PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

INGENIERÍA INFORMÁTICA

Implementación de un plug-in para GStreamer para el seguimiento de rostros aplicado a Cheese

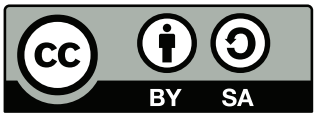
Tesis para optar por el título de Ingeniero Informático que presenta el bachiller:

César Fabián Orcccón Chipana

20105515

Asesor: Dr. Ivan Sipiran

22 de junio de 2018



Índice de figuras

Índice general

Capítulo 1

Generalidades

1.1. Problemática

Por más de 130 años las personas han usado cámaras fotográficas para documentar sus vidas (Snap Inc, 2017). Cuando las primeras cámaras fueron inventadas, era difícil y costoso tomarse una fotografía (Snap Inc, 2017). Un filtro fotográfico es un accesorio para las cámaras fotográficas que se inserta en la parte frontal de esta para conseguir cierto efecto deseado. La historia de los filtros fotográficos también es muy antigua, y se remonta al siglo XIX. En 1878, Frederick Wratten, un innovador e investigador del campo de la fotografía introdujo un proceso conocido como "noodling" que permitía crear placas fotográficas más sensitivos que las existentes en esa época (Hannavy, 2013). En 1906, él junto con su hijo y el doctor C. E. Kenneth Mees fundaron una compañía llamada Wainwright Ltd., en la cual Mees ayudó a Wratten a desarrollar las primeras placas pancromáticas y los primeros filtros de luz, los cuales se harían conocidos como Wratten filters (Hannavy, 2013). Kodak compraría más tarde Wainwright Ltd. en el 1912 (Hannavy, 2013). Desde entonces, los filtros fotográficos han ganado popularidad, y renovándose hasta la actualidad.

En las dos últimas décadas, la aparición de la fotografía digital ha dejado atrás a las cámaras de película (CIPA, 2008). En la actualidad, es común contar si no con una cámara digital, entonces al menos con una cámara web integrada al computador personal portátil o al telefóno inteligente. El término "filtro" (el cual de aquí y en adelante será entendido como "filtro digital") también se ha mudado al contexto digital y se ha introducido en el hablar del común de muchas personas. Las razones por las cuales las personas aplican filtros a sus fotografías son varias. Algunos aplican filtros para mejorar la estética regulando el brillo, contraste y focalización (Bakhshi, Shamma, Kennedy, y Gilbert, 2015). Otros desean aplicar efectos antiguos, por ejemplo algunos colocan las

fotografías en blanco y negro para concentrar la atención en cierta textura de la toma y evitar que el espectador se distraiga en los colores, otros simplemente desean darle un aspecto de antigüedad (Bakhshi y cols., 2015). Otras personas simplemente desean cambiar los colores (Bakhshi y cols., 2015). Finalmente, algunos otros simplemente desean darle un aspecto único y divertido a las fotografías (Bakhshi y cols., 2015). El único propósito de estos últimos es el agregar características divertidas (a través de filtros) que no pudieron ser capturadas originalmente por la cámara (Bakhshi y cols., 2015).

Hoy en día, diversas empresas se han apalancado del fácil acceso a la cámara web por parte de los usuarios de teléfonos inteligentes destacando entre ellas empresas como Skype Technologies S.A.R.L (Skype), Oath Inc. (Flickr), Facebook Inc. (Facebook e Instagram) y Snap Inc. (Snapchat) han permitido la edición de imágenes aplicando filtros de manera sencilla para sus usuarios. Por ejemplo, la aplicación para móviles Snapchat se ha hecho muy popular por no solo aplicar filtros que permiten regular los colores de la foto o video en tiempo real, sino que permite detectar objetos capturados por la cámara del dispositivo y seleccionar entre una diversa variedad de lentes [1] animaciones interactivas sobre la toma (Snap Inc, 2017). Este tipo de filtros se han popularizado especialmente entre la gente que conforma la generación del milenio. Por ejemplo, para Snapchat, los jóvenes de entre 18 y 24 años representan un 36 % de sus usuarios activos por día en Estados Unidos (Snap Inc, 2017). De hecho, muchos jóvenes adultos sostienen que este tipo de filtros resuelve problemas de comunicación dado en ciertas redes sociales debido a que la sobreposición de imágenes y texto puede clarificar el mensaje que ellos desean expresar en sus fotografías (Vaterlaus, Barnett, Roche, y Young, 2016). El deseo por agregar imágenes sobre las fotografías, no se limita a occidente. De hecho, esta práctica ha sido común en Japón desde la década de los noventas. En el año 1995, Altus, una empresa japonesa, desarrolló un fotomatón (un quiosco para tomarse fotos, generalmente insertando una moneda) que permitía agregar pegatinas sobre las fotografías (Edwards y Hart, 2004). Atlus patentó una máquina con el nombre de Purikura, término que se volvería más adelante en un término del uso cotidiano de las personas en Japón principalmente los adolescentes (Edwards y Hart, 2004). Recientemente, estas máquinas también se han renovado incluyendo filtros digitales para el agregado de imágenes sobre las fotografías (*The Original Snapchat | Elise Tries | NPR - YouTube*, 2017).

En las computadoras de escritorio, podemos dividir la situación actual de las aplicaciones de webcam por el Sistema Operativo en los cuales están soportadas. Por un lado, en macOS, la aplicación para capturar fotos y videos con la cámara web por defecto es Photo Booth, la cual permite aplicar filtros en tiempo real, así como detectar a una persona para reemplazar el fondo por un video preestablecido o foto colocada manualmente por el usuario (Apple Inc., s.f.). Fun Booth es una aplicación para macOS. desarrollada por Spoonjuice LLC, la cual permite detectar un rostro humano frente a la

[1] Snapchat, la aplicación móvil, llama "lentes" a la funcionalidad que permite aplicar diversos efectos sobre las imágenes capturadas por la cámara del móvil

cámara para colocar imágenes sobre este (Spoonjuice, LLC., 2013). Por otro lado, Windows 10 incluye una aplicación para la toma de fotos y grabación de video, pero esta no dispone de filtros tan sofisticados como los de Photo Booth. Sin embargo, existen aplicaciones de terceros con varias funcionalidades. CyberLink YouCam, de CyberLink Corp. permite no solo detectar el rostro del individuo capturado y sobreponer filtros sobre esto, sino que detecta las emociones de las personas (CyberLink, s.f.). ManyCam que entre diversas funcionalidades que ofrece, permite además aplicar filtros para colocar máscaras y animaciones que rastrean el rostro de la persona ubicada frente a la webcam (ManyCam, s.f.). Para sistemas operativos basados en GNU/Linux, existen programas que permiten ajustar los colores de las capturas con la cámara web, pero la aplicación, al parecer, más popular y a la vez con filtros más sofisticados es Cheese, proyecto de la GNOME Foundation (The GNOME Project, s.f.-a).

Cheese, a diferencia del resto de programas mencionados con licencia privativa, es un software libre (Siegel, 2007). El software libre a diferencia del software privativo, según la Free Software Foundation, es el software que respeta la libertad de sus usuarios y comunidad permitiéndoles tener el derecho de ejecutar, copiar, distribuir, estudiar, modificar y mejorar el programa (Free Software Foundation, s.f.-d). Cheese está licenciado bajo la licencia GNU General Public License versión 2 (GPLv2) (Siegel, 2007). El hecho de que este tenga una licencia libre es fundamental para el desarrollo de este proyecto de fin de carrera, pues permite a otros estudiar su código fuente, mejorarlo y modificarlo. Este software, además, ha sido financiado por Alphabet Inc. en varias ocasiones. De hecho, este fue inicialmente desarrollado en el 2007 por Daniel G. Siegel como parte de un proyecto del Google Summer of Code (GSoC), programa ideado por Larry Page y Sergey Brin, fundadores de Google, para difundir el software libre y de código abierto remunerando a estudiantes por desarrollar en este tipo de proyectos (Byfield, 2005).

La necesidad por agregar soporte para sobreposición de imágenes sobre el rostro de las capturas de Cheese fue expresado en el 2010 (Yann, 2011) y, además ya hubo intentos de agregar esta funcionalidad. En el GNOME Outreach Program for Women (programa para mujeres similar al GSoC pero financiado por la GNOME Foundation, ahora llamado Outreachy) del 2010, la participante Laura Elisa Lucas Alday, trabajó en desarrollar un filtro llamado gstfaceoverlay para GStreamer (un framework especializado en el procesamiento multimedia). Este filtro permitía la adición de imágenes en formato svg sobre un rostro detectado en una imagen (Alday, 2011)("GNOME Project Announces Outreach Program for Women Interns", 2010). Posteriormente fue aceptado por los desarrolladores GStreamer, y agregado como parte de GNOME Video Effects, librería que contiene una lista de filtros de GStreamer o combinación de estos en archivos de texto que son leídos por Cheese para aplicar el filtro especificado en tal archivo de texto. Sin embargo, el filtro mencionado (gstfaceoverlay) fue posteriormente retirado de GNOME Video Effects puesto que algunos archivos svg hacían lento (el "framerate" caía) a Cheese (Lucas Alday, 2016).

En setiembre 2012, GStreamer 1.0 fue lanzado, y desde esa fecha, gstfaceoverlay no fue portado a la nueva versión de GStreamer. Esto llevó a que el filtro fuera borrado de la librería el 21 de diciembre del 2016. Ante esto, en el 2016, se propuso un parche en el sistema informático de seguimiento de errores de GNOME para portar el filtro a las series GStreamer 1.x, el cual sería aceptado en GStreamer en el año 2017 (Orccón Chipana, 2016a). Una vez portado, se probó su funcionamiento en Cheese (Orccón Chipana, 2016c) y se observó que además de solo soportar la sobreposición de imágenes con formato SVG, este filtro no puede detectar el rostro de más de una persona. Ello condujo a proponer otro parche que solucione este problema (Orccón Chipana, 2016b). Sin embargo, esta solución si bien al parecer es correcta en un contexto aislado, arrastra un problema en su base. El problema fundamental en este filtro es que no ha debido implementarse sobre la base del filtro gstfacedetect[2] para soportar múltiples rostros debido a lo que se detallará a continuación. El filtro *gstfacedetect* aplica detección facial para cada cuadro (o *"frame"* en inglés), pero sin respetar un orden alguno, es decir que una persona etiquetada en el primer cuadro con la imagen A podría ser etiquetada en el segundo cuadro con la imagen B, y en el tercero otra vez con A o tal vez con C. Lo esperado sería que si se sobrepuso la imagen A sobre el rostro de una persona, esta imagen A se mantenga sobrepuesta en los siguientes cuadros. Es probablemente, por esta razón que *gstfaceoverlay* solo soporta un rostro.

Para conseguir que Cheese soporte filtros de sobreposición de imágenes sobre múltiples rostros de personas de un video en vivo (como de la cámara web) no es suficiente escribir un nuevo filtro para GStreamer. Tampoco es suficiente, luego de escribir el filtro, agregar la configuración del "pipeline" de GStreamer a GNOME Video Effects de tal modo que Cheese pueda leerlo. Cheese debería implementar una interfaz gráfica en la cual el usuario pueda seleccionar e importar imágenes que desea usar, de este modo se evita el uso de efectos estáticos predeterminados. Es preciso resaltar que realizar tales cambios a Cheese no es fácil para nuevos colaboradores del proyecto. Exceptuando actualizaciones de traducción y actualizaciones mínimas de diseño, Cheese no ha recibido actualizaciones importantes [3], y una de las razones puede ser que ha sido programado en Vala, un lenguaje de programación orientado a objetos con un compilador que traduce el código a C para compilarlo con gcc (The GNOME Project, s.f.-h), y existe pocas personas que sepan programar en este lenguaje. Además Cheese usa Clutter (*cheese - Take photos and videos with your webcam, with fun graphical effects*, 2018), librería de GNOME basada en OpenGL, para mostrar la salida de la captura y los efectos aplicados en la pantalla. Ambos no disponen de mucha documentación, y por lo general, uno debe guiarse en base a otros programas que han sido escritos usando tanto Clutter como Vala.

[2]gstfacedetect es un filtro para GStreamer especializado en la detección de rostros.

[3]Al 21/06/2018, la mayoría de actualizaciones en el año 2018 han sido de actualizaciones. Más información puede verse en el repositorio oficial en la siguiente dirección: https://gitlab.gnome.org/GNOME/cheese/commits/master

Por otro lado, entre los filtros a usar para el rastreamiento de imágenes, algunos como 4dface no tienen licencias compatibles con GPLv2 [4], para lo cual no solo es necesario escribir software libre, sino que el código debe ser compatible con esta licencia.

En conclusión, se puede observar que los usuarios de software para la cámara web tienen varias razones para preferir usar filtros. El problema es más acentuado para usuarios de software libre. Los programas de escritorio con licencias libres o de código abierto para capturar imagen y video están quedándose atrasados tecnológicamente. Sin embargo, el problema no solo está en este tipo de software, sino que pareciera que las aplicaciones para macOS también están en la misma situación. Recientemente, la aparición de Snapchat ha creado una tendencia entre millennials de 18 a 24 años a tener una preferencia por filtros que sobreponen imágenes que siguen sus rostros capturados por la cámara web a otros medios de comunicación convencionales. Siendo Cheese un software libre, lo cual representa una ventaja social sobre el software privativo, es posible estudiarlo y mejorarlo (sin mencionar otras libertades) puede ser considerado como la mejor opción para implementar la característica de sobreposición de imágenes sobre cuadros capturados por la webcam, ya que tiene mejores características que otros proyectos de software libre similares para escritorio. Agregar la funcionalidad de aplicar filtros que sobrepongan imágenes sobre los rostros de las personas capturados por una webcam en tiempo real no solo es beneficioso para los jóvenes de la generación del milenio, sino también para la GNOME Foundation y podría aumentar la competitividad entre aplicaciones de escritorio de captura de video y entre entornos de escritorio para sistemas basados en UNIX (incluyendo macOS).

1.2. Objetivos

1.2.1. Objetivo general

Implementar e integrar en Cheese un plugin para GStreamer que mediante el uso de algoritmos para detección y seguimiento de múltiples rostros (o una adaptación de estos) permita sobreponer imágenes sobre los rostros detectados en videos (particularmente, por la cámara web).

[4] 4dface es ua librería para el rastreamiento de rostros licenciada según la *Apache License*. Según la Free Software Foundation, esta licencia no es compatible cona la licencia GPL versión 2 (Free Software Foundation, s.f.-c)

1.2.2. Objetivos específicos

1.2.2.1. OE1: Objetivo Específico 1

Desarrollar un algoritmo e implementarlo como un filtro de GStreamer para la detección y rastreamiento de múltiples rostros con soporte para oclusión así como la detección de puntos faciales del rostro.

1.2.2.2. OE2: Objetivo Específico 2

Escribir el código fuente de un filtro para GStreamer que basado en el filtro implementado en *OE1* permita la sobreposición de imágenes sobre puntos predeterminados en los rostros detectados.

1.2.2.3. OE3: Objetivo Específico 3

Diseñar una interfaz gráfica para Cheese que permita al usuario de Cheese aplicar el filtro implementado en *OE2*.

1.2.3. Resultados esperados

1.2.3.1. REOE1: Resultados esperados para el Objetivo Específico 1

Un filtro que implemente un algoritmo de rastreo de múltiples rostros, siendo este algoritmo propio o una implementación de alguno existente. Este filtro definirá una estructura o interfaz donde se guardarán las coordenadas de los rostros y sus puntos puntos faciales. Esta interfaz servirá como medio de comunicación entre el filtro y otros filtros o entre el filtro y la aplicación final. Además este filtro debe ser personalizable a través de parámetros, lo cual contribuirá a regular la latencia.

1.2.3.2. REOE2: Resultados esperados para el Objetivo Específico 2

Un filtro que tomando como referencia las coordenadas de diversos puntos de cada rostro detectado ofrezca un conjunto parámetros de entrada para colocar imágenes en lugares predefinidos del rostro como filtrum (parte entre la boca y nariz), boca, ojos, nariz, orejas y cara.

1.2.3.3. REOE3: Resultados esperados para el Objetivo Específico 3

Una interfaz gráfica con un conjunto de imágenes clasificadas por categoría. De estas imágenes el usuario puede seleccionar una o más de estas imágenes las cuales serán sobrepuestas en los rostros detectados aplicando el filtro del *OE2*.

1.2.4. Mapeo de objetivos, resultados y verificación

A continuación se presenta la siguiente tabla para facilitar la identificación de objetivos específicos, resultados y medios de verificación.

Objetivo (*OE1*): Desarrollar un algoritmo que implemente un filtro para *GStreamer* para la detección y rastreamiento de múltiples rostros con soporte para oclusión y detección de puntos faciales.		
Resultado	**Meta física**	**Medio de verificación**
Filtro para *GStreamer* que permita el rastreamiento de rostros y detección de puntos faciales del rostro a partir de un modelo entrenado permitiendo regular el retardo ajustando parámetros.	Software	*Precision*, *recall*, *accuracy* y *F1 score* del detector y rastreador de rostros, y comparación del *latency* creado entre el filtro existente y el desarrollado.
Objetivo (*OE2*): Escribir el código fuente de un filtro para GStreamer que basado en el filtro implementado en *OE1* permita la sobreposición de imágenes sobre puntos predeterminados en los rostros detectados.		
Resultado	**Meta física**	**Medio de verificación**
Filtro para *GStreamer* para sobreponer imágenes sobre las coordenadas detectadas.	Software	*Latency* y pruebas unitarias de partes importantes del código.
Objetivo (*OE3*): Diseñar una interfaz gráfica para Cheese que permita al usuario de Cheese aplicar el filtro implementado en *OE2*.		
Resultado	**Meta física**	**Medio de verificación**
Una copia modificada (*fork*) de Cheese que permita sobreponer imágenes sobre los rostros usando los filtros implementados como parte de los resultados de los objetivos específicos 1 y 2.	Software	

1.2.5. Herramientas, métodos y procedimientos

1.2.5.0.1. Metodologías para detección de rostros

1.2.5.0.1.1. Metodología 1: Detector de objetos de Viola-Jones

Este es el método que el filtro existente de *GStreamer* llamado *gstfacedetect* que se usa actualmente. Este método basado en aprendizaje de máquina para detectar objetos y procesar imágenes de manera muy rápida. A continuación se resume tres principales contribuciones de este método:

1. **Imagen integral y vector de características tipo Haar**: la imagen integral (figura 1.1a y 1.1b) es manera de representar una imagen la cual se usa para generar un vector de características tipo Haar (Viola y Jones, 2001). Por ejemplo, si se tiene el rostro de una persona (figura 1.2a), y se tiene dos rectángulos (figura 1.2b), el vector característica se puede calcular substrayendo la imagen integral del rectángulo negro con la imagen integral del rectángulo blanco (Wang, 2014).

1	2	2	4	1
3	4	1	5	2
2	3	3	2	4
4	1	5	4	6
6	3	2	1	3

(a) Imagen de entrada.

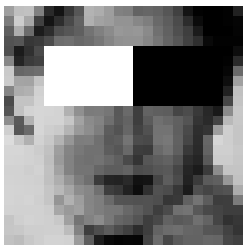
0	0	0	0	0	0
0	1	3	5	9	10
0	4	10	13	22	25
0	6	15	21	32	39
0	10	20	31	46	59
0	16	29	42	58	74

(b) Imagen integral.

Figura 1.1: Ejemplo de imagen integral. La imagen integral siempre tendrá su primera columna y primera fila llena de ceros. Luego las demás celdas se calcula como la suma de los acumulados de las celdas anteriores.

Imágenes tomadas y adaptadas de (Mathworks, s.f.)

(a) Imagen de un simple rostro en escalas grises.



(b) Rectángulos sobre los cuáles se calculará la imagen integral.

Figura 1.2: Ilustración de técnica usada para calcular los vectores de características tipo Haar. Una vez calculados la imagen integral en el rectángulo negro y blanco, estos se restan. El proceso se repite para todos los subrectángulos posibles y algunos métodos incluyen la rotación de estos.

Imágenes tomadas de (Wang, 2014)

2. **Entrenamiento con Adaboost**: Se usa una variante de Adaboost para descartar una gran cantidad de vectores característica, ya que tan solo usar un rectángulo de 24 x 24 píxeles crea 162336 vectores característica (Wang, 2014). Lo que se busca es seleccionar los vectores característica más representativos (Viola y Jones, 2001). De esta manera se puede seleccionar una característica de 180000 posibles características (Viola y Jones, 2001).

3. **Cascadas**: Se usa un árbol de decisiones llamado "cascada" (Viola y Jones, 2001). Se divide el clasificador en varios clasificadores (Viola y Jones, 2001). Un resultado positivo en un clasificador dispara otro clasificador, y así sucesivamente (Viola y Jones, 2001). Si un clasificador arroja un resultado negativo, la región rectangular ("subventana") es descartada (Viola y Jones, 2001). Cada clasificador consecutivo se conoce como *stage*, y la ventana que pasa todos los *stages* es una región de rostro (Viola y Jones, 2001).

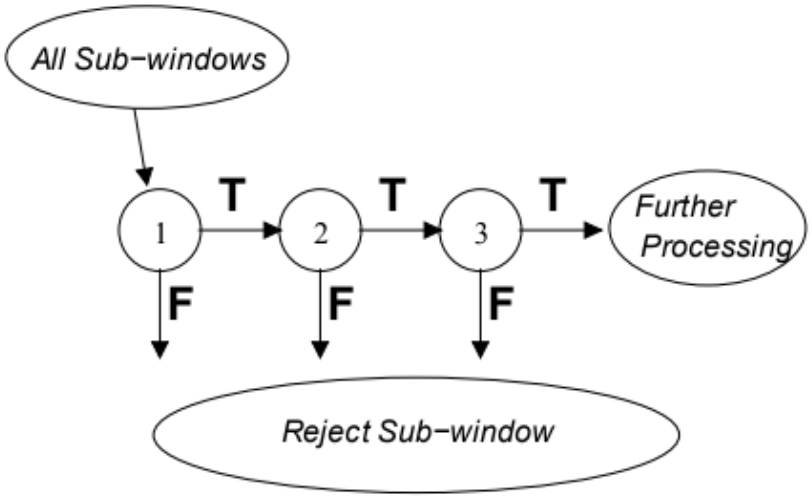


Figura 1.3: Diagrama de cascadas con 3 *stages*. Cada subventana, un trozo de imagen que se desliza por la imagen a analizar, es pasada por este clasificador. Las ventanas que pasan satisfactoriamente el clasificador se consideran rostros.

Fuente: (Viola y Jones, 2001)

1.2.5.0.1.2. Metodología 2: Histograma de gradientes para detección humana

Este método se basa en la creación de vectores característica a partir de la concatenación de desriptores de histogramas de gradientes. Según la publicación original, el primer paso es aplicar una normalización gamma (o corrección gamma). Esta suele ser usada para ajustar el brillo de las imágenes. Sin embargo, los resultados de la publicación muestran que la ganancia es mínima (Dalal y Triggs, 2005).

Una vez preprocesada la imagen con el filtro gamma, se toma una porción de la imagen (una ventana) la cual será filtrada con el kernel $[-1 \quad 0 \quad 1]$ y el kernel $[-1 \quad 0 \quad 1]^T$ (Dalal y Triggs, 2005). El resultado de aplicar este filtro son dos matrices en la que cada una representa una aproximación a las derivadas parciales en x y y. Por tanto, dadas las derivadas parciales en ambas coordenadas se puede calcular su magnitud y dirección. Estas dos matrices son, posteriormente usadas para formar el histograma de gradientes el cual tiene un rango de $[0°, 180°]$ y está formado por 9 *bins* (Dalal y Triggs, 2005).

El histograma de gradientes generado es un vector característica de una celda de 8×8 píxeles (Dalal y Triggs, 2005). Ya que se usan bloques de 16×16 píxeles, entonces se tiene 4 celdas de 8×8 píxeles (Dalal y Triggs, 2005). Por tanto, por cada celda se tiene un vector característica de 9 componentes y si se concatena los 4 vectores se tendría un vector en 36-D. Debido a que la iluminación o contraste puede influir en los valores de

los vectores, este vector concatenado de 36 componentes es normalizado (Dalal y Triggs, 2005). De la misma manera, deslizando el bloque de 16×16 por toda la imagen se va generando vectores característica los cuales combinados (concatenados) dan un vector característica final (Dalal y Triggs, 2005).
Finalmente, para un conjunto de imágenes positivas (imágenes que contienen el objeto a detectar, en este caso rostros) y para un conjuto de imágenes negativas (que no contienen el objeto a detectar) se calcula los vectores característica de ambos. Estos son pasados a un clasificador *SVM* (Dalal y Triggs, 2005).

1.2.5.0.2. Metodologías y algoritmos para el seguimiento de rostros

Según Zaheer Shaik y Vijayan Asari, los métodos de seguimiento de rostros pueden ser clasificados en cuatro categorías según la forma, las características, el modelo o color de piel de la cara (Shaik y Asari, 2007). Estos se detallan a continuación:

1. Seguimiento basado en la forma: en este método se debe usar un modelo que represente la locación de un rostro. Se toma la elipse como el modelo más simple que mejor se ajusta a la forma de un rostro (Eleftheriadis y Jacquin, 1995). Este método de seguimiento no es influenciado por el color de fondo ni por los cambios en la iluminación (Shaik y Asari, 2007). Sin embargo, este falla si la cara rastreada es ocluida ya sea por otro objeto u otra persona (Shaik y Asari, 2007).

2. Seguimiento basado en características: en este método, las características de los rostros son extraídas usando el filtro de Gabor el cual es usado como característica para el seguimiento de rostros (Shaik y Asari, 2007). Sin embargo, este método no es sensible a los cambios en iluminación y la pose de la cara (Shaik y Asari, 2007). Hacer que este método soporte esta característica tiene un alto costo computacional por lo que no es adecuado para streaming.

3. Seguimiento basado en el modelo: este usa un modelo en 3D del rostro, el cual es proyectado en los cuadros detectados por la cámara (Shaik y Asari, 2007). Sin embargo, estos pueden tener dificultad cuando hay cambios de iluminación o si la cara es ocluida (Smolyanskiy, Huitema, Liang, y Anderson, 2014). Si bien esta metodología suele ser precisa tiene el problema de tener un alto costo computacional por lo que no es adecuado para streaming.

4. Seguimiendo en base a los colores: este método usa el modelo de color de piel (Shaik y Asari, 2007). Sin embargo, aunque falla cuando el rostro es ocluido, es adecuado para streaming.

A continuación se describirán algunos métodos y algoritmos usados para el seguimiento de rostros , de los cuales el proyecto puede potencialmente tomar referencia en algunas partes o en su totalidad para implementar el filtro de rastreo de rostros.

1.2.5.0.2.1. Metodología 1: Rastreamiento del landmark en un dispositivo móvil

Es un método para rastrear puntos faciales de la cara. Se toma como objetivo de rastreo cinco puntos faciales: la parte externa de los ojos izquierdo y derecho, punta de la nariz y esquinas del labio (Wettum, 2017). Véase la figura la 1.4.

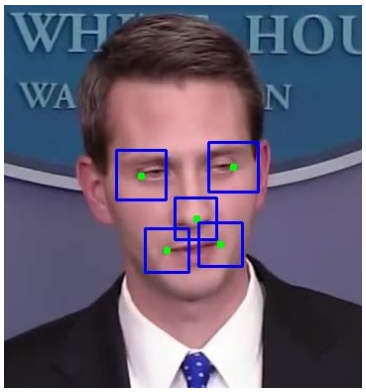


Figura 1.4: Cada 50 *frames* se detecta cinco puntos faciales del rostro y se inicializa un rastreador por cada punto facial con un *ROI* (región de interés) cuyo tamaño es proporcional a la distancia interocular (distancia entre entre los ojos).

Fuente: (Wettum, 2017)

La técnica consiste en detectar dichos puntos en el primer cuadro (*frame*), luego desde el cuadro número 2 hasta el cuadro 50 se usa algún rastreador tales como *DFLD*, *KCF*, *DSST*, *LK* o *Struck* (Wettum, 2017). En el cuadro número 51, se vueleve a detectar los puntos faciales para corregir, y así sucesivamente. Es decir, cada 50 cuadros se detecta los puntos faciales y en las brechas cada uno de los cinco rastreadores se encarga de rastrear sus respectivos puntos asignados (Wettum, 2017). Véase la figura 1.5.

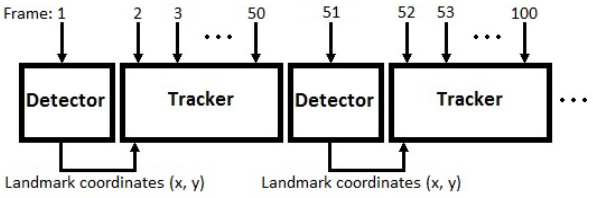


Figura 1.5: Diagrama que muestra las alternaciones de detección y rastreo.

Fuente: (Wettum, 2017)

1.2.5.0.2.2. Metodología 2: Rastreamiento de múltiples objetos usando el método húngaro y el filtro de Kalman

En un artículo académico, se desarrolló un sistema para el seguimiento y clasificación de objetos (peces) en movimiento (Szűcs, Papp, y Lovas, 2015). El primer paso consiste en detectar los objetos, guardando sus regiones de interés con sus respectivos *timestamp*, las cuales serían entrada para el proceso de clasificación (Szűcs y cols., 2015). Para la detección se usan diversos métodos, los cuales no son de interés mencionar ya que su objetivo estaba enfocado en peces (Szűcs y cols., 2015). Sin embargo, el método de rastreo que usa para rastrear a los peces detectados resulta útil pues es aplicable a objetos en general.

Luego de detectar los objetos, se usa el filtro de Kalman para rastrear los objetos en los tres pasos básicos del filtro de Kalman: inicialización, predicción y corrección (Szűcs y cols., 2015). En la etapa de inicialización, se inicializa el filtro de Kalman por cada objeto detectado (Szűcs y cols., 2015). En la etapa de predicción, se estimaba la posición del objeto en el siguiente cuadro. En la etapa de corrección, se usa la información de la medición de los objetos detectados para corregir los resultados de la predicción del filtro (Szűcs y cols., 2015). Sin embargo, esto no era suficiente por sí solo, y había que realizar un pareo entre los objetos detectados y para ello se usó el método húngaro (Szűcs y cols., 2015).

Sea v_i para $i \in 0...k$ un objeto rastreado en el cuadro actual, y sea w_j para $j \in 0...l$ el nuevo objeto detectado en el siguiente cuadro, el primer paso es crear una matriz $M_{k \times l}$ donde cada celda de la matriz $M[i, j]$ representa la distancia (euclideana) de v_i a w_j (Szűcs y cols., 2015). Las filas y columnas que superaban cierto límite (*threshold*) eran borradas para evitar un pareo (o asignación) erróneo (Szűcs y cols., 2015). Si v_i se removía significaba que el valor del coeficiente de confianza (CFV) devía bajarse (Szűcs y cols., 2015). Si se removía w_j, significaba que este objeto era probablemente un nuevo objeto y se creaba un nuevo filtro de Kalman para este (Szűcs y cols., 2015).

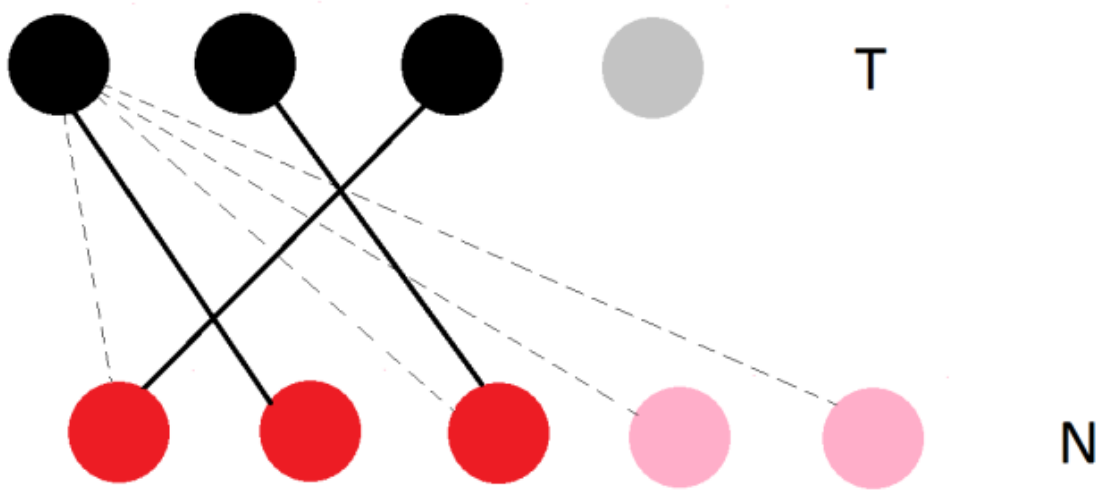


Figura 1.6: En la figura, los círculos negros y rojos representan los objetos rastreados en el cuadro actual y los objetos detectados en el siguiente cuadro, respectivamente. Luego de aplicar el método Húngaro, ciertos círculos negros son asignados a los círculos rojos. Los círculos con baja opacidad representan objetos que no fueron asignados, porque superaron el *threshold*

Fuente: (Szűcs y cols., 2015)

1.2.5.0.2.3. Metodología 3: Rastreador de características de Kanade-Lucas-Tomasi

Es un método basado en dos artículos académicos. El primero publicado por Bruce D. Lucas y Tadeo Kanade titulado *An Iterative Image Registration Technique with an Application to Stereo Vision* (Lucas, Kanade, y cols., 1981) y el segundo publicado por Carlo Tomasi y Takeo Kanade (Tomasi y Kanade, 1991) con el título de *Detection and Tracking of Point Features*. A continuación se describe brevemente cada uno de los dos artículos.

En el primer artículo se explica que por lo general dos imágenes estarán prácticamente en el mismo registro. Por ejemplo, en un video, dos cuadros consecutivos serán muy parecidos. Dado este supuesto, se explica que dada las funciones $F(x)$ y $G(x)$ (las cuales x representan el valor de un píxel en la posición x), se desea encontrar un vector de disparidad h tal que se minimice la diferencia entre $F(x+h)$ y $G(x)$. Siendo las métricas típicas para hallar la disparidad las siguientes:

$$Norma \quad L_1 = \sum_{x \in R} |F(x+h) - G(X)|$$

$$Norma \quad L_2 = (\sum_{x \in R} [F(x+h) - G(X)]^2)^{\frac{1}{2}}$$

$$Negación \quad de \quad la \quad correlación \quad normalizada = \frac{(\sum_{x \in R} [F(x+h)G(X)]}{(\sum_{x \in R} F(x+h)^2)^{\frac{1}{2}} (\sum_{x \in R} G(x)^2)^{\frac{1}{2}}}$$

Se dice que es posible hallar tal vector de disparidad h buscando todos los valores posibles de tal vector o también usando una heurística conocida como "hill climbing", pero que esos métodos son muy costosos. El artículo propone un algoritmo para encontrar tal valor de h de una manera eficiente, en la cual se inicializa la variable con un valor arbitrario, la cual se va actualizando iterativamente usando el método de Newton. El segundo artículo busca responder a dos preguntas: cómo seleccionar las características y cómo rastrearlas entre cada cuadro. En cuanto a la segunda pregunta, como propuesta de solución se dice que es difícil rastrear un píxel de un cuadro a otro, por lo que se propone rastrear ventanas (regiones de imagen) en vez de cada píxel. Se propone un modelo para la transformación de la imagen de un frame a otro agregando cierto ruido gausiano (para hacer énfasis en el centro de la ventana), buscando minimizar el error resultante de manera similar a como propone el primer artículo, de tal modo que el desplazamiento (y por tanto, la nueva posición del punto a rastrear) puede ser hallado. De otro lado, con respecto a la primera pregunta se dice que en trabajos pasados las características usadas solían ser ideas arbitrarias de lo que sería una "buena" ventana, por ejemplo escoger las esquinas. El artículo propone un método basado en valores propios (o eigenvectores) para seleccionar estas características.

1.2.5.0.2.4. Metodología 4: Flujo de medianas (*Median flow*)

A continuación se describe el rastreador llamado *Median flow* que se menciona en el artículo académico titulado en inglés como *Forward-Backward Error: Automatic Detection of Tracking Failures* (Kalal, Mikolajczyk, y Matas, 2010). Este un rastreador basado en el rastreador de Lucas-Kanade. Recibe como entrada una imagen y un rectángulo delimitador (*bounding box*). Sobre este rectángulo delimitador se ubican puntos en forma de grilla, los cuales se rastrearan usando el rastreador KL (Kanade-Lucas). De las predicciones de los puntos en el nuevo cuadro, el 50 % de las peores predicciones será descartado en base a un error asignado, mientras que los puntos restantes se usan para calcular el rectángulo delimitador. El 50 % restante de los puntos son usados para estimar el desplazamiento del rectángulo delimitador calculando la mediana sobre cada dimensión especial de los puntos restantes. Este rastreador también soporta el cambio de escala (lo cuál es útil, por ejemplo, cuando el objetivo a rastrear, por ejemplo, un rostro, se aleja o se acerca a la cámara). El factor de escala se calcula usando la mediana de la relación entre la distancia inicial y la nueva distancia entre cada par de puntos.

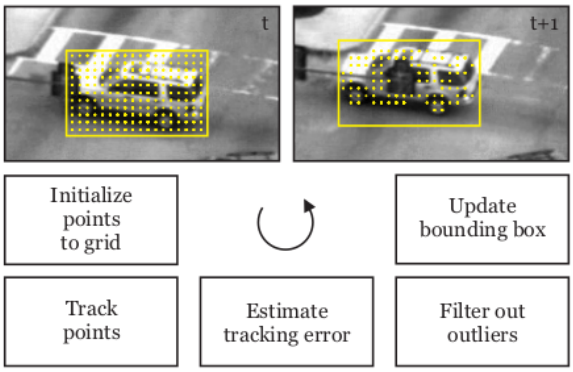


Figura 1.7: Como primer paso se inicializa un rectángulo delimitador con ciertos puntos a rastrear en forma de grilla, luego se rastrea con Lucas-Kanade, se estima el error y se filtran los valores atípicos y finalmente se actualiza el rectángulo delimitador en base a los puntos restantes y la mediana entre las distancias entre los puntos.

Figura tomada de *Forward-Backward Error: Automatic Detection of Tracking Failures* (Kalal y cols., 2010).

1.2.5.0.3. Herramientas empleadas

1.2.5.1. En relación al Objetivo Específico 1 (OE1)

1. OpenCV, ya que implementa algunos algoritmos y técnicas útiles para el seguimiento, detección de objetos (rostros) y procesamiento de imágenes (OpenCV team, s.f.-a):

   a) Conversión a espacios de color, usado para trabajar las imágenes RGB en distintos espacios de color entre estos *GRAY* y YC_bC_r.

   b) Proporciona una librería de rastreadores de rostros, entre ellos Median Flow y otros rastreadores tales como *MIL*, *Boosting*, *TLD*, *KCF* y además implementa el filtro de Kalman.

   c) Erosión y dilatación de imágenes.

2. *GStreamer*, ya que es el framework sobre el cual se implementará el filtro (Taymans, Baker, y Wingo, 2016).

3. *gstfacedetect*, filtro para *GStreamer* que usa OpenCV para la detección de rostros y envía señales al *bus* con las coordenadas localizadas.

4. *C++*, ya que OpenCV y GStreamer lo soportan.

5. *mesonbuild* es un sistema de compilación de código fuente más fácil de usar y un tiempo de compilación más bajo que el de *autotools* (Pakkanen, s.f.). De hecho, este sistema es el que se empieza a usar en muchos programas de *GNOME* últimamente en reemplazo de *autotools*.

6. gst-launch-1.0, un programa para construir *pipelines* de streaming de audio y video por línea de comandos, para probar manualmente la salida del streaming.

7. Dlib, un *toolkit* en *C++* que proporciona algoritmos de machine learning, y que entre una de sus características está la de detectar el *landmark* o puntos faciales de un rostro (King, 2014b).

1.2.5.2. En relación al Objetivo Específico 2 (OE2)

1. *gstcairooverlay*, un filtro para *GStreamer* que usa la librería *Cairo* la cual permite dibujar sobre las imágenes o cuadros recibidos por este filtro.

2. *vapigen*, para generar *bindings* que permitan la compatibilidad entre el código escrito en *C* y el código escrito en *Vala*, el cual será usado para el siguiente objetivo (Project, s.f.).

3. *GStreamer*, *C*, *mesonbuild* y *gst-launch-1.0* por las mismas razones mencionadas en el caso anterior.

1.2.5.3. En relación al Objetivo Específico 3 (OE3)

1. *Gtk+*, librería para diseñar interfaces gráficas (Team, s.f.) y que es usada por *Cheese* (The GNOME Project, s.f.-d) y se usará para diseñar la interfaz gráfica para seleccionar las imágenes a sobreponer sobre los rostros.

2. *Glade*, una herramienta gráfica escrita en GTK+ para construir interfaces gráficas en GTK+ (The Glade project, s.f.).

3. *Vala*, lenguaje en el cual se desarrolla *Cheese* (The GNOME Project, s.f.-h).

4. *GStreamer*, integrando los filtros implementados con *Cheese*.

5. *GLib Test Suite*, para ejecutar pruebas sobre el nuevo código agregado a *Cheese*.

1.3. Alcance, limitaciones y riesgos

1.3.0.1. Alcance

Este proyecto se concentrará en funcionar en las computadoras con CPUs (que implementen el conjunto de instrucciones SSE4.1, SSE4.2, AVX o AVX2) y con sistemas operativos GNU/Linux con entorno de escritorio *GNOME*. Es decir, la detección, seguimiento y sobreposición de imágenes en Cheese debería dar la sensación de fluidez a usuarios de estas computadoras. Además, este proyecto debe funcionar sobre *Wayland* ya que es el protocolo más usado recientemente por las distribuciones de GNU/Linux reemplazando a *X.Org* (Magazine, 2017).

1.3.0.2. Limitaciones

El desempeño del software desarrollado puede tener comportamientos distintos según la computadora en la cual se pruebe. Este puede ser afectado por diversos factores desde el retardo de la cámara web hasta la velocidad de procesamiento de la CPU.

1.3.0.3. Riesgos

Un proyecto de fin de carrera puede estar asociado a diversos riesgos. Los riesgos listados en la tabla de abajo no solo implican situaciones del mismo proyecto, sino también riesgos externos los cuales pueden afectar negativamente el desarrollo de este proyecto.

Riesgo identificado	**Impacto**	**Medidas para mitigar el riesgo**
La brecha de seguridad, principalmente a través de un ataque cibernético por *hackers* puede poner en riesgo cuentas donde se tiene publicados los avances del proyecto de fin de carrera. Estas amenazas puede incluir acceso no autorizado a cuentas como *github*, *Google Drive* y correo electrónico del autor del proyecto.	Medio	Mantener los avances del proyecto en más de una cuenta, y evitar limitarse a una sola cuenta.

La aparición de *blocker bugs* en actualizaciones de dependencias pueden impactar negativamente en el cumplimiento del cronograma previsto y además afectar la compatibilidad del software presentado con las últimas versiones en dependencias.	Bajo	Los *blocker bugs* suelen ser temporales, por tanto, si en el transcurso del proyecto apareciera alguno, se debería reportar el *bug* en caso de no estar reportado y simplemente esperar hasta que aparezca una actualización. Mientras tanto se deberá trabajar sobre una versión anterior de la dependencia.
Actualizaciones y cambios radicales en las librerías como *GStreamer*, *OpenCV* y *dlib* que cambien la forma de trabajo puede retrasar el avance del desarrollo del software debido a que implicará un esfuerzo extra el aprender la nueva forma de uso de estos programas.	Bajo	Estar al pendiente de páginas, blogs, foros y listas de correos de las respectivas librerías con el fin de estar al tanto de las últimas actualizaciones con el fin de evitar de que el software use funciones obsoletas.

1.4. Justificación y viabilidad

1.4.1. Justificación

Este proyecto se desarrolla por varias razones La primera, y la principal, es la de entregar a las personas, y en especial a la comunidad de software libre, un software le permita divertirse haciendo uso de la cámara web de su computadora de escritorio sin tener que verse en la necesidad de recurrir a software privativo o propietario, y por tanto respetando la libertad de su informática. Esta manera de "divertirse" se desarrolla a través de la introducción de una mejora, un filtro que permite sobreposicionar imágenes en la captura de la cámra web. En un contexto social, donde prima los valores y ética, bajo ninguna razón se debería desarrollar ni usar software privativo. Por un lado, cuando un desarrollador licencia un programa con licencia privativa este automáticamente toma control sobre sus usuarios. Sin embargo, el usuario, quien es el dueño de la computadora, es quien debería tener el control de esta. Cuando un desarrollador tiene el control este suele verse tentado a introducir de manera intencional funcionalidades en los programas que dañan a sus usuarios actuando en contra de sus intereses (*malware*). Existe evidencia para calificar a tanto el software de *Apple, Inc* como el de *Microsoft, Inc* como malware (Free Software Foundation, s.f.-b) según la *Free Software Foundation*. Un ejemplo de ello

es *Skype* en el cual la NSA podía tener accesso a los datos de los usuarios de Skype (tanto en video, voz, mensajes en texto y archivos compartidos entre sus usuarios) (Rushe, 2013). Por otro lado, el usuario de software privativo constantemente cae en dilemas éticos: ¿quién debería decidir si compartir el software: el usuario o el desarrollador? Si el mejor amigo de este usuario pidiera una copia a su amigo, este tendría dos opciones: quedar bien con su amigo copiando el software, pero actuando ilegalmente o no prestar el software y quedar mal con su amigo (Stallman, 2009). Escribiendo software libre se respeta la libertad del usuario. La segunda razón tiene como propósito potenciar la creación y mejora de otros software multimedia que se pueda apalancar de los filtros que quedarán como resultado de este proyecto. Por ejemplo, algunos usuarios de *Pitivi*, editor de video no lineal basado en GStreamer, han solicitado que se agregue el mecanismo para poder agregar parches sobre los rostros de personas en los videos. El filtro de seguimiento de rostros implementado para *Cheese* podría escanear los rostros de diversos individuos a lo largo de un *clip* de video en *Pitivi* y automáticamente generar *keyframes* los cuales indicarían la posición de múltiples parches sobre los rostros de las peronas y que podrían ser manipulados o borrados simplemente agregando símbolos que representen parches en el visor (*viewer*) de *Pitivi* los cuales pueden ser arrastrados o borrados.

1.4.2. Análisis de viabilidad

1.4.2.1. Viabilidad técnica

Con respecto a la implementación del filtro en *GStreamer*, si bien desarrollar plugins para GStreamer tiene una curva de aprendizaje relativamente larga, se tiene experiencia desarrollando tanto *src pads* como filtros para *GStreamer*, por lo que desarrollar este filtro no sería un problema. Con respecto a la implementación del algoritmo de rastreo de rostros, se cuenta con los conocimientos fundamentales para poder implementarlos satisfactoriamente, además se tiene la disposición de herramientas de código abierto como *OpenCV* y *dlib* (sobre los cuales se ha realizado experimentos previos), así como la disposición de un asesor quien domina el tema de visión artificial. En cuanto a *GTK+* y otras librerías de GNOME, no solo se ha desarrollado anteriormente aplicaciones usando estas tecnologías (*Pitivi*), sino que incluso se ha aportado en el desarrollo en proyectos como Libpeas y GObject, por lo que se tiene conocimiento de algunas librerías a profundidad. En relación a los lenguajes a usar *Vala* y *C++*, el primero tiene un parecido a *C#*, el cual ha sido enseñado en un curso de la universidad así como el segundo. Por otra parte, en relación a *Cheese*, además de tener a disposición el código fuente, se ha intentado con anterioridad implementar la funcionalidad del proyecto por lo que se conoce ciertas partes relevantes del programa, además se tiene contacto directo con el actual mantenedor del proyecto *Cheese*, David King, quien ha mostrado interés en el proyecto y ofrecido su ayuda incluso para revisar el código.

1.4.2.2. Viabilidad temporal

La implementación del proyecto dará inicio el 11/12/2017 y se contará con tres meses aproximadamente para tener la mayor parte del proyecto concluida. Los siguientes cuatro meses (aproximados) se dedicará a realizar algunos ajustes, y realizar la documentación requerida. A continuación, se muestra las tareas identificadas que involucra cumplir con cada resultado esperado y el tiempo por tarea estimado en horas.

Tarea	**Horas estimadas**
Completar el resultado esperado *REOE0*	
Preparar el *dataset*	24
Implementar alguna de las metodologías propuestas o una variación o combinación de estas *GStreamer* en C++	144
Agregar soporte para detección de los *landmarks* usando *dlib*	32
Medir *accuracy*, *precision* y *recall*	16
Definir propiedades del objeto y estructura (*GstStruct*) que se enviará al *bus*	2
Construir el *Boilerplate* del plugin para *GStreamer* incluyendo *OpenCV* como dependencia	6
Integrar el algoritmo implementado en un filtro de *GStreamer*	60
Medir *accuracy*, *precision* y *recall* en el filtro	16
Medir la latencia (*latency*) del filtro o *pipeline*	16
Completar el resultado esperado *REOE1*	
Definir propiedades del objeto	3
Implementar filtro de sobreposición de imágenes	24
Medir el retraso (*latency*) agregado por el filtro	8
Completar el resultado esperado *REOE2*	
Diseñar *mockups* de la interfaz gráfica	8
Implementar interfaz gráfica en *Cheese* para los filtros	24
Integrar filtros del proyecto con el *pipeline* de *Cheese*	80
Escribir *unit tests*	16

1.4.2.3. Viabilidad económica

Debido a que el proyecto se apalancará no solo de herramientas con licencias de software libre sino que gratuitas, no se puede identificar costos fijos relacionados en este aspecto. Se identifican costos fijos aunque menores provenientes de la energía eléctrica y costos para un total de 500 horas de trabajo aproximadas. También se debe precis-

ar que este proyecto agregaría costos relevantes de mantenimiento en caso de que las contribuciones sean oficialmente aceptadas por *The GNOME Project*.

Capítulo 2

Marco Conceptual

2.1. Objetivos del marco conceptual

A continuación se explicará algunos conceptos que se consideran básicos para poder comprender los siguientes capítulos (Shaik y Asari, 2007). En primer lugar, se explica qué es el software libre ya que a veces es un concepto que suele ser confundido. Luego, se introduce a GStreamer, librería en la cual Cheese se basa. Finalmente, se muestra una visión en conjunto del flujo de datos multimedia en Cheese, así como las librerías que se usarán para implementar los algoritmos de seguimiento de rostros.

2.2. Software libre

El software libre es el software que respeta la libertad de sus usuarios y comunidad. Según la Free Software Foundation, para que un programa informático sea calificado de software libre, su licencia debe otorgar las siguientes cuatro libertades:

- La libertad de ejecutar el programa como se desea, con cualquier propósito (libertad 0).
- La libertad de estudiar cómo funciona el programa, y cambiarlo para que haga lo que usted quiera (libertad 1). El acceso al código fuente es una condición necesaria para ello.
- La libertad de redistribuir copias para ayudar a su prójimo (libertad 2).

> - La libertad de distribuir copias de sus versiones modificadas a terceros (libertad 3). Esto le permite ofrecer a toda la comunidad la oportunidad de beneficiarse de las modificaciones. El acceso al código fuente es una condición necesaria para ello.
>
> (Free Software Foundation, s.f.-e)

Si estas cuatro libertades no se cumplieran, el software se consideraría no libre. Como se puede observar además, el término "precio" no se ha mencionado. Por lo que el software libre no es un tema de precios, sino como se mencionó inicialmente, de libertad, lo cual está relacionado directamente con un tema ético (Free Software Foundation, s.f.-e). Sin embargo, se aclara esto ya que muchos suelen confundirlo con "freeware".

2.3. Software no libre o software privativo

El software no libre también llamado software privativo es todo software que no cumple alguna de las cuatro libertades del software libre, y por tanto no es software libre (Free Software Foundation, s.f.-a).

2.4. GNOME

Es un entorno de escritorio para distribuciones de GNU/Linux y BSD. Está desarrollado por The GNOME Project, parte del proyecto GNU. GNOME es software libre y está formado por una amplia variedad de proyectos más pequeños los cuales están licenciados en su mayoría bajo las licencias GPL y LGPL. GNOME está financiado por algunas empresas grandes como Google Inc. y Red Hat Inc (GNOME Foundation, s.f.). Entre estos proyectos se encuentran librerías importantes como un conjunto de librerías para la creación de programas para el usuario final y programas para el usuario final como editores de texto como un navegador de archivos, un emulador de terminal, navegadores web, clientes IRC, programas gráficos para el control de versiones en Git, reproductores de audio y video, visores de imágenes entre una lista de más de cien proyectos (The GNOME Project, s.f.-g). GNOME es uno de lo escritorios libres más populares y está traducido en más de 40 idiomas (The GNOME Project, s.f.-e).

2.5. GStreamer

Es un proyecto de GNOME. GStreamer es una librería de código abierto y multiplataforma escrita en C con bindings en varios otros lenguages de programación y basada en GObject. GStreamer implementa un modelo de "pipelines" que enlaza una serie de elementos, por ejemplo, demultiplexores, decodificadores, multiplexores, filtros y codificadores (GStreamer Team, s.f.). Ello permite que cada elemento cumpla una tarea específica. GStreamer permite la creación de aplicaciones multimedia y streaming. Para ejemplificar, considerar el caso de la reproducción reproducción de un archivo con formato ogg, se puede pasar el elemento a un elemento llamado gstfilesrc, donde este lee el archivo por buffers y pasa un buffer a un demuxer que separa el audio del video. Los buffers de audio van a un decodificador de Vorbis (códec de audio), mientras que los buffers de video son pasados a un decodificador de Theora (códec de vídeo). Cada buffer de audio es pasado a un elemento que se encargará de pasar los datos recibidos, por ejemplo, a PulseAudio. Cada buffer de video, por su parte, podría ser enviado a Wayland. PulseAudio se encargaría de trasladar esos datos al driver de audio para emitir la salida por el parlante y Wayland se encargaría de mostrar el video en pantalla.

2.5.1. Elemento (GstElement)

Un elemento (GstElement) Es una abstracción sobre la cual se basan la mayoría de objetos de GStreamer para construir un pipeline, el cual es una cadena de objetos GstElement enlazados. Un GstElement puede tener una entrada y una o más salidas. En un demultiplexor gstoggdemux por ejemplo, la entrada serían buffers del archivo en formato ogg, y las salidas serían por un lado, el audio, y por el otro lado el vídeo. Existe tres tipos de elementos muy usados:

2.5.1.1. Sources

Son productores de datos, puede ser por ejemplo *gstvideotestsrc* que produce datos para videos de prueba; otro ejemplo, podría ser *gstfilesrc* que carga datos de un archivo. Los source elements solo tienen un source pad.

2.5.1.2. Sinks

Representan consumidores de datos. Por ejemplo, *glimagesink* recibe frames en buffers y los renderiza usando OpenGL mostrando el resultado en una ventana. *waylandsink*,

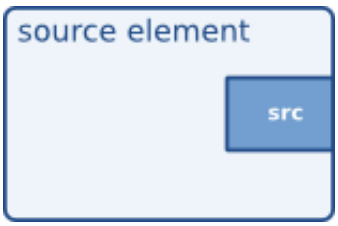


Figura 2.1: Representación de un *source*

Fuente: (Boulton y cols., 2017, p. 5)

xvimagesink, *cluttersink* cumplen roles similares pero usan internamente Wayland, X y Clutter respectivamente. Un "sink element" solo tiene un sink pad.

Figura 2.2: Representación de un *sink*

Fuente: (Boulton y cols., 2017, p. 5)

2.5.1.3. Filtros

Un filtro recibe ciertos datos, los procesa, y envía los datos modificados por una o más salidas. Un ejemplo de filtro es agingtv que agrega un efecto de televisión antigua. Otro ejemplo es *glfiltercube* que usa OpenGL para crear un cubo cuyas caras están texturizadas con los frames que este filtro recibe. Un filtro tiene un source pad y un sink pad.

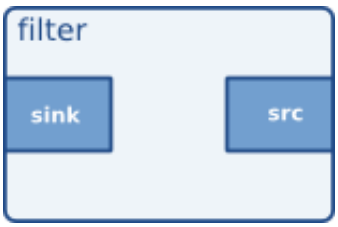


Figura 2.3: Representación de un filtro

Fuente: (Boulton y cols., 2017, p. 5)

2.5.1.4. Bin

Un *bin* es un tipo de elemento que contiene más elementos. La cantidad de *source pads* que posee y la cantidad de sink pads (ambos llamados *ghost pads* cuando se trata de un *bin*) que posee puede ser variable, y por lo general, depende de los elementos que están en sus extremos.

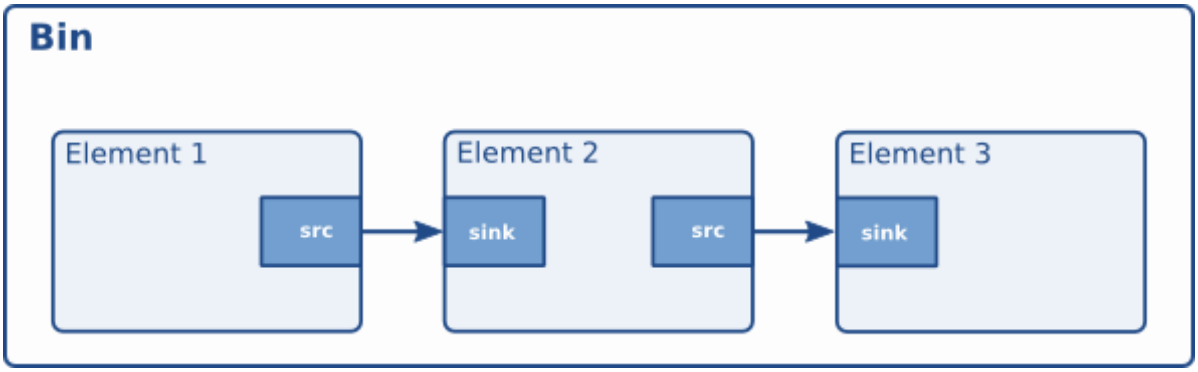


Figura 2.4: Representación de un *bin*

Fuente: (Taymans y cols., 2016)

2.5.1.5. Pipeline

Es, por lo general, el bin de máximo nivel en una aplicación. Un *pipeline* controla el reloj (*GstClock*) global del cual dispone GStreamer, además dispone de un bus el cual controla, por lo que la aplicación no tiene la necesidad de crear un bus. Un pipeline puede recibir consultas de la aplicación como por ejemplo para hacer *seeking* o para obtener duración del video reproducido con el pipeline.

2.5.2. *Pads* y *Capabilities*

Cada elemento de *GStreamer* podría ser visto como una caja negra. La manera de cómo se expone esta caja negra al mundo está definida por los *pads*. Existen dos tipos de *pads*: *sink pads* y *source pads* (Taymans y cols., 2016). Esquemáticamente, un *sink pad* se representa en el lado izquierdo de un elemento y un *source pad* se representado al lado derecho de este (ver figura 2.3) (Taymans y cols., 2016). Las *capabilities* de un *pad* representan qué tipo de datos puede fluir a través de este *pad* (Taymans y cols., 2016). Por ejemplo, las *capabilities* describen el tipo de media (*video/x-raw* o *audio/x-raw*, entre otros), el formato (*RGB*, *RGBA*, *YUV*, entre otros), canales, rango de resoluciones del cuadro de video, cantidad de fotogramas por segundo y otras propiedades.

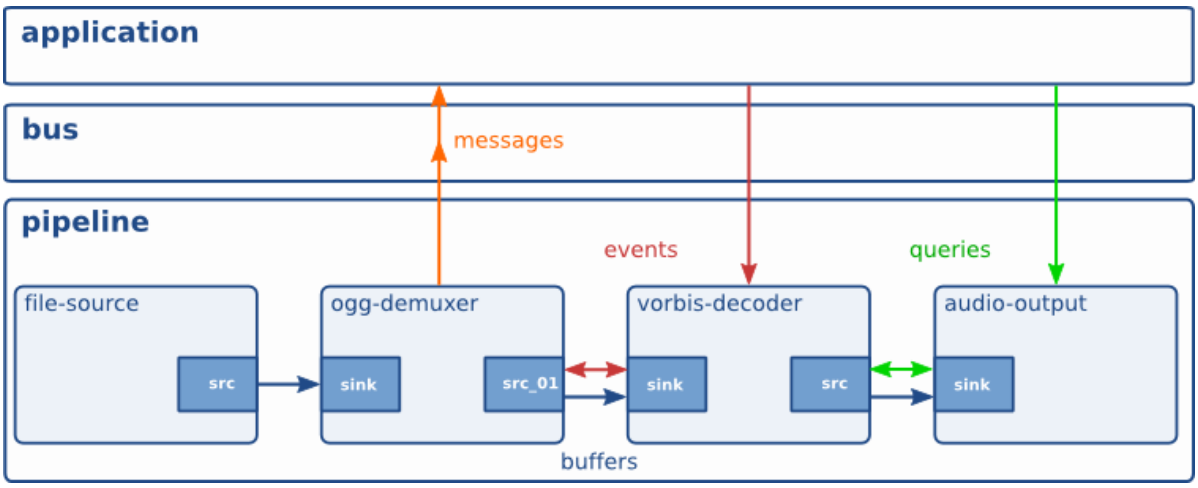


Figura 2.5: Representación de un *pipeline*

Fuente: (Taymans y cols., 2016)

2.5.3. Bus (GstBus)

GStreamer es una librería multihilo. Cada GStreamer el bus es el encargado de recibir mensajes del pipeline y enviarlos a la aplicación. La aplicación puede capturar estos mensajes usando una retrollamado (*callback*, similar a un *signal handler*). Por ejemplo, *facedetect* usa *OpenCV* para detectar los rostros, y el tamaño y coordenadas de los rostros detectados son enviadas al bus. De esta manera, una aplicación u otro elemento como *gstfaceoverlay* puede leer los mensajes en el bus y decidir qué hacer con la información recibida, en este caso sobreponer imágenes.

2.5.4. Plugin (GstPlugin)

GStreamer es extendible a través de plugins. Un plugin puede contener uno o un conjunto de elementos. Un plugin no es un elemento. En GStreamer, entre los sets de plugins más populares se encuentran:

- **gst-plugins-base** es un set de plugins con los elementos esenciales para la creación de elementos más complejos.
- **gst-plugins-good** es un set de plugins soportado activamente por la comunidad de GStreamer. Está licenciado bajo LGPL.
- **gst-plugins-ugly** también licenciado bajo la licencia LGPL (versión 2.1), es un set de plugins que puede carecer de revisiones, un mantenedor activo, documentación,

pruebas unitarias o cuyos desarrolladores pueden tener dudas respecto a ciertas patentes.

- **gst-plugins-bad** es un set de plugins que puede tener problemas de distribución.

2.6. Clutter

Es una librería que usa OpenGL para crear interfaces gráficas ocultando la complejidad de este mismo. Coloca a la disposición del desarrollador herramientas para agregar texto, animaciones, imágenes los cuales, por ejemplo, pueden ser arbitrariamente colocados o rotados (Bassi, s.f.). Clutter se integra tanto con Gtk como con Gstreamer.

2.7. Vala

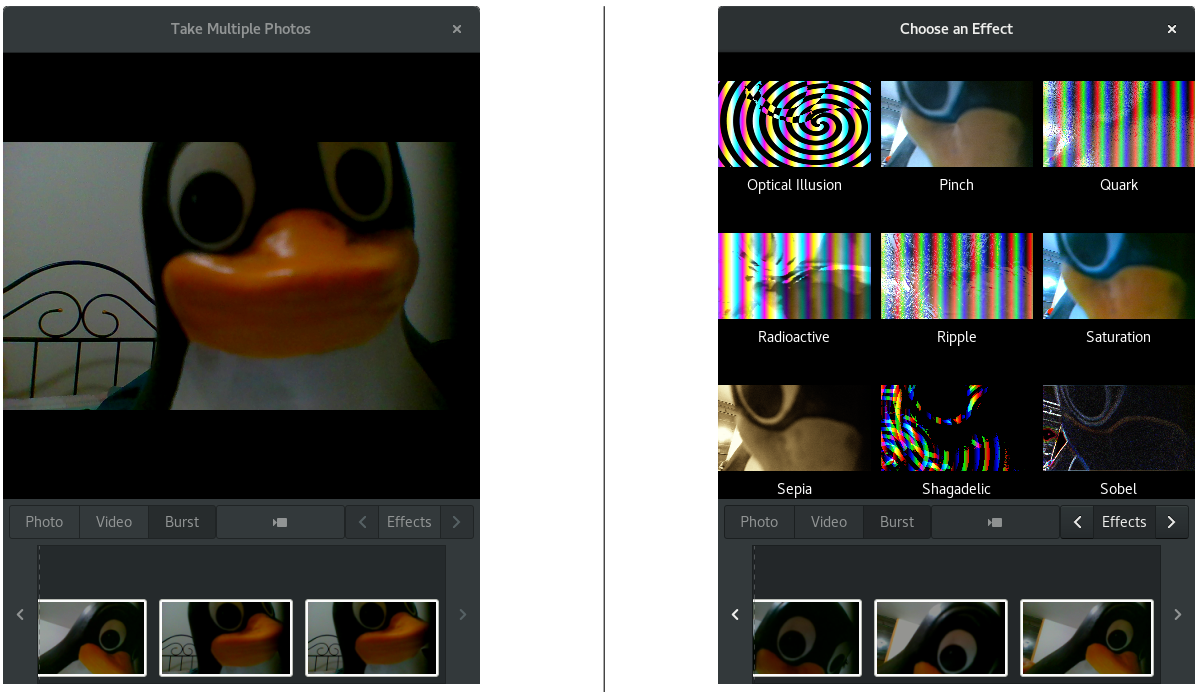
Es un lenguaje de programación orientado a objetos desarrollado para adaptarse mejor a las librerías de GNOME. El compilador de Vala es valac, el cual compila código en Vala, generando archivos del lenguaje C, los cuales son compilados con *gcc*. La orientación a objetos está dada debido a que internamente usa GObject, librería de GNOME que simula la programación orientada a objetos en C. El hecho de que compilar programas escritos en Vala genere código fuente en C, la necesidad de escribir bindings. Sintácticamente, es idéntico a C# (The GNOME Project, s.f.-h).

2.8. GNOME Video Effects

Es una colección de filtros de GStreamer (The GNOME Project, s.f.-f) guardados en archivos de texto con un título, una pequeña descripción y la descripción del *pipeline*.

2.9. Cheese

Es un proyecto de GNOME nacido en el 2007 como parte de un proyecto de Google Summer of Code. Escrito inicialmente por el entonces estudiante Daniel G. Siegel y mantenido actualmente por David King. Cheese es un programa para capturar fotos y videos con la cámara web con la opción de aplicar filtros. Está licenciado bajo la licencia GPLv2, y escrito principalmente en el lenguaje de programación Vala, aunque con partes



(a) Ventana principal de Cheese

(b) Ventana de efectos de Cheese

Figura 2.6: Cheese

Fuente: autoría propia

escritas en C (The GNOME Project, s.f.-c) (The GNOME Project, s.f.-b).

Cheese lee los filtros o efectos de GNOME Video Effects y los usa para mostrarlos en su interfaz gráfica. Tanto la captura de videos (o fotos) como la aplicación de filtros en Cheese está dada internamente por GStreamer. Cheese crea internamente una pipeline bastante compleja, de la cual se muestra una simplificación en el diagrama de la *figura 2.7*. El diagrama ignora elementos como *capsfilters*, *bins* y algunos otros filtros y elementos.

2.10. OpenCV

Es una librería para visión artificial en tiempo real. Es software de código abierto siendo licenciada bajo la licencia BSD. Fue inicialmente desarrollado por Intel Inc (OpenCV team, s.f.-b). Está escrita en C y C++ con bindings para otros lenguajes de programación como Python y Java. Entre los distintos algoritmos que implementa OpenCV se encuentra la detección de objetos usando el algoritmo propuesto por Paul Viola y Michael Jones en "Rapid Object Detection using a Boosted Cascade of Sim-

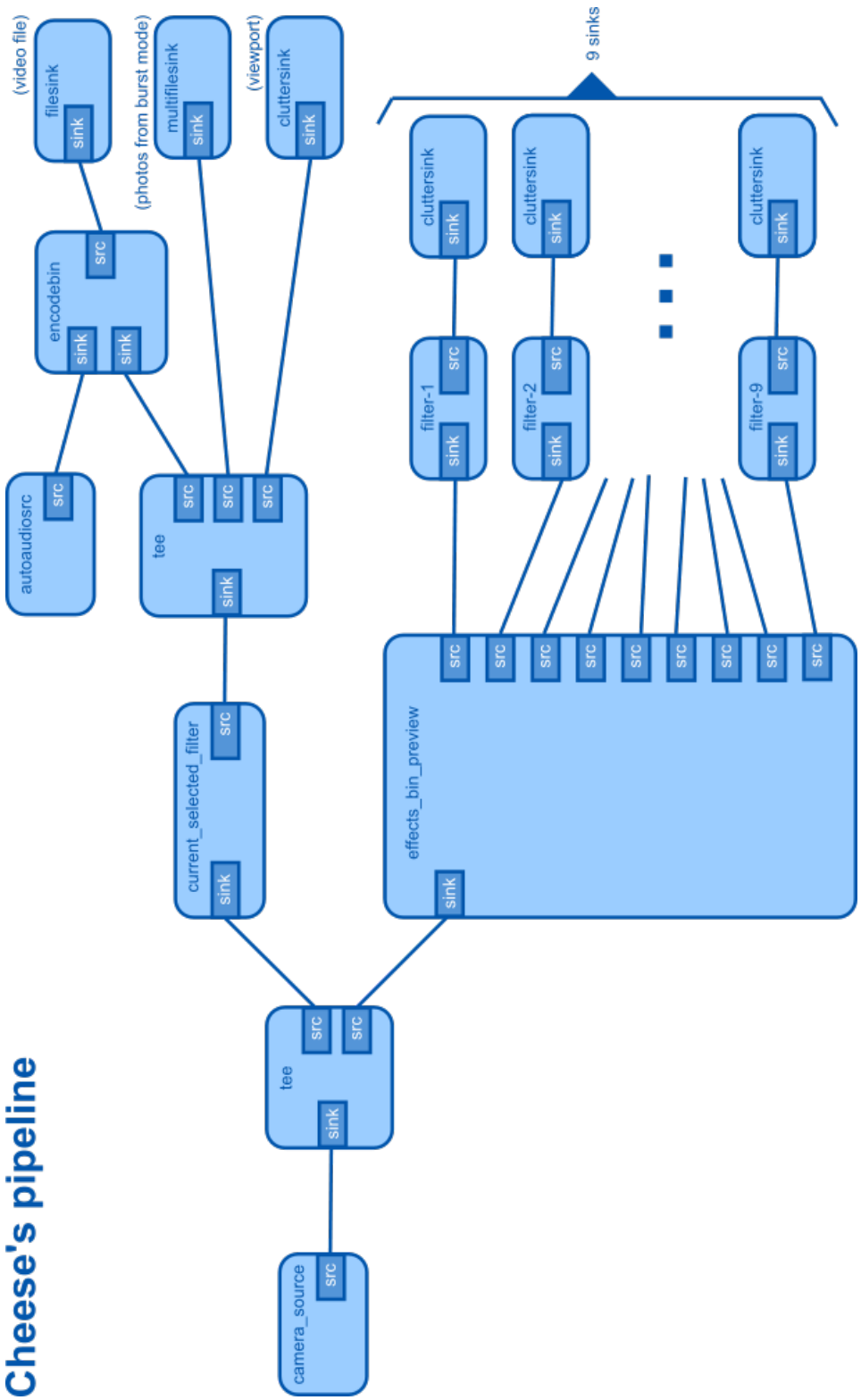


Figura 2.7: Simplificación del *pipeline* de Cheese. Imagen inspirada a partir de la generación de un diagrama en Graphviz con GStreamer en modo DEBUG

Fuente: Autoría propia

ple Features" y el algoritmo para la estimación de estados llamado filtro de Kalman. (OpenCV team, s.f.-b).

2.11. dlib

Es una librería que implementa una amplia cantidad de algoritmos en aprendizaje de máquina. Es software de código abierto licenciado bajo la licencia Boost License y escrita en C++ (dlib C++ Library, s.f.)(dlib C++ Library - License, s.f.). Entre los distintos algoritmos que implementa se encuentra el propuesto por Navneet Dalal y Bill Triggs en "Histograms of Oriented Gradients for Human Detection" (King, 2014a) y la detección del "landmark" del rostro con el algoritmo propuesto por Vahid Kazemi y Josephine Sullivan en "One Millisecond Face Alignment with an Ensemble of Regression Trees" (King, 2014b).

Capítulo 3

Estado del Arte

3.1. Revisión y discusión

En esta sección, se analiza qué librerías se debería escoger no solo por la disponibilidad de algoritmos útiles para realizar el seguimiento de rostros sino que sus licencias deben ser compatibles con Cheese. El método a seguir en este análisis es el de la revisión sistemática. En este método se plantea preguntas que son de la duda del autor para posteriormente analizarlas a detalle.

3.1.0.1. Palabras clave

A continuación, se muestra una lista de algunas palabras clave que fueron introducidos en los buscadores de Google y Github con el fin de resolver dudas.

En relación a librerías a usar:

- face tracking open source free software library
- dlib users
- opencv users
- face tracking dlib
- face tracking open cv
- 4dface

- 3d face tracking
- Landmark

En relación a las licencias:

- GPLv2 compatibility Boost license
- GPLv2 compatibility MIT license
- GPLv2 compatibility BSD license
- GPLv2 compatibility Apache license

3.1.0.2. Preguntas a resolver

P1: ¿Qué librería se debería usar siendo esta la que tenga mayor soporte por la comunidad de software libre de tal manera que se pueda esperar que Cheese dure en el largo plazo sin revertir los cambios debido a problemas en las dependencias?
P2: ¿Qué librería que sea compatible con GPLv2 usar?
P3: ¿Qué librería se debe usar?

3.1.1. Librerías a considerar según su soporte para realizar seguimiento de rostros

Según la búsqueda realizada dio como posibles opciones a usar *4dface*, *OpenCV*, *dlib* y *FaceX*. Estas librerías están escritas tanto en C++ como en C, lo cual significa que pueden ser empleadas sobre GStreamer cuya implementación de plugins está soportada especialmente en C y C++ por la comunidad. *4dface* es una librería escrita en C++ que soporta directamente el seguimiento de rostros en tiempo real y la reconstrucción de la forma de un rostro a partir de imágenes en tiempo real (un aproximado de 5fps); esta librería está desarrollada para integrarse con OpenCV (Huber, 2015a). *OpenCV*, a diferencia de *4dface* no soporta directamente el seguimiento de rostros en tiempo real ni tampoco la detección del landmark. No obstante, está soportada la detección de rostros, y aunque no es suficiente por sí misma, debido a lo expuesto en el capítulo de la problemática, se puede combinar con el uso del algoritmo del filtro de Kalman, el cual está implementado. En una situación parecida se encuentra *dlib* que tampoco está especializado en el rastreo de rostros, pero soporta la detección de *landmarks*. Finalmente, *FaceX* está especializado en la detección de landmarks (Cao, 2014a).

3.1.2. Librerías de detección y/o segumiento de rostros según su popularidad

Es importante buscar una librería ampliamente soportada por la comunidad de software libre que pueda ser útil para implementar el algoritmo de seguimiento de rostros y landmark a partir de videos (como capturas de la cámara web) en tiempo real.

Para ello solo serán consideradas librerías que aparecieron en las primeras páginas de las búsquedas, pues esto da un indicio de que son las más populares. Además se descartarán todas aquellas librerías cuya última actualización haya sido mayor a 2 años; por ejemplo, en caso de usar Git para el control de versiones, cuyo último commit sea mayor a 2 años. Además, en caso de usar Git, solo se considerarán librerías con más "forks", más "watch" y más "stars" (métricas de Github). En la búsqueda realizada se encontró que las siguientes librerías podrían ser útiles para resolver el problema.

Programa	último commit	forks	stars	watch
OpenCV	09-09-2017	13494	18075	1693
4dface	02-12-2016	133	268	44
dlib	10-09-2017	936	2701	262
FaceX	15-11-2015	86	95	18

Cuadro 3.1: Tabla construida en base a las métricas de los proyectos en Github al 10-09-2017.

Fuente: Repositorios de los proyectos en *git*

Se puede observar que las más populares son OpenCV y dlib. OpenCV según su página web cuenta con más 47 mil usuarios y un número de descargas que exceden los 14 millones (?, ?). Mientras dlib en su página web también asegura tener varios usados además de estar citado en distintas investigaciones académicas (Davis King, 2009).

3.1.3. Librerías de detección y/o segumiento de rostros según su compatibilidad con GPLv2

Saber que licencia es compatible con GPLv2 es importante porque si no fuera así, sería complicado que el cambio sea aceptado en Cheese o que sea aplicado a un fork de este. De hecho, si no se realiza esta tarea desde el principio, y se optara por error por una dependencia con licencia incompatible, el trabajo realizado podría resultar vano. GPLv2 es una licencia de software libre, pero tiene ciertas limitaciones respecto a GPLv3.

Programa	Licencia	¿Compatible con GPLv2?
OpenCV	BSD License	Sí
4dface	Apache License 2.0	No
dlib	Boost Software License 1.0	Sí
FaceX	MIT License	Sí
FaceX-train	GPLv3	Sí

Cuadro 3.2: Tabla de comparación de programas según su compatibilidad con GPLv2.

Fuente: Repositorios de los proyectos en *git* (Pavlenko, 2013)(King, 2008)(Cao, 2014b) (Huber, 2015b) y para la compatibilidad con GPLv2 la web de la FSF (Free Software Foundation, s.f.-c)

3.2. Conclusiones

Si bien *4dface* podía ser una buena alternativa por sus características, se descarta la posibilidad de su uso esencialmente debido a ser incompatible con la licencia de Cheese. *FaceX* también se descarta porque no hay indicios para pensar que su desarrollo continuará en los siguientes años. Tanto *OpenCV* como *dlib* parecen ser la mejor opción no solo porque son compatibles con la licencia de Cheese, sino también porque implementan algoritmos que son útiles para el seguimiento de objetos (en este caso rostros) en tiempo real y detección de landmark. La combinación de ambos representa una alternativa factible para solucionar el problema.

Capítulo 4

Filtro de detección, rastreamiento y detección de puntos faciales de múltiples rostros: *gstcheesefacetrack*

Este capítulo consta de tres partes específicas. En la primera sección, se detalla paso a paso y explica el algoritmo implementado para el rastreamiento de múltiples rostros y detección de puntos faciales cuya implementación como filtro para *GStreamer* se llamó *gstcheesefacetrack*. En la siguiente sección, se explica cómo usar el filtro con algunos ejemplos incluidos. Y finalmente, en la última sección, se mostrarán algunas métricas como precisión, exhaustividad (*recall*) y valor-F (*f1-score*).

4.1. Algoritmo de detección y rastreamiento de múltiples rostros y de detección puntos faciales

4.1.1. Introducción

El algoritmo desarrollado consiste en ciclos, en los cuales por cada ciclo en el primer cuadro se detectan los rostros y en los siguientes cuadros se realiza rastrea a los rostros. La detección de rostros suele ser un proceso costoso comparado con el rastreo, es por ello que mientras más largo sea el tiempo en el cual se rastrea los rostros habrá menos latencia. Sin embargo, a veces puede que nuevas personas entren a la escena capturada por la cámara web o que algunas personas salgan de esta. Además puede suceder que el rastreador pierda su objetivo o el resultado de este sea poco probable. Es por esta razón que es necesario cada cierto tiempo detectar rostros. La detección de rostros que solo se raliza una vez cada ciclo es aprovechada para verificar si rostros de personas entraron o

salieron de la escena. Además esta fase de detección es una oportunidad para verificar si el rastreador está dando resultados probablemente correctos o no. Si el rastreador dice que hay un rostro donde en verdad según el detector no existe ninguno, esto se corrige en esta fase. En esta fase de detección, además, se reasignan (o reordenan) los rostros de tal manera que cada rostro detectado preserve su identidad. Además de ello, a veces puede suceder que el rastreador pierda su objetivo (el rostro rastreado). Este caso es un caso de urgencia y excepción al ciclo explicado inicialmente, y se entrará en una fase de detección.

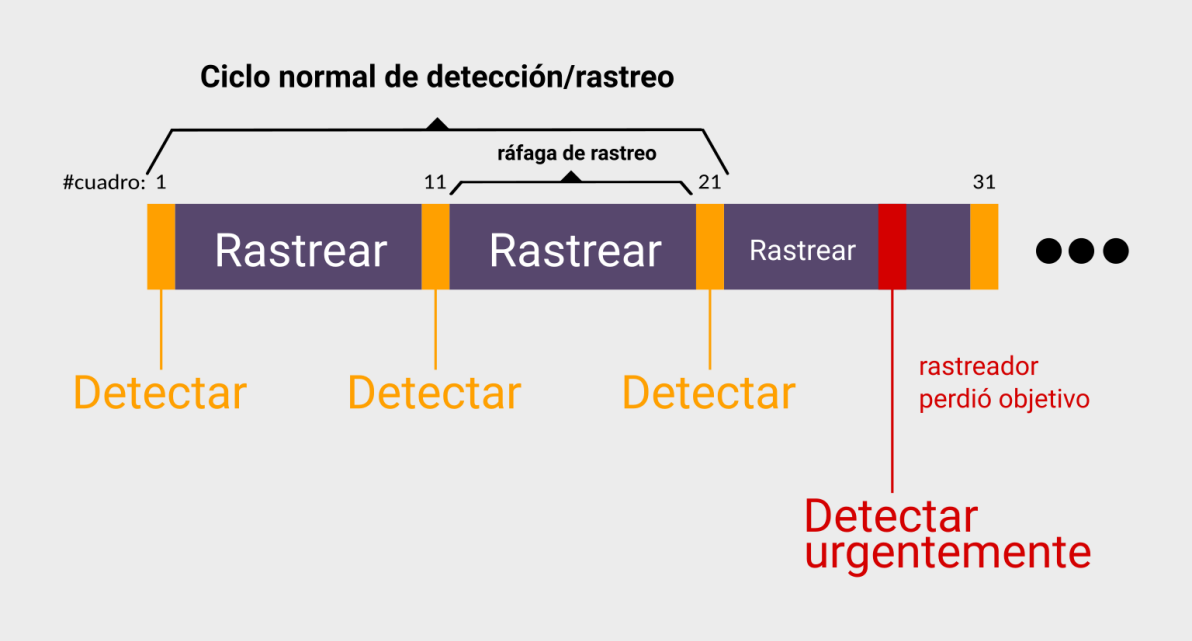


Figura 4.1: Diagrama de ciclo de ciclo de detección y rastreo realizado por el algoritmo

Fuente: Autoría propia

4.1.2. ¿Cómo se almacena la información de los rostros?

Cada rostro es único. Las tablas hash permiten tener llaves únicas (que no se repiten) y además tienen tiempo de acceso $O(1)$, por tanto guardar información de cada rostro en una tabla hash ha resultado conveniente para este algoritmo. La llave de la tabla hash es un número identificador (*id*) y el valor relacionado a esta llave es una estructura que contiene información específica del rostro.
La estructura definida contiene la información del rectángulo (*rectngulo*) y puntos faciales (*landmark*) que delimita el rostro en el último cuadro en el cual este fue detectado o rastreado. Cada estructura además tiene un rastreador (*rastreador*) el cual es definido a gusto del usuario y puede ser Median Flow, MIL, KCF, TLD o Boosting de los cuales se recomienda Median Flow debido a que es capaz de rastrear un rostro en distintas

escalas (como se explicó en 1.2.5.0.2.4). Antes de crear un rastreador, ningún rastreador ha sido asignado al rostro (estado $NO\_ASIGNADO$), luego al asignarse un rastreador (*rastreador*) este pasa al estado ($NO\_INICIALIZADO$): ambos estados mencionados son solo de paso. Cuando se inicializa el rastreador se pasa al estado de inicializado ($INICIALIZADO$) y cuando se empieza el proceso de rastreo este pasa a estado de rastreando ($RASTREANDO$). Finalmente, se guarda el último número de cuadro en el cual el rostro fue detectado.

Estructura 1 Estructura que contiene la información de la ***identidad*** de un rostro

estructura RostroT {

rectngulo, Rectángulo que delimita el rostro de la cara.

landmark, Arreglo de puntos faciales. Inicialmente vacía.

rastreador, El rastreador qué puede ser Median Flow, MIL, KCF, TLD o Boosting, según sea especificado como parámetro.

estadoRastreador, El estado del rastreador el cuál puede ser $NO\_ASIGNADO$, $NO\_INICIALIZADO$, $INICIALIZADO$, o $RASTREANDO$. Inicialmente $NO\_INICIALIZADO$.

ultimoCuadroDetectado, el número de cuadro en el cual el rostro fue detectado o rastreado por última vez.

}

4.1.3. Explicación detallada del algoritmo

El algoritmo propuesto cuyo (pseudocódigo principal puede verse en el cuadro titulado algoritmo 2) ejecuta un procedimiento cíclicamente para cada cuadro de video capturado (en este caso por la cámara web). Este procedimiento inicia escalando la imagen por un factor de escala ($factor\_escala$), la razón de ello es para agilizar la detección de rostros. Una variable global $ROSTROS\_TABLA$ la cual es una tabla hash tal como se describió en la sesión anterior está inicialmente vacía, pues aun no se ha procesado ningún rostro, por ello las líneas 5-14 serán explicadas más adelante. Las siguientes subsecciones explican a detalle cada parte del algoritmo.

Algoritmo 2 Algoritmo de rastreamiento y detección de puntos faciales

Global: $ROSTROS\_TABLA$, tabla hash **inicialmente vacía** (llave: *entero*, valor: rectángulo delimitador y *landmark*)
Global: $ÚLTIMO\_ID$, el siguiente identificador a asignar. Valor inicial: 0
Global: $TIPO\_RASTREADOR$, el tipo de rastreador a usar que puede ser Median Flow, MIL, KCF, TLD o Boosting, según sea especificado como parámetro.
Global: N, el número de cuadro actual. Valor inicial: 1
Parámetro: $factor\_escala \in [0, 1]$
Parámetro: $factor\_distancia \in [0, 1]$
Parámetro: $brecha\_detección \in \mathbb{N}$
Parámetro: $borrado\_threshold \in \mathbb{N}$
Entrada: I, una imagen de dimensiones $ancho \times largo$

1: **procedure** PROCESARCUADRO(I)
2: $I_s \leftarrow I \times factor\_escala$ ▷ Escalar imagen.
3: IdsRostrosPerdidos← [] ▷ Lista de *ids* de rostros cuyo rastreador perdió el objetivo.
4: IdsRostrosNoCreados← [] ▷ Lista de *ids* de rostros no creados en el cuadro actual.
5: INTENTARBORRARROSTROS
6: **para** (id, rostro) **en** $ROSTROS\_TABLA$ **hacer**
7: *éxito* ← rostro.RASTREAR(I_s)
8: **if** *éxito* **then**
9: rostro.ultimoCuadroDetectado ← N
10: **else**
11: IdsRostrosPerdidos.AGREGAR(id)
12: **end if**
13: IdsRostrosNoCreados.AGREGAR(id)
14: **end para**
15: enFaseDeDeteccion← N **mod** $brecha\_detección = 1$
16: **if** enFaseDeDeteccion **o** LONGITUD(IdsRostrosPerdidos) > 0 **then**
17: detsRz ← detectarRostros(I_s) ▷ Lista de rectángulos.
18: **if** LONGITUD($ROSTROS\_TABLA$) $= 0$ **then**
19: CREARYAGREGARROSTROS($ROSTROS\_TABLA$, I_s, detsRz);
20: **end if**
21: **if** LONGITUD(IdsRostrosNoCreados) > 0 **y** LONGITUD(detsRz) > 0 **then**
22: centroidesRostrosDetectados ← [CENTROIDE(r) **para** r **en** detsRz]
23: centroidesRostrosNoCreados ← [CENTROIDE(r) **para** r **en** $ROSTROS\_TABLA$[IdsRostrosNoCreados].rectángulos]
24: MatrizDeCostos←INICIALIZARMATRIZDECOSTOS(centroidesRostrosDetectados, centroidesRostrosNoCreados)
25: asignaciones←MÉTODOHÚNGARO(MatrizDeCostos) ▷ Continúa...

4.1.3.1. Fase de detección

Se empieza por analizar si el número de cuadro actual corresponde a una fase de detección normal (algoritmo 2, línea 15). La fase de detección se da en el primer cuadro y cada cierta cantidad de cuadros $brecha\_detección \in \mathbb{N}$. También existe un caso en el cual se puede entrar a la fase de detección que sucede si alguno de los rastreadores asignado a un rostro perdió su objetivo (algoritmo 2, línea 11).
La fase de detección (algoritmo 2, líneas 16-55) inicia con la detección de rostros (algoritmo 2, líneas 17) usando un detector de objetos siguiendo el método de histograma de gradientes (HOG). De no haber rostros previamente registrados en la tabla hash $ROSTROS\_TABLA$, se procede a crear una entrada en esta tabla para cada rostro detectado (algoritmo 2, línea 19, con más detalles en el apartado 4.1.3.4). Para el cuadro actual, de no haber habido rostros previamente a los detectados (o creados) durante la fase de detección y de haber nuevos rostros detectados en el cuadro actual (algoritmo 2, líneas 21), se procede a construir el problema de asignación.
Debido a que en esta fase el detector no mantiene el orden de los rostros. El problema de asignación planteado se pregunta: "¿Qué rostro no creado antes de la fase de detección (en el cuadro actual) debe asignarse o está más cerca (y por tanto más probable) al rostro actualmente detectado"? A continuación se detalla un ejemplo sencillo de este problema.

Figura 4.2: Representación de la disposición de los rostros rastreados un cuadro antes de la fase de detección. Los círculos amarillo y rojo representan cada uno un rostro, mientras las letras A y B representan los identificadores de cada rostro

Fuente: Autoría propia

En la figura 4.2, se observa la representación de los rostros que fueron rastreados un cuadro antes de la fase de detección (izquierda: A, derecha: B). Luego, en la figura

Algoritmo 2 Algoritmo de rastreamiento y detección de puntos faciales (continación)

```
26:             para i ← 0 hasta LONGITUD(centroidesRostrosDetectados) hacer
27:                 asignado←asignaciones[i]≠ −1
28:                 crearRostro←verdadero
29:                 if asignado then
30:                     id←IdsRostrosNoCreados[asignaciones[i]]
31:                     rostro← ROSTROS_TABLA[id]
32:                     crearRostro←falso
33:                     c1 ←CENTROIDE(centroidesRostrosDetectados[i])
34:                     c2 ←CENTROIDE(rostro.rectángulo)
35:                     distancia←DISTANCIAEUCLIDIANA(c1,c2)
36:                     maxDistancia←factor_distancia×detsRz.ancho
37:                     if distancia ≥ maxDistancia then
38:                         rostro.rastreador.LIBERAR()
39:                     end if
40:                 end if
41:                 if crearRostro then
42:                     CREARYAGREGARROSTROS(ROSTROS_TABLA, I_s,
    [detsRz[i]])
43:                 else
44:                     id←IdsRostrosNoCreados[asignaciones[i]]
45:                     rostro← ROSTROS_TABLA[id]
46:                     if rostro.estadoRastreador = NO_ASIGNADO then
47:                         rostro.rectángulo←detsRz[i]
48:                         rostro.CREARRASTREADOR(tracker)
49:                         rostro.INICIALIZARRASTREADOR(I_s)
50:                         rostro.ultimoCuadroDetectado ← N
51:                     end if
52:                 end if
53:             end para
54:         end if
55:     end if
56:     para (id, rostro) en ROSTROS_TABLA hacer
57:         mostrarRostro← rostro.ultimoCuadroDetectado = N
58:         if mostrarRostro then
59:             if rostro.estadoRastreador = INICIALIZADO then
60:                 rostro.landmark ← DETECTARPUNTOSFACIALES(I_s)
61:             end if
62:             MOSTRARROSTRO(rostro)
63:         end if
64:     end para
65:     N ← N + 1
66: end procedure
```

Figura 4.3: Representación de lo que sucede durante la fase de detección. Las líneas punteadas representan las posiciones anteriores de los rostros. Se observa que el detector de rostros intercambió los identificadores A y B en el orden B y A. Evidentemente, el orden debería ser A y B

Fuente: Autoría propia

4.3, se representa la fase de detección, donde los rostros se han movido ligeramente pero según el detector, las etiquetas se han cambiado (izquierda: B, derecha: A). Lo esperado, sería seguir manteniendo el orden inicial de las etiquetas. Es poco probable que de un cuadro a otro un rostro se haya movido mucho, por lo tanto debe estar muy cercano a su posición previa. En la figura 4.4, se procede a medir las combinaciones de distancias con el propósito de encontrar cuáles son los rostros más cercanos a los rostros detectados actualmente. Se observa a simple vista, que para el rostro más cercano al rostro detectado de la izquierda (en amarillo) en el cuadro N la distancia más cercana es $d1$ y que para el rostro detectado de la derecha (en rojo) es $d2$. Finalmente, en la figura 4.5 se ve que las etiquetas A y B son reasignadas correctamente. El problema de asignación planteado anteriormente, es fácil de resolver en el ejemplo de la figura. Sin embargo, este se complica cuando hay más rostros y estos están muy cerca, es por ello que debe usarse un método más genérico.

El problema de asiganación planteado se resuelve mediante el método húngaro. Se define una matriz de costos ($MatrizDeCostos$) cuyas filas corresponde a "la cantidad de rostros no creados en el cuadro actual antes de la fase de detección" y cuyas columnas corresponde a la cantidad de "rostros detectados en el cuadro actual". Cada elemento de la matriz ($MatrizDeCostos_{i,j}$) corresponde a la distancias euclidianas entre el centroide del "rostro no creados antes de entrar a la fase de detección" en la posición i respecto al centroide del "rostro detectado en el cuadro actual" en la posición j (algoritmo 2, líneas 22-24).

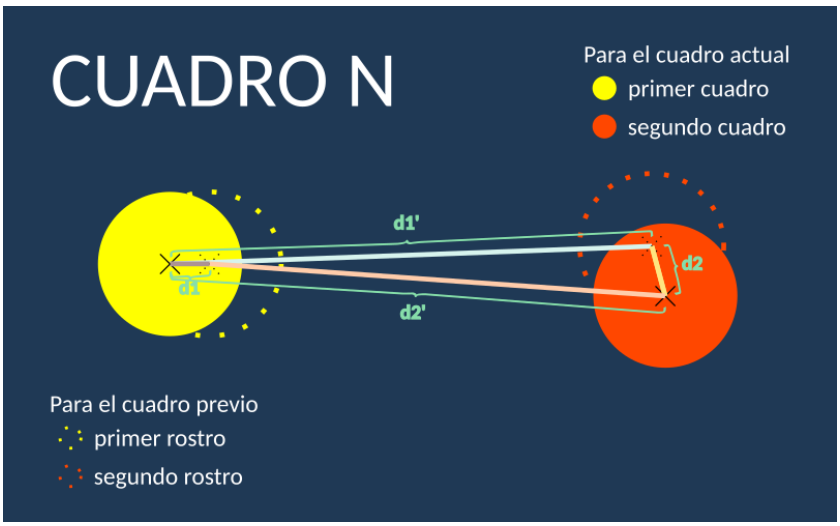


Figura 4.4: Cálculo de las distancias entre los rostros no detectados en el cuadro anterior a la fase de detección y los rostros detectados durante la fase de detección. Se observa que $d1$ y $d2$ son las distancias menores.

Fuente: Autoría propia

Figura 4.5: Estado final de las etiquetas A y B tras resolver el problema de asignación.

Fuente: Autoría propia

Debido a que la matriz de costos puede no ser cuadrada, habrá algunos rostros que no serán asignados. Otro problema es que el resultado de asignar "un rostro no detectado antes de la fase de detección" a uno "detectado en el cuadro actual" puede estar muy lejos por diversas razones (por ejemplo, el detector detectó un rostro cuando anterior-

mente se registraba dos rostros). Para verificar si sucede algunos de estos problemas se itera sobre cada rostro detectado (algoritmo 2, líneas 26). Si el rostro detectado fue asignado, se verifica que esté suficientemente cerca (es decir que la distancia al rostro asignado no exceda un límite o *threshold* equivalente a $factor\_distancia$ veces el ancho del rectángulo delimitador del rostro detectado). De no estar suficientemente cerca, el rastreador relacionado a este rostro se libera y su estructura pasa a estado no disponible ($NO\_ASIGNADO$). Ver algoritmo 2, líneas 29-40. De no haber sido asignado el rostro detectado, se procede a crear una entrada en esta tabla para cada rostro detectado (algoritmo 2, línea 19, con más detalles en el apartado 4.1.3.4), ya que puede tratarse de un nuevo rostro que ingresó a la escena. En caso contrario (algoritmo 2, línea 44-51), se reinicia un rastreador para la estructura correspondiente y se actualiza el último cuadro en el cual el rostro fue detectado en la estructura ($RostroT.ultimoCuadroDetectado$).

4.1.3.2. Borrado o limpieza de rostros no detectados recientemente

En la línea 5, se llama al procedimiento $IntentarBorrarRostros$. Aquí se borran los rostros que no fueron detectados o rastreados hace $borrado\_threshold$ cuadros de la tabla hash de rostros. Esta limpieza se debe a que a medida que pasa el tiempo es posible que un rostro se haya movido mucho o incluso que el rostro haya salido de la escena. El identificador para este rostro será simplemente desechado y nunca más podrá volver a usarse. Ver algoritmo 4 para más detalles.

4.1.3.3. Fase de rastreo

La mayor parte del tiempo se realiza rastreo de rostros. Si bien se puede usar una amplia variedad de rastreadores a elegir como Median Flow, MIL, KCF, TLD, Boosting, entre otros, se recomienda Median Flow, puesto que es capaz de estimar la escala del objetivo (por ejemplo, si el rostro se aleja o se acerca) tal como se explicó en el apartado 1.2.5.0.2.4. Durante cada cuadro en la fase de rastreo que son los cuadros restantes luego de la fase detección (durante $brecha\_detección - 1$ cuadros), se intenta predecir (algoritmo 2, línea 7) la posición actual de cada rostro en la tabla hash de estructuras ($ROSTRO\_TABLA$). De predecir la posición del rostro exitosamente, se modifica la estructura del rostro respectivo actualizando el último cuadro detectado al actual (algoritmo 2, línea 9). De lo contrario, si el rastreador perdió su objetivo[1], se agrega el identificador del rostro a una lista para ser usada en la fase de detección (algoritmo 2, línea 11).

[1]Un rastreador Median Flow pierde su objetivo cuando la mediana de la longitud de la diferencia entre desplazamientos supera un threshold. En este caso, si supera los 10 píxeles.

4.1.3.4. Creación y agregado rostros a la tabla hash

Este procedimiento se encarga de crear para cada rectángulo delimitadir las estructuras de rostros y asignarles el siguiente identificador disponible (algoritmo 3, línea 3). Para cada nueva estructura de rostro creada se le asigna el rectángulo delimitador y se crea e inicializa un rastreador para cada estructura (algoritmo 3, línea 4-9).

Algoritmo 3 Procedimiento que crea nuevos rostros con rastreadores inicializados y los inserta en tabla hash

Global: $ROSTROS\_TABLA$, tabla hash **inicialmente vacía** (llave: *entero*, valor: rectángulo delimitador y *landmark*)
Global: $TIPO\_RASTREADOR$, el tipo de rastreador a usar que puede ser Median Flow, MIL, KCF, TLD o Boosting, según sea especificado como parámetro.
Global: $ÚLTIMO\_ID$, el siguiente identificador a asignar. Valor inicial: 0
Global: N, el número de cuadro actual. Valor inicial: 1

```
1: procedure crearYAgregarRostros(ROSTROS_TABLA, I, rectángulos)
2:     para rectngulo en rectángulos hacer
3:         ÚLTIMO_ID ← ÚLTIMO_ID + 1
4:         rostro← instanciar RostroT
5:         rostro.ultimoCuadroDetectado← N
6:         rostro.rectángulo ← rectngulo
7:         ROSTRO[ÚLTIMO_ID] ←rostro
8:         ROSTRO[ÚLTIMO_ID].crearRastreador(TIPO_RASTREADOR)
9:         ROSTRO[ÚLTIMO_ID].inicializarRastreador(I)
10:    end para
11: end procedure
```

4.1.3.5. Detección de puntos faciales y mostrado de rostros

Una vez terminada la fase de detección, con la tabla hash $ROSTROS\_TABLA$ (probablemente) llena, para cada rostro se estima los puntos faciales (*landmarks* ignorando los rostros cuyo último cuadro de detección (*ultimoCuadroDetectado*) fue anterior a N. Para ello, tomando el rectángulo delimitador de cada rostro se aplica un algoritmo de alineación de rostros con un conjunto de árboles de regresión (algoritmo 3, línea 60), el mismo que se usaba en la metodología detallada en el apartado 1.2.5.0.2.1, que permite detectar los *landmarks* de cada rostro en milisegundos. Finalmente, todos estos rostros son considerados rostros presentes en el cuadro actual y son objeto de procesamiento (algoritmo 3, línea 62; por ejemplo, para dibujar los rectángulos delimitadores y los puntos faciales).

Algoritmo 4 Borrado de información de rostros que no han sido detectados o rastreados durante un tiempo considerable

Global: $ROSTROS\_TABLA$, tabla hash **inicialmente vacía** (llave: *entero*, valor: rectángulo delimitador y *landmark*)
Global: N, el número de cuadro actual. Valor inicial: 1
Parámetro: $borrado\_threshold \in \mathbb{N}$

```
1: procedure IntentarBorrarRostros
2:     para (id, rostro) en ROSTROS_TABLA hacer
3:         Δ ← N−rostro.ultimoCuadroDetectado
4:         if Δ > borrado_threshold then
5:             borrar(ROSTROS[id])
6:         end if
7:     end para
8: end procedure
```

4.1.4. Explicación del nombre *gstcheesefacetrack*

Al filtro desarrollado se le ha asignado el nombre de *gstcheesefacetrack* y **en adelante se hará referencia a este filtro por este nombre**. La asignación del nombre se explica a continuación: como la mayoría de elementos de *GStreamer* tiene en su nombre el prefijo *gst* y además se le pone el prefijo *cheese* debido al programa al cual se aplicará. Luego, lleva el término *face* debido a que está ubicado en el directorio *gst-plugins-cheese/gst/face* lugar donde se ubica filtros que usan datos de lo rostros. Finalmente, *track* se debe a que este algoritmo se encarga principalmente de rastrear rostros.

4.2. Diseño y *capabilities*

El filtro *gstcheesefacetrack* hereda del filtro *gstopencvvideofilter*. Consecuentemente, los *pad templates* son los mismos que los que usa el filtro *gstopencvvideofilter*, y por ende, las *capabilities* (formatos de video, *fps*, dimensiones, entre otras propiedades que se definen para ser aceptadas) son las mismas que las de *gstopencvvideofilter*.

4.3. Propiedades

Las propiedades del filtro son las siguientes:

Propiedad	Tipo	Descriptión de la propiedad
display-bounding-box	*bool*	Establece si se debería dibujar el rectángulo delimitador (en color amarillo) rodeando cada rostro detectado o rastreado.
display-id	*bool*	Establece si se debería mostrar el identificador para cada rostro detectado o rastreado.
display-landmark	*bool*	Establece si se debería dibujar los puntos faciales para cada rostro detectado o rastreado.
display-detection-phase	*bool*	Establece si se debería dibujar el rectángulo delimitador (en color azul) rodeando cada rostro detectado durante la fase de detección.
landmark	*string*	La ruta donde se encuentra el modelo entrenado para estimar los puntos faciales. Este modelo debe haber sido serializado con el objeto de *dlib shape_predictor_trainer* para 68 puntos faciales. Un ejemplo de modelo entrenado se puede encontrar en https://github.com/davisking/dlib-models.
tracker	*enum*	El tipo de rastreador a usar. El cual puede ser: 1. *boosting*: Boosting 2. *goturn*: GOTURN 3. *kcf*: Kernelized Correlation Filters 4. *median-flow*: Median Flow (**por defecto**) 5. *tld*: Tracking Learning Detection
delete-threshold	*unsigned int*	Establece el número de cuadros que debe pasar para borrar un rostro si este no fue detectado durante ese periodo.
scale-factor	*float*	Establece el factor de escala con el cual un cuadro será escalado antes del proceso de rastreo o detección.
max-distance-factor	*float*	Establece el máximo factor de distancia para calcular la máxima distancia que será aplicada entre los centroides del resultado del rastreador y detector. Este factor será aplicado contra el ancho del rectángulo delimitador. El valor de esta propiedad debe ser muy bajo.
detection-gap-duration	*unsigned int*	Establece el máximo número de cuadros entre cada fase de detección.

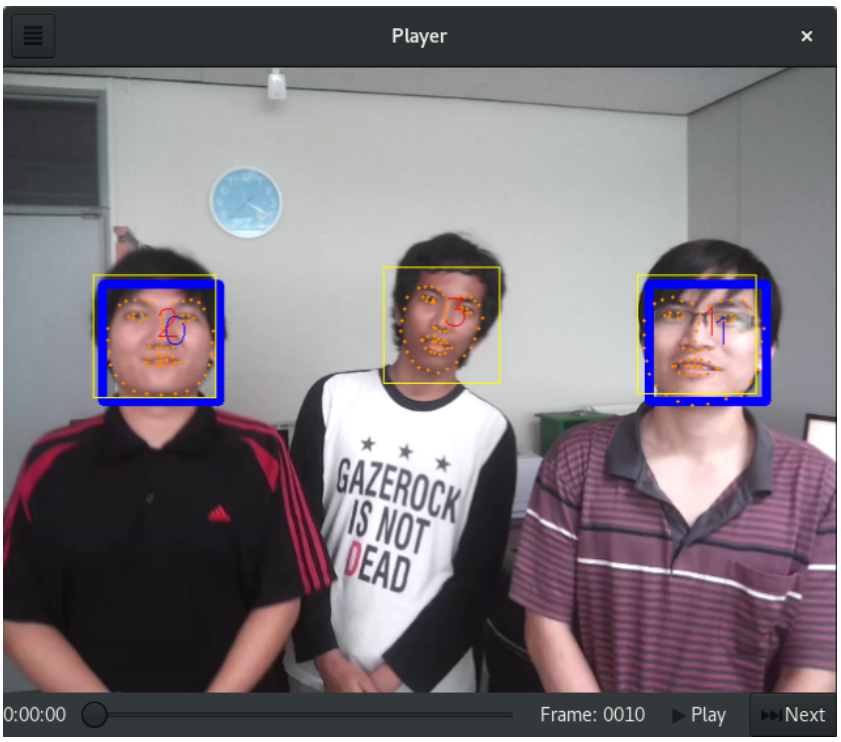


Figura 4.6: Se diseñó una herramienta que permitía verificar cuadro por cuadro el proceso del algoritmo. En la imagen se observa una instantánea de un video en plena fase de detección. Los rectángulos amarillos representan el resultado del rastreador, mientras que los rectángulos en color azul representan el resultado del detector. Se observa que mientras el rastreador sigue a tres rostros, el detector solo detecta dos rostros. En color naranja se muestran los puntos faciales detectados por cada rostro.

La imagen mostrada dentro de la ventana de escritorio corresponde a un fotograma del un video de un dataset del *Smart Computing LABoratory* (LAB, s.f.)

4.4. Ejemplos de *pipelines*

Existe una lista muy grande de *pipelines* que se pueden formar con este filtro. Se muestra algunos ejemplos sencillos:

Desde la cámara web:

```
$ gst-launch-1.0 v4l2src ! videoconvert ! \
  cheesefacetrack scale-factor=0.5 \
  landmark=shape_predictor_68_face_landmarks.dat detection-gap-duration=18 ! \
  videoconvert ! xvimagesink
```

Desde un archivo de video:

```
$ gst-launch-1.0 filesrc location=archivo.ogv ! \
  decodebin ! videoconvert ! \
  cheesefacetrack scale-factor=0.5 max-distance-factor=0.15 \
  landmark=shape_predictor_68_face_landmarks.dat ! \
  videoconvert ! xvimagesink
```

4.5. Métricas

4.5.1. *Dataset*

Para evaluar métricas de desempeño del algoritmo se tomó usó un *dataset* del reporitorio de publicaciones del *Smart Computing LABoratory* de la Universidad Nacional de Chonnam llamado *Multiple faces* (LAB, s.f.). El dataset contiene varios videos en los cuales múltiples personas aparecen en la escena donde en algunos videos la cámara está quieta y en otros casos incluso en movimiento. Los detalles del dataset empleado se puede revisar con mayor detenimiento en el artículo *Construction of a Video Dataset for Face Tracking Benchmarking Using a Ground Truth Generation Tool* (Do y cols., 2014) desarrollado por expertos de tal universidad. El *dataset* además con cada video tiene un archivo de texto plano donde se anota los rectángulos delimitadores de cada rostro por cuadro (Do y cols., 2014). Sin embargo, este dataset tenía un problema: tampoco se respetaba el orden o identidad de los rostros. Es por ello que se tenía que o bien modificar el archivo de texto manualmente o buscar un método automático para corregir el problema. Se optó por el método automático, y fue el método húngaro fue de gran ayuda para resolver este problema.

4.5.2. Parámetros empleados

De manera empírica se observó que valores adecuados para los parámetros del algoritmo 2 eran los siguientes:

Parámetro	**Valor**
$factor\_distancia$	0.15
$brecha\_detección$	10
$borrado\_threshold$	72

4.5.3. Métricas

Se usan las métricas sugeridas en el paper de construcción del dataset descrito anteriormente. Sea S_R el área del rectángulo delimitador resultado del algoritmo de rastreo y S_G el área del recctángulo de los datos reales del *dataset* y M el área de la intersección de ambos (Do y cols., 2014). Dado ello, la precisión (*precision*), exhaustividad (*recall*) y valor-F (*f-score* o f) están dados según las siguientes fórmulas:

$$precision = \frac{M}{S_R}$$

$$recall = \frac{M}{S_G}$$

$$f = \frac{1}{\frac{0,5}{S_R} + \frac{0,5}{S_G}}$$

(Do y cols., 2014)

4.5.4. Resultados

A continuación se muestra el desempeño del algoritmo de rastreo de rostros para distintos videos del dataset:

Para los videos en la ruta *Multiple_faces/Camera/* en el cual dos personas muestran sus rostros en la escena ante una cámara en movimiento, se obtuvieron los siguientes resultados:

Métrica	**Valor promedio**
precision	0.905996934162
recall	0.790615895094875
f-score	0.827930771659125

Los resultados mostrados anteriormente son resultado de promediar el promedio de cada métrica por cuadro de los resultados mostrados a continuación:

Nota: al final del documento, se anexa diagramas de la variación de cada métrica por buffer y por rostro para el primer video probado (cuyos promedios se muestra en la siguiente tabla).

Resultados para el video en la ruta *Multiple_faces/Camera/multiple_camera1.mp4*	
Primer rostro	
Métrica	**Valor promedio por cuadro**
precision	0.935330265503
recall	0.766618699373
f-score	0.836228162369
Segundo rostro	
Métrica	**Valor promedio por cuadro**
precision	0.956341677422
recall	0.78393741114
f-score	0.854720557981

Resultados para el video en la ruta *Multiple_faces/Camera/multiple_camera2.mp4*	
Primer rostro	
Métrica	**Valor promedio por cuadro**
precision	0.940617557867
recall	0.8063989967
f-score	0.859244287497
Segundo rostro	
Métrica	**Valor promedio por cuadro**
precision	0.844132382431
recall	0.876186023593
f-score	0.841806407169

Resultados para el video en la ruta *Multiple_faces/Camera/multiple_camera3.mp4*	
Primer rostro	
Métrica	**Valor promedio por cuadro**
precision	0.985272026991
recall	0.66406228513

f-score	0.792761547765
Segundo rostro	
Métrica	**Valor promedio por cuadro**
precision	0.964246487081
recall	0.671698742669
f-score	0.785508782058

Resultados para el video en la ruta *Multiple_faces/Camera/multiple_camera4.mp4*	
Primer rostro	
Métrica	**Valor promedio por cuadro**
precision	0.878257187382
recall	0.831826127569
f-score	0.84208427769
Segundo rostro	
Métrica	**Valor promedio por cuadro**
precision	0.743777888619
recall	0.924198874585
f-score	0.811092150744

4.6. Medición de latencia por *buffer*

En esta sección se muestra los resultados de la medición de latencia de un *pipeline* usando *gstcheesefacetrack*. Para medir la latencia se empleó la herramienta *GST_TRACE*. Debido a que esta medición no es exacta ya que puede estar afectada por el entorno, se desarrolló una herramienta que permite ejecutar *gst-launch-1.0* por N veces consecutivas y promediar la latencia por *buffer*.

4.6.0.1. Detalles de hardware y software empleado

Para realizar la medición se empleó un computador portátil *Lenovo G50* con las siguientes características:

Memoria	
Tamaño	8GiB
CPU	
Producto	Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz
Vendedor	Intel
Tamaño	2425MHz
Capacidad	2700MHz
Ancho	64 bits
Video	
Producto	HD Graphics 5500
Vendedor	Intel Corporation
Cámara	
Producto	Lenovo EasyCamera: Lenovo EasyC
Driver	uvcvideo
Versión del driver	4.14.11
Formatos aceptados	MJPG (Motion-JPEG), YUYV (YUYV 4:2:2)
Resoluciones aceptadas	En MJPG: ▪ 1280x720 (intervalo: 0.033s, 30fps) ▪ 160x120 (intervalo: 0.033s, 30fps) ▪ 320x240 (intervalo: 0.033s, 30fps) ▪ 640x360 (intervalo: 0.033s, 30fps) ▪ 640x480 (intervalo: 0.033s, 30fps) ▪ 800x600 (intervalo: 0.033s, 30fps) En YUV: ▪ 1280x720 (intervalo: 0.100s, 10fps) ▪ 160x120 (intervalo: 0.033s, 30fps) ▪ 320x240 (intervalo: 0.033s, 30fps) ▪ 640x360 (intervalo: 0.033s, 30fps) ▪ 640x480 (intervalo: 0.033s, 30fps) ▪ 800x600 (intervalo: 0.100s, 10fps)

El filtro fue compilado con el sistema de compilación *mesonbuild* usando el modo *release* (con optimizaciones y sin depuración) y compilado con el conjunto de instruc-

ciones de procesador *SSE4.1*. Es preciso indicar que es importante compilar el filtro en modo al menos en modo *release* para tener un desempeño aceptable.

4.6.0.2. *Pipeline* probado

El pipeline que se ejecutó fue el siguiente:

```
$ gst-launch-1.0 v4l2src ! videoconvert ! video/x-raw,framerate=30/1 ! \
    cheesefacetrack max-distance-factor=0.15 scale-factor=0.5 \
    landmark=shape.dat display-landmark=true detection-gap-duration=10 ! \
    fakesink num-buffers=500
```

4.6.0.3. Resultados de la medición de latencia

Se ejecutó la *pipeline* mencionada en la subsección anteriormente para 500 *buffers* (donde cada *buffer* contenía 1 cuadro) por 30 veces (como se observa, a 30 fps). Los cuadros de entrada eran en formato YUV y tenían una resolución de 1280x720.
La primera medición se ejecutó para un rostro frente a la webcam (ver figura 4.7), mientras que en la segunda medición se realizó para dos rostros (ver figura 4.8). La cámara estaba estática y cada rostro realizaba poco movimiento con el fin de estresar el algoritmo. Los resultados fueron los siguientes:

Resultado de latencia para 1 rostro (en segundos)	
Latencia media	0.013854726
Desviación estándar	0.006164001
Resultado de latencia para 2 rostros (en segundos)	
Latencia media	0.017991275
Desviación estándar	0.005947033

Debido a que el *pipeline* se ejecutó a 30 fotogramas por cada segundo, el intervalo máximo debería ser de 0.033 segundos para un cuadro o fotograma por *buffer*. En ambos casos, en promedio se supera con amplio margen tal intervalo. En las figuras, se observan algunos picos. Estos se dan cada 10 cuadros y son originados por las fases de detección. Ello refleja que tan costoso puede ser detectar en contraposición a rastrear. De haber

usado solo detección de rostros, la latencia hubiera estado muy cercana al intervalo de los 0.033 segundos.

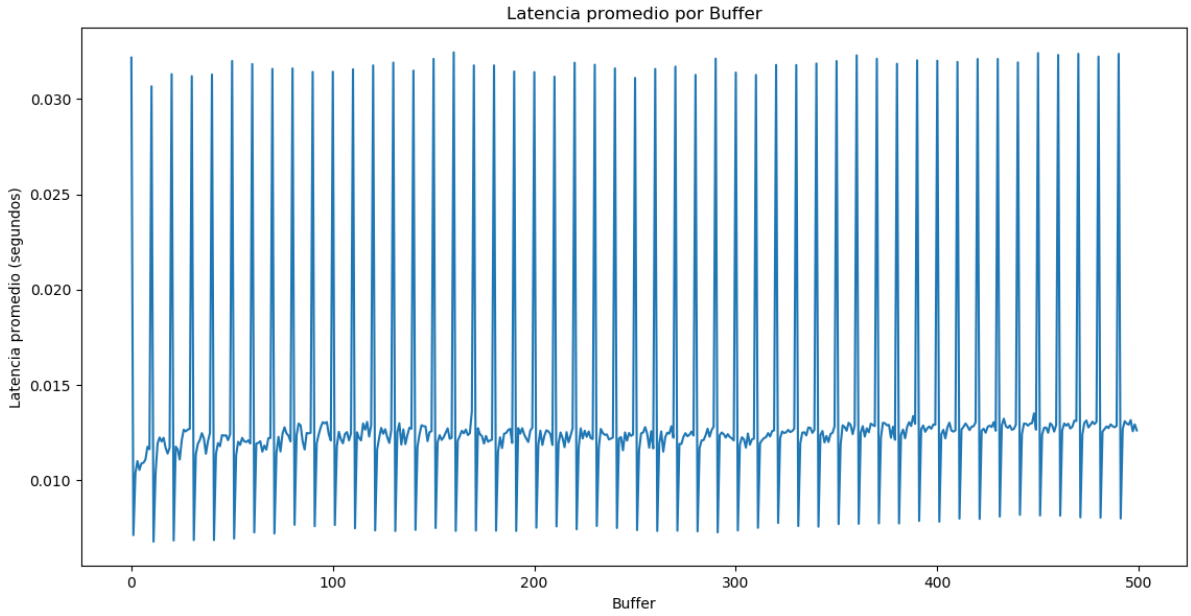


Figura 4.7: Gráfico de latencia promedio por *buffer* para 500 *buffers* repetido 30 veces. Cada *buffer* contiene 1 cuadro. Un rostro posó frente a la cámara web.

Fuente: Autoría propia

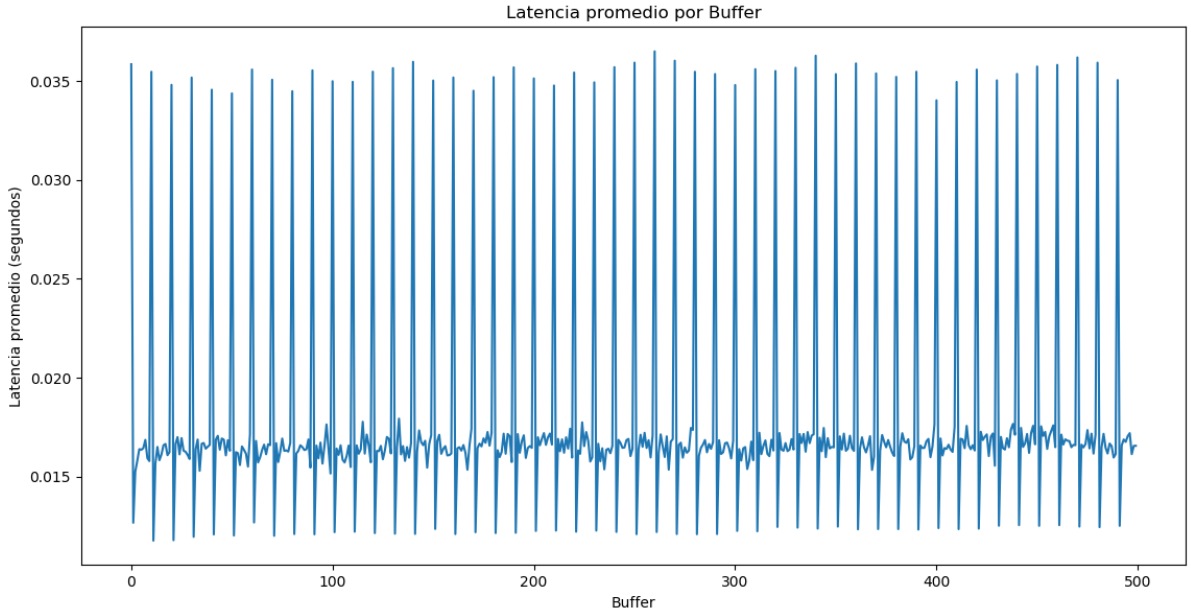


Figura 4.8: Gráfico de latencia promedio por *buffer* para 500 *buffers* repetido 30 veces. Cada *buffer* contiene 1 cuadro. Dos rostros posaron frente a la cámara web.

Fuente: Autoría propia

Capítulo 5

Filtro de sobreposición de imágenes: *gstcheesefaceoverlay*

Este capítulo desarrolla y explica cómo se construyó el filtro de sobreposición de imágenes, los formatos que soporta y cómo funciona. En términos sencillos, para el desarrollo de este filtro se ha definido una estructura en un archivo JSON que representa una animación. Este formato *JSON* le dice al filtro en qué punto facial colocar cierta imagen y con qué duración. Si bien el filtro ha sido basado, en parte, en *gstfaceoverlay* de Laura Lucas Alday, gran parte del código ha sido cambiado incluyendo su estructura interna. Finalmente, se ha elaborado pruebas unitarias de partes importantes del filtro las cuales tienen una cobertura de más del 85 %.

5.1. Explicación del nombre *gstcheesefaceoverlay*

Al filtro desarrollado se le ha asignado el nombre de *gstcheesefaceoverlay* y **en adelante se hará referencia a este filtro por este nombre**. La asignación del nombre se explica a continuación: como la mayoría de elementos de *GStreamer* tiene en su nombre el prefijo *gst* y además se le pone el prefijo *cheese* debido al programa al cual se aplicará. Luego, lleva el término *face* debido a que está ubicado en el directorio *gst-plugins-cheese/gst/face* lugar donde se ubica filtros que usan datos de lo rostros. Finalmente, *overlay* se debe a que sobrepone imágenes en los rostros. **Nótese que *gstcheesefaceoverlay* es un filtro distinto a *gstfaceoverlay*, siendo este último desarrollado por Laura Lucas Alday**.

5.2. Diseño

El filtro de *gstfaceoverlay* consistía en un *GstBin* que contenía un filtro *gstfacedetect*, un filtro *videoconvert* y un filtro *rsvgoverlay* (véase figura 5.1. A diferencia de este filtro, *gstcheesefaceoverlay* consiste en un *GstBin* que contiene un único elemento: *gstcairooverlay* (véase la figura 5.2). La razón por la cual se usa un *bin* en vez de simplemente heredar de la clase *GstCairoOverlay* es que *GStreamer* no expone las cabeceras (archivos *.h*) de *gstcairooverlay*, y por tanto no es posible la herencia.

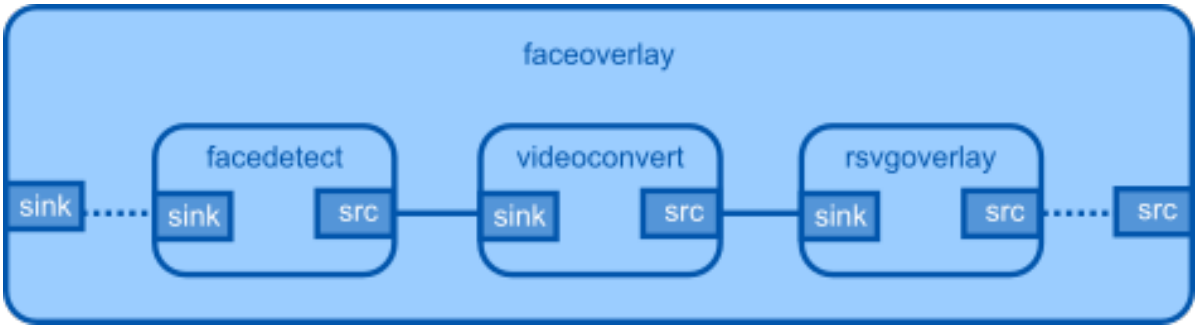


Figura 5.1: Simplificación de diagrama del elemento *gstfaceoverlay*.

Fuente: Autoría propia

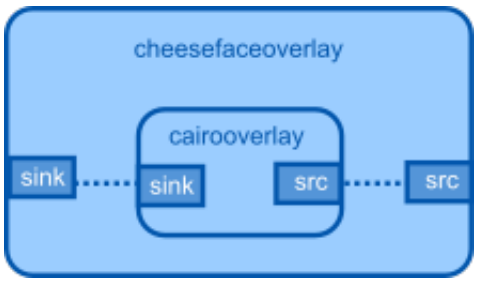


Figura 5.2: Simplificación de diagrama del elemento *gstcheesefaceoverlay*.

Fuente: Autoría propia

5.3. Capabilities

Debido a que este elemento es en términos prácticos una herencia de (en realidad un *bin* que contiene a *gstcairooverlay*), este dispone de un único *sink pad* y único *src pad* los cuales están disponibles siempre. Además tanto los formatos de imagen que acepta y que genera como salida son *BGRx*, *BGRA*, *RGB16*. La descripción completa respecto a las plantillas de sus *pad* mostrada según el comando `gst-inspect-1.0 cheesefaceoverlay` se muestra a continuación:

```
Pad Templates:
  SINK template: 'sink'
    Availability: Always
    Capabilities:
      video/x-raw
                 format: { (string)BGRx, (string)BGRA, (string)RGB16 }
                  width: [ 1, 2147483647 ]
                 height: [ 1, 2147483647 ]
              framerate: [ 0/1, 2147483647/1 ]

  SRC template: 'src'
    Availability: Always
    Capabilities:
      video/x-raw
                 format: { (string)BGRx, (string)BGRA, (string)RGB16 }
                  width: [ 1, 2147483647 ]
                 height: [ 1, 2147483647 ]
              framerate: [ 0/1, 2147483647/1 ]
```

5.4. Propiedades

Las propiedades de este objeto es una sola:

Propiedad	**Tipo**	**Descriptión de la propiedad**
location	*string*	Es la ubicación (ruta) del archivo en formato *JSON* que representa un *sprite*
data	*string*	Es una cadena de carácteres que sigue la estructura de un *sprite*. Puede ser visto como el contenido del archivo *JSON*.

5.5. Sprite

Con el fin de generar una animación a partir de imágenes estáticas, se ha ideado una clase que además es representada en una estructura en formato *JSON*. Esta estructura sirve para que *gstcheesefaceoverlay* pueda saber en que coordenadas en píxeles, en qué escala, durantes cuántos cuadros y para qué personas colocar las imágenes. Ello permite

crear animaciones complejas y buenas visualmente. Este archivo *JSON* es deserializado por el filtro en un objeto llamado *CheeseMultifaceSprite*, el cual a su vez contiene otros objetos, los cuales se describen a continuación:

5.5.1. CheeseFaceSpriteFrame

Representa una imagen estática (la cual se almacena en un *buffer* de píxeles) con una duración. A continuación se lista las propiedades del objeto que están representadas también en formato *JSON*:

Propiedad	**Tipo**	**Descriptión de la propiedad**
duration	*unsigned int*	Es el número de frames durante el cual la imagen será mostrada.
location	*string*	La ruta del archivo de la imagen.
base-scale-factor	*string*	El factor por el cual se escalará la imagen. *gstcheese-faceoverlay* escala las imágenes con dos factores: el primer factor es respecto al tamaño del *bounding box* del rostro y el segundo es este factor.

En *JSON*, esta estructura podría ser representada de la siguiente manera:

```
{
    "base-scale-factor": 1.2,
    "duration": 80,
    "location": "/home/cfoch/Pictures/sprite/head/3.png"
}
```

Lo cual significa que la imagen en la ubicación */home/cfoch/Pictures/sprite/head/3.png* será renderizada desde solo por una duración de 80 frames. La escala de esta imagen se aplica primero escalando al mismo tamaño del *bounding box* y luego multiplicando por el factor de escala base.

5.5.2. CheeseFaceSpriteKeypoint

Representa una animación para un punto específico *key point* de un rostro. Es un conjunto de *CheeseFaceSpriteFrame*. Lo que significa que una vez terminada la duración de un *CheeseFaceSpriteFrame* se muestra la imagen del siguiente *CheeseFaceSpriteFrame*. Así sucesivamente, hasta que la duración total se agote. La animación puede repetirse indefinidamente si así se indica.

Propiedad	**Tipo**	**Descriptión de la propiedad**
keypoint	*CheeseFaceKeypoint*	Es el tipo de punto facial del cual este objeto representará la animación.
rotate	*boolean*	Si es verdadero, indica que la imagen debe rotar en la orientación del ángulo formado por los ojos. Si es falso, la imagen no debe rotarse.
loop	*boolean*	Si es verdadero, una vez culminada la duración se vuelve a mostrar la imagen del primer *CheeseFaceSpritFrame* en la lista y se continúa de esta manera indefinidamente. Si es falso, una vez culminada la duración total, no se mostrará una sola imagen más.
frames	*Lista de CheeseFaceSpriteFrame*	Contiene objetos del tipo CheeseFaceSpriteFrame.

En *JSON*, esta estructura podría ser representada de la siguiente manera:

```
"head": {
    "rotate": true,
    "loop": true,
    "frames": [
        {
            "base-scale-factor": 1.2,
            "duration": 80,
            "location": "/home/cfoch/Pictures/sprite/head/3.png"
        },
        {
            "base-scale-factor": 1.2,
            "duration": 20,
            "location": "/home/cfoch/Pictures/sprite/head/2.png"
        }
```

```
    ]
}
```

Lo cual significa que en la posición de la cabeza, desde que un rostro se detecta, los primeros 80 cuadros, se mostrará la imagen en */home/cfoch/Pictures/sprite/head/3.png* y los siguientes 20 cuadros se mostrará la imagen en */home/cfoch/Pictures/sprite/head/2.png*. Si la cara rota frente a la cámara, la imagen rotará en su misma orientación (según el ángulo formado por sus ojos).

Los puntos faciales disponibles se describen en la siguiente tabla:

En *C*	**Representación en *JSON***	**Descripción**
CHEESE_FACE_KEYPOINT_PHILTRUM	*philtrum*	Representa el filtrum (zona de la cara entre nariz y labio superior).
CHEESE_FACE_KEYPOINT_MOUTH	*mouth*	Representa el centro de la boca.
CHEESE_FACE_KEYPOINT_LEFT_EYE	*left-eye*	Representa el centro del ojo izquiedo.
CHEESE_FACE_KEYPOINT_RIGHT_EYE	*right-eye*	Representa el centro del ojo derecho.
CHEESE_FACE_KEYPOINT_NOSE	*nose*	Representa la posición de la punta de la nariz.
CHEESE_FACE_KEYPOINT_LEFT_EAR	*left-ear*	Representa la zona de la oreja izquierda. Una imagen colocada sobre este punto, tendrá su zona lateral centrada en tal punto.
CHEESE_FACE_KEYPOINT_RIGHT_EAR	*right-ear*	Representa la zona de la oreja derecha. Una imagen colocada sobre este punto, tendrá su zona lateral centrada en tal punto.
CHEESE_FACE_KEYPOINT_FACE	*face*	Representa el centro del rostro.

CHEESE_FACE_KEYPOINT_HEAD	*head*	Representa la zona de la cabeza. Una imagen colocada sobre este punto, tendrá su zona inferior centrada en tal punto.

5.5.3. CheeseFaceSprite

Un objeto de este tipo tiene un conjunto de *CheeseFaceSpriteKeypoint*. Por tanto, este representa el conjunto de animaciones por cada punto facial para un rostro en específico.

En *JSON*, esta estructura podría ser representada de la siguiente manera:

```
{
    "head": {
        "rotate": true,
        "loop": true,
        "base-scale-factor": 0.5,
        "frames": [
            {
                "base-scale-factor": 1.2,
                "duration": 80,
                "location": "/home/cfoch/Pictures/sprite/head/3.png"
            },
            {
                "base-scale-factor": 1.2,
                "duration": 20,
                "location": "/home/cfoch/Pictures/sprite/head/2.png"
            }
        ]
    },
    "philtrum": {
        "rotate": true,
        "loop": true,
        "base-scale-factor": 0.5,
```

```
        "frames": [
            {
                "base-scale-factor": 0.5,
                "duration": 20,
                "location": "/home/cfoch/Pictures/sprite/philtrum/2.png"
            }
        ]
    }
}
```

Lo cual significa que el rostro tendrá animaciones en la zona de su cabeza y filtrum, según lo explicado en la subsección anterior.

5.5.4. CheeseFaceMultiSprite

Un objeto de este tipo tiene un conjunto de *CheeseFaceSprite*. Por lo que este representa el conjunto de animaciones para múltiples rostros en los puntos faciales especificados. **Este es el formato aceptado por *gstcheesefaceoverlay***.

En *JSON*, esta estructura podría ser representada de la siguiente manera:

```
[
    {
        "philtrum": {
            "rotate": true,
            "loop": true,
            "base-scale-factor": 0.5,
            "frames": [
                {
                    "base-scale-factor": 0.5,
                    "duration": 5,
                    "location": "/home/cfoch/Pictures/sprite/philtrum/2.png"
                }
            ]
        },
        "left-eye": {
            "rotate": true,
            "loop": true,
            "base-scale-factor": 0.5,
            "frames": [
```

```
            {
                "base-scale-factor": 0.5,
                "duration": 5,
                "location": "/home/cfoch/Pictures/sprite/eye/1.png"
            },
            {
                "base-scale-factor": 0.5,
                "duration": 10,
                "location": "/home/cfoch/Pictures/sprite/eye/2.png"
            }
        ]
    },
    "right-eye": {
        "rotate": true,
        "loop": true,
        "base-scale-factor": 0.5,
        "frames": [
            {
                "base-scale-factor": 0.5,
                "duration": 5,
                "location": "/home/cfoch/Pictures/sprite/eye/1.png"
            },
            {
                "base-scale-factor": 0.5,
                "duration": 10,
                "location": "/home/cfoch/Pictures/sprite/eye/2.png"
            }
        ]
    },
    "head": {
        "rotate": true,
        "loop": true,
        "base-scale-factor": 0.5,
        "frames": [
            {
                "base-scale-factor": 1.2,
                "duration": 20,
                "location": "/home/cfoch/Pictures/sprite/head/1.png"
            }
        ]
    }
},
{
    "head": {
```

```
            "rotate": true,
            "loop": true,
            "base-scale-factor": 0.5,
            "frames": [
                {
                    "base-scale-factor": 1.2,
                    "duration": 80,
                    "location": "/home/cfoch/Pictures/sprite/head/3.png"
                },
                {
                    "base-scale-factor": 1.2,
                    "duration": 20,
                    "location": "/home/cfoch/Pictures/sprite/head/2.png"
                }
            ]
        },
        "philtrum": {
            "rotate": true,
            "loop": true,
            "base-scale-factor": 0.5,
            "frames": [
                {
                    "base-scale-factor": 0.5,
                    "duration": 20,
                    "location": "/home/cfoch/Pictures/sprite/philtrum/2.png"
                }
            ]
        }
    }
]
```

Como se puede observar acá se detalla la animación de dos rostros. El primer rostro tiene animaciones asignadas a su filtrum, ojo izquierdo, ojo derecho y cabeza. El segundo rostro tiene animaciones asignadas a solamente su cabeza y philtrum.

5.6. Ejemplos de *pipelines*

Existe una lista muy grande de *pipelines* que se pueden formar con este filtro. Se muestra algunos ejemplos sencillos:

Desde la cámara web:

```
$ gst-launch-1.0 v4l2src ! videoconvert ! \
  cheesefacetrack scale-factor=0.5 \
  landmark=shape_predictor_68_face_landmarks.dat ! \
  videoconvert ! cheesefaceoverlay location=sprite.json ! \
  videoconvert ! xvimagesink
```

Desde un archivo de video:

```
$ gst-launch-1.0 filesrc location=archivo.ogv ! \
  decodebin ! videoconvert ! \
  cheesefacetrack scale-factor=0.5 \
  landmark=shape_predictor_68_face_landmarks.dat ! \
  videoconvert ! cheesefaceoverlay location=sprite.json ! \
  videoconvert ! xvimagesink
```

5.7. Medición latencia por *buffer*

En esta sección se muestra los resultados de la medición de latencia en un *pipeline* que usa *gstcheesefaceoverlay* y *gstcheesefaceoverlay*. Los detalles del hardware y forma de compilación del software, así como las herramientas usadas fueron las mismas descritas en la sección 4.6.

Para el elemento *gstcheesefaceoverlay* se usó una única imagen que se sobrepuso sobre cada punto facial soportado. Esta tenía un tamaño de 298 Bytes, dimensiones de 100x100 píxeles y canal alpha uniforme de 0.5. La imagen fue generada usando *ImageMagick* con el siguiente comando:

```
convert -size 100x100 xc:red \
    -alpha set -channel A -evaluate set 50% red.png
```

5.7.0.1. *Pipeline* probado

El pipeline que se ejecutó fue el siguiente:

```
$ gst-launch-1.0 v4l2src ! videoconvert ! video/x-raw,framerate=30/1 ! \
    cheesefacetrack max-distance-factor=0.15 scale-factor=0.5 \
```

```
landmark=shape.dat display-landmark=true detection-gap-duration=10 ! \
videoconvert ! cheesefaceoverlay location=red.json ! \
fakesink num-buffers=500
```

5.7.0.2. Resultados de la medición de latencia

Se ejecutó la *pipeline* mencionada en la subsección anteriormente para 500 *buffers* (donde cada *buffer* contenía 1 cuadro) por 30 veces (como se observa, a 30 fps). Los cuadros de entrada eran en formato YUV y tenían una resolución de 1280x720.

La primera medición se ejecutó para un rostro frente a la webcam (ver figura 5.4), mientras que en la segunda medición se realizó para dos rostros (ver figura 5.4). La cámara estaba estática y cada rostro realizaba poco movimiento con el fin de estresar el algoritmo. Los resultados fueron los siguientes:

Resultado de latencia para 1 rostro (en segundos)	
Latencia media	0.019675417
Desviación estándar	0.006173316
Resultado de latencia para 2 rostros (en segundos)	
Latencia media	0.026642162
Desviación estándar	0.005889697

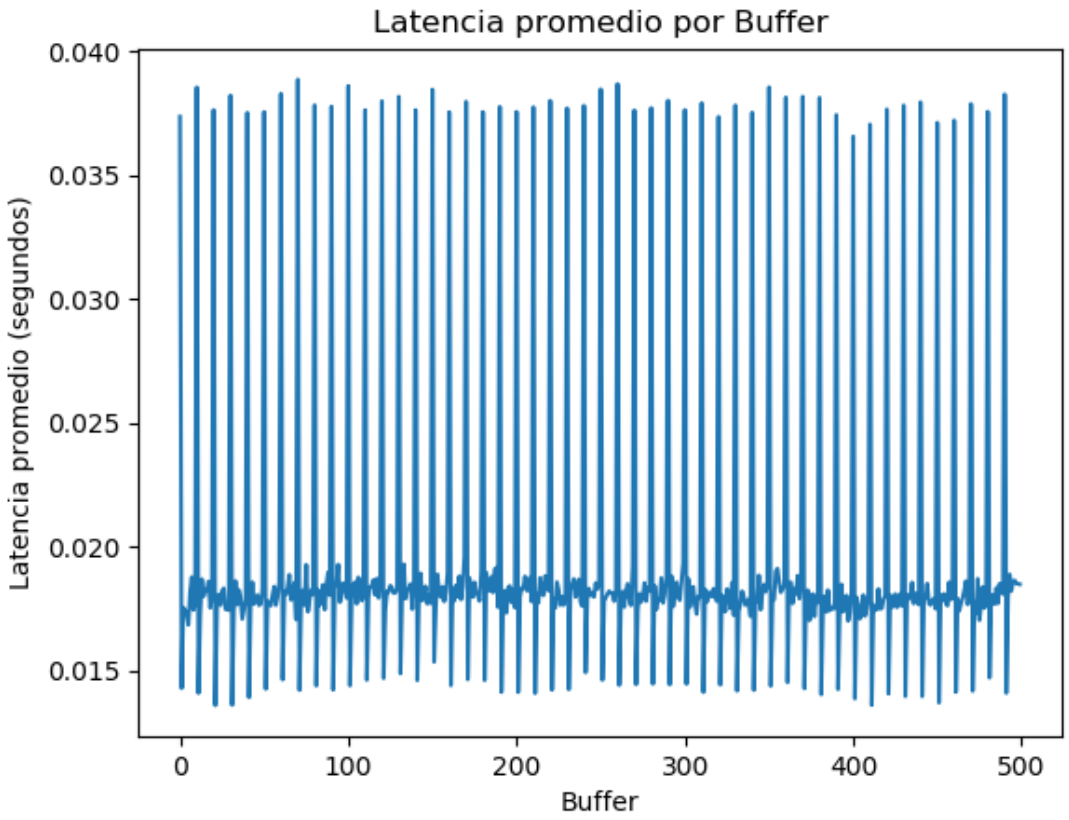


Figura 5.3: Gráfico de latencia promedio por *buffer* para 500 *buffers* repetido 30 veces. Cada *buffer* contiene 1 cuadro. Un rostro posó frente a la cámara web.

Fuente: Autoría propia

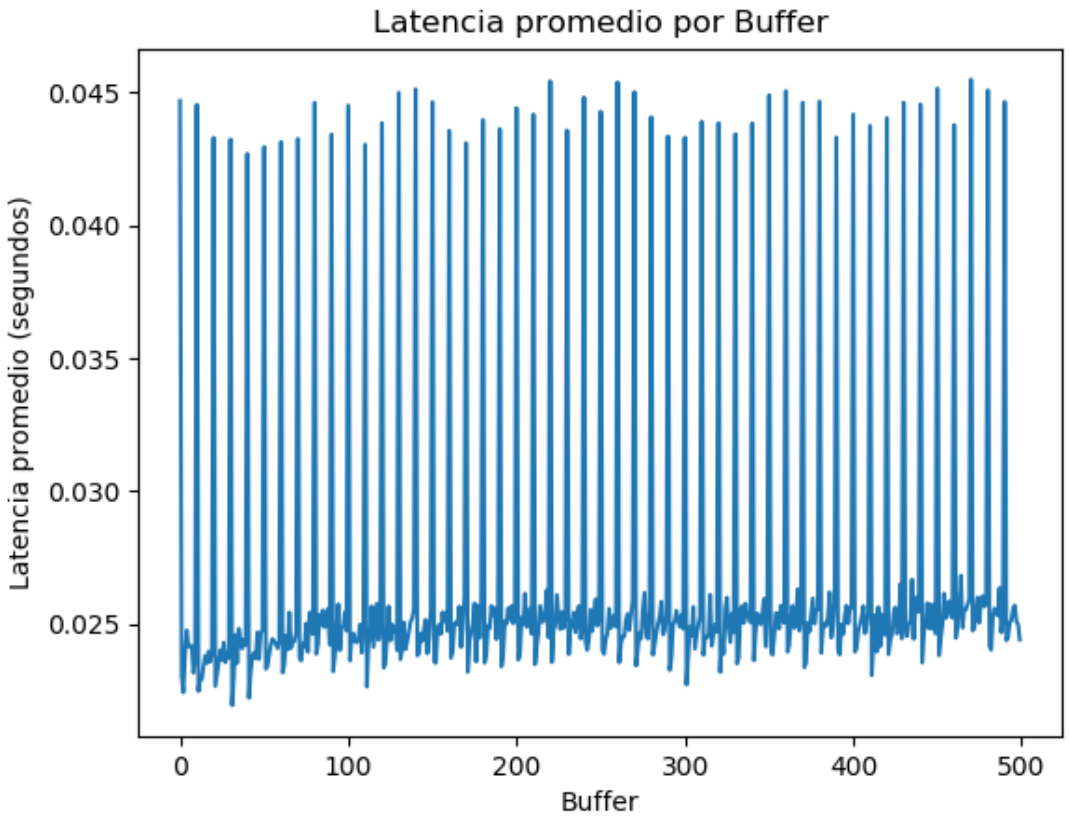


Figura 5.4: Gráfico de latencia promedio por *buffer* para 500 *buffers* repetido 30 veces. Cada *buffer* contiene 1 cuadro. Dos rostros posaron frente a la cámara web.

Fuente: Autoría propia

Capítulo 6

Integración de ambos filtros en Cheese

En este capítulo se desarrolla el tercer objetivo el cual comprende el diseño de una interfaz gráfica para los filtros desarrollados en *Cheese*. Para cumplir este objetivo, el primer paso fue desarrollar un mockup. Luego se diseñó diagramas de clases y secuencias con la finalidad de tener una interfaz gráfica a futuro que no solo permita efectos del tipo a los realizados en *gstcheesefaceoverlay*, sino que se pueda extender a aplicación a tener una librería donde se sobrepongan imágenes estáticas sin el uso de visión artificial o filtros más complicados que permitan cargar modelos en 3D para sobreponerlos de manera similar a como aplicaciones como *Snapchat* y *Facebook* soportan actualmente. Finalmente, ya que *gstcheesefacetrack* y *gstcheesefacedetect* dependen de un modelo entrenado para detectar los puntos faciales del rostro se tuvo que modificar el diáologo de preferencias de *Cheese* para que permita al usuario seleccionar entre el modelo que *dlib* incluye por defecto o cualquier otro. Como se mencionó, *Cheese* está escrito en Vala, y por ello se usó este lenguaje. Además se tuvo que crear una *API* que permite usar el código escrito en *C* (para el objetivo específico 2) en Vala.

6.1. Mock-up

El primer paso fue desarrollar el *mock-up* de la interfaz gráfica. Véase la figura 6.1. En la figura se aprecia un botón sobre la cabecera (*header bar* o área de título) de la ventana principal de *Cheese*. Al hacer clic en tal botón, se debe mostrar un *pop-up* donde se muestre una grilla de imágenes. Cada imagen representa un *sprite* (un *CheeseFaceSprite*). Internamente, *Cheese* debe construir un *CheeseMultifaceSprite* en el orden de la selección. Al hacer clic en el botón con el ícono del visto bueno, el filtro de sobreposición de imágenes debería aplicarse sobre el *viewport* de Cheese.

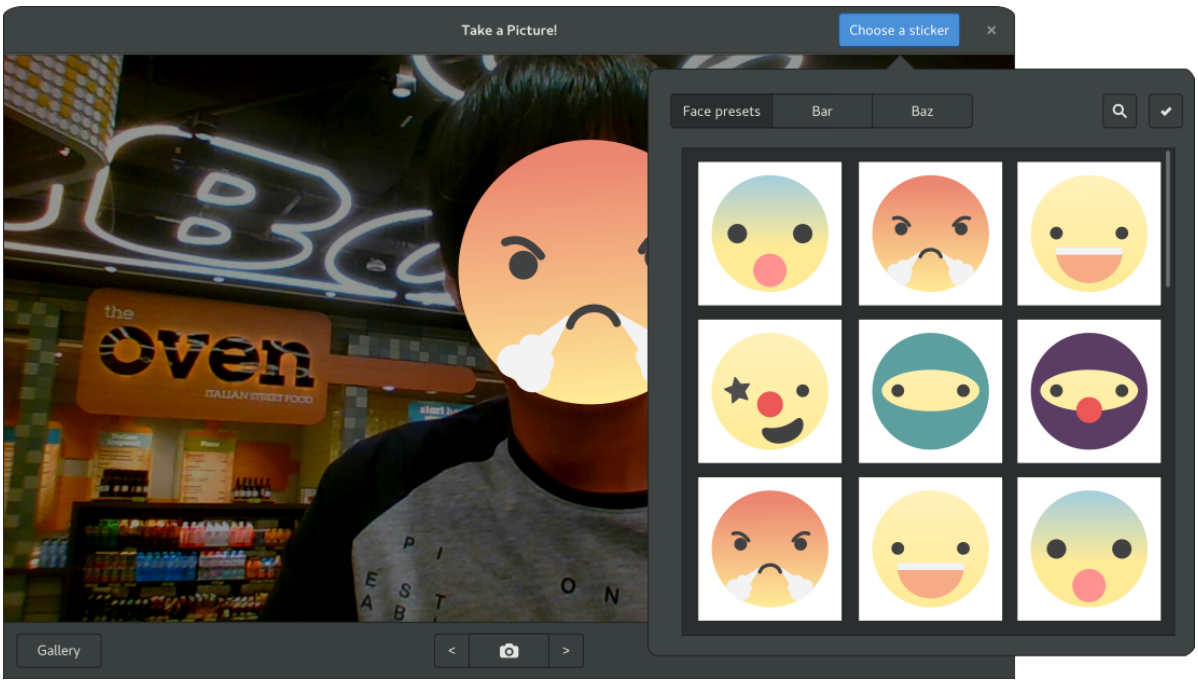


Figura 6.1: Mock-up de interfaz gráfica para *gstcheesefaceoverlay*. Las imágenes de emoticonos que se muestran son solo referenciales. La parte inferior de la imagen fue una idea propuesta por Allan Day, y no se ha desarrollado pues no formaba parte del objetivo.

Fuente: Autoría propia y basado en un *mock-up* desarrollado por Allan Day y otros archivos del repositorio en *git* de *mock-ups* del *GNOME Design Team*. Los emoticonos son parte de Antü Plasma Suite y diseñados por Fabián Alexis.

6.2. Archivos *.sprite* para *face presets*

Según la especificación de *freedesktop.org* (un proyecto creado para la interoperabilidad entre entornos de escritorio para *UNIX* y *Linux*), los archivos de **datos** deben colocarse relativos a un directorio base a la variable de entorno *$XDG_DATA_DIRS* (freedesktop.org, s.f.-b). Para este objetivo, se ha escogido guardar los archivos de extensión *.sprite* en la ruta *$XDG_DATA_DIRS/cheese/sprites/face-presets*.

Los archivos de extensión de *.sprite* en tal ruta representan un *CheeseFaceSprite* y son archivos "key files" que su sintaxis sigue también la especificación *Desktop Entry Specification* de la *freedesktop.org* (freedesktop.org, s.f.-a). Un ejemplo de este archivo puede ser:

```
[Face]
```

```
Name=Face Test Sprite
Location=/usr/share/cheese/stickers/sprites/face-presets/face1.json
Thumbnail=/usr/share/cheese/stickers/sprites/face-presets/thumbnails/face1.png
```

La clave "Location" representa la ubicación del archivo de *sprite .json* aceptado por *gstcheesefaceoverlay*. Cuando *Cheese* lea este archivo usará el valor en "Location" para "decirle" a *gstcheesefaceoverlay* que renderice las imágenes a sobreponer según ese archivo. Luego, la clave "Thumbnail" representa la imagen que se muestra como vista previa en el *grid* que se explicará en la siguiente sección.

Cheese puede distribuir su propio conjunto de "sprites" pero ya que se usa el directorio base *$XDG_DATA_DIRS*, los usuarios podrían definir sus propios *sprites* sin tener que modificar el directorio de instalación de *Cheese*.

6.3. Diagrama de clases

El diagrama de clases (figura 6.4) se ha diseñado pensando en tener una interfaz que sea extendible a futuro. Cada clase que empieza con *Cheese* significa que está dentro de tal *namespace*.

CheeseStickerGrid es una clase heredada del *widget Gtk.FlowBox* que representa un contenedor (en formal de grilla) de botones (el atributo *button* es solo representativo) del tipo *CheeseStickerButton*. Además dispone de un atributo *filter_func* el cual representa la función de búsqueda en caso el *widget* soporte la búsqueda. Luego se tiene un método *load_buttons* el cual las clases que hereden de esta deberán sobreescribir. Por ejemplo, en este proyecto, como se ve en la figura 6.1 los botones a cargar serían los archivos *.sprite* que están localizados en *$XDG_DATA_DIRS/cheese/sprites/face-presets*. Si tuviéramos archivos similares a los *.sprite* que describan filtros en 3D, entonces bastaría con crear otra clase que herede de esta y que cargue los archivos necesarios.

CheeseStickerButton es una clase que hereda del *widget GtkToggleButton* al cual se le aplica un estilo CSS para que el botón se vea grande y de forma cuadrada. Esta clase solo representa un botón con una imagen. Al heredar *CheeseFacePresetButton* de *CheeseStickerButton* este botón puede guardar información del archivo *sprite* en el atributo *face_sprite_json*. La manera de crear un botón *CheeseFacePresetButton* es mediante una llamada al método estático *new_from_keyfile*.

CheeseFacePresetGrid como se mencionó, hereda de *CheeseStickerGrid*. Este además

Figura 6.2: Representación de *CheeseFacePresetButton*

Fuente: el emoticón es parte de Antü Plasma Suite y ha sido diseñado por Fabián Alexis.

implementa una interfaz *CheeseStickerPageIFace* y un *CheeseFaceGridInterface*. La primera interfaz es útil para que a partir de unos botones seleccionados se cree un *Cheese.Effect* (el cual representa un efecto o filtro) y además es útil también porque provee señales (signals) que se emiten cada vez que es posible crear un efecto (por ejemplo si se seleccionaron 1 o más botones) o si no es posible (si no hay botón seleccionado) para habilitar o deshabilitar el botón de "visto bueno" como se puede ver en la figura 6.1.

Finalmente, *CheeseStickersPopup* representa el *pop-up* que se muestra cada vez que se hace clic en el botón azul de la barra de título. Este dispone de métodos de retrollamada para realizar la acción de aplicar el efecto o de buscar un efecto (o filtro) cuando se hace clic en el botón del "visto bueno" o cuando se realiza una búsqueda, respectivamente. Un pop-up de este tipo puede contener varios *widgets* que implementen *CheeseFaceGridInterface*.

6.4. Diagrama de secuencias

El diagrama de secuencias (figura 6.5) representa el flujo de la interacción entre el usuario y el *pop-up* agregado en *Cheese*. El flujo inicia cuando el usuario hace clic en el botón "Choose a sticker". Al hacer clic, el bucle principal de *GLib* recibe el evento y se crea y muestra un *pop-up* sobre la ventana de *Cheese*. Al momento de la creación de un *CheeseStickersPopup* se crea además un *CheeseFacePresetsGrid* el cual a su vez crea botones como tantos archivos *sprite* existan en *$XDG_DATA_DIRS/cheese/sprites/face-presets*.

Una vez cargado el *pop-up*, el botón del icono de "visto bueno" está desactivado. Apenas el usuario hace clic en uno los botones en la grilla, una señal *"effect-available"* es emitida de tal modo que activa el botón del icono de "visto bueno". Si se deseleccionan

Figura 6.3: Representación de *CheeseFacePresetGrid*

Fuente: Autoría propia y basado en archivos del repositorio en *git* de *mock-ups* del *GNOME Design Team*. Los emoticonos son parte de Antü Plasma Suite y diseñados por Fabián Alexis.

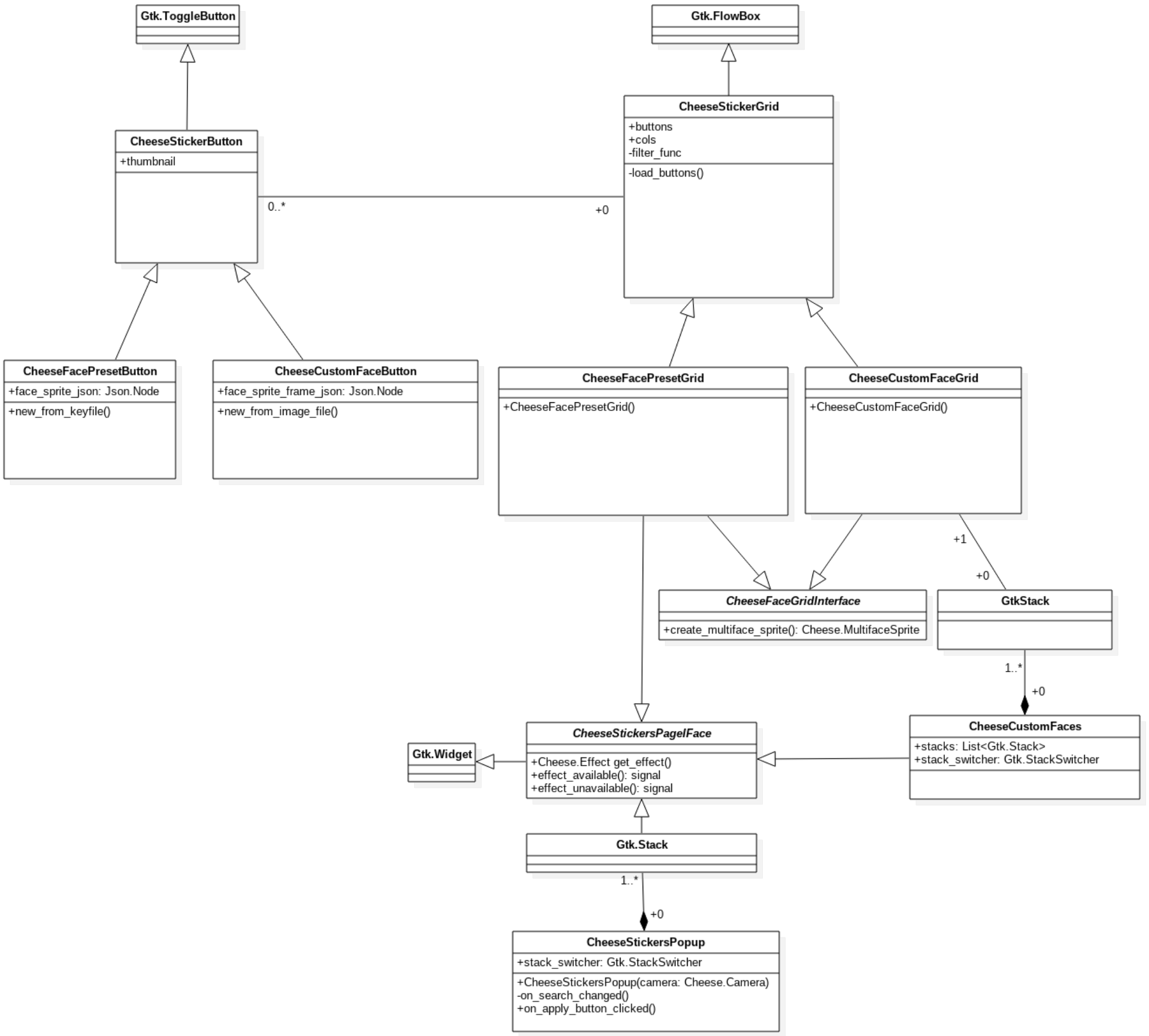


Figura 6.4: Diagrama de clases de interfaz gráfica para *gstcheesefaceoverlay* en *Cheese*. Las clases *CheeseCustomFaceGrid*, *CheeseCustomFaceButton* y *CheeseCustomFaces* no se han implementado, pues está fuera del alcance.

Imagen de autoría propia

todos los botones, se emite la señal *"effect-unavailable"* y el botón es desactivado.

Finalmente, al hacer clic en el botón del "visto bueno", internamente, se construye un *sprite* en un formato aceptado por *gstcheesefaceoverlay* en el orden de la selección. De no haber ningún problema, se aplica el efecto y se muestra en el *viewport* de *Cheese*. De haberlo, se muestra un *GtkInfoBar* que avisa que hubo un error.

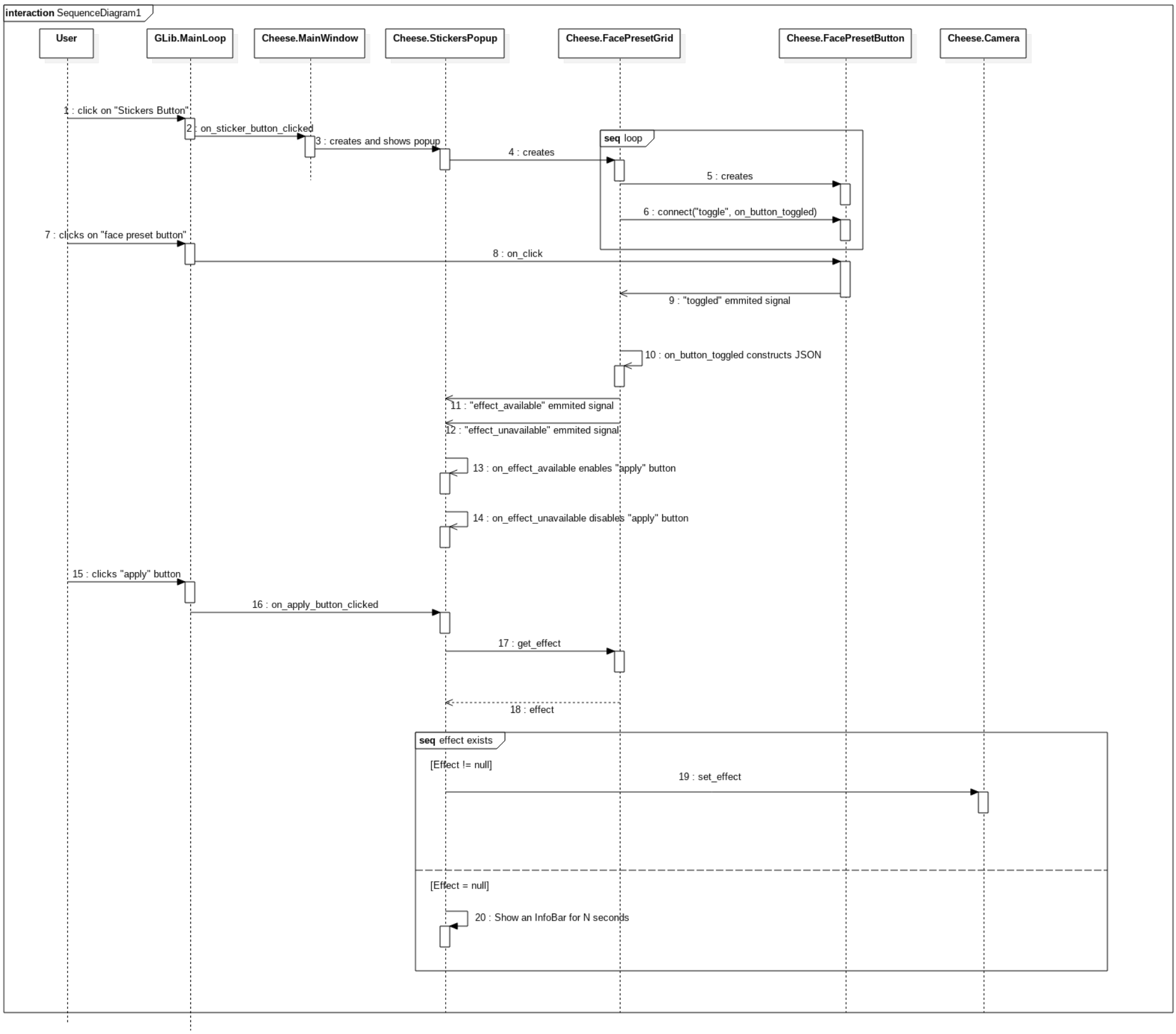


Figura 6.5: Diagrama de secuencias.

Imagen de autoría propia

Capítulo 7

Conclusiones, recomendaciones y trabajos futuros

En este capítulo se detallan las conclusiones del proyecto desarrollado, y se muestran algunas sugerencias e ideas de posibles trabajos a futuro.

7.1. Conclusiones

El trabajo desarrollado representa una alternativa y mejora a la implementación desarrollada durante el *GNOME Outreach Program for Women*. Las conclusiones son las siguientes:

- El filtro desarrollado como parte del primer resultado corresponde una alternativa al filtro *gstfacedetect*. Se observó que la combinación de distintas metodologías pueden crear un rastreador de múltiples rostros con alta precisión. Pero también se observó que la latencia promedio del *pipeline* que usa con filtro *gstcheesefaceoverlay* era mayor cuando dos rostros se colocaban frente a la webcam que cuando uno solo estaba frente a la webcam. Pese a ello, la latencia promedio se mantenía por debajo del intervalo (0.033s, 30fps) soportado por la cámara para la resolución probada. Evidentemente, a medida que más rostros se coloquen frente a la cámara, la latencia tendería a aumentar.
- El filtro desarrollado para el segundo resultado representa no solo a una alternativa sino también una mejora al filtro *gstfaceoverlay*. El filtro desarrollado se apalanca del filtro mencionado en el punto anterior. Este filtro soporta imágenes estáticas en varios formatos a diferencia de *gstfaceoverlay* que solo soporta el formato *SVG*. Además estás imágenes estáticas se pueden colocar secuencialmente con el fin de crear animaciones, característica que no era considerada por el *gstfaceoverlay*. Finalmente, se diseñó un formato en JSON con el fin de facilitar la creación de animaciones.

- La implementación en Cheese del soporte para sobreposición de imágenes a diferencia de la realizada durante el *GNOME Outreach Program for Women* requirió modificar la interfaz gráfica de *Cheese*. La implementación anterior solo agregaba archivos *.effect* con el fin de crear nuevos efectos. El tener una interfaz gráfica dedicada permite al usuario seleccionar una o más secuencias de imágenes que se colocarán sobre los rostros.

7.2. Recomendaciones y trabajos a futuro

- Se recomienda compilar el filtro en modo *release* sin opciones de *debug*. Empíricamente se ha visto que si no se compila de esta manera el desempeño del algoritmo es muy pobre. Es posible incluso que el desempeño mejore compilando los filtros con el conjunto de instrucciones *SSE4.2*, *AVX* o *AVX2*. Sin embargo, esto último no ha sido probado.
- Una recomendación para mejorar en este caso el *pipeline* completo, es reemplazar el filtro *videoconvert* (que realiza conversión de color usando CPU) por uno que realice conversión de color mediante GPU como por ejemplo el filtro *glcolorconvert*. Cheese actualmente realiza conversiones de color usando CPU, y posiblemente usar GPU significaría una mejora.
- Una idea de trabajo a futuro es la mejora del filtro *gstcheesefaceoverlay*. Este carga imágenes al iniciar el *streaming*; sin embargo, se podría hacer uso de un caché de imágenes. Incluso, en vez de usar el filtro *cairooverlay* se podría escribir un nuevo filtro o modificar el filtro desarrollado para componer imágenes usando *OpenGL*. Para desarrollar lo último, se sugeriría ver el código fuente de *gstgloverlay*, el cual se encuentra en los plugins *gst-plugins-base*.
- Otra idea de trabajo a futuro es agregar soporte para cargar modelos 3D. Este trabajo, de hecho, ha sido intentado, pero está aun en progreso. Como primer paso habría que modificarse el filtro *gstcheesefacetrack*. Tal modificación consistiría en resolver el problema *Perspective-n-Point*, el cual consiste en estimar la pose de una cámara previamente calibrada dado un conjunto de coordenadas en 3D en el mundo (en este caso podría ser algunos puntos de un modelo 3D) y dado un conjunto de puntos en 2D que representarían la proyección de tales puntos en 3D. Por tanto, estimando la pose de la cámara se puede estimar la pose del rostro. La estimación de la pose se ha implementado en un filtro que no fue mencionado en este proyecto llamado *gstcheesefacedetect*. Sin embargo, parece que la pose no se estima muy bien, al parecer porque faltó calibrar la cámara web. Una vez ese problema esté resuelto el siguiente paso sería crear un filtro nuevo que cargue modelos en algún formato de geometría como *OBJ*. Ello también se intentó usando un cargador de modelos en 3D de código abierto llamado *assimp*. Sin embargo, algunos modelos específicos daban problemas, y se dejó inconcluso el proyecto.

Capítulo 8

Anexos

8.0.0.1. Gráficas de precisión, exhaustividad y valor-F por cuadro y por rostro para uno de los videos probados

Según el dataset usado en la ruta *Multiple_faces/Camera/multiple_camera1.mp4*, se obtuvieron las siguientes gráficas por rostro:

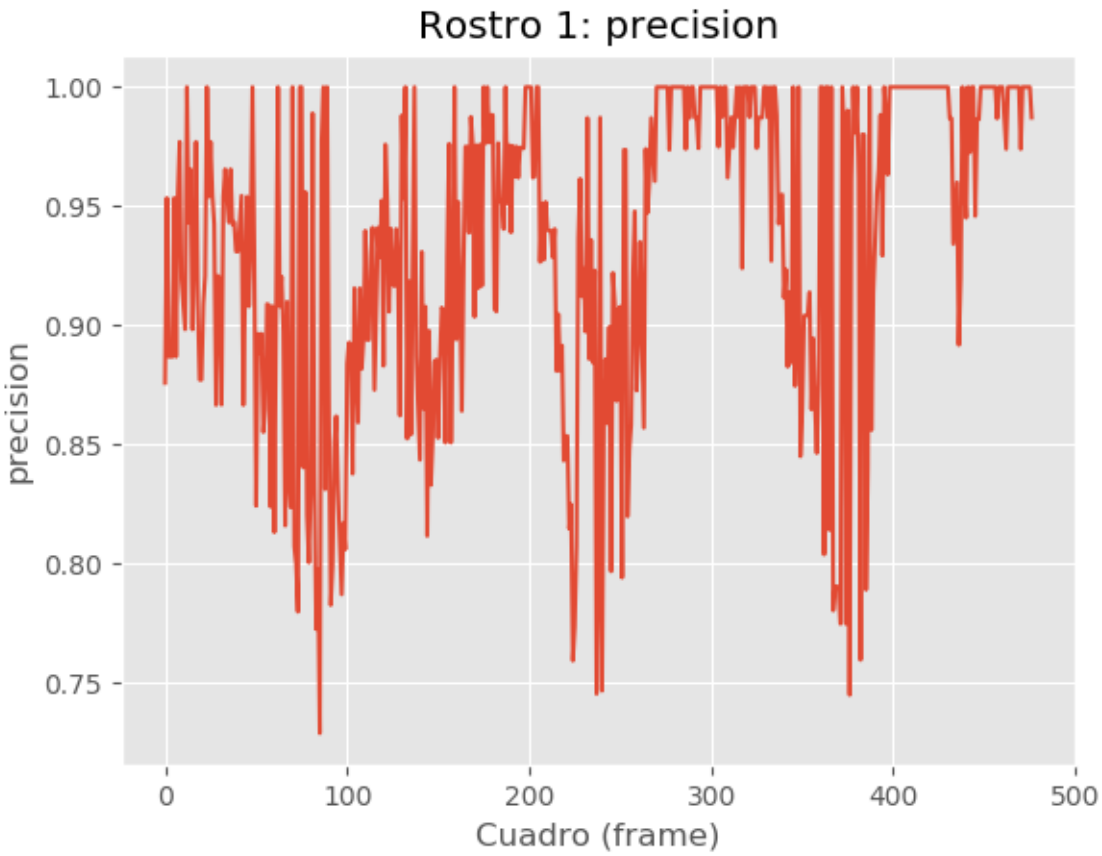


Figura 8.1: Precisión por cuadro para el primer rostro.

Fuente: Autoría propia

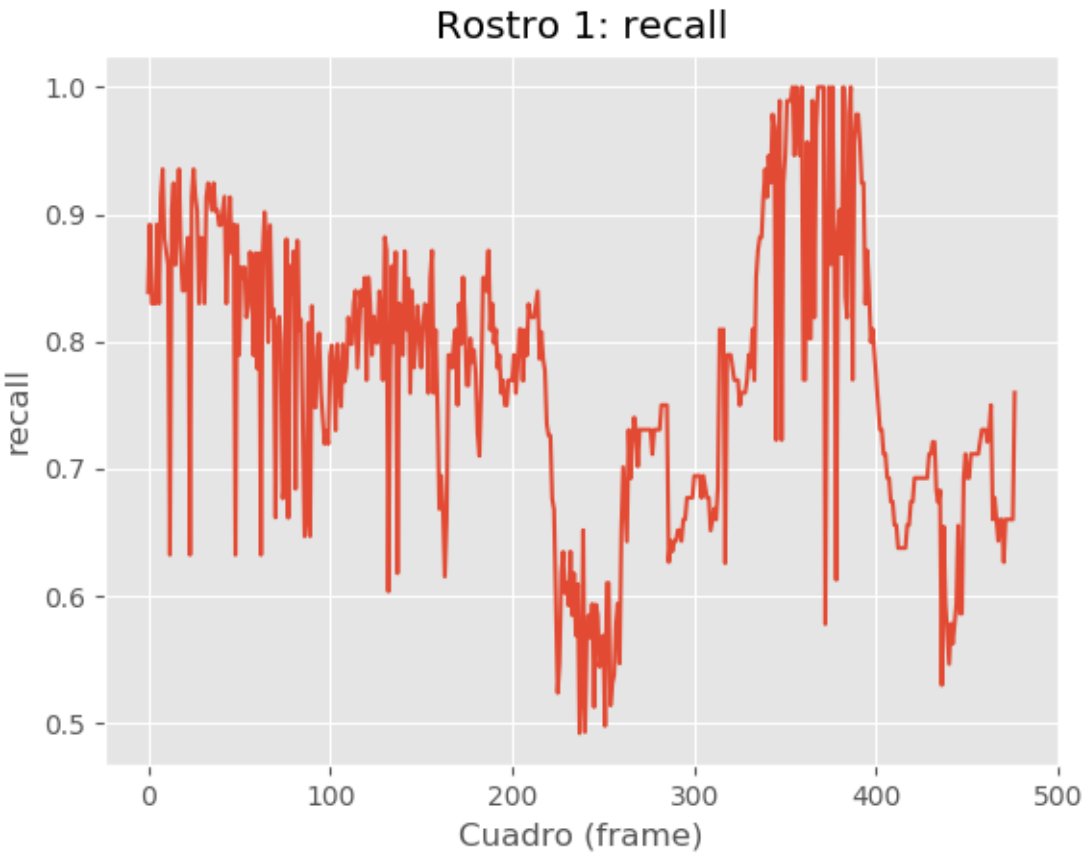


Figura 8.2: Exhaustividad por cuadro para el primer rostro.

Fuente: Autoría propia

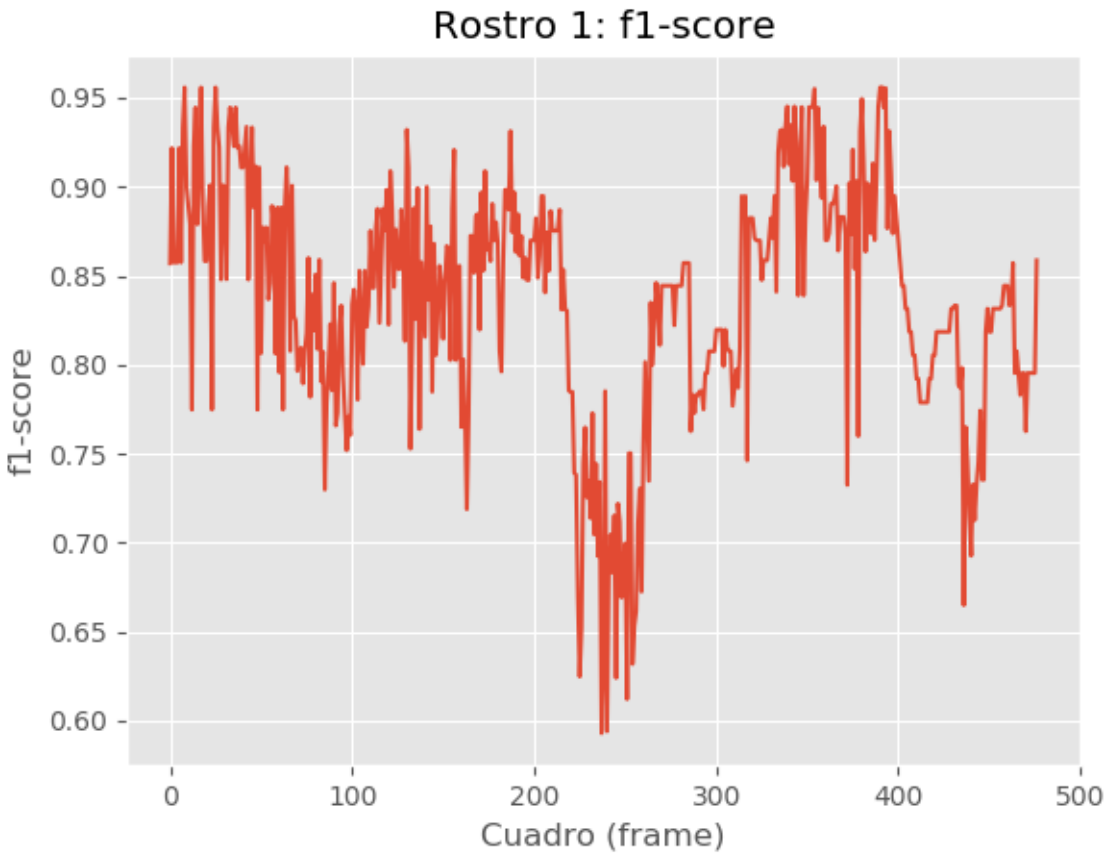


Figura 8.3: Valor-F por cuadro para el primer rostro.

Fuente: Autoría propia

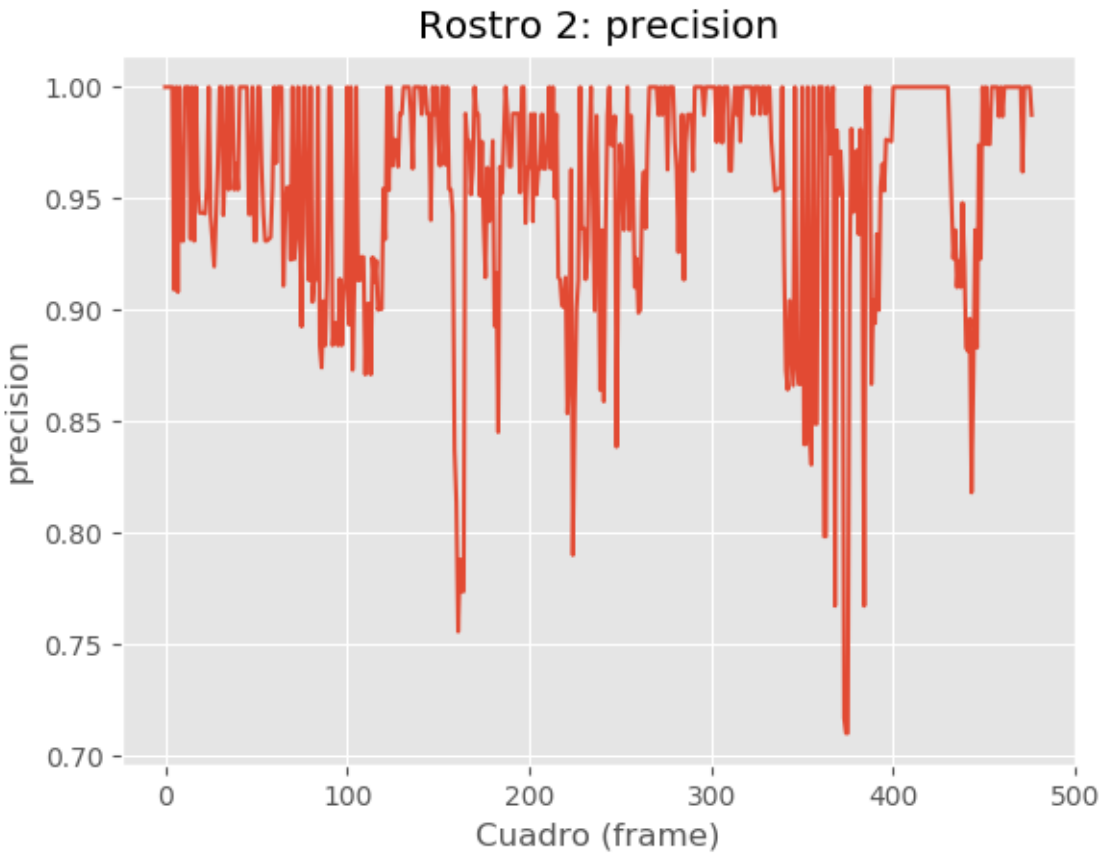


Figura 8.4: Precisión por cuadro para el segundo rostro.

Fuente: Autoría propia

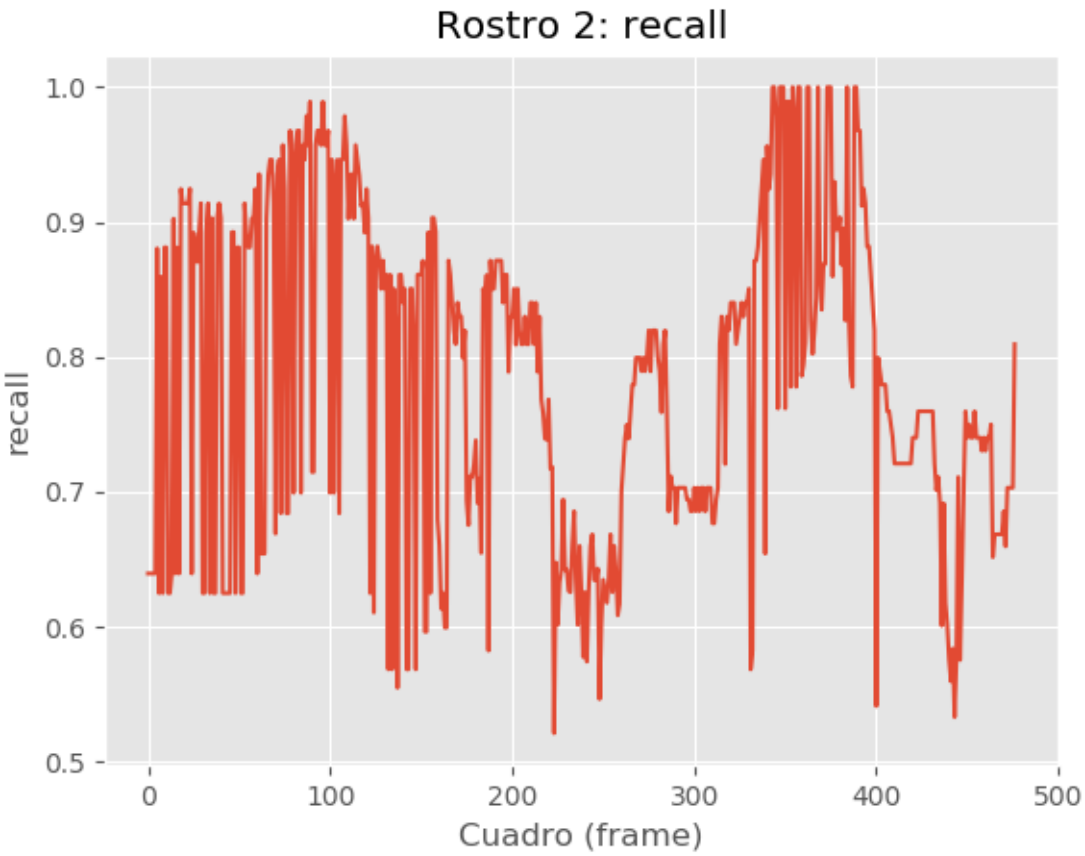


Figura 8.5: Exhaustividad por cuadro para el segundo rostro.

Fuente: Autoría propia

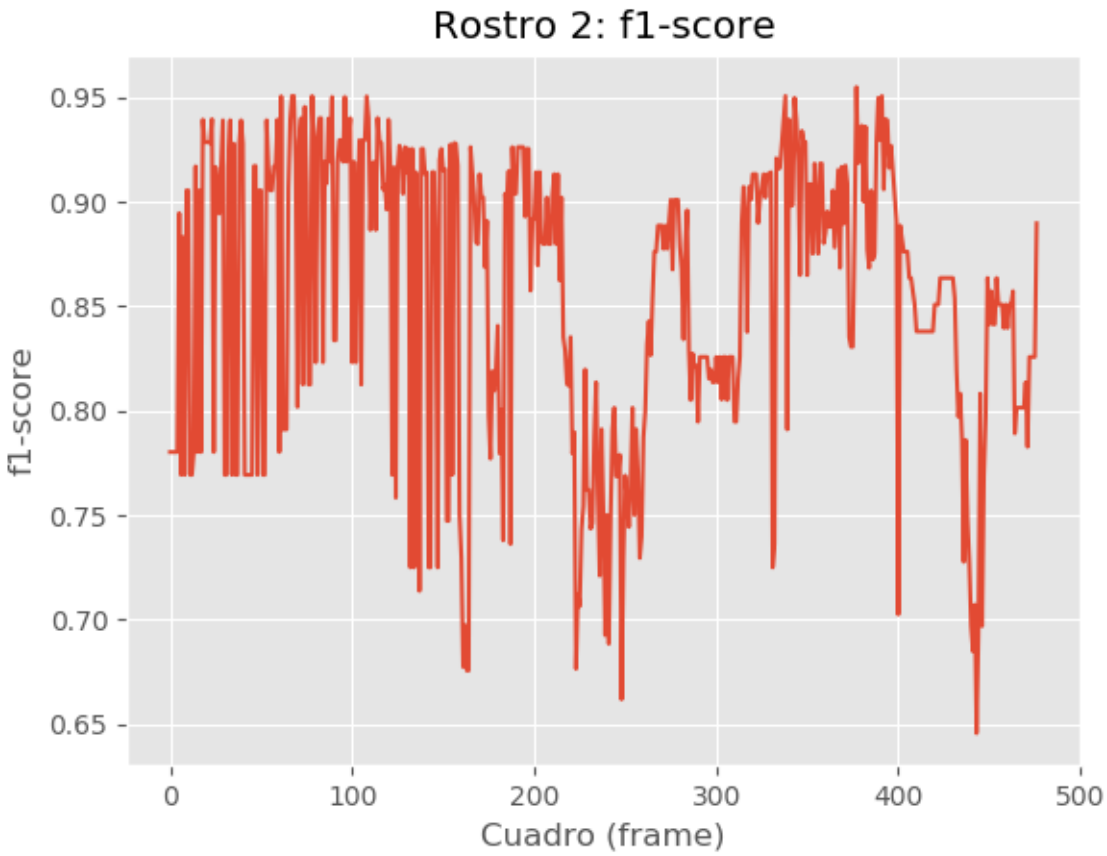


Figura 8.6: Valor-F por cuadro para el segundo rostro.

Fuente: Autoría propia

Glosario

GObject Una librería de GNOME escrita en C que implementa un sistema orientado a objetos. Una amplia variedad de clases heredan de GObject.. 28, 32, 36

gstcairooverlay Un filtro de GStreamer que permite dibujar sobre cada cuadro del stream usando la librería cairo.. 25, 66

gstfacedetect Un filtro de GStreamer para la detección de rostros usando internamente el detector de objetos de Viola Jones de OpenCV.. 13, 17, 24, 66, 87

gstopencvvideofilter Un filtro de GStreamer escrito con el propósito de ser heredado por otros filtros como por ejemplo gstfacedetect. Este filtro recibe una imagen en un buffer y lo encapsula en un tipo de dato reconocido por OpenCV.. 54

gstvideotestsrc Un elemento de GStreamer tipo *source* que produce datos de prueba. Las imágenes producidas corresponden a un patrón de señal de ajuste ampliamente usado para pruebas de video en televisión.. 32

GtkToggleButton Un widget de Gtk+ que dibuja un botón de dos estados que se comporta como un interruptor.. 81

landmark Es un punto significativo de un objeto u organismo que es comun a su clase o especie.. 29, 41, 53

signals GObject tiene un mecanismo basado en señales. Cada vez que una señal es emitida una función de retrollamada se activa. Por ejemplo, GtkButton emite una señal “clicked” cada vez que el usuario hace clic sobre el botón. Las señales de GObject no tienen relación alguna con las señales de UNIX.. 82

Referencias

Alday, L. L. (2011). *Gstreamer faceoverlay plugin.* https://cgit.freedesktop.org/gstreamer/gst-plugins-bad/tree/gst/faceoverlay/gstfaceoverlay.c?id=ee021c04187f17562c9e8b5693fb3b4765648ca5.

Apple Inc. (s.f.). *Photo booth help.* Autor Descargado de https://support.apple.com/guide/photo-booth/welcome ((Accessed on 03/13/2018))

Bakhshi, S., Shamma, D. A., Kennedy, L., y Gilbert, E. (2015). Why we filter our photos and how it impacts engagement. En *Icwsm* (pp. 12–21).

Bassi, E. (s.f.). *Clutter reference manual.* https://developer.gnome.org/clutter/stable/clutter-overview.html.

Boulton, R., Walthinsen, E., y Baker, S. (2017). *Gstreamer 1.10 plugin writer's guide.* Samurai Media Limited. Descargado de https://books.google.com.pe/books?id=P7vZAQAACAAJ

Byfield, B. (2005, Sep). Google's summer of code concludes. *Linux.com | The source for Linux information*. Descargado de https://www.linux.com/news/googles-summer-code-concludes

Cao, Y. (2014a). *delphifirst/facex · github.* https://github.com/delphifirst/FaceX/blob/master/LICENSE.

Cao, Y. (2014b). *Facex/license at master · delphifirst/facex · github.* https://github.com/delphifirst/FaceX/blob/master/LICENSE.

cheese - take photos and videos with your webcam, with fun graphical effects. (2018). https://git.gnome.org/browse/cheese/tree/configure.ac#n72. ((Accessed on 03/13/2018))

CIPA. (2008, mar). *Total shipments of film cameras(classified by model).* CIPA. Descargado de http://www.cipa.jp/stats/documents/common/cr400.pdf

CyberLink. (s.f.). *Youcam 7 - app para camara y webcam | cyberlink.* https://es.cyberlink.com/products/youcam/features_es_ES.html. ((Accessed on 03/13/2018))

Dalal, N., y Triggs, B. (2005). Histograms of oriented gradients for human detection. En *Computer vision and pattern recognition, 2005. cvpr 2005. ieee computer society conference on* (Vol. 1, pp. 886–893).

Davis King, G. (2009). *dlib c++ library / wiki / known_ users.* https://sourceforge.net/p/dclib/wiki/Known_users/.

dlib c++ library. (s.f.). http://dlib.net/.

dlib c++ library - license. (s.f.). http://dlib.net/license.html.

Do, L. N., Yang, H. J., Kim, S. H., Lee, G. S., Na, I. S., y Kim, S. H. (2014). Construction of a video dataset for face tracking benchmarking using a ground truth generation tool. *International Journal of Contents*, *10*(1), 1–11.

Edwards, E., y Hart, J. (2004). *Photographs objects histories: On the materiality of images*. Taylor & Francis. Descargado de https://books.google.com.pe/books?id=TIeBAgAAQBAJ

Eleftheriadis, A., y Jacquin, A. (1995). Automatic face location detection and tracking for model-assisted coding of video teleconferencing sequences at low bit-rates. *Signal Processing: Image Communication*, *7*(3), 231–248.

Free Software Foundation. (s.f.-a). *Categories of free and nonfree software.* https://www.gnu.org/philosophy/categories.en.html.

Free Software Foundation. (s.f.-b). *Proprietary software is often malware.* https://www.gnu.org/proprietary/proprietary.html.

Free Software Foundation. (s.f.-c). *Various licenses and comments about them - gnu project - free software foundation.* https://www.gnu.org/licenses/license-list.en.html. ((Accessed on 09/13/2017))

Free Software Foundation. (s.f.-d). *What is free software? - gnu project - free software foundation.* https://www.gnu.org/philosophy/free-sw.en.html.

Free Software Foundation. (s.f.-e). *¿qué es el software libre? - proyecto gnu - free software foundation.* https://www.gnu.org/philosophy/free-sw.es.html.

freedesktop.org. (s.f.-a). *Desktop entry specification.* https://specifications.freedesktop.org/desktop-entry-spec/latest/. ((Accessed on 04/15/2018))

freedesktop.org. (s.f.-b). *Xdg base directory specification.* https://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html. ((Accessed on 04/15/2018))

GNOME Foundation. (s.f.). *The gnome foundation – gnome.* https://www.gnome.org/foundation/.

Gnome project announces outreach program for women interns. (2010). *GNOME Foundation*. Descargado de https://www.gnome.org/press/2010/11/gnome-project-announces-outreach-program-for-women-interns/

GStreamer Team. (s.f.). *Gstreamer: features.* https://gstreamer.freedesktop.org/features/index.html.

Hannavy, J. (2013). *Encyclopedia of nineteenth-century photography*. Taylor & Francis. Descargado de https://books.google.com.pe/books?id=Kd5cAgAAQBAJ

Huber, P. (2015a). *4dface.* https://github.com/patrikhuber/4dface.

Huber, P. (2015b). *4dface/license at master · patrikhuber/4dface · github.* https://github.com/patrikhuber/4dface/blob/master/LICENSE.

Kalal, Z., Mikolajczyk, K., y Matas, J. (2010). Forward-backward error: Automatic detection of tracking failures. En *Pattern recognition (icpr), 2010 20th international conference on* (pp. 2756–2759).

King, D. (2008). *dlib/license.txt at master · davisking/dlib · github.* https://github.com/davisking/dlib/blob/master/dlib/LICENSE.txt.

King, D. (2014a). *dlib c++ library: Dlib 18.6 released: Make your own object de-*

tector! http://blog.dlib.net/2014/02/dlib-186-released-make-your-own-object.html.

King, D. (2014b). *dlib c++ library: Real-time face pose estimation.* http://blog.dlib.net/2014/08/real-time-face-pose-estimation.html.

LAB, S. (s.f.). *Chonnam university : Smart computing lab.* http://sclab.cafe24.com/publications.jsp. ((Accessed on 05/19/2018))

Lucas, B. D., Kanade, T., y cols. (1981). An iterative image registration technique with an application to stereo vision.

Lucas Alday, L. (2016). *Bug 664140 - add new effect: Svg image overlay.* https://bugzilla.gnome.org/show_bug.cgi?id=664140.

Magazine, L. (2017, 10). *What's taking wayland so long? » linux magazine.* http://www.linux-magazine.com/Online/Features/What-s-Taking-Wayland-So-Long/(language)/eng-US. ((Accessed on 03/13/2018))

ManyCam. (s.f.). *Webcam software for windows and mac features |manycam.* https://manycam.com/features/.

Mathworks. (s.f.). *Integral image - matlab & simulink - mathworks america latina.* https://la.mathworks.com/help/images/integral-image.html. ((Accessed on 04/14/2018))

OpenCV team. (s.f.-a). *About - http://opencv.org/.* http://opencv.org.

OpenCV team. (s.f.-b). *About - opencv library.* http://opencv.org/about.html.

Orccón Chipana, C. F. (2016a). *Bug 764011 - faceoverlay: port to gstreamer 1.x.* https://bugzilla.gnome.org/show_bug.cgi?id=764011.

Orccón Chipana, C. F. (2016b). *Bug 769176 - gstfaceoverlay: Detect multiple faces and allow to overlay multiple images with different types.* https://bugzilla.gnome.org/show_bug.cgi?id=769176.

Orccón Chipana, C. F. (2016c). *Funny stickers in cheese.* https://cfoch.github.io/2016/03/24/funny-stickers-in-cheese.html.

The original snapchat | elise tries | npr - youtube. (2017). https://www.youtube.com/watch?v=kUeQcjM5Zuc.

Pakkanen, J. (s.f.). *A simple comparison.* http://mesonbuild.com/Simple-comparison.html#build-times. ((Accessed on 05/25/2018))

Pavlenko, A. (2013). *adding license file and a sample of its reference in ocl.hpp · opencv/opencv@96f8ff2 · github.* https://github.com/opencv/opencv/commit/96f8ff2ab8d37f362f90e10d05e3c6039c51e92b.

Project, T. G. (s.f.). *Projects/vala/bindings - gnome wiki!* https://wiki.gnome.org/Projects/Vala/Bindings. ((Accessed on 06/22/2018))

Rushe, D. (2013). *Skype's secret project chess reportedly helped nsa access customers' data | technology | the guardian.* https://www.theguardian.com/technology/2013/jun/20/skype-nsa-access-user-data. ((Accessed on 11/03/2017))

Shaik, Z., y Asari, V. (2007). A robust method for multiple face tracking using kalman filter. En *Applied imagery pattern recognition workshop, 2007. aipr 2007. 36th ieee* (pp. 125–130).

Siegel, D. G. (2007). *Cheese - take photos and videos with your webcam, with fun graphical effects.* https://git.gnome.org/browse/cheese/tree/COPYING.

Smolyanskiy, N., Huitema, C., Liang, L., y Anderson, S. E. (2014). Real-time 3d face tracking based on active appearance model constrained by depth data. *Image and Vision Computing*, *32*(11), 860–869.

Snap Inc. (2017, mar). *Prospectus.* U.S. Securities and Exchange Commission. Descargado de https://www.sec.gov/Archives/edgar/data/1564408/000119312517068848/d270216d424b4.htm

Spoonjuice, LLC. (2013). *Fun booth.* Apple Inc. Descargado de https://itunes.apple.com/us/app/fun-booth/id412715794?mt=12

Stallman, R. (2009). *El peligro de las patentes informáticas.* Conferencia en la Universidad Nacional Mayor de San Marcos. (El peligro de las patentes informáticas)

Szűcs, G., Papp, D., y Lovas, D. (2015). Svm classification of moving objects tracked by kalman filter and hungarian method. En *Working notes of clef 2015 conference, toulouse, france.*

Taymans, W., Baker, S., y Wingo, A. (2016). *Gstreamer 1.8.3 application development manual.* ARTPOWER International PUB. Descargado de https://books.google.com.pe/books?id=6ow7vgAACAAJ

Team, T. G. (s.f.). *The GTK+ Project.* https://www.gtk.org/. ((Accessed on 06/22/2018))

The Glade project. (s.f.). *Glade - a user interface designer.* https://glade.gnome.org/. ((Accessed on 06/22/2018))

The GNOME Project. (s.f.-a). *Apps/cheese - gnome wiki!* https://wiki.gnome.org/Apps/Cheese.

The GNOME Project. (s.f.-b). *Apps/cheese - gnome wiki!* https://wiki.gnome.org/Apps/Cheese.

The GNOME Project. (s.f.-c). *cheese: Cheese reference manual.* https://developer.gnome.org/cheese/stable/cheese.html.

The GNOME Project. (s.f.-d). *Cheese overview: Cheese reference manual.* https://developer.gnome.org/cheese/stable/cheese-overview.html. ((Accessed on 06/22/2018))

The GNOME Project. (s.f.-e). *Gnome translation teams.* https://l10n.gnome.org/teams/.

The GNOME Project. (s.f.-f). *Projects/gnomevideoeffects - gnome wiki!* https://wiki.gnome.org/Projects/GnomeVideoEffects.

The GNOME Project. (s.f.-g). *Projects - gnome wiki!* https://wiki.gnome.org/Projects.

The GNOME Project. (s.f.-h). *Vala reference manual.* https://wiki.gnome.org/Projects/Vala/Manual/Overview.

Tomasi, C., y Kanade, T. (1991). Detection and tracking of point features.

Vaterlaus, J. M., Barnett, K., Roche, C., y Young, J. A. (2016). "snapchat is more personal": An exploratory study on snapchat behaviors and young adult interpersonal relationships. *Computers in Human Behavior*, *62*, 594–601.

Viola, P., y Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. En *Computer vision and pattern recognition, 2001. cvpr 2001. proceedings of the 2001 ieee computer society conference on* (Vol. 1, pp. I–I).

Wang, Y.-Q. (2014). An analysis of the viola-jones face detection algorithm. *Image Processing On Line*, *4*, 128–148.

Wettum, Y. (2017). *Facial landmark tracking on a mobile device* (B.S. thesis). University of Twente.

Yann. (2011). *Bug 627928 - add purikura effects (gstgldifferencematte and gstglpixbufoverlay)*. https://bugzilla.gnome.org/show_bug.cgi?id=627928. (Yann es un nombre de pantalla)