INGENIERÍA TÉCNICA DE INFORMÁTICA DE GESTIÓN

PROYECTO FINAL DE CARRERA

MakeUP *Stylist*

Tutor

Manuel Agustí- Melchor

Alumnos

Nedim Sabic

Silvia Sahuquillo Falaguera

Oscar Fortuny Elvira

Valencia, 27 de Julio 2012

PROYECTO FINAL DE CARRERA:

MAKEUP *STYLIST*

CONTENIDO

1-. INTRODUCCIÓN

Este proyecto, es una aplicación de escritorio que simula los efectos que se producen al aplicar un determinado maquillaje, o un complemento como podrían ser unas gafas, sobre un rostro. Por eso hemos llamado a este proyecto, MakeUP Stylist, algo así como estilista de maquillaje.

Lo hemos desarrollado entre tres alumnos, que hemos estudiado tres intensificaciones diferentes: Ingeniería del software, Tecnologías y servicios para web y Multimedia, por lo que cada uno de nosotros, hemos tratado de poner en práctica todos estos conocimientos en nuestro proyecto. Aunque podemos decir que todos hemos colaborado en todas las partes del proyecto, Nedim ha participado en el diseño general de la interfaz y estructura del proyecto, pintado con GDI+ en los elementos: sombra de ojos, colorete y labios, además de desarrollar la parte del gestor de productos y almacenamiento XML. Oscar ha desarrollado la parte de la interfaz de la captura de imágenes con la cámara, galería de imágenes de accesorios, estrategia de pintado en capas y la parte de guardar el resultado bajo la extensión .mkup. Por último Silvia ha desarrollado la estrategia de detección en la carga de la imagen, pintado de accesorios, pintado de la piel, boca y raya de ojos.

Sobre las características técnicas de este proyecto, hemos de decir que está programado en C#, con un diseño de interfaces, basado en la tecnología WPF. Para detectar las caras a maquillar o sus elementos, ojos y boca, hemos empleado una potente librería de visión por computador llamada OpenCv, en su versión .Net: Emgu. Utiliza un estilo de programación basado en el modelo 3 capas y un patrón de diseño llamado Model-View-ViewModel. Más adelante, explicaremos cómo hemos aplicado todos estos conceptos y tecnologías de programación a nuestro proyecto.

A groso modo, este programa funciona de la siguiente forma, realiza una captura de de imágenes empleando una cámara web o una imagen cargada del sistema, procesa la información obtenida buscando los elementos de la cara y muestra una serie de respuestas y/o soluciones basadas en el maquillaje.

En un principio, planteamos la posibilidad de que nuestra aplicación funcionase en modo video. Pero tras hacer algunas pruebas, vimos que era prácticamente imposible hacer todo el

proceso en un tiempo razonable para obtener el resultado del maquillaje y mostrarlo manteniendo la ilusión óptica del vídeo, al menos, con los medios de los que disponíamos.

Para la localización de la cara y sus elementos hemos utilizado unos algoritmos, llamados algoritmos Haar. Son el producto de analizar múltiples imágenes que contienen un determinado objeto. Este producto, se almacena en unos fichemos XML que posteriormente lee nuestro programa a fin de obtener la detección de un objeto. También se les conoce como clasificadores, ya que obtienen "entrenando", procesando esas imágenes, y cuantas más imágenes se les proporciona mejores resultados ofrecen en cuanto a detección de ese objeto, dentro de una imagen cualquiera que lo contenga.

Este proceso, teniendo en cuenta su complejidad, es realmente ágil, pero sólo es una parte de la detección. Con detección de los algoritmos Haar, obtenemos una serie de rectángulos marcando el objeto a encontrar, el problema es que en muchas ocasiones marca varias veces un mismo objeto. Este ha sido uno de nuestros trabajos, elegir cual de las caras obtenidas era la elegida para maquillar, comprobando que esta cara contenía 2 ojos en la parte media superior y una boca en la inferior. De esta forma, no tendremos que analizar todos los píxeles de la imagen. Nos centraremos únicamente en los píxeles que componen la cara o los elementos que sean de nuestro interés y descartar el resto, para hacer esta localización de una forma eficiente.

Una vez localizados todos los elementos de nuestro interés, ya podemos comenzar a proceder con nuestro objetivo de maquillar virtualmente a nuestro usuario empleando las técnicas más adecuadas al tipo de maquillaje que se seleccione.

2-. ESTRATEGIAS Y TÉCNICAS DE DETECCIÓN Y PROCESAMIENTO DE IMAGEN

2.1-. INTRODUCCIÓN

El proceso de detección comienza cada vez que se carga la foto seleccionada.

En primer lugar buscamos la localización de la cara, para ello hacemos valer el algoritmo de detección Haar proporcionado por la plataforma de detección. En la clase FaceDetector, encontramos un algoritmo con el mismo nombre, facedetector al cual le pasamos por parámetros la imagen a maquillar cargada en un objeto de tipo Bitmap. El algoritmo Haar nos devolverá el conjunto de rectángulos que encuadran las caras detectadas en la imagen que proporcionamos, entre los cuales escogeremos el primero que contenga en su interior los dos ojos y la boca lo cual significará con mayor exactitud que se trata de una cara, ya que los algoritmos de detección son realmente buenos en su trabajo, pero no infalibles.

Comentamos que nuestro algoritmo facedetector escoge el rectángulo que contiene los 2 ojos y la boca, pero… ¿cómo hace este proceso? Muy sencillo, llama a los respectivos algoritmos de detección de las clases detectoras de los ojos y la boca, y comprueba que devuelven el rectángulo que los contiene. Este mecanismo lo hace mediante hilos de ejecución para que resulte más eficiente. A la misma vez que llama a los algoritmos de detección llama al algoritmo que detecta la piel de la cara. Estos tres métodos son los más costosos de la parte de detección por eso los emparejamos para que se ejecuten simultáneamente lanzando tres hilos de ejecución. Cuando termina el hilo que detecta la boca devolviendo el rectángulo que la localiza invocamos al método que detecta el contorno de los labios.

Aparte, ha sido muy útil realizar un estudio de las proporciones de un rostro, *(Imagen 2.1)* porque muchos problemas de detección y/o de eficiencia han sido resueltos empleando estos métodos. La ventaja de estos conocimientos, radica en que se puede obtener un resultado aproximado, con una probabilidad de éxito muy alta, en unos milisegundos. Esto es así, porque se realizan operaciones matemáticas muy sencillas, que apenas consumen recursos

del procesador y no hacen un estudio del píxel que es lo que de alguna manera encarece el coste de detección.

Teniendo en cuenta este estudio, hemos enriquecido nuestro algoritmo de detección, ayudando a los algoritmos de detección Haar, a encontrar los ojos y la boca con mayor eficiencia. Fruto de esta proporcionalidad y simetría, les hemos indicado el rectángulo que describe la zona de la cara detectada dónde deberían encontrarse estos objetos. De esta forma, hemos reducido el área de búsqueda dónde los algoritmos tendrían que realizar el trabajo.

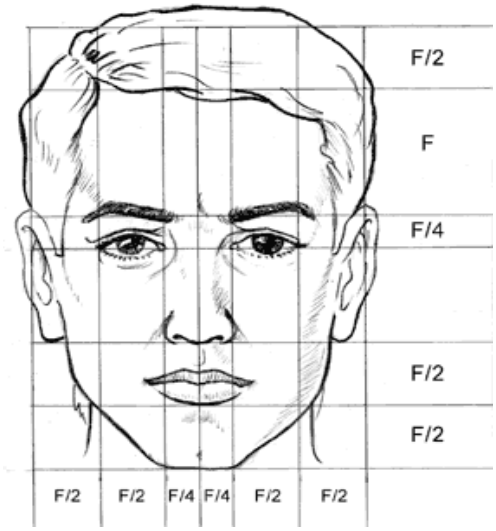


Imagen 2.1 Proporciones de la cara
Siendo F el ancho de la frente

fuente:
http://bitacoraricardomiranda.blogspot.com.es/2011/08/investigacion-2da-experiencia-de.html

Cuando terminamos esta fase de detección comprobamos que todas las variables contienen los objetos que requeríamos. De ser así asignamos a la variable que almacena el área del rectángulo de la cara, el número entero máximo para que no tenga en cuenta el resto de rectángulos detectados por el Haar de la cara. Ya que otra de las condiciones que ponemos en este algoritmo es que siempre busque un rostro en un rectángulo mayor. Esto es así porque entendemos que la cara mayor será la más cercana a la cámara por lo que será la más

interesante a maquillar. También calculamos el coste del cómputo total de la detección a fin de crear estadísticas sobre lo que cuesta realizar este proceso.

Este método de detección sólo es invocado una única vez por cada imagen que cargamos y naturalmente, se invoca nada más terminar la carga, o si la hemos obtenido mediante la webcam, cuando la seleccionamos.

Al iniciar el proyecto barajamos la posibilidad de hacer todo este proceso en formato de video, capturando las imágenes de la webcam, detectando y maquillando por cada frame. Como hemos explicado hemos hecho un cálculo del coste del proceso de detección. En el mejor de los casos hemos registrado un coste de 0.7 segundos aproximadamente y en el peor hasta 3 y 4 segundos. Este coste temporal es demasiado elevado para presentar un formato de video ya que no conseguiríamos esta ilusión óptica. El retardo entre frame y frame es demasiado elevado, teniendo en cuenta que para ello deberíamos obtener entre 24 y 30 fps (Frames o fotogramas por segundo) procesados, en nuestro mejor caso no llegaría a 2. Por ello descartamos la idea y nos centramos en trabajar con imágenes estáticas.

2.2-. DETECCIÓN Y MAQUILLADO DE LA CARA

2.2.1-. INTRODUCCIÓN

La fase de detección de la cara, es una de las más importantes nuestro proyecto, ya que toda la funcionanidad del proyecto recae sobre el resultado obtenido en esta fase. Es así, porque en esta parte inicial del proyecto, obtendremos la zona de interés de la imagen a tratar y de hacer una buena detección depende el éxito del proceso. Por ello como hemos explicado antes es fundamental asegurarnos de que tras esta zona de interés hay rostro fotografiado frontalmente.

Para ello sometemos a cada resultado de la detección a un control muy riguroso respecto a la localización de todos los elementos que contiene un rostro frontal así como a la cromática de los píxeles que lo forman.

Esto implica, que por lógica, al aplicar el algoritmo de detección de los ojos sobre esta región de interés, debemos obtener al menos 2 resultados en la zona media superior de la región, situados a la misma altura, uno en el cuarto superior izquierdo y otro en el cuarto superior derecho de la zona de interés.

De la misma forma, para la boca buscamos que se localice en la zona central de la parte media inferior de la zona de interés.

Si todo esto se cumple, entonces, podremos afirmar que el rectángulo, objeto de la detección contiene una cara, y por tanto podemos considerarlo que define nuestra zona de interés.

2.2.2-. DETECCIÓN DE LA PIEL PARA APLICAR UNA BASE DE MAQUILLAJE.

Para realizar un maquillado base de la cara, antes debemos detectar todas aquellas zonas con piel dentro de la cara, en la introducción de este apartado ya vimos como conseguir la región de interés de la cara y es aquí donde retomamos.

Con la región de interés localizada, buscamos en ella los píxeles que contengan piel, para ello hacemos un análisis del RGB de los mismos. Aquellos que superen una serie de condiciones, por las cuales podemos decir que son piel, los pintamos sobre un objeto de tipo Bitmap de las mismas dimensiones de la imagen original y en la misma posición que ocupaban en aquella, para crear una especie de máscara.

Lo primero que hacemos es un estudio de la intensidad en escala de grises. Para ello tomamos una muestra de la parte central de nuestra región de interés y definimos una intensidad máxima y mínima en función si la cara detectada es de piel blanca o de color.

```
//piel blanca
if (acc / contador > 125)
{
      intensity = Math.Max(65, acc / contador - 40);
      intensityM = Math.Min(245, intensity + 130);            }
//piel de color
else
{
      intensity = Math.Max(25, acc / contador - 100);
      intensityM = intensity + 150;

}
```

*acc: acumulador de la intensidad

*contador: contador de los píxeles recorridos

Por otro lado, tras consultar en varias fuentes y hacer un estudio del color de la piel, definimos las siguientes condiciones del RGB de los píxeles para decir si un píxel es o no piel.

```
→ image[k, j].Red > 70

→ image[k, j].Green > 30

→ image[k, j].Blue > 10

→ Math.Max(image[k, j].Red, Math.Max(image[k, j].Green, image[k,
j].Blue)) - Math.Min(image[k, j].Red, Math.Min(image[k, j].Green,
image[k, j].Blue))) > 10

→ Math.Abs(image[k, j].Red - image[k, j].Green) > 10

→ image[k, j].Red > image[k, j].Green && image[k, j].Red >
image[k, j].Blue))
```

Para que éste algoritmo sea más eficiente aparte de centrarnos únicamente en la región de interés, tratamos de forma distinta los píxeles de la derecha y los de la izquierda y empleamos la siguiente política para el recorrido de la imagen.

Tomamos el punto central del rectángulo que describe nuestra región de interés y hacemos los recorridos desde ese punto de la siguiente forma:

- abajo-derecha

-abajo-izquierda

-arriba-derecha

-arriba-izquierda

Como apuntamos anteriormente, tratamos el lado derecho de distinta forma al izquierdo, ya que como condición de parada decimos que cuando encuentre un 4% del ancho del rectángulo que define la cara, en la misma fila sin piel finalice. En caso de recorrer hacia la derecha, asignamos el ancho de la imagen como condición de parada.

Si estamos tratando el lado izquierdo (decrementando la coordenada x) asignamos el principio de la imagen: 0.

En el caso de que no llegue a conseguir 4% de píxeles seguidos sin piel (puede que se trate de un brillo en el centro de la cara, una pequeña sombra…) estos puntos se habrán encolado en una estructura de tipo List<Point> y serán tomado como piel y pintados sobre nuestro lienzo para formar esa máscara de la que hablamos.

Esta lista recibe el nombre de **pixeles** y se vacía cuando rompemos la condición del bucle o cuando encontramos un nuevo píxel considerado como piel en esa fila, en ese caso se pintan todos los anteriores.

Podemos encontrar zonas en la cara en las que encontremos 4% de píxeles sin piel y que éstos no sean el contorno. Por ello debemos enriquecer nuestro algoritmo haciendo que cada píxel compruebe si su superior está pintado. En caso de que no lo esté comprobaremos si lo podríamos considerar como piel y en caso afirmativo pintaremos recursivamente todos los píxeles superiores no pintados y considerados como piel. Estos casos son muy frecuentes en la zona de los ojos, ya que los mismos normalmente no tienen RGB de piel hasta una longitud superior a 4% de píxeles por lo que debemos considerarlo.

En nuestra estrategia también tenemos en cuenta la intensidad de los píxeles en la imagen convertida a escala de grises con profundidad **Single**, para así descartar zonas que puedan pasar como piel y que no lo sean como es en ocasiones el pelo, los labios que no incurren en nuestro interés en esta parte de la aplicación, las cejas que pueden haber sido pintadas en el tratamiento explicado en el párrafo anterior…. *(Imagen 2.4)*

A continuación, se muestra un ejemplo gráfico del funcionamiento de este algoritmo.

Imagen 2.2: Imagen original.

Imagen 2.3: Imagen en escala de grises

Esta imagen: imaGray es de tipo Image<Gray, Single> y siendo intensidad o negro y 255 blanco.

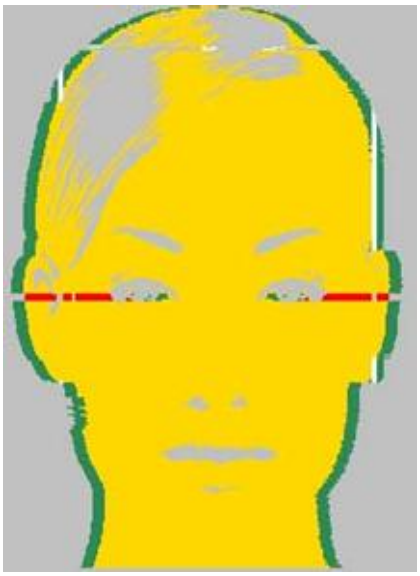


Imagen 2.4: Imagen que representa el funcionamiento del algoritmo.

VERDE: borde obtenido tras encontrar el 4% de píxeles seguidos sin piel

ROJO: partes pintadas recursivamente

AMARILLO: parte considerada piel

GRIS: partes no pintadas por no superar la condición de la intensidad

**Para aplicar el maquillaje, descartaremos las partes en verde y las partes en rojo y amarillo las pintaremos del color de maquillaje elegido.

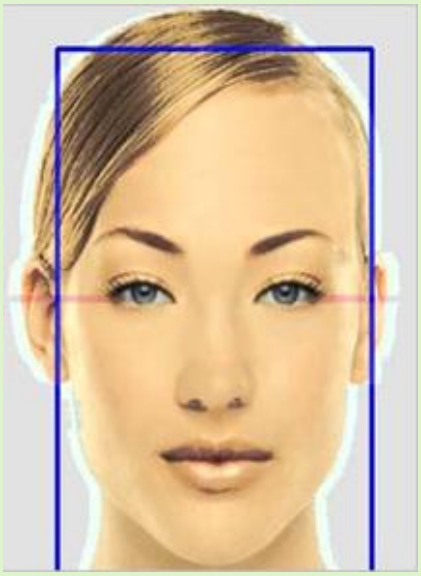


Imagen 2.5: Imagen original tras

Resultado de superponer la máscara anterior, con un factor de transparencia adecuado para que se pueda apreciar el efecto sobre la imagen original.

**El rectángulo azul es el obtenido a partir del algoritmo de detección.

Puesto que nuestro objetivo era que este algoritmo fuese capaz de detectar tanto la piel blanca como la de color, presenta algunos errores. Muchas veces, como se puede apreciar en las imágenes anteriores, toma como parte de la piel, partes del pelo.

2.2.3-. APLICAR UNA BASE DE MAQUILLAJE

Vamos a aplicar un maquillaje base, para ello, tomamos el bitmap que contiene los píxeles considerados piel, que quedó inicializado en la fase de detección. Este bitmap llamado skin se encuentra en la clase faceDetector.

Como ya habíamos explicado, este bitmap tiene las dimensiones de la imagen original, con la peculiaridad de que sólo tiene los píxeles que hemos considerado piel, tras cumplir las condiciones explicadas en el apartado anterior, y discriminando aquellos que no cumpliesen alguna de éstas.

Este bitmap es una especie de máscara y los píxeles que la componen serán los que se pintarán con el color seleccionado, y con opacidad 100. Hemos tomado este valor de opacidad tras comprobar que era el que imitaba el maquillaje de una forma más realista, respetando la sensación de volumen de la imagen original.

Esta máscara ya está preparada para visualizarse, por lo que sólo tendremos que añadirla a nuestro gestor de capas *(Véase apartado: 3.1-. Gestor de capas)* para poder visualizarla sobre la imagen original.

Y obtendremos un efecto un efecto similar como se muestra en: *(Imagen 2.6)*

Imagen 2.6: Efectos obtenidos al aplicar el maquillado base de la cara

2.3-. DETECCIÓN Y MAQUILLADO DE LOS PÓMULOS

2.3.1-. DETECCIÓN DE LA ZONA A MAQUILLAR

En este caso, detectar la zona a maquillar es relativamente fácil. No se requieren algoritmos complejos como los Haar, sino que simplemente, nos vamos a basar en las proporciones de la cara, a partir del rectángulo de la cara detectada.

Este proceso se lleva a cabo en la fase de detección inicial que hemos explicado en el apartado de introducción 2.1 y se realiza únicamente durante la carga inicial de la imagen. Para obtener los 2 rectángulos que definirán la zona donde posterior mente se pintará el colorete, hay que llamar a los métodos definidos en la clase FaceProportions. Estos rectángulos se basarán en el rectángulo de la cara en la siguiente proporcionalidad:

Ancho = 1 * Ancho cara / 4

Alto = 1 * Altura cara / 6

Según las siguientes fórmulas para definir el punto (0,0) correspondiente a la esquina superior izquierda de cada rectángulo:

Mejilla derecha:

X = P(x,0) cara + 8 * Ancho cara / 12;

Mejilla izquierda:

X = P(x,0) cara + 1 * Ancho cara / 12;

En ambos casos, la altura de los rectángulos será la misma. Ya que por la simetría en una cara frontal, las mejillas están más o menos a la misma altura.

Y = P(0,y) cara + 8 * Altura cara / 15;

*Siendo P(x,y) el punto a la esquina superior izquierda de cada rectángulo que define la cara.

2.3.2-. MAQUILLANDO: EL COLORETE

Para maquillar el colorete, emplearemos la interfaz de usuario GDI+ *(Véase capítulo 5)*, por su amplia gama de efectos gráficos.

En este caso, nos interesa un degradado, además, que tenga la forma esférica, por lo tanto, usaremos una brocha definida en GDI que defina la forma del objeto a colorear, en este caso una elipse, y otra, que rellene dicha elipse, empleando el efecto de degradado. Para ello, creamos la brocha e invocamos el método para rellenar el interior de la forma.

La clase en GDI+ que nos permite crear formas personalizadas es GraphicsPath. Nuestra forma va a tener aspecto de una elipse, con el origen en 0, 0 del mapa de bits donde se dibujará esta forma, con las dimensiones de la mejilla.

```
using (GraphicsPath cheek = new GraphicsPath())
{
       cheek.AddEllipse(0, 0, getBlushWidth(), getBlushHeight());

}
```

Ahora creamos la brocha que recibirá como parámetro la forma gráfica que acabamos de crear.

```
using (PathGradientBrush pgb = new PathGradientBrush ())
{
              pgb.CenterColor = Color.FromArgb(100, clr));
              pgb.SurroundColors = new Color[] { Color.Transparent };
              g.FillPath(pgb, cheek);
}
```

Como vemos, antes de realizar el pintado, llamando a la función FillPath, es necesario inicializar algunas de las propiedades de la brocha. La primera de ellas, CenterColor, indica el color central que tendrá el colorete. Como no queremos un color totalmente sólido usamos el método FromArgb para especificar la opacidad / transparencia (en nuestro caso el valor 100). La segunda propiedad recibe un array de colores, empezando en la frontera de la esfera y degradándose hacia el centro.

Por lo tanto en nuestro caso, la esfera se pintará desde el color transparente hacía el color del colorete.

Finalmente invocamos al método para pintar nuestra esfera pasándole la brocha y la forma a pintar. Es posible especificar la posición donde empieza y acaba cada color de degradado mediante la propiedad InterpolationColors. El código completo queda como sigue:

```
using (Bitmap cheekBitmap = new Bitmap(getBlushWidth(), getBlushHeight()))
{
	using (Graphics g = Graphics.FromImage(cheekBitmap))
	{
		using (GraphicsPath cheek = new GraphicsPath())
		{
			cheek.AddEllipse(0, 0, getBlushWidth(),
			getBlushHeight());

			using (PathGradientBrush pgb = new PathGradientBrush ())

			{

				pgb.CenterColor = Color.FromArgb(100, clr));

				pgb.SurroundColors = new
				Color[]{Color.Transparent};

				g.FillPath(pgb, cheek);

			}

		}

	}

}
```

El último paso consiste en dibujar nuestro colorete sobre la imagen original, en la posición calculada por el detector de cara.

Primero tenemos que obtener el contexto gráfico desde la imagen original. La clase CvImage de OpenCV dispone del método ToBitmap que nos permite convertir la imagen OpenCV al mapa de bits de GDI+.

```
using (Bitmap canvas = faceDetector.getDetectableImage().ToBitmap())
{
     using (Graphics gfx = Graphics.FromImage(canvas))
    {    }
}
```

Finalmente dibujamos nuestro colorete sobre la imagen. Para ello usamos el método DrawImage, pasándole el mapa de bits, y la posición X e Y donde se dibujará en la imagen original.

```
gfx.DrawImage(cheekBitmap, (m_leftCheekRect.Left + m_leftCheekRect.Width / 2)
- getBlushWidth() / 2, m_leftCheekRect.Top + getAdjustFactor());
```

El código final queda así:

```
using (Bitmap canvas = faceDetector.getDetectableImage().ToBitmap())
{
       using (Graphics gfx = Graphics.FromImage(canvas))
       {
       gfx.DrawImage(cheekBitmap, (m_leftCheekRect.Left + m_leftCheekRect.Width /
       2) - getBlushWidth() / 2, m_leftCheekRect.Top + getAdjustFactor());
        }
}
```

Imagen 2.7: Efecto obtenido al aplicar el maquillado del colorete, con 5 colores diferentes

2.4-. DETECCIÓN Y MAQUILLADO DE LOS OJOS

2.4.1-. DETECCIÓN DE LA ZONA A MAQUILLAR

Para detectar la zona de los párpados, se lanza un hilo de ejecución desde el proceso inicial de detección. Este hilo llama al método de detección de la clase EyeDetector, y empieza todo el proceso.

Primero se consulta la zona de la cara donde se encuentran los ojos, para ello hacemos uso de las proporciones. Situamos la zona a detectar en base a la siguiente fórmula: A 1/5 de la altura total de la cara y con 1/3 de altura:

Y = Alto cara / 5;

X=P(x,0) cara;

Width = Ancho cara ;

Height = * Ancho cara / 3;

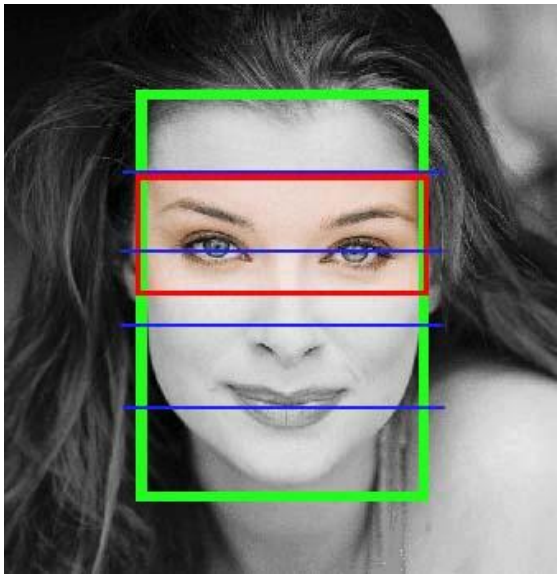


Imagen 2.8: Muestra la zona donde por simetría y por la proporcionalidad definida en la izquierda, estarán situados los ojos.

Tomaremos como zona donde se encontrará el ojo izuierdo la mitad izquierda del rectángulo anterior, y como derecho la zona derecha.

A continuación, utilizamos el algoritmo Haar sobre la zona definida anteriormente, para detectar la zona exacta donde se encontrarán ambos ojos. Todo el proceso anterior supone ganar en eficiencia y reducir la probabilidad de error, ya que el algoritmo Haar recibe un recorte de la imagen en que se encuentran los ojos de un tamaño muy inferior a la que contiene la cara. De esta forma el algoritmo nos devuelve una cantidad de rectángulos a analizar muy inferior y con resultados válidos, que si evaluaramos el algoritmo sobre la totalidad de la imagen o de la cara.

A lo largo del estudio, antes de llegar a la solución escogida, hicimos pruebas localizando los ojos sobre la totalidad del rectángulo que contiene la cara y sobre la mitad superior de éste, obteniendo en ocasiones, resultados poco apropiados o poco eficientes. Aparte teníamos que comprobar sobre los resultados obtenidos cuales eran ojos derechos o izquierdos y comprobar que estubiesen a la misma altura, teniendo en cuenta alguna desviación, lo cual, encarecía considerablemente el coste del algoritmo de detección, por lo que finalmente optamos por la solución anteriormente descrita.

2.4.2-. MAQUILLANDO: SOMBRA DE OJOS

Una vez obtenidas las posiciones, de los rectángulos obtenidos en la fase detección de los ojos, se procede al dibujo de la sombra. Para ello, creamos una forma geométrica en forma de elipse, definida dentro de un rectángulo de anchura 5/4 y altura 3/5 sobre la anchura y altura de uno de los dos rectángulos obtenidos en la fase de detección.

```
Rectangle eyeRect = new Rectangle(0, 0, leftEyeRect.Width + leftEyeRect.Width/4, 3 * leftEyeRect.Height / 5);
```

```
using (GraphicsPath shadowPath = new GraphicsPath())
{

      shadowPath.AddEllipse(eyeRect);

      MakeUpUtils.createColorBlendPath(gs, shadowPath, lColors);

}
```

Una vez establecida la región de recorte con el rectángulo del ojo, mediante la técnica de degradado múltiple creamos la mezcla de colores y dibujamos la forma.

```
public static void createColorBlendPath(Graphics g, GraphicsPath
path, List<BlendMakeupColor> lBlends)
{
      using (PathGradientBrush pgb = new PathGradientBrush(path))
      {
            pgb.InterpolationColors = doColorBlend(lBlends);

            g.SmoothingMode = SmoothingMode.AntiAlias;
            g.FillPath(pgb, path);
      }
 }
```

Puntualizar que en la mezcla se puede incluir número ilimitado de colores, con lo que se podrán crear efectos bastante atractivos.

Siendo esto la primera versión del método, hace falta realizar más experimentos hasta conseguir que la aplicación del maquillaje sea los más realista posible.

Imagen 2.9: Efecto obtenido al aplicar la sombra de ojos

2.5-. ACCESORIOS: LAS GAFAS

Para crear el efecto de la prueba de accesorios, en este caso prueba de gafas, hemos empleado la siguiente estrategia.

Cuando se selecciona la gafa deseada, esta pasa a ajustarse a la cara para que tenga una apariencia real. Para realizar esto al seleccionar las gafas primero se obtiene la posición que deberían tomar en relación a la cara.

Esta posición se obtiene a través de la clase **FaceDetector**, obtenemos la posición de los dos ojos. Para que las gafas encajen perfectamente, reducimos el tamaño de éstas por los 2 lados, ya que como hemos visto en el apartado anterior, FaceDetectors nos devuelve un rectángulo más ancho que el tamaño de la cara.

A su vez, lo posicionamos unos pixeles por encima de la posición de la altura del rectángulo de los ojos.

Una vez tenemos la posición de las gafas, solo queda el redimensionado del fichero que contiene la imagen con las gafas.

Primero obtenemos las proporciones de la fotografía con una simple formula:

Proporción = ancho Imagen / ancho Gafas.

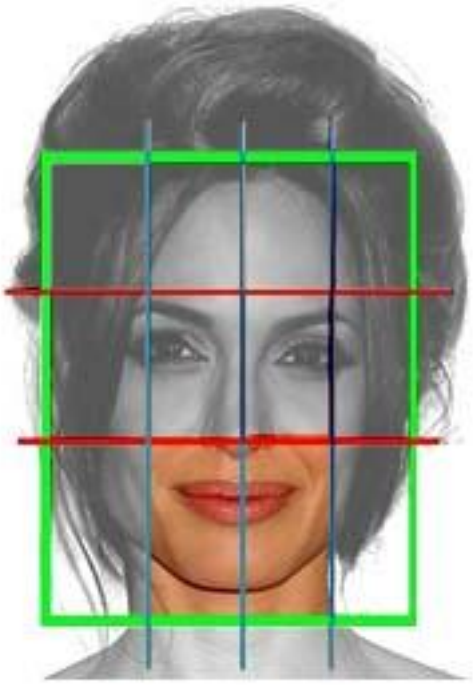
Con esa proporción simplemente la aplicamos a la imagen de las gafas para calcular la altura de la imagen que tendrán las gafas y la situamos en la posición calculada anteriormente. Y ya tenemos las gafas situadas y proporcionales a la cara.

Imagen 2.10: Efecto obtenido en la prueba de las gafas

2.6-. DETECCIÓN Y MAQUILLADO DE LA BOCA

2.6.1-. DETECCIÓN DE LA BOCA

Para realizar la detección de la boca, empleamos una estrategia semejante a la expuesta en el anterior apartado. Nos apoyamos en un algoritmo Haar esta vez dirigido a la detección de la boca. Retomando el apartado de la detección de la cara, dijimos que para decir que el rectángulo obtenido por el algoritmo Haar contiene realmente una cara, comprobamos que tiene 2 ojos y una boca en su interior. Pues bien, empezamos a analizar el conjunto de rectángulos que nos proporciona el algoritmo Haar, esta vez correspondiente a la boca. Y nos quedamos con el primero que se encuentre en el tercio inferior y en los 2 cuartos centrales del rectángulo que supuestamente contendrá una cara. *(Ver Imagen 2.11)*



X = P(x,0) cara + Ancho cara / 4;

Y = P(0,y) cara + 2 * + Ancho cara / 4

Width = Ancho cara / 2;

Height = + Ancho cara / 4;

Imagen 2.11: Proporciones de la cara que definen la boca

Además, el área del rectángulo analizado, tenga un área inferior a la del rectángulo de la cara partido 16.

Área Rec. < Área Rec. Cara / 16;

Hemos considerado fundamentales estas condiciones para decir que una boca pertenece a una cara determinada.

Una vez localizada la zona dónde se encuentra la boca hemos de centrarnos exclusivamente en aquellos píxeles que la conforman. Para ello hemos desarrollado un algoritmo que analiza los píxeles del interior del rectángulo que contiene la boca y almacenamos aquellos que cumplen una condición definida. Del mismo modo que para la detección del a piel, esta condición se basa en el análisis del RGB de los píxeles de la imagen original. Aquí termina el proceso que se hace concurrentemente con otras detecciones en el algoritmo que detecta la cara.

2.6.2-. MAQUILLADO LABIAL

Cuando marcamos la opción de pintar los labios, cogemos unos puntos de control estratégicos que son los que dibujarán el contorno de la boca. Estos puntos definen una curva Bezier que define el contorno de los labios.

Estos puntos son obtenidos a partir de los píxeles marcados como pertenecientes a los labios, se dibuja la forma de la boca y se pinta todo el interior de esta área, procediendo después a añadirlo al gestor de capas para que se muestre el efecto sobre la imagen.

Imagen 2.12 : Efecto obtenido al maquillar los labios

3-. INTERFAZ

3.1-. GESTOR DE CAPAS

En el proyecto la clase encargada de gestionar las distintas capas del maquillaje es LayerManager.cs.

Esta clase, tiene varias funcionalidades:

- Establecer el orden de la visualización del maquillaje.

- Guardar los diferentes efectos del maquillaje.

- Indicar el estado de los diferentes estados del maquillaje (Visible, oculto o borrarlo).

- Se encarga de avisar al control de la imagen, del re-dibujado del maquillaje.

3.1.1-. VISUALIZACIÓN DEL MAQUILLAJE.

Los diferentes efectos del maquillaje se van visualizando encima de la foto. Un problema que surge en el posicionamiento de los diferentes efectos es que forzosamente han de seguir un orden lógico, es decir, unas gafas no pueden estar por debajo de la sombra de ojos. Por lo tanto hay que maneja este orden para su correcto dibujado.

Este orden se controla a través de un Enumerado que contiene las diferentes partes del maquillaje.

Un ejemplo que clarifique el orden entre las diferentes partes sería por ejemplo que la base del maquillaje en la piel, siempre ha de estar situada inmediatamente a continuación de la imagen sobre la que hacemos la detección y pintado ya que en otro caso ocultaría el resto de capas.

Para representar este orden en programación , como se ha comentado antes, se ha utilizado un tipo enumerado. Este enumerado posee los elementos en el orden explicado anteriormente. El primer elemento de esta colección sería el situado en el fondo de la imagen (base maquillaje) , mientras que el último sería el situado primero en la imagen (Gafas).

3.1.2-. GUARDADO DE LOS EFECTOS DE MAQUILLAJE

La clase LayerManager es una clase con inteligencia propia. Gracias al patrón de diseño **MVVM,** aparte de guardar los efectos del maquillaje también se encarga del visualizado de los menús. Al ir generando efectos de maquillaje también se irán generando una serie de capas. *(Ver Imagen 3.1)*

Para poder dotar a la clase **LayerManager** de esta funcionalidad se ha tenido que crear una colección personalizada llamada **LayerObservableCollection**. Esta colección deriva de **ObservableCollection** y contendrá una lista de objetos también personalizados del tipo **Layer**. Esta clase es la que posee todos los parámetros necesarios para dibujar el maquillaje, desde la forma , color , posición hasta las acciones del menú de la *Ver Imagen 3.1*

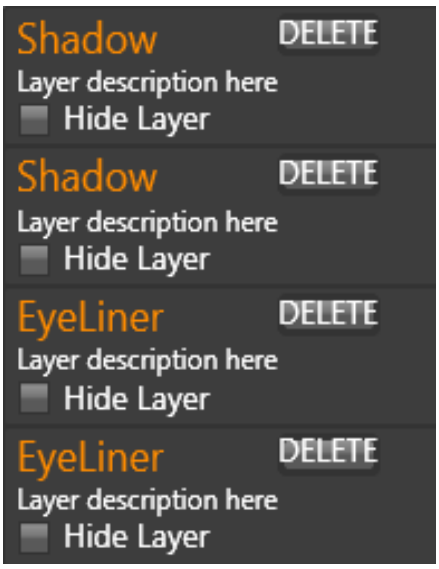


Imagen 3.1 : Barra lateral derecha que contiene las capas

3.1.3-. ESTADO DEL MAQUILLAJE

Como se puede comprobar en la *Imagen 3.1*, tenemos varias opciones para gestionar el efecto, esconderlo o incluso de eliminarlo.

Siguiendo la filosofía de **MVVM** todos los elementos del tipo Layer explicados anteriormente pueden poseer un Comando. Este comando seria el botón Delete de la figura1.

Al indicar que la clase LayerManager posee inteligencia propia, nos referimos a que al pulsar el botón DELETE el elemento de la lista avisara a LayerManager para que sea borrado. En la sección correspondiente a Model-View-ViewModel *(Capítulo 9)* desarrollaremos con todo detalle este patrón de diseño, pero adelantamos que esta filosofía no usa eventos sino que asocia un comando que contiene la funcionalidad que se requiera a un determinado control control.

En este preciso instante el único proceso que se realizará será borrar el elemento de la **ObservableCollection** que contiene a todos los efectos del maquillaje.

Gracias al patrón del diseño MVVM y a que **ObservableCollection** hereda de la interfaz **INotifyPropertyChanged**, internamente **LayerManager** volverá a re-dibujar los efectos sobre la imagen, sin incluir el último elemento eliminado, esto es gracias a que la colección de efectos disparará un evento para notificar que ha cambiado.

A su vez, volverá a re-dibujar el menú de las capas dejando excluido el elemento borrado.

Del mismo modo ocurrirá si añadimos un efecto. Tras la detección pertinente y creación del objeto Layer se invocara el método AddLayer() el cual añadirá el efecto a la colección y se repetirá el mismo proceso de re-dibujado de la foto y del menú.

La funcionalidad de dejar oculto el elemento será un proceso parecido al anterior. La diferencia reside en que **LayerManager** posee una colección auxiliar para los efectos que se encuentran invisibles, al optar por establecer que una capa se ponga invisible es borrada de la colección principal (recordemos que automáticamente refrescara la imagen) y es insertada en la colección auxiliar. El proceso se repetirá a la inversa si deseamos volver a poner visible la capa.

3.1.4-. DIBUJADO MAQUILLAJE

Como hemos dicho antes, LayerManager se encarga del dibujado del maquillaje. El proceso es muy sencillo. Tenemos una referencia a la imagen original, y la lista de los efectos del maquillaje. Cada vez que se detecte que hay algún cambio en los efectos del maquillaje, se invoca el método **doDrawLayers()**.

El proceso a realizar en el método **doDrawLayers()**, es el siguiente. Obtiene la imagen original, lee la colección de los efectos del maquillaje, que se encuentra ordenada según el Enumerador nombrado anteriormente y empieza a dibujarlos sobre la imagen.

Este proceso se puede realizar gracias a que cada elemento de la colección de efectos de maquillaje posee los parámetros necesarios para su dibujado (Posición, color, forma, etc ...).

Una vez lo tiene, simplemente devuelve un Bitmap para la representación de este sobre el contenedor principal del proyecto.

3.1.5-. OTRAS CARACTERÍSTICAS

Por último comentar que LayerManager posee otras características de limpieza. Como por ejemplo si deseamos cargar otra imagen. Esta se encargará de reiniciar las colecciones y cambiar la imagen original que deseemos guardar.

Volver a mencionar las características de actualización dinámica de MVVM, ya que gracias a éstas, el menú, también se reiniciará, sin tener que implementar una función que fuerce su refresco.

3.2-. CAPTURA DE IMÁGENES CON LA CÁMARA

Para la obtención de fotos, se ha creado una clase llamada Webcam. Al inicializar la aplicación, inicializamos esta clase y la webcam por defecto del equipo, se pone en marcha. Automáticamente empezará a mostrar por pantalla en formato de video los frame que va recibiendo.

Esta clase, tiene varias opciones una de ellas será capturar la foto. Para esto, se realiza un proceso sencillo, como una vez activada la webcam ésta procede al proceso de frames sin parar, simplemente al indicarle que capture una foto, obtendremos el frame que procesa en ese mismo instante. Para capturar un frame, basta con pulsar la tecla espacio o el botón: "cargar desde webcam" situado en la barra de herramientas. Luego lo tratamos para poder convertirlo en **Bitmap** y poder empezar a trabajar en los efectos de maquillaje.

Otra característica bastante útil, es la configuración de la webcam. Hemos desarrollado una función para poder acceder a la configuración de sistema de nuestra webcam y modificar gran cantidad de parámetros como brillo saturación y contraste, para obtener una imagen con mejor calidad.

Una vez obtenida la foto, se procederá a mostrarla en una galería de fotos. El funcionamiento de esta galería de fotos se basa en los conceptos del patrón de diseño **MVVM** que explicaremos más adelante con más detalle.

Para ello, tenemos una **ObservableCollection**, que es una colección genérica de elementos del tipo **ImageItem** creados expresamente para este uso. En estos objetos almacenaremos la foto obtenida y las funciones que nos permitirán volcarla en el contenedor principal de la aplicación para proceder al añadido de efectos de maquillaje.

Al capturar la foto, automáticamente se añade a nuestra colección de imágenes. Por el procedimiento de MVVM, automáticamente detectará que hay un nuevo elemento y la galería de imágenes se refrescará para visualizar la nueva fotografía. Como hemos dicho antes, todas estas fotografías tendrán la funcionalidad de poder ser tratadas en cuanto a términos de maquillaje se refiere. Esto se debe a que cada ítem de la colección, posee una propiedad **ICommand,** en este caso se llama: **CargarFotoCommand**. Este comando se asocia a un botón que tiene la funcionalidad para cargar una imagen de la galería, en el contenedor principal de

la aplicación. Y este comando se desencadenará haciendo clic sobre la imagen que deseemos cargar.

3.3-. EXTENSIÓN .MKUP

Para almacenar el resultado del maquillado, sin perder el contenido de las capas, hemos creado ficheros de extensión .mkup . Estos ficheros contienen la foto original y todas las capas, con su correspondiente contenido, que el usuario haya realizado en la aplicación. De esta forma, el usuario podrá guardar el proyecto y seguir maquillando cuando lo desee.

3.3.1-. PROCESO DE GUARDADO DEL PROYECTO .MKUP

Al terminar de maquillar la foto elegida, podemos pulsar en el proyecto el botón de Guardar proyecto. Primero seleccionaremos el lugar donde deseamos guardar la fotografía, una vez seleccionado, el programa comenzara el proceso de guardado.

Primero se transformará la imagen original en una array de bytes. Tras la transformación escribirá 2 líneas, la primera indicará que lo que se va a volcar es una foto, escribiendo la frase Foto. En la segunda línea se volcara el array de bytes originado tras la conversión de la imagen. Cada elemento del array estará separado por un punto y coma: ";" .

Después volcaremos al fichero, las capas que el usuario haya creado. Para ello, primero se escribe en el fichero el nombre de la capa que se haya creado, en la siguiente línea se inserta un booleano que indica si la capa se encontraba escondida o visible. Las siguientes cuatro líneas indicaran el color de la capa, escribiendo en el fichero el color formado por el R G B y la variable Alpha que contiene la opacidad o transparencia con la que se aplica el color de la capa de maquillaje.

3.3.2-. PROCESO DE CARGA DEL PROYECTO .MKUP

Para hacer el proceso de carga se comenzara primero leyendo la array de bytes de la imagen original guardada en el fichero.

Una vez se ha leído se transformara en la foto y se indica al programa que tenemos una foto original, estableciéndola en la clase LayerManager, al realizar esto, también se ejecutara el proceso de detección de las diferentes partes de la cara.

Posteriormente se comienzan a generar las capas que se leen del fichero. Este proceso es muy sencillo, ya que simplemente se lee que tipo de capa se va a insertar y el color. Una vez obtenidos estos dos parámetros, se realiza el mismo proceso que se ejecuta cuando pintamos una foto (jpg , gif , . . .) cargada desde archivo utilizando los colores de los menús .

Por ejemplo, como anteriormente ya hemos detectado los ojos de la cara, si cargamos desde el fichero la sombra de ojos izquierda y su color correspondiente, se llamara al proceso de pintado de sombra de ojos, con el color. Este se encargará de pintar el ojo correspondiente y de añadir la capa de sombra de ojos al LayerManager.

Para finalizar el proceso tan solo estableceremos en la clase LayerManager si la capa estaba visible o invisible cuando fue guardada.

4-. TECNOLOGÍAS UTILIZADAS

4.1-.LIBRERÍA DE VISIÓN: OPENCV

4.1.1-. ¿QUÉ ES OPENCV?

OpenCV es una potente librería útil para la visión por computador, es decir, para la captura y procesamiento de imágenes, como sus siglas en términos anglosajones indican: **Open Source Computer Vision**.

Fue desarrollada por Intel en 1999 y se emplea en multitud de aplicaciones que requieran de la visión artificial para su funcionamiento. Está programada en c y c++, compuesta por más de 500 algoritmos optimizados que aprovechan el rendimiento de los procesadores multinúcleo.

OpenCV tiene una licencia BSD, lo que la convierte en una librería de libre disposición, tanto para el empleo académico como comercial. Este tipo de licencia, es si cabe menos restrictiva que la GPL, ya que permite el uso del código fuente para software no libre.

OpenCV es una librería multiplataforma, está preparada para su uso en diferentes sistemas operativos, tanto Linux como Windows y Mac OS X.

Como hemos podido comprobar, la potencia de esta librería radica en su capacidad para el reconocimiento de objetos, detección de movimiento y tratamiento de imágenes. Nosotros, la hemos empleado para localización de la cara y sus partes: ojos y boca.

Ésta librería nos ha sido muy útil, ya que soporta toda la parte de la detección y localización de la cara y sus elementos dentro de una imagen dada.

Puesto que propiamente esta librería está enfocada como tal para la programación con c, c++ y python, hemos utilizado para el desarrollo de nuestra aplicación, una librería envoltorio de OpenCV, Emgu, que nos permite utilizar los algoritmos implementados por OpenCV en el lenguaje de programación c#.

4.1.2-. WRAPPERS: LIBRERÍAS ENVOLTORIO DE OPENCV

A lo largo del proceso de desarrollo del proyecto, hemos probado distintos "wrappers", adaptándonos a las novedades del momento. Los "wrappers" son librerías envoltorio, reciben este nombre, porque su propósito principal es llamar a otras funciones pudiendo añadirles funcionalidad y sin tener en cuenta el lenguaje de programación en que han sido desarrolladas. En nuestro caso, estos "wrappers" nos han servido de nexo para interactuar con las funciones programadas en c y c++ de OpenCV.

4.1.2.1-. OPENCVDOTNET

En un principio empleamos OpenCVDotNet. OpenCVDotNet es un proyecto desarrollado para la Universidad de IDC Herzliya (Israel) por el Dr. Yael Moses.

Para nosotros fue un descubrimiento, ya que esta librería envoltorio nos permitía combinar la calidad de OpenCV en cuanto a detección y reconocimiento de formas se trata y la potencia de la plataforma .NET en el framework 3.5 con C#.

Hicimos varios proyectos ejemplo empleando esta librería, enfrentándonos a los problemas que planteaba nuestro proyecto. Los resultados en cuanto a detección y reconocimiento de formas eran bastante satisfactorios, aunque había algunas funciones de OpenCv que no estaban implementadas o mejor dicho "envueltas".

Seguimos investigando y nos encontramos con Emgu.

4.1.2.2-. COMPARATIVA ENTRE EMGU CV Y OPENCVDOTNET

Emgu CV aporta muchas más funciones que OpenCVDotNet y es más operativo.

Además a diferencia de otras envolturas como OpenCVDotNet o SharperCV que utilizan para su desarrollo código no seguro, Emgu CV está escrito enteramente en C #. La parte más interesante es que puede ser compilado en mono y por lo tanto es capaz de funcionar en cualquier plataforma compatible con Mono, incluyendo Linux, Solaris y Mac OS X, por lo que es multi-plataforma.

Emgu trabaja con la última versión de OpenCV la 2.1. Además, nos proporciona las funciones de OpenCV que se utilizan para entrenar los clasificadores para la detección. Llamado HaarTraining, de manera que podemos crear nuestros propios clasificadores: Haars.

Este proceso, explicado de una forma sencilla se realiza cargando en un programa una gran cantidad de imágenes que contienen un determinado objeto, a mayor cantidad de imágenes procesadas la calidad de la detección será mayor. Tras procesarlas, este algoritmo nos proporciona un documento .xml que contiene los parámetros necesarios para la detección de dicho objeto en una determinada imagen.

De una forma más sencilla también se pueden implementar a través de matrices que contienen determinadas características de los objetos a tratar

Emgu nos proporciona un manejador de excepciones y depurador de código fuente.

Los creadores de Emgu nos garantizan la actualización de la librería en un periodo máximo de un año. Tanto es así que nuestro proyecto ha ido amoldándose a varias de estas actualizaciones, empezamos programando en la versión 1.5 y hemos actualizado a la versión 2.3.0, la más reciente. Desde la versión 1.5 a la actual, esta librería ha experimentado cambios extraordinarios. Ha solucionado los bugs de sus principios como por ejemplo, uno que nos afectaba directamente, el que definía la región de interés de una imagen (ROI), en la primera versión de nuestro proyecto hacíamos algunas operaciones bastante engorrosas para esquivar este bug que desaparecieron en la versión 2.1. La versión 2.3.0 aporta una mayor eficiencia en los cálculos, ya que el esfuerzo computacional se distribuye entre la CPU y la GPU, siendo especialmente eficiente en las gráficas más modernas de nVidia de la serie G8X que implementan algoritmos CUDA (Arquitectura de Dispositivos de Cómputo Unificado)

Emgu viene con una licencia GPL, Licencia Pública General, esto nos permite utilizar esta librería gratuitamente con fines académicos y no comerciales, aunque parece ser que para los fines comerciales han puesto un precio bastante económico .

Name	Emgu CV	OpenCVDotNet
Cross Platform (Mono)	✓	X
OpenCV 2.1	✓	X
Machine Learning	✓	X
Exception Handling	✓	X
Debugger Visualizer	✓	✓
Actively Maintained [1]	✓	X
License	GPL or Commercial License with a small fee	Non-commercial GPL

Imagen 4.1 : Tabla comparativa de las librerías envoltorio EmguCV y OpenCVDotNet

Podemos decir que Emgu CV está mucho más desarrollado que OpenCVDotNet, posiblemente esto se debe a que Emgu es una librería en constante desarrollo mientras que OpenCVDotNet fue ideada como un proyecto "cerrado" por decirlo de alguna manera, aunque quien sabe si sus desarrolladores siguen trabajando en ella y pronto nos sorprenden con nuevas e interesantes características.

Por todo ello, nosotros hemos apostado por Emgu para el desarrollo de nuestro proyecto.

4.1.2.3-. ARQUITECTURA DE EMGU

EmguCV tiene dos capas de envoltorio.

1- La capa de base (Layer 1) contiene la función, estructura y asignaciones de enumeración que reflejan directamente los de OpenCV.

2- La segunda capa (Layer 2) contiene las clases de . NET.

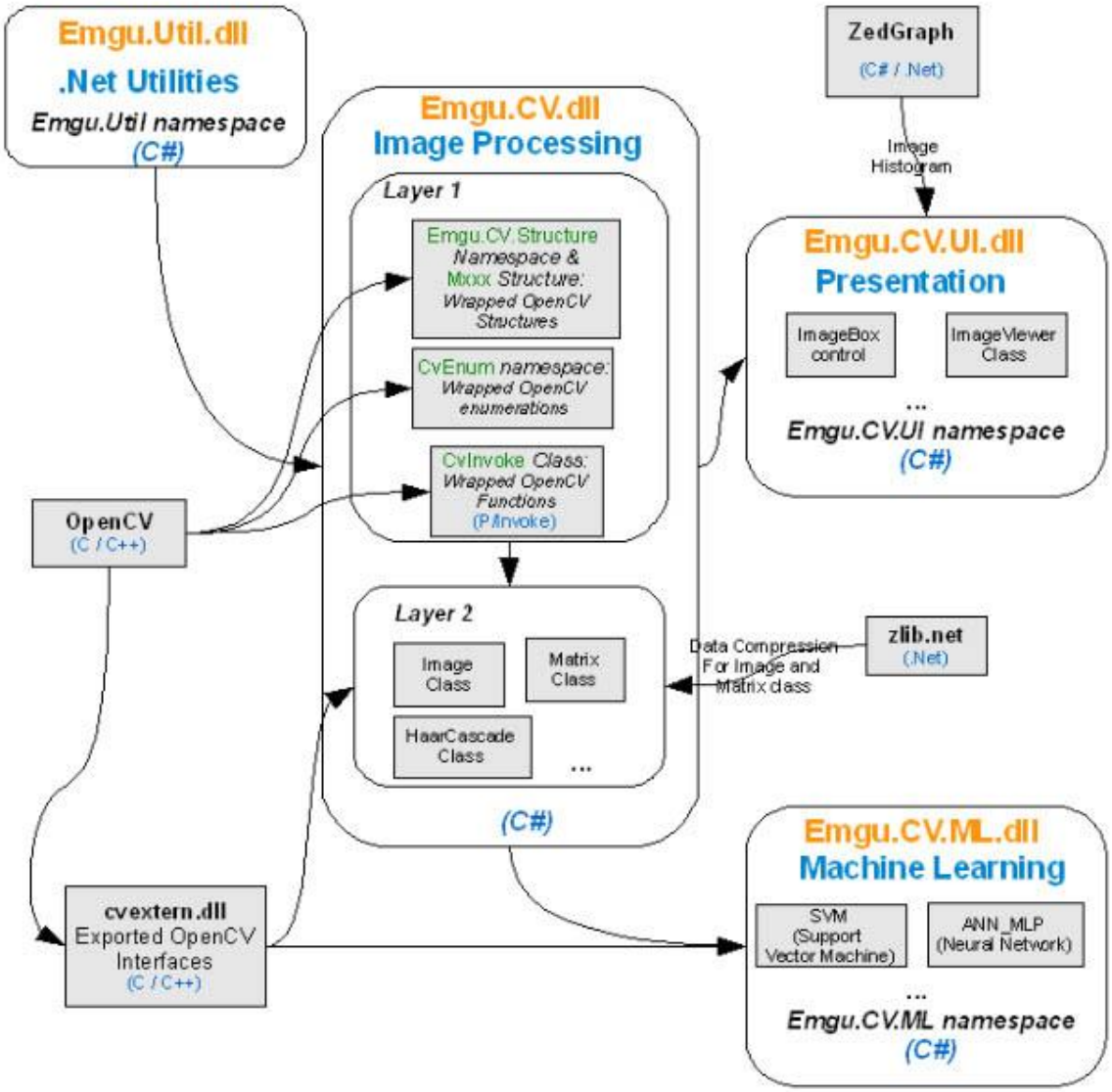


Imagen 4.2 : Imagen que representa la arquitectura de la librería EmguCV

4.2-. INTERFAZ DE PROGRAMACIÓN DE APLICACIONES

GDI+ (GRAPHICS DEVICE INTERFACE)

La interfaz GDI, es uno de los tres componentes o subsistemas de la interfaz de usuario de Microsoft Windows.

En GDI+ la clase principal para obtener acceso a las operaciones de dibujo es la clase Graphics. Como es abstracta, no se puede instanciar, sino que dispone de una serie de métodos estáticos que permiten obtener el contexto gráfico. Entre ellas esta el método FromImage() que permite obtener el contexto desde una imagen o un mapa de bits.

Por lo tanto, en nuestro caso para realizar las operaciones de maquillaje, la estrategia, ha sido dibujar en un mapa de bits que representará una capa, y una vez finalizado el dibujo volcar / dibujar el contenido de la capa sobre la imagen original, en este caso la imagen del rostro humano.

GDI+ nos proporciona una cantidad bastante amplia de métodos, propiedades, etc. , para el dibujo 2D, dibujo de líneas simple, líneas Béizer, degradados, formas personalizadas, transformaciones, etc.

Empecemos creando un nuevo mapa de bits donde vamos a realizar el pintado:

```
using (Bitmap cheekBitmap = new Bitmap(getBlushWidth(), getBlushHeight())) {
        using (Graphics g = Graphics.FromImage(cheekBitmap)) {

        }
}
```

Como se observa en este fragmento de código, en la primera línea creamos el mapa de bits con las dimensiones de la mejilla, que se calculan previamente mediante los métodos

getBlushWidth() y getBlushHeight(). Una vez creado el mapa de bits, obtenemos el contexto de dibujo desde el mismo. Notar el uso de los bloques *using* – permiten liberar automáticamente los recursos usados por los objetos, invocando al método *Dispose*() de los mismos. De esta manera evitaremos *leaks* de memoria.

Teniendo el contexto gráfico correctamente obtenido podemos empezar a "dibujar" sobre él. En GDI+ muchos métodos que empiezan por *draw*, tienen su correspondiente método *fill*. El primero de ellos dibujará el borde del objeto, séase, un rectángulo, circulo, forma personalizada etc., mientras que su correspondiente pintará el interior del objeto.

Existen diferentes tipos de brochas con las cuales rellenar el interior del objeto, entre los cuales están SolidBrush, LinearGradientBrush y PathGradientBrush. La primera de ellas rellena el interior del objeto con un color sólido, mientras que las otras dos realizan un relleno con 2 o varios colores de manera degradada – la primera de ellas rellena el interior de un rectángulo, mientras que la segunda el interior de cualquier forma personalizada que podemos imaginar / crear.

4.3-. LENGUAJE DE PROGRAMACIÓN:

C SHARP

Hemos elegido este lenguaje de programación ya que por sus características se adapta perfectamente al desarrollo de este proyecto. C Sharp es un lenguaje de programación orientado a objetos de la empresa Microsoft, que forma parte de su plataforma .NET.

C# es un lenguaje **sencillo,** que elimina muchos elementos innecesarios incluidos por otros lenguajes, sin por ello, establecer limitaciones en cuanto al desarrollo se refiere. Es **moderno** ya que incorpora elementos en el propio lenguaje que son muy útiles, como por ejemplo un tipo básico decimal que permita realizar operaciones de alta precisión.

También, como todo lenguaje actual es **orientado a objetos**, soporta todas las características propias de estos lenguajes: **encapsulación, herencia y polimorfismo.** Gracias a estas propiedades, C# nos permite la creación y diseño de componentes para su reutilización.

Por otra parte, como cualquier lenguaje moderno, C# se encarga de la gestión automática de la memoria, disponiendo de un recolector de basura. Ofrece posibilidades para la optimización del uso de recursos como por ejemplo: la instrucción **using** para la liberación de recursos, que hemos hecho valer en nuestro programa.

Por último, comentar otras características, como su **sistema de tipos unificados,** en el cual, a diferencia de C++, todos los tipos de datos que se definan, siempre derivarán de una clase común llamada **System.Object**. La **eficiencia** de este lenguaje, se encuentra en que con sus numerosas restricciones, puede asegurar la seguridad del desarrollo.

Aportando más características de C# podemos comentar los distintos patrones de diseño usados, como **Singelton**. Este patrón de diseño nos permite el uso de una clase que se instancia automáticamente, permitiendo su uso en cualquier momento.

Además de todas estas características, C# pertenece a la plataforma .NET, de la cual hereda características, como la lectura de un XML, necesario para la implementación de los algoritmos Haar.

4.4-. TECNOLOGÍA DE LA CAPA DE PRESENTACIÓN:

WINDOWS PRESENTATION FOUNDATION (WPF)

Windows Presentation Foundation, a lo que a partir de ahora nos referiremos como WPF, es la solución de Microsoft para los desarrolladores de software y los diseñadores gráficos que quieren crear interfaces de usuario modernas sin tener que utilizar otras tecnologías más complicadas. Básicamente, es una tecnología incluida a partir del framework 3.5 para crear aplicaciones modernas con un diseño atractivo y funcionalidades muy variadas.

4.4.1-. LOS ASPECTOS MÁS DESTACADOS DE WPF:

4.4.1.1-. Amplia integración:

Antes de la existencia de la tecnología WPF, un desarrollador que quería usar 3d, video, audio, visualización de documentos (tipo pdf) y 2d en una misma aplicación tenía que aprender varias tecnologías independientes con varias inconsistencias y tenía que intentar fusionarlas sin mucho soporte integrado. WPF integra todas estas funcionalidades en el mismo framework.

En nuestra aplicación hemos podido obtener y visualizar imágenes desde la webcam sin mucho trabajo en cuanto a programación se refiere. Con esta tecnología resulta muy sencillo visualizar un vídeo o una imágen. Sólo utilizando librerías para la correcta interacción con la webcam .

4.4.1.2-. Independencia de la resolución:

La idea de Microsoft fue crear un mundo en el que cambiar a una resolución más alta o de mayor DPI (Dots per inch),es decir, puntos por pulgada, no signifique que todo se vea más pequeño, a cambio, los gráficos y los textos se ven más nítidos.

Esto es realmente útil ya que garantiza donde quiera que se instale este proyecto, no habrá problemas de visualización.

4.4.1.3-. Aceleración por hardware:

Aunque WPF es una nueva tecnología, está construida encima de Direct3d. Específicamente, el contenido de una aplicación WPF (sea 2d o 3d) es convertido en triángulos 3d, texturas o otros objetos Direct3d y luego renderizados por hardware.

Esta característica, nos aporta un mejor rendimiento y optimización de recursos. El peso de la aplicación, no recae únicamente sobre los recursos software, sino que utiliza también el hardware para un funcionamiento más ligero.

4.4.1.4-. Programación declarativa:

La combinación de XAML con WPF es parecida a la de html para definir una interfaz, pero con un rango de expresividad increíble. Esta expresividad se extiende más allá de las interfaces de usuario, ya que WPF también usa XAML como formato para los documentos, representación de modelos 3d y más.

4.4.1.5-. Alta composición y Personalización:

Los controles WPF son altamente personalizables en formas nunca antes vistas. Se puede crear un ComboBox lleno de botones animados, o un menu lleno de videoclips. Aunque este tipo de personalizaciones no es muy usable, es importante saber que ahora no se necesita escribir mucho código para lograrlo. De la misma forma también se pueden crear múltiples diseños de una aplicación con formas muy radicalmente diferentes.

Queríamos para nuestro proyecto una interfaz moderna y agradable al usuario. Algo novedoso que se distanciara de los típicos controles e interfaces Windows. Esta tecnología nos ha posibilitado que nuestra interfaz esté diseñada al detalle y tenga una apariencia única.

4.5-. ENTORNO DE DESARROLLO: VS2008

Para desarrollar nuestra aplicación elegimos el entorno de desarrollo de Microsoft, el Visual Studio 2008 con el framework 3.5, con licencia de la Universidad Politécnica de Valencia.

El Visual Studio 2008, nos permite emplear todas las tecnologías que escogimos para el desarrollo de esta aplicación en el lenguaje de programación C#.

Aparte de aplicaciones en WPF, puede ser usado para desarrollar aplicaciones de consola y aplicaciones gráficas a través de Windows Forms, Sitios Web, Aplicaciones Web y web Services. Todo esta desarrollado en en código nativo para que sea soportado para todas las plataformas de Microsoft .

Visual Studio incluye un editor de código que posee IntelliSense para la refactorización de código. También incluye un debbugger , un diseñador para construir interfaces, diseñador web, etc ..

Incluye diferentes lenguajes de programación como C/C++ (Visual C++), VB.Net, J#, C# en nuestro caso y F#. Además soporta otros lenguajes como M, Python y Ruby.

A partir de la version de 2002 se introdujo la plataforma .Net. Esta es una plataforma de ejecución intermedia, de forma que los lenguajes desarrollados en .Net no se compilan en lenguaje maquina, si no en un lenguaje intermedio denominado MSIL (Microsoft Intermediate Language). De esta forma el código no se convierte a código máquina hasta que no se ejecuta, de manera que el código puede ser independiente de la plataforma.

Las plataformas han de tener una implementación de Infraestructura de Lenguaje Común (CLI) para poder ejecutar programas MSIL. Actualmente usando implementaciones de .Net externas a Microsoft, como Mono y DotGNU se pueden ejecutar aplicaciones MSIL en Linux y Mac OS X.

La versión de Visual Studio 2008 ha introducido numerosas mejoras como "Windows Communication Foundation" (WCF) y "Windows Presentation Foundation" (WPF).El primero tiene como objetivo la construcción de aplicaciones orientadas a servicios mientras que el último apunta a la creación de interfaces de usuario más dinámicas.

4.6 ENTORNO DE DESARROLLO PARA LA COMPATIBILIDAD CON LINUX: MONO

MONO, es una potente plataforma, que nos permite desarrollar aplicaciones .NET con independencia del sistema operativo anfitrión donde se vaya a ejecutar.

Para realizar las pruebas con los algoritmos de detección en sistemas operativos Linux, deberemos compilar nuestra solución de Visual Studio usando MONO. Primero, deberemos instalar una serie de paquetes en nuestra máquina Linux usando el siguiente comando:

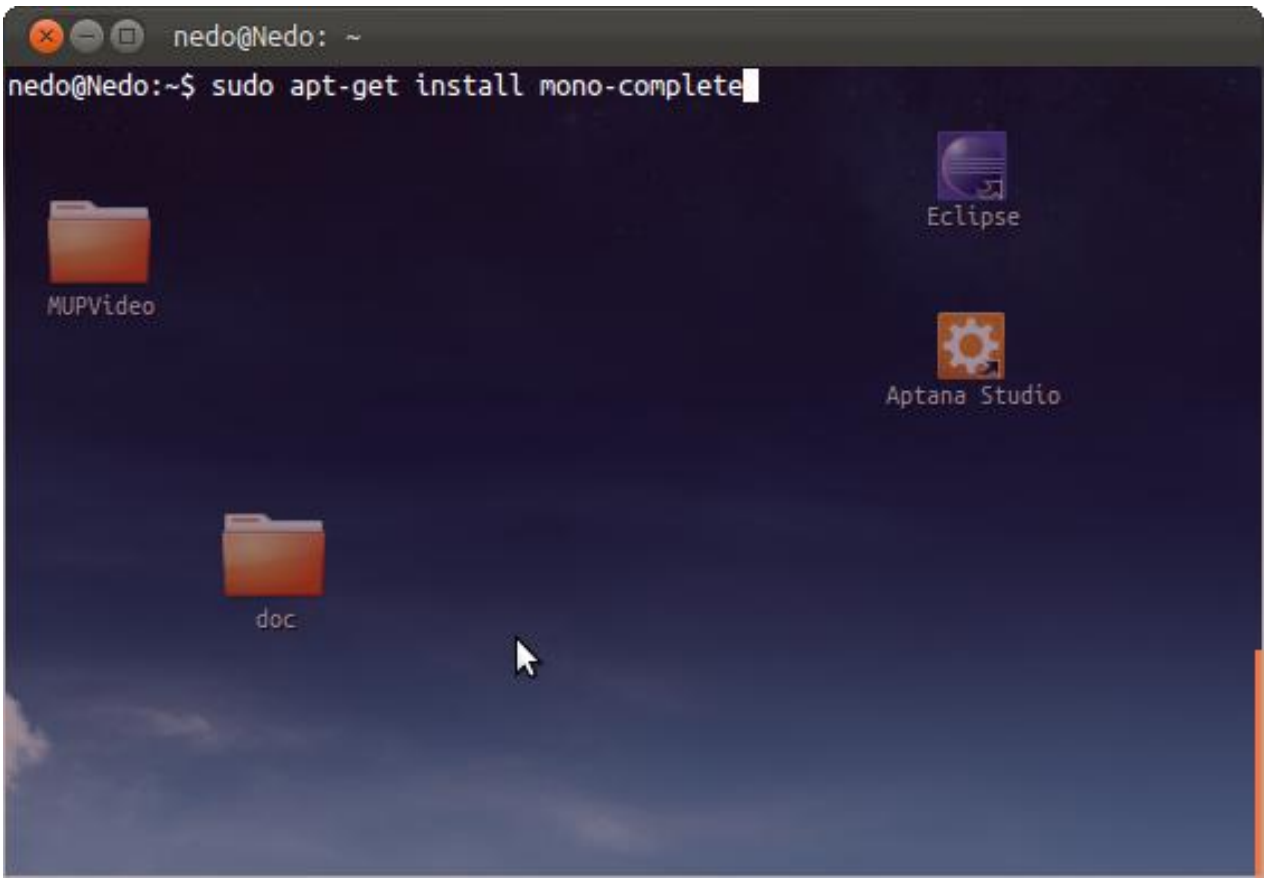


Imagen 4.3 : Imagen que representa el modo consola para la instalacióncompleta de MONO.

Este comando, instalará MONO con todas las dependencias requeridas. Debido a que ya hemos compilado la librería OpenCV en Linux, tan solo deberemos entrar en el directorio de

salida de nuestra solución Visual Studio y lanzar el siguiente comando para ejecutar nuestra aplicación de prueba:

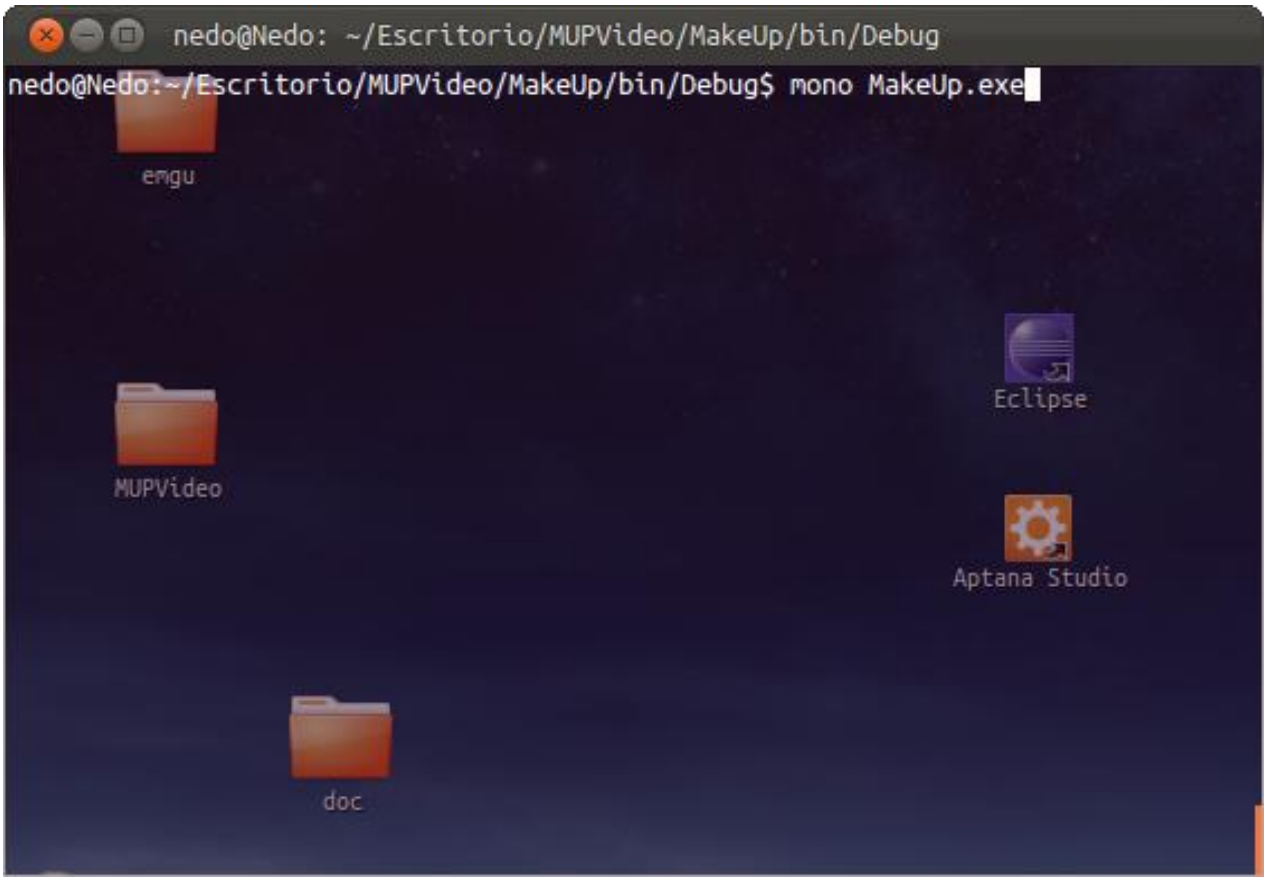


Imagen 4.4 : Imagen que representa el modo consola para la instalación todas las dependencias requeridas por el proyecto.

Una vez puesto en funcionamiento, queda demostrado que la capa de negocio puede ser considerada multiplataforma. Para que este proyecto funcionase correctamente en sistemas operativos Linux, habría que desarrollar una interfaz compatible, utilizando para ello por ejemplo MONO, y añadiendo a la solución este mismo proyecto compilado utilizando este mismo entorno.

5-. PATRÓN DE DISEÑO: MODEL-VIEW-VIEWMODEL MVVM

5.1-. INTRODUCCIÓN

Con la llegada del marco de trabajo WPF, la puesta en práctica del patrón MVVM es cada vez más popular. Por lo tanto hemos optado, además de la arquitectura en capas, usar dicho patrón para llevar a cabo la implementación del proyecto.

Unas de las principales ventajas de este patrón es la separación de los distintos artefactos. Las vistas nunca interactúan directamente con el modelo de datos, sino que lo hacen a través del controlador llamado *ViewModel,* que representa la abstracción de la vista, además de encargarse de enlazar los datos entre el modelo y la vista. El lenguaje declarativo XAML nos ofrece la posibilidad de establecer esos enlaces (*data bindings*), que pueden ser unidireccionales o bidireccionales, evitando que la vista contenga código correspondiente a la lógica de negocio y además consiguiendo el nivel de acoplamiento muy bajo. Las acciones del usuario que interactúa con los elementos de la vista (*View*) se abstraen en forma de comandos que se enlazan con los componentes de la interfaz (botones, listas, cajas de verificación, etc.). Al ejecutarse los comandos, invocarán a los métodos definidos en nuestros modelos (*Model*). Otras de las ventajas de este estilo arquitectónico es la división del trabajo por roles cuando se trabaja en equipo. Como los requisitos funcionales del diseño de la interfaz de usuario son más propios del diseñador gráfico que del programador, la GUI se podrá crear en otras herramientas, simultáneamente e independientemente de la parte de lógica de negocios o datos. Por último, cuando nuestra aplicación está ensamblada usando MVVM las pruebas unitarias son mucho más llevaderas y comprensivas, ya que hay poca dependencia entre las diferentes partes que la forman.

5.2-. CONCEPTOS BÁSICOS

Antes de empezar a explicar con detalle como está implementado el patrón MVVM en Makeup Stylist, se explicarán las nociones básicas a nivel de código en el lenguaje de programación C#.

Todo manejador de vista (*ViewModel*) debe extender de la clase abstracta *ViewModelBase* que tiene el siguiente aspecto:

```
public abstract class ViewModelBase : INotifyPropertyChanged {
  public event PropertyChangedEventHandler PropertyChanged;
  protected void NotifyPropertyChanged(String sData) {
     if (PropertyChanged != null) {
        PropertyChanged(this, new   PropertyChangedEventArgs(sData));
     }
  }
}
```

Como vemos, debe implementar la interfaz INotifyPropertyChanged, y concretamente el método NotifyPropertyChanged que notificará al sistema WPF de datos enlazados que la propiedad dispone de nuevo valor y que en consecuencia tiene que actualizarse. El primer argumento del método NotifyPropertyChanged tiene que ser el nombre de la propiedad que ha cambiado, que al establecerse un nuevo valor disparará el evento de tipo PropertyChangedEventHandler. Una vez definida la clase base, podemos empezar a crear nuevos manejadores de vistas. Para el ejemplo vamos definir un ViewModel encargado de manejar la información de los contactos de usuarios.

```
public class ContactViewModel : ViewModelBase

{

        private ObservableCollection<Contact> contacts = new
        ObservableCollection<Contact> ();

        private void addContact(Contact contact)
        {
            contacts.Add(contact);
        }

        private void delContact(Contact contact)
        {
            contacts.Remove(contact);
        }

        private ObservableCollection<Contact> getContacts()
        {
            return this.contacts;
        }

}
```

Para mantener la implementación los más sencilla posible no se ha creado ningún servicio empresarial , sino que el propio *ViewModel* ya define los métodos que se encargarán de añadir nuevo contacto, eliminar un contacto dado o obtener todos los contactos de la lista. El modelo es el siguiente.

```
public class Contact {

  private String name;
  private String phone;

  public String Name
  {
        get
        {
            return this.name;
        }
        set
        {
            this.name = value;
        }
  }

  public String Phone
  {
        get
        {
            return this.phone;
        }
        set
        {
            this.name = phone;
        }
  }

}
```

Definimos dos atributos que representan el nombre del contacto y su número de teléfono. Ahora vamos a ampliar el *ViewModel* declarando el comando que se ejecutará al añadir nuevo contacto desde la interfaz de usuario. El procedimiento para la eliminación del contacto es el mismo así que para más claridad del código se omitirá.

```
public class ContactViewModel : ViewModelBase {

        private ObservableCollection<Contact> contacts = new
      ObservableCollection<Contact> ();

        private ICommand addContact;

        public ICommand AddContact
        {
                get
                {
                    return this.addContact;
                }
                set
                {
                    this.name = addContact;
                }
        }

      public ContactViewModel()
      {
            doInitCommand();
      }

      private void doInitCommand()
      {
            this.addContact = new GenericCommand(){
            CanExecuteDelegate = c => c != null,
            ExecuteDelegate = c => addContact ((String) c )};
      }
      private void addContact(String name)
      {
          Contact contact = new Contact();
          contact.Name = name;
          contacts.Add(contact);
      }

      private void delContact(Contact contact)
      {
          contacts.Remove(contact);
      }

      private ObservableCollection<Contact> getContacts()
      {
          return this.contacts;
      }

}
```

El método *doInitCommand* se encarga de inicializar los delegados del comando pudiendo solamente ejecutarse la adición del nuevo contacto cuando este sea distinto de *null*. Finalmente nos queda enlazar el comando con la vista. Para ello vamos a usar el lenguaje declarativo XAML.

```
<Window
x:Class="ContactManager.MainWindowView"    xmlns="http://schemas.microsoft.co
m/winfx/2006/xaml/presentation"    xmlns:x="http://schemas.microsoft.com/winf
x/2006/xaml"    Title="Contactos" Height="300" Width="300">

  <Grid>

    <TextBox Margin="ContactName" Margin="50,34,50,160"></TextBox>

    <Button Margin="70,122,70,120" Content="Insertar contacto" Command=
"{Path= AddContact }" CommandParameter ="{Binding ElementName= ContactName,
Path=Text}"></Button>

  </Grid>
</Window>
```

Como se ve en el anterior código estamos inicializando la propiedad *Command* del botón con el comando que está definido en nuestro *ContactViewModel*, y como parámetro le pasamos el valor que se proporciona en la caja de texto. Para tener acceso a las propiedades de nuestro *ContactViewModel* tenemos que inicializar el contexto de datos en nuestra vista.

```
public partial class MainWindowView
{
  private ContactViewModel contactViewModel = new ContactViewModel();

  public MainWindowView (){
    InitializeComponent();
    this.DataContext = contactViewModel;
  }
}
```

Una vez vistos los conceptos básicos veamos como se ha llevado a cabo la implementación del patrón MVVM en MakeupStylist.

5.3-. ARQUITECTURA DE MAKEUP STYLIST

A continuación mostramos la arquitectura de Makeup Stylist desde el punto de vista del patrón MVVM. *(Véase Imagen 5.1)*

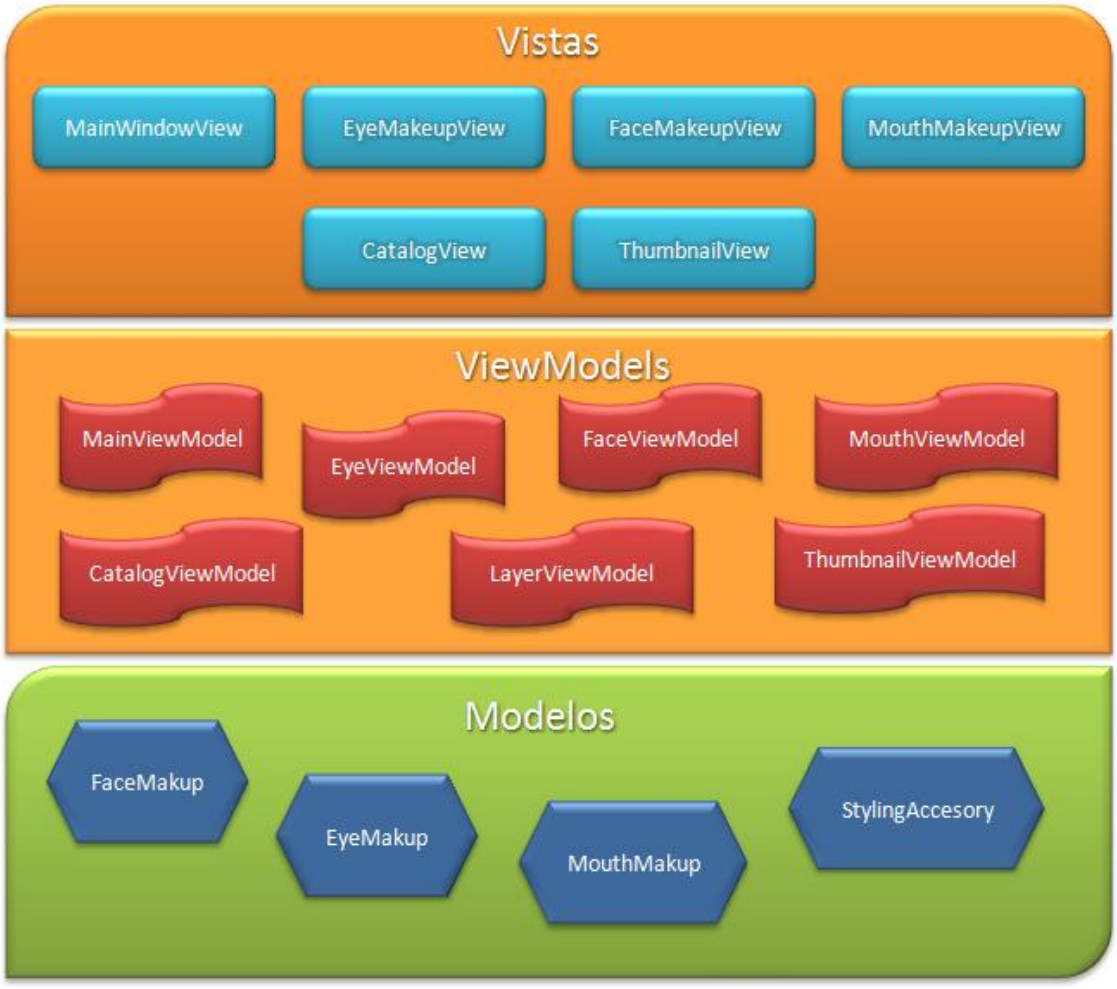


Imagen 5.1: Arquitectura del proyecto

A continuación vamos a explicar brevemente cada uno de los artefactos presentes en el diagrama.

5.3.1-. VISTAS

MainWindowView es la vista principal donde se alojan los elementos que componen la interfaz del usuario.

EyeMakeupView es el control de usuario que permite realizar el maquillaje de los ojos (sombra y lápiz).

FaceMakeupView vista encargada de proporcionar elementos para realizar el maquillaje de la cara.

MouthMakeupView es la vista que permite realizar el pintado de los labios.

CatalogView visualiza listas de catálogos de productos que se obtienen desde el archivo XML. Al seleccionar un catálogo se muestran todos los productos con información detallada de cada uno. Asimismo es posible realizar las acciones de inserción, borrado, edición tanto de catálogos como de productos.

ThumbnailView proporciona acceso a imágenes en miniatura seleccionables, para representar accesorios como las gafas.

5.3.2-. VIEWMODELS

MainViewModel inicializa los comandos de carga / captura de la imagen y las opciones de la webcam. También se encarga de notificar a los demás controladores de vista cuando se carga la imagen desde el cuadro de diálogo, usando el patrón Mediator que se explicará más adelante.

EyeViewModel invoca los métodos del modelo para realizar el pintado y eliminación de la sombra y el lápiz de ojos, definiendo los comandos necesarios. También añade las capas correspondientes a la sombra / lápiz en la lista usando el manejador de capas.

FaceViewModel inicializa los comandos usados para dibujar el colorete y el maquillaje sólido y registra las capas correspondientes.

MouthViewModel registra los comandos que se ejecutaran en respuesta al pintado de labios. También notifica cuando la imagen está a punto de repintarse.

CatalogViewModel invoca los métodos de la capa de datos que se ha construido usando la tecnología *XLINQ.* Entre los métodos están los encargados de persistir la información relacionada con los catálogos de productos, así como recuperar dicha información en el arranque de la aplicación.

LayerViewModel lleva a cabo la gestión de las capas renderizadas en MakeupStylist. Ofrece métodos de eliminación y ocultación de las capas.

ThumbnailViewModel contiene métodos para llevar a cabo la carga de imágenes que representan los accesorios de maquillaje.

Para ver como se ha implementado el patrón MVVM en MakeupStylist, vamos a tomar como ejemplo de referencia el pintado de mejillas, operación perteneciente a FaceViewModel. Este artefacto contiene la referencia al modelo *FaceMakeup* que define e implementa las operaciones de dibujo sobre el rostro.

```
public class FaceViewModel : ViewModelBase
    {
        private FaceMakeup faceMakeup = new FaceMakeup();
        private ICommand drawBlushCommand;

        public FaceViewModel() {
            doInitCommands();
            doRegisterImageLoad();
            doRegisterWebCamImage();
        }

        public ICommand DrawBlushCommand
        {
            get
            {
                return this.drawBlushCommand;
            }
            set
            {
                this.drawBlushCommand = value;
                NotifyPropertyChanged("DrawBlushCommand");
            }
        }
```

En el constructor se inicializan los comandos invocando al método *doInitCommands()*. Mediante las expresiones lambda se inicializa la acción y el predicado del comando.

```
protected override void doInitCommands()
{
    this.drawBlushCommand = new GenericCommand()
    {
       CanExecuteDelegate = c => LayerManager.Instance.hasImageLoaded(),
       ExecuteDelegate = c => doBlushMakeUp((Color) c )
    };
}
```

Con *CanExecuteDelegate* indicamos que la acción solamente se podrá ejecutar si la imagen ha sido cargada. Cuando se cumpla la condición del primer delegado, entonces el comando se

podrá ejecutar invocando al método *doBlushMakeUp* que recibe como parámetro el color que se usará para pintar las mejillas.

```
private void doBlushMakeUp(Color blushColor) {
            faceMakeup.doCheeksMakeUp(blushColor);
            redrawLayers();
}
```

Como vemos este método simplemente invoca doCheeksMakeUp del modelo FaceMakeup, y seguidamente registra las notificaciones que se disparan cuando las capas se redibujan. Esto se ha conseguido mediante la implantación del patrón Mediator, sobre cual se hablará más adelante. Por último nos queda conectar FaceViewModel con la vista. Para ello primero tenemos que inicializar el contexto de datos de la vista con nuestro ViewModel, para poder acceder a los comandos publicados por éste.

```
public partial class FaceMakeupView : UserControl
    {
        private FaceViewModel faceViewModel = new FaceViewModel();

        public FaceMakeupView()
        {
            InitializeComponent();
            doDataContext();
        }

        private void doDataContext()
        {

            this.DataContext = faceViewModel;

        }

}
```

Ahora podemos hacer referencia al comando, y asociarlo al botón en la vista de manera declarativa mediante el atributo *Command* cuyo valor de enlazamiento *Binding* tiene que ser

la propiedad que representa el comando definido en FaceViewModel. Con el atributo *CommandParameter* se pasa el color actualmente seleccionado para pintar las mejillas.

```
<controls:GelMakeupPolygonButton GelColor="{Binding Path=Color,
Converter={StaticResource ProductColorConverter}}"

Command="{Binding DataContext.DrawBlushCommand,
RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type
views:FaceMakeupView}}}"

CommandParameter="{Binding Path=Color, Converter={StaticResource
ProductColorConverter}}"

ToolTip="{Binding Path=Name}">
</controls:GelMakeupPolygonButton>
```

Observad también el atributo *Converter* que se inicializa con el recurso local que representa el convertidor. Se encarga de a partir de la representación hexadecimal del color obtenido desde el fichero XML, crear el objeto de tipo *Color*.

5.4-. MEDIATOR. COMUNICACIÓN ENTRE VIEWMODELS

Mediator es el patrón similar a un sistema de mensajería que permite la comunicación entre las distintas clases del MVVM. En nuestro caso lo hemos usado para comunicar ViewModels. Para llevar a cabo el proceso de comunicación, el productor publica el mensaje y luego todos los consumidores interesados en consumir el mensaje se registran en el canal. Con esto conseguimos acoplamiento muy bajo entre los componentes de nuestra aplicación, debido a que no se comunican directamente sino a través de Mediator.

Para demostrar el uso de este patrón en MakeupStylist vamos a seguir con el ejemplo de pintado de mejillas. Como ya hemos visto anteriormente, cada vez que se realiza el pintado de mejillas, se invoca al método *redrawLayers*.

```
private void redrawLayers() {

Mediator.Mediator.Instance.NotifyColleagues(ViewModelMessages.RedrawLayers,
true);
 }
```

Este método obtiene la referencia estática del objeto Mediator, y notifica a todos los objetos suscritos al mensaje ViewModelMessages.RedrawLayers, también llamados objetos Colleagues. Su propósito es indicar que las capas han de redibujarse para reflejar los cambios. En el siguiente fragmento de código podemos ver la implemetación del método NotifyColleagues.

```
public void NotifyColleagues(ViewModelMessages message, object args)
{
     if (internalList.ContainsKey(message))
     {

         foreach (Action<object> callback in internalList[message])
              callback(args);
     }
}
```

Simplemente lo que hace es invocar la acción (función callback) para cada tipo de mensaje registrado en el diccionario.

Por otro lado, la clase interesada para poder consumir el mensaje, antes tiene que registrarse. En el Mediator esa acción la realiza el siguiente método.

```
public void Register(Action<Object> callback, ViewModelMessages message)
{
     internalList.AddValue(message, callback);
}
```

En nuestro caso el ViewModel encargado de consumir el mensaje es MainWindowViewModel mediante este fragmento de código.

```
private void doRegisterLayerRedraw() {
            Mediator.Mediator.Instance.Register((Object o) => {
RedrawLayers = (bool) o; }, ViewModelMessages.RedrawLayers);
        }

        /// <summary>
        /// Se asigna al redibujar las capas.
        /// </summary>
        private bool RedrawLayers
        {
            set
            {
                GlobalContext.LoadedImageControl.Source =
LayerManager.Instance.doDrawLayers();
            }
        }
}
```

Siempre que llega un mensaje de tipo *RedrawLayers*, se invocará la propiedad RedrawLayers, estableciendo la nueva imagen en la superficie de dibujo, con las capas debidamente actualizadas.

Con este simple patrón hemos conseguido reducir y en algunos casos prácticamente eliminar las dependencias entre los diferentes ViewModels.

A continuación mostramos la infraestructura del proyecto en el entorno Visual Studio. *(Véase Imagen 5.2)*

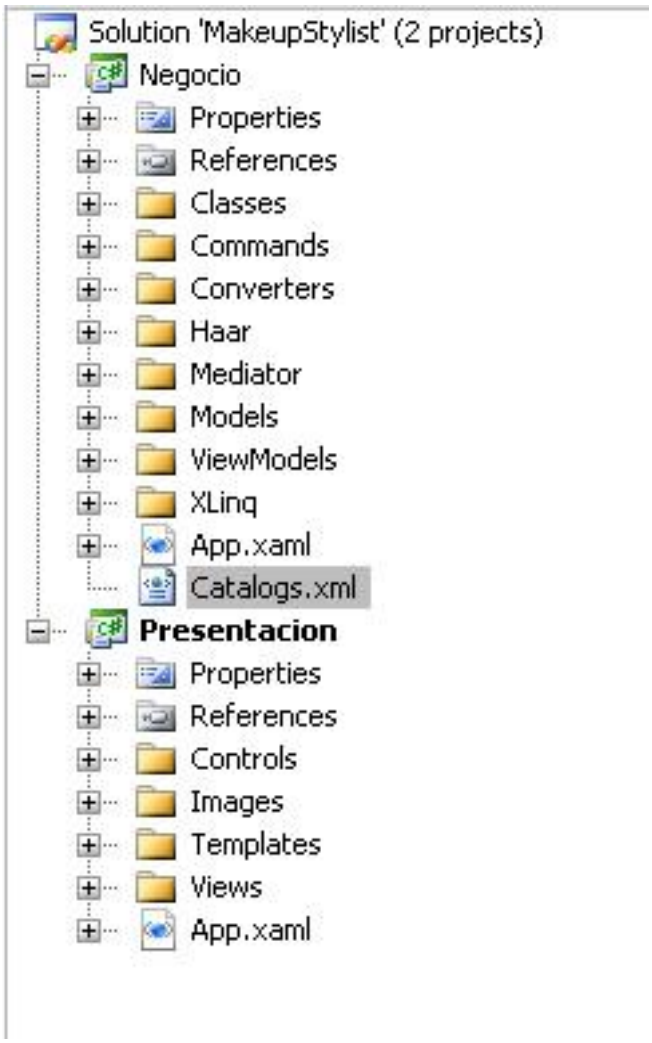


Imagen 5.2: Estructura del proyecto

6-. EL MODELO DE DATOS Y PERSISTENCIA

6.1-. MODELO DE DATOS

El diagrama de clases (UML) *(Véase Imagen 6.1)* de MakeupStylist se muestra en la siguiente imagen. Los elementos de color azul representan las clases bases de las cuales toda las demás van a heredar. La clase *BaseDetector* contiene la funcionalidad básica propia de cada detector (facial, ojos, rostro) y define los diferentes objetos de entrenamiento *HaarCascade*. Se ha ofrecido la posibilidad de parametrazición de tipos en función del tipo de detector a usar. La clase *BaseMakeup* también se ha creado como clase base en la jerarquía de herencia, aunque en realidad no contiene funcionalidad ninguna y se deja en caso de futuras ampliaciones.

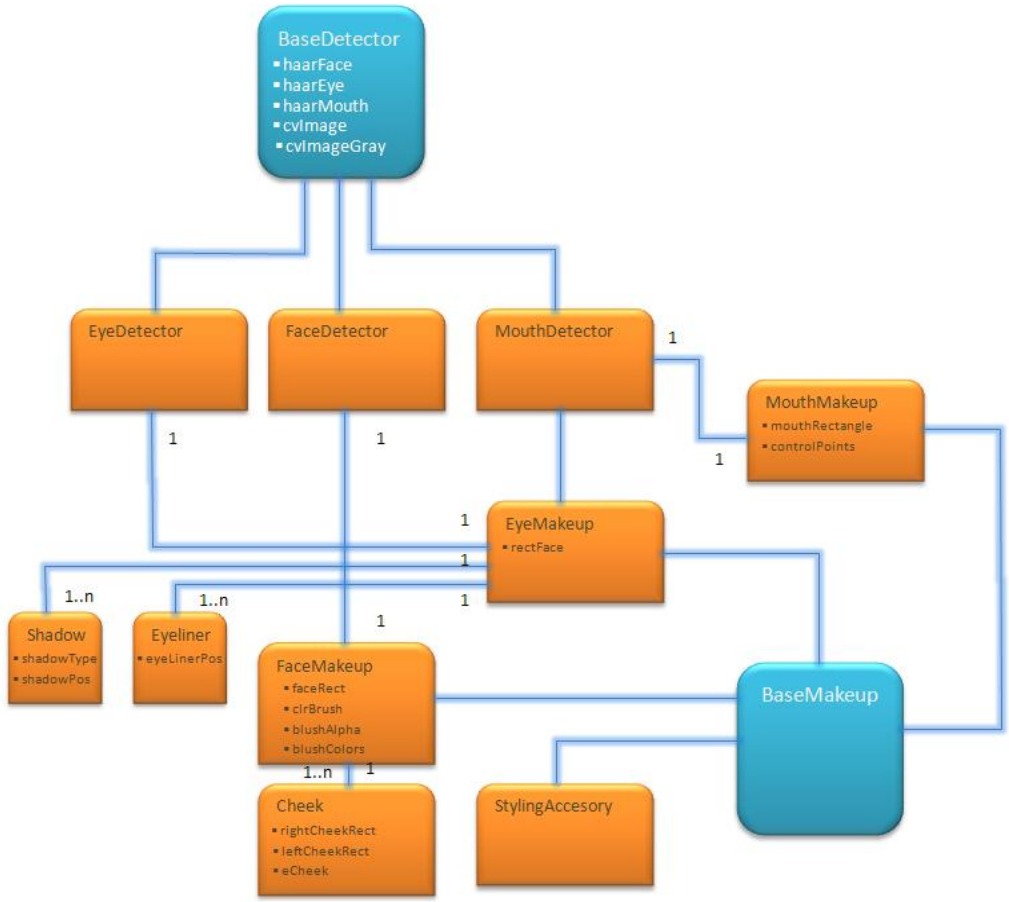


Imagen 6.1: Modelo de datos

```
public abstract class BaseDetector<T> : IBaseDetector {

        /// <summary>
        /// El archivo haar de entrenamiento.
        /// </summary>
        protected HaarCascade haarFace = new
HaarCascade(Constants.FACE_HAAR_XML);
        /// <summary>
        /// El archivo haar de entrenamiento.
        /// </summary>
        protected HaarCascade haarEye = new
HaarCascade(Constants.EYE_HAAR_XML);
        /// <summary>
        /// El archivo haar de entrenamiento.
        /// </summary>
        protected HaarCascade haarMouth = new
HaarCascade(Constants.MOUTH_HAAR_XML);

        /// <summary>
        /// Imagen sobre cual se realizará la detección / pintado:
        /// </summary>
        private Image<Bgra, Byte> cvImage;
        /// <summary>
        /// Imagen original pasada a escala de grises.
        /// </summary>
        protected Image<Gray, Byte> cvImageGray;
}
```

En la clase FaceDetector se definen varios métodos para la detección del rostro usando para ellos diferentes técnicas y algoritmos. Una vez detectadas todas las partes de interés, se procede al pintado sobre la imagen original. Por ejemplo la clase *Cheek* representa la mejilla a pintar, basándose en los rectángulos obtenidos previamente por el detector.

```
private class Cheek
{
      /// <summary>
      /// El rectángulo delimitador de la cara.
      /// </summary>
      private Rectangle m_rightCheekRect = new Rectangle();
      private Rectangle m_leftCheekRect = new Rectangle();
      /// <summary>
      /// Indica que mejilla se pintará.
      /// </summary>
      public enum CheekEnum
      {
            /// <summary>
            /// Se pinta la mejilla izquierda.
            /// </summary>
            Left,
            /// <summary>
            /// Se pinta la mejilla derecha.
            /// </summary>
            Right
      }
      private CheekEnum m_eCheek = CheekEnum.Left;

      public Cheek(CheekEnum cheekEnum) {

            this.m_eCheek = cheekEnum;

      }

      private void doInitRects(Rectangle faceRect) {

            m_rightCheekRect = FaceDetector.cheekR;

            m_leftCheekRect = FaceDetector.cheekL;

      }

      public void drawCheekFill(Rectangle faceRect, List<BlendMakeupColor>
colors )
      {
            Bitmap cheekBitmap = new Bitmap(getBlushWidth(faceRect),
getBlushHeight(faceRect));
            doInitRects(faceRect);
            using (Graphics g = Graphics.FromImage(cheekBitmap))
            {
                  createCheekFill(g, colors, faceRect);
            }
            if (m_eCheek == CheekEnum.Left)
            {

LayerManager.Instance.addLayer(LayerManager.LayerType.LeftCheek, new
Layer(cheekBitmap, new Point((m_leftCheekRect.Left + m_leftCheekRect.Width / 2)
- getBlushWidth(faceRect) / 2, m_leftCheekRect.Top +
getAdjustFactor(faceRect)), "Blush",LayerManager.LayerType.LeftCheek));
            }
            else
            {

LayerManager.Instance.addLayer(LayerManager.LayerType.RightCheek, new
Layer(cheekBitmap, new Point((m_rightCheekRect.Left + m_rightCheekRect.Width /
2) - getBlushWidth(faceRect) / 2, m_rightCheekRect.Top +
getAdjustFactor(faceRect)), "Blush",LayerManager.LayerType.RightCheek));
            }
     }

}
```

En el método *drawCheekFill* se crea un mapa de bits donde inicialmente se pintará la capa correspondiente a la mejilla. Para ellos se usan las funciones del API de GDI+. Una vez

dibujada la capa, se invoca al método *addLayer* de la clase *LayerManager* para añadir la capa en la lista. Las clases *Shadow* que representa la sombra, *Eyeliner* el pincel de ojos, usan técnicas similares para realizar el pintado de las capas. Queda por mencionar la clase *StylingAccessory* encargada de dibujar elementos accesorios.

```
public class StylingAccesory : BaseMakup
    {
        public StylingAccesory() { }

        public void doWearGlasses(string path)
        {
            Bitmap btm = new Bitmap(path);
            Bitmap glasses = getProportionalGlasses(btm);
            LayerManager.Instance.addLayer(LayerManager.LayerType.Glasses,
new Layer(glasses, new Point(EyeDetector.getGlasses().X,
EyeDetector.getGlasses().Y), "Glasses", LayerManager.LayerType.Glasses));
        }

        private Bitmap getProportionalGlasses(Bitmap btmGlasses)
        {
            double prop = 0.0;
            double anchoC = EyeDetector.getGlasses().Width;
            double anchoG =  btmGlasses.Width;
            prop =  anchoC/anchoG;

            int anchoN = int.Parse((Math.Truncate((btmGlasses.Width) *
prop)).ToString());
            int altoN = int.Parse((Math.Truncate(btmGlasses.Height *
prop)).ToString());

            btmGlasses = new Bitmap(btmGlasses, new Size(anchoN, altoN));
            return btmGlasses;
        }

    }
```

Antes de dibujar el elemento, es necesario calcular las proporciones de la imagen fuente, y en función de ello escalar el mapa de bits destino antes de que se efectué el pintado. También mencionar la clase *MouthMakeup* que dispone de métodos para dibujar el contorno del labio usando para ellos las curvas Bézier.

6.2-. PERSISTENCIA DE DATOS

Para persistir la información de catálogos y productos de maquillaje asociados, MakeupStylist usa como soporte un archivo XML con la siguiente estructura:

```
<catalogs>
 <catalog type="Pintalabios" name="Loreal">
  <products>
   <product>
    <name>Natural Shimmer (N)</name>
    <color>#950000</color>
   </product>
 </catalog>
</catalogs>
```

Como podemos apreciar, todos los catálogos se encuentran dentro del elemento raíz <*catalogs*> y tienen dos atributos – *type* que determina el tipo de catálogo y *name* que se corresponde con el nombre del mismo. Cada catálogo puede tener indeterminado número de elementos <*product*> que representan el producto de maquillaje. A su vez estos contienen dos elementos, *name* que determina el nombre del producto y *color* que es la cadena en formato hexadecimal para representar los niveles del color (ARGB).

La escritura y obtención de elementos desde el fichero XML se lleva a cabo mediante la tecnología XLINQ(XML Language Integrated Query) disponible a partir de la versión 3.5 del .NET framework. Esta tecnología dota al lenguaje de programación C# de operadores que permiten realizar consultas estilo lenguaje SQL. Las ventajas de XLINQ se reflejan en la facilidad del uso de su API, su robustez, y rendimiento, combinados con las características del lenguaje como la genericidad. Veamos, como se ha implementado el mecanismo de persistencia en MakeupStylist. En primer lugar, se ha definido la clase base abstracta de la que van a heredar todos los objetos DAO (Data Access Object) encargados de consultar o escribir información en el almacén de catálogos (fichero XML).

```
public abstract class AbstractXLinqDao {

        protected XDocument xml;

        public AbstractXLinqDao(Stream catalogXmlStream) {
           this.xml = XDocument.Load(new XmlTextReader(catalogXmlStream));
        }

        public void save(Stream catalogXmlStream) {
           this.xml.Save(new XmlTextWriter(catalogXmlStream, Encoding.UTF8));
        }
    }
```

Esta clase abstracta declara un atributo protegido que representa el documento XML. En el constructor de la misma se realiza la carga del documento desde un el flujo de datos obtenido a partir del recurso embebido. El método *save* realiza la escritura sobre el flujo pasado como parámetro. Una vez definida la clase base, creamos la interfaz con todas las operaciones posibles que se pueden realizar sobre el almacén de catálogos.

```
public interface ICatalogXLinqDao {

         List<CatalogDto> getCatalogs();

         CatalogDto getCatalog(String catalogName);

         void addNewCatalog(String catalogName, String catalogType);

         void delCatalog(String catalogName);

         void updateCatalog(String catalogName, String newCatalogName);

         void addNewProduct(String productName, String catalogName, String
color );

         void delProduct(String catalogName, String productName);

         void updateProduct(String catalogName, String productName, String
newProductName, String newProductColor);
    }
```

Finalmente creamos la clase que heredará la clase base e implementará la interfaz anterior.

```
public class CatalogXLinqDao : AbstractXLinqDao, ICatalogXLinqDao {
        private Stream catalogXmlStream;
        public CatalogXLinqDao(Stream catalogXmlStream)
            : base(catalogXmlStream) {
             this.catalogXmlStream = catalogXmlStream;
        }

        public List<CatalogDto> getCatalogs() {
            List<CatalogDto> lCatalogs = new List<CatalogDto>();
            List<ProductDto> lProducts = new List<ProductDto>();
            IEnumerable<XElement> catalogs = ( from c in
xml.Root.Elements("catalog") select c );
            foreach (XElement catalog in catalogs ) {
                  CatalogDto cat = new CatalogDto();
                  cat.Name = catalog.Attribute("name").Value;
                  cat.Type = catalog.Attribute("type").Value;
                  if (catalog.Element("products") != null) {
                      IEnumerable<XElement> products = ( from prod in
catalog.Element("products").Elements("product") select prod );
                      foreach( XElement prod in products ) {
                          ProductDto p = new ProductDto();
                          p.Name = prod.Element("name").Value;
                          p.Color = prod.Element("color").Value;
                          p.CatalogName = cat.Name;
                          lProducts.Add( p );
                      }
                      cat.Products = lProducts;
                  }


                  lCatalogs.Add( cat );

            }

            return lCatalogs;

        }
```

El método *getCatalogs()* devolverá la lista de objetos de tipo *CatalogDto* con la información de todos los catálogos recuperados desde el fichero XML. Obsérvese la facilidad de uso del API XLINQ y la similitud de las construcciones con las consultas SQL.

```
IEnumerable<XElement> catalogs = ( from c in xml.Root.Elements("catalog")
select c );
```

Esta instrucción devuelve la lista de elementos de tipo *XElement* de todos los nodos *catalog* encontrados en la raíz del documento. A partir de cada catálogo encontrado, se obtienen todos los productos pertenecientes al catálogo. En la escritura de información se invoca al método *save* definido en la clase base.

```
public void addNewCatalog(String catalogName, String catalogType)

{

    xml.Root.Add(new XElement("catalog", new XAttribute("name", catalogName),
new XAttribute("type", catalogType), new XElement("products")));

     save(catalogXmlStream);

}
```

6.3-. CLASES

Además del modelo de datos, MakeupStylist también usa una serie de clases para llevar a cabo operaciones de distinta índole. En la tabla adjunta se listan dichas clases junto con su descripción.

Nombre clase	Descripción
LayerManager	Dispone de métodos que permiten realizar el pintado de capas, dependiendo del tipo.
FaceProportions	Permite obtener el rectángulo delimitador de las mejillas y ojos basándose en las proporciones faciales.
GlobalContext	Contiene referencia a la imagen actualmente cargada en la superficie de dibujo de MakeupStylist.
Helper	Clase utilitaria con los métodos para persistir en disco el resultado de las operaciones de pintado sobre la imagen.
ImageUtils	Operaciones de conversión entre el Bitmap y el tipo de dato compatible con el framework WPF.
IOUtils	Dispone de métodos para obtener la ruta relativa de los archivos Haar.
Layer	Representa la capa en MakepStylist, que

	entre otros atributos contiene el mapa de bits, rectángulo delimitador de la capa, nombre, etc.
LayerObservableCollection	Tipo de colección que hereda la clase ObservableCollection<Layer>.
MakeUpUtils	Usa los métodos de GDI+ para realizar pintados de varios colores.
MakeUpColors	Contiene las constantes que identifican los colores de los productos de maquillaje.
NativeMethods	Contiene declaraciones de los métodos nativos del API de Windows.
Pixel	Clase para representar el pixel.
WebCam	Inicializa la webcam y ofrece métodos para obtener la captura de imagen desde la misma.
NonFaceDetectableException	Excepción que se lanza cuando no se detecta el rostro en la imagen proporcionada.

7-. CONCLUSIONES

Con este proyecto, hemos tenido la posibilidad de familiarizarnos con el apasionante mundo de la Inteligencia Artificial y la Visión por Computador. Hemos utilizado varias librerías que apuestan por este mundo como son OpenCv, OpenCVDotnet y por último Emgu que ha sido la elegida para el desarrollo por ser las que más posibilidades nos ofrecía.

Los algoritmos Haar, han sido todo un descubrimiento, hemos comprobado su potencia y eficiencia. Nos hemos informado sobre la forma de crearlos mediante el entrenamiento con imágenes.

Hemos descubierto fórmulas que definen la cromática de la piel humana y las hemos sintetizado en algoritmos que analizan los píxeles de una imagen y determinan aquellos susceptibles de contener piel.

Con este proyecto nos hemos sumergido en el mundo de WPF. Hemos podido trabajar conjuntamente bajo un patrón de diseño Model View Viewmodel (MV-VM) que nos ha facilitado la comprensión del código. A su vez también nos ha ahorrado mucho trabajo, ya que gracias a las nuevas clases que proporciona WPF junto con MV-VM no hemos tenido que realizar el refresco de las interfaces, porque se ejecuta automáticamente.

Con WPF, hemos podido observar la evolución en las interfaces. Se han adaptado a los nuevos tiempos, pudiendo tener el trabajo de un diseñador y un programador totalmente independientes, lo que ahorra molestias y simplifica el trabajo.

8-. MANUAL DE USUARIO

8.1-. PRESENTACIÓN DE LA INTERFAZ.

1- Barra de herramientas y otras opciones.

2- Panel de selección de productos.

3- Escritorio de trabajo.

4- Capturas con la webcam.

5- Catálogo: gestión de los productos.

6- Ventana donde se muestran las capas

7- Ventana donde se muestran los accesorios

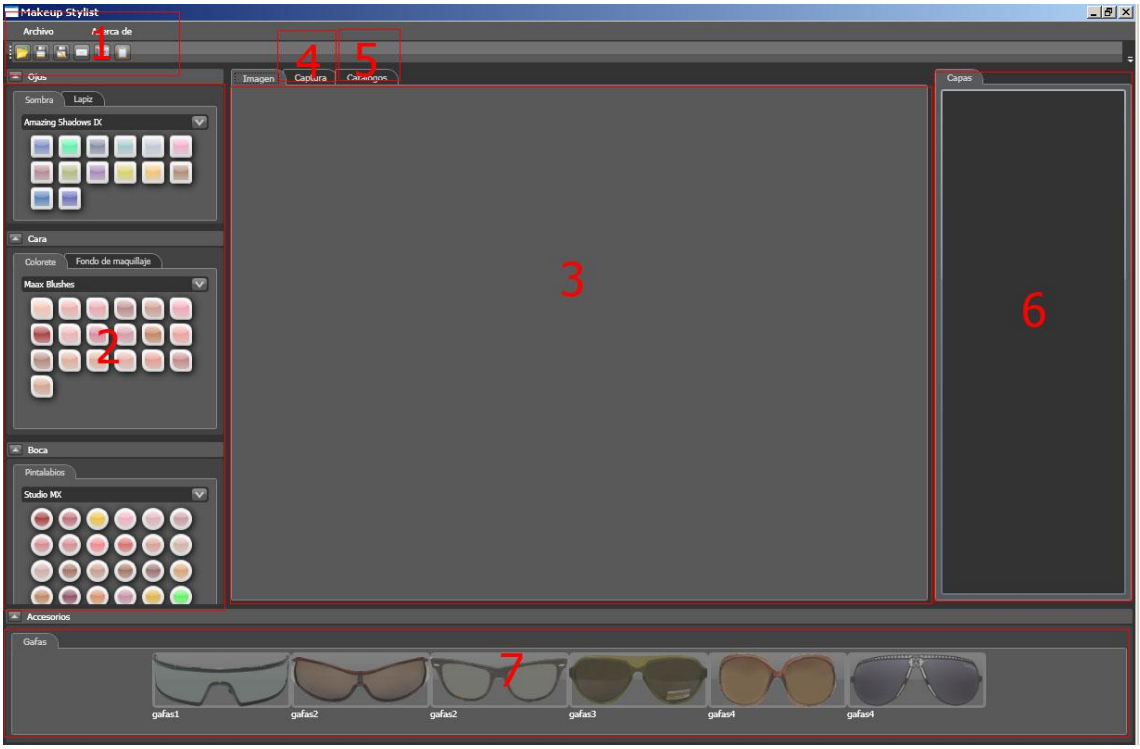


Imagen 8.1: Interfaz de usuario

8.2-. SELECCIONAR UNA IMÁGEN

Para empezar a utilizar la aplicación, primero debemos seleccionar una fotografía. Tenemos dos opciones, por un lado, importar un archivo de imagen guardado en nuestro ordenador.

Para ello, vamos al panel (1) y pulsamos sobre el icono de abrir archivo:  Se abre una ventana, para que seleccionemos la imagen a tratar. Una vez cargada la foto nos aparecerá en el escritorio de trabajo (3).

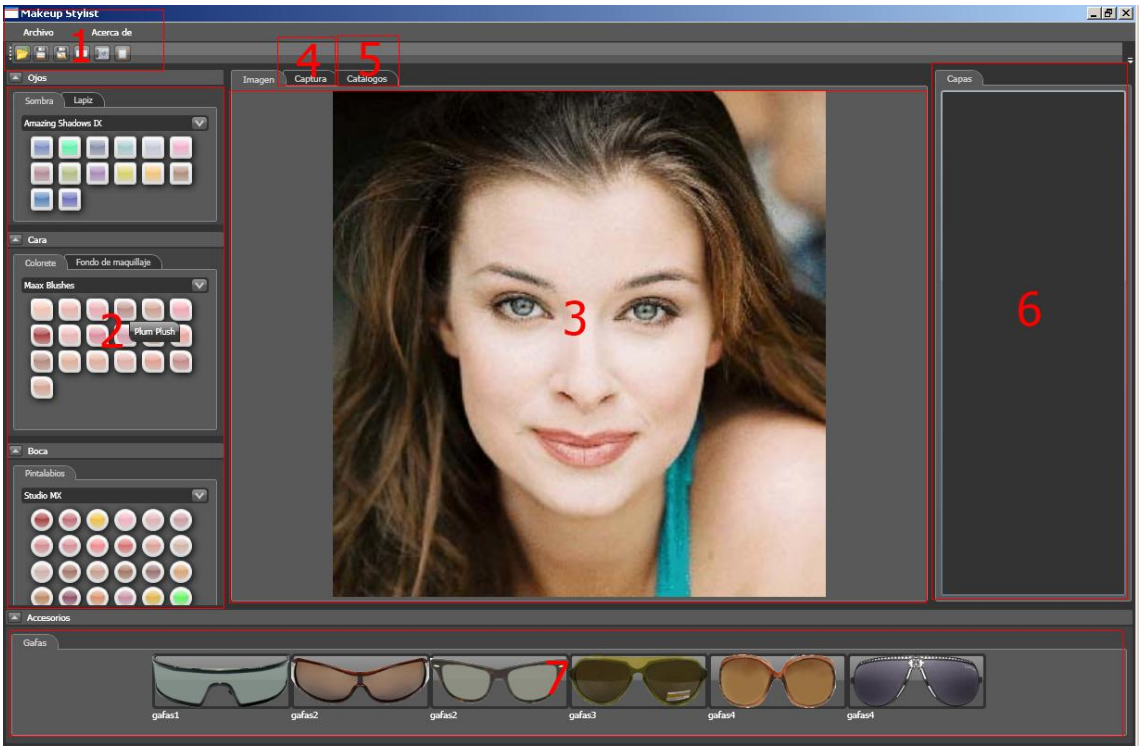


Imagen 8.2: Interfaz de usuario tras cargar una imagen de archivo

Por otro lado, podemos hacer una captura con la webcam:

Para utilizar una captura hecha con la webcam, tenemos que dirigirnos a la pestaña Captura (4). A partir de este instante se muestra aquello que está capturando la webcam. Para capturar una imagen basta con dirigirnos a la barra de herramientas (1) y pulsar el icono  o simplemente pulsar la tecla espacio. Podemos hacer esto cuantas veces queramos, las

imágenes capturadas se irán almacenando en una galería situada en la parte inferior de esta ventana (8).

8-Galería donde se almacenan las imágenes capturadas

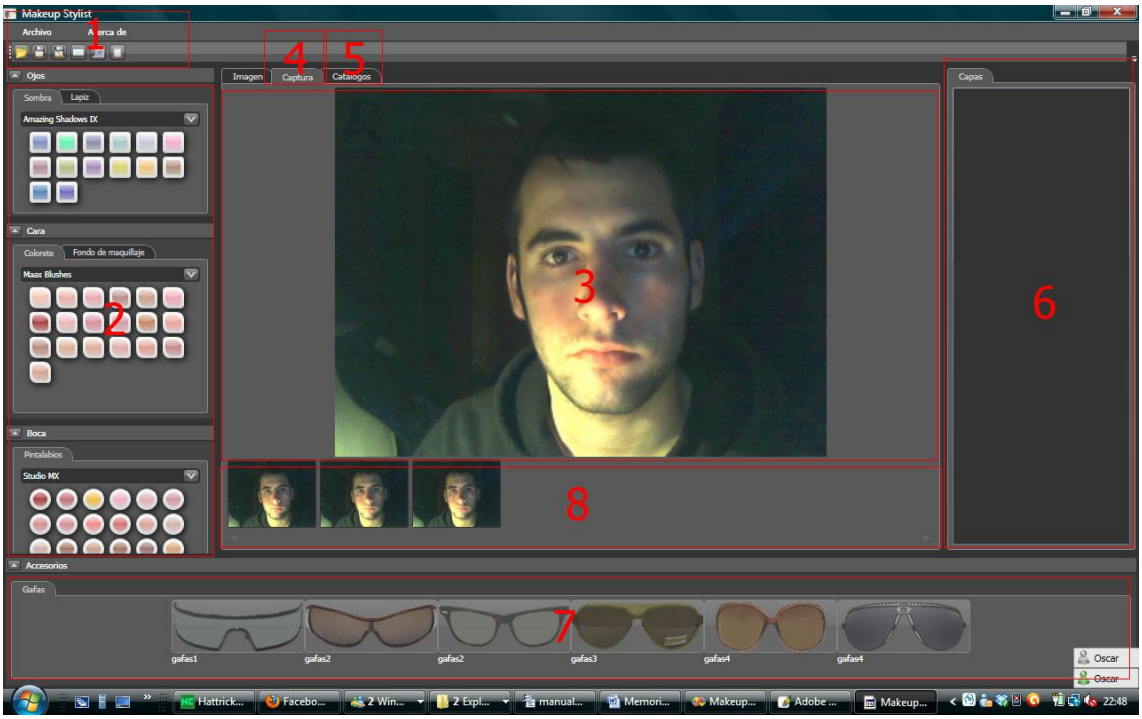


Imagen 8.3: Interfaz de usuario tras cargar una imagen capturada con la webcam

Ahora pinchamos sobre una de las fotos y cuando cambiamos a la pestaña de Imagen que es nuestro escritorio de trabajo, tenemos la imagen a tratar.

8.3-. APLICAR FILTROS DE MAQUILLAJE

A partir de este momento ya podemos empezar a trabajar con la foto.

Para ello iremos al panel de selección de productos (2). Por ejemplo, vamos a maquillar la sombra de ojos de la foto.

Buscamos el filtro de la sombra de ojos y seleccionamos el color que deseemos. Posteriormente pulsamos sobre él.

Imagen 8.4: Panel de selección de productos

En este momento veremos 2 cambios en el programa. Primero podremos observar como la foto ha sido maquillada.

Imagen 8.5: Cambio efectuado en la imagen inicial

Y como en el panel con el listado de las capas (6) han sido añadidos dos elementos. Uno por cada ojo.

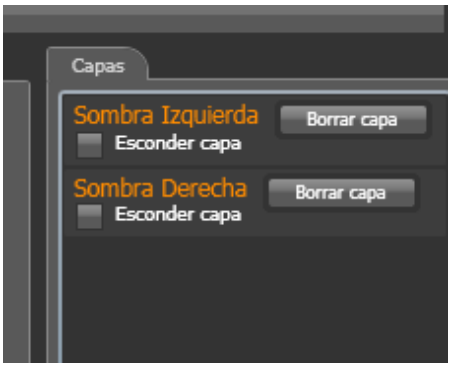


Imagen 8.6: Panel de gestión de capas

Con la ayuda de este menú, podremos esconder la capa seleccionada pulsando sobre el "tick " Esconder Capa.

Imagen 8.7: Efecto al ocultar una capa

Tras seleccionar todos los filtros deseados, podemos proceder al guardado del proyecto en un archivo. Para eso pulsaremos sobre el icono de guardar en la barra de herramientas(1)  e indicamos el fichero donde queramos guardarlo. Como se detalló en el apartado 3.3 se guarda

un fichero con extensión ".mkup", que contiene la información detallada por capas, para poder volver a cargarse en el programa y seguir trabajando posteriormente.

8.4 GESTIÓN DE CATÁLOGOS.

En la tercera pestaña (5) podemos configurar los colores que se mostraran en los filtros. (2).

En esta parte se nos muestra una lista de catálogos de colores y para qué tipo de filtro está relacionado. Todo esto se carga/guarda en un archivo XML.

1-Catálogos

2-Productos

3-Gestión de productos

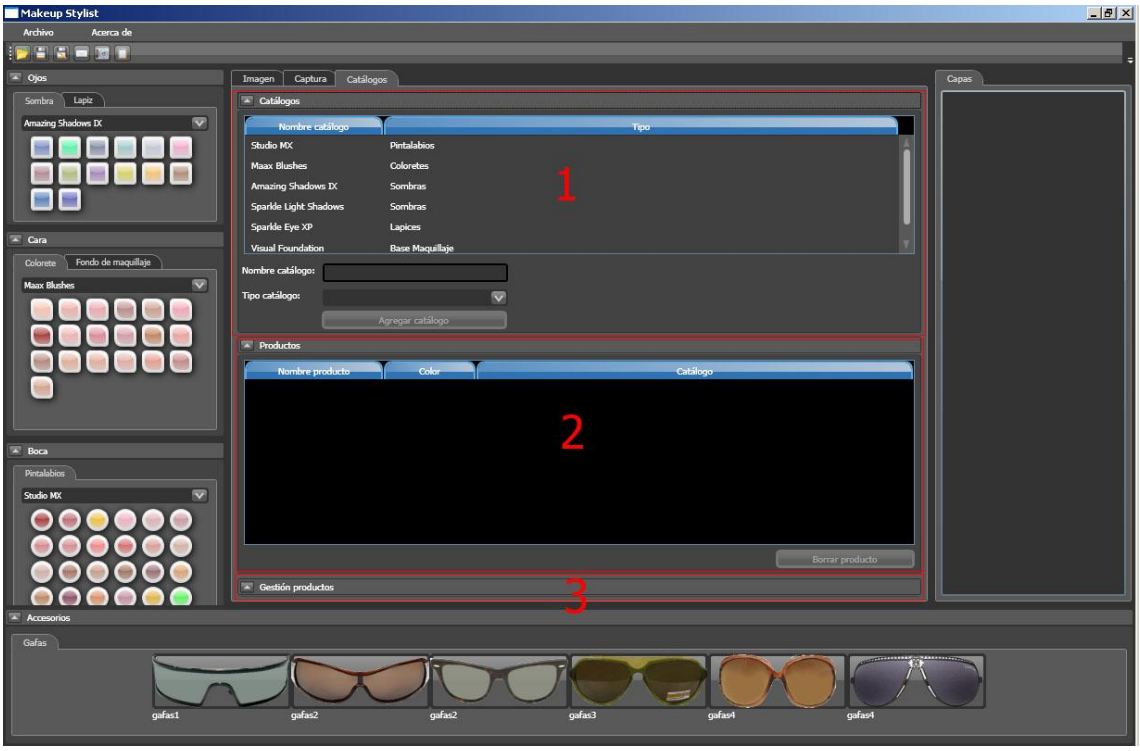


Imagen 8.8: Interfaz de usuario. Gestión de productos

Si quisiéramos modificar los colores del catálogo. Seleccionamos el catálogo(1) a cargar. Se nos cargara en la lista inferior, los colores que componen el catálogo.

Para borrar un color (producto). Lo seleccionamos y pulsamos el botón "Borrar Producto".

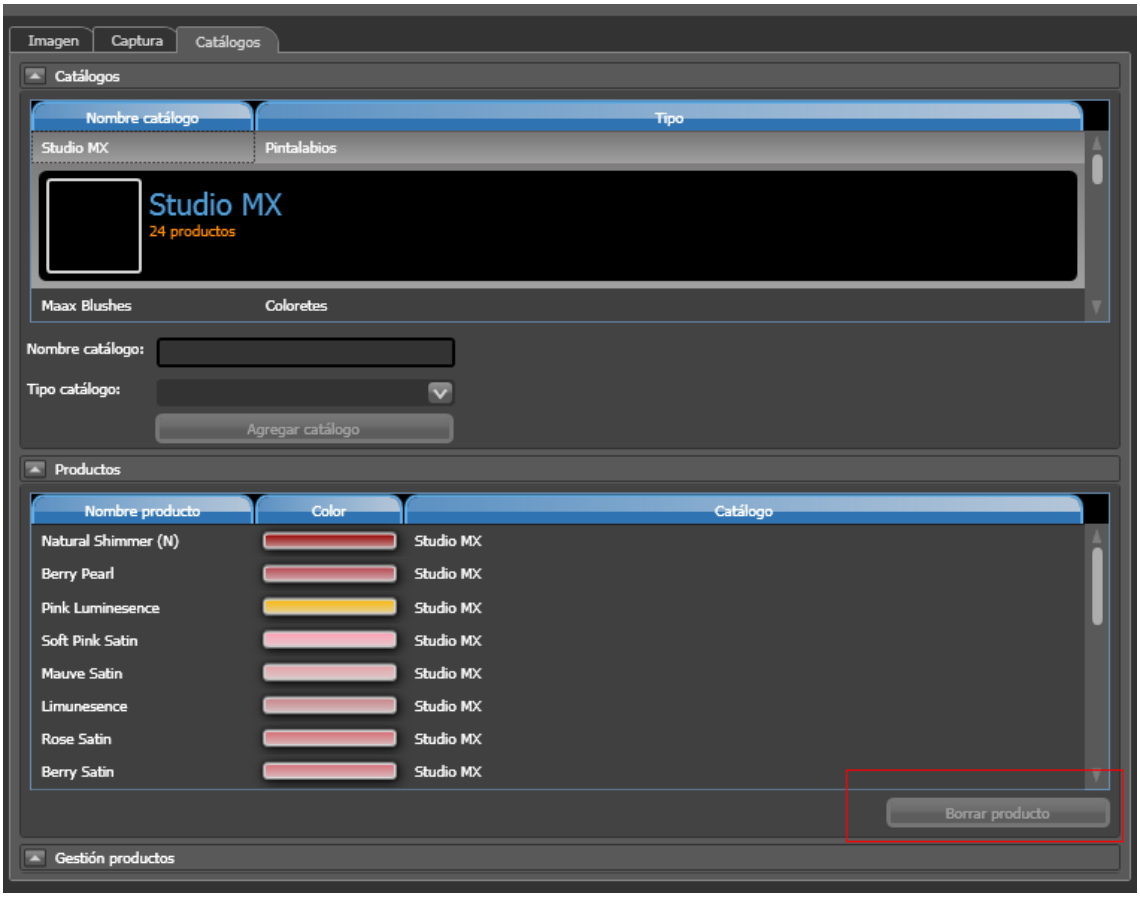


Imagen 8.9: Panel de gestión de productos. Borrar un producto

Para crear un producto, desplegamos el menú Gestión Productos (3) Seleccionamos el color del producto, insertamos el nombre y pulsamos el botón "Agregar producto" para agregar el producto al catálogo.

Imagen 8.10: Panel de gestión de productos. Dar de alta un producto.

9-. FUTUROS DESARROLLOS

Este proyecto deja muchas posibilidades para hacer futuros desarrollos, a continuación exponemos algunas ideas:

Enriquecer la funcionalidad del programa introduciendo otros complementos como pelucas, piercings, gorras, sombreros, diademas…

Enriquecer el programa ofreciendo también la detección de caras de perfil.

Enfocar el programa en vez de al maquillaje realista a la creación de personajes ficticios, añadiendo para ello otras herramientas de maquillado y accesorios, por Ej: tipos de orejas, pelucas, maquillaje con varios colores incluso con texturas…

Conseguir bases de datos con imágenes que contengan caras u ojos o boca y hacer algoritmos de detección propios que mejoren las prestaciones de los proporcionados por OpenCV.

Como hemos comentado anteriormente, la última versión de Emgu, permite optimizar el tiempo de detección empleando unas gráficas que acepten CUDA. Un posible futuro desarrollo es la implementación de la detección de la boca y ojos para que acepte este algoritmo, la parte de la detección de una cara está implementada, pero habría que probar con dicha gráfica que el algoritmo es correcto. Esto abriría la posibilidad de hacer un estudio de eficiencia comparando los tiempos obtenidos empleando o no el CUDA.

Mejorar y enriquecer el algoritmo detector de la piel para que sea lo más correcto posible, evitando confundir zonas de pelo con las de la piel.

Otra de las bazas a explotar en este proyecto, seria la programación multiprocesador. Se podría gestionar la carga de los diferentes procesos repartiendo el trabajo en diferentes núcleos.

Si se desarrollara la idea de programación multiprocesador, se podrían superar las limitaciones para mostrar el resultado en forma de vídeo. Así, además de desarrollar el proceso de maquillaje en foto, se podría realizar en video. Actualmente, el coste con un procesador para realizar todo esto, es tan elevado que hemos preferido no continuar el desarrollo en esa dirección.

10-. BIBLIOGRAFÍA:

[Ahlberg 1999] AHLBERG, J. 1999. A system for face localization and facial feature extraction. Tech. Rep.

LiTH-ISY-R-2172, Linkoping University.

[Brand and Mason 2000] BRAND, J., AND MASON, J. 2000. A comparative assessment of three approaches to pixellevel human skin-detection. In *Proc. of de International Conference on Pattern Recognition*.

[Vezhnevets et al. 2003] VEZHNEVETS, V., SAZONOV, V., AND ANDREEVA, A. 2003. A Survey on Pixel-Based Skin Color Detection Techniques. In *Proc. Graphicon-2003*, pp. 85 – 92. Moscow, Rusia.

[Chen et al. 1995]CHEN, Q., WU, H., AND YAHCIDA, M. 1995. Face detection by fuzzy pattern machining. In *Proc. of the Fifth International Conference on Computer Vision*.

[Esqueda 2002] ESQUEDA, J.J. 2002. Fundamentos de procesamientos de imágenes. CONACTEC 2002. Instituto Tecnológico de Ciudad Madero. México.

[Gomez 2000] GOMEZ, G. 2000. On selecting colour components for skin detection. In *Proc. of the ICPR*, vol. 2.

[Gomez and Morales 2002] GOMEZ, G., AND MORALES, E. 2002. Automatic feature construction and a simple rule induction algorithm for skin detection. In *Proc. of the ICML Workshop on Machine Learning in Computer Vision*.

[Hsu et al. 2002] HSU, R. L., ABDEL-MOTTALEB, M., AND JAIN, A. K. 2002. Face detection in color images. *IEEE Trans. Pattern Analysis and Machine Intelligence 24*, 5.

[Jordao et al. 1999] JORDAO, L., PERRONE, M., COSTEIRA, J., AND SANTOS-VICTOR, J. 1999. Active face and feature tracking. In *Proceeding of the 10th International Conference on Image Analysis and Processing*.

[Menser and Wien 2000] MENSER, B., AND WIEN, M. 2000. Segmentation and tracking of facial regions in color image sequences. In *Proc. SPIE Visual Communications and Image Processing 2000*.

[Oliver et al. 1997] OLIVER, N., PENTLAND, A., AND BERARD, F. 1997. Lafter: Lips and face real time tracker. In *Proc. Computer Vision and Pattern Recognition*.

[Peer et al. 2003] PEER, P., KOVAC, J., AND SOLINA, F. 2003. Human skin colour clustering for face detection. In *submitted to EUROCON 2003 – International Conference on Computer as a Tool*..

[Saber and Tekalp 1998] SABER, E., AND TEKALP, A. 1998. Frontal-view face detection and facial feature extraction using color, shape and symmetry based cost functions. In *Pattern Recognition Letters*, vol.9, 669 – 680.

[Schied 1997] SCHIED, A., 1997. Geometric aspects of Fleming-Viot and Dawson-Watanebe processes. Humboldt-Universität zu Berlin. In *The annals of Probability*, 1997, vol. 25, Nº 3.

[Schumeyer and Barner 1998] SCHUMEYER, R., AND BARNER, K. 1998. A color-based classifier for region identification in video. In *Visual Communications and Image Processing 1998*, SPIE.

[Terrillon et al. 2000] TERRILLON, J. C., SHIRAZI, M. N., FUKAMACHI, H., AND AKAMATSU, S. 2000. Comparative performance of different skin chrominance models and chrominance spaces for the automatic detection of human faces in color images. In *Proc. of the International Conference on Face and Gesture Recognition*.

[Vélez et al. 2003] VÉLEZ, J., SANCHEZ, A., MORENO, A. B., ESTEBAN, J. L. 2003. Visión por computador. Dykinson S. L.

[web1] http://www.cs.bu.edu/techreports/pdf/2005-012-blink-detection.pdf

[web2] http://miron.disca.upv.es/VxC/Trabajos/Detecci%F3n%20de%20caras%20y%20an%E1lisis%20de%20expresiones%20faciales.pdf

[web3] http://www.google.es/url?sa=t&source=web&cd=1&sqi=2&ved=0CCAQFjAA&url=http%3A%2F%2Frepository.tudelft.nl%2Fassets%2Fuuid%3A13074be2-bf06-4c9b-8260-9f84c2db8fd4%2Fewi_Choy!Chung!Harahap!Natadarma!Wu.pdf&rct=j&q=ewi_Choy!Chung!Harahap!Natadarma!Wu.pdf&ei=Cr-ZTomzCsfoOdrZsIkK&usg=AFQjCNEwZ_6LpVOtNRfISbOMXt9SuKTWcQ&sig2=1Zdpgfy8ab7Hs36kyUPsaw&cad=rja

[web4] http://cpdsi-fich.wdfiles.com/local--files/tpsaplicacion/2005_Ramello-Piel.pdf

[web5] http://www.codeproject.com/KB/cs/Intel_OpenCV.aspx?fid=1525118&df=90&mpp=25&noise=3&sort=Position&view=Quick&select=2699733

[web6] http://www.codeproject.com/KB/cpp/TrackEye.aspx?fid=1403145&df=90&mpp=25&noise=3&sort=Position&view=Quick&select=2726159

[web7] http://www.codeproject.com/KB/cpp/TrackEye.aspx

[web8] http://iie.fing.edu.uy/investigacion/grupos/gti/timag/trabajos/2005/caras/Features/deteye.htm

[web9] http://www.ubuntu-es.org/index.php?q=node/51097

[web10] http://www.bernardotti.it/portal/showthread.php?p=78640

[web11] http://www710.univ-lyon1.fr/~bouakaz/OpenCV-0.9.5/docs/ref/OpenCVRef_Experimental.htm#decl_cvHaarDetectObjects

[web12] http://www.cs.indiana.edu/cgi-pub/oleykin/website/OpenCVHelp/

[web13] http://es.scribd.com/doc/4547752/opencv-objectdetection-2007june10

[web14] http://www.openframeworks.cc/addons/opencv

[web15] http://forum.openframeworks.cc/index.php?topic=432.0

[web16] http://seniordesign.deegrayve.net/Paper.pdf

[web17] http://madcomputerscientist.net/projects/face_extractor/report.pdf

[web18] http://www.google.com/notebook/public/13533147217413439515/BDRwQSwoQo664msYh

[web19] http://www.ralfwilke.com/download/puc/tarea1_ralf_wilke_imagenes.pdf

[web20] http://alereimondo.no-ip.org/OpenCV/uploads/41/tema2.pdf

[web21] http://note.sonots.com/SciSoftware/haartraining.html#f3d39ccf

[web22] http://www.gts.tsc.uvigo.es/PRESA/papers/URSI07.pdf

[web23] http://groups.google.com/group/opencvdotnet-discuss/unlock?_done=/group/opencvdotnet-discuss/browse_thread/thread/f75243b9c1527d6d/5b65e1bca15fa825%3Flnk%3Dgst%26q%3Dface

[web24] http://www.youtube.com/watch?v=5sPh6-J9wOA

[web25] http://www.emgu.com/wiki/index.php/Main_Page

[web26] http://code.google.com/p/opencvdotnet/