kernel es el NAIF0009.TLS, y el próximo será el NAIF0010.TLS.
En la siguiente entrega veremos un tercer sistema para medir el tiempo, usado por los satélites para registrar los eventos que ocurren a bordo.
En el cuadro sobre sistemas de referencia hemos definido el sistema J2000, pero nos faltaba una pieza. El eje de rotación de la Tierra no está fijo, sino que sufre un efecto que se llama precesión (se mueve como una peonza), de forma que, por ejemplo, dentro de unos años (más bien milenios) la estrella polar se habrá desviado tanto del polo norte geográfico que ya no será útil para marcar ese punto cardinal de la Tierra. Esto significa que, tal como lo hemos definido, J2000, se movería con el tiempo y ya no sería inercial. La forma de evitar esto es referir las posiciones que definen el sistema a la época J2000. Por ejemplo, el eje Z se define como el eje de rotación medio de la Tierra en la época J2000.

RECURSOS

GENERAL

Página Oficial del *Navigation and Ancillary Information Facility* NAIF:
Página Principal: http://naif.jpl.nasa.gov
Descarga CSPICE: http://naif.jpl.nasa.gov/naif/toolkit_C.html

KERNELS NASA y ESA

Misiones de la NASA:
ftp://naif.jpl.nasa.gov/pub/naif

Misiones de la ESA:
FTP: ftp://ssols01.esac.esa.int/pub/data/SPICE
Interfaz Web: http://www.rssd.esa.int/index.php?project=SPICE&page=RepositoryBrowser

KERNELS UTILIZADOS EN EL ARTÍCULO:

Efemérides Sistema Solar:
ftp://ssols01.esac.esa.int/pub/data/SPICE/MEX/kernels/spk/DE405.BSP

Conversiones tiempo UTC/ET:
ftp://ssols01.esac.esa.int/pub/data/SPICE/MEX/kernels/lsk/NAIF0009.TLS

Kernel que define diferentes constantes para el sistema solar, entre ellas los parámetros que describen el sistema IAU para la tierra:
ftp://ssols01.esac.esa.int/pub/data/SPICE/MEX/kernels/pck/PCK00009.TPC

Efemérides de Rosetta:
ftp://ssols01.esac.esa.int/pub/data/SPICE/ROSETTA/kernels/spk/ORHR_____________00122.BSP

Numquam ponenda est pluralitas sine necessitate.

Quaestiones et decisiones in quattuor libros Sententiarum Petri Lombardi (ed. Lugd., 1495), i, dist. 27, qu. 2, K

Teclados Personalizados

Constrúyelos fácilmente con Teensy

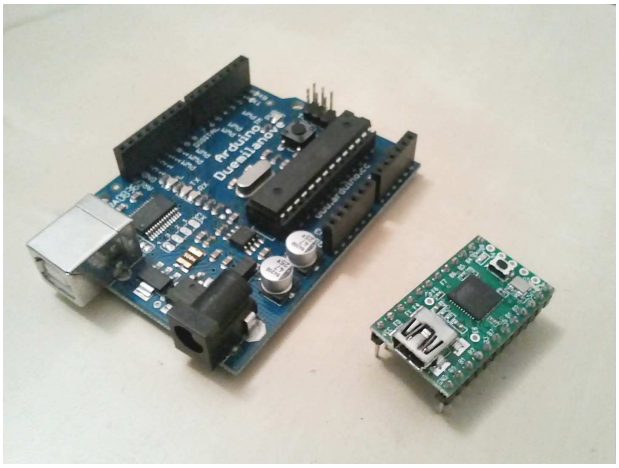
por er Teo Cla D'Or

Teensy es un completo sistema de desarrollo para microcontroladores basado en un interfaz USB. Un simple cable USB es todo lo que necesitamos para empezar a utilizarlo y sus posibilidades son muy interesantes, como veremos en este artículo.

En el momento de publicar este artículo (finales de 2011), la versión actual de esta pequeña placa es la 2.0. Podemos encontrarla en dos configuraciones denominadas, respectivamente, Teensy 2.0 y Teensy++ 2.0. Las diferencias fundamentales entre ambas versiones las podéis comprobar en la siguiente tabla, extraída de la web del proyecto.

Elemento	Teensy 2.0	Teensy++ 2.0
Procesador	ATMEGA32U4	AT90USB1286
Memoria Flash	32256	130048
Memoria RAM	2560	8192
EEPROM	1024	4096
I/O	25	46
Entradas Analógicas	12	8
PWM	7	9
UART,I2C,SPI	1,1,1	1,1,1
Precio	$ 16	$ 24

Además, la versión ++, es más larga, unos 5 cm frente a los 3cm del Teensy 2.0. Nosotros vamos a trabajar con el Teensy 2.0, más que suficiente para lo que os queremos contar. Para que os hagáis una idea del tamaño de esta pequeña placa, aquí lo podéis ver junto a un Arduino.



La versión de la foto es un Teensy 2.0 con los "pines" ya soldados. Podéis comprarlo así, bastante práctico para utilizarlo en placas de prototipos o sin los pines, de forma que podréis soldar los que mejor os convengan.

Con un Teensy y unos cuantos botones podremos extender nuestro teclado muy fácilmente

MÁS CARACTERÍSTICAS

Además de lo que ya hemos comentado, su bajo precio, su pequeño tamaño y unas especificaciones suficientes para abordar varios proyectos de distinta naturaleza, hay dos cosas más que hacen esta placa bastante interesante.

La primera es que utilizando un Teensy es prácticamente trivial construir dispositivos de entrada para nuestros ordenadores, tales como teclados, ratones o joysticks. Esto lo veremos en seguida.

La otra característica interesante de esta placa se llama Teensyduino. Se trata de un paquete que extiende el entorno de desarrollo de Arduino, para trabajar con Teensy. Además de ellos, este paquete incluye la mayoría de las librerías a las que estamos acostumbrados con Arduino, y un sinfín de ejemplos.

Para terminar con esta introducción, solo comentar que en la web del proyecto podréis encontrar los esquemas de los circuitos de todas las versiones de Teensy y una librería para la utilidad Eagle con la que desarrollar vuestras propias placas, basadas en este diseño.

SOFTWARE

Para empezar a trabajar con nuestro pequeño amigo, necesitamos descargar el paquete llamado "Teensy Loader" de la página web del proyecto.

La aplicación está disponible para los principales sistemas operativos. Existe también una versión de línea de comandos que podéis compilar vosotros mismos si así lo preferís. Sólo tenéis que descargarla y ejecutarla.

El paquete Teensyduino nos permite programar Teensy desde el entorno Arduino

La aplicación "Teensy Loader" nos permite reprogramar la tarjeta, en otras palabras, volcar el código que deseemos ejecutar en el microcontrolador. Para generar ese código necesitamos un "toolchain".

En el caso de los sistemas GNU/Linux, este no es otro que el proporcionado por GCC para AVR. Compilador, linker y librería estándar están disponibles en la mayoría de los sistemas GNU/Linux. Si estáis utilizando otro sistema, consultad la página del proyecto donde encontraréis instrucciones detalladas para los distintos sistemas soportados.

Por tratarse este de un artículo introductorio, no vamos a profundizar en este tema. Nos centraremos en el entorno Teensyduino más sencillo y familiar. Sin embargo, no os olvidéis de que al final, lo que escribimos en el editor del entorno de Arduino, acaba siendo compilador por gcc y descargado en Teensy por el "Teensy Loader".

La web de Teensy contiene instrucciones detalladas sobre como trabajar directamente utilizando estas herramientas, para aquellos de vosotros que queráis sacar el máximo de ella. Para los demás vamos a instalar Teensyduino.

TEENSYDUINO

La instalación de Teensyduino no puede ser más fácil. Solo tenemos que descargar el paquete adecuado para nuestro sistema y ejecutarlo. Una aplicación gráfica realizará todo el trabajo. Lo único que nos preguntará es donde tenemos instalado el entorno de desarrollo de Arduino.

El paquete Teensyduino, no funciona con la versión del software de Arduino que de distribuye con Ubuntu. En este caso tenemos que descargar el software de la web de Arduino, y descomprimirlo donde más rabia nos dé. Luego solo tendremos que decirle a Teensyduino cual es ese lugar.

Una vez hecho esto solo tenemos que lanzar `arduino` y seleccionar en el menú `Tools >Board` nuestra versión de Teensy. Ahora comprobemos que todo funciona correctamente.

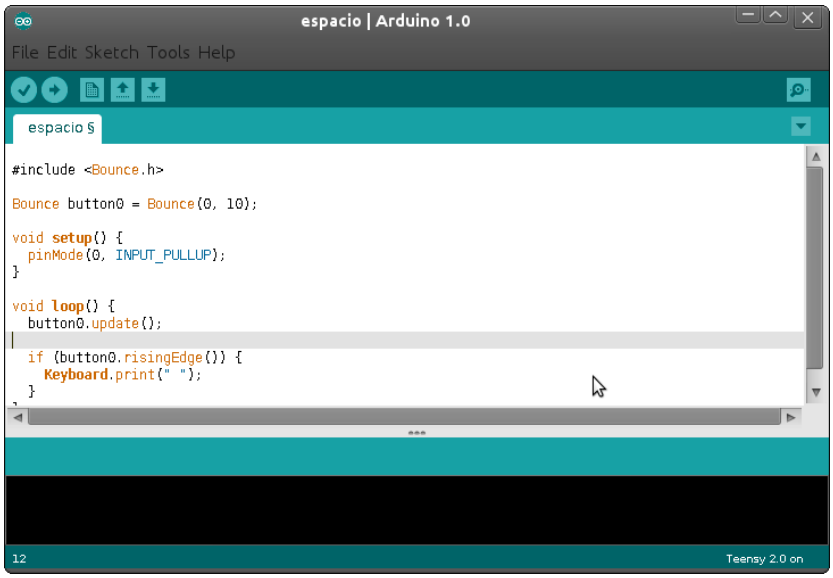
En el menú `File >Examples >Teensy >Tutotial 1` cargaremos el *Sketch* llamado `Blink` (es el único que aparece). Al principio del sketch nos encontramos una variable que indica el pin por defecto asociado al LED en Teensy. Para Teensy 2.0 es el 11 (el que aparece en el código). Si estamos utilizando un Teensy++, deberemos modificar esa línea para utilizar el pin 6.

```
const int ledPin = 11; // Teensy
const int ledPin = 6; // Teensy++
```

Ahora solo tenemos que pulsar el botón `Upload` y el programa se compilará y se descargará en la tarjeta. El "Teensy Loader" aparecerá en pantalla y nos informará del estado de la tarjeta así como del proceso de descarga del software en el microcontrolador.

Si Teensy está ejecutando ya un programa, el interfaz nos informará de que debemos pulsar el pequeño botón en la tarjeta para que el nuevo programa se descargue. Lo hacemos y el pequeño LED naranja sobre Teensy comenzará a cambiar su estado cada segundo.

Tras comprobar que todo funciona perfectamente, vamos a intentar hacer algo más útil con nuestro nuevo juguete.



Interfaz Teensiduino.

DISPOSITIVOS DE INTERACCIÓN HUMANA (HID)

Con nuestra configuración actual, podremos utilizar nuestro Teensy para hacer casi cualquier cosa que podamos hacer con nuestro Arduino, pero lo que Teensy hace realmente bien es simular dispositivos HID.

HID (*Human Interface Device* o Dispositivo de Interfaz Humano) es un protocolo estándar utilizado por dispositivos diseñados para la interacción entre hombres y máquinas. La mayoría de teclados, ratones, joysticks, etc. actuales son dispositivos de este tipo.

Teensy tiene la capacidad de presentarse ante el ordenador al que se conecta como uno de estos dispositivos. Teensy además nos permite enviarle información al ordenador utilizando este mismo protocolo y por lo tanto, puede comportarse como cualquiera de esos dispositivos. O dicho de otra forma, utilizando Teensy podemos crear, muy fácilmente, teclados, ratones o joysticks (entre otras cosas).

TECLADOS DE UNA TECLA

Vamos a empezar por lo más sencillo, un teclado de una sola tecla. Aunque pueda parecer un poco inútil, hay un gran número de aplicaciones en la que nos puede interesar ofrecer a nuestros usuarios solo un número reducido de teclas para interactuar con nuestro sistema, en lugar de un teclado completo. Pensad por ejemplo en un kiosko electrónico en un lugar público.

Como veréis enseguida, quien dice una tecla, dice dos, o tres o cuatro. Todo dependerá de cuantos botones queramos conectar a nuestras entradas digitales.

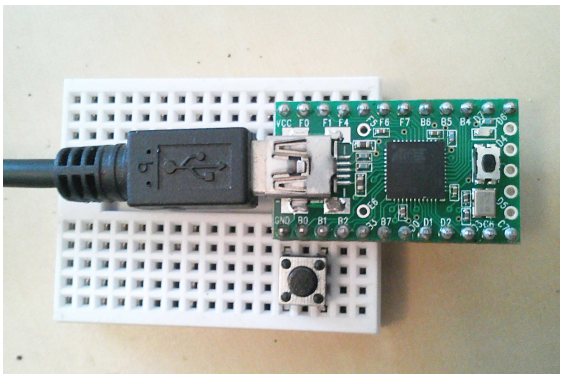
Nuestro primer programa será un simple simulador de barra espaciadora.

```
#include <Bounce.h>

Bounce button0 = Bounce(1, 10);

void setup() {
  pinMode(1, INPUT_PULLUP);
}

void loop() {
  button0.update();

  if (button0.risingEdge()) {
    Keyboard.print(" ");
  }
}
```

Aún siendo un programa muy sencillo, es necesario hacer un par de comentarios. El primero es el uso de la librería `Bounce`. Esta librería simplifica el uso de pulsadores. Normalmente, este tipo de botones, produce un rebote de la señal. Cuando pulsamos el botón, no obtenemos un único pulso, sino una secuencia de ellos que, si no tomamos medidas serían interpretados como varias pulsaciones muy rápidas.

Este problema se puede solucionar con un sencillo circuito *anti-rebotes* o en el software con una librería como `Bounce`. El uso de la librería es muy sencillo. Solamente tenemos que crear un objeto `Bounce` indicándole que pin queremos filtrar, y que intervalo de tiempo en milisegundos queremos considerar para el filtrado de los *rebotes*.

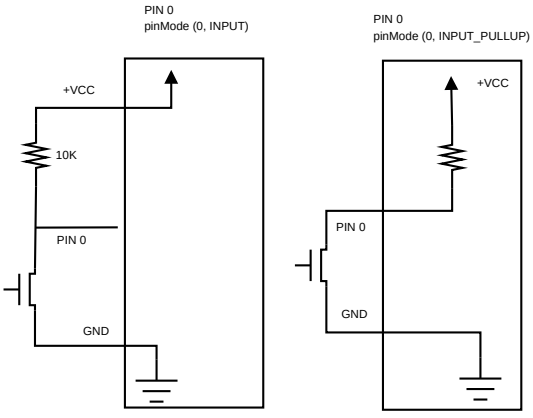
Una vez hecho esto, solo tenemos que llamar al método update en nuestro bucle principal, ya que esta librería no utiliza interrupciones y, por lo tanto, nosotros tenemos que forzar la lectura de los pines.



Los objetos `Bounce` nos ofrecen varios métodos, pero los más importantes para nuestra aplicación son `fallingEdge` y `risingEdge` que nos permiten comprobar si se ha producido un flanco de bajada o de subida respectivamente. En otras palabras, nos indican si hemos pulsado la tecla o la hemos soltado.

En nuestro caso estamos enviando el carácter espacio al ordenador cuando liberamos la tecla.

El otro comentario que debemos hacer sobre este programa es una extensión de Teensy al entorno Arduino. En la función `setup`, configuramos el pin 0 de la placa como una entrada, pero además le estamos pidiendo que se configure en modo *pull-up* de forma que leeremos un valor 1 si nada está conectado, y un valor cero, cuando conectemos la entrada a tierra, es decir cuando pulsemos el botón.



Para que este programa funcione, tenemos que configurar la placa en modo teclado. Para ello en el menú `Tools >USB Type` deberemos seleccionar uno de los tipos que incluye el teclado. Ahora solo tenemos que conectar nuestro pulsador a los pines indicados en la figura anterior y ya tenemos nuestra barra espaciadora personalizada.

TECLADOS SIN TECLAS. SEGURIDAD FÍSICA

Acabamos de ver como simular un teclado de una sola tecla, pero ¿qué os parecería un teclado sin teclas?. Puede parecer un poco inútil, pero que sucede si solo queremos escribir algo muy concreto y una sola vez... como por ejemplo una puerta trasera para tener acceso a un ordenador más tarde.

Hacer lo que acabamos de decir es un poco más complicado que lo que os vamos a contar a continuación, pero no mucho más. Hay algunas personas ahí fuera que ya lo han hecho, pero si alguno os animáis no dudéis en contarnos vuestros resultados.

Nosotros os vamos a mostrar una prueba de concepto. En lugar de un caballo de Troya vamos a generar un sencillo "Hola Mundo" y vamos a suponer que una consola ya se encuentra abierta y lista para ser utilizada.

Además vamos a mantener nuestro pulsador. La razón es que, para depurar el programa y hacer pruebas es más práctico que conectar y desconectar la placa todo el tiempo. En cualquier caso es recomendable tener cuidado cuando juguéis con el siguiente programa, es fácil escribir algo inapropiado en el lugar incorrecto y que las cosas vayan mal.

Sin más, aquí tenéis el programa que lanza vim, escribe el programa "Hola Mundo", sale de vim, lo compila y luego lo ejecuta :)

```
#include <Bounce.h>

Bounce button0 = Bounce(0, 10);

void setup() {
  pinMode(0, INPUT_PULLUP);
}

void loop() {
  button0.update();

  if (button2.risingEdge()) {
    Keyboard.println("vim kk.c\n");
    Keyboard.println(":\%d\ni#include <stdio.h>"
    Keyboard.println("int main()");
    Keyboard.println("{printf (\"hello\\n\");}");
    Keyboard.set_modifier(0);
    Keyboard.set_key1(KEY_ESC);
    Keyboard.send_now();
    Keyboard.println (":wq\n");
    delay (100);
    Keyboard.println ("make kk\n./kk\n");
  }
}
```

Como podéis comprobar, el programa hace uso extensivo de la función `println` que nos permite enviar una cadena de texto seguida de un retorno de carro. En el momento de soltar el pulsador, simplemente enviamos todo el texto que escribiríamos en el caso normal.

Hay una pequeña salvedad. La tecla ESC para volver a modo comando en vim.

TECLAS ESPECIALES Y COMBINACIONES

Teclas como ESC o F1 no las podemos enviar utilizando el método `print` que conocemos. De la misma forma, combinaciones de teclas como `CTRL+C` tampoco se pueden simular con esas funciones. En ese caso tenemos que construir nuestra combinación de teclas a mano.

Podemos almacenar un programa en nuestro Teensy y hacer que se escriba y compile automáticamente al ser conectado.

Para estos casos, disponemos de los métodos `set_modifier` y `set_keyN`. El primero nos permite configurar los modificadores de teclado, esto es, las teclas CTRL, SHIFT, ALT y GUI (las teclas de Windows y MAC).

La segunda nos permite configurar hasta seis teclas que se enviarán simultáneamente cuando ejecutemos el método `send_now`. En nuestro ejemplo anterior podemos ver como lo primero que hacemos es borrar los modificadores para, a continuación, configurar la pulsación de la tecla ESC y finalmente enviar todo junto.

Esto nos devuelve al modo comando de vim, lo que nos permite grabar el fichero en disco con el bien conocido comando `:wq`.

Observad que tras enviar el comando esperamos un poco. En nuestra primera prueba sin el retardo, el carácter `m` del comando `make kk` se perdía, al parecer porque el ordenador estaba ocupado grabando el fichero en disco. Si experimentáis un problema similar probad a aumentar el retardo.

MUCHO MÁS QUE TECLADOS

Aunque no hemos hablado de ello, para mantener el artículo corto, Teensy puede ser utilizado para construir ratones o joysticks de una forma tan sencilla como la que acabamos de ver para nuestros teclados especiales. Estad atentos a la lista de correo.

Las posibilidades son muchas. desde un panel de botones para nuestro ordenador de salón (hay un interesante sketch en los ejemplos de Teensyduino), hasta un botón `CTRL+ALT+SUPR` para reiniciar servidores.

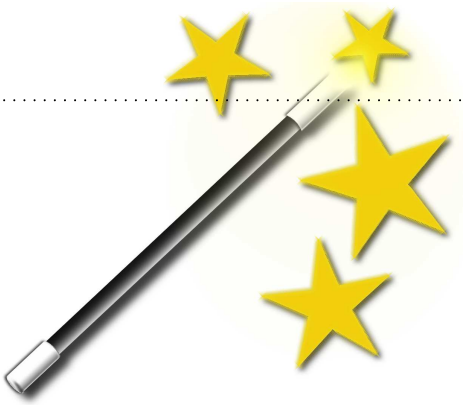
Esperamos vuestros experimentos. ■

RECURSOS

Página Web de Teensy
http://www.pjrc.com/teensy/index.html

Todo sobre Teensiduino
http://www.pjrc.com/teensy/teensyduino.html

Algunas ideas de las posibilidades
http://www.pjrc.com/teensy/projects.html



Con un par... de líneas

Chuletillas para hacer cosas mú rápido

por Tamariz el de la Perdíz

SERVIDOR WEB IMPROVISADO

Python es un lenguaje de programación que se ha popularizado mucho en los últimos años. Como consecuencia de ello una enorme cantidad de módulos están disponibles para hacer nuestras vidas mucho más cómodas.

Uno de estos módulos es `SimpleHTTPServer` que nos permite desarrollar sencillos servidores web de forma muy sencilla. De forma tan sencilla como esta:

```
$ python -m SimpleHTTPServer
```

Ahora solo tenemos que poner nuestros ficheros HTML en el mismo directorio en el que hemos ejecutado nuestro comando, y apuntar nuestro servidor a `localhost:8000`, o a la IP de nuestra máquina si pretendemos acceder de forma remota.

Por defecto el módulo escucha en el puerto 8000. Si queremos utilizar un puerto diferente solo tenemos que añadirlo al final del comando.

```
$ python -m SimpleHTTPServer 8080
```

Un módulo muy útil para sacarnos de un aprieto "mú rápido"

EL HOMBRE

Existen dos flags de la utilidad `man` que resultan muy útiles en el día a día. El primero es `-k`. Este flag le dice a `man` que busque cosas relacionadas con la palabra clave que se pasa como parámetro. El flag `-f` es similar pero permite buscar cadenas.

El otro es `N` donde N es el número de la sección del manual que queremos consultar. Para los que no lo sepáis estas son las secciones del manual:

- 1 Comandos Generales
- 2 Llamadas al Sistemas y Números de Error
- 3 Librería C
- 3p perl
- 4 Dispositivos y Drivers de Dispositivos
- 5 Formatos de ficheros y Ficheros de Configuración
- 6 Instrucciones de Juegos
- 7 Miscelánea
- 8 Mantenimiento del Sistema
- 9 Interioridades del Kernel

Algunos ejemplos:

```
$ man -k thread
$ man socket
$ man 7 socket
```

Otras páginas del manual interesantes son:

```
$ man 7 tcp
$ man 7 ip
$ man syscalls
$ man standards
$ man feature_test_macros
$ man ascii
```

La utilidad `man` tienen un montón de flags interesantes, no tenéis más que pedir la página del manual, sobre el manual...

```
$ man man
```

DISCO SUPERRÁPIDO /dev/shm

El dispositivo `/dev/shm` es realmente memoria, y por lo tanto muy útil para almacenar información a la que queremos acceder "mú rápido". Y para muestra un botón.

```
$ dd if=/dev/zero of=/dev/shm/test count=102400 bs=1024
102400+0 records in
102400+0 records out
104857600 bytes (105 MB) copied, 0.105547 s, 993 MB/s
$ dd if=/dev/zero of=/tmp/test count=102400 bs=1024
102400+0 records in
102400+0 records out
104857600 bytes (105 MB) copied, 0.629624 s, 167 MB/s
```

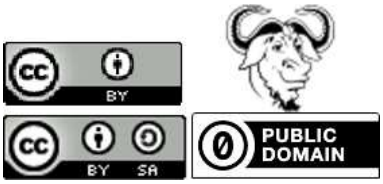
EDITAR COMANDOS BASH SUPER LARGOS

Bash tiene un comando interno llamado `fc` que permite editar el último comando tecleado en un editor. Cuando salimos del editor, lo que hayamos escrito en el editor se ejecutará

Envía tus trucos

Puedes enviarnos esos trucos que usas a diario para compartirlos con el resto de lectores a la dirección:

`occams-razor@uvigo.es`



LICENCIAS DE LAS IMÁGENES EN ESTE NÚMERO

Gracias a todos los que comparten su trabajo!

por The Occam's Razor Team

Todas las imágenes de esta publicación, salvo las excepciones listadas en esta página y las imágenes que forman parte de los artículos se distribuyen bajo los siguientes términos:

Licencia: CC-BY (Creative Commons Attribution 3.0)

Autor/Creditos: Laura I. Rodríguez González/Revista Occam's Razor

Los términos anteriormente mencionados se aplican a las secciones generales de la revista: Editorial, La Comunidad de Occam y Trucos.

PORTADA

Imagen del satelite SMOS utilizada de fondo

Licencia: ESA (http://www.esa.int/esa-mmg/mmgdownload.pl)

Autor/Creditos: ESA

CIENCIA Y ESPACIO

Todas las imágenes en este artículo pertenecen han sido cedidas por ESA o CESBIO. Consulte el pie de foto de cada una de ellas.

Licencia: ESA (http://www.esa.int/esa-mmg/mmgdownload.pl)

Autor/Creditos: ESA/CESBIO, ESA/CNES

EN LA PRÁCTICA

Imagen de Fondo Cabecera

Licencia: ESA (http://www.esa.int/esa-mmg/mmgdownload.pl)

Autor/Creditos: ESA

HISTORIA

Figuras 1, 2 y 4

Licencia: CC-BY (Creative Commons Attibution 3.0)

Autor/Creditos: Fernando Martín y Xulio Fernández

Figuras 3, 5, 6, a1, a2, a3, 10

Licencia: CC-BY (Creative Commons Attibution 3.0)

Autor/Creditos: Fernando Martín

Figuras 7 y 8

Licencia: Dominio Público

Autor/Creditos: Wikipedia

Figura 10

Licencia: Creative Commons Attribution-Share Alike 3.0 Unported

Autor/Creditos: Ed g2s (Fuente Wikipedia)

Figura 10

Licencia: Licencia de arte libre (http://artlibre.org/licence/lal/es)

Autor/Creditos: Jari Laamanen (Fuente Wikipedia)

RATAS DE BIBLIOTECA, MÚ RÁPIDO, LA CACHARRERÍA

Licencia: CC-BY (Creative Commons Attibution 3.0)

Autor/Creditos: Laura I. Rodríguez González/Occam's Razor

OCCAM'S RAZOR

Porque lo más sencillo es lo más probable

Descarga Todos los Números en http://webs.uvigo.es/occams-razor