UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Interacción por reconocimiento de objetos con OpenCV: reconocimiento de caras

Apellidos, nombre	Agustí i Melchor, Manuel (magusti@disca.upv.es)
Departamento	Departamento de Informática de Sistemas y Computadores (DISCA)
Centro	Escola Tècnica Superior d'Enginyeria Informàtica Universitat Politècnica de València

1 Resumen de las ideas clave

El presente documento muestra cómo se puede interactuar entre el entorno externo al computador, que se ve a través de una cámara digital (mundo real) y el escritorio que genera el computador (mundo virtual) y que es el que vemos en la pantalla. **Se plantea construir una aplicación que funcione como un espejo de mano**, esto es, en la pantalla se mostrará la imagen que capta la cámara, reflejada especularmente sobre un objeto virtual. Este objeto se acerca o aleja a la pantalla, proporcionalmente a la distancia del usuario a la cámara.

Para ello se hará uso de las bibliotecas de funciones OpenCV [1] y OpenGL [2] y para ver cómo se puede pasar la información de una a la otra se revisará el tutorial de D. Millán [3] que muestra cómo tratar las imágenes que gestiona OpenCV como texturas en OpenGL.

No es objetivo de este documento revisar el código fuente (que se puede obtener a vuelta de un correo electrónico dirigido al autor de este documento) hasta sus mínimos detalles, sino mostrar un ejemplo de uso de técnicas multimedia factibles en computadores actuales que permiten realizar el análisis de una escena visual. El lector centra así su atención en la exploración del diseño de una posible solución al problema que se plantea. Aprenderá cómo se adaptan y enlazan diferentes ejemplos existentes. El lector interesado puede participar modificando, posteriormente, la solución que se ofrece a sus necesidades.

2 Objetivos

La estrategia para llevar a cabo el tema propuesto consiste en desarrollar la idea inicial en una serie de pasos más simples que puedan ser abordados y probados de forma independiente. Estos son los objetivos de este documento y, una vez que el lector haya explorado los contenidos relacionados con la aplicación que aquí se diseña, será capaz de:

- Utilizar las funciones de OpenCV para tomar imágenes de una cámara digital (p. ej., una cámara web con conexión USB) y, así, seguir las acciones del usuario delante del computador.
- Identificar la operativa de OpenCV para que la aplicación sea capaz de detectar la existencia de una cara de una persona en la imagen RGB obtenida de una cámara, en tiempo de ejecución.
- Integrar la imagen capturada en una escena 3D generada sintéticamente con OpenGL. Para ello se simulará el efecto que se observa al mirarse en un espejo de mano, mapeando la imagen a una “textura” del “cristal”.
  Esto es, el tamaño de la imagen reflejada en la escena debe ocupar un espacio acorde a la cercanía del rostro a la cámara. Si hay más de una cara detectadas, la más “cercana” a la cámara deberá guiar el proceso.. Para ello es necesario.

La cercanía del usuario se determinará en base a cuan grande, respecto al tamaño de la imagen que proporcione la cámara, sea el área en que se inscribe la cara. De este manera, el usuario, indicará a la aplicación qué grande ha de mostrar un objeto virtual en una escena, imitando el gesto diario de acercarse a una cosa que quiere ver con más detalle.

La detección de caras es uno de los posibles objetos del mundo real que el usuario y la máquina pueden nombrar, buscar en una imagen e identificar su presencia. Constituye un ejemplo de **detección de objetos** semánticamente

significativos, como podría ser la detección de los ojos, la mano o cualquier otro concepto que se pueda describir porqué tiene una **morfología característica** muy evidente entre las posibles variantes de ese "objeto".

3 Introducción

Este documento se vertebra alrededor de un ejemplo de integración de medios: el usuario interactuará con el computador en una escena tridimensional a través de objetos externos al propio sistema y el computador hará uso de las técnicas de Visión por Computador (en adelante VxC) para la entrada de datos y, para la salida, utilizará técnicas de síntesis de imagen para obtener una escena tridimensional.

Los tradicionales sistemas de entrada de información, constituidos por el teclado y el ratón, generan eventos hacia el computador y permiten al usuario la interacción con la máquina. Hablamos en este caso de lo que podríamos denominar **eventos visuales,** como concepto que permite ampliar el abanico de estímulos a los que puede reaccionar un computador. En este caso, se usará la información que proporciona una cámara digital y que ha de ser analizada para detectar la presencia o no de un determinado contenido.

Al usuario se le va a exponer una interfaz al estilo de un espejo que se aleja o acerca a la pantalla, de manera proporcional a como se detecta, de cercana o lejana, la cara de un usuario respecto a la cámara. Se utilizará OpenCV [1] para el procesado de imagen encaminado a la detección de caras y OpenGL [2] para la generación de la imagen 3D. Vamos a poner en una misma aplicación ambas a colaborar, para obtener un sistema que ofrezca una extensión del área de trabajo del computador a su entorno físico inmediato. Se mostrará información visual en una escena tridimensional, para dotarla de mayor realismo.

La fig. 1 muestra el diagrama de bloques del funcionamiento del ejemplo *facedetec* que acompaña a la distribución de OpenCV. Este ejemplo remarca con un círculo la presencia de "caras" en la imagen examinada. Para ello utiliza un "modelo" genérico de cara humana y examina la imagen que se le presenta para generar una lista ordenada, por tamaño decreciente, de los rostros encontrados. Existen diferentes modelos para diferentes poses de la cara a buscar, el algoritmo intentará encontrar la que se le indique como parámetro, con independencia de su tamaño. Aunque los autores de los modelos han procurado que su fiabilidad sea alta, no se puede asumir que sea totalmente preciso: se pueden detectar caras en dibujos hechos a mano alzada sobre una hoja de papel o impresos en una prenda de ropa.

Veamos cómo está construido este ejemplo que utilizaremos como **introducción a la detección de caras**. En lo que sigue nos ceñiremos a la versión 2.3.1 de OpenCV. No entraremos en detalles internos de las técnicas utilizadas, pero se puede encontrar la información de las mismas en [4] y [5]

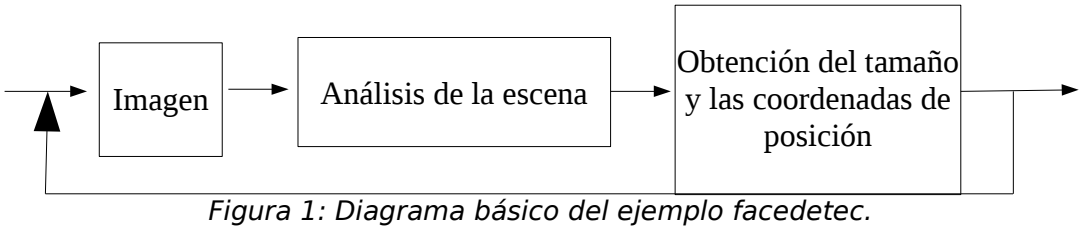


Figura 1: Diagrama básico del ejemplo facedetec.

El ejemplo (*facedetect*[1]) disponible en la distribución de OpenCV utilizada viene reescrito en C++, anteriormente estaba escrito en C. Los cambios no son trascendentes en cuanto al uso de las técnicas base por lo que nos limitaremos a explicar el funcionamiento para dar a entender cómo se puede utilizar y cómo se puede integrar en nuestra aplicación.

```
$ gcc facedetect.cpp -o facedetect `pkg-config opencv --cflags --libs`
$ facedetect --cascade=data/haarcascades/haarcascade_frontalface_default.xml
$ facedetect --cascade="../../data/haarcascades/haarcascade_frontalface_alt.xml"
        --nested-cascade="../../data/haarcascades/haarcascade_eye.xml" --scale=1.3
$ facedetect --cascade=data/lbpcascades/lbpcascade_frontalface.xml –nested=data/haarcascades/haarcascade_m
```

Listado 1: Generación y uso del ejemplo facedetect.

Lo compilaremos, véase la primera orden del listado 1 y lo ejecutaremos invocando al binario creado con, al menos, un parámetro que indica el tipo de objeto a buscar, véase la segunda y tercera orden del listado 1. Estos parámetros son ficheros que contienen descripciones genéricas de objetos y ahí reside el interés de este ejemplo: se le pueden pasar modelos de descripciones de objetos y, en cada caso, buscará el objeto descrito por ellos. Esto es, lo ejecutamos con información de clasificadores[2] previamente desarrollados para un tipo de objeto. Así que se dedicará a buscar caras (cualquiera de los *haarcascade_*face* *) o caras y ojos (*haarcascade_*eye*.xml*), como muestra el listado 1. También podría ser cualquier otro de los que pudiéramos crear o de los que vienen ya preparados, para la detección de:

- Cuerpo, como *haarcascade_fullbody.xml. haarcascade_upperbody.xml, haarcascade_lowerbody.xml* o *haarcascade_mcs_upperbody.xml*
- Cara, como *haarcascade_frontalface_default.xml, haarcascade_frontalface_alt.xml, haarcascade_frontalface_alt2.xml, haarcascade_frontalface_alt_tree.xml o haarcascade_profileface.xml*
- Ojos, como *haarcascade_eye.xml, haarcascade_righteye_2splits.xml, haarcascade_mcs_lefteye.xml haarcascade_lefteye_2splits.xml, haarcascade_eye_tree_eyeglasses.xml, haarcascade_mcs_eyepair_small.xml, haarcascade_mcs_righteye.xml* o *haarcascade_mcs_eyepair_big.xml.*
- Oidos, como *haarcascade_mcs_rightear.xml o haarcascade_mcs_leftear.xml.*
- Boca, con *haarcascade_mcs_mouth.xml.*
- Nariz, con *haarcascade_mcs_nose.xml*

Actualmente podemos encontrar dos tipos de clasificadores[3], los basados en características de Haar, como los mencionados, y los que usan LBP (véase los trabajos de [4] y [5] para ampliar la información al respecto de estas técnicas), como el de *lbpcascade_frontalface.xml*. Así que se puede utilizar un clasificador

[1]En mi caso en */usr/share/doc/opencv-doc/examples/c/*, de donde me hago una copia en mi espacio de usuario.

[2]En mi caso, en */usr/share/opencv/haarcascades/* y en */usr/share/opencv/lbpcascades*.

[3]En la distribución utilizada, se encuentran en */usr/share/opencv/haarcascades/* y en /usr/share/opencv/lbpcascades.

u otro para la detección de caras y, por el momento, los de Haar para el resto de partes del cuerpo disponibles, véase la última orden del listado 1.

A lo que nos interesa, asumiendo que tiene el ejemplo delante para estudiarlo, ¿todavía no lo tiene? Espero ... ¿lo tiene ya? ¡Sigamos!. ¿Cómo funciona? Si examina el programa principal podrá ver que:

- En primer lugar, analiza la línea de órdenes para determinar los parámetros con que se le ha llamado. De ahí obtendrá las rutas para los clasificadores que ha de ejecutar. El programa no asume uno en particular, sino que buscará lo que se le indique en los posibles argumentos siguiendo el formato indicado en el listado 2. Siendo *cascade_path* el que describe el principal tipo de objeto a buscar; *nested-cascade* otro posible tipo de objeto (diferente del primario a buscar y para el que se restringe el espacio de búsqueda a lo que ocupa el primero) a buscar; *scale* un valor para reducir la escala de la imagen y así acelerar el proceso de búsqueda y, por si no se quiere utilizar la secuencia de vídeo que proporciona la cámara por defecto, un nombre de fichero (*filename*) o el número de cámara a utilizar (*camera_index*).

```
facedetect [--cascade=<cascade_path> this is the primary trained classifier such as frontal face]
  [--nested-cascade[=nested_cascade_path this an optional secondary classifier such as eyes]]
  [--scale=<image scale] [filename|camera_index]
```

Listado 2: Formato de la línea de órdenes para el ejemplo facedetect.

- En segundo lugar, se carga el/los clasificador/es indicados y se obtiene la imagen a procesar de dónde haya indicado el usuario.
- A partir de ahí, el tercer paso, se ejecuta la llamada a la función *detectAndDraw*. Este es el centro de todo el proceso, por lo que nos vamos a detener un poco a analizarla con detalle enseguida.
- El programa termina cerrando las ventanas que haya abierto y liberando los recursos que ha utilizado durante su ejecución.

```
...
  Mat gray, smallImg( cvRound (img.rows/scale), cvRound(img.cols/scale), CV_8UC1 );
  cvtColor( img, gray, CV_BGR2GRAY );
  resize( gray, smallImg, smallImg.size(), 0, 0, INTER_LINEAR );
  equalizeHist( smallImg, smallImg );
  cascade.detectMultiScale( smallImg, faces, 1.1, 2, 0 |CV_HAAR_SCALE_IMAGE,  Size(30, 30) );
 ...
  for( vector<Rect>::const_iterator r = faces.begin(); r != faces.end(); r++, i++ )
  {
…
    center.x = cvRound((r->x + r->width*0.5)*scale);        center.y = cvRound((r->y + r->height*0.5)*scale);
    radius = cvRound((r->width + r->height)*0.25*scale);        circle( img, center, radius, color, 3, 8, 0 );
…
  }
...
```

Listado 3: Normalización de la imagen y primera búsqueda de objetos.

¿Cómo determina este ejemplo el tamaño y la posición del tipo de objeto que se le ha indicado buscar? El grueso del proceso está en una función que se encarga de detectar cuántos objetos del tipo primario encuentre y los marca en

pantalla con un círculo, sobre una versión normalizada (convertida a escala de grises) y opcionalmente de tamaño menor que la imagen de partida. Véase listado 3 donde se ha destacado la parte de código encargado de esta misión, obviando los detalles menores que tendrá ya a su vista …, si tiene el código del ejemplo delante.

Dentro de cada uno de ellos, si se ha indicado un segundo tipo de objeto a buscar, se vuelve a lanzar la búsqueda sólo en el área de la imagen ocupada por el rectángulo (o *bounding box*) que contiene al primer objeto, con su propio clasificador, pero siempre con la misma función de búsqueda: *nestedCascade*.*detectMultiScale*. Del resultado devuelto por esta función se obtiene la información para dimensionar un círculo circunscrito a este área y pintarlo alrededor de cada objeto encontrado.

Sin pérdida de generalidad, si se le pide que detecte caras en una imagen, esta función de búsqueda devolverá una lista de las que ha encontrado ordenadas de mayor a menor tamaño del rectángulo que las contiene. Por lo que devuelve una estructura de datos (un vector) que, así ordenado, guarda para cada objeto encontrado, la posición en coordenadas de la imagen y el ancho y alto del rectángulo centrado en esa posición que delimita la extensión del objeto. Tomando esta información como referencia, cada área es analizada como una escena en la que buscar el segundo tipo de objeto, listado 4.

```
...
    for( vector<Rect>::const_iterator r = faces.begin(); r != faces.end(); r++, i++ )
    {
     ...
        center.x = cvRound((r->x + r->width*0.5)*scale);        center.y = cvRound((r->y + r->height*0.5)*scale);
        radius = cvRound((r->width + r->height)*0.25*scale);        circle( img, center, radius, color, 3, 8, 0 );
...
        smallImgROI = smallImg(*r);
        nestedCascade.detectMultiScale( smallImgROI, nestedObjects, 1.1, 2, 0 | CV_HAAR_SCALE_IMAGE, Size(30, 30) );
        for( vector<Rect>::const_iterator nr = nestedObjects.begin(); nr != nestedObjects.end(); nr++ )
        {
            center.x = cvRound((r->x + nr->x + nr->width*0.5)*scale);
            center.y = cvRound((r->y + nr->y + nr->height*0.5)*scale);
            radius = cvRound((nr->width + nr->height)*0.25*scale);          circle( img, center, radius, color, 3, 8, 0 );
        }
    }
```

Listado 4: Búsqueda del segundo tipo de objetos.

4 Desarrollo

Abordemos ahora el **diseño y generación del ejecutable.** La aplicación es un caso de integración de medios en el que la escena que “ve” el computador es analizada para tomar la acción de salida, que es la generación de una escena tridimensional que verá el usuario y a la que se incorpora la imagen (del mundo real) obtenida de la cámara digital. El usuario interacciona con la escena al moverse delante de la cámara y el sistema responde en función de la distancia del usuario al computador, variando el contenido de la escena: la imagen de la cámara proyectada sobre un espejo simulado.

Ahora que hemos examinado las piezas que vamos a utilizar como base, se va a proponer una posible secuenciación de las mismas de forma que se establezca la relación buscada entre la entrada y la salida: la imagen digital capturada del mundo real y la escena generada por el computador, respectivamente.

4.1 Diseño de la aplicación

Para implementar los objetivos propuestos se va basar la solución en el diagrama de bloques que muestra la fig. 2. En ella podemos ver las dos partes diferenciadas que hemos venido exponiendo: la generación de una escena tridimensional que muestra una imagen en vivo (tiempo real) obtenida de una cámara digital y, en paralelo, un proceso de análisis de la imagen que permita determinar la posición del objeto móvil (la cara del usuario).

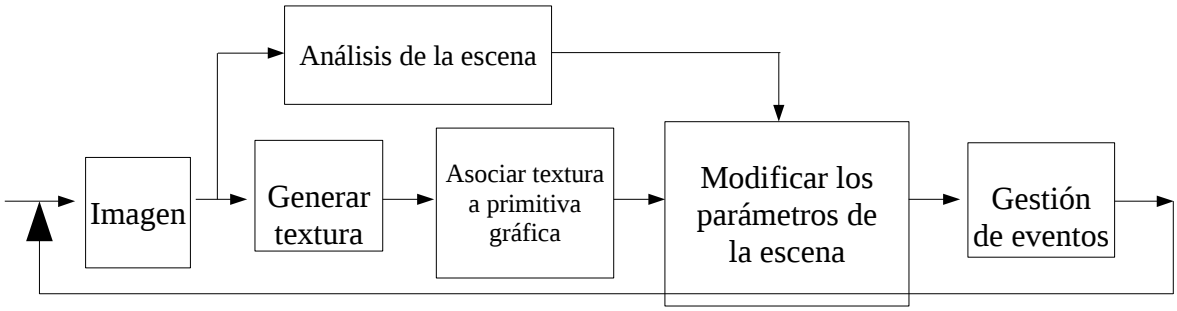


Figura 2: Diagrama básico de la aplicación a desarrollar.

Para implementar los objetivos propuestos se va basar la solución en las acciones siguientes:

1. Tomaremos el código de [3] como base para la generación de la escena. Ahora la escena ya no consiste en el plano que gira continuamente, sino que ahora se ha de mover el plano en el sentido perpendicular a la cámara, para reflejar que el usuario se acerca o aleja. En lugar de rotar el plano pintado por la función *plane*, simplemente trasladaremos su posición a lo largo del eje Z.
   Asociado al mismo, la textura de la imagen, ha de ir actualizándose conforme avance la ejecución del proceso. Para ello recuperaremos la función *loadTexture_Ipl* del tutorial [3] y le pasaremos como parámetro la imagen que toma la cámara conectada al computador. Así, en lugar de ejecutarla una única vez en la función principal, será necesario integrarla en la generación de la escena llamándola con la última captura disponible de la cámara como parámetro.
   Por lo que será necesario inicializar el sistema de captura en la función principal y posponer la toma de imagen a este momento. Estas funciones las recuperamos de lo expuesto en el apartado 3 “Introducción“. Y, cuando sea pertinente, cambiaremos de posición el plano.
2. Esto viene determinado por el tamaño de la mayor de las caras que ve el computador. La detección de caras es proporcionada por OpenCV; a partir de ella obtenemos la medida de distancia. No es esta una media calibrada, esto es, dimensionada en algún sistema métrico en el sistema de referencia del mundo real. Es una estimación que se obtendrá como la relación entre el tamaño de la cara más grande detectada y el de la imagen que obtiene la cámara. Sólo es necesario ahora extraer la información de la primera cara de la lista que devuelve *nestedCascade.detectMultiScale*, por lo que será

necesario reformular la función *detectAndDraw* e incorporarla a la generación de la escena, dentro de la función *display. Ambas se han expuesto en el apartado* 3 "Introducción".
Será necesario simplificar el contenido de *detectAndDraw*, puesto que ahora sólo necesitamos que analice la imagen y devuelva, si la hay, los parámetros de la cara. más grande que encuentre. Así que eliminaremos todo lo referente a pintar en la imagen donde se encuentra la cara. Esta información podría ser interesante para depurar la ejecución de la aplicación.

3. La imagen capturada por OpenCV hay que transformarla en una textura de OpenGL para situarla sobre el objeto que hace de espejo en la escena sintética. Para ello recuperaremos la función *loadTexture_Ipl* del apartado 3 "Introducción" y la integraremos en la generación de la escena.

4.2 Implementación

Como se proponía en la fig. 2, es necesario introducir la detección de caras a cada nueva imagen que se toma para, al mismo tiempo que se está actualizando la imagen en la escena 3D se sitúe esta en la posición más o menos cerca de la cámara de la escena sintética como lo está de la cámara digital en el mundo real.

Para ello, tomaremos la función *detectAndDraw* del ejemplo de *facedetec* (véase punto. 3 "Introducción") y la utilizaremos como base para nuestra función *analizarProporcioCara*. Esta será la encargada de, si se ha detectado una cara, obtener las propiedades de la primera para establecer un valor de distancia que sea función de lo que ocupa la cara en la imagen que se obtiene de la cámara, con un valor normalizado entre 0 y 100. Tal que cuanto más cerca esté la persona, más pequeña debe ser la distancia devuelta y al revés. El ratio y la distancia son inversamente proporcionales, véase en el listado 5 para los pasos clave señalados en negrita y con un tamaño de letra ligeramente mayor.

```
float analizarProporcioCara( IplImage* img, float anteriorDistancia )
…
    if ( faces->total > 0 )
    {
      r = (CvRect*)cvGetSeqElem( faces, 0);
... pt1.x = r->x*scale;  pt2.x = (r->x+r->width)*scale; pt1.y = r->y*scale; pt2.y = (r->y+r->height)*scale;
      ancho = abs( pt2.x-pt1.x); alto = abs(pt2.y-pt1.y);
      area= (float)alto * (float)ancho;
       proporcioTamany = ((area/fAreaMax)*100.0);
       novaDistancia = 100.0 - proporcioTamany;
```

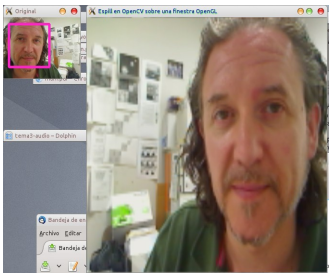
Listado 5: Modificación de la función original de detección de caras.

Esta información de distancia es actualizada cada vez que se obtiene una nueva imagen de la cámara y se construye la imagen, así que la función *display* debe cambiar ligeramente, como se muestra en el listado 6. Hago notar los cambios con la letra en negrita. En primer lugar hay que considerar si se tiene imagen de la cámara, en cuyo caso se puede pasar a construir la textura y a analizar la distancia. En segundo lugar hay que cambiar el movimiento de la cámara de la escena sintética (*gluLookAt*) por el del plano en la misma (*glTranslate*), acorde a la variación de la posición del usuario respecto a al cámara digital.

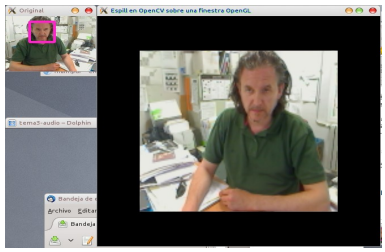
```
void display( void)
 if (cargarCuadroDeCamara() == 0)
 {
  loadTexture_Ipl(imageCamara, &texture );
  distanciaUsuari = analizarProporcioCara( imageCamara, distanciaUsuari ); // En rango 0..100
 }
...
   //gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glTranslatef(0, 0, -distanciaUsuari);
   glEnable( GL_TEXTURE_2D );
   plane();
...
```

Listado 6: Modificación de la función de visualización (display).

De esta forma el usuario puede ver como su imagen en la escena 3D se hace más grande si se acerca a la cámara, fig. 3a. O bien que si se va alejando, se hace más pequeña, fig. 3b. Esta figura muestra a la izquierda y arriba recuadrada la cara encontrada en la imagen (OpenCV) que captura la cámara digital. A la derecha de ellas, la ventana de la escena OpenGL donde se observa el efecto del plano que se hace de espejo y que se aleja si lo hace el usuario.



a)



b)

Figura 3: Captura de la aplicación en funcionamiento: a) de cerca y b) alejándose.

Es interesante, aunque lo hemos dejado fuera por brevedad en la exposición, considerar un rango diferente de distancia. Ya que en cámaras con una resolución baja se pueden dar “saltos” importantes y porque puede que la cara más grande no ocupe al 100% la imagen. También, si la luz ambiente no es estable pueden darse casos de desaparición momentánea de la cara del usuario o la detección de otra. Para solucionar este caso se recomienda suavizar estas variaciones de presencia/ausencia puntual de la cara detectada y ser un poco conservadores.

Una cosa importante: en el programa principal, hay que inicializar la captura de imágenes desde la cámara digital y, al terminar, hay que liberar los recursos correspondientes.

4.3 Construcción del ejecutable y distribución

Para construir el ejecutable será necesario compilar el código fuente con las cabeceras de las bibliotecas de funciones que implementan tanto el API de OpenCV como el OpenGL y enlazar los ficheros objetos con las librerías correspondientes.

Básicamente, nuestra aplicación utiliza unas librerías que ha indicado con la inclusión de los ficheros de cabecera para compilación. Podemos agruparlas en tres grupos: las propias del lenguaje C (*stdio* y *cvtype*), las de OpenGL (*glut* y *gl*) y las de OpenCV (*cv* y *highgui*). Con la ayuda de la orden *dpkg* averiguamos qué paquetes hay que utilizar en nuestro sistema. El listado 7 recoge las órdenes que ejecutamos para averiguar los paquetes que es necesario tener preguntando por cosas relacionadas con cada uno de los ficheros de cabecera.

```
$ dpkg -l "*glut*"

$ dpkg -l "*gl*"|less

$ dpkg -l "*opencv*"


gcc espill__OpenCV_OpenGL.c -o espill__OpenCV_OpenGL -g `pkg-config --cflags opencv glu gl` -lglut `pkg-config --libs opencv glu gl`
```

Listado 7: Paquetes relacionados con las bibliotecas de funciones de OpenGL (GLUT y GL) y OpenCV.

Para generar la línea de compilación haremos uso de la orden *pkg-config*. Esta orden nos facilita saber qué parámetros pasar al compilador y al enlazador, entre otras cosas. Esa información la escribe el que desarrolla la librería por lo que es bastante fiable.

5 Conclusión

Una vez que el lector haya explorado los contenidos relacionados con la aplicación que aquí se diseña, será capaz de:

- Capturar vídeos, secuencias de imágenes, a través de una cámara digital.
- Identificar objetos semánticamente significativos en la escena a través del uso de clasificadores.
- Dar una respuesta basada en eventos visuales, como mecanismo que activa la respuesta de la aplicación.

Como continuación, dado que el mecanismo es el mismo, animo al lector a experimentar con otros tipos de objetos; en lugar de las caras se puede jugar con la detección de ojos (u otras partes de la cara), manos, personas, etc.

6 Bibliografía

[1] Open Computer Vision Library URL <http://sourceforge.net/projects/opencvlibrary/>. Consultada el 17 de marzo de 2015.

[2] Introduction to OpenGL. OpenGL Programming, Guide. Capítulo 1 . URL: <http://www.glprogramming.com/red/>. Consultada el 17 de marzo de 2015.

[3] D. Millán. (2008). OpenCV & OpenGL. URL <http://blog.damiles.com/2008/10/opencv-opengl/>. Consultada el 17 de marzo de 2015.

[4] P. Viola and M. Jones. (2001). Rapid object detection using a boosted cascade of simple features. Proc. of CVPR, pages 511-518. URL <http://docs.opencv.org/master/d7/d8b/tutorial_py_face_detection.html>. Consultada el 17 de marzo de 2015.

[5] T. Ojala y M. Pietikainen. (2002). Multiresolution Gray-Scale and Rotation Invariant Texture Classification with Local Binary Patterns, IEEE Trans. on PAMI, Vol. 7.