UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DESARROLLO DE UN SISTEMA DE LOCALIZACIÓN Y SEGUIMIENTO VISUAL DE OBJETOS

Autor: Enrique Real Perdomo

Tutor: Antonio José Sánchez Salmerón

Fecha: Septiembre de 2017

Master Universitario en Automática e Informática Industrial

ETSII/DISA (Escuela Técnica Superior de Ingeniería Industrial/Dep. Ing. Sist. Aut.)

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Agradecimientos

Ha llegado el momento de cerrar mi vida universitaria y me gustaría agradecer a todas aquellas personas con las que he vivido a mi lado esta experiencia.

En primer lugar me gustaría dar las gracias a mi tutor, Antonio, ya que me ha brindado la oportunidad de realizar este proyecto. Por los consejos recibidos y sobre todo, por el tiempo que me ha dedicado para poder llevarlo a cabo.

También a todos los profesores que me han formado durante la carrera y master, por todos los conocimientos que he adquirido. Agradecer a todos los buenos amigos dentro y fuera la universidad. Asimismo, agradecer a aquellas personas que he conocido en mi vida laboral. A mis compañeros, por todos los consejos y ayuda prestada. En definitiva, gracias a toda la gente que he conocido durante todos estos años.

Por último, gracias a toda mi familia, en especial a mis padres y mis hermanos por haber estado a mi lado en todo momento, dándome ánimos y ayuda durante toda mi vida.

Muchas gracias a todos y hasta pronto.

Enrique Real Perdomo. Septiembre 2017

Memoria

Autor: Enrique Real Perdomo

Tutor: Antonio José Sánchez Salmerón

Fecha: Septiembre de 2017

Master Universitario en Automática e Informática Industrial

ETSII/DISA (Escuela Técnica Superior de Ingeniería Industrial/Dep. Ing. Sist. Aut.)

Índice

Capítulo 1: Introducción

1.1 Objeto

El presente proyecto va a consistir en la creación y programación de un sistema de visión para la localización y el seguimiento de objetos. Es un trabajo real en el ámbito de la I+D+i. El objeto de éste se va a centrar en el desarrollo, a nivel de programación, de un nuevo prototipo mediante unas restricciones y necesidades facilitadas del cliente. Se tendrá que tener además en cuenta, no solo estas necesidades del cliente, si no las necesidades de los dispositivos electrónicos, mecánicos o eléctricos que se empleen en él, lo cual afectará directamente al desarrollo del mismo.

Para este desarrollo se va a pretender ajustar las variables físicas correspondientes a un sistema de visión, tales como distancia focal, iluminación, lentes, etc. Además tendremos que aprender a utilizar herramientas de programación y calibración de imagen, utilizando el lenguaje de programación de C++ con las librerías de OpenCV, el cual se implementará en una Raspberry Pi con el editor de texto Emacs.

En este proyecto se ha implementado un sistema de detección y seguimiento de objetos con dos cámaras, denominado Moving Object Detection and Tracking, cuya abreviatura será MODET.

1.2 Antecedentes del proyecto

A lo largo de la historia ha habido diferentes sistemas de medición y observación. En la actualidad estos sistemas han aumentado en complejidad al aumentar la demanda en diferentes campos de investigación.

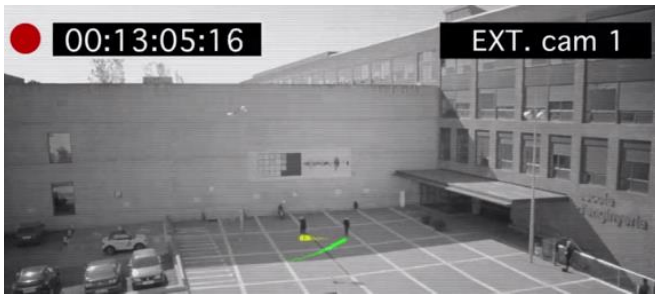
El seguimiento de objetos o tracking es una tarea importante de visión artificial, con múltiples aplicaciones, que explicaré más adelante. El seguimiento de un objeto puede definirse como la estimación, a partir de una secuencia de *frames* obtenida con una cámara de vídeo, de la trayectoria del objeto sobre el plano de la imagen a medida que se mueve por el escenario.

Dado que las cámaras tienen un campo de visión limitado, normalmente es necesario utilizar más de una cámara para cubrir todo el espacio de interés. Por ejemplo las aplicaciones de video vigilancia presentan las imágenes de todas las cámaras a un observador para su análisis. Aún así es muy difícil que una persona consiga concentrarse en varios videos a la vez, de aquí que, últimamente, haya habido un gran interés en el desarrollo de un conjunto de cámaras y que este análisis se muestre de forma más compacta y fácil de entender para los usuarios finales. Además, al usar varias cámaras podemos conseguir trayectorias reales de los objetos en 3D.

Al tratar imágenes en 3D normalmente utilizamos cámaras cuyos campos de visión están solapados. Esto presenta ventajas e inconvenientes. Por un lado, al contar con varias vistas de los objetos es posible extraer información sobre la estructura tridimensional de la escena. Pero, por otro lado, se genera cierta ambigüedad a la hora de seguir estos objetos, puesto que un mismo objeto puede tener un aspecto diferente bajo distintos puntos de vista. Es por tanto necesario identificar múltiples proyecciones de los objetos como el mismo elemento 3D, y etiquetarlas de manera consistente en todas las cámaras, si se quiere utilizar este tipo de sistemas en la aplicación.

Nos encontramos a día de hoy con muchísimas aplicaciones que utilizan estas técnicas de visión, citaré algunos ejemplos:

- **Medios de comunicación y realidad aumentada**. El seguimiento y localización de objetos es un elemento importante para la post-producción y la captura de movimiento para industrias de cine y televisión.
- **Aplicaciones médicas e investigación biológica**. El seguimiento y localización de objetos es utilizado para ayudar al diagnóstico del paciente y así poder ayudar al cirujano con el diagnóstico. Un ejemplo sería la determinación de objetos como agujas durante una intervención.
- **Vigilancia**. Se utiliza en la vigilancia automática en vídeo para seguridad, la vida asistida y las aplicaciones de inteligencia de negocio.



Fuente: svcl.ucsd.edu

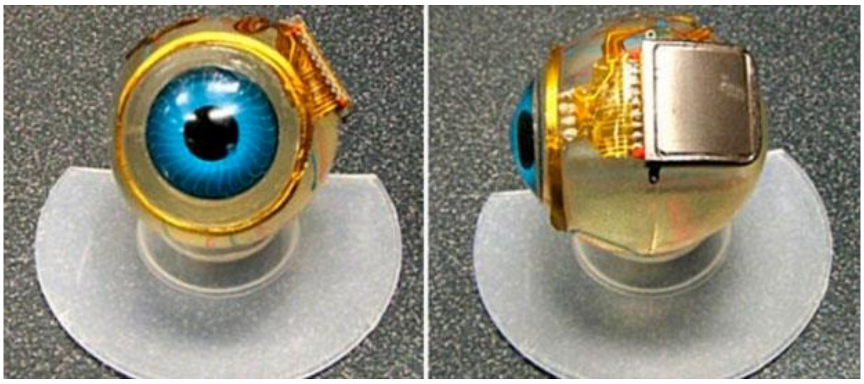
- **Instalaciones de arte y espectáculos.** El seguimiento de objetos se utiliza cada vez más en instalaciones de arte y en actuaciones donde la interacción es posible gracias al uso de cámaras de vídeo. La interactividad puede ser utilizada para mejorar la narrativa de una obra o para crear acciones inesperadas o reacciones del entorno.

- **Tele-colaboración y videojuegos**. Las cámaras web estándar ya incluyen el software de seguimiento que localiza y sigue la cara de un usuario para videoconferencias. Por otro lado, el seguimiento de ojos se utiliza para estimar el contacto visual entre los asistentes de una reunión y así mejorar la eficacia de la interacción en videoconferencias. También se están creando nuevos prototipos, de modalidad interacción natural, utilizados cada vez más en el mundo de los videojuegos.

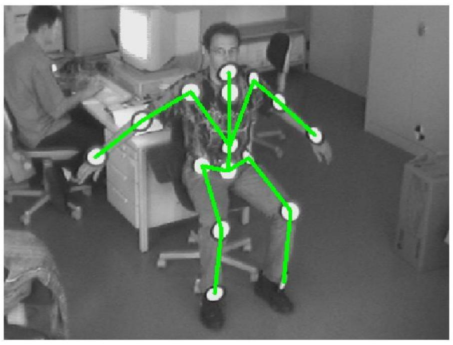
Fuente: Google Imágenes

1.3 Motivación

Una de las líneas de investigación y desarrollo (I+D+i) más interesantes son aquellas que tienen relación con la imitación y simulación de los seres vivos inteligentes. Estudiando sus diferentes estructuras físicas, habilidades y sentidos se pueden crear herramientas de gran interés para el avance tecnológico, y una vez conseguidas es interesante integrarlas en la sociedad, con todos los beneficios que traen a la misma. Unos de estos sentidos a estudiar pueden ser la visión o la audición. En lo que se basa a la capacidad de computadores de percibir sonidos y poder identificarlos se ha avanzado mucho, y ya son muchos los dispositivos cotidianos que son capaces de realizar esta tarea, conocidos como dispositivos con reconocimiento de voz. Sin embargo en lo que se refiere a los dispositivos de visión aún queda un largo camino por recorrer, pero poco a poco se van encontrando nuevos campos para trabajar y avanzar y todo gracias a la ciencia de la visión artificial y su visión-interpretación.


Fuente: Google Imágenes

La visión artificial por computador es una disciplina en creciente auge debido a multitud de aplicaciones, inspección automática, reconocimiento de objetos, mediciones, robótica, inteligencia artificial, microbiología, etc. El futuro que nos encontramos es cada vez más prometedor debido a la creación de máquinas autónomas capaces de interaccionar de forma inteligente con el entorno, para que esto sea posible necesariamente debemos mejorar en las técnicas que doten a estas nuevas máquinas a percibir el entorno. Las nuevas máquinas están dotadas de ordenadores cada vez más potentes, pero aun así, encuentran sus mayores limitaciones en el procesamiento e interpretación de la información que les suministran sus sensores visuales.


Fuente: Google Imágenes

En muchísimas aplicaciones los robots que realizan diferentes tareas de seguimiento se encuentran con un conjunto de objetos posicionados en orientaciones, posiciones y formas arbitrarias, con lo que es bueno tener un buen sistema técnico de reconocimiento de objetos, centrándonos en el reconocimiento estéreo, reconocimiento 3D.

1.4 Justificación del proyecto

Por lo expuesto en la motivación: la necesidad de mejorar el reconocimiento de formas y el seguimiento de diversos objetos para diferentes tareas dentro del campo de la robótica y del sector industrial en general, se llega a la conclusión de que este proyecto es útil y necesario, además de tratar temas que se encuentran actualmente en desarrollo e investigación.

Capítulo 2: Factores Físicos de la visión

Como estamos trabajando en un ámbito técnico tendremos que tener en cuenta diferentes factores físicos del entorno, algunos efectos de física general y otros más específicos dentro de la visión artificial.

En los sistemas de visión artificial, el éxito de las fases de procesamiento y análisis depende altamente de la calidad de la información en las imágenes.

- La cámara es el dispositivo que, utilizando un juego de lentes (objetivo) reconstruye una imagen sobre un elemento sensible (sensor) y la transmite al sistema de adquisición del computador.
- Esa transmisión puede ser digital o analógica. En el segundo caso, deberá ser convertida a digital en el subsistema de adquisición (digitalizador de imágenes o 'frame grabber').

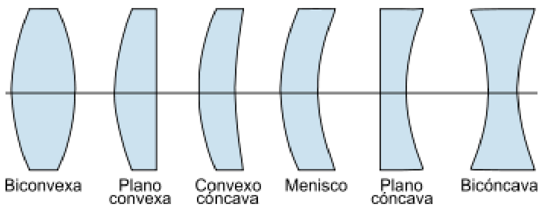
La calidad de la imagen depende pues de la iluminación, la óptica y los sensores utilizados para capturar la imagen.

A continuación explicaré algunos de los efectos físicos que afectan a nuestros sistemas de visión.

2.1 Lentes

Para la percepción o adquisición de una imagen, ésta debe proyectarse sobre una pantalla o superficie fotosensible. Para conseguirlo se utilizan conjuntos de lentes, que refractan la luz hasta situarla en las condiciones deseadas sobre este elemento fotosensible.

Como en muchos sistemas de visión trabajaremos con lentes, y estas tienen diferentes características que debemos de tener en cuenta. A continuación presentaré varias definiciones de términos que se van a utilizar a lo largo de este proyecto, y los cuales debemos conocer y utilizar.

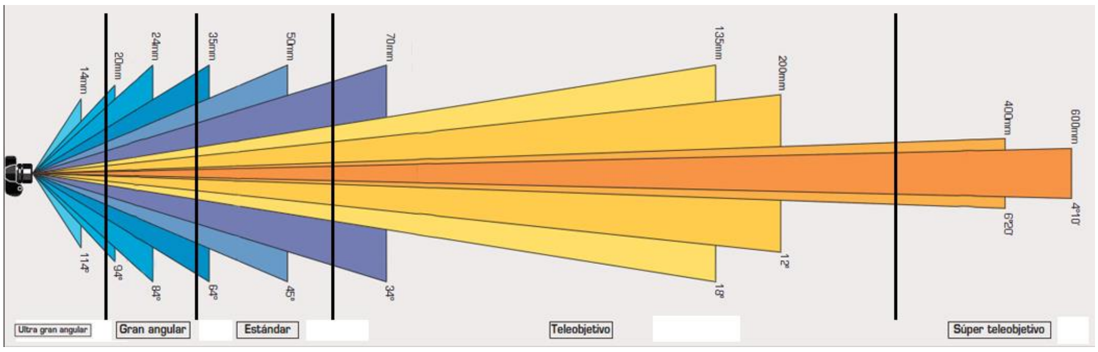


Refracción.

Las lentes funcionan por el efecto de refracción que sufre la luz al atravesar una interfaz entre dos medios de índice de refracción distinto (dióptrico). Es el cambio de dirección de un rayo de luz u otra radiación que se produce al pasar oblicuamente de un medio a otro de distinta densidad.

Distancia focal.

La distancia o longitud focal de una lente se define como la distancia entre el centro óptico de la lente, también llamado plano nodal posterior, y un lugar de la cámara donde la luz forma una imagen enfocada de lo que estamos registrando.



Fuente: fotografiaparaprincipiantes

Enfoque.

Un punto del objeto se proyecta como un pequeño círculo en el plano imagen. Ese punto está bien enfocado si el círculo en la imagen es menor que un tamaño mínimo denominado círculo de confusión.

Enfocar una lente consiste en mover la lente (o el plano imagen) para hacer que los puntos se distingan con nitidez.

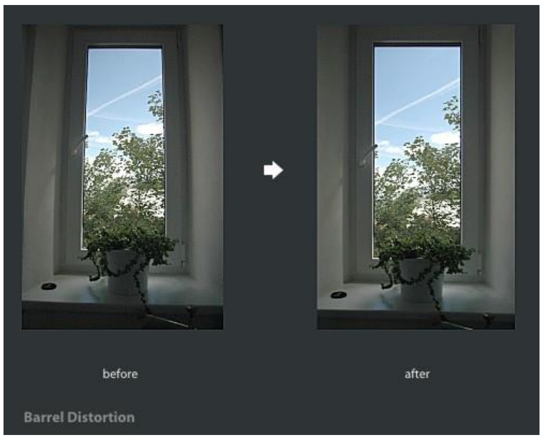
Objetivo.

Generalmente es necesario emplear un conjunto de lentes que proyecten la imagen libre de deformaciones y aberraciones sobre el plano donde reside el elemento sensible (plano imagen). A estos ensamblajes se les llaman lentes compuestas u objetivos.

Otro de los efectos a tener en cuenta es la **distorsión radial**, donde nos encontramos diferentes tipos.

Distorsión de barril.

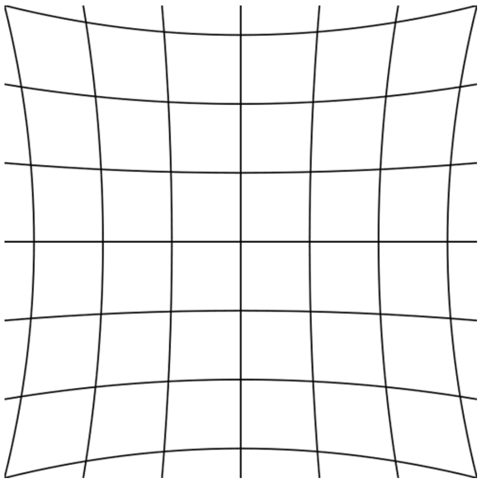
La distorsión de barril es producida por diseños imperfectos de los sistemas de lentes, estos hacen que una fotografía a un cuadrado se vea con los lados curvos, ligeramente salidos hacia afuera. Se observa muy frecuentemente en los extremos "angular" de los objetos.



Fuente: www.gimp.org

Distorsión de cojín.

La distorsión de cojín ocurre cuando la magnificación de la imagen aumenta con la distancia a los ejes ópticos. El efecto visible de esta distorsión es que las líneas tanto verticales como horizontales que no pasan por el centro de la imagen se curvan hacia dentro. Es el efecto contrario a la distorsión de barril, tanto en apariencia como en que suele ocurrir con distancias focales largas.



Fuente: Foto321

Este tipo de distorsión es, excepto en muy raras ocasiones, bastante más leve que la distorsión de barril y en la mayoría de los casos, incluso despreciable. Y por tanto, así como en la distorsión de barril puede usarse de forma creativa, la de cojín realmente no nos da esa opción.

2.2 Sistemas de iluminación

El éxito de un sistema de visión industrial depende mucho del buen diseño del sistema de iluminación más que de un análisis sofisticado de la imagen.

- Las características de la fuente luminosa tiene una gran repercusión sobre las prestaciones y el coste de un SV.
- La imagen es una función de:
    - Las fuentes de luz (tipo de iluminación)
    - La forma de aplicar la luz sobre la escena (técnica de iluminación)
    - Las características de la superficie (capacidad de reflexión, rugosa, metálica, etc.)
    - Las relaciones (distancia y ángulos) entre sensores, superficies y fuentes.
- Un buen sistema de iluminación reduce la complejidad de la imagen a analizar y debe resaltar características de la escena u objetos de interés, o suprimir aquellos detalles que no sean relevantes para el problema.

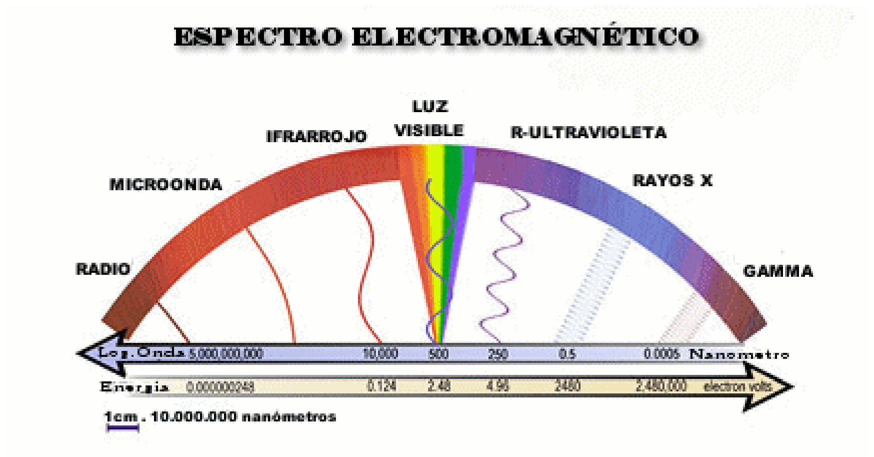
Hay muchos procesos físicos y químicos que generan luz:

- Incandescencia: Emisión de luz por excitación térmica.
- Luminiscencia: Excitación electrónica por energía no térmica.
- Descarga de arco: Ionización de un gas.

Para un sistema de visión son más interesantes aquellos procesos que emiten radiaciones ópticas, es decir, luz en el infrarrojo, visible y/o en el ultra violeta.

Las fuentes de luz (lámparas) más típicas son:

- Incandescente: Lámparas de tungsteno - Halógenos
- Fluorescente. – Láser
- Gas estroboscopico (Xenon) – LEDs


ESPECTRO ELECTROMAGNÉTICO
LUZ VISIBLE
IFRARROJO
R-ULTRAVIOLETA
MICROONDA
RAYOS X
RADIO
GAMMA
Log. Onda 5,000,000,000 10,000 500 250 0.5 0.0005 Nanometro
Energia 0.000000248 0.124 2.48 4.96 2480 2,480,000 electron volts
1cm . 10.000.000 nanómetros

Fuente: meteobasica

Elección de sistema de iluminación.

Los aspectos a considerar para escoger el sistema de iluminación se puede agrupar en:

<u>Por tipo de iluminación:</u>

- Color y composición espectral
- Intensidad de la radiación luminosa
- Regulación de intensidad
- Velocidad de respuesta
- Estabilidad en el tiempo (vibraciones o parpadeos)
- Tiempo de vida
- Requerimientos de refrigeración
- …

Por Técnica de iluminación:

- Costo (adquisición y mantenimiento)
- Ángulo de emisión
- Forma (puntual, esférica, lineal) y tamaño físico
- Tipo de superficies de los objetos
- Tipo de características a resaltar
- Requisitos mecánicos y ambientales
- …

Tipos de iluminación

Básicamente se pueden definir 4 tipos de iluminación utilizables en los SV:

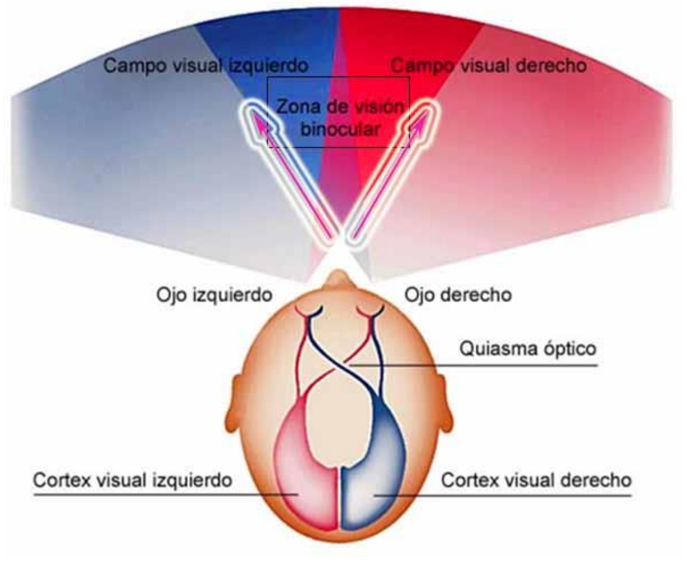
- Iluminación por fibra óptica
- Iluminación por fluorescentes
- Iluminación por diodos LED
- Iluminación por láser

Los factores más importantes para escoger el tipo de iluminación son:

- Intensidad lumínica
- Duración
- Tiempo de respuesta
- Requerimientos de refrigeración
- Coste
- Flexibilidad de diseño

2.3 Visión estereoscópica

La visión estereoscópica, también conocido como visión tridimensional (3D) es cualquier técnica capaz de recoger información visual tridimensional y/o crear la ilusión de profundidad en una imagen. De manera natural, nuestro mecanismo de visión es estéreo, somos capaces de apreciar, las diferentes distancias y volúmenes en el entorno que nos rodea, gracias a nuestro par de ojos (visión binocular). Nuestros ojos, debido a su separación, obtienen dos imágenes con pequeñas diferencias entre ellas, a lo que denominamos disparidad. Nuestro cerebro procesa las diferencias entre ambas imágenes y las interpreta de forma que percibimos la sensación de profundidad, lejanía o cercanía de los objetos que nos rodean. Este proceso se denomina estereopsis.



En la estereopsis intervienen diversos mecanismos. Cuando observamos objetos muy lejanos, los ejes ópticos de nuestros ojos son paralelos. Cuando observamos un objeto cercano, nuestros ojos giran para que los ejes ópticos estén alineados sobre él, es decir, convergen. A su vez se produce la acomodación o enfoque para ver nítidamente el objeto. Este proceso conjunto se llama fusión. La agudeza estereoscópica es la capacidad de discernir, mediante la estereopsis, detalles situados en planos diferentes y a una distancia mínima. Hay una distancia límite a partir de la cual no somos capaces de apreciar la separación de planos, y que varía de unas persona a otras.

Los dos ojos, al estar situados en posiciones diferentes, recogen cada uno en sus retinas una imagen ligeramente distinta de la realidad que tienen delante. Esas pequeñas diferencias se procesan en el cerebro para calcular la distancia a la que se encuentran los objetos mediante la técnica del paralelaje. El cálculo de las distancias sitúa los objetos que estamos viendo en el espacio tridimensional, obteniendo una sensación de profundidad o volumen. Por lo que si tomamos o creamos dos imágenes con un ángulo ligeramente distinto y se las mostramos a cada ojo por separado, el cerebro podrá reconstruir la distancia y por lo tanto la sensación de tridimensionalidad.

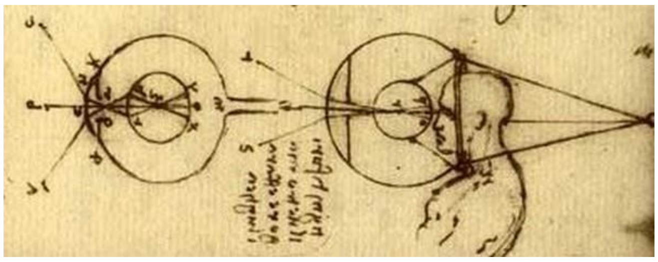
Las variaciones verticales son indiferentes en lo que respecta a creación de sensación de volumen. Solo las variaciones horizontales, producidas por la diferente ubicación de los ojos, resultan en sensación de profundidad. Debe añadirse que, si bien es la esteroscopía la principal fuente de información del cerebro para la composición tridimensional de los objetos que estamos viendo, no es la única. Existen otras fuentes de información como son el enfoque o la interpretación inteligente de las imágenes, que también son utilizadas por el cerebro.

2.3.1 Historia de la visión esteoroscópica

Euclides y Leonardo da Vinci ya observaron y estudiaron el fenómeno de la visión binocular. También el astrónomo Kepler dejó unos estudios que comentaban los principios de la misma.

Posteriormente, en 1838, el físico Sir Charles Wheatstone construyó el primer aparato que permitía percibir la tridimensionalidad partiendo de dos imágenes (visor estereoscópico), considerándose a partir de entonces el inventor de la visión esteoroscópica. Este hecho curiosamente sucedió antes del descubrimiento de la fotografía.

En el año 1849, Sir David Brewster diseñó y contruyó la primera cámara esteroscópica. La cámara disponía de un visor que permitía ver las imágenes tomadas por las lentes. Algunos años más tarde, Oliver Wendell Holmes contruyó lo que sería el estereoscopio de mano más popular del s. XIX.



En los años 30 resurgió la estereofotografía con la aparición de las cámaras 3D, como Realist o la ViewMaster. Actualmente son aparatos obsoletos.

A mediados de siglo hubo diferentes intentos de impulsar las películas 3D sin demasiado éxito, puesto que las técnicas utilizadas provocaban problemas de visión. No fue hasta los años 80 cuando surgen películas de alta resolución, como IMAX 3D, que no se ha popularizado.

La estereoscopía se usa en fotogrametría y también para entretenimiento con la producción de estereogramas. La estereoscopía es útil para ver imágenes renderizadas de un conjunto de datos multidimensionales como los producidos por datos experimentales. La fotografía tridimensional de la industria moderna puede usar escáner 3D para detectar y guardar la información tridimensional. La información tridimensional de profundidad puede ser reconstruida partir de dos imágenes usando una computadora para hacer relacionar los pixels correspondientes en las imágenes izquierda y derecha. Actualmente podemos disfrutar de la estereoscopía en cine con el nuevo formato Digital 3D.

Una aplicación novedosa es la tv3d, sobre la cual hace falta más investigación y las universidades y centros de desarrollo deberán fomentar el análisis a través de proyectos de titulación y relacionados.

2.4 Homografía

En el campo de la visión por computador, cualquieras dos imágenes tienen una superficie plana en el espacio relacionada por una homografía (si asumimos el modelo de cámara *pinole*). Esto tiene muchas aplicaciones prácticas, tales como la rectificación de imágenes, registro de imágenes, o cálculo de los movimientos de la cámara, rotación y traslación entre dos imágenes. Una vez que la rotación de la cámara y la traducción se han extraído una matriz homográfica estimada, esta información puede ser utilizada para la navegación, o para insertar modelos de objetos 3D en una imagen o vídeo , de modo que se muestren con la perspectiva correcta y que parezcan parte de la escena original, realidad aumentada.

A la hora de calibrar nuestra cámara es muy importante conocer bien como trabaja la homografía, más adelante se entrará en más detalle de cómo se tuvo que calibrar la cámara, pero ahora explicaré más conceptos sobre la misma.

Si hablamos de geometría, una homografía establece una transformación de proyección entre dos planos diferentes.

Pongo un ejemplo matemático.

Sean *Pa* y *Pb* son dos puntos que pertenecen a un plano *A* y a otro plano *B*.

$$p_a = \begin{bmatrix} x_a \\ y_a \\ 1 \end{bmatrix}, p_b = \begin{bmatrix} x_b \\ y_b \\ 1 \end{bmatrix}$$

La homografía se definiría matemáticamente como una matriz de dimensiones 3x3, *Hab.*

$$\mathrm{H}_{ab} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

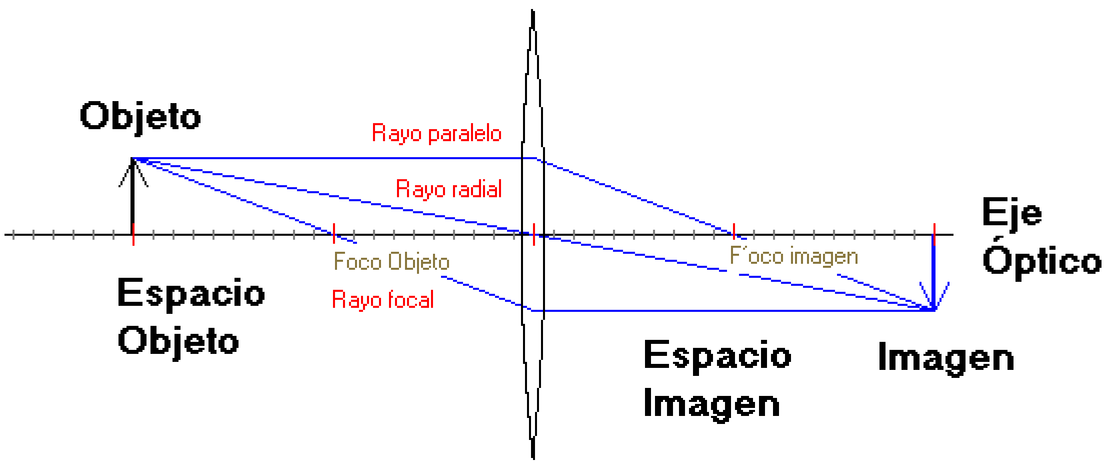
Esta matriz permite transformar un punto *Pa* del plano *A* a un punto *Pb* del en el plano *B* y un punto *Pb* del plano *B* a un punto en el plano *A*, donde *Hba* sería igual a la inversa de *Hab.*

$$p_b = \mathrm{H}_{ab} \times p_a$$

$$p_a = \mathrm{H}_{ba} \times p_b$$

2.5 Cálculos opticos

Debido a que tanto el vidrio/plástico de otras partes captadas pueden producir refracción o reflexión afectando al correcto funcionamiento del objetivo de la cámara, afectadas por luz del sistema de iluminación o incluso a la luz natural, se deben llevar a cabo una serie de cálculos relacionados con el campo de la óptica, para poder prevenir diferentes problemas:



Ecuación fundamental:

$$\frac{n_2}{s_2} - \frac{n_1}{s_1} = \frac{n_2 - n_1}{r}$$

Distancias focales imagen y objeto:

$$f_2 = \frac{n_2}{n_2 - n_1}; f_1 = -r\frac{n_1}{n_2 - n_1}$$

Relación entre las distancias focales y los índices de refracción:

$$\frac{f_1}{f_2} = -\frac{n_1}{n_2}$$

Ecuación de Gauss:

$$\frac{f_2}{s_2} + \frac{f_1}{s_1} = 1$$

Capítulo 3: Seguimiento y Localización visual

En este capítulo presentaremos, de forma global, en qué estado se encuentran los sistemas de seguimiento de objetos, usando una o varias cámaras, así como los algoritmos y las técnicas más utilizadas.

Primeramente proporcionaremos una descripción general de los métodos de tracking de objetos en vídeo y después en el seguimiento de objetos con múltiples cámaras.

3.1 Seguimiento vía vídeo.

El seguimiento de un objeto o tracking es el proceso de estimar, a partir de una secuencia de vídeo, la trayectoria del objeto a medida que se mueve por una escena. Todos los métodos de tracking tienen que localizar primero un objeto en la escena, para posteriormente seguirlo en el tiempo. Muchos estudios se enfocan en la investigación del seguimiento, y consiguen resolver la localización del objeto marcándolo manualmente.

Normalmente la obtención de secuencias se realiza mediante el uso de una o varias cámaras de vídeo. El tracking ha suscitado un alto interés en el área del análisis de video por ordenador, gracias al avance de los procesadores, ligado al menor coste de cámaras con HD.

Algunas de sus aplicaciones serían:

- Identificación humana.
- Automatización de la seguridad y vídeo vigilancia.
- Comunicación y compresión de video.
- Control de tráfico.
- Realidad aumentada.
- Detección automática de objetos.
- Etc.

La estimación de la trayectoria de un objeto mientras se mueve por un escenario puede ser muy compleja. Dependiendo del tipo de objeto a seguir este seguimiento se puede complicar, debido principalmente a la causa de:

- Baja calidad de la imagen.
- Ruido y cambios de iluminación en la escena.
- Alta velocidad de movimiento y cambios de orientación.
- Oclusiones y formas complejas de los objetos.
- Pérdida de información, debido a la proyección del mundo 3D en una imagen 2D.

Con el fin de evitar algunos de estos problemas se imponen restricciones en base a la aparición y al movimiento de los objetos. Por ejemplo, algunos algoritmos de tracking suponen que el movimiento de los objetos es uniforma, otros restringen el movimiento del objeto a que la velocidad y su aceleración sean constantes. También se podría restringir el tamaño y número de objetos, con el fin de evitar apariciones no deseadas.

A continuación mostraré diferentes clasificaciones de los métodos más utilizados en sistemas de detección y seguimiento de objetos.

3.1.1 Detección de objetos

La detección de objetos es la primera fase de todo sistema de seguimiento. Tiene por objetivo reconocer los objetos de interés que aparecen en la escena.

Todos los sistemas de tracking requieren un mecanismo de detección de objetos, ya sea en cada *frame* o sólo cuando el objeto aparece por primera vez en el vídeo. La localización de objetos puede darse manualmente, seleccionando el contorno del objeto, o automáticamente, mediante diferentes algoritmos de detección.

Un enfoque común de la detección de objetos es el uso de información de un único *frame*. Sin embargo, algunos métodos utilizan la información adquirida temporalmente en varias secuencias de *frames* para reducir el número de falsas detecciones.

El primer paso de la detección consiste en distinguir entre las regiones en movimiento, que corresponden a objetos móviles como personas o vehículos, y el fondo. Una vez que se adquieren las regiones en movimiento hay que extraer las características de cada uno de los objetos. Posteriormente, se define el tipo de representación que se va a utilizar para definir al objeto detectado.

A continuación describiremos las técnicas más habituales en la localización de regiones en movimiento y, en segundo lugar, estudiaremos los distintos tipos de representación que se pueden utilizar para delimitar un objeto y realizar su posterior seguimiento.

3.1.1.1 Localización de regiones en movimiento.

El objetivo principal de la localización de regiones en movimiento es la distinción entre el primer plano de los objetos y el fondo estático de la escena. Debido a los cambios que se producen en los entornos naturales es bastante complicado llegar a detecciones de movimiento fiables.

Las técnicas más frecuentes serían las siguientes:

- **Sustracción de fondo:** Es una técnica muy utilizada en la localización de escenas con fondos estáticos. Este método consiste en detectar las regiones de movimiento restando pixel a pixel la imagen actual con una imagen de referencia. En los píxeles donde la imagen de referencia y la actual son superiores al umbral "T", se clasificarían como primer plano.

  $$\left|I_t(x,y) - B_t(x,y)\right| > T$$

  La imagen de referencia se actualiza con el tiempo para adaptarse a los posibles cambios en la escena. La ecuación que definirá la nueva imagen será:

  $$B_{t+1}(x,y) = \alpha I_t(x,y) + (1-\alpha)B_t(x,y)$$

  Donde *alpha* será el coeficiente de adaptación.

- **Métodos estadísticos:** Se han desarrollado métodos estadísticos más avanzados que superan a los de sustracción de fondo. Las estadísticas del fondo se actualizan dinámicamente durante su proceso. En cada fotograma este almacena y actualiza las estadísticas correspondientes al fondo.

  Para identificar los puntos del primer plano, se comparan las estadísticas de cada píxel con las estadísticas de la imagen de referencia. Esta técnica es cada vez más popular debido a su robustez frente a escenas que contengan ruido y cambios de iluminación.

- **Diferencias temporales:** Los métodos de diferenciación temporal comparan píxel a píxel entre dos o tres *frames* consecutivos para detectar regiones en movimiento. Una de las ventajas de este método es la adaptabilidad a los cambios dinámicos en la escena, pero no obtiene todos los píxeles correspondientes al primer plano en objetos que tienen una textura uniforme o se mueven lentamente. En esta clase de métodos, la extracción de los píxeles en movimiento es simple y rápida.

  $$|I_t(x,y) - I_{t-1}(x,y)| > T$$

- **Flujo óptico:** Los algoritmos de flujo óptico detectan las regiones en movimiento a través del estudio de los vectores de flujo de la secuencia. Se pueden detectar objetos en movimiento, incluso en secuencias grabadas con cámaras en movimiento. Sin embargo, la mayoría de los métodos de flujo óptico son computacionalmente complejos, lo que conlleva altos tiempos de ejecución.

- **Detección de cambios de luz:** Los algoritmos de detección de movimiento descritos anteriormente son susceptibles a cambios locales de iluminación, haciendo que estas técnicas sean inexactas. Se presenta un algoritmo de detección de objetos en movimiento en un escenario con cambios de iluminación. En el cada píxel se representa por un modelo de color que separa el brillo de la componente de cromaticidad. Se clasificarían los píxeles en una categoría. Fondo, fondo sombreado, fondo resaltado y objeto de primer plano en movimiento.

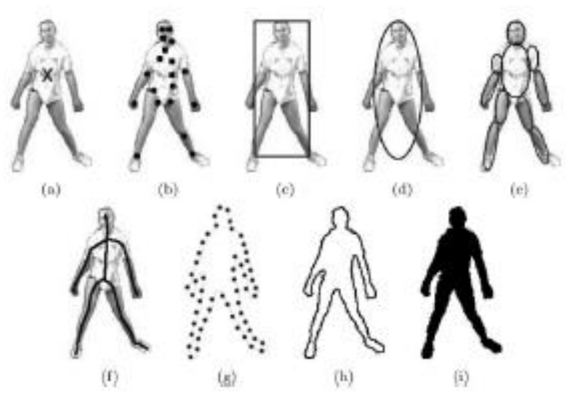
  En todos los algoritmos de localización de movimiento, las regiones en movimiento detectadas corresponden a diferentes objetivos. Para poder diferenciarlos los píxeles de primer plano se agrupan en regiones conectadas, las cuales se representan de una manera que se pueda realizar un posterior seguimiento a lo largo de distintos fotogramas.

3.1.1.2 Representación de objetos

Para realizar un seguimiento fiable es muy importante representar de manera apropiada el objeto detectado. Cada tipo de representación se ajusta más a un objetivo que a otro.
Los diversos patrones que existen se pueden dividir en dos grandes grupos: modelos basados en formas y modelos basados en apariencia.

Los *métodos basados en formas* utilizan las siguientes características:

- **Puntos:** El objeto se representa por un punto, centroide, o por un conjunto de puntos. Este método es adecuado para objetos que ocupan regiones pequeñas de la escena.
- **Formas geométricas**: El objeto se representa por formas geométricas tales como un rectángulo o una elipse. Estas formas son adecuadas para la representación de objetos rígidos, aunque también son utilizadas en el seguimiento de objetos flexibles con gran éxito.
- **Silueta y contorno:** El objeto se representa por su contorno, o por la silueta. Este tipo de modelo es ideal para el seguimiento de formas flexibles como personas.
- **Formas articuladas:** Este método se basa en la representación de formas geométricas con la variación de utilizar una forma geométrica para cada una de las partes rígidas del objeto.
- **Esqueleto:** Consiste en la utilización del esqueleto del objeto para representarlo. Esta técnica es muy utilizada en aplicaciones de reconocimiento de objetos.



Fuente: CarlosLeguaCruz

Los *métodos basados en apariencia* utilizan información cromática o de contorno de los píxeles del objeto. Las representaciones más comunes en el seguimiento de objeto son:

- **Densidad de probabilidad:** Los objetos pueden definirse por las funciones de densidad de probabilidad de que su color o textura sean localizados en sucesivos fotogramas. Para conseguir estas medidas se pueden utilizar diferentes funciones de densidad como Gaussianas o mezclas de Gaussianas. Las densidades de probabilidad de la apariencia de los objetos, como color o textura, pueden calcularse a partir de regiones de la imagen especificadas por representaciones de forma, como elipses o rectángulos.

- **Patrones:** Están formados por formas geométricas o siluetas predefinidas. Tiene un gran rendimiento en el seguimiento de cabezas y manos. Una de las ventajas de los patrones es que contienen información espacial e información cromática del objeto.

- **Modelo de apariencia activo:** Este modelo está basado en la generación de información sobre la forma y apariencia del objeto. La forma se define por un conjunto de puntos de referencia para cada uno de estos puntos, se almacena un vector con información de su apariencia como el color, textura o magnitud del gradiente.

- **Modelo de apariencia multivista:** Esta representación almacena información de diferentes vistas del objeto para alcanzar un sistema menos susceptible a cambios de forma u orientación del objeto.

La detección de objetos tiene una alta sensibilidad, una detección errónea de un objetivo será arrastrada a lo largo del sistema de tracking.

3.1.2 Seguimiento de objetos

El objetivo de seguimiento de objetos es identificar las relaciones afines de un mismo objeto entre *frames* adyacentes con el fin de obtener su transformación de movimiento, su posición y el área que lo delimita en cada *frame* del vídeo. Se ejecuta a continuación de la detección del objeto y depende del modelo de representación del objeto seleccionado.

Cada ciclo del algoritmo se divide en dos etapas:

- **Predicción:** se estima la posición del objeto en el siguiente fotograma de la secuencia.
- **Fase de corrección:** se actualiza la información del objeto con los datos extraídos en el nuevo *frame* para obtener un mejor resultado.

El seguimiento es la parte con mayor carga computacional ya que necesita información de la escena e información previa del objeto. Dependiendo de la calidad, tamaño y duración del vídeo, también puede ser necesaria una gran capacidad de almacenamiento. Los algoritmos de seguimiento de objetos pueden clasificarse en tres grandes grupos: Tracking por puntos, núcleo y silueta.

El ***tracking por puntos*** se define como la correspondencia entre los puntos de un *frame* que definen un objeto con respecto a los mismos puntos del *frame* anterior, para así poder predecir su movimiento. Estos se eligen de modo que sean robustos frente a cambios de escala, rotación o transformaciones afines. Estos algoritmos se pueden clasificar en dos categorías:

- **Métodos deterministas:** Buscan correspondencias entre los puntos del objeto en el instante de la secuencia con los puntos del objeto en el instante. Para ello se utilizan todas las combinaciones posibles de correspondencia de puntos y posteriormente se escogen las correctas mediante métodos de asignación óptimos. Para estas correspondencias, generalmente se definen una combinación de limitaciones. este tipo de restricciones también se utilizan en métodos estadísticos.

- **Métodos estadísticos:** Estas técnicas, al igual que las anteriores, buscan correspondencias entre puntos de cada objeto en distintos instantes de tiempo y también utilizan las restricciones vistas en los métodos deterministas. La diferencia entre ambos métodos es la adopción de un cierto modelado en las medidas por parte de éste, cuantificado en forma de error.

El ***tracking de núcleo***, se basa en el seguimiento de la forma y apariencia de cada objeto. Los objetos son seguidos mediante el cálculo del movimiento del núcleo en *frames* consecutivos.

Este tipo de tracking se puede dividir en dos métodos:

- **Tracking basado en patrones y modelos de apariencia:** Este modelo es muy utilizado debido a su sencillez y bajo coste computacional. Se basa en la utilización de patrones o características del objeto para prever el movimiento del objeto a través de las distintas imágenes, como por ejemplo, histogramas de color y gradientes para el seguimiento de caras.

- **Tracking mediante modelos de apariencia multivista:** Consiste en aumentar la información de un objeto, realizando un aprendizaje previo, utilizando múltiples vistas del mismo. Con esto, el sistema será robusto ante problemas como cambios de orientación o apariencia del objeto.

El tracking basado en siluetas es muy utilizado en el seguimiento de objetos con formas complejas, como manos, cabeza u hombros, que no pueden ser descritos de manera eficiente por formas geométricas simples.

Se cataloga en dos grupos:

- **Tracking de siluetas**: Utiliza como patrones de búsqueda siluetas complejas.

- **Tracking de contorno**: Esta técnica intenta seguir el objeto adaptando iterativamente la transformación de su silueta.

3.2 Tracking de objetos basado en puntos de interés

Hoy en día, muchos sistemas de seguimiento de objetos proponen la localización de puntos y la descripción de sus características para posteriormente encontrar correspondencias entre los puntos. Aunque los sistemas difieren, generalmente todos siguen las pautas que se citan a continuación:

Capturar un nuevo frame de la secuencia de vídeo.

Detectar los puntos de interés en la escena.

Describir las características de cada uno de los puntos candidatos.

Buscar correspondencias entre los descriptores detectados en el *frame* actual y los descriptores del objeto encontrados en *frames* anteriores.

Estimar el movimiento del objeto utilizando la información de estas correspondencias.

3.2.1 Detección de puntos de interés

Debido a la alta carga computacional para describir un píxel y al gran número de píxeles que hay en una imagen, es necesario utilizar un conjunto de puntos acotado para realizar el seguimiento. Por esta razón nacen los algoritmos de detección de puntos de interés. Estos puntos pueden ser, por ejemplo, esquinas y bordes de objetos.

Los principales detectores utilizados para la localización de puntos de interés son:

- Detector Harris.
- Detector Shi-Tomasi
- Diferencia de Gaussianas (DoG)
- Detector Fast Hessian
- Detector FAST
- Detector de Harris-Laplace
- Detector SUSAN

3.2.2 Descripción de características

Una vez que hemos localizado los puntos de interés de la imagen, describimos sus características. Un descriptor está definido por las especificaciones locales del punto. Un descriptor es único para el mismo punto en cualquier instante de una secuencia.

La principal flaqueza de estas técnicas suele ser la complejidad de cálculo de los descriptores que representan cada uno de los puntos detectados, pero debido a los avances computacionales se ha convertido en una técnica estándar para el seguimiento de objetos.

Me dispongo a citar los descriptores más conocidos:

- Descriptor SIFT
- Descriptor SURF
- Descriptor BRISK
- Descriptor BRIEF
- Descriptor ORB
- Descriptor ORB
- Descriptor FREAK

3.2.3 Correspondencias de puntos y estimación de movimiento

En los sistemas de tracking por puntos de interés, se buscan correspondencias entre los *keypoints* detectados en el *frame* actual y los puntos almacenados. Los puntos almacenados pertenecen a *keypoints* detectados en *frames* anteriores. El objetivo de buscar estas correspondencias es calcular el modelo de transformación que siguen los puntos para estimar, utilizando ese modelo de transformación, la posición de los puntos en el *frame* actual.

Para la localización de correspondencias entre dos conjuntos de *keypoints* se utiliza el cálculo del valor de similitud, conocido como distancia o *score*.

3.3 Sistemas de múltiples cámaras

Al igual que en los sistemas de seguimiento en una cámara, el objetivo de los sistemas de seguimiento multicámara es fusionar los datos para detectar y seguir de forma más robusta recursivamente los objetos de interés que hay en la escena aumentando además el plano de visión.

El análisis de información en sistemas de múltiples cámaras se puede clasificar en tres tipos: basados en apariencia, basados en la geometría y sistemas híbridos. Además existe otro método de clasificación en función del orden de análisis de la información: fusionar la información antes del tracking y fusionar la información después del tracking.

La fusión de datos es de gran utilidad para poder ampliar el escenario de una sola vista a un marco más amplio.

Los métodos basados en geometría utilizan generalmente geometría epipolar, homografía o calibración de la cámara. Utilizaremos la homografía como método para relacionar las cámaras. Como veremos en los siguientes apartados, para adquirir la relación geométrica entre las cámaras hay que disponer de un conjunto de puntos físicos en el escenario visible desde todas las cámaras.

3.3.1 Homografía

En geometría, se denomina homografía a toda transformación proyectiva que determina una correspondencia entre dos figuras geométricas planas, de forma que a cada uno de los puntos y las rectas de una de ellas le corresponden, respectivamente, un punto y una recta de la otra. La homografía tiene multitud de aplicaciones en el tratamiento de imágenes como son la reconstrucción 3D, la calibración de cámaras, la creación de panoramas o la ayuda en el tracking.

Normalmente, se utiliza la homografía para lograr una transformación perspectiva entre planos y así llevar la proyección de un objeto desde una vista a otra.

Ya se explicó en el apartado de factores físicos la homografía y en que se basa matemáticamente de forma más detallada.

3.3.2 Geometría epipolar

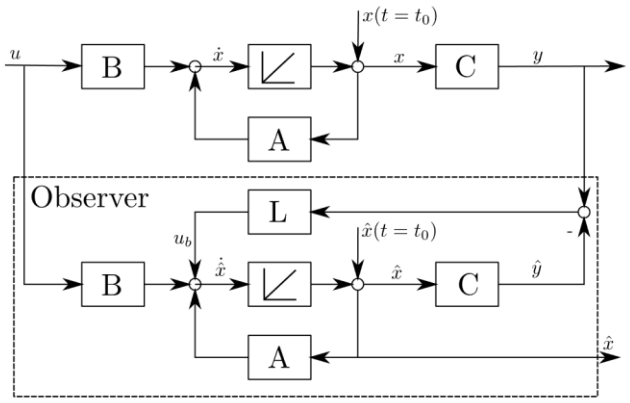
Otro método de fusión de información basada en la geometría utilizado para el tracking es la geometría epipolar. La geometría epipolar expresa la relación geométrica entre dos proyecciones de un mismo punto físico. Una de las principales aplicaciones que podemos darle a la geometría epipolar es la de determinar si la proyección en una vista, y la proyección en otra vista, corresponden al mismo punto en la escena.

Independientemente de lo que forma la escena tridimensional, todo par de imágenes tiene unas restricciones geométricas. Una de estas restricciones es la restricción epipolar, que ayuda a limitar el espacio de búsqueda de correspondencia de puntos. Esta restricción dice que todos los planos epipolares originan líneas horizontales al cortarse con los planos de las imágenes reduciendo el espacio de búsqueda a una dimensión.

3.4 Observador de Luenberger

El observador de Luenberger es un estimador de estado, es decir, un algoritmo que permite estimar el estado interno oculto (no medible) de un sistema dinámico lineal a partir de las mediciones de la entrada y la salida de dicho sistema.

La idea en la cual se basa este observador es en generar un sistema "clon" del original, al cual sí se le pueda medir el estado interno directamente. Si el sistema original y su clon son sometidos a los mismos estímulos (la misma entrada), se puede esperar que, a medida que pase el tiempo, se comiencen a comportar del mismo modo debido a que sus estados internos tienden a parecerse cada vez más (lo anterior funciona siempre que el sistema original y su clon sean estables). De este modo, el estado interno del clon se puede usar como una aproximación del estado interno del sistema original.



Fuente: Wikipedia

Para acelerar la convergencia del estado del sistema clon al estado del sistema original, se puede estimular al clon con una entrada corregida, que consiste en la misma entrada que el sistema original más la diferencia entre la salida de los dos sistemas multiplicada por una constante. De este modo, se logra modificar la dinámica del sistema clon de modo que logre estimar el estado del sistema original en un tiempo arbitrariamente pequeño (al menos en teoría). Es decir, el clon es capaz de observar tanto la entrada del sistema original como la diferencia entre su salida y la del sistema original, lo que le permite converger más rápido.

3.5 Filtro de Kalman

El Filtro de Kalman es un algoritmo que desarrolló Rudolf E. Kalman, en 1960. Este filtro sirve para identificar el estado oculto, el que no es medible, de un sistema dinámico lineal. La diferencia con el observador de Luenberger, es que en este la ganancia K de realimentación del error debe ser elegida a ojo, mientras que en el filtro de Kalman es capaz de escogerla de forma óptima cuando se conocen las varianzas de los ruidos que afectan al sistema, ya que este filtro es un algoritmo recursivo. El filtro de Kalman puede correr en tiempo real usando únicamente las mediciones de entrada actuales, el estado calculado previamente y su matriz de incertidumbre, no requiere ninguna información adicional.

Fuente: alchetron

El filtro de Kalman es un algoritmo que se basa en el modelo de espacio de estados de un sistema para estimar el estado futuro y la salida futura realizando un filtrado optimo a la señal de salida, y dependiendo de las muestras puede

además cumplir una función de estimador de parámetros o solo de filtro, pero en ambos casos eliminara el ruido. A
diferencia de otros filtros este puede filtrar en todo el espectro de frecuencias. Además sus ecuaciones solo dependen de una muestra anterior y la muestra presente lo que ya que se basa en un método recursivo.

El filtro de Kalman tiene numerosas aplicaciones en tecnología. Un ejemplo es la guía o navegación y control de vehículos, como podrían ser naves espaciales. Además este filtro es ampliamente usado en campos como procesamiento de señales y econometría.

3.5.1 Algoritmo del Filtro de Kalman

El Filtro de Kalman es un algoritmo recursivo en el que el estado Xk es considerado una variable aleatoria Gaussiana. El filtro de Kalman suele describirse en dos pasos: Predicción y Corrección.

Predicción

Estimación *a priori*	$\hat{\mathbf{x}}_{k\|k-1} = \mathbf{\Phi}_k \, \mathbf{x}_{k-1\|k-1}$
Covarianza del error asociada a la estimación *a priori*	$\mathbf{P}_{k\|k-1} = \mathbf{\Phi}_k \mathbf{P}_{k-1\|k-1} \mathbf{\Phi}_k^{\mathrm{T}} + \mathbf{Q}_k$

Corrección

Actualización de la medición	$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k \hat{\mathbf{x}}_{k\|k-1}$
Ganancia de Kalman	$\mathbf{K}_k = \mathbf{P}_{k\|k-1} \mathbf{H}_k^{\mathrm{T}} (\mathbf{H}_k \mathbf{P}_{k\|k-1} \mathbf{H}_k^{\mathrm{T}} + \mathbf{R}_k)^{-1}$
Estimación *a posteriori*	$\hat{\mathbf{x}}_{k\|k} = \hat{\mathbf{x}}_{k\|k-1} + \mathbf{K}_k \tilde{\mathbf{y}}_k$
Covarianza del error asociada a la estimación *a posteriori*	$\mathbf{P}_{k\|k} = (I - \mathbf{K}_k \mathbf{H}_k) \mathbf{P}_{k\|k-1}$

donde:

$\mathbf{\Phi}_k$: Matriz de Transición de estados.

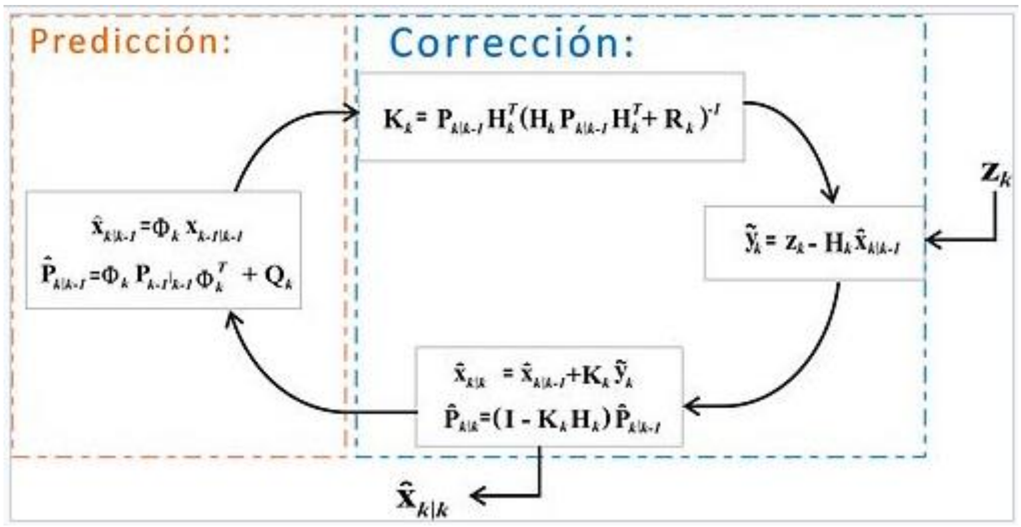
$\mathbf{x}_{k|k-1}$: El estimado apriori del vector de estados.

$\mathbf{P}_{k|k-1}$: Covarianza del error asociada a la estimación a priori.

$\mathbf{z}_k$: Vector de mediciones al momento *k*.

$\mathbf{H}_k$: La matriz que indica la relación entre mediciones y el vector de estado al momento *k*
en el supuesto ideal de que no hubiera ruido en las mediciones.

$\mathbf{R}_k$: La matriz de covarianza del ruido de las mediciones


Predicción:
Corrección:
$K_k = P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1}$
Z_k
$\hat{x}_{k|k-1} = \Phi_k x_{k-1|k-1}$
$\hat{P}_{k|k-1} = \Phi_k P_{k-1|k-1} \Phi_k^T + Q_k$
$\tilde{y}_k = z_k - H_k \hat{x}_{k|k-1}$
$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k$
$\hat{P}_{k|k} = (I - K_k H_k) \hat{P}_{k|k-1}$
$\hat{x}_{k|k}$

3.5.2 Extensibilidad del Filtro de Kalman

En el caso de que el sistema dinámico sea no lineal, es posible usar una modificación del algoritmo de Kalman llamada "Filtro de Kalman Extendido", el cual linealiza el sistema en torno al X(t) identificado realmente, para calcular la ganancia y la dirección de corrección adecuada. En vez de haber matrices A, B y C, hay dos funciones f(x,u,w)}f(x,u,w) y h(x,v) que entregan la transición de estado y la observación (la salida contaminada) respectivamente. El modelo lineal dinámico con observación no lineal y no Gaussiano se estudia en este caso. Se extiende el teorema de Masreliez como una aproximación de filtrado no Gaussiano con ecuación de estado lineal y ecuación de observaciones también lineal, al caso en que la ecuación de observaciones no lineal pueda aproximarse mediante el desarrollo en serie de Taylor de segundo orden.

3.5.3 Conclusiones del filtro de Kalman

- Usa el método de mínimos cuadrados para generar recursivamente un estimador del estado en el momento “K”, que es lineal y de varianza mínima.
- Destacar su habilidad para predecir el estado de un modelo en el pasado, presente y futuro.
- Supone un conocimiento amplio en teoría de probabilidades, en especial con la condicional gaussiana en las variables aleatorias, lo cual puede originar una limitante para su estudio y aplicación.

Capítulo 4: *Programas y material de trabajo*

4.1 RaspBerry y Cámara de la RaspBerry

Se va a utilizar una RaspBerry Pi 2, que irá conectada a una cámara:

Se trata de una diminuta placa base de 85 x 54 milímetros (un poco más grande que una cajetilla de tabaco) en el que se aloja un chip Broadcom BCM2835 con procesador ARM hasta a 1 GHz de velocidad, GPU VideoCore IV y hasta 512 Mbytes de memoria RAM.

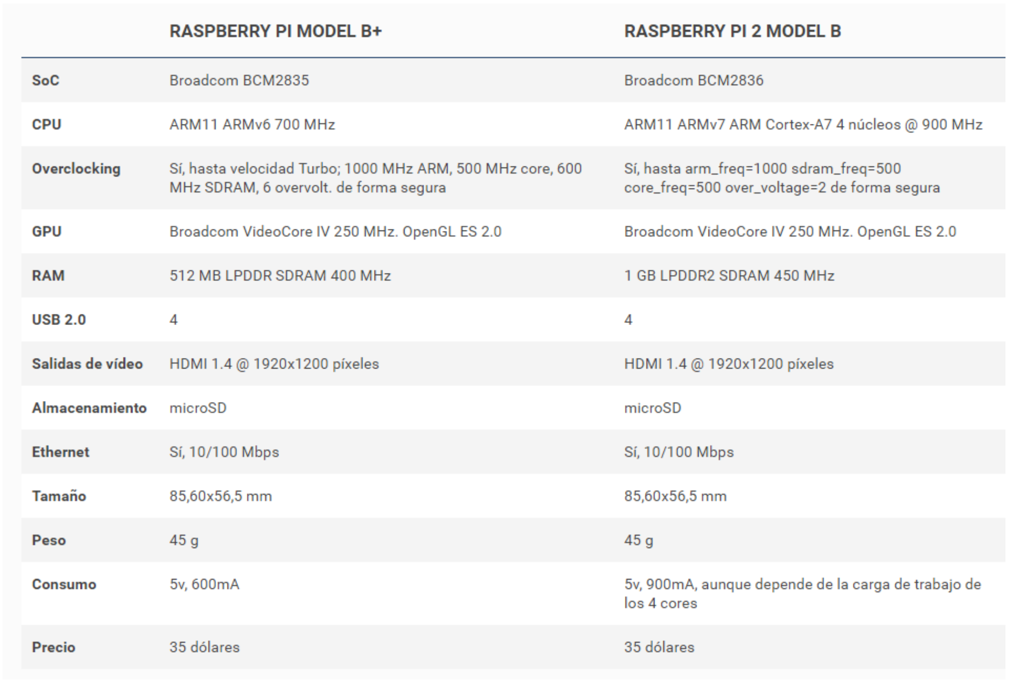
Para que funcione, basta con que se añada un medio de almacenamiento (como por ejemplo una tarjeta de memoria SD), enchufarlo a la corriente gracias a cualquier cargador de tipo micro USB (el mismo que sirve para recargar la mayoría de los teléfonos móviles actuales, su coste es muy bajo) y, si se desea, incorporar un chasis para que todo quede a buen recaudo y su apariencia sea más estética.

En función del modelo que se escoja, se dispone de más o menos opciones de conexión, pero siempre tendremos al menos un puerto de salida de video HDMI y otro de tipo RCA, minijack de audio y un puerto USB 2.0 al que conectar un teclado y ratón.

En cuanto a conexión de red se refiere, puede disponer de Ethernet para enchufar un cable RJ-45 directamente al router o recurrir a adaptadores inalámbricos WiFi.

Para el almacenamiento, Raspberry Pi recomienda utilizar una tarjeta SD con una capacidad mínima de 4 Gbytes y de clase 4 (este valor aparece siempre impreso en la tarjeta, e indica su rendimiento en cuanto a velocidad se refiere).

Para enchufar nuestra RaspBerry Pi a un monitor o televisor, se necesita un cable HDMI o, si no se dispone de tal entrada de video, un cable HDMI a DVI. También es posible recurrir en su lugar a la salida analógica RCA (identificada en nuestra tele por un cable amarillo).Hay diferentes modelos, de los cuales, se va a utilizar el 2, que tiene las siguientes características:



	RASPBERRY PI MODEL B+	RASPBERRY PI 2 MODEL B
SoC	Broadcom BCM2835	Broadcom BCM2836
CPU	ARM11 ARMv6 700 MHz	ARM11 ARMv7 ARM Cortex-A7 4 núcleos @ 900 MHz
Overclocking	Sí, hasta velocidad Turbo; 1000 MHz ARM, 500 MHz core, 600 MHz SDRAM, 6 overvolt. de forma segura	Sí, hasta arm_freq=1000 sdram_freq=500 core_freq=500 over_voltage=2 de forma segura
GPU	Broadcom VideoCore IV 250 MHz. OpenGL ES 2.0	Broadcom VideoCore IV 250 MHz. OpenGL ES 2.0
RAM	512 MB LPDDR SDRAM 400 MHz	1 GB LPDDR2 SDRAM 450 MHz
USB 2.0	4	4
Salidas de vídeo	HDMI 1.4 @ 1920x1200 píxeles	HDMI 1.4 @ 1920x1200 píxeles
Almacenamiento	microSD	microSD
Ethernet	Sí, 10/100 Mbps	Sí, 10/100 Mbps
Tamaño	85,60x56,5 mm	85,60x56,5 mm
Peso	45 g	45 g
Consumo	5v, 600mA	5v, 900mA, aunque depende de la carga de trabajo de los 4 cores
Precio	35 dólares	35 dólares

Instalación del software

Una vez se tenga la Raspberry Pi y los componentes necesarios listos, es hora de instalar el sistema operativo. Lo más sencillo es insertar la tarjeta SD de nuestra elección en nuestro ordenador y formatearla, preferiblemente con la ayuda de SD Formatter 4.0. En la guía de inicio rápido que proporciona la web oficial encontraremos todos los detalles para hacerlo correctamente en función del sistema operativo que estemos utilizando para el formateo (Windows, OS X o Linux).

En este momento, se está ya listo para instalar el sistema operativo de nuestra Raspberry Pi, que se descarga desde downloads.raspberrypi.org/noobs con la ayuda de la herramienta New Out Of Box Software (NOOBS). Se descomprime el archivo en la tarjeta SD y al insertarla en la RaspBerry Pi se va a ver una serie de opciones de configuración, incluyendo un listado de sistemas que se puede instalar fácilmente.

Por otra parte, se dispone de una cámara para toma imágenes de la muestra:

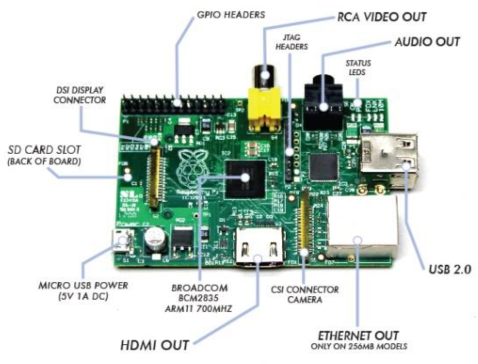
Una cámara para raspberry de 5 megapíxeles que dispone de un diseño personalizado para Raspberry Pi. La cámara se une a la placa a través de unas pequeñas tomas de corriente en la parte superior de la raspberry.

Características:

- Soporta 1080p / 720p / 640x480p Video
- Capaz de 2592 x 1944 píxeles Imágenes estáticas
- 25mm x 20mm x 9mm Huella
- Peso 3g
- Totalmente compatible con los casos ModMyPi

4.1.1 Ventilación

Se va a utilizar ventilación natural.


GPIO HEADERS
RCA VIDEO OUT
JTAG HEADERS
AUDIO OUT
STATUS LEDS
DSI DISPLAY CONNECTOR
SD CARD SLOT (BACK OF BOARD)
USB 2.0
MICRO USB POWER (5V 1A DC)
BROADCOM BCM2835 ARM11 700MHZ
CSI CONNECTOR CAMERA
ETHERNET OUT ONLY ON 256MB MODELS
HDMI OUT

Además añadiremos a la RapsBerry un disipador de calor para que, en conjunto con la ventilación natural, evitar el incorrecto funcionamiento de los dispositivos a causa del calor.

Anteriormente se pensó en utilizar ventiladores, pero se observó que creando una corriente de aire natural y con la ayuda de un disipador de calor es suficiente. Aunque han de estar funcionando de forma prolongada, no necesitan de ventilación forzada según el fabricante de la RaspBerry, con un disipador es suficiente.

Con la ventilación natural nos ahorramos: espacio, ya que por pequeño que sea el ventilador ocupa más que nada; dinero, ya que cada ventilador y fuente de energía asociada tiene un coste; y también reducimos el consumo de energía.

4.1.2 Fuente de alimentación

Se va a utilizar una fuente de alimentación móvil de 5 V y 35 W, 7A para alimentar a la raspi.

Se ha optado por una exterior ya que así se evitan los problemas de calentamiento de la fuente, que podría afectar a la Rapsberry Pi.

Características:

Gross Weight (kg)	0.3850
Manufacturer	EastRising
Continuity Supply	We promise the long term continuity supply for this product no less than 10 years since 20
Part Number(Order Number)	ER-TFTV070-5
Display Format	800x480 Dots
Interface	Video(CVBS), VGA, HDMI
IC or Equivalent	OTA7001A
Appearance	RGB on Black
Diagonal Size	7.0“
Connection	FPC-Connector
Outline Dimension	164.9(W)x100.0(H)mm
Visual Area	156.9x89.00mm
Active Area	154.08(W)x85.92(H)mm
Character Size	No
Dot (Pixel) Size	0.0642(W)x0.1790(H)mm
Dot (Pixel) Pitch	No
IC Package	COG
Display Type	TFT-LCD Color
Touch Panel Optional	Yes
Sunlight Readable	No
Response Time(Typ)	20ms
Contrast Ratio(Typ)	500:1
Colors	65K/262K/16.7M
Viewing Direction	No

Viewing Angle Range	Left:60.0 , Right:60.0 , Up:60.0 , Down:60.0 degree
Brightness(Typ)	200cd/m2
Backlight Color	White Color
Backlight Current (Typ)	No
Power Supply(Typ)	12V
Supply Current for LCM(Max)	No
Operating Temperature	-20℃~70℃
Storage Temperature	-30℃~80℃
Series Number	ER-TFT070-4

4.2 OpenCV

Para lo que viene a ser el desarrollo a nivel programación de este proyecto usaremos *Open Source Computer Vision*, más conocido como *OpenCV*.

OpenCV es una librería de software *open-source* de visión artificial, que originariamente fue desarrollada por Intel. Desde que apareció su versión alfa en el mes de enero de 1999, se ha utilizado en infinidad de aplicaciones. Desde sistemas de seguridad con detección de movimiento, hasta aplicaciones de control de procesos donde se requiere reconocimiento de objetos. Esto se debe a que su publicación se da bajo licencia BSD, que permite que sea usada libremente para propósitos comerciales y de investigación con las condiciones en ella expresadas.

Open CV es multiplataforma, existiendo versiones para GNU/Linux, Mac OS X y Windows. Contiene más de 500 funciones que abarcan una gran gama de áreas en el proceso de visión, como reconocimiento de objetos (reconocimiento facial), calibración de cámaras, visión estérea y visión robótica.

El proyecto pretende proporcionar un entorno de desarrollo fácil de utilizar y altamente eficiente. Esto se ha logrado realizando su programación en código C y C++ optimizados, aprovechando además las capacidades que proveen los procesadores multinúcleo. OpenCV puede además utilizar el sistema de primitivas de rendimiento integradas de Intel, un conjunto de rutinas de bajo nivel específicas para procesadores Intel.

4.3 Emacs

Emacs es un editor de texto con una gran cantidad de funciones, muy popular entre programadores y usuarios técnicos. GNU Emacs es parte del proyecto GNU y la versión más popular de Emacs con una gran actividad en su desarrollo. GNU Emacs se podría describir como un editor extensible, personalizable, auto-documentado y de tiempo real.

El EMACS original significa, Editor MACroS para el TECO. Fue escrito en 1975 por Richard Stallman junto con Guy Steele. Se han lanzado muchas versiones de EMACS hasta el momento, pero actualmente hay dos que son usadas comúnmente: GNU Emacs, iniciado por Richard Stallman en 1984, y XEmacs, una fork de GNU Emacs, que fue iniciado en 1991.

GNU Emacs comenzó siendo un editor de texto, debido al uso de Emacs Lisp (un dialecto del lenguaje de programación Lisp) al ser este extensible es en la actualizad una herramienta útil para un gran número de tareas.

Con GNU Emacs es posible editar un conjunto de lenguajes de programación al existir un sin número de modos, estos modos nos permitirán usarlo como navegador web, administrador de archivos, reproductor multimedia, cliente de mensajería instantánea, cliente para microblog, cliente irc, lector de noticias, calculadora, calendario, jugar (tetris, pong, entre otros), además de un largo etcetera.

Capítulo 5: Desarrollo del proyecto.

Nos encontramos ya en lo que viene a ser el desarrollo del proyecto, para ello realizaremos una serie de pasos, pero antes hagamos mención a trabajos realizados anteriormente por el grupo de robótica del ai2, los cuales me han ayudado a la hora de realizar el proyecto que posteriormente me dispongo a explicar.

Esta Bibliografía también la dividiré en las principales partes del proyecto; Calibración, procesamiento de imágenes, localización y seguimiento.

Calibración:

Ricolfe, C., & Sánchez, A. J. (2008). PROCEDIMIENTO COMPLETO PARA EL CALIBRADO DE CÁMARAS UTILIZANDO UNA PLANTILLA PLANA. RIAII, 5(1), 93-101.

Ricolfe-Viala, C., & Sanchez-Salmeron, A. J. (2007). Improved camera calibration method based on a two-dimensional template. Pattern Recognition and Image Analysis, 420-427.

Procesamiento de imágenes:

Benlloch, J. V., Agusti, M., Sanchez, A., & Rodas, A. (1995, October). Colour segmentation techniques for detecting weed patches in cereal crops. In Proc. of Fourth Workshop on Robotics in Agriculture and the Food-Industry (pp. 30-31).

Benlloch, J. V., Sanchez, A., Christensen, S., & Walger, M. (1996). Weed mapping in cereal crops using image analysis techniques. AgEng96, 2, 1059-1060.

Benlloch, J. V., Sánchez, A., Agustí, M., & Albertos, P. (1996). Weed Detection in Cereal Fields Using Image Processing Techniques. Precision Agriculture, (precisionagricu3), 903-903.

Seguimiento:

Sánchez, A., & Marchant, J. (1997). Fast and robust method for tracking crop rows using a two point Hough transform. Acts of BioRobotics, 97.

Antes de iniciar nuestro desarrollo estudiaremos las condiciones del encargo.

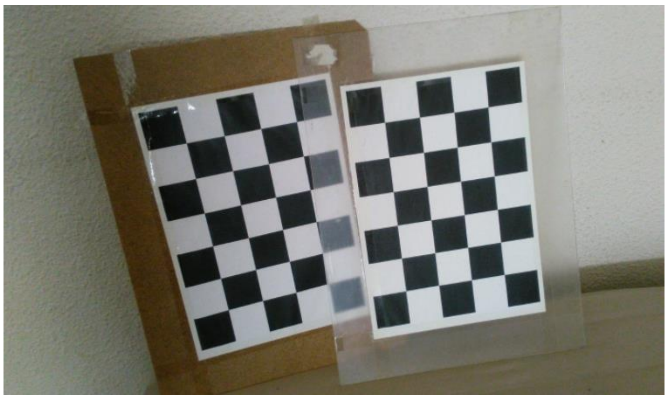
5.1 Condiciones del encargo

Las condiciones de encargo de las que se dispone para llevar a cabo el diseño, programación y funcionalidad del sistema son las siguientes a rasgos generales.

- Necesitaremos usar una rapsberry Pi con su cámara correspondiente.
- Se necesitarán usar dos cámaras, ya que se quiere usar un seguimiento 3D.
- Se deberá programar en C++ Usando Emacs y las librerías de OpenCV

5.2 Calibración de la cámara

Para poder realizar la calibración hemos utilizado unos patrones de tipo ajedrez, como los que se pueden ver en la siguiente fotografía.



Inicialmente realizaremos la calibración de una sola cámara, para familiarizarnos con el programa y con la calibración.

Lo primero que tendríamos que tener en cuenta es la distorsión para ello OpenCV tiene en cuenta los factores radiales y tangenciales. Para el factor radial se utiliza la siguiente fórmula:

$$x_{distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{distorted} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Para un punto distorsionado del pixel con las coordenadas (x,y) su posición en la imagen distorsionada será (Xdistorsionada, Ydistorsionada). La presencia de distorsión radial generará distorsión de barril y el conocido como efecto de ojo de pez.

La distorsión tangencial ocurre porque la imagen tomada por las lentes no es perfectamente paralela al plano de la imagen. Se representa con las siguientes formulas:

$$x_{distorted} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$
$$y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

Entonces tendremos cinco parámetros los cuales en OpenCV se representan como una matrix de 5 columnas:

$$distortion\_coefficients = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

Ahora para la conversión de unidades utilizaremos la siguiente formula:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

He aquí la presencia de w, la cual se explica por uso de la homografía en un sistema de coordenadas (w=Z). Los parámetros desconocidos son fx y fy (longitudes focales de la cámara) y (cx,cy) las cuales son centros ópticos que representan las coordenadas de los píxeles. Si en ambos ejes una longitud focal común se utiliza como aspecto de relación (normalmente 1), entonces fy=fx*a y en la fórmula superior tendremos una longitud focal única f. La matriz que contiene estos cuatro parámetros se conoce como la matriz de la cámara. Mientras que los coeficientes de distorsión son los mismos independientemente de las resoluciones de la cámara utilizados, estos deben ser escalados, junto con la resolución actual de la resolución calibrada.

El proceso de determinar estas dos matrices es la calibración. El cálculo de estos parámetros se realiza a través de ecuaciones geométricas básicas. Las ecuaciones utilizadas dependen de los objetos de calibración elegidos. Actualmente OpenCV es compatible con tres tipos de objetos para calibrar:

1. Clásico tablero de ajedrez blanco y negro.
2. Modelo de círculo simétrico.
3. Modelo de círculo asimétrico.

Básicamente lo que necesita es tomar instantáneas de estos patrones con su cámara y dejar a OpenCV encontrarlos. Cada patrón encuentra como resultado una nueva ecuación. Para resolver la ecuación necesitarás al menos un número predeterminado de patrones de instantáneas para formar un sistema de ecuaciones bien planteado. Este número es mayor para el patrón de tablero de ajedrez y menos para los de los círculos.

Por ejemplo, en teoría el tablero de ajedrez requiere por lo menos dos fotografías. Pero, en la práctica, tenemos una buena cantidad de ruido presente en nuestras imágenes, por lo que para obtener un buen resultado necesitaremos al menos diez buenas instantáneas del patrón de entrada en diferentes posiciones.

5.2.1 Código y explicación

Ahora me dispongo a explicar y a mostrar el código implementado para lo que viene a ser el desarrollo del proyecto.

Inicialmente tendremos que insertar todas las librerías que tengan relación con la calibración, para ello usaremos un código base facilitado por OpenCV, el tutorial_code de calibración.

```
#include "opencv2/core.hpp"

#include <opencv2/core/utility.hpp>

#include "opencv2/imgproc.hpp"

#include "opencv2/calib3d.hpp"

#include "opencv2/imgcodecs.hpp"

#include "opencv2/videoio.hpp"

#include "opencv2/highgui.hpp"

#include "/home/pi/raspicam-0.1.3/src/raspicam_cv.h"

#include "opencv2/xphoto.hpp"

#include <cctype>

#include <stdio.h>

#include <string.h>

#include <time.h>
```

Deberemos modificar los archivos de VID5 e IN_VID5, en VID5 asignaremos la dirección donde nos encontramos y en el IN_VID5 tendremos que indicar el tamaño del cuadrado o del grid, además de indicar si la entrada será tomada por cámara, desde el VID5.xml, desde video o desde cámara.

VID5

```
<?xml version="1.0"?>

<opencv_storage>

<images>

/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto1.jpg
```

```
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto2.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto3.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto4.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto5.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto6.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto7.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto8.jpg
</images>
</opencv_storage>
```

In_VID5

```
<?xml version="1.0"?>
<opencv_storage>
<Settings>
<!-- Number of inner corners per a item row and column. (square, circle) -->
<BoardSize_Width>8</BoardSize_Width>
<BoardSize_Height>6</BoardSize_Height>
<!-- The size of a square in some user defined metric system (pixel, millimeter)-->
<Square_Size>106</Square_Size>
<!-- The type of input used for camera calibration. One of: CHESSBOARD CIRCLES_GRID
ASYMMETRIC_CIRCLES_GRID -->
<Calibrate_Pattern>"CHESSBOARD"</Calibrate_Pattern>
<!-- The input to use for calibration.
                To use an input camera -> give the ID of the camera, like "1"
                To use an input video  -> give the path of the input video, like "/tmp/x.avi"
                To use an image list   -> give the path to the XML or YAML file containing the
list of the images, like "/tmp/circles_list.xml"-->
<Input>"VID5.xml"</Input>
<!--  If true (non-zero) we flip the input images around the horizontal axis.-->
<Input_FlipAroundHorizontalAxis>0</Input_FlipAroundHorizontalAxis>
<!--  Time delay between frames in case of camera.  -->
<Input_Delay>100</Input_Delay>
<!--  How many frames to use, for calibration.  -->
```

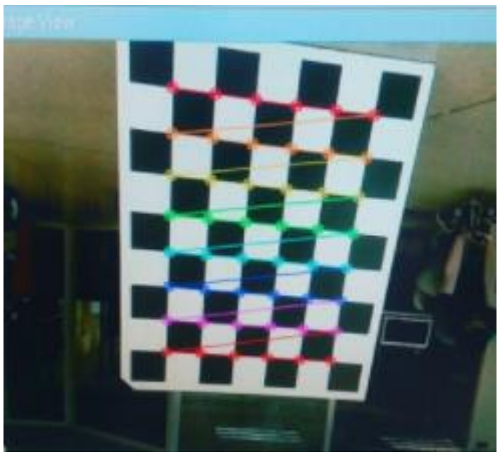
```
<Calibrate_NrOfFrameToUse>25</Calibrate_NrOfFrameToUse>

<!-- Consider only fy as a free parameter, the ratio fx/fy stays the same as in the input cameraMatrix.

            Use or not setting. 0 - False Non-Zero - True-->

<Calibrate_FixAspectRatio>1</Calibrate_FixAspectRatio>

<!-- If true (non-zero) tangential distortion coefficients  are set to zeros and stay zero.-->

<Calibrate_AssumeZeroTangentialDistortion>1</Calibrate_AssumeZeroTangentialDistortion>

<!-- If true (non-zero) the principal point is not changed during the global optimization.-->

<Calibrate_FixPrincipalPointAtTheCenter>1</Calibrate_FixPrincipalPointAtTheCenter>

<!--  The name of the output log file.  -->

<Write_outputFileName>"out_camera_data.xml"</Write_outputFileName>

<!-- If true (non-zero) we write to the output file the feature points.-->

<Write_DetectedFeaturePoints>1</Write_DetectedFeaturePoints>

<!-- If true (non-zero) we write to the output file the extrinsic camera parameters.-->

<Write_extrinsicParameters>1</Write_extrinsicParameters>

<!-- If true (non-zero) we show after calibration the undistorted images.-->

<Show_UndistortedImage>1</Show_UndistortedImage>

</Settings>

</opencv_storage>
```

En este caso usaremos la entrada desde cámara, debido a problemas de administración en la Raspberry pi facilitada, esto tampoco supone ningún problema para el desarrollo del proyecto.

Montaríamos el entorno visual de trabajo, iniciaríamos la captura de imágenes, indicaremos que se va a utilizar, en este caso, la calibración por tablero de ajedrez.

Hallaríamos las esquinas del tablero, como se indica en el código y en la imagen siguiente.

```
enum { DETECTION = 0, CAPTURING = 1, CALIBRATED = 2 };

enum Pattern { CHESSBOARD, CIRCLES_GRID, ASYMMETRIC_CIRCLES_GRID };

static double computeReprojectionErrors(

    const vector<vector<Point3f> >& objectPoints,

    const vector<vector<Point2f> >& imagePoints,

    const vector<Mat>& rvecs, const vector<Mat>& tvecs,

    const Mat& cameraMatrix, const Mat& distCoeffs,
    vector<float>& perViewErrors )
{
    vector<Point2f> imagePoints2;
    int i, totalPoints = 0;
    double totalErr = 0, err;
    perViewErrors.resize(objectPoints.size());
    for( i = 0; i < (int)objectPoints.size(); i++ )
    {
    projectPoints(Mat(objectPoints[i]), rvecs[i], tvecs[i],
                  cameraMatrix, distCoeffs, imagePoints2);
    err = norm(Mat(imagePoints[i]), Mat(imagePoints2), NORM_L2);
    int n = (int)objectPoints[i].size();
    perViewErrors[i] = (float)std::sqrt(err*err/n);
    totalErr += err*err;
    totalPoints += n;
    }
    return std::sqrt(totalErr/totalPoints);
}
static void calcChessboardCorners(Size boardSize, float squareSize, vector<Point3f>& corners,
Pattern patternType = CHESSBOARD)
{
    corners.resize(0);
    switch(patternType)
    {
      case CHESSBOARD:
      case CIRCLES_GRID:
    for( int i = 0; i < boardSize.height; i++ )
        for( int j = 0; j < boardSize.width; j++ )
            corners.push_back(Point3f(float(j*squareSize),
                                      float(i*squareSize), 0));
    break;
      case ASYMMETRIC_CIRCLES_GRID:
    for( int i = 0; i < boardSize.height; i++ )
        for( int j = 0; j < boardSize.width; j++ )
            corners.push_back(Point3f(float((2*j + i % 2)*squareSize),
                                      float(i*squareSize), 0));
    break;
      default:
    CV_Error(Error::StsBadArg, "Error en el patrón\n");
    }
}
static bool runCalibration( vector<vector<Point2f> > imagePoints,
                    Size imageSize, Size boardSize, Pattern patternType,
                    float squareSize, float aspectRatio,
                    int flags, Mat& cameraMatrix, Mat& distCoeffs,
                    vector<Mat>& rvecs, vector<Mat>& tvecs,
                    vector<float>& reprojErrs,
                    double& totalAvgErr)
{
    cameraMatrix = Mat::eye(3, 3, CV_64F);
    if( flags & CALIB_FIX_ASPECT_RATIO )
    cameraMatrix.at<double>(0,0) = aspectRatio;
    distCoeffs = Mat::zeros(8, 1, CV_64F);
    vector<vector<Point3f> > objectPoints(1);
    calcChessboardCorners(boardSize, squareSize, objectPoints[0], patternType);
    objectPoints.resize(imagePoints.size(),objectPoints[0]);
    double rms = calibrateCamera(objectPoints, imagePoints, imageSize, cameraMatrix,
                    distCoeffs, rvecs, tvecs, flags|CALIB_FIX_K4|CALIB_FIX_K5);

    printf("RMS error reportado por calibrateCamera: %g\n", rms);
    bool ok = checkRange(cameraMatrix) && checkRange(distCoeffs);
    totalAvgErr = computeReprojectionErrors(objectPoints, imagePoints,
            rvecs, tvecs, cameraMatrix, distCoeffs, reprojErrs);
   return ok;
}
```



Una vez tomadas las capturas tendremos que guardar los datos de distorsión, calibración, los vectores de rotación, etc en un archivo .txt, se tendrá que hacer esto tanto con los datos extrínsecos como con los datos de la cámara. Todo esto será en que cargaremos en un futuro en nuestro código principal.

```
static void saveCameraParams( const string& filename,
                 Size imageSize, Size boardSize,
                 float squareSize, float aspectRatio, int flags,
                 const Mat& cameraMatrix, const Mat& distCoeffs,
                 const vector<Mat>& rvecs, const vector<Mat>& tvecs,
                 const vector<float>& reprojErrs,
                 const vector<vector<Point2f> >& imagePoints,
                 double totalAvgErr )
{
   FileStorage fs( filename, FileStorage::WRITE );
   time_t tt;
   time( &tt );
   struct tm *t2 = localtime( &tt );
   char buf[1024];
   strftime( buf, sizeof(buf)-1, "%c", t2 );
   fs << "calibration_time" << buf;
```

```
if( !rvecs.empty() || !reprojErrs.empty() )
    fs << "nframes" << (int)std::max(rvecs.size(), reprojErrs.size());
fs << "image_width" << imageSize.width;
fs << "image_height" << imageSize.height;
fs << "board_width" << boardSize.width;
fs << "board_height" << boardSize.height;
fs << "square_size" << squareSize;
if( flags & CALIB_FIX_ASPECT_RATIO )
    fs << "aspectRatio" << aspectRatio;
if( flags != 0 )
{
    sprintf( buf, "flags: %s%s%s%s",
        flags & CALIB_USE_INTRINSIC_GUESS ? "+use_intrinsic_guess" : "",
        flags & CALIB_FIX_ASPECT_RATIO ? "+fix_aspectRatio" : "",
        flags & CALIB_FIX_PRINCIPAL_POINT ? "+fix_principal_point" : "",
        flags & CALIB_ZERO_TANGENT_DIST ? "+zero_tangent_dist" : "" );
}
fs << "flags" << flags;
fs << "rvecs" << rvecs;
fs << "tvecs" << tvecs;
fs << "camera_matrix" << cameraMatrix;
fs << "distortion_coefficients" << distCoeffs;
fs << "avg_reprojection_error" << totalAvgErr;
if( !reprojErrs.empty() )
    fs << "per_view_reprojection_errors" << Mat(reprojErrs);
if( !rvecs.empty() && !tvecs.empty() )
{
    CV_Assert(rvecs[0].type() == tvecs[0].type());
    Mat bigmat((int)rvecs.size(), 6, rvecs[0].type());
    for( int i = 0; i < (int)rvecs.size(); i++ )
    {
            Mat r = bigmat(Range(i, i+1), Range(0,3));
            Mat t = bigmat(Range(i, i+1), Range(3,6));
            CV_Assert(rvecs[i].rows == 3 && rvecs[i].cols == 1);
            CV_Assert(tvecs[i].rows == 3 && tvecs[i].cols == 1);
            r = rvecs[i].t();
            t = tvecs[i].t();
        }
        //fs << "extrinsic_parameters" << bigmat;
    }
    if( !imagePoints.empty() )
    {
        Mat imagePtMat((int)imagePoints.size(), (int)imagePoints[0].size(), CV_32FC2);
        for( int i = 0; i < (int)imagePoints.size(); i++ )
        {
            Mat r = imagePtMat.row(i).reshape(2, imagePtMat.cols);
            Mat imgpti(imagePoints[i]);
            imgpti.copyTo(r);
        }
        fs << "image_points" << imagePtMat;
    }
}
static bool runAndSave(const string& outputFilename,
                const vector<vector<Point2f> >& imagePoints,
                Size imageSize, Size boardSize, Pattern patternType, float squareSize,
                float aspectRatio, int flags, Mat& cameraMatrix,
                Mat& distCoeffs, bool writeExtrinsics, bool writePoints )
{
    vector<Mat> rvecs, tvecs;
    vector<float> reprojErrs;
    double totalAvgErr = 0;
    bool ok = runCalibration(imagePoints, imageSize, boardSize, patternType, squareSize,
              aspectRatio, flags, cameraMatrix, distCoeffs,
              rvecs, tvecs, reprojErrs, totalAvgErr);
   printf("%s. avg reprojection error = %.2f\n",
        ok ? "Calibration succeeded" : "Calibration failed",
        totalAvgErr);
   if( ok )
      saveCameraParams( outputFilename, imageSize,
                  boardSize, squareSize, aspectRatio,
                  flags, cameraMatrix, distCoeffs,
                  writeExtrinsics ? rvecs : vector<Mat>(),
                  writeExtrinsics ? tvecs : vector<Mat>(),
                  writeExtrinsics ? reprojErrs : vector<float>(),
                  writePoints ? imagePoints : vector<vector<Point2f> >(),
                  totalAvgErr );
   return ok;
}
```

5.2.2 Calibración de dos cámaras.

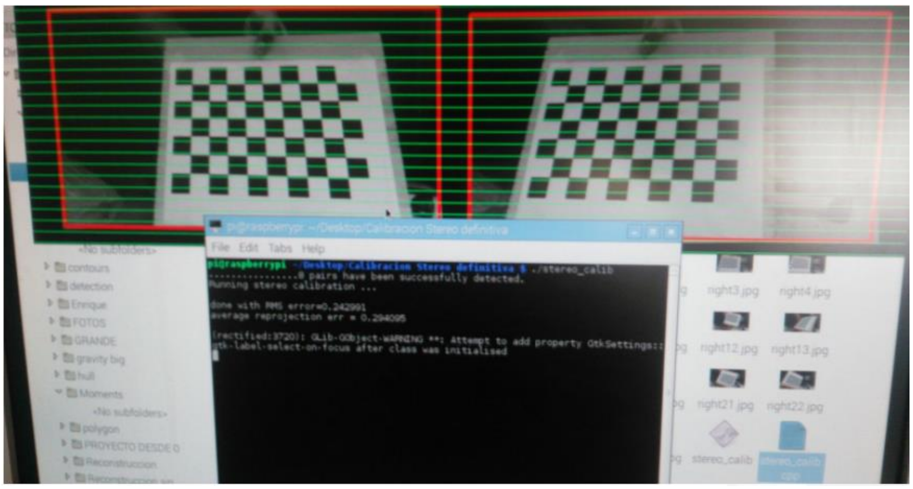
Una vez familiarizados con la calibración de las cámaras y de su entorno, al tratarse de un proyecto en el cual usaremos visión 3D, lo que viene a decir, que usaremos dos cámaras para realizar el proyecto, tendremos que hacer las modificaciones del código oportunas para esta calibración estéreo.

Podréis encontrar todos los códigos en los anexos.

Para realizar la calibración de las dos cámaras en una situación normal deberíamos utilizar el código de Stereo Calib, facilitado en los tutoriales.

Siguiendo el código del tutorial necesitaremos usar un conjunto de pares de imágenes tomadas previamente.

Al ejecutar podremos ver lo siguiente:



```
pi@raspberrypi ~/Desktop/Calibracion Stereo definitiva $ ./stereo_calib
..................8 pairs have been successfully detected.
Running stereo calibration ...

done with RMS error=0.242991
average reprojection err = 0.294095

(rectified:3720): GLib-GObject-WARNING **: Attempt to add property GtkSettings::
gtk-label-select-on-focus after class was initialised
```

Además de mostrarnos esto en pantalla nos creará dos archivos Yml, extrinsics.yml e intrinsics.yml. En los cuales nos encontraremos con todos los datos y matrices intrinsicas y extrínsecas del sistema.

Pero volviendo a lo que viene a ser nuestro proyecto, al no tener disponibilidad de tomar usar dos cámaras simultáneamente conectadas con un multiplexor (no dispongo de dichas cámaras de ni de dicho conector) se ha decidido "falsear" el uso de una segunda cámara, simplemente calibrando ambas cámaras por separado, pero siempre con coherencia entre ellas.

Realizaremos el proceso realizado durante la calibración normal, únicamente cambiará que la última fotografía tomada por ambas cámaras debe hacer referencia a la misma posición del patrón (por eso mismo hemos usado dos patrones en vez de uno) y además tenemos que tener en cuenta que las cámaras van a tener que tener una distancia entre ellas siempre idéntica en cada captura.

Una vez realizada la calibración por separada nos encontramos con esta información dentro de los archivos .yml, la cual será utilizada más adelante.

Ejemplo de .Yml con datos de la calibración.

```
%YAML:1.0

calibration_time: "Mon 25 Jul 2016 02:01:14 AM UTC"

nframes: 10

image_width: 640

image_height: 480

board_width: 6

board_height: 4

square_size: 1.

flags: 0

rvecs: !!opencv-matrix

   rows: 3
```

```
    cols: 1
    dt: d
    data: [ -3.8168970275374370e-01, -4.8192551187908084e-01,
       -1.5304248482209646e+00 ]
tvecs: !!opencv-matrix
    rows: 3
    cols: 1
    dt: d
    data: [ -1.0744150845693301e+00, 2.1240715234552567e+00,
       2.8371457784444260e+01 ]
camera_matrix: !!opencv-matrix
  rows: 3
  cols: 3
  dt: d
  data: [ 6.8710141544645626e+02, 0., 3.2535282067810499e+02, 0.,
     6.8611790088889813e+02, 2.0853874919724078e+02, 0., 0., 1. ]
distortion_coefficients: !!opencv-matrix
  rows: 5
  cols: 1
  dt: d
  data: [ 2.3791990077953137e-01, -3.5714658240184600e+00,
     -1.2166589443379566e-02, 2.6743818419039467e-03,
     5.7016194116800996e+01 ]
avg_reprojection_error: 2.2263853364706571e-01
per_view_reprojection_errors: !!opencv-matrix
  rows: 10
  cols: 1
  dt: f
  data: [ 1.68970317e-01, 2.40029961e-01, 4.01509732e-01,
     1.37740389e-01, 1.68004587e-01, 1.81966394e-01, 1.73426762e-01,
     1.91365048e-01, 1.73505127e-01, 2.66819388e-01 ]
```

```
extrinsic_parameters: !!opencv-matrix
  rows: 10
  cols: 6
  dt: d
  data: [ -9.8573068130552888e-02, -8.7568302636586284e-02,
     -3.9696330880263243e-03, -2.1757372852312975e+00,
     -1.9222890132714814e+00, 2.1196499153310953e+01,
     -2.9087477435170145e-01, -2.3391674116166467e-01,
     -3.3950972981014675e-02, -2.1887390638730979e+00,
     -2.0017390668337538e+00, 2.1200366522593878e+01,
     -3.8398009512376063e-01, -7.3737672307561553e-01,
     -1.5419123528674505e-02, -2.1493507993355458e+00,
     -2.0755826790112106e+00, 2.1541994878464092e+01,
     -3.5521193145672370e-02, 1.6123742831161098e-01,
     -5.8717488903129895e-02, -2.9052324263964109e+00,
     -3.6538704723292397e+00, 2.3073118487507777e+01,
     -1.1613194921674712e-01, 1.0083027741218353e-01,
     -6.5834783879958969e-01, -2.8319651074328847e+00,
     -2.3219833400725514e+00, 2.4202544876030004e+01,
     -1.4968909667285557e-01, -3.7452870173521759e-02,
     -9.7485228830145931e-01, -1.8433762950810721e+00,
     -1.3892471509543285e+00, 2.4975117594837414e+01,
     1.5120718598786453e-01, -3.6657176487751847e-01,
     -1.4838233295599184e+00, -7.6949959294334092e-01,
     -7.3619011681702173e-01, 2.5122754231550978e+01,
     -2.8687796716336217e-01, 3.1595699508440483e-01,
     1.5200960477824692e+00, 2.7914444424586877e+00,
     -4.4067979247695011e+00, 2.7150979398001542e+01,
     -2.9698080685722567e-01, 3.3591252546252337e-01,
     1.5071490659616467e+00, 2.9400380357224938e+00,
```

-3.3205098200001930e+00, 2.8688795449130147e+01,

-3.8168970275374370e-01, -4.8192551187908084e-01,

-1.5304248482209646e+00, -1.0744150845693301e+00,

2.1240715234552567e+00, 2.8371457784444260e+01]

5.3 Procesamiento de imagen

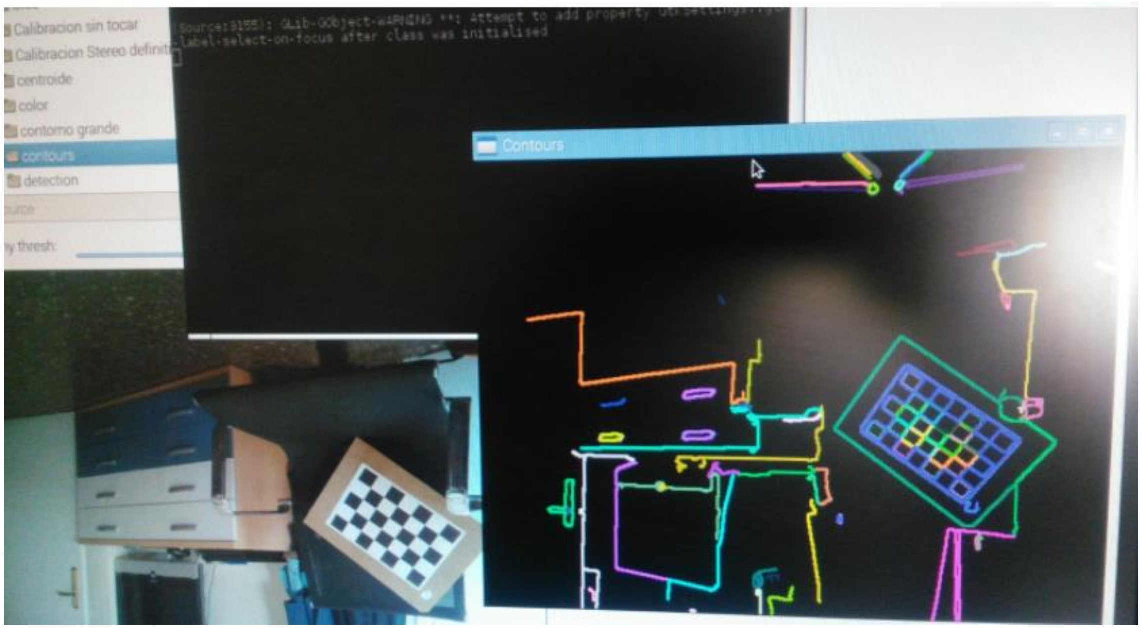
Una vez realizada la calibración y conocidos sus datos y matrices de calibración deberemos procesar las imágenes.

Para ello tendremos que saber de qué manera vamos a enfocar la localización de objetos. Como anteriormente se ha explicado, en el apartado de seguimiento de objetos, hay muchas formas de localizar objetos, así que inicialmente las dos primeras que se me ocurrieron fueron; por color y por contornos.

Al no tener claro de que objeto se iba a tratar el que se debía seguir inicialmente decidí descartar la opción por color, y me decanté por contornos.

Siguiendo el tutorial conseguí dar con un ejemplo de localización de contornos en una fotografía (adjunto código modificado en anexos).

Gracias a este código contours.cpp pude localizar varios contornos en las imagen de muestra.

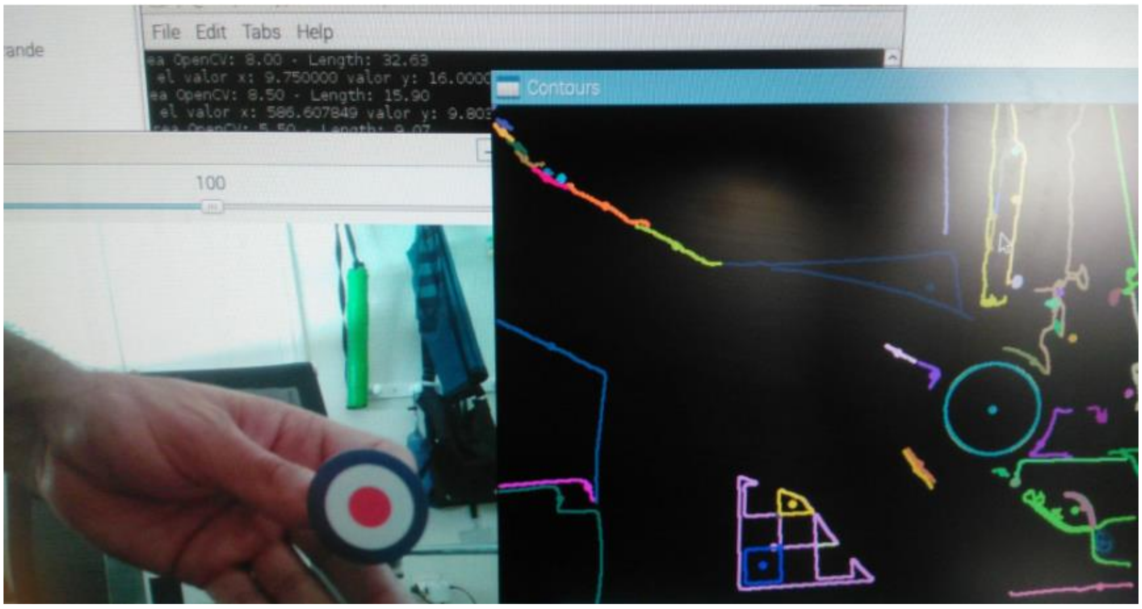


```
Source:3155): GLib-GObject-WARNING **: Attempt to add property
label-select-on-focus after class was initialised
```

Pero claro, aquí hay mucha información que no es de interés, y si el objeto se va a encontrar en movimiento decidí que no solo debería basarme en el contorno del mismo, si no también debería usar el centroide de este contorno para localizar de una forma más sencilla los puntos homogéneos entre ambas imágenes.

Así que me puse a la labor de encontrar estos centróides de los contornos.

Partiendo del código moments.cpp, con el cual podemos hallar los momentos de estos contornos, y por consecuencia calcular los centroides de los mismos nos encontramos con el siguiente resultado.

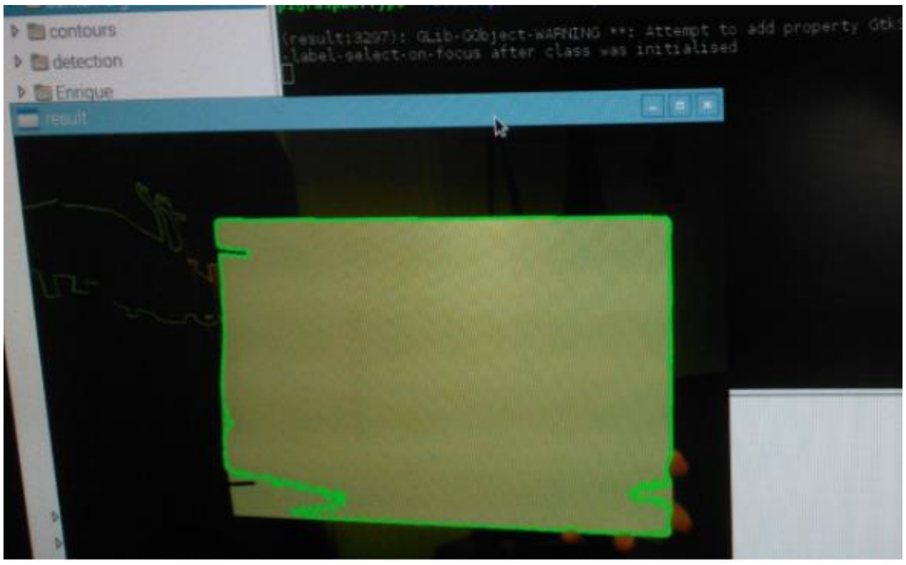


```
ea OpenCV: 8.00 - Length: 32.63
 el valor x: 9.750000 valor y: 16.0000
ea OpenCV: 8.50 - Length: 15.90
 el valor x: 586.607849 valor y: 9.803
rea OpenCV: 5.50 - Length: 9.07
```

En este caso intenté probar a localizar el centroide de un circulo, pero lógicamente al no haber optimizado el sistema no solo encontraba el centroide de dicho circulo, si no que encontraba los centroides de todos los contornos de la imagen, lo cual podría crear mucha confusión a la hora de trabajar con puntos homogéneos.

Así que decidí dar un paso atrás y modificar el código de contornos, para que en su lugar, en vez de localizar todos los contornos únicamente localizase uno.

Utilizando el código de big_contour únicamente encontré el contorno mayor, el más grande.



```
(result:3297): GLib-GObject-WARNING **: Attempt to add property Gtk
-label-select-on-focus after class was initialised
```

Mezclando ambos conceptos podemos empezar con la localización del punto homogéneo en dos imágenes.

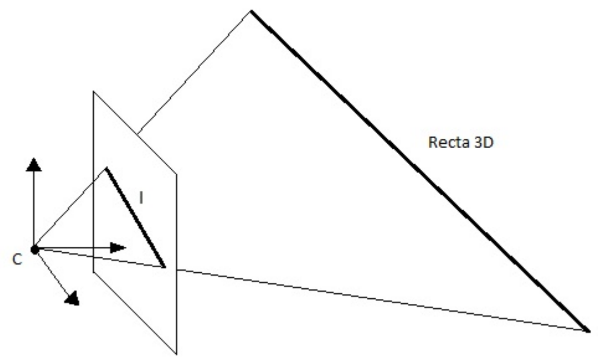
5.4 Localización y seguimiento

Una vez ya conocemos como podríamos hallar el punto centro de un cortorno deseado nos disponemos a trabajar con él, para poder hallar los puntos homogéneos y con ello cumplir un caso de genérico de localización de objetos.

Este caso genérico será explicado a continuación.

5.4.1 Caso genérico

- La proyección de una recta 3D es una recta en la imagen
- El centro óptico y la recta 3D definen el plano proyectante de la recta
- Partiendo de la recta en la imagen determinarse el plano proyectante



Ecuación plano π en 3D

AX+BY+CZ+D=0

$$\begin{pmatrix} A & B & C & D \end{pmatrix} \times \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = 0$$

Luego podemos representar un plano π mediante un vector de 4 filas y 1 columna.

$$\pi = [A\ B\ C\ D]^T$$

Ecuación recta **"l"** en el plano 2D Ax+By+C=0, luego en coordenadas homogéneas

$[A\ B\ C]\begin{matrix} x \\ y \\ 1 \end{matrix}=0,$

Luego podemos representar una recta un vector de 3 filas y 1 columna

l=$[A\ B\ C]^T$

sea un punto 3D **X** su imagen por una cámara P es:

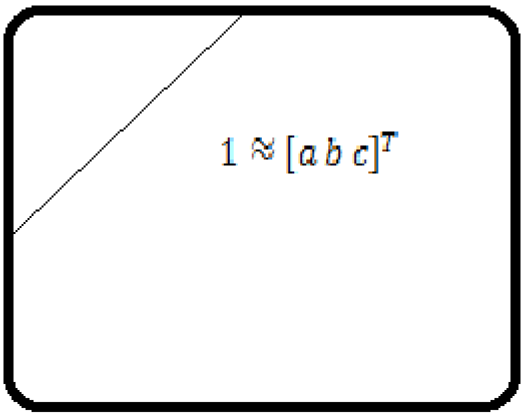
x=λPX

xϵ a la recta l⟷l^Tx=0⟷λl^TPX=0 luego, la ecuación del plano proyectante π=l^TP

Sea una recta, coordenadas pixélicas,
$1 \approx [a\ b\ c]^T$ en la imagen de una cámara P.

El plano proyectante π, definido por el centro óptico y la recta en la imagen es:

$\pi \approx [a\ b\ c]P$



Sea X un punto 3D, su imagen, x, por la cámara P

X=PC, x∊1↔1^T x=0↔1^TPX=0↔X∊ al plano proyectante, ↔1^TP≈ π

Sea X≈ $[u\ v\ 1]^T$, en pixélicas, la imagen de un punto a través de la cámara P.

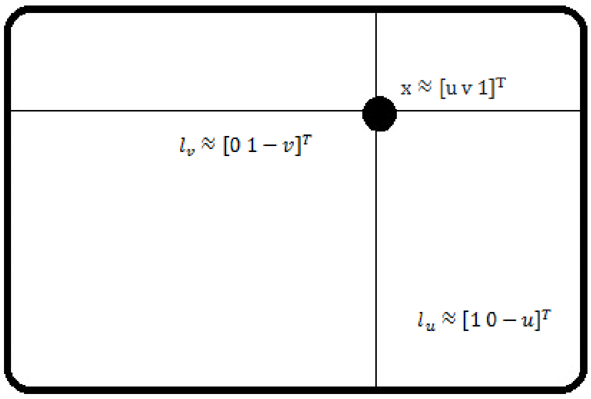
En la imagen, el punto x, puede expresarse con intersección de 2 rectas

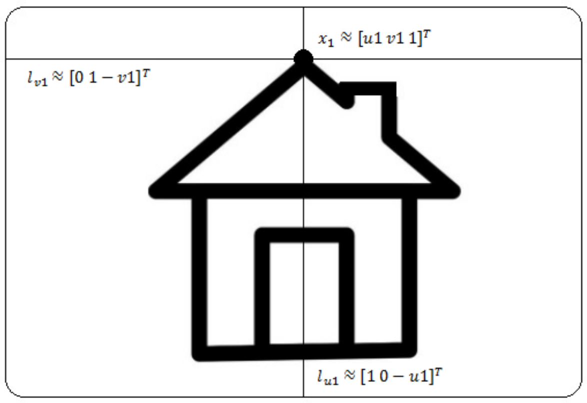
$$l_u \approx [1\ 0 - u]^T$$
$$l_v \approx [0\ 1 - v]^T$$

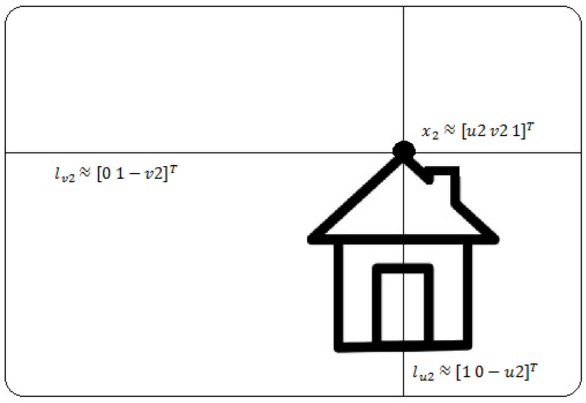
El rayo proyectante en 3D correspondiente a x puede definirse por los 2 planos proyectantes correspondientes

$[1\ 0 - u]^T$ P

$[0\ 1 - v]^T$ P




$x_1 \approx [u1\ v1\ 1]^T$
$l_{v1} \approx [0\ 1 - v1]^T$
$l_{u1} \approx [1\ 0 - u1]^T$


$x_2 \approx [u2\ v2\ 1]^T$
$l_{v2} \approx [0\ 1 - v2]^T$
$l_{u2} \approx [1\ 0 - u2]^T$

Sean

2 cámaras P1, P2

(u1,v1) y (u2,v2) las coordenadas pixélicas de un punto en ambas cámaras

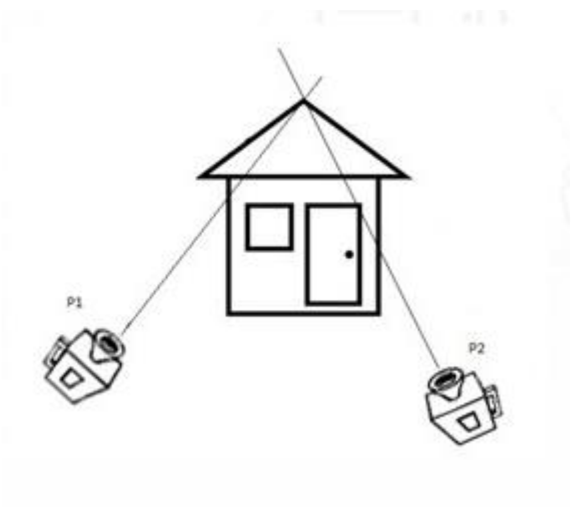
Las coordenadas homogéneas del punto en 3D son la solución del sistema homogéneo

AX=0

$$A=\begin{bmatrix}[1\ 0\ -u1]P1\\ [0\ 1\ -v1]P1\\ [1\ 0\ -u2]P2\\ [0\ 1\ -u2]P2\end{bmatrix}$$

Sólo tiene solución única si rango(A)=3. X λv4

[U,S,V]=svd(A) V=[v1 v2 v3 v4]



$$l_{v1} \approx [0\ 1-v1]^T$$

$$l_{u1} \approx [1\ 0-u1]^T$$

$$l_{v2} \approx [0\ 1-v2]^T$$

$$l_{u2} \approx [1\ 0-u2]^T$$

$$x_2 \approx [u2\ v2\ 1]^T$$

$$x_1 \approx [u1\ v1\ 1]^T$$

En lo que viene a ser el código principal de esta parte, inicialmente leeremos las matrices y datos de ambas cámaras.

Para eso usaremos un Filestorage, un ejemplo en el siguiente código.

```
FileStorage fs2;
 fs2.open("outbueno.yml", FileStorage::READ);
 if (!fs2.isOpened())
   {
     cout << "failed to open"  << endl;
     return 1;
   }
 Mat P;
 Mat tvecs;
 Mat rvecs;
 fs2["camera_matrix"] >> P;
 fs2["rvecs"] >> rvecs;
 fs2["tvecs"] >> tvecs;
 cout << "camera matrix: " << P << endl;
 cout << "rvecs: " << rvecs << endl;
 cout << "tvecs: " << tvecs << endl;
fs2.release();
```

Una vez tenemos todos los datos de interés conseguidos con la calibración de la cámara tendremos que conseguir las matrices de rotación, para ello usaremos los datos de los vectores de rotación e implementaremos Rodrigues.

```
Mat rotmat;
 Mat rotmat1;
 Rodrigues(rvecs ,rotmat);
 Rodrigues(rvecs1 ,rotmat1);
cout << "rotmat: " << rotmat << endl;
cout << "rotmat1: " << rotmat1 << endl;
```

Con las matrices de rotación ya halladas el siguiente paso será abrir las imágenes.

```
Mat src = imread( argv[1] );
```

Y hallaremos los contornos y el área mayor, con esto podríamos hallar el contorno mayor de todos, que es del cual queremos hallar su centroide.

```
   findContours( thr, contours, RETR_CCOMP, CHAIN_APPROX_SIMPLE ); //encontrar el
contorno de la imagen
   for( size_t i = 0; i< contours.size(); i++ ) // itinerar a través de cada contorno
   {
       double area = contourArea( contours[i] );  //  encontrar el área del contorno
       if( area > largest_area )
       {
           largest_area = area;
           largest_contour_index = i;             //almacenar el índice del contorno mayor
       }
```

El siguiente paso sería hallar los momentos, con los cuales podremos calcular el centro de masas.

```
vector<Moments> mu(contours.size() );
 for( size_t i = 0; i < contours.size(); i++ )
    { mu[i] = moments( contours[i], false ); }
vector<Point2f> mc( contours.size() );
```

```
 for( size_t i = 0; i < contours.size(); i++ )
    { mc[i] = Point2f( static_cast<float>(mu[i].m10/mu[i].m00) ,
static_cast<float>(mu[i].m01/mu[i].m00) ); }
```

Crearíamos la matriz de extrínsecos.

```
 Mat Mext(3,4,CV_64FC1);
 Mext.at<double>(0,0)=rotmat.at<double>(0,0);
 Mext.at<double>(0,1)=rotmat.at<double>(0,1);
 Mext.at<double>(0,2)=rotmat.at<double>(0,2);
 Mext.at<double>(0,3)=tvecs.at<double>(0,0);
 Mext.at<double>(1,0)=rotmat.at<double>(1,0);
 Mext.at<double>(1,1)=rotmat.at<double>(1,1);
 Mext.at<double>(1,2)=rotmat.at<double>(1,2);
 Mext.at<double>(1,3)=tvecs.at<double>(0,1);
 Mext.at<double>(2,0)=rotmat.at<double>(2,0);
 Mext.at<double>(2,1)=rotmat.at<double>(2,1);
 Mext.at<double>(2,2)=rotmat.at<double>(2,2);
 Mext.at<double>(2,3)=tvecs.at<double>(0,2);
```

Calculamos P (lo llamaremos M) y P1 (lo llamaremos M1)

```
 Mat M(3,4,CV_64FC1);
 for(int i=0;i<3;i++){
   for(int j=0;j<4;j++){
           M.at<double>(i,j)=0;
           for(int k=0;k<3;k++){
             M.at<double>(i,j)=M.at<double>(i,j)+(P.at<double>(i,k)*Mext.at<double>(k,j));
        }
     }
 }
 cout << "M " << M << endl;


 Mat M1(3,4,CV_64FC1);
 for(int i=0;i<3;i++){
  for(int j=0;j<4;j++){
            M1.at<double>(i,j)=0;
            for(int k=0;k<3;k++){
              M1.at<double>(i,j)=M1.at<double>(i,j)+(P1.at<double>(i,k)*Mext1.at<double>(k,j));
        }
     }
 }
 cout << "M1 " << M1 << endl;
```

Calcularíamos la Matriz A, podéis ver el detalle de los cálculos en los códigos anexos. Ponemos un ejemplo unicamente

```
// v1 x M
 Mat A1(1,4,CV_64FC1);
 for(int f=0;f<1;f++){
   for(int c=0;c<4;c++){
     A1.at<double>(f,c)=0;
            for(int h=0;h<3;h++){
              A1.at<double>(f,c)=A1.at<double>(f,c)+(v1.at<double>(f,h)*M.at<double>(h,c));
        }
   }
 }
 cout << "A1 " << A1 << endl;
 cout<< "v1 (2.0)="  <<v1.at<double>(0,2)<<endl;
```

Continuaríamos con la descomposición en valores singulares (svd) de la matriz A

```
Mat  w, u, vt;
 SVD::compute(A,w,u,vt);
cout << "w " << w << endl;
cout << "u " << u << endl;
cout << "vt " << vt << endl;
```

y hallaremos X

```
Mat X(4,1,CV_64FC1);
 X.at<double>(0,0)=vt.at<double>(3,0)/vt.at<double>(3,3);
```

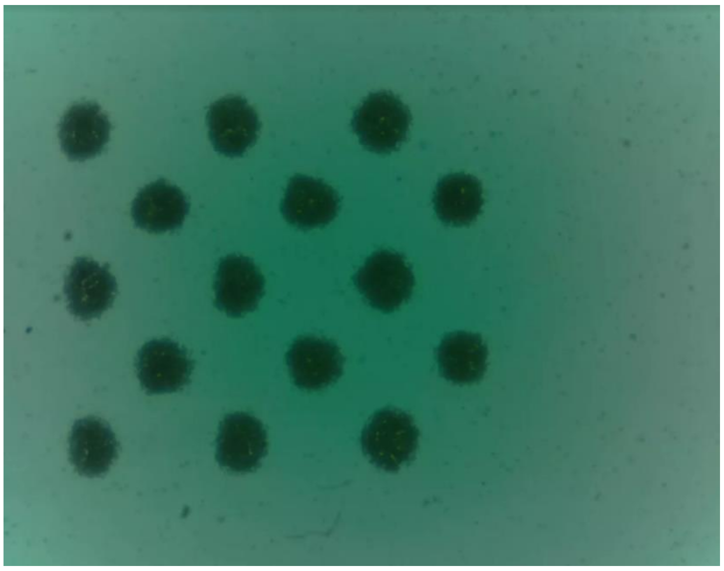
```
X.at<double>(1,0)=vt.at<double>(3,1)/vt.at<double>(3,3);

X.at<double>(2,0)=vt.at<double>(3,2)/vt.at<double>(3,3);

X.at<double>(3,0)=vt.at<double>(3,3)/vt.at<double>(3,3);
```

Después de esto ya tendríamos el valor de X, para comprobar que X es el correcto nos inventaremos un nuevo valor, calcularíamos de nuevo la A y nos tendría que dar de nuevo los nuevos centros de homogéneos y la X.

Los cálculos de esto los podrán encontrar al final del código *grance.cpp* comentado.

5.4.2 Seguimiento: filtro de Kalman

Por último realizaremos el seguimiento, para ello aplicaremos un filtro de Kalman, ya que consideraremos que nuestro sistema es lineal y que las distorsiones son gaussianas, se utilizará este método porque considero que será rápido y efectivo. Lo utilizaremos para filtrar la distorsión en nuestro sistema y, poder así, hacer un seguimiento más eficiente.



Inicialmente en el proyecto utilizamos un patrón de tablero de ajedrez, pero ahora tendremos que utilizar un *grid*, ya que finalmente los objetos a seguir serán c.elegans, una especie de pequeños gusanos de tamaño muy reducido, y con el *grid* reduciremos la distorsión para este caso.

Por otro lado, y para optimizar el proceso en este caso específico no utilizaremos la Rapsberry Pi, usaremos las librerías de OpenCV en un ordenador, esto es debido a que las imágenes tomadas son demasiado grandes, y a la raspberry pi le costaría demasiado poder trabajar con ellas.

Finalmente, no dispusimos del equipo necesario para tomar las imágenes de los C.elegans y trabajar con ellos, pero explicaremos de forma teórica y con un ejemplo cómo funcionaría el filtro de Kalman.

A continuación podéis ver como pasarían

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

double frand() {

   return 2*((rand()/(double)RAND_MAX) - 0.5);

}
```

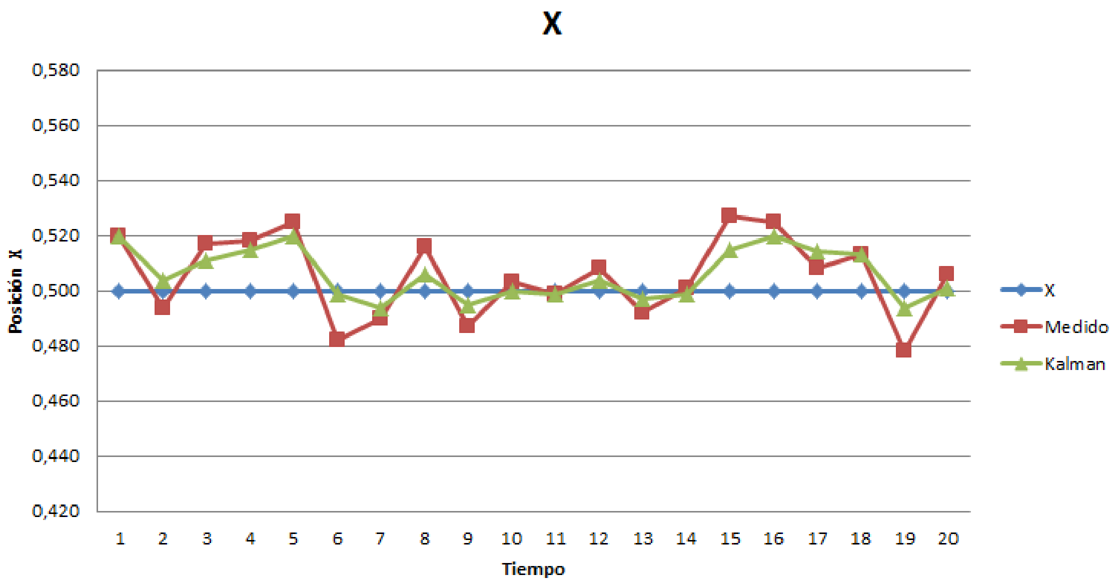
```
int main() {
   float z_real_x = 0.5; //El valor real x que queremos medir
   float z_real_y = 1; //El valor real y que queremos medir
   float z_real_z = 0; //El valor real z que queremos medir

   //Valores iniciales para el filtro de Kalman (simularemos gráficamente como si el primer
valor medido fuese el primer valor de Kalman)
   float x_est_last = z_real_x + frand()*0.03;
   float y_est_last = z_real_y + frand()*0.03;
   float z_est_last = z_real_z + frand()*0.03;
   float P_last _x= z_real_x + frand()*0.03;
   float P_last _y= z_real_y + frand()*0.03;
   float P_last _z= z_real_z + frand()*0.03;
   //Ruido en el sistema
   float Q = 0.4;
   float R = 0.6;
   float K;
   float P;
   float P_temp;
   float x_temp_est;
   float x_est;
   float z_measured; //el valor del ruido que medimos
   srand(0);
   //////////////////XXXXXXXXXXXXXXXXXXXXX/////////////////
   //inicializar la medida
   x_est_last = z_real_x + frand()*0.03;
   float sum_error_kalman = 0;
   float sum_error_measure = 0;
      printf("X Original: %6.3f \n",z_real_x);
      printf("Medido: %6.3f [diferencia:0]\n",x_est_last);
      printf("Kalman: %6.3f [diferencia:0]\n", x_est_last);
```

```
  for (int i=0;i<20;i++) {
    //Predicción
    x_temp_est = x_est_last;
    P_temp = P_last_x + Q;
    //Calculamos la ganancia de Kalman
    K = P_temp * (1.0/(P_temp + R));
    //Medido
    z_measured = z_real_x + frand()*0.03; //Valor real+ruido
    //Correcto
    x_est = x_temp_est + K * (z_measured - x_temp_est);
    P = (1- K) * P_temp;
    //Tenemos nuestro nuevo sistema
    printf("X Original: %6.3f \n",z_real_x);
    printf("Medido: %6.3f [diferencia:%.3f]\n",z_measured,fabs(z_real_x-z_measured));
    printf("Kalman: %6.3f [diferencia:%.3f]\n",x_est,fabs(z_real_x - x_est));
    sum_error_kalman += fabs(z_real_x - x_est);
    sum_error_measure += fabs(z_real_x-z_measured);
    //Actualizamos los "last"
    P_last_x = P;
    x_est_last = x_est;
  }
  printf("Error Total sin kalman:  %f\n",sum_error_measure);
  printf("Error total con kalman: %f\n",sum_error_kalman);
  printf("Porcentaje de error reducido: %d%% \n",100-
(int)((sum_error_kalman/sum_error_measure)*100));
  return 0;
}
```

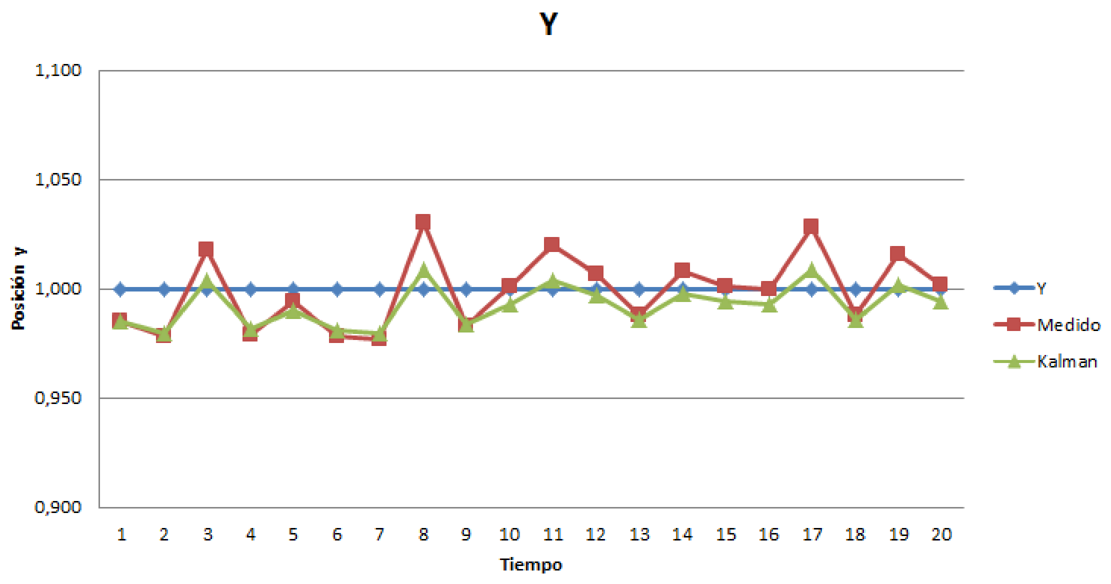
Tendríamos que continuar con el mismo proceso anterior con el valor de “Y” y el valor de “X”.

A continuación podremos ver unas gráficas bastante detalladas, realizadas con Excell, en las cuales podremos observar cómo trabaja el filtro de Kalman, el cual reduce el error provocado por las distorsiones, filtrando el sistema.

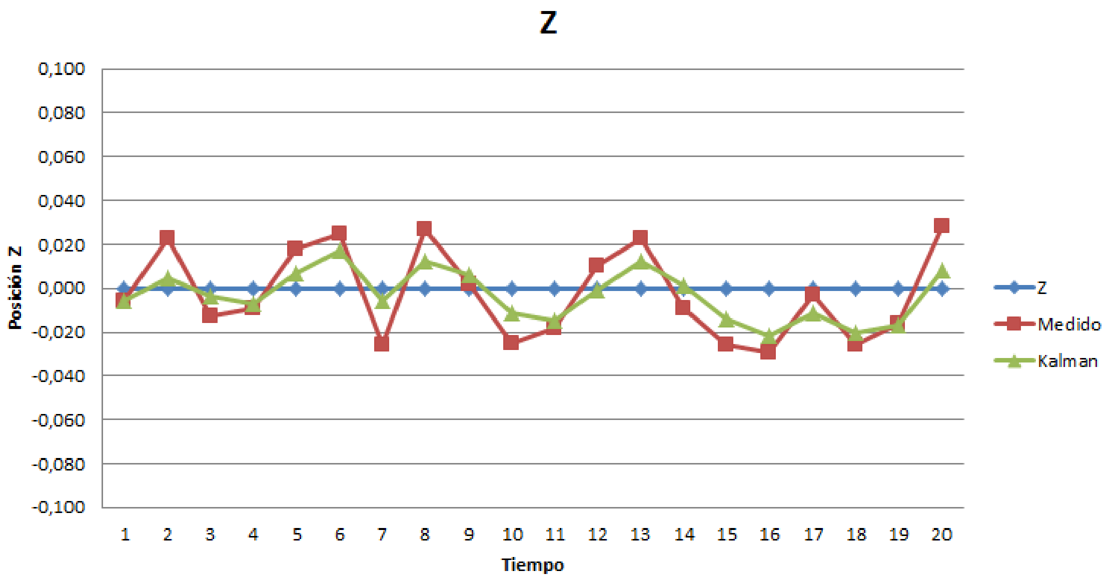
x	medido	kalman	X	Medido	Kalman
500	520	520	0,500	0,520	0,520
500	494	504	0,500	0,494	0,504
500	517	511	0,500	0,517	0,511
500	518	515	0,500	0,518	0,515
500	525	520	0,500	0,525	0,520
500	482	499	0,500	0,482	0,499
500	490	494	0,500	0,490	0,494
500	516	506	0,500	0,516	0,506
500	487	495	0,500	0,487	0,495
500	503	500	0,500	0,503	0,500
500	499	499	0,500	0,499	0,499
500	508	504	0,500	0,508	0,504
500	492	497	0,500	0,492	0,497
500	501	499	0,500	0,501	0,499
500	527	515	0,500	0,527	0,515
500	525	520	0,500	0,525	0,520
500	508	514	0,500	0,508	0,514
500	513	513	0,500	0,513	0,513
500	478	494	0,500	0,478	0,494
500	506	501	0,500	0,506	0,501
			Error Total sin kalman: 0.274863		
			Error total con kalman: 0.160758		
			Porcentaje de error reducido: 42%		



y	medido	kalman	Y	Medido	Kalman
1000	985	985	1,000	0,985	0,985
1000	978	980	1,000	0,978	0,980
1000	1018	1004	1,000	1,018	1,004
1000	979	982	1,000	0,979	0,982
1000	994	990	1,000	0,994	0,990
1000	978	981	1,000	0,978	0,981
1000	977	980	1,000	0,977	0,980
1000	1030	1009	1,000	1,030	1,009
1000	983	984	1,000	0,983	0,984
1000	1001	993	1,000	1,001	0,993
1000	1020	1004	1,000	1,020	1,004
1000	1007	997	1,000	1,007	0,997
1000	988	986	1,000	0,988	0,986
1000	1008	998	1,000	1,008	0,998
1000	1001	994	1,000	1,001	0,994
1000	1000	993	1,000	1,000	0,993
1000	1028	1009	1,000	1,028	1,009
1000	988	986	1,000	0,988	0,986
1000	1016	1002	1,000	1,016	1,002
1000	1002	994	1,000	1,002	0,994
			Error Total sin kalman: 0.284194		
			Error total con kalman: 0.192934		
			Porcentaje de error reducido: 33%		



z	medido	kalman	Z	Medido	Kalman
0	-6	-6	0,000	-0,006	-0,006
0	23	5	0,000	0,023	0,005
0	-13	-4	0,000	-0,013	-0,004
0	-9	-7	0,000	-0,009	-0,007
0	18	7	0,000	0,018	0,007
0	25	17	0,000	0,025	0,017
0	-26	-6	0,000	-0,026	-0,006
0	27	12	0,000	0,027	0,012
0	2	6	0,000	0,002	0,006
0	-25	-11	0,000	-0,025	-0,011
0	-18	-15	0,000	-0,018	-0,015
0	10	-1	0,000	0,010	-0,001
0	23	12	0,000	0,023	0,012
0	-9	1	0,000	-0,009	0,001
0	-26	-14	0,000	-0,026	-0,014
0	-29	-22	0,000	-0,029	-0,022
0	-3	-11	0,000	-0,003	-0,011
0	-26	-20	0,000	-0,026	-0,020
0	-16	-17	0,000	-0,016	-0,017
0	28	8	0,000	0,028	0,008
			Error Total sin kalman: 0.380636		
			Error total con kalman: 0.213682		
			Porcentaje de error reducido: 44%		



Antes de finalizar este apartado me gustaría hacer mención a futuros proyectos, en los cuales se podrá aplicar lo aquí realizado. Aplicaciones como localización de malezas, caracterización de uvas para vino y seguimiento de posturas humanas o brazos robots.

Sachez, A. J., & Marchant, J. A. (2000). Fusing 3D information for crop/weeds classification. In Pattern Recognition, 2000. Proceedings. 15th International Conference on (Vol. 4, pp. 295-298). IEEE.

Ivorra, E., Sánchez, A. J., Camarasa, J. G., Diago, M. P., & Tardaguila, J. (2015). Assessment of grape cluster yield components based on 3D descriptors using stereo vision. Food Control, 50, 273-282.

Berti, E. M., Salmerón, A. J. S., & Benimeli, F. (2012). Kalman filter for tracking robotic arms using low cost 3D vision systems. In The Fifth International Conference on Advances in Computer-Human Interactions (pp. 236-240).

Berti, E. M., Salmerón, A. J. S., & Benimeli, F. (2012). Human-Robot Interaction and Tracking Using low cost 3D Vision Systems. Romanian J. Tech. Sci. Appl. Mech, 7, 1-15.

Capítulo 6: Normativas y Regulaciones

Aquí un pequeño listado de las normativas y regulaciones que tenemos que tener en cuenta a la hora de realizar nuestro proyecto.

- UNE-EN 60335-1:2012/A11: 2014: Aparatos electrodomésticos y análogos. Seguridad. Parte 1: Requisitos generales.

- UNE-EN 60335-1:2012/AC: 2014: Aparatos electrodomésticos y análogos. Seguridad. Parte 1: Requisitos generales.

- UNE-EN 60947-1:2008: Aparamenta de baja tensión. Parte 1: Reglas generales.

- UNE-EN 60947-1:2008/A1:2011: Aparamenta de baja tensión. Parte 1: Reglas generales.

- UNE 21144-3-1:1997: Cables eléctricos. Cálculo de la intensidad admisible. Parte 3: Secciones sobre condiciones de funcionamiento. Sección 1: Condiciones de funcionamiento de referencia y selección del tipo de cable.

Capítulo 7: Presupuesto y Seguimiento de Horas

7.1 Presupuesto.

Materiales					
Código	**Ud**	**Denominación del material**	**Precio**	**Cantidad**	**Total**
Electrónica					
G5-m001	ud	Placa raspberry Pi Modelo B	26,05 €	1	25,05€

G5-m002	ud	Cámaras raspberry Pi	25,95 €	2	51,90€
G5-m003	ud	Fuente de Alimentación	22,80 €	1	22,80€
G5-m004	ud	Cableado	2,00 €	1	2,00€
Mecánica y materiales					
G5-m010	ud	Tornillos métrica 2	1,30 €	1	1,30€
G5-m011	ud	Tornillos Métrica 3	0,83 €	1	0,83€
G5-m012	ud	Tornillos Métrica 5	1,05 €	1	1,05€
G5-m013	ud	Cola Blanca	3,75 €	1	3,75€

Mano de Obra					
Código	**Ud**	**Descripción**	**Precio**	**Cantidad**	**Total**
MOOE.8a	h	Oficial 1ª electronica y automática	18,65 €	477	8896,05€

Costes Auxiliares					
%		Costes Directos Complementarios	9004,73€	0,02	180,10€

Sistema de Visión	80,70 €
Presupuesto de ejecución y material	9191,76€
13% de gastos generales	1194,93 €
6% de beneficio industrial	551,51 €
Suma	11018,90€
21% IVA	2313,97 €
Presupuesto de ejecución por contrata	13332,87 €

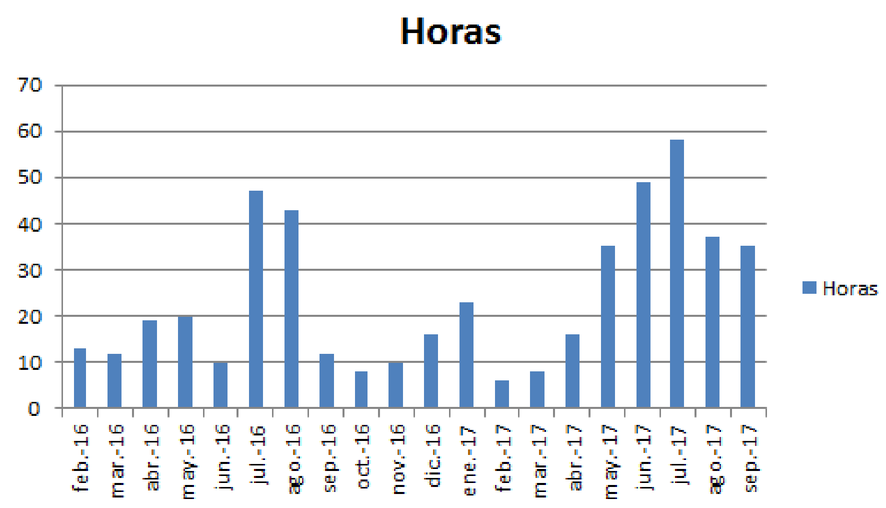
*Consideraciones:

- El precio de cada uno de los tornillos de diferente métrica, es el de la bolsa que se ha comprado, por eso solo hay una unidad.

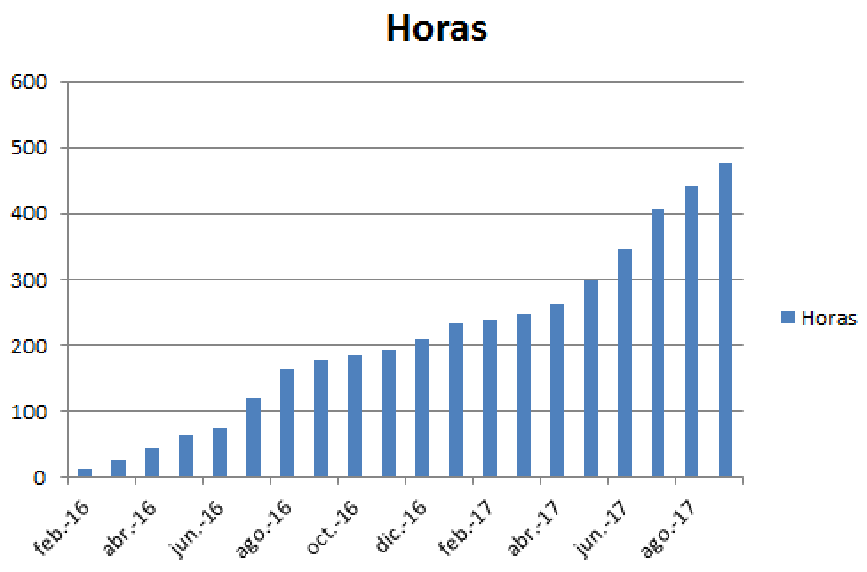
7.2 Seguimiento de Horas

El seguimiento de las horas se puede ver claramente en las siguientes gráficas.

	Horas
feb-16	13
mar-16	12
abr-16	19
may-16	20
jun-16	10
jul-16	47
ago-16	43
sep-16	12
oct-16	8
nov-16	10
dic-16	16
ene-17	23
feb-17	6
mar-17	8
abr-17	16
may-17	35
jun-17	49
jul-17	58
ago-17	37
sep-17	35



	Horas
feb-16	13
mar-16	25
abr-16	44
may-16	64
jun-16	74
jul-16	121
ago-16	164
sep-16	176
oct-16	184
nov-16	194
dic-16	210
ene-17	233
feb-17	239
mar-17	247
abr-17	263
may-17	298
jun-17	347
jul-17	405
ago-17	442
sep-17	477



Capítulo 8: Conclusiones

Este proyecto me ha servido para conocer cómo se trabajaría en un ámbito cercano al cliente (interpretado por el tutor), el cual informando de unos requerimientos y necesidades nos daba la libertad de crear el sistema que considerásemos más útil y efectivo para suplir dichas necesidades. Además de ello, el "cliente" iría valorando diferentes prototipos, hasta llegar a uno que se adaptase correctamente a sus necesidades, y así, quedar satisfecho. Creo que esto es una parte muy útil, ya que muchas veces los clientes en la vida real no saben exactamente que necesitan, y el trato continuo con ellos es muy valioso, al igual que su satisfacción. Lo cual nos premiará de cara a nuevos clientes o para nuevos pedidos con clientes anteriores.

También me ha permitido conocer diferentes programas o dispositivos además de reforzar otros muchos. Como puede ser el uso de la Raspberry Pi, la cual, pese a conocer de su existencia nunca había podido trabajar con ella. He aprendido también a reforzar mis conocimientos dentro del campo de la programación, en especial de C++.

Se han podido comprobar las ventajas de utilizar nuevas tecnologías para el desarrollo de nuestros proyectos. En este caso, un ejemplo sería el uso de una Raspberry pi con sus cámaras, lo que sobre todo nos permite ahorrar en tiempo y dinero el desarrollo de nuestros prototipos. Además he adquirido conocimientos sobre diferentes ámbitos relacionados con la visión artificial y sus características y de diferentes componentes electrónicos.

En definitiva, con este proyecto se han podido poner en práctica algunos conocimientos aprendidos previamente en la carrera, y en especial muchos nuevos conocimientos y aptitudes no aprendidos en ella. También la experiencia de enfrentarse a un proyecto de la vida real, con todos sus problemas y modificaciones que van surgiendo durante el tiempo que dura su elaboración, así como la posterior redacción de la documentación necesaria.

Bibliografía

- http://sabia.tic.udc.es/gc/trabajos%202011-12/VisionEstereoscopica/docs/introduccion.html
- Learning OpenCV: Computer Vision with the OpenCV Library Escrito por Gary Bradski,Adrian Kaehler TEMA 11 y TEMA 12
- http://www.cse.iitk.ac.in/users/vision/dipakmj/papers/OReilly%20Learning%20OpenCV.pdf
- http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration_square_chess/camera_calibration_square_chess.html
- Visión Stereoscópica https://www.youtube.com/watch?v=zyR3NbdzHmg
- http://www.raspberrypi.org/
- http://ricardforner.blogspot.com.es/2012/08/disipador-para-la-raspberry-pi.html
- http://www.pccomponentes.com/camara_para_raspberry_pi_5mp.html
- https://es.wikipedia.org/wiki/Seguimiento_de_objetos#Aplicaciones
- http://imaginacionbinaria.blogspot.com.es/2016/03/vision-artificial-ii-seguimiento-de.html

- http://sabia.tic.udc.es/gc/trabajos%202011-12/VisionEstereoscopica/docs/introduccion.html
- Learning OpenCV: Computer Vision with the OpenCV Library Escrito por Gary Bradski,Adrian Kaehler TEMA 11 y TEMA 12
- http://www.cse.iitk.ac.in/users/vision/dipakmj/papers/OReilly%20Learning%20OpenCV.pdf
- http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration_square_chess/camera_calibration_square_chess.html
- Visión Stereoscópica https://www.youtube.com/watch?v=zyR3NbdzHmg
- http://www.raspberrypi.org/
- http://ricardforner.blogspot.com.es/2012/08/disipador-para-la-raspberry-pi.html
- http://www.pccomponentes.com/camara_para_raspberry_pi_5mp.html
- https://es.wikipedia.org/wiki/Seguimiento_de_objetos#Aplicaciones
- http://imaginacionbinaria.blogspot.com.es/2016/03/vision-artificial-ii-seguimiento-de.html

- http://docs.opencv.org/3.0-beta/doc/tutorials/tutorials.html
- http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- http://docs.opencv.org/3.1.0/d4/d94/tutorial_camera_calibration.html#gsc.tab=0
- https://github.com/opencv/opencv/blob/master/samples/cpp/stereo_calib.cpp#L1
- http://sourishghosh.com/2016/stereo-calibration-cpp-opencv/
- http://stackoverflow.com/questions/9141372/3d-surface-reconstruction-using-opencv-examples
- http://www.technolabsz.com/2012/07/camera-calibration-using-opencv.html
- https://dialnet.unirioja.es/descarga/articulo/4320424.pdf
- J. G. Díaz, A. M. Mejía y F. Arteaga, Aplicación de los filtro de Kalman a sistemas de Control, 2001
- G. Welch, G. Bishop, An Introduction to the Kalman Filter, SIGGRAPH, 2001.
- R. J. Meinhold, N. D. Singpurwalla, Understanding the Kalman Filter, 1983.

- Filtro de Kalman extendido y filtro de partículas Kalman extendido para problemas de estimación No Lineal. Luis Sánchez ; Joan Ordóñez ; Saba Infante ;
- http://www.redalyc.org/articulo.oa?id=70732640002
- https://www.emacswiki.org/
- http://www.gnu.org/software/emacs/emacs.html
- http://www0.cs.ucl.ac.uk/staff/gridgway/kalman/kalman_example/Kalman_Example.html
- https://statweb.stanford.edu/~candes/acm116/Handouts/Kalman.pdf
- http://www.konradlorenz.edu.co/images/stories/suma_digital_matematicas/Trabajo_Grado_Guillermo_Prado_613003.pdf
- https://gist.github.com/jannson/9951716
- Sachez, A. J., & Marchant, J. A. (2000). Fusing 3D information for crop/weeds classification. In Pattern Recognition, 2000. Proceedings. 15th International Conference on (Vol. 4, pp. 295-298). IEEE
- Ivorra, E., Sánchez, A. J., Camarasa, J. G., Diago, M. P., & Tardaguila, J. (2015). Assessment of grape cluster yield components based on 3D descriptors using stereo vision. Food Control, 50, 273-282.
- Berti, E. M., Salmerón, A. J. S., & Benimeli, F. (2012). Kalman filter for tracking robotic arms using low cost 3D vision systems. In The Fifth International Conference on Advances in Computer-Human Interactions (pp. 236-240).
- Berti, E. M., Salmerón, A. J. S., & Benimeli, F. (2012). Human-Robot Interaction and Tracking Using low cost 3D Vision Systems. Romanian J. Tech. Sci. Appl. Mech, 7, 1-15.
- Sánchez, A., & Marchant, J. (1997). Fast and robust method for tracking crop rows using a two point Hough transform. Acts of BioRobotics, 97.
- Ricolfe-Viala, C., & Sanchez-Salmeron, A. J. (2007). Improved camera calibration method based on a two-dimensional template. Pattern Recognition and Image Analysis, 420-427.
- Ricolfe, C., & Sánchez, A. J. (2008). PROCEDIMIENTO COMPLETO PARA EL CALIBRADO DE CÁMARAS UTILIZANDO UNA PLANTILLA PLANA. RIAII, 5(1), 93-101.
- Benlloch, J. V., Sánchez, A., Agustí, M., & Albertos, P. (1996). Weed Detection in Cereal Fields Using Image Processing Techniques. Precision Agriculture, (precisionagricu3), 903-903.
- Benlloch, J. V., Sanchez, A., Christensen, S., & Walger, M. (1996). Weed mapping in cereal crops using image analysis techniques. AgEng96, 2, 1059-1060.
- Benlloch, J. V., Agusti, M., Sanchez, A., & Rodas, A. (1995, October). Colour segmentation techniques for detecting weed patches in cereal crops. In Proc. of Fourth Workshop on Robotics in Agriculture and the Food-Industry (pp. 30-31).

ANEXOS

Camera Calibration

Camera_calibration.cpp

```
#include "opencv2/core.hpp"

#include <opencv2/core/utility.hpp>

#include "opencv2/imgproc.hpp"

#include "opencv2/calib3d.hpp"

#include "opencv2/imgcodecs.hpp"

#include "opencv2/videoio.hpp"

#include "opencv2/highgui.hpp"

#include "/home/pi/raspicam-0.1.3/src/raspicam_cv.h"

#include "opencv2/xphoto.hpp"

#include <cctype>

#include <stdio.h>

#include <string.h>

#include <time.h>

using namespace cv;

using namespace std;

const char * usage =

" \nexample command line for calibration from a live feed.\n"

  "   calibration  -w 4 -h 5 -s 0.025 -n 10 -o camera.yml -op -oe\n";

const char* liveCaptureHelp =

   "When the live video from camera is used as input, the following hot-keys may be used:\n"

      "  <ESC>, 'q' - quit the program\n"

      "  'g' - start capturing images\n"

      "  'u' - switch undistortion on/off\n";


static void help()
```

```
{
   printf( "This is a camera calibration sample.\n"
      "Usage: calibration\n"
      "     -w <board_width>         # the number of inner corners per one of board dimension\n"
      "     -h <board_height>        # the number of inner corners per another board dimension\n"
      "     [-pt <pattern>]          # the type of pattern: chessboard or circles' grid\n"
      "     [-n <number_of_frames>]  # the number of frames to use for calibration\n"
      "                              # (if not specified, it will be set to the number\n"
      "                              #  of board views actually available)\n"
      "     [-d <delay>]             # a minimum delay in ms between subsequent attempts to capture
a next view\n"
      "     [-s <squareSize>]        # square size in some user-defined units (1 by default)\n"
      "     [-o <out_camera_params>] # the output filename for intrinsic [and extrinsic]
parameters\n"
      "     [-op]                    # write detected feature points\n"
      "     [-oe]                    # write extrinsic parameters\n"
      "     [-zt]                    # assume zero tangential distortion\n"
      "     [-a <aspectRatio>]       # fix aspect ratio (fx/fy)\n"
      "     [-p]                     # fix the principal point at the center\n"
      "     [-v]                     # flip the captured images around the horizontal axis\n"
      "     [-su]                    # show undistorted images after calibration\n"
      "\n" );
   printf("\n%s",usage);
   printf( "\n%s", liveCaptureHelp );
}
enum { DETECTION = 0, CAPTURING = 1, CALIBRATED = 2 };
enum Pattern { CHESSBOARD, CIRCLES_GRID, ASYMMETRIC_CIRCLES_GRID };
static double computeReprojectionErrors(
        const vector<vector<Point3f> >& objectPoints,
        const vector<vector<Point2f> >& imagePoints,
        const vector<Mat>& rvecs, const vector<Mat>& tvecs,
        const Mat& cameraMatrix, const Mat& distCoeffs,
        vector<float>& perViewErrors )
{
    vector<Point2f> imagePoints2;
    int i, totalPoints = 0;
    double totalErr = 0, err;
    perViewErrors.resize(objectPoints.size());
    for( i = 0; i < (int)objectPoints.size(); i++ )
    {
        projectPoints(Mat(objectPoints[i]), rvecs[i], tvecs[i],
                      cameraMatrix, distCoeffs, imagePoints2);
        err = norm(Mat(imagePoints[i]), Mat(imagePoints2), NORM_L2);
        int n = (int)objectPoints[i].size();
        perViewErrors[i] = (float)std::sqrt(err*err/n);
        totalErr += err*err;
        totalPoints += n;
    }
    return std::sqrt(totalErr/totalPoints);
}
static void calcChessboardCorners(Size boardSize, float squareSize, vector<Point3f>& corners,
Pattern patternType = CHESSBOARD)
{
    corners.resize(0);
    switch(patternType)
    {
     case CHESSBOARD:
     case CIRCLES_GRID:
       for( int i = 0; i < boardSize.height; i++ )
           for( int j = 0; j < boardSize.width; j++ )
               corners.push_back(Point3f(float(j*squareSize),
                                         float(i*squareSize), 0));
       break;
     case ASYMMETRIC_CIRCLES_GRID:
       for( int i = 0; i < boardSize.height; i++ )
           for( int j = 0; j < boardSize.width; j++ )
               corners.push_back(Point3f(float((2*j + i % 2)*squareSize),
                                         float(i*squareSize), 0));
       break;
     default:
       CV_Error(Error::StsBadArg, "Unknown pattern type\n");
   }
}
static bool runCalibration( vector<vector<Point2f> > imagePoints,
                    Size imageSize, Size boardSize, Pattern patternType,
                    float squareSize, float aspectRatio,
                    int flags, Mat& cameraMatrix, Mat& distCoeffs,
                    vector<Mat>& rvecs, vector<Mat>& tvecs,
                    vector<float>& reprojErrs,
                    double& totalAvgErr)
{
    cameraMatrix = Mat::eye(3, 3, CV_64F);
    if( flags & CALIB_FIX_ASPECT_RATIO )
```

```
        cameraMatrix.at<double>(0,0) = aspectRatio;
    distCoeffs = Mat::zeros(8, 1, CV_64F);
    vector<vector<Point3f> > objectPoints(1);
    calcChessboardCorners(boardSize, squareSize, objectPoints[0], patternType);
    objectPoints.resize(imagePoints.size(),objectPoints[0]);
    double rms = calibrateCamera(objectPoints, imagePoints, imageSize, cameraMatrix,
                    distCoeffs, rvecs, tvecs, flags|CALIB_FIX_K4|CALIB_FIX_K5);
                    ///*|CALIB_FIX_K3*/|CALIB_FIX_K4|CALIB_FIX_K5);
    printf("RMS error reported by calibrateCamera: %g\n", rms);
    bool ok = checkRange(cameraMatrix) && checkRange(distCoeffs);
    totalAvgErr = computeReprojectionErrors(objectPoints, imagePoints,
                rvecs, tvecs, cameraMatrix, distCoeffs, reprojErrs);
    return ok;
}
static void saveCameraParams( const string& filename,
                       Size imageSize, Size boardSize,
                       float squareSize, float aspectRatio, int flags,
                       const Mat& cameraMatrix, const Mat& distCoeffs,
                       const vector<Mat>& rvecs, const vector<Mat>& tvecs,
                       const vector<float>& reprojErrs,
                       const vector<vector<Point2f> >& imagePoints,
                       double totalAvgErr )
{
    FileStorage fs( filename, FileStorage::WRITE );
    time_t tt;
    time( &tt );
    struct tm *t2 = localtime( &tt );
```

```
char buf[1024];

strftime( buf, sizeof(buf)-1, "%c", t2 );

fs << "calibration_time" << buf;

if( !rvecs.empty() || !reprojErrs.empty() )

   fs << "nframes" << (int)std::max(rvecs.size(), reprojErrs.size());

fs << "image_width" << imageSize.width;

fs << "image_height" << imageSize.height;

fs << "board_width" << boardSize.width;

fs << "board_height" << boardSize.height;

fs << "square_size" << squareSize;

if( flags & CALIB_FIX_ASPECT_RATIO )

   fs << "aspectRatio" << aspectRatio;

if( flags != 0 )

{

   sprintf( buf, "flags: %s%s%s%s",

      flags & CALIB_USE_INTRINSIC_GUESS ? "+use_intrinsic_guess" : "",

      flags & CALIB_FIX_ASPECT_RATIO ? "+fix_aspectRatio" : "",

      flags & CALIB_FIX_PRINCIPAL_POINT ? "+fix_principal_point" : "",

      flags & CALIB_ZERO_TANGENT_DIST ? "+zero_tangent_dist" : "" );

   //cvWriteComment( *fs, buf, 0 );

}

fs << "flags" << flags;

fs << "rvecs" << rvecs;

fs << "tvecs" << tvecs;

fs << "camera_matrix" << cameraMatrix;

fs << "distortion_coefficients" << distCoeffs;

fs << "avg_reprojection_error" << totalAvgErr;
```

```
    if( !reprojErrs.empty() )
        fs << "per_view_reprojection_errors" << Mat(reprojErrs);
    if( !rvecs.empty() && !tvecs.empty() )
    {
        CV_Assert(rvecs[0].type() == tvecs[0].type());
        Mat bigmat((int)rvecs.size(), 6, rvecs[0].type());
        for( int i = 0; i < (int)rvecs.size(); i++ )
        {
            Mat r = bigmat(Range(i, i+1), Range(0,3));
            Mat t = bigmat(Range(i, i+1), Range(3,6));
            CV_Assert(rvecs[i].rows == 3 && rvecs[i].cols == 1);
            CV_Assert(tvecs[i].rows == 3 && tvecs[i].cols == 1);
            //*.t() is MatExpr (not Mat) so we can use assignment operator
            r = rvecs[i].t();
            t = tvecs[i].t();
        }
        //cvWriteComment( *fs, "a set of 6-tuples (rotation vector + translation vector) for each
view", 0 );
        //fs << "extrinsic_parameters" << bigmat;
    }
    if( !imagePoints.empty() )
    {
        Mat imagePtMat((int)imagePoints.size(), (int)imagePoints[0].size(), CV_32FC2);
        for( int i = 0; i < (int)imagePoints.size(); i++ )
        {
            Mat r = imagePtMat.row(i).reshape(2, imagePtMat.cols);
            Mat imgpti(imagePoints[i]);
            imgpti.copyTo(r);
        }
        fs << "image_points" << imagePtMat;
    }
}
static bool runAndSave(const string& outputFilename,
                const vector<vector<Point2f> >& imagePoints,
                Size imageSize, Size boardSize, Pattern patternType, float squareSize,
                float aspectRatio, int flags, Mat& cameraMatrix,
                Mat& distCoeffs, bool writeExtrinsics, bool writePoints )
{
    vector<Mat> rvecs, tvecs;
    vector<float> reprojErrs;
    double totalAvgErr = 0;
    bool ok = runCalibration(imagePoints, imageSize, boardSize, patternType, squareSize,
                   aspectRatio, flags, cameraMatrix, distCoeffs,
                   rvecs, tvecs, reprojErrs, totalAvgErr);
    printf("%s. avg reprojection error = %.2f\n",
           ok ? "Calibration succeeded" : "Calibration failed",
           totalAvgErr);
    if( ok )
        saveCameraParams( outputFilename, imageSize,
                         boardSize, squareSize, aspectRatio,
                         flags, cameraMatrix, distCoeffs,
                         writeExtrinsics ? rvecs : vector<Mat>(),
                         writeExtrinsics ? tvecs : vector<Mat>(),
                         writeExtrinsics ? reprojErrs : vector<float>(),
                         writePoints ? imagePoints : vector<vector<Point2f> >(),
                    totalAvgErr );

    return ok;

}

int main( int argc, char** argv )

{

    Size boardSize, imageSize;

    float squareSize = 1.f, aspectRatio = 1.f;

    Mat cameraMatrix, distCoeffs;

    const char* outputFilename = "out_camera_data.yml";



    int i, nframes = 10;

    bool writeExtrinsics = true, writePoints = false;

    bool undistortImage = false;

    int flags = 0;

    VideoCapture capture;

    bool flipVertical = false;

    bool showUndistorted = false;

    int delay = 1000;

    clock_t prevTimestamp = 0;

    int mode = DETECTION;

    int cameraId = 0;

    vector<vector<Point2f> > imagePoints;

    vector<string> imageList;

    Pattern pattern = CHESSBOARD;

            raspicam::RaspiCam_Cv Camera;

            Camera.set(CV_CAP_PROP_FRAME_WIDTH, 640);
```

```
     Camera.set(CV_CAP_PROP_FRAME_HEIGHT, 480);
     Camera.set(CV_CAP_PROP_BRIGHTNESS, 50);
     Camera.set(CV_CAP_PROP_CONTRAST, 50);
     Camera.set(CV_CAP_PROP_SATURATION, 50);
     Camera.set(CV_CAP_PROP_GAIN, 50);
     Camera.set(CV_CAP_PROP_FORMAT, CV_8UC3); //BGR
     bool conected=false;
if( argc < 2 )
{
   help();
   return 0;
}
for( i = 1; i < argc; i++ )
{
   const char* s = argv[i];
   if( strcmp( s, "-w" ) == 0 )
   {
      if( sscanf( argv[++i], "%u", &boardSize.width ) != 1 || boardSize.width <= 0 )
         return fprintf( stderr, "Invalid board width\n" ), -1;
   }
   else if( strcmp( s, "-h" ) == 0 )
   {
      if( sscanf( argv[++i], "%u", &boardSize.height ) != 1 || boardSize.height <= 0 )
         return fprintf( stderr, "Invalid board height\n" ), -1;
   }
   else if( strcmp( s, "-pt" ) == 0 )
   {
```

```
  i++;
  if( !strcmp( argv[i], "circles" ) )
    pattern = CIRCLES_GRID;
  else if( !strcmp( argv[i], "acircles" ) )
    pattern = ASYMMETRIC_CIRCLES_GRID;
  else if( !strcmp( argv[i], "chessboard" ) )
    pattern = CHESSBOARD;
  else
    return fprintf( stderr, "Invalid pattern type: must be chessboard or circles\n" ), -1;
}
else if( strcmp( s, "-s" ) == 0 )
{
  if( sscanf( argv[++i], "%f", &squareSize ) != 1 || squareSize <= 0 )
    return fprintf( stderr, "Invalid board square width\n" ), -1;
}
else if( strcmp( s, "-n" ) == 0 )
{
  if( sscanf( argv[++i], "%u", &nframes ) != 1 || nframes <= 3 )
    return printf("Invalid number of images\n" ), -1;
}
else if( strcmp( s, "-a" ) == 0 )
{
  if( sscanf( argv[++i], "%f", &aspectRatio ) != 1 || aspectRatio <= 0 )
    return printf("Invalid aspect ratio\n" ), -1;
  flags |= CALIB_FIX_ASPECT_RATIO;
}
else if( strcmp( s, "-d" ) == 0 )
```

```
{
    if( sscanf( argv[++i], "%u", &delay ) != 1 || delay <= 0 )
        return printf("Invalid delay\n" ), -1;
}
else if( strcmp( s, "-op" ) == 0 )
{
    writePoints = true;
}
else if( strcmp( s, "-oe" ) == 0 )
{
    writeExtrinsics = true;
}
else if( strcmp( s, "-zt" ) == 0 )
{
    flags |= CALIB_ZERO_TANGENT_DIST;
}
else if( strcmp( s, "-p" ) == 0 )
{
    flags |= CALIB_FIX_PRINCIPAL_POINT;
}
else if( strcmp( s, "-v" ) == 0 )
{
    flipVertical = true;
}
else if( strcmp( s, "-o" ) == 0 )
{
    outputFilename = argv[++i];
    }
    else if( strcmp( s, "-su" ) == 0 )
    {
        showUndistorted = true;
    }
    else if( s[0] != '-' )
    {
        if( isdigit(s[0]) )
            sscanf(s, "%d", &cameraId);
    }
    else
        return fprintf( stderr, "Unknown option %s", s ), -1;
}
  conected=Camera.open();
  //capture.open(cameraId);
if( !conected && imageList.empty() )
    return fprintf( stderr, "Could not initialize video (%d) capture\n",cameraId ), -2;
if( !imageList.empty() )
    nframes = (int)imageList.size();
if( conected)
    printf( "%s", liveCaptureHelp );
namedWindow( "Image View", 1 );
for(i = 0;;i++)
{
    Mat view, viewGray;
    bool blink = false;
    if(conected)
```

```
        {
            Mat view0;
            //capture >> view0;
                                Camera.grab();
                                Camera.retrieve(view0);
                                cv::xphoto::balanceWhite(view0, view,
    cv::xphoto::WHITE_BALANCE_SIMPLE);
        }
        if(view.empty())
        {
            if( imagePoints.size() > 0 )
                runAndSave(outputFilename, imagePoints, imageSize,
                           boardSize, pattern, squareSize, aspectRatio,
                           flags, cameraMatrix, distCoeffs,
                           writeExtrinsics, writePoints);
            break;
        }
        imageSize = view.size();
        if( flipVertical )
            flip( view, view, 0 );
        vector<Point2f> pointbuf;
        cvtColor(view, viewGray, COLOR_BGR2GRAY);
        bool found;
        switch( pattern )
        {
            case CHESSBOARD:
                found = findChessboardCorners( view, boardSize, pointbuf,
                    CALIB_CB_ADAPTIVE_THRESH | CALIB_CB_FAST_CHECK |
    CALIB_CB_NORMALIZE_IMAGE);

                        //printf("Chessboard Corners pattern ");

                        //printf(found? "Found":"Not Found");

                        //printf("\n");

                break;

            case CIRCLES_GRID:

                found = findCirclesGrid( view, boardSize, pointbuf );

                break;

            case ASYMMETRIC_CIRCLES_GRID:

                found = findCirclesGrid( view, boardSize, pointbuf, CALIB_CB_ASYMMETRIC_GRID );

                break;

            default:

                return fprintf( stderr, "Unknown pattern type\n" ), -1;

        }

       // improve the found corners' coordinate accuracy

               if( pattern == CHESSBOARD && found) cornerSubPix( viewGray, pointbuf,
    Size(11,11),

            Size(-1,-1), TermCriteria( TermCriteria::EPS+TermCriteria::COUNT, 30, 0.1 ));

        if( mode == CAPTURING && found &&

            (!conected|| clock() - prevTimestamp > delay*1e-3*CLOCKS_PER_SEC) )

        {

            imagePoints.push_back(pointbuf);

            prevTimestamp = clock();

            blink = conected;

        }

        if(found)

            drawChessboardCorners( view, boardSize, Mat(pointbuf), found );
```

```
string msg = mode == CAPTURING ? "100/100" :
   mode == CALIBRATED ? "Calibrated" : "Press 'g' to start";
int baseLine = 0;
Size textSize = getTextSize(msg, 1, 1, 1, &baseLine);
Point textOrigin(view.cols - 2*textSize.width - 10, view.rows - 2*baseLine - 10);
if( mode == CAPTURING )
{
   if(undistortImage)
      msg = format( "%d/%d Undist", (int)imagePoints.size(), nframes );
   else
      msg = format( "%d/%d", (int)imagePoints.size(), nframes );
}
putText( view, msg, textOrigin, 1, 1,
       mode != CALIBRATED ? Scalar(0,0,255) : Scalar(0,255,0));

if( blink )
   bitwise_not(view, view);

if( mode == CALIBRATED && undistortImage )
{
   Mat temp = view.clone();
   undistort(temp, view, cameraMatrix, distCoeffs);
}
imshow("Image View", view);
int key = 0xff & waitKey(conected ? 50 : 500);
if( (key & 255) == 27 )
   break;
```

```
        if( key == 'u' && mode == CALIBRATED )

           undistortImage = !undistortImage;

        if( conected && key == 'g' )

        {

           mode = CAPTURING;

           imagePoints.clear();

        }

        if( mode == CAPTURING && imagePoints.size() >= (unsigned)nframes )

        {

                if( runAndSave(outputFilename, imagePoints, imageSize,

                   boardSize, pattern, squareSize, aspectRatio,

                   flags, cameraMatrix, distCoeffs,

                   writeExtrinsics, writePoints))

              mode = CALIBRATED;

           else

              mode = DETECTION;

           if( !conected )

              break;

        }

     }

     if( !conected && showUndistorted )

     {

        Mat view, rview, map1, map2;

        initUndistortRectifyMap(cameraMatrix, distCoeffs, Mat(),

                              getOptimalNewCameraMatrix(cameraMatrix, distCoeffs, imageSize, 1,
imageSize, 0),

                              imageSize, CV_16SC2, map1, map2);

        for( i = 0; i < (int)imageList.size(); i++ )
        {
            view = imread(imageList[i], 1);
            if(view.empty())
                continue;
            remap(view, rview, map1, map2, INTER_LINEAR);
            imshow("Image View", rview);
            int c = 0xff & waitKey();
            if( (c & 255) == 27 || c == 'q' || c == 'Q' )
                break;
        }
    }
    return 0;
}
```

Makefile

```
OBJS = camera_calibration.o
CC = g++
DEBUG = -g
INC_PATH = -I/home/pi/raspicam-0.1.3/src
CFLAGS = -Wall -c $(DEBUG) $(INC_PATH)
LFLAGS = -Wall $(DEBUG) -L/home/pi/raspicam-0.1.3/build/src
calibration: $(OBJS)
	$(CC) $(LFLAGS) $(OBJS) -lopencv_calib3d -lopencv_xphoto -lopencv_imgcodecs -
lopencv_core -lopencv_imgproc -lopencv_highgui -lopencv_ml -lraspicam -lraspicam_cv  -
lopencv_features2d -lopencv_videoio -o camera_calibration
calibration.o:camera_calibration.cpp
	$(CC) $(CFLAGS) canera_calibration.cpp
info:
	@echo OBJS = $(OBJS)
	@echo CC = $(CC)info:
	@echo OBJS = $(OBJS)
	@echo CC = $(CC)
	@echo DEBUG = $(DEBUG)
	@echo INC_PATH = $(INC_PATH)
	@echo CFLAGS = $(CFLAGS)
	@echo LFLAGS = $(LFLAGS)
	@echo DEBUG = $(DEBUG)
	@echo INC_PATH = $(INC_PATH)
	@echo CFLAGS = $(CFLAGS)
	@echo LFLAGS = $(LFLAGS)
clean:
	rm -f $(OBJS)
```

<u>VID5</u>

```
<?xml version="1.0"?>
<opencv_storage>
<images>
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto1.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto2.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto3.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto4.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto5.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto6.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto7.jpg
/home/pi/opencv/samples/cpp/tutorial_code/calib3d/camera_calibration/Fotos/foto8.jpg
```

```
</images>

</opencv_storage>
```

In VID5

```
<?xml version="1.0"?>

<opencv_storage>

<Settings>

<!-- Number of inner corners per a item row and column. (square, circle) -->

<BoardSize_Width>8</BoardSize_Width>

<BoardSize_Height>6</BoardSize_Height>

<!-- The size of a square in some user defined metric system (pixel, millimeter)-->

<Square_Size>106</Square_Size>

<!-- The type of input used for camera calibration. One of: CHESSBOARD CIRCLES_GRID
ASYMMETRIC_CIRCLES_GRID -->

<Calibrate_Pattern>"CHESSBOARD"</Calibrate_Pattern>

<!-- The input to use for calibration.

                To use an input camera -> give the ID of the camera, like "1"

                To use an input video  -> give the path of the input video, like "/tmp/x.avi"

                To use an image list   -> give the path to the XML or YAML file containing the
list of the images, like "/tmp/circles_list.xml"-->

<Input>"VID5.xml"</Input>

<!--  If true (non-zero) we flip the input images around the horizontal axis.-->

<Input_FlipAroundHorizontalAxis>0</Input_FlipAroundHorizontalAxis>

<!--  Time delay between frames in case of camera.  -->

<Input_Delay>100</Input_Delay>

<!--  How many frames to use, for calibration.  -->

<Calibrate_NrOfFrameToUse>25</Calibrate_NrOfFrameToUse>

<!-- Consider only fy as a free parameter, the ratio fx/fy stays the same as in the input
cameraMatrix.
```

```
        Use or not setting. 0 - False Non-Zero - True-->

<Calibrate_FixAspectRatio>1</Calibrate_FixAspectRatio>

<!-- If true (non-zero) tangential distortion coefficients  are set to zeros and stay zero.-->

<Calibrate_AssumeZeroTangentialDistortion>1</Calibrate_AssumeZeroTangentialDistortion>

<!-- If true (non-zero) the principal point is not changed during the global optimization.-->

<Calibrate_FixPrincipalPointAtTheCenter>1</Calibrate_FixPrincipalPointAtTheCenter>

<!--  The name of the output log file.  -->

<Write_outputFileName>"out_camera_data.xml"</Write_outputFileName>

<!-- If true (non-zero) we write to the output file the feature points.-->

<Write_DetectedFeaturePoints>1</Write_DetectedFeaturePoints>

<!-- If true (non-zero) we write to the output file the extrinsic camera parameters.-->

<Write_extrinsicParameters>1</Write_extrinsicParameters>

<!-- If true (non-zero) we show after calibration the undistorted images.-->

<Show_UndistortedImage>1</Show_UndistortedImage>

</Settings>

</opencv_storage>
```

Stereo Calibration

Stereo_calib.cpp

```
#include "opencv2/calib3d/calib3d.hpp"

#include "opencv2/imgcodecs.hpp"

#include "opencv2/highgui/highgui.hpp"

#include "opencv2/imgproc/imgproc.hpp"

#include <vector>

#include <string>

#include <algorithm>

#include <iostream>
```

```
#include <iterator>

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

using namespace cv;

using namespace std;

static int print_help()

{

    cout <<

            " Given a list of chessboard images, the number of corners (nx, ny)\n"

            " on the chessboards, and a flag: useCalibrated for \n"

            "   calibrated (0) or\n"

            "   uncalibrated \n"

            "     (1: use cvStereoCalibrate(), 2: compute fundamental\n"

            "         matrix separately) stereo. \n"

            " Calibrate the cameras and display the\n"

            " rectified results along with the computed disparity images.   \n" << endl;

    cout << "Usage:\n ./stereo_calib -w board_width -h board_height [-nr /*dot not view
results*/] <image list XML/YML file>\n" << endl;

    return 0;

}

static void

StereoCalib(const vector<string>& imagelist, Size boardSize, bool useCalibrated=true, bool
showRectified=true)

{

    if( imagelist.size() % 2 != 0 )

    {

        cout << "Error: the image list contains odd (non-even) number of elements\n";
        return;
    }
    bool displayCorners = false;//true;
    const int maxScale = 2;
    const float squareSize = 1.f;  // Set this to your actual square size
    // ARRAY AND VECTOR STORAGE:
    vector<vector<Point2f> > imagePoints[2];
    vector<vector<Point3f> > objectPoints;
    Size imageSize;
    int i, j, k, nimages = (int)imagelist.size()/2;
    imagePoints[0].resize(nimages);
    imagePoints[1].resize(nimages);
    vector<string> goodImageList;

    for( i = j = 0; i < nimages; i++ )
    {
        for( k = 0; k < 2; k++ )
        {
            const string& filename = imagelist[i*2+k];
            Mat img = imread(filename, 0);
            if(img.empty())
                break;
            if( imageSize == Size() )
                imageSize = img.size();
            else if( img.size() != imageSize )
            {
                cout << "The image " << filename << " has the size different from the first image size.
Skipping the pair\n";
```

```
    break;
}
bool found = false;
vector<Point2f>& corners = imagePoints[k][j];
for( int scale = 1; scale <= maxScale; scale++ )
{
    Mat timg;
    if( scale == 1 )
        timg = img;
    else
        resize(img, timg, Size(), scale, scale);
    found = findChessboardCorners(timg, boardSize, corners,
        CALIB_CB_ADAPTIVE_THRESH | CALIB_CB_NORMALIZE_IMAGE);
    if( found )
    {
        if( scale > 1 )
        {
            Mat cornersMat(corners);
            cornersMat *= 1./scale;
        }
        break;
    }
}
if( displayCorners )
{
    cout << filename << endl;
    Mat cimg, cimg1;
```

```
            cvtColor(img, cimg, COLOR_GRAY2BGR);
            drawChessboardCorners(cimg, boardSize, corners, found);
            double sf = 640./MAX(img.rows, img.cols);
            resize(cimg, cimg1, Size(), sf, sf);
            imshow("corners", cimg1);
            char c = (char)waitKey(500);
            if( c == 27 || c == 'q' || c == 'Q' ) //Allow ESC to quit
                exit(-1);
        }
        else
            putchar('.');
        if( !found )
            break;
        cornerSubPix(img, corners, Size(11,11), Size(-1,-1),
                     TermCriteria(TermCriteria::COUNT+TermCriteria::EPS,
                                  30, 0.01));
    }
    if( k == 2 )
    {
        goodImageList.push_back(imagelist[i*2]);
        goodImageList.push_back(imagelist[i*2+1]);
        j++;
    }
}
cout << j << " pairs have been successfully detected.\n";
nimages = j;
if( nimages < 2 )
```

```
{
   cout << "Error: too little pairs to run the calibration\n";
   return;
}
imagePoints[0].resize(nimages);
imagePoints[1].resize(nimages);
objectPoints.resize(nimages);
for( i = 0; i < nimages; i++ )
{
   for( j = 0; j < boardSize.height; j++ )
      for( k = 0; k < boardSize.width; k++ )
         objectPoints[i].push_back(Point3f(k*squareSize, j*squareSize, 0));
}
cout << "Running stereo calibration ...\n";
Mat cameraMatrix[2], distCoeffs[2];
cameraMatrix[0] = Mat::eye(3, 3, CV_64F);
cameraMatrix[1] = Mat::eye(3, 3, CV_64F);
Mat R, T, E, F;
double rms = stereoCalibrate(objectPoints, imagePoints[0], imagePoints[1],
            cameraMatrix[0], distCoeffs[0],
            cameraMatrix[1], distCoeffs[1],
            imageSize, R, T, E, F,
            CALIB_FIX_ASPECT_RATIO +
            CALIB_ZERO_TANGENT_DIST +
            CALIB_SAME_FOCAL_LENGTH +
            CALIB_RATIONAL_MODEL +
            CALIB_FIX_K3 + CALIB_FIX_K4 + CALIB_FIX_K5,
                TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 100, 1e-5) );
    cout << "done with RMS error=" << rms << endl;
// CALIBRATION QUALITY CHECK
// because the output fundamental matrix implicitly
// includes all the output information,
// we can check the quality of calibration using the
// epipolar geometry constraint: m2^t*F*m1=0
    double err = 0;
    int npoints = 0;
    vector<Vec3f> lines[2];
    for( i = 0; i < nimages; i++ )
    {
        int npt = (int)imagePoints[0][i].size();
        Mat imgpt[2];
        for( k = 0; k < 2; k++ )
        {
            imgpt[k] = Mat(imagePoints[k][i]);
            undistortPoints(imgpt[k], imgpt[k], cameraMatrix[k], distCoeffs[k], Mat(),
cameraMatrix[k]);
            computeCorrespondEpilines(imgpt[k], k+1, F, lines[k]);
        }
        for( j = 0; j < npt; j++ )
        {
            double errij = fabs(imagePoints[0][i][j].x*lines[1][j][0] +
                                imagePoints[0][i][j].y*lines[1][j][1] + lines[1][j][2]) +
                           fabs(imagePoints[1][i][j].x*lines[0][j][0] +
                                imagePoints[1][i][j].y*lines[0][j][1] + lines[0][j][2]);
            err += errij;
        }
        npoints += npt;
    }
    cout << "average reprojection err = " <<  err/npoints << endl;
    // save intrinsic parameters
    FileStorage fs("intrinsics.yml", FileStorage::WRITE);
    fs.open("intrinsics.yml", FileStorage::WRITE);
    if( fs.isOpened() )
    {
        fs << "M1" << cameraMatrix[0] << "D1" << distCoeffs[0] <<
            "M2" << cameraMatrix[1] << "D2" << distCoeffs[1];
        fs.release();
    }
    else
        cout << "Error: can not save the intrinsic parameters\n";
    Mat R1, R2, P1, P2, Q;
    Rect validRoi[2];
    stereoRectify(cameraMatrix[0], distCoeffs[0],
                  cameraMatrix[1], distCoeffs[1],
                  imageSize, R, T, R1, R2, P1, P2, Q,
                  CALIB_ZERO_DISPARITY, 1, imageSize, &validRoi[0], &validRoi[1]);
    fs.open("extrinsics.yml", FileStorage::WRITE);
    if( fs.isOpened() )
    {
        fs << "R" << R << "T" << T << "R1" << R1 << "R2" << R2 << "P1" << P1 << "P2" << P2 << "Q"
<< Q;
        fs.release();
    }
    else
        cout << "Error: can not save the extrinsic parameters\n";
    // OpenCV can handle left-right
    // or up-down camera arrangements
    bool isVerticalStereo = fabs(P2.at<double>(1, 3)) > fabs(P2.at<double>(0, 3));
// COMPUTE AND DISPLAY RECTIFICATION
    if( !showRectified )
        return;

    Mat rmap[2][2];
// IF BY CALIBRATED (BOUGUET'S METHOD)
    if( useCalibrated )
    {
        // we already computed everything
    }
// OR ELSE HARTLEY'S METHOD
    else
 // use intrinsic parameters of each camera, but
 // compute the rectification transformation directly
 // from the fundamental matrix
    {
        vector<Point2f> allimgpt[2];
        for( k = 0; k < 2; k++ )
        {
            for( i = 0; i < nimages; i++ )
                std::copy(imagePoints[k][i].begin(), imagePoints[k][i].end(),
back_inserter(allimgpt[k]));
        }
        F = findFundamentalMat(Mat(allimgpt[0]), Mat(allimgpt[1]), FM_8POINT, 0, 0);

        Mat H1, H2;

        stereoRectifyUncalibrated(Mat(allimgpt[0]), Mat(allimgpt[1]), F, imageSize, H1, H2, 3);

        R1 = cameraMatrix[0].inv()*H1*cameraMatrix[0];

        R2 = cameraMatrix[1].inv()*H2*cameraMatrix[1];

        P1 = cameraMatrix[0];

        P2 = cameraMatrix[1];

    }


    //Precompute maps for cv::remap()

    initUndistortRectifyMap(cameraMatrix[0], distCoeffs[0], R1, P1, imageSize, CV_16SC2,
rmap[0][0], rmap[0][1]);

    initUndistortRectifyMap(cameraMatrix[1], distCoeffs[1], R2, P2, imageSize, CV_16SC2,
rmap[1][0], rmap[1][1]);

    Mat canvas;

    double sf;

    int w, h;

    if( !isVerticalStereo )

    {

        sf = 600./MAX(imageSize.width, imageSize.height);

        w = cvRound(imageSize.width*sf);

        h = cvRound(imageSize.height*sf);

        canvas.create(h, w*2, CV_8UC3);

    }

    else

    {

        sf = 300./MAX(imageSize.width, imageSize.height);

        w = cvRound(imageSize.width*sf);
        h = cvRound(imageSize.height*sf);
        canvas.create(h*2, w, CV_8UC3);
    }
    for( i = 0; i < nimages; i++ )
    {
        for( k = 0; k < 2; k++ )
        {
            Mat img = imread(goodImageList[i*2+k], 0), rimg, cimg;
            remap(img, rimg, rmap[k][0], rmap[k][1], INTER_LINEAR);
            cvtColor(rimg, cimg, COLOR_GRAY2BGR);
            Mat canvasPart = !isVerticalStereo ? canvas(Rect(w*k, 0, w, h)) : canvas(Rect(0, h*k, w,
h));
            resize(cimg, canvasPart, canvasPart.size(), 0, 0, INTER_AREA);
            if( useCalibrated )
            {
                Rect vroi(cvRound(validRoi[k].x*sf), cvRound(validRoi[k].y*sf),
                          cvRound(validRoi[k].width*sf), cvRound(validRoi[k].height*sf));
                rectangle(canvasPart, vroi, Scalar(0,0,255), 3, 8);
            }
        }
        if( !isVerticalStereo )
            for( j = 0; j < canvas.rows; j += 16 )
                line(canvas, Point(0, j), Point(canvas.cols, j), Scalar(0, 255, 0), 1, 8);
        else
            for( j = 0; j < canvas.cols; j += 16 )
                line(canvas, Point(j, 0), Point(j, canvas.rows), Scalar(0, 255, 0), 1, 8);
        imshow("rectified", canvas);
        char c = (char)waitKey();
        if( c == 27 || c == 'q' || c == 'Q' )
            break;
    }
}
static bool readStringList( const string& filename, vector<string>& l )
{
    l.resize(0);
    FileStorage fs(filename, FileStorage::READ);
    if( !fs.isOpened() )
        return false;
    FileNode n = fs.getFirstTopLevelNode();
    if( n.type() != FileNode::SEQ )
        return false;
    FileNodeIterator it = n.begin(), it_end = n.end();
    for( ; it != it_end; ++it )
        l.push_back((string)*it);
    return true;
}
int main(int argc, char** argv)
{
    Size boardSize;
    string imagelistfn;
    bool showRectified = true;
    for( int i = 1; i < argc; i++ )
    {
        if( string(argv[i]) == "-w" )
        {
            if( sscanf(argv[++i], "%d", &boardSize.width) != 1 || boardSize.width <= 0 )
            {
                cout << "invalid board width" << endl;
                return print_help();
            }
        }
        else if( string(argv[i]) == "-h" )
        {
            if( sscanf(argv[++i], "%d", &boardSize.height) != 1 || boardSize.height <= 0 )
            {
                cout << "invalid board height" << endl;
                return print_help();
            }
        }
        else if( string(argv[i]) == "-nr" )
            showRectified = false;
        else if( string(argv[i]) == "--help" )
            return print_help();
        else if( argv[i][0] == '-' )
        {
            cout << "invalid option " << argv[i] << endl;
            return 0;
        }
        else
            imagelistfn = argv[i];
    }
    if( imagelistfn == "" )
```

```
{
        imagelistfn = "stereo_calib.xml";
        boardSize = Size(9, 6);
    }
    else if( boardSize.width <= 0 || boardSize.height <= 0 )
    {
        cout << "if you specified XML file with chessboards, you should also specify the board
width and height (-w and -h options)" << endl;
        return 0;
    }
    vector<string> imagelist;
    bool ok = readStringList(imagelistfn, imagelist);
    if(!ok || imagelist.empty())
    {
        cout << "can not open " << imagelistfn << " or the string list is empty" << endl;
        return print_help();
    }
    StereoCalib(imagelist, boardSize, true, showRectified);
    return 0;
}
```

Makefile:

```
OBJS = stereo_calib.o
CC = g++
DEBUG = -g
INC_PATH = -I/home/pi/raspicam-0.1.3/src
CFLAGS = -Wall -c $(DEBUG) $(INC_PATH)
LFLAGS = -Wall $(DEBUG) -L/home/pi/raspicam-0.1.3/build/src
```

```
stereo_calib: $(OBJS)

	$(CC) $(LFLAGS) $(OBJS) -lopencv_calib3d -lopencv_xphoto -lopencv_imgcodecs -lopencv_core -lopencv_imgproc -lopencv_highgui -lopencv_ml -lraspicam -lraspicam_cv  -lopencv_features2d -lopencv_videoio -o stereo_calib

stereo_calib.o:stereo_calib.cpp

	$(CC) $(CFLAGS) stereo_calib.cpp

info:	@echo OBJS = $(OBJS)

	@echo CC = $(CC)info:

	@echo OBJS = $(OBJS)

	@echo CC = $(CC)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)

clean:

	rm -f $(OBJS)
```

Contours

Contours.cpp

```
#include "opencv2/highgui/highgui.hpp"

#include "opencv2/imgproc/imgproc.hpp"

#include <iostream>
```

```
#include <stdio.h>

#include <stdlib.h>

using namespace cv;

using namespace std;


Mat src; Mat src_gray;

int thresh = 100;

int max_thresh = 255;

RNG rng(12345);

/// Function header

void thresh_callback(int, void* );

/** @function main */

int main( int argc, char** argv )

{

  /// Load source image and convert it to gray

  src = imread( argv[1], 1 );

  /// Convert image to gray and blur it

  cvtColor( src, src_gray, CV_BGR2GRAY );

  blur( src_gray, src_gray, Size(3,3) );

  /// Create Window

  char* source_window = "Source";

  namedWindow( source_window, CV_WINDOW_AUTOSIZE );

  imshow( source_window, src );

  createTrackbar( " Canny thresh:", "Source", &thresh, max_thresh, thresh_callback );

  thresh_callback( 0, 0 );


  waitKey(0);
  return(0);

}

/** @function thresh_callback */

void thresh_callback(int, void* )

{

  Mat canny_output;

  vector<vector<Point> > contours;

  vector<Vec4i> hierarchy;

  /// Detect edges using canny

  Canny( src_gray, canny_output, thresh, thresh*2, 3 );

  /// Find contours

  findContours( canny_output, contours, hierarchy, CV_RETR_TREE,
CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );

  /// Draw contours

  Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );

  for( int i = 0; i< contours.size(); i++ )

     {

       Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );

       drawContours( drawing, contours, i, color, 2, 8, hierarchy, 0, Point() );

     }

  /// Show in a window

  namedWindow( "Contours", CV_WINDOW_AUTOSIZE );

  imshow( "Contours", drawing );

}
```

Makefile

```
OBJS = contours.o

CC = g++
DEBUG = -g

INC_PATH = -I/home/pi/raspicam-0.1.3/src

CFLAGS = -Wall -c $(DEBUG) $(INC_PATH)

LFLAGS = -Wall $(DEBUG) -L/home/pi/raspicam-0.1.3/build/src

contours: $(OBJS)

	$(CC) $(LFLAGS) $(OBJS) -lopencv_calib3d -lopencv_xphoto -lopencv_imgcodecs -
lopencv_core -lopencv_imgproc -lopencv_highgui -lopencv_ml -lraspicam -lraspicam_cv  -
lopencv_features2d -lopencv_videoio -o contours

contours.o: contours.cpp

	$(CC) $(CFLAGS) contours.cpp

info:

	@echo OBJS = $(OBJS)

	@echo CC = $(CC)info:

	@echo OBJS = $(OBJS)

	@echo CC = $(CC)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)

clean:

	rm -f $(OBJS)
```

Moments

moments.cpp

```
#include "opencv2/imgcodecs.hpp"

#include "opencv2/highgui/highgui.hpp"

#include "opencv2/imgproc/imgproc.hpp"

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

using namespace cv;

using namespace std;

Mat src; Mat src_gray;

int thresh = 100;

int max_thresh = 255;

RNG rng(12345);

/// Function header

void thresh_callback(int, void* );

/**

 * @function main

 */

int main( int, char** argv )

{

  /// Load source image and convert it to gray

  src = imread( argv[1], 1 );

  /// Convert image to gray and blur it

  cvtColor( src, src_gray, COLOR_BGR2GRAY );

  blur( src_gray, src_gray, Size(3,3) );

  /// Create Window
  const char* source_window = "Source";

  namedWindow( source_window, WINDOW_AUTOSIZE );

  imshow( source_window, src );

  createTrackbar( " Canny thresh:", "Source", &thresh, max_thresh, thresh_callback );

  thresh_callback( 0, 0 );

  waitKey(0);

  return(0);

}

/**

 * @function thresh_callback

 */

void thresh_callback(int, void* )

{

  Mat canny_output;

  vector<vector<Point> > contours;

  vector<Vec4i> hierarchy;

  /// Detect edges using canny

  Canny( src_gray, canny_output, thresh, thresh*2, 3 );

  /// Find contours

  findContours( canny_output, contours, hierarchy, RETR_TREE, CHAIN_APPROX_SIMPLE,
Point(0, 0) );

  /// Get the moments

  vector<Moments> mu(contours.size() );

  for( size_t i = 0; i < contours.size(); i++ )

     { mu[i] = moments( contours[i], false ); }

  ///  Get the mass centers:

  vector<Point2f> mc( contours.size() );

  for( size_t i = 0; i < contours.size(); i++ )
```

```
    { mc[i] = Point2f( static_cast<float>(mu[i].m10/mu[i].m00) ,
static_cast<float>(mu[i].m01/mu[i].m00) ); }

  /// Draw contours

  Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );

  for( size_t i = 0; i< contours.size(); i++ )

    {

      Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );

      drawContours( drawing, contours, (int)i, color, 2, 8, hierarchy, 0, Point() );

      circle( drawing, mc[i], 4, color, -1, 8, 0 );

    }

  /// Show in a window

  namedWindow( "Contours", WINDOW_AUTOSIZE );

  imshow( "Contours", drawing );

  /// Calculate the area with the moments 00 and compare with the result of the OpenCV
function

  printf("\t Info: Area and Contour Length \n");

  for( size_t i = 0; i< contours.size(); i++ )

    {

      printf(" * Contour[%d] - Area (M_00) = %.2f - Area OpenCV: %.2f - Length: %.2f \n", (int)i,
mu[i].m00, contourArea(contours[i]), arcLength( contours[i], true ) );

      printf(" el valor x: %f valor y: %f", mc[i].x,mc[i].y);

      Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );

      drawContours( drawing, contours, (int)i, color, 2, 8, hierarchy, 0, Point() );

      circle( drawing, mc[i], 4, color, -1, 8, 0 );

    }

}
```

Makefile

```
OBJS = moments.o

CC = g++

DEBUG = -g

INC_PATH = -I/home/pi/raspicam-0.1.3/src

CFLAGS = -Wall -c $(DEBUG) $(INC_PATH)

LFLAGS = -Wall $(DEBUG) -L/home/pi/raspicam-0.1.3/build/src

moments: $(OBJS)

	$(CC) $(LFLAGS) $(OBJS) -lopencv_calib3d -lopencv_xphoto -lopencv_imgcodecs -
lopencv_core -lopencv_imgproc -lopencv_highgui -lopencv_ml -lraspicam -lraspicam_cv  -
lopencv_features2d -lopencv_videoio -o moments

stereo_calib.o:moments.cpp

	$(CC) $(CFLAGS) moments.cpp

info:

	@echo OBJS = $(OBJS)

	@echo CC = $(CC)info:

	@echo OBJS = $(OBJS)

	@echo CC = $(CC)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)

clean:

	rm -f $(OBJS)
```

Big contour

Big_contour.cpp

```
#include <opencv2/imgproc.hpp>

#include <opencv2/highgui.hpp>

using namespace cv;

using namespace std;

int main( int argc, char** argv )

{

   Mat src = imread( argv[1] );

   int largest_area=0;

   int largest_contour_index=0;

   Rect bounding_rect;

   Mat thr;

   cvtColor( src, thr, COLOR_BGR2GRAY ); //Convert to gray

   threshold( thr, thr, 125, 255, THRESH_BINARY ); //Threshold the gray

   vector<vector<Point> > contours; // Vector for storing contours

   findContours( thr, contours, RETR_CCOMP, CHAIN_APPROX_SIMPLE ); // Find the contours
in the image

   for( size_t i = 0; i< contours.size(); i++ ) // iterate through each contour.

   {

      double area = contourArea( contours[i] );  //  Find the area of contour

      if( area > largest_area )

      {

         largest_area = area;

         largest_contour_index = i;               //Store the index of largest contour

         bounding_rect = boundingRect( contours[i] ); // Find the bounding rectangle for biggest
contour

      }
   }

   drawContours( src, contours,largest_contour_index, Scalar( 0, 255, 0 ), 2 ); // Draw the
largest contour using previously stored index.

   imshow( "result", src );

   waitKey();

   return 0;

}
```

Makefile:

```
OBJS = BigContour.o

CC = g++

DEBUG = -g

INC_PATH = -I/home/pi/raspicam-0.1.3/src

CFLAGS = -Wall -c $(DEBUG) $(INC_PATH)

LFLAGS = -Wall $(DEBUG) -L/home/pi/raspicam-0.1.3/build/src

BigContour: $(OBJS)

	$(CC) $(LFLAGS) $(OBJS) -lopencv_calib3d -lopencv_xphoto -lopencv_imgcodecs -
lopencv_core -lopencv_imgproc -lopencv_highgui -lopencv_ml -lraspicam -lraspicam_cv  -
lopencv_features2d -lopencv_videoio -o BigContour

BigContour.o:BigContour.cpp

	$(CC) $(CFLAGS) BigContour.cpp

info:

	@echo OBJS = $(OBJS)

	@echo CC = $(CC)info:

	@echo OBJS = $(OBJS)

	@echo CC = $(CC)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)
	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)

clean:

	rm -f $(OBJS)
```

Localización Pixel homogéneo

Grande.cpp

```
#include "opencv2/core.hpp"

#include <opencv2/core/utility.hpp>

#include "opencv2/opencv.hpp"

#include "opencv2/calib3d.hpp"

#include "opencv2/imgcodecs.hpp"

#include "opencv2/imgcodecs.hpp"

#include "opencv2/highgui/highgui.hpp"

#include "opencv2/imgproc/imgproc.hpp"

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

using namespace cv;

using namespace std;

int main( int argc, char** argv )

{

//Leer la matriz de cámara 1

 FileStorage fs2;
 fs2.open("outbueno.yml", FileStorage::READ);
 if (!fs2.isOpened())
   {
     cout << "failed to open"  << endl;
     return 1;
   }
 Mat P;
 Mat tvecs;
 Mat rvecs;
 fs2["camera_matrix"] >> P;
 fs2["rvecs"] >> rvecs;
 fs2["tvecs"] >> tvecs;
 cout << "camera matrix: " << P << endl;
 cout << "rvecs: " << rvecs << endl;
 cout << "tvecs: " << tvecs << endl;
fs2.release();
//Leer la matriz de cámara 2
 FileStorage fs3;
 fs3.open("outbueno1.yml", FileStorage::READ);
Mat P1;
Mat tvecs1;
Mat rvecs1;
fs3["camera_matrix"] >> P1;
fs3["rvecs"] >> rvecs1;
fs3["tvecs"] >> tvecs1;
cout << "camera matrix2: " << P1 << endl;
cout << "rvecs2: " << rvecs1 << endl;
cout << "tvecs2: " << tvecs1 << endl;

 cout << "camera matrix: " << P1.at<double>(0,0) << endl;

fs3.release();

 Mat rotmat;

 Mat rotmat1;

 Rodrigues(rvecs ,rotmat);

 Rodrigues(rvecs1 ,rotmat1);

cout << "rotmat: " << rotmat << endl;

cout << "rotmat1: " << rotmat1 << endl;

//abro las imagenes, en argv[1] leo la primera,

  Mat src = imread( argv[1] );

    int largest_area=0;

    int largest_contour_index=0;

    Rect bounding_rect;

    Mat thr;

    cvtColor( src, thr, COLOR_BGR2GRAY ); //Convert to gray

    threshold( thr, thr, 125, 255, THRESH_BINARY ); //Threshold the gray

    vector<vector<Point> > contours; // Vector for storing contours

    findContours( thr, contours, RETR_CCOMP, CHAIN_APPROX_SIMPLE ); // Find the contours
in the image

    for( size_t i = 0; i< contours.size(); i++ ) // iterate through each contour.

    {

        double area = contourArea( contours[i] );  //  Find the area of contour

        if( area > largest_area )

        {

            largest_area = area;

            largest_contour_index = i;               //Store the index of largest contour

        }
    }
 /// Get the moments
  vector<Moments> mu(contours.size() );
  for( size_t i = 0; i < contours.size(); i++ )
     { mu[i] = moments( contours[i], false ); }
  ///  Get the mass centers:
  vector<Point2f> mc( contours.size() );
  for( size_t i = 0; i < contours.size(); i++ )
     { mc[i] = Point2f( static_cast<float>(mu[i].m10/mu[i].m00) ,
static_cast<float>(mu[i].m01/mu[i].m00) ); }
    drawContours( src, contours,largest_contour_index, Scalar( 0, 255, 0 ), 2 );
    // en argv[2] leo la segunda.
 Mat src1 = imread( argv[2] );
    int largest_area2=0;
    int largest_contour_index1=0;
    Rect bounding_rect2;
    Mat thr1;
    cvtColor( src1, thr1, COLOR_BGR2GRAY ); //Convert to gray
    threshold( thr1, thr1, 125, 255, THRESH_BINARY ); //Threshold the gray
    vector<vector<Point> > contours1; // Vector for storing contours
    findContours( thr1, contours1, RETR_CCOMP, CHAIN_APPROX_SIMPLE ); // Find the
contours in the image
    for( size_t i = 0; i< contours1.size(); i++ ) // iterate through each contour.
    {
        double area2 = contourArea( contours1[i] );  //  Find the area of contour
        if( area2 > largest_area2 )
        {
            largest_area2 = area2;
```

```
          largest_contour_index1 = i;              //Store the index of largest contour

        }

    }

 /// Get the moments

  vector<Moments> mu1(contours.size() );

  for( size_t i = 0; i < contours.size(); i++ )

     { mu1[i] = moments( contours[i], false ); }

  ///  Get the mass centers:

  vector<Point2f> mc1( contours.size() );

  for( size_t i = 0; i < contours.size(); i++ )

     { mc1[i] = Point2f( static_cast<float>(mu1[i].m10/mu1[i].m00) ,
static_cast<float>(mu1[i].m01/mu1[i].m00) ); }

  drawContours( src1, contours1,largest_contour_index1, Scalar( 0, 255, 0 ), 2 );

// Mostrar por pantalla los resultados y los puntos centro

      printf(" el valor x: %f valor y: %f\n\n",
mc[largest_contour_index].x,mc[largest_contour_index].y);

      printf(" el valor x: %f valor y: %f\n\n",
mc[largest_contour_index1].x,mc[largest_contour_index1].y);

  ///////////////////////////////////////////////

  //Crear matrix de extrinsecos

  Mat Mext(3,4,CV_64FC1);

  Mext.at<double>(0,0)=rotmat.at<double>(0,0);

  Mext.at<double>(0,1)=rotmat.at<double>(0,1);

  Mext.at<double>(0,2)=rotmat.at<double>(0,2);

  Mext.at<double>(0,3)=tvecs.at<double>(0,0);

  Mext.at<double>(1,0)=rotmat.at<double>(1,0);

  Mext.at<double>(1,1)=rotmat.at<double>(1,1);

  Mext.at<double>(1,2)=rotmat.at<double>(1,2);

  Mext.at<double>(1,3)=tvecs.at<double>(0,1);
  Mext.at<double>(2,0)=rotmat.at<double>(2,0);

  Mext.at<double>(2,1)=rotmat.at<double>(2,1);

  Mext.at<double>(2,2)=rotmat.at<double>(2,2);

  Mext.at<double>(2,3)=tvecs.at<double>(0,2);

  // Mext.at<double>(3,0)=0;

  // Mext.at<double>(3,1)=0;

  // Mext.at<double>(3,2)=0;

  // Mext.at<double>(3,3)=1;

  cout << "Mext: " << Mext << endl;

  //Crear matrix de extrinsecos 2

  Mat Mext1(3,4,CV_64FC1);

  Mext1.at<double>(0,0)=rotmat1.at<double>(0,0);

  Mext1.at<double>(0,1)=rotmat1.at<double>(0,1);

  Mext1.at<double>(0,2)=rotmat1.at<double>(0,2);

  Mext1.at<double>(0,3)=tvecs1.at<double>(0,0);

  Mext1.at<double>(1,0)=rotmat1.at<double>(1,0);

  Mext1.at<double>(1,1)=rotmat1.at<double>(1,1);

  Mext1.at<double>(1,2)=rotmat1.at<double>(1,2);

  Mext1.at<double>(1,3)=tvecs1.at<double>(0,1);

  Mext1.at<double>(2,0)=rotmat1.at<double>(2,0);

  Mext1.at<double>(2,1)=rotmat1.at<double>(2,1);

  Mext1.at<double>(2,2)=rotmat1.at<double>(2,2);

  Mext1.at<double>(2,3)=tvecs1.at<double>(0,2);

  // Mext1.at<double>(3,0)=0;

  // Mext1.at<double>(3,1)=0;

  // Mext1.at<double>(3,2)=0;

  // Mext1.at<double>(3,3)=1;
```

```
 cout << "Mext1: " << Mext1 << endl;
//Calculamos P (lo llamaremos M) y P1 (lo llamaremos M1)
 Mat M(3,4,CV_64FC1);
 for(int i=0;i<3;i++){
   for(int j=0;j<4;j++){
           M.at<double>(i,j)=0;
           for(int k=0;k<3;k++){
             M.at<double>(i,j)=M.at<double>(i,j)+(P.at<double>(i,k)*Mext.at<double>(k,j));
        }
     }
}
 cout << "M " << M << endl;


Mat M1(3,4,CV_64FC1);
 for(int i=0;i<3;i++){
   for(int j=0;j<4;j++){
           M1.at<double>(i,j)=0;
           for(int k=0;k<3;k++){
             M1.at<double>(i,j)=M1.at<double>(i,j)+(P1.at<double>(i,k)*Mext1.at<double>(k,j));
        }
     }
 }
 cout << "M1 " << M1 << endl;
//Definimos los vectores
Mat v1(1,3,CV_64FC1);
  v1.at<double>(0,0)=1;
  v1.at<double>(0,1)=0;
  v1.at<double>(0,2)=-mc[largest_contour_index].x;
cout << "v1 " << v1 << endl;
Mat v2(1,3,CV_64FC1);
  v2.at<double>(0,0)=0;
  v2.at<double>(0,1)=1;
  v2.at<double>(0,2)=-mc[largest_contour_index].y;
cout << "v2 " << v2 << endl;
Mat v3(1,3,CV_64FC1);
  v3.at<double>(0,0)=1;
  v3.at<double>(0,1)=0;
  v3.at<double>(0,2)=-mc[largest_contour_index1].x;
cout << "v3 " << v3 << endl;
Mat v4(1,3,CV_64FC1);
  v4.at<double>(0,0)=0;
  v4.at<double>(0,1)=1;
  v4.at<double>(0,2)=-mc[largest_contour_index1].y;
cout << "v4 " << v4 << endl;
//Calculamos los valores de A
// v1 x M
 Mat A1(1,4,CV_64FC1);
 for(int f=0;f<1;f++){
    for(int c=0;c<4;c++){
      A1.at<double>(f,c)=0;
            for(int h=0;h<3;h++){
              A1.at<double>(f,c)=A1.at<double>(f,c)+(v1.at<double>(f,h)*M.at<double>(h,c));
        }
    }
 }
  cout << "A1 " << A1 << endl;
  cout<< "v1 (2.0)="  <<v1.at<double>(0,2)<<endl;
// v2 x M
 Mat A2(1,4,CV_64FC1);
 for(int f=0;f<1;f++){
    for(int c=0;c<4;c++){
      A2.at<double>(f,c)=0;
              for(int h=0;h<3;h++){
                A2.at<double>(f,c)=A2.at<double>(f,c)+(v2.at<double>(f,h)*M.at<double>(h,c));
          }
    }
 }
  cout << "A2 " << A2 << endl;
  cout<< "v2 (2.0)="  <<v2.at<double>(0,2)<<endl;
// v3 x M1
 Mat A3(1,4,CV_64FC1);;
 for(int f=0;f<1;f++){
    for(int c=0;c<4;c++){
      A3.at<double>(f,c)=0;
              for(int h=0;h<3;h++){
                A3.at<double>(f,c)=A3.at<double>(f,c)+(v3.at<double>(f,h)*M1.at<double>(h,c));
          }
    }
 }
  cout << "A3 " << A3 << endl;
cout<< "v3 (2.0)="  <<v3.at<double>(0,2)<<endl;
```

```
// v4 x M1


Mat A4(1,4,CV_64FC1);

 for(int f=0;f<1;f++){

    for(int c=0;c<4;c++){

      A4.at<double>(f,c)=0;

             for(int h=0;h<3;h++){

               A4.at<double>(f,c)=A4.at<double>(f,c)+(v4.at<double>(f,h)*M1.at<double>(h,c));

         }

    }

 }

  cout << "A4 " << A4 << endl;

cout<< "v4 (2.0)="  <<v4.at<double>(0,2)<<endl;

  //Montamos la matriz A

Mat A(4,4,CV_64FC1);

 A.at<double>(0,0)=A1.at<double>(0,0);

 A.at<double>(0,1)=A1.at<double>(0,1);

 A.at<double>(0,2)=A1.at<double>(0,2);

 A.at<double>(0,3)=A1.at<double>(0,3);

 A.at<double>(1,0)=A2.at<double>(0,0);

 A.at<double>(1,1)=A2.at<double>(0,1);

 A.at<double>(1,2)=A2.at<double>(0,2);

 A.at<double>(1,3)=A2.at<double>(0,3);

 A.at<double>(2,0)=A3.at<double>(0,0);

 A.at<double>(2,1)=A3.at<double>(0,1);

 A.at<double>(2,2)=A3.at<double>(0,2);

 A.at<double>(2,3)=A3.at<double>(0,3);
 A.at<double>(3,0)=A4.at<double>(0,0);
 A.at<double>(3,1)=A4.at<double>(0,1);
 A.at<double>(3,2)=A4.at<double>(0,2);
 A.at<double>(3,3)=A4.at<double>(0,3);
cout << "A " << A << endl;
 Mat  w, u, vt;
 SVD::compute(A,w,u,vt);
cout << "w " << w << endl;
cout << "u " << u << endl;
cout << "vt " << vt << endl;
 Mat X(4,1,CV_64FC1);
 X.at<double>(0,0)=vt.at<double>(3,0)/vt.at<double>(3,0);
 X.at<double>(1,0)=vt.at<double>(3,1)/vt.at<double>(3,1);
 X.at<double>(2,0)=vt.at<double>(3,2)/vt.at<double>(3,2);
 X.at<double>(3,0)=vt.at<double>(3,3)/vt.at<double>(3,3);
 // X.at<double>(0,0)=1;
 // X.at<double>(1,0)=1;
 // X.at<double>(2,0)=1;
 // X.at<double>(3,0)=1;
cout << "X " << X << endl;
//Metodo 2
//  Mat mi(3,1,CV_64FC1);
// for(int f=0;f<3;f++){
//    for(int c=0;c<1;c++){
//      mi.at<double>(f,c)=0.0;
//         for(int h=0;h<4;h++){
//           mi.at<double>(f,c)=mi.at<double>(f,c)+M.at<double>(f,h)*(X.at<double>(h,c));
//          }
//     }
//  }
//  cout << "mi " << mi << endl;
// Mat mii(3,1,CV_64FC1);
// mii.at<double>(0,0)=mi.at<double>(0,0)/mi.at<double>(0,2);
// mii.at<double>(0,1)=mi.at<double>(0,1)/mi.at<double>(0,2);
// mii.at<double>(0,2)=mi.at<double>(0,2)/mi.at<double>(0,2);
//  cout << "mii " << mii << endl;
//Comprobar que la X sea correcta
Mat Xtest(4,1,CV_64FC1);
 Xtest.at<double>(0,0)=vt.at<double>(3,0)/vt.at<double>(3,3);
 Xtest.at<double>(1,0)=vt.at<double>(3,1)/vt.at<double>(3,3);
 Xtest.at<double>(2,0)=vt.at<double>(3,2)/vt.at<double>(3,3);
 Xtest.at<double>(3,0)=vt.at<double>(3,3)/vt.at<double>(3,3);
cout << "Xtest " << Xtest << endl;
//Multiplicamos xi por P (M) (punto homogeneo 1)
Mat xi(3,1,CV_64FC1);
for(int f=0;f<3;f++){
    for(int c=0;c<1;c++){
      xi.at<double>(f,c)=0.0;
             for(int h=0;h<4;h++){
               xi.at<double>(f,c)=xi.at<double>(f,c)+M.at<double>(f,h)*(Xtest.at<double>(h,c));
             }
    }
 }
  cout << "xi " << xi << endl;
```

```
Mat xii(3,1,CV_64FC1);
 xii.at<double>(0,0)=xi.at<double>(0,0)/xi.at<double>(0,2);
 xii.at<double>(0,1)=xi.at<double>(0,1)/xi.at<double>(0,2);
 xii.at<double>(0,2)=xi.at<double>(0,2)/xi.at<double>(0,2);
  cout << "xii " << xii << endl;
  //Multiplicamos X por P1 (M1) (punto homogeneo 2)
Mat yi(3,1,CV_64FC1);
for(int f=0;f<3;f++){
    for(int c=0;c<1;c++){
      yi.at<double>(f,c)=0.0;
              for(int h=0;h<4;h++){
                yi.at<double>(f,c)=yi.at<double>(f,c)+(M1.at<double>(f,h)*Xtest.at<double>(h,c));
              }
    }
 }
  cout << "yi " << yi << endl;
Mat yii(3,1,CV_64FC1);
 yii.at<double>(0,0)=yi.at<double>(0,0)/yi.at<double>(0,2);
 yii.at<double>(0,1)=yi.at<double>(0,1)/yi.at<double>(0,2);
 yii.at<double>(0,2)=yi.at<double>(0,2)/yi.at<double>(0,2);
 cout << "yii " << yii << endl;
 //Con los 2 nuevos puntos homogeneos comprobamos si funciona bien el programa.
 // Definimos los vectores i
Mat v1i(1,3,CV_64FC1);
  v1i.at<double>(0,0)=1;
  v1i.at<double>(0,1)=0;
  // v1i.at<double>(0,2)=-xii.at<double>(0,0);
  v1i.at<double>(0,2)=-349;

cout << "v1i " << v1i << endl;

Mat v2i(1,3,CV_64FC1);

  v2i.at<double>(0,0)=0;

  v2i.at<double>(0,1)=1;

  // v2i.at<double>(0,2)=-xii.at<double>(0,1);

 v2i.at<double>(0,2)=-197;

cout << "v2i " << v2i << endl;

Mat v3i(1,3,CV_64FC1);

  v3i.at<double>(0,0)=1;

  v3i.at<double>(0,1)=0;

  // v3i.at<double>(0,2)=-yii.at<double>(0,0);

  v3i.at<double>(0,2)=-146;

cout << "v3i " << v3i << endl;

Mat v4i(1,3,CV_64FC1);

  v4i.at<double>(0,0)=0;

  v4i.at<double>(0,1)=1;

  // v4i.at<double>(0,2)=-yii.at<double>(0,1);

 v4i.at<double>(0,2)=-264;

cout << "v4i " << v4i << endl;

//Calculamos los valores de Ai

// v1i x M

 Mat A1i(1,4,CV_64FC1);;

 for(int f=0;f<1;f++){

    for(int c=0;c<4;c++){

      A1i.at<double>(f,c)=0.0;

             for(int h=0;h<3;h++){
               A1i.at<double>(f,c)=A1i.at<double>(f,c)+(v1i.at<double>(f,h)*M.at<double>(h,c));
         }
    }
 }
  cout << "A1i " << A1i << endl;
  cout<< "v1i" << v1i <<endl;
// v2i x M
 Mat A2i(1,4,CV_64FC1);;
 for(int f=0;f<1;f++){
    for(int c=0;c<4;c++){
      A2i.at<double>(f,c)=0.0;
             for(int h=0;h<3;h++){
               A2i.at<double>(f,c)=A2i.at<double>(f,c)+(v2i.at<double>(f,h)*M.at<double>(h,c));
         }
    }
 }
  cout << "A2i " << A2i << endl;
 cout<< "v2i" << v2i <<endl;
// v3i x M1
 Mat A3i(1,4,CV_64FC1);;
 for(int f=0;f<1;f++){
    for(int c=0;c<4;c++){
      A3i.at<double>(f,c)=0.0;
             for(int h=0;h<3;h++){
               A3i.at<double>(f,c)=A3i.at<double>(f,c)+(v3i.at<double>(f,h)*M1.at<double>(h,c));
         }
    }
 }
  cout << "A3i " << A3i << endl;
 cout<< "v3i" << v3i <<endl;
// v4i x M1
Mat A4i(1,4,CV_64FC1);
 for(int f=0;f<1;f++){
    for(int c=0;c<4;c++){
      A4i.at<double>(f,c)=0.0;
             for(int h=0;h<3;h++){
               A4i.at<double>(f,c)=A4i.at<double>(f,c)+(v4i.at<double>(f,h)*M1.at<double>(h,c));
         }
    }
 }
  cout << "A4i " << A4i << endl;
 cout<< "v4i" << v4i <<endl;
  //montamos Ai
Mat Ai(4,4,CV_64FC1);
 Ai.at<double>(0,0)=A1i.at<double>(0,0);
 Ai.at<double>(0,1)=A1i.at<double>(0,1);
 Ai.at<double>(0,2)=A1i.at<double>(0,2);
 Ai.at<double>(0,3)=A1i.at<double>(0,3);
 Ai.at<double>(1,0)=A2i.at<double>(0,0);
 Ai.at<double>(1,1)=A2i.at<double>(0,1);
 Ai.at<double>(1,2)=A2i.at<double>(0,2);
 Ai.at<double>(1,3)=A2i.at<double>(0,3);
 Ai.at<double>(2,0)=A3i.at<double>(0,0);
 Ai.at<double>(2,1)=A3i.at<double>(0,1);
 Ai.at<double>(2,2)=A3i.at<double>(0,2);

 Ai.at<double>(2,3)=A3i.at<double>(0,3);

 Ai.at<double>(3,0)=A4i.at<double>(0,0);

 Ai.at<double>(3,1)=A4i.at<double>(0,1);

 Ai.at<double>(3,2)=A4i.at<double>(0,2);

 Ai.at<double>(3,3)=A4i.at<double>(0,3);

cout << "Ai " << Ai << endl;

 Mat  wi, ui, vti;

 SVD::compute(Ai,wi,ui,vti);

cout << "wi " << wi << endl;

cout << "ui " << ui << endl;

cout << "vti " << vti << endl;

Mat vtii(4,1,CV_64FC1);

 vtii.at<double>(0,0)=vti.at<double>(3,0)/vti.at<double>(3,3);

 vtii.at<double>(1,0)=vti.at<double>(3,1)/vti.at<double>(3,3);

 vtii.at<double>(2,0)=vti.at<double>(3,2)/vti.at<double>(3,3);

 vtii.at<double>(3,0)=vti.at<double>(3,3)/vti.at<double>(3,3);


  cout << "vtii " << vtii << endl;

//   //Comprobar que X sea correcta.

//  Mat Xj(4,1,CV_64FC1);

//  // Xj.at<double>(0,0)=1;

//  // Xj.at<double>(1,0)=1;

//  // Xj.at<double>(2,0)=1;

//  // Xj.at<double>(3,0)=1;

//  Xj.at<double>(0,0)=vti.at<double>(3,0)/vti.at<double>(3,3);

//  Xj.at<double>(1,0)=vti.at<double>(3,1)/vti.at<double>(3,3);
//  Xj.at<double>(2,0)=vti.at<double>(3,2)/vti.at<double>(3,3);
//  Xj.at<double>(3,0)=vti.at<double>(3,3)/vti.at<double>(3,3);
// cout << "Xj " << Xj << endl;
// //Multiplicamos xj por P (M) (punto homogeneo 1)
// Mat xk(3,1,CV_64FC1);
// for(int f=0;f<3;f++){
//    for(int c=0;c<1;c++){
//      xk.at<double>(f,c)=0.0;
//         for(int h=0;h<4;h++){
//           xk.at<double>(f,c)=xk.at<double>(f,c)+M.at<double>(f,h)*(Xj.at<double>(h,c));
//         }
//    }
//  }
//   cout << "xk " << xk << endl;
// Mat xkk(3,1,CV_64FC1);
//  xkk.at<double>(0,0)=xk.at<double>(0,0)/xk.at<double>(0,2);
//  xkk.at<double>(0,1)=xk.at<double>(0,1)/xk.at<double>(0,2);
//  xkk.at<double>(0,2)=xk.at<double>(0,2)/xk.at<double>(0,2);
//   cout << "xkk " << xkk << endl;
//   //Multiplicamos Xj por P1 (M1) (punto homogeneo 2)
// Mat yk(3,1,CV_64FC1);
// for(int f=0;f<3;f++){
//    for(int c=0;c<1;c++){
//      yk.at<double>(f,c)=0.0;
//         for(int h=0;h<4;h++){
//           yk.at<double>(f,c)=yk.at<double>(f,c)+(M1.at<double>(f,h)*Xj.at<double>(h,c));
//         }
//    }
//  }
//   cout << "yk " << yk << endl;
// Mat ykk(3,1,CV_64FC1);
//  ykk.at<double>(0,0)=yk.at<double>(0,0)/yk.at<double>(0,2);
//  ykk.at<double>(0,1)=yk.at<double>(0,1)/yk.at<double>(0,2);
//  ykk.at<double>(0,2)=yk.at<double>(0,2)/yk.at<double>(0,2);
//  cout << "ykk " << ykk << endl;
//  ///////////////////////////////////////////////////////////////////////////////
 imshow( "result", src );
 imshow( "result2", src1 );
 waitKey();
 return 0;
}
```

Makefile:

```
OBJS = Grande.o
CC = g++
DEBUG = -g
INC_PATH = -I/home/pi/raspicam-0.1.3/src
CFLAGS = -Wall -c $(DEBUG) $(INC_PATH)
LFLAGS = -Wall $(DEBUG) -L/home/pi/raspicam-0.1.3/build/src
Grande: $(OBJS)
	$(CC) $(LFLAGS) $(OBJS) -lopencv_calib3d -lopencv_xphoto -lopencv_imgcodecs -
lopencv_core -lopencv_imgproc -lopencv_highgui -lopencv_ml -lraspicam -lraspicam_cv  -
lopencv_features2d -lopencv_videoio -o Grande
Grande.o:Grande.cpp
	$(CC) $(CFLAGS) Grande.cpp
info:
	@echo OBJS = $(OBJS)

	@echo CC = $(CC)info:

	@echo OBJS = $(OBJS)

	@echo CC = $(CC)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)

clean:

	rm -f $(OBJS)
```

Simulación Filtro de Kalman

fil.cpp

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>


double frand() {

   return 2*((rand()/(double)RAND_MAX) - 0.5

);

}


int main() {
   float z_real_x = 0.5; //El valor real x que queremos medir

   float z_real_y = 1; //El valor real y que queremos medir

   float z_real_z = 0; //El valor real z que queremos medir


   //Valores iniciales para el filtro de Kalman (simularemos graficamente como si el primer
valor medido fuese el primer valor de Kalman)

   float x_est_last = z_real_x + frand()*0.03;

   float y_est_last = z_real_y + frand()*0.03;

   float z_est_last = z_real_z + frand()*0.03;

   float P_last_x =  z_real_x + frand()*0.03;

   float P_last_y = z_real_y + frand()*0.03;;

   float P_last_z = z_real_z + frand()*0.03;;


   //Ruido en el sistema

   float Q = 0.4;

   float R = 0.6;



   float K;

   float P;

   float P_temp;

   float x_temp_est;

   float x_est;

   float z_measured; //el valor del ruido que medimos


   srand(0);
```

```
///////////////////XXXXXXXXXXXXXXXXXXXX//////////////////
//inicializar la medida
x_est_last = z_real_x + frand()*0.03;

float sum_error_kalman = 0;
float sum_error_measure = 0;

  printf("X Original: %6.3f \n",z_real_x);
  printf("Medido: %6.3f [diferencia:0]\n",x_est_last);
  printf("Kalman: %6.3f [diferencia:0]\n",x_est_last);

for (int i=0;i<20;i++) {
  //Predicción
  x_temp_est = x_est_last;
  P_temp = P_last_x + Q;
  //Calculamos la ganancia de Kalman
  K = P_temp * (1.0/(P_temp + R));
  //Medido
  z_measured = z_real_x + frand()*0.03; //Valor real+ruido
  //correcto
  x_est = x_temp_est + K * (z_measured - x_temp_est);
  P = (1- K) * P_temp;

  //Tenemos nuestro nuevo sistema

  printf("X Original: %6.3f \n",z_real_x);
  printf("Medido: %6.3f [diferencia:%.3f]\n",z_measured,fabs(z_real_x-z_measured));
```

```
        printf("Kalman: %6.3f [diferencia:%.3f]\n",x_est,fabs(z_real_x - x_est));


        sum_error_kalman += fabs(z_real_x - x_est);

        sum_error_measure += fabs(z_real_x-z_measured);


        //Actualizamos los "last"

        P_last_x = P;

        x_est_last = x_est;

    }


    printf("Error Total sin kalman:  %f\n",sum_error_measure);

    printf("Error total con kalman: %f\n",sum_error_kalman);

    printf("Porcentaje de error reducido: %d%% \n",100-
(int)((sum_error_kalman/sum_error_measure)*100));


    ////////////////////YYYYYYYYYYYYY/////////////////


//inicializar la medida



    y_est_last = z_real_y + frand()*0.03;


    float sum_error_kalman2 = 0;

    float sum_error_measure2 = 0;


    printf("Y Original: %6.3f \n",z_real_x);

    printf("Medido: %6.3f [diferencia:0]\n",y_est_last);

    printf("Kalman: %6.3f [diferencia:0]\n",y_est_last);
```

```
  for (int i=0;i<20;i++) {

    //Predicción

    x_temp_est = y_est_last;

    P_temp = P_last_y + Q;

    //Calculamos la ganancia de Kalman

    K = P_temp * (1.0/(P_temp + R));

    //Medido

    z_measured = z_real_y + frand()*0.03; //Valor real+ruido

    //correcto


    x_est = x_temp_est + K * (z_measured - x_temp_est);

    P = (1- K) * P_temp;

    //Tenemos nuestro nuevo sistema


    printf("Y Original: %6.3f \n",z_real_y);

    printf("Medido: %6.3f [diferencia:%.3f]\n",z_measured,fabs(z_real_y-z_measured));

    printf("Kalman: %6.3f [diferencia:%.3f]\n",x_est,fabs(z_real_y - x_est));


    sum_error_kalman2 += fabs(z_real_y - x_est);

    sum_error_measure2

+= fabs(z_real_y-z_measured);


    //Actualizamos los "last"

    P_last_y = P;

    x_est_last = x_est;

  }
```

```
    printf("Error Total sin kalman:  %f\n",sum_error_measure2);

    printf("Error total con kalman: %f\n",sum_error_kalman2);

    printf("Porcentaje de error reducido: %d%% \n",100-
(int)((sum_error_kalman2/sum_error_measure2)*100));


//    ///////////////////ZZZZZZZZZZZZZZZZZZZZ//////////////////////

//inicializar la medida

    z_est_last = z_real_z + frand()*0.03;


    float sum_error_kalman3 = 0;

    float sum_error_measure3 = 0;

    printf("Z Original: %6.3f \n",z_real_x);

    printf("Medido: %6.3f [diferencia:0]\n",z_est_last);

    printf("Kalman: %6.3f [diferencia:0]\n",z_est_last);


    for (int i=0;i<20;i++) {

        //Predicción

        x_temp_est = z_est_last;

        P_temp = P_last_z + Q;

        //Calculamos la ganancia de Kalman

        K = P_temp * (1.0/(P_temp + R));

        //Medido

        z_measured = z_real_z + frand()*0.03; //Valor real+ruido

        //correcto

        x_est = x_temp_est + K * (z_measured - x_temp_est);

        P = (1- K) * P_temp;

        //Tenemos nuestro nuevo sistema
        printf("Z Original: %6.3f \n",z_real_z);

        printf("Medido: %6.3f [diferencia:%.3f]\n",z_measured,fabs(z_real_z-z_measured));

        printf("Kalman: %6.3f [diferencia:%.3f]\n",x_est,fabs(z_real_z - x_est));

        sum_error_kalman3 += fabs(z_real_z - x_est);

        sum_error_measure3 += fabs(z_real_z-z_measured);

        //Actualizamos los "last"

        P_last_z = P;

        z_est_last = x_est;

    }

    ////////////////////////////////////////////

    printf("Error Total sin kalman:  %f\n",sum_error_measure3);

    printf("Error total con kalman: %f\n",sum_error_kalman3);

    printf("Porcentaje de error reducido: %d%% \n",100-
(int)((sum_error_kalman3/sum_error_measure3)*100));


    return 0;

}
```

Makefile

```
OBJS = fil.o

CC = g++

DEBUG = -g

INC_PATH = -I/home/pi/raspicam-0.1.3/src

CFLAGS = -Wall -c $(DEBUG) $(INC_PATH)

LFLAGS = -Wall $(DEBUG) -L/home/pi/raspicam-0.1.3/build/src


fil: $(OBJS)
```

```
	$(CC) $(LFLAGS) $(OBJS) -lopencv_calib3d -lopencv_xphoto -lopencv_imgcodecs -
lopencv_core -lopencv_imgproc -lopencv_highgui -lopencv_ml -lraspicam -lraspicam_cv  -
lopencv_features2d -lopencv_videoio -lopencv_video -o fil


fil.o:fil.cpp

	$(CC) $(CFLAGS) fil.cpp


info:

	@echo OBJS = $(OBJS)

	@echo CC = $(CC)info:

	@echo OBJS = $(OBJS)

	@echo CC = $(CC)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)

	@echo DEBUG = $(DEBUG)

	@echo INC_PATH = $(INC_PATH)

	@echo CFLAGS = $(CFLAGS)

	@echo LFLAGS = $(LFLAGS)


clean:

	rm -f $(OBJS)
```

CHAPTER 11
Camera Models and Calibration

Vision begins with the detection of light from the world. That light begins as rays emanating from some source (e.g., a light bulb or the sun), which then travels through space until striking some object. When that light strikes the object, much of the light is absorbed, and what is not absorbed we perceive as the color of the light. Reflected light that makes its way to our eye (or our camera) is collected on our retina (or our imager). The geometry of this arrangement—particularly of the ray's travel from the object, through the lens in our eye or camera, and to the retina or imager—is of particular importance to practical computer vision.

A simple but useful model of how this happens is the pinhole camera model.* A *pinhole* is an imaginary wall with a tiny hole in the center that blocks all rays except those passing through the tiny aperture in the center. In this chapter, we will start with a pinhole camera model to get a handle on the basic geometry of projecting rays. Unfortunately, a real pinhole is not a very good way to make images because it does not gather enough light for rapid exposure. This is why our eyes and cameras use lenses to gather more light than what would be available at a single point. The downside, however, is that gathering more light with a lens not only forces us to move beyond the simple geometry of the pinhole model but also introduces distortions from the lens itself.

In this chapter we will learn how, using *camera calibration*, to correct (mathematically) for the main deviations from the simple pinhole model that the use of lenses imposes on us. Camera calibration is important also for relating camera measurements with measurements in the real, three-dimensional world. This is important because scenes are not only three-dimensional; they are also physical spaces with physical units. Hence, the relation between the camera's natural units (pixels) and the units of the

* Knowledge of lenses goes back at least to Roman times. The pinhole camera model goes back at least 987 years to al-Hytham [1021] and is the classic way of introducing the geometric aspects of vision. Mathematical and physical advances followed in the 1600s and 1700s with Descartes, Kepler, Galileo, Newton, Hooke, Euler, Fermat, and Snell (see O'Connor [O'Connor02]). Some key modern texts for geometric vision include those by Trucco [Trucco98], Jaehne (also sometimes spelled Jähne) [Jaehne95; Jaehne97], Hartley and Zisserman [Hartley06], Forsyth and Ponce [Forsyth03], Shapiro and Stockman [Shapiro02], and Xu and Zhang [Xu96].

physical world (e.g., meters) is a critical component in any attempt to reconstruct a three-dimensional scene.

The process of camera calibration gives us both a model of the camera's geometry and a *distortion* model of the lens. These two informational models define the *intrinsic parameters* of the camera. In this chapter we use these models to correct for lens distortions; in Chapter 12, we will use them to interpret a physical scene.

We shall begin by looking at camera models and the causes of lens distortion. From there we will explore the *homography transform*, the mathematical instrument that allows us to capture the effects of the camera's basic behavior and of its various distortions and corrections. We will take some time to discuss exactly how the transformation that characterizes a particular camera can be calculated mathematically. Once we have all this in hand, we'll move on to the OpenCV function that does most of this work for us.

Just about all of this chapter is devoted to building enough theory that you will truly understand what is going into (and what is coming out of) the OpenCV function `cvCalibrateCamera2()` as well as what that function is doing "under the hood". This is important stuff if you want to use the function responsibly. Having said that, if you are already an expert and simply want to know how to use OpenCV to do what you already understand, jump right ahead to the "Calibration Function" section and get to it.

Camera Model

We begin by looking at the simplest model of a camera, the pinhole camera model. In this simple model, light is envisioned as entering from the scene or a distant object, but only a single ray enters from any particular point. In a physical pinhole camera, this point is then "projected" onto an imaging surface. As a result, the image on this *image plane* (also called the *projective plane*) is always in focus, and the size of the image relative to the distant object is given by a single parameter of the camera: its *focal length*. For our idealized pinhole camera, the distance from the pinhole aperture to the screen is precisely the focal length. This is shown in Figure 11-1, where f is the focal length of the camera, Z is the distance from the camera to the object, X is the length of the object, and x is the object's image on the imaging plane. In the figure, we can see by similar triangles that $-x/f = X/Z$, or

$$-x = f\frac{X}{Z}$$

We shall now rearrange our pinhole camera model to a form that is equivalent but in which the math comes out easier. In Figure 11-2, we swap the pinhole and the image plane.* The main difference is that the object now appears rightside up. The point in the pinhole is reinterpreted as the *center of projection*. In this way of looking at things, every

* Typical of such mathematical abstractions, this new arrangement is not one that can be built physically; the image plane is simply a way of thinking of a "slice" through all of those rays that happen to strike the center of projection. This arrangement is, however, much easier to draw and do math with.

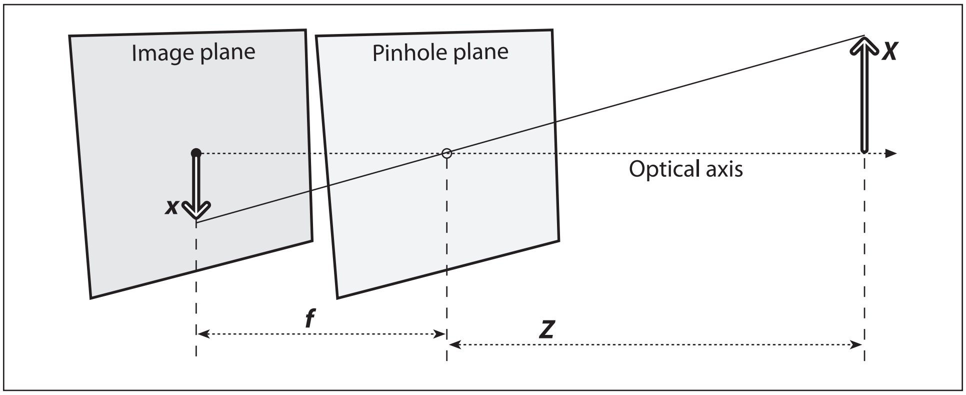


Figure 11-1. Pinhole camera model: a pinhole (the pinhole aperture) lets through only those light rays that intersect a particular point in space; these rays then form an image by "projecting" onto an image plane

ray leaves a point on the distant object and heads for the center of projection. The point at the intersection of the image plane and the optical axis is referred to as the *principal point*. On this new frontal image plane (see Figure 11-2), which is the equivalent of the old projective or image plane, the image of the distant object is exactly the same size as it was on the image plane in Figure 11-1. The image is generated by intersecting these rays with the image plane, which happens to be exactly a distance f from the center of projection. This makes the similar triangles relationship $x/f = X/Z$ more directly evident than before. The negative sign is gone because the object image is no longer upside down.

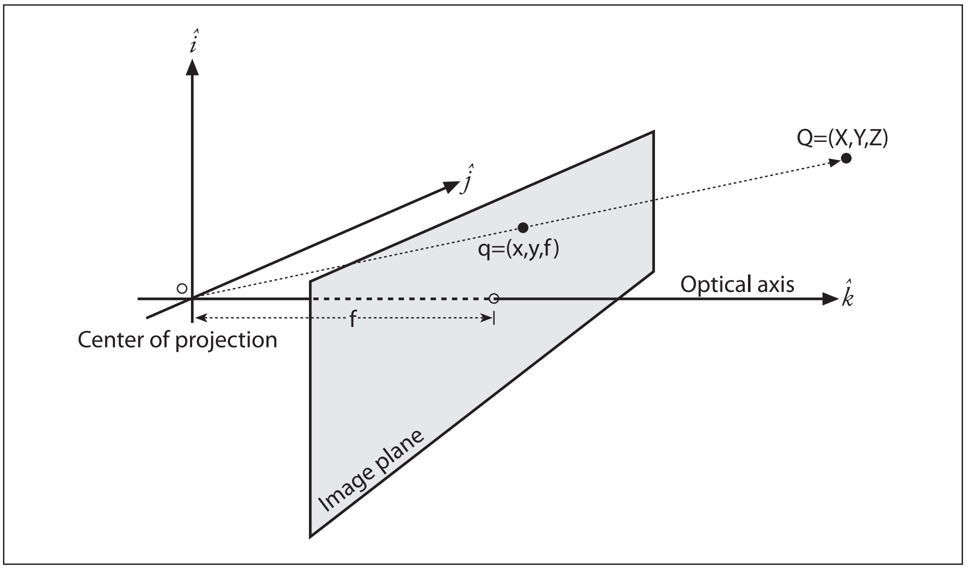


Figure 11-2. A point Q = (X, Y, Z) is projected onto the image plane by the ray passing through the center of projection, and the resulting point on the image is q = (z, y, f); the image plane is really just the projection screen "pushed" in front of the pinhole (the math is equivalent but simpler this way)

You might think that the principle point is equivalent to the center of the imager; yet this would imply that some guy with tweezers and a tube of glue was able to attach the imager in your camera to micron accuracy. In fact, the center of the chip is usually not on the optical axis. We thus introduce two new parameters, c_x and c_y, to model a possible displacement (away from the optic axis) of the center of coordinates on the projection screen. The result is that a relatively simple model in which a point Q in the physical world, whose coordinates are (X, Y, Z), is projected onto the screen at some pixel location given by $(x_{\text{screen}}, y_{\text{screen}})$ in accordance with the following equations:*

$$x_{\text{screen}} = f_x\left(\frac{X}{Z}\right)+c_x, \qquad y_{\text{screen}} = f_y\left(\frac{Y}{Z}\right)+c_y$$

Note that we have introduced two different focal lengths; the reason for this is that the individual pixels on a typical low-cost imager are rectangular rather than square. The focal length f_x (for example) is actually the product of the physical focal length of the lens and the size s_x of the individual imager elements (this should make sense because s_x has units of pixels per millimeter† while F has units of millimeters, which means that f_x is in the required units of pixels). Of course, similar statements hold for f_y and s_y. It is important to keep in mind, though, that s_x and s_y cannot be measured directly via any camera calibration process, and neither is the physical focal length F directly measurable. Only the combinations $f_x = Fs_x$ and $f_y = Fs_y$ can be derived without actually dismantling the camera and measuring its components directly.

Basic Projective Geometry

The relation that maps the points Q_i in the physical world with coordinates (X_i, Y_i, Z_i) to the points on the projection screen with coordinates (x_i, y_i) is called a *projective transform*. When working with such transforms, it is convenient to use what are known as *homogeneous coordinates*. The homogeneous coordinates associated with a point in a projective space of dimension n are typically expressed as an $(n + 1)$-dimensional vector (e.g., x, y, z becomes x, y, z, w), with the additional restriction that any two points whose values are proportional are equivalent. In our case, the image plane is the projective space and it has two dimensions, so we will represent points on that plane as three-dimensional vectors $q = (q_1, q_2, q_3)$. Recalling that all points having proportional values in the projective space are equivalent, we can recover the actual pixel coordinates by dividing through by q_3. This allows us to arrange the parameters that define our camera (i.e., f_x, f_y, c_x, and c_y) into a single 3-by-3 matrix, which we will call the *camera intrinsics matrix* (the approach OpenCV takes to camera intrinsics is derived from Heikkila and

* Here the subscript "screen" is intended to remind you that the coordinates being computed are in the coordinate system of the screen (i.e., the imager). The difference between $(x_{\text{screen}}, y_{\text{screen}})$ in the equation and (x, y) in Figure 11-2 is precisely the point of c_x and c_y. Having said that, we will subsequently drop the "screen" subscript and simply use lowercase letters to describe coordinates on the imager.

† Of course, "millimeter" is just a stand-in for any physical unit you like. It could just as easily be "meter," "micron," or "furlong." The point is that s_x converts physical units to pixel units.

Silven [Heikkila97]). The projection of the points in the physical world into the camera is now summarized by the following simple form:

$$q = MQ, \quad \text{where} \quad q = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad Q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Multiplying this out, you will find that $w = Z$ and so, since the point q is in homogeneous coordinates, we should divide through by w (or Z) in order to recover our earlier definitions. (The minus sign is gone because we are now looking at the noninverted image on the projective plane in front of the pinhole rather than the inverted image on the projection screen behind the pinhole.)

While we are on the topic of homogeneous coordinates, there is a function in the OpenCV library which would be appropriate to introduce here: `cvConvertPointsHomogenious()`* is handy for converting to and from homogeneous coordinates; it also does a bunch of other useful things.

```
void cvConvertPointsHomogenious(
  const CvMat* src,
  CvMat*       dst
);
```

Don't let the simple arguments fool you; this routine does a whole lot of useful stuff. The input array `src` can be M_{scr}-by-N or N-by-M_{scr} (for $M_{scr} = 2$, 3, or 4); it can also be 1-by-N or N-by-1, with the array having $M_{scr} = 2$, 3, or 4 channels (N can be any number; it is essentially the number of points that you have stuffed into the matrix `src` for conversion). The output array `dst` can be any of these types as well, with the additional restriction that the dimensionality M_{dst} must be equal to M_{scr}, $M_{scr} - 1$, or $M_{scr} + 1$.

When the input dimension M_{scr} is equal to the output dimension M_{dst}, the data is simply copied (and, if necessary, transposed). If $M_{scr} > M_{dst}$, then the elements in `dst` are computed by dividing all but the last elements of the corresponding vector from `src` by the last element of that same vector (i.e., `src` is assumed to contain homogeneous coordinates). If $M_{scr} < M_{dst}$, then the points are copied but with a 1 being inserted into the final coordinate of every vector in the `dst` array (i.e., the vectors in `src` are extended to homogeneous coordinates). In these cases, just as in the trivial case of $M_{scr} = M_{dst}$, any necessary transpositions are also done.

One word of warning about this function is that there can be cases (when $N < 5$) where the input and output dimensionality are ambiguous. In this event, the function will throw an error. If you find yourself in this situation, you can just pad out the matrices with some bogus values. Alternatively, the user may pass multichannel N-by-1 or 1-by-N matrices, where the number of channels is M_{scr} (M_{dst}). The function `cvReshape()` can be used to convert single-channel matrices to multichannel ones without copying any data.

* Yes, "Homogenious" in the function name is misspelled.

With the ideal pinhole, we have a useful model for some of the three-dimensional geometry of vision. Remember, however, that very little light goes through a pinhole; thus, in practice such an arrangement would make for very slow imaging while we wait for enough light to accumulate on whatever imager we are using. For a camera to form images at a faster rate, we must gather a lot of light over a wider area and bend (i.e., focus) that light to converge at the point of projection. To accomplish this, we use a lens. A lens can focus a large amount of light on a point to give us fast imaging, but it comes at the cost of introducing distortions.

Lens Distortions

In theory, it is possible to define a lens that will introduce no distortions. In practice, however, no lens is perfect. This is mainly for reasons of manufacturing; it is much easier to make a "spherical" lens than to make a more mathematically ideal "parabolic" lens. It is also difficult to mechanically align the lens and imager exactly. Here we describe the two main lens distortions and how to model them.* *Radial distortions* arise as a result of the shape of lens, whereas *tangential distortions* arise from the assembly process of the camera as a whole.

We start with radial distortion. The lenses of real cameras often noticeably distort the location of pixels near the edges of the imager. This bulging phenomenon is the source of the "barrel" or "fish-eye" effect (see the room-divider lines at the top of Figure 11-12 for a good example). Figure 11-3 gives some intuition as to why radial distortion occurs. With some lenses, rays farther from the center of the lens are bent more than those closer in. A typical inexpensive lens is, in effect, stronger than it ought to be as you get farther from the center. Barrel distortion is particularly noticeable in cheap web cameras but less apparent in high-end cameras, where a lot of effort is put into fancy lens systems that minimize radial distortion.

For radial distortions, the distortion is 0 at the (optical) center of the imager and increases as we move toward the periphery. In practice, this distortion is small and can be characterized by the first few terms of a Taylor series expansion around $r = 0$.† For cheap web cameras, we generally use the first two such terms; the first of which is conventionally called k_1 and the second k_2. For highly distorted cameras such as fish-eye lenses we can use a third radial distortion term k_3. In general, the radial location of a point on the imager will be rescaled according to the following equations:

* The approach to modeling lens distortion taken here derives mostly from Brown [Brown71] and earlier Fryer and Brown [Fryer86].

† If you don't know what a Taylor series is, don't worry too much. The Taylor series is a mathematical technique for expressing a (potentially) complicated function in the form of a polynomial of similar value to the approximated function in at least a small neighborhood of some particular point (the more terms we include in the polynomial series, the more accurate the approximation). In our case we want to expand the distortion function as a polynomial in the neighborhood of $r = 0$. This polynomial takes the general form $f(r) = a_0 + a_1r + a_2r^2 + \cdots$, but in our case the fact that $f(r) = 0$ at $r = 0$ implies $a_0 = 0$. Similarly, because the function must be symmetric in r, only the coefficients of even powers of r will be nonzero. For these reasons, the only parameters that are necessary for characterizing these radial distortions are the coefficients of r^2, r^4, and (sometimes) r^6.

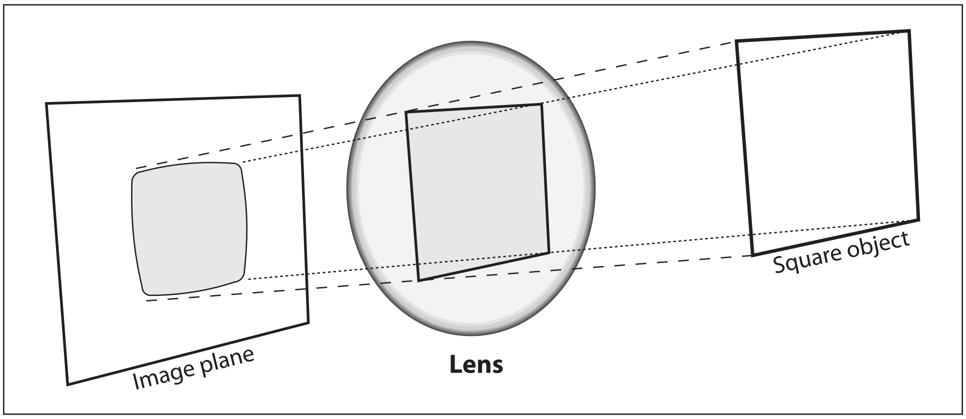


Figure 11-3. Radial distortion: rays farther from the center of a simple lens are bent too much compared to rays that pass closer to the center; thus, the sides of a square appear to bow out on the image plane (this is also known as barrel distortion)

$$x_{\text{corrected}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$
$$y_{\text{corrected}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Here, (x, y) is the original location (on the imager) of the distorted point and $(x_{\text{corrected}}, y_{\text{corrected}})$ is the new location as a result of the correction. Figure 11-4 shows displacements of a rectangular grid that are due to radial distortion. External points on a front-facing rectangular grid are increasingly displaced inward as the radial distance from the optical center increases.

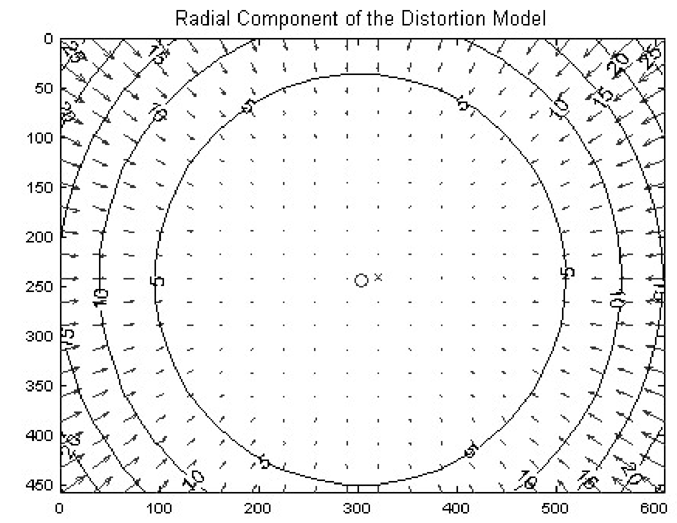
The second-largest common distortion is *tangential distortion*. This distortion is due to manufacturing defects resulting from the lens not being exactly parallel to the imaging plane; see Figure 11-5.

Tangential distortion is minimally characterized by two additional parameters, p_1 and p_2, such that:*

$$x_{\text{corrected}} = x + [2p_1 y + p_2(r^2 + 2x^2)]$$
$$y_{\text{corrected}} = y + [p_1(r^2 + 2y^2) + 2p_2 x]$$

Thus in total there are five distortion coefficients that we require. Because all five are necessary in most of the OpenCV routines that use them, they are typically bundled into one *distortion vector*; this is just a 5-by-1 matrix containing k_1, k_2, p_1, p_2, and k_3 (in that order). Figure 11-6 shows the effects of tangential distortion on a front-facing external rectangular grid of points. The points are displaced elliptically as a function of location and radius.

* The derivation of these equations is beyond the scope of this book, but the interested reader is referred to the "plumb bob" model; see D. C. Brown, "Decentering Distortion of Lenses", *Photometric Engineering* 32(3) (1966), 444–462.



```
Pixel error            = [0.1174, 0.1159]
Focal Length           = (657.303, 657.744)              +/- [0.2849, 0.2894]
Principal Point        = (302.717, 242.334)              +/- [0.5912, 0.5571]
Skew                   = 0.0004198                            +/- 0.0001905
Radial coefficients    = (-0.2535, 0.1187, 0)         +/- [0.00231, 0.009418, 0]
Tangential coefficients = (-0.0002789, 5.174e-005)  +/- [0.0001217, 0.0001208]
```

Figure 11-4. Radial distortion plot for a particular camera lens: the arrows show where points on an external rectangular grid are displaced in a radially distorted image (courtesy of Jean-Yves Bouguet)

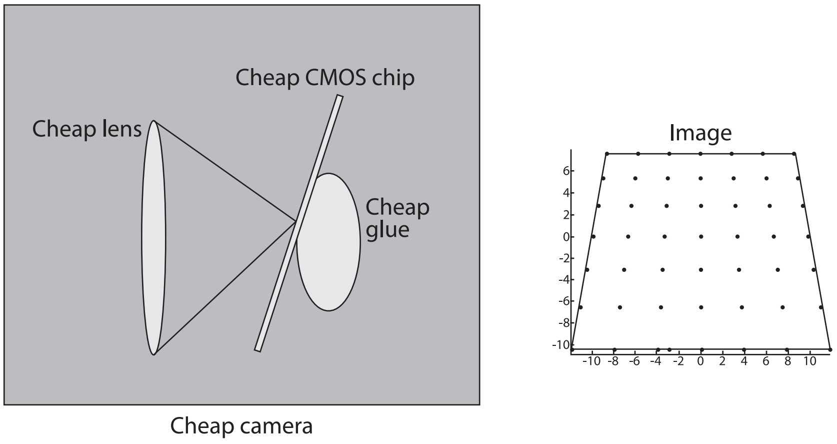
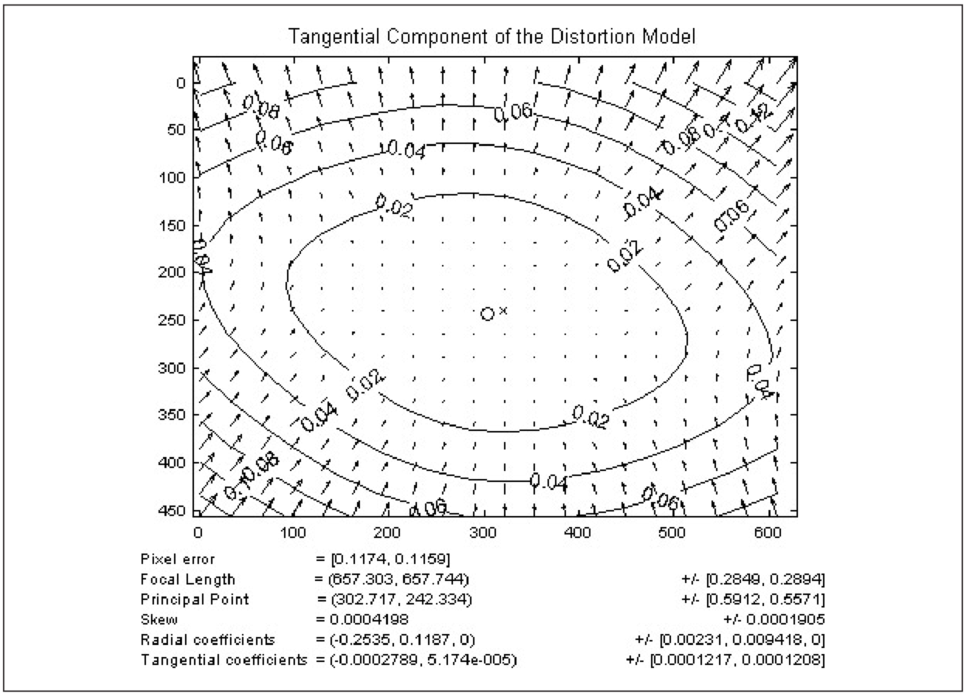


Figure 11-5. Tangential distortion results when the lens is not fully parallel to the image plane; in cheap cameras, this can happen when the imager is glued to the back of the camera (image courtesy of Sebastian Thrun)

There are many other kinds of distortions that occur in imaging systems, but they typically have lesser effects than radial and tangential distortions. Hence neither we nor OpenCV will deal with them further.



```
Pixel error              = [0.1174, 0.1159]
Focal Length             = (657.303, 657.744)              +/- [0.2849, 0.2894]
Principal Point          = (302.717, 242.334)              +/- [0.5912, 0.5571]
Skew                     = 0.0004198                       +/- 0.0001905
Radial coefficients      = (-0.2535, 0.1187, 0)            +/- [0.00231, 0.009418, 0]
Tangential coefficients  = (-0.0002789, 5.174e-005)        +/- [0.0001217, 0.0001208]
```

Figure 11-6. Tangential distortion plot for a particular camera lens: the arrows show where points on an external rectangular grid are displaced in a tangentially distorted image (courtesy of Jean-Yves Bouguet)

Calibration

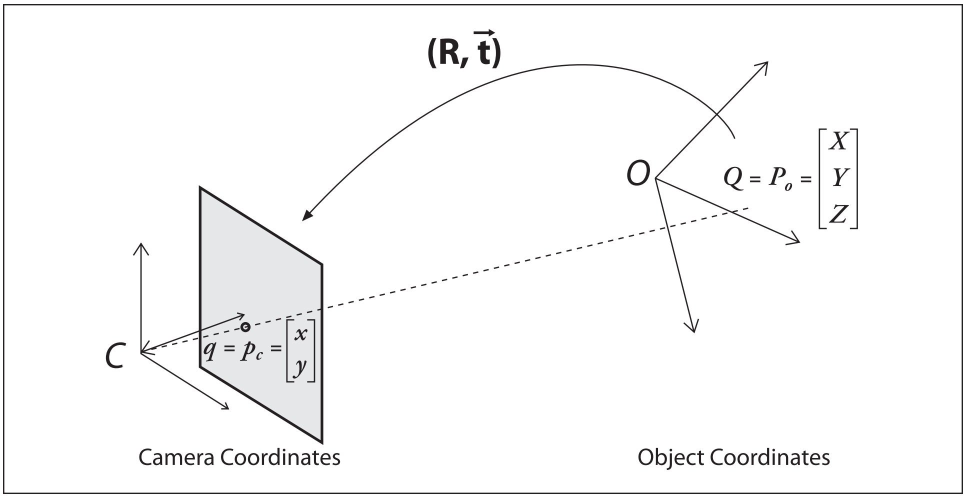
Now that we have some idea of how we'd describe the intrinsic and distortion properties of a camera mathematically, the next question that naturally arises is how we can use OpenCV to compute the intrinsics matrix and the distortion vector.*

OpenCV provides several algorithms to help us compute these intrinsic parameters. The actual calibration is done via `cvCalibrateCamera2()`. In this routine, the method of calibration is to target the camera on a known structure that has many individual and identifiable points. By viewing this structure from a variety of angles, it is possible to then compute the (relative) location and orientation of the camera at the time of each image as well as the intrinsic parameters of the camera (see Figure 11-9 in the "Chessboards" section). In order to provide multiple views, we rotate and translate the object, so let's pause to learn a little more about rotation and translation.

* For a great online tutorial of camera calibration, see Jean-Yves Bouguet's calibration website (*http://www.vision.caltech.edu/bouguetj/calib_doc*).

Rotation Matrix and Translation Vector

For each image the camera takes of a particular object, we can describe the *pose* of the object relative to the camera coordinate system in terms of a rotation and a translation; see Figure 11-7.



$(R, \vec{t})$
O
$Q = P_o = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$
C
$q = p_c = \begin{bmatrix} x \\ y \end{bmatrix}$
Camera Coordinates
Object Coordinates

Figure 11-7. Converting from object to camera coordinate systems: the point P on the object is seen as point p on the image plane; the point p is related to point P by applying a rotation matrix R and a translation vector **t** *to P*

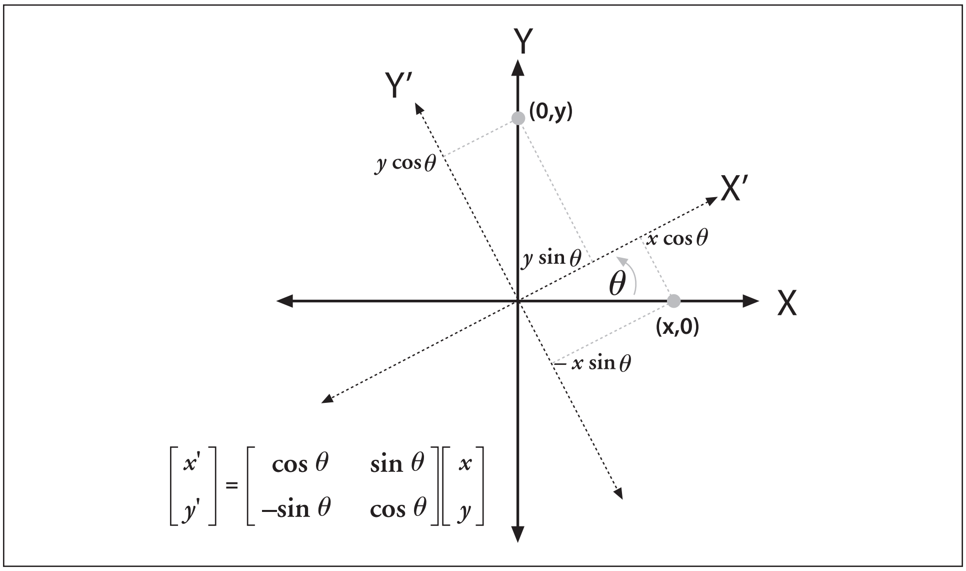
In general, a rotation in any number of dimensions can be described in terms of multiplication of a coordinate vector by a square matrix of the appropriate size. Ultimately, a rotation is equivalent to introducing a new description of a point's location in a different coordinate system. Rotating the coordinate system by an angle θ is equivalent to counterrotating our target point around the origin of that coordinate system by the same angle θ. The representation of a two-dimensional rotation as matrix multiplication is shown in Figure 11-8. Rotation in three dimensions can be decomposed into a two-dimensional rotation around each axis in which the pivot axis measurements remain constant. If we rotate around the x-, y-, and z-axes in sequence* with respective rotation angles ψ, φ, and θ, the result is a total rotation matrix R that is given by the product of the three matrices $R_x(\psi)$, $R_y(\varphi)$, and $R_z(\theta)$, where:

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & \sin\psi \\ 0 & -\sin\psi & \cos\psi \end{bmatrix}$$

* Just to be clear: the rotation we are describing here is first around the z-axis, then around the *new* position of the y-axis, and finally around the *new* position of the x-axis.

$$R_y(\varphi)=\begin{bmatrix}\cos\varphi & 0 & -\sin\varphi\\ 0 & 1 & 0\\ \sin\varphi & 0 & \cos\varphi\end{bmatrix}$$

$$R_z(\theta)=\begin{bmatrix}\cos\theta & \sin\theta & 0\\ -\sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{bmatrix}$$



Y
Y′
(0,y)
$y\cos\theta$
X′
$x\cos\theta$
$y\sin\theta$
θ
X
(x,0)
$-x\sin\theta$
$\begin{bmatrix}x'\\y'\end{bmatrix}=\begin{bmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$

Figure 11-8. Rotating points by θ (in this case, around the Z-axis) is the same as counterrotating the coordinate axis by θ; by simple trigonometry, we can see how rotation changes the coordinates of a point

Thus, $R = R_z(\theta), R_y(\varphi), R_x(\psi)$. The rotation matrix R has the property that its inverse is its transpose (we just rotate back); hence we have $R^TR = RR^T = I$, where I is the identity matrix consisting of 1s along the diagonal and 0s everywhere else.

The *translation vector* is how we represent a shift from one coordinate system to another system whose origin is displaced to another location; in other words, the translation vector is just the offset from the origin of the first coordinate system to the origin of the second coordinate system. Thus, to shift from a coordinate system centered on an object to one centered at the camera, the appropriate translation vector is simply $T = \text{origin}_{\text{object}} - \text{origin}_{\text{camera}}$. We then have (with reference to Figure 11-7) that a point in the object (or world) coordinate frame P_o has coordinates P_c in the camera coordinate frame:

$$P_c = R(P_o - T)$$

Combining this equation for P_c above with the camera intrinsic corrections will form the basic system of equations that we will be asking OpenCV to solve. The solution to these equations will be the camera calibration parameters we seek.

We have just seen that a three-dimensional rotation can be specified with three angles and that a three-dimensional translation can be specified with the three parameters (x, y, z); thus we have six parameters so far. The OpenCV intrinsics matrix for a camera has four parameters (f_x, f_y, c_x, and c_y), yielding a grand total of ten parameters that must be solved for each view (but note that the camera intrinsic parameters stay the same between views). Using a planar object, we'll soon see that each view fixes eight parameters. Because the six parameters of rotation and translation change between views, for each view we have constraints on two additional parameters that we use to resolve the camera intrinsic matrix. We'll then need at least two views to solve for all the geometric parameters.

We'll provide more details on the parameters and their constraints later in the chapter, but first we discuss the *calibration object*. The calibration object used in OpenCV is a flat grid of alternating black and white squares that is usually called a "chessboard" (even though it needn't have eight squares, or even an equal number of squares, in each direction).

Chessboards

In principle, any appropriately characterized object could be used as a calibration object, yet the practical choice is a regular pattern such as a chessboard.* Some calibration methods in the literature rely on three-dimensional objects (e.g., a box covered with markers), but flat chessboard patterns are much easier to deal with; it is difficult to make (and to store and distribute) precise 3D calibration objects. OpenCV thus opts for using multiple views of a planar object (a chessboard) rather than one view of a specially constructed 3D object. We use a pattern of alternating black and white squares (see Figure 11-9), which ensures that there is no bias toward one side or the other in measurement. Also, the resulting grid corners lend themselves naturally to the subpixel localization function discussed in Chapter 10.

Given an image of a chessboard (or a person holding a chessboard, or any other scene with a chessboard and a reasonably uncluttered background), you can use the OpenCV function `cvFindChessboardCorners()` to locate the corners of the chessboard.

```
int cvFindChessboardCorners(
  const void*    image,
  CvSize         pattern_size,
  CvPoint2D32f* corners,
  int*           corner_count = NULL,
  int            flags        = CV_CALIB_CB_ADAPTIVE_THRESH
);
```

* The specific use of this calibration object—and much of the calibration approach itself—comes from Zhang [Zhang99; Zhang00] and Sturm [Sturm99].

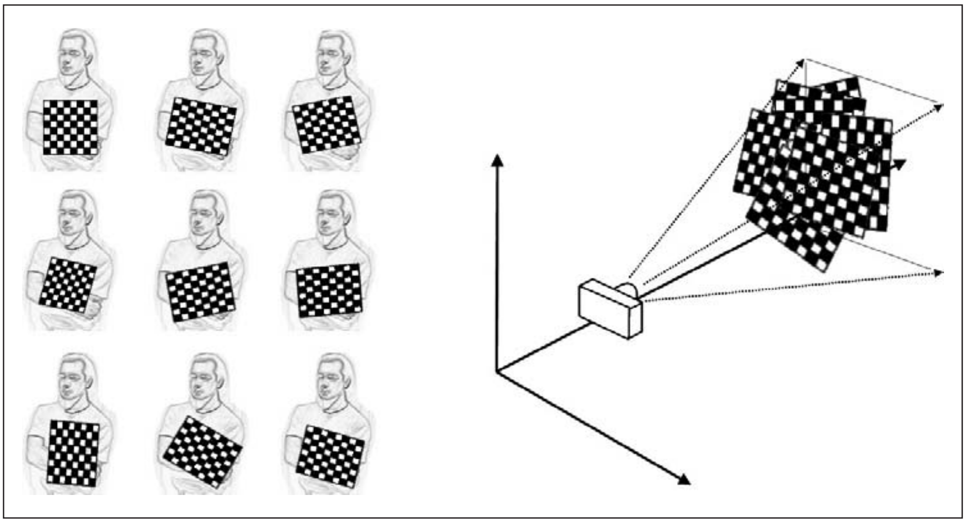


Figure 11-9. Images of a chessboard being held at various orientations (left) provide enough information to completely solve for the locations of those images in global coordinates (relative to the camera) and the camera intrinsics

This function takes as arguments a single image containing a chessboard. This image must be an 8-bit grayscale (single-channel) image. The second argument, `pattern_size`, indicates how many corners are in each row and column of the board. This count is the number of *interior* corners; thus, for a standard chess game board the correct value would be `cvSize(7,7)`.* The next argument, `corners`, is a pointer to an array where the corner locations can be recorded. This array must be preallocated and, of course, must be large enough for all of the corners on the board (49 on a standard chess game board). The individual values are the locations of the corners in pixel coordinates. The `corner_count` argument is optional; if non-`NULL`, it is a pointer to an integer where the number of corners found can be recorded. If the function is successful at finding all of the corners,† then the return value will be a nonzero number. If the function fails, 0 will be returned. The final `flags` argument can be used to implement one or more additional filtration steps to help find the corners on the chessboard. Any or all of the arguments may be combined using a Boolean OR.

`CV_CALIB_CB_ADAPTIVE_THRESH`
: The default behavior of `cvFindChessboardCorners()` is first to threshold the image based on average brightness, but if this flag is set then an adaptive threshold will be used instead.

* In practice, it is often more convenient to use a chessboard grid that is asymmetric and of even and odd dimensions—for example, (5, 6). Using such even-odd asymmetry yields a chessboard that has only one symmetry axis, so the board orientation can always be defined uniquely.

† Actually, the requirement is slightly stricter: not only must all the corners be found, they must also be ordered into rows and columns as expected. Only if the corners can be found and ordered correctly will the return value of the function be nonzero.

`CV_CALIB_CB_NORMALIZE_IMAGE`
: If set, this flag causes the image to be normalized via `cvEqualizeHist()` before the thresholding is applied.

`CV_CALIB_CB_FILTER_QUADS`
: Once the image is thresholded, the algorithm attempts to locate the quadrangles resulting from the perspective view of the black squares on the chessboard. This is an approximation because the lines of each edge of a quadrangle are assumed to be straight, which isn't quite true when there is radial distortion in the image. If this flag is set, then a variety of additional constraints are applied to those quadrangles in order to reject false quadrangles.

Subpixel corners

The corners returned by `cvFindChessboardCorners()` are only approximate. What this means in practice is that the locations are accurate only to within the limits of our imaging device, which means accurate to within one pixel. A separate function must be used to compute the exact locations of the corners (given the approximate locations and the image as input) to subpixel accuracy. This function is the same `cvFindCornerSubPix()` function that we used for tracking in Chapter 10. It should not be surprising that this function can be used in this context, since the chessboard interior corners are simply a special case of the more general Harris corners; the chessboard corners just happen to be particularly easy to find and track. Neglecting to call subpixel refinement after you first locate the corners can cause substantial errors in calibration.

Drawing chessboard corners

Particularly when debugging, it is often desirable to draw the found chessboard corners onto an image (usually the image that we used to compute the corners in the first place); this way, we can see whether the projected corners match up with the observed corners. Toward this end, OpenCV provides a convenient routine to handle this common task. The function `cvDrawChessboardCorners()` draws the corners found by `cvFindChessboard-Corners()` onto an image that you provide. If not all of the corners were found, the available corners will be represented as small red circles. If the entire pattern was found, then the corners will be painted into different colors (each row will have its own color) and connected by lines representing the identified corner order.

```
void cvDrawChessboardCorners(
    CvArr*         image,
    CvSize         pattern_size,
    CvPoint2D32f*  corners,
    int            count,
    int            pattern_was_found
);
```

The first argument to `cvDrawChessboardCorners()` is the image to which the drawing will be done. Because the corners will be represented as colored circles, this must be an 8-bit color image; in most cases, this will be a copy of the image you gave to `cvFindChessboardCorners()` (but you must convert it to a three-channel image yourself).

The next two arguments, pattern_size and corners, are the same as the corresponding arguments for cvFindChessboardCorners(). The argument count is an integer equal to the number of corners. Finally the argument pattern_was_found indicates whether the entire chessboard pattern was successfully found; this can be set to the return value from cvFindChessboardCorners(). Figure 11-10 shows the result of applying cvDrawChessboardCorners() to a chessboard image.

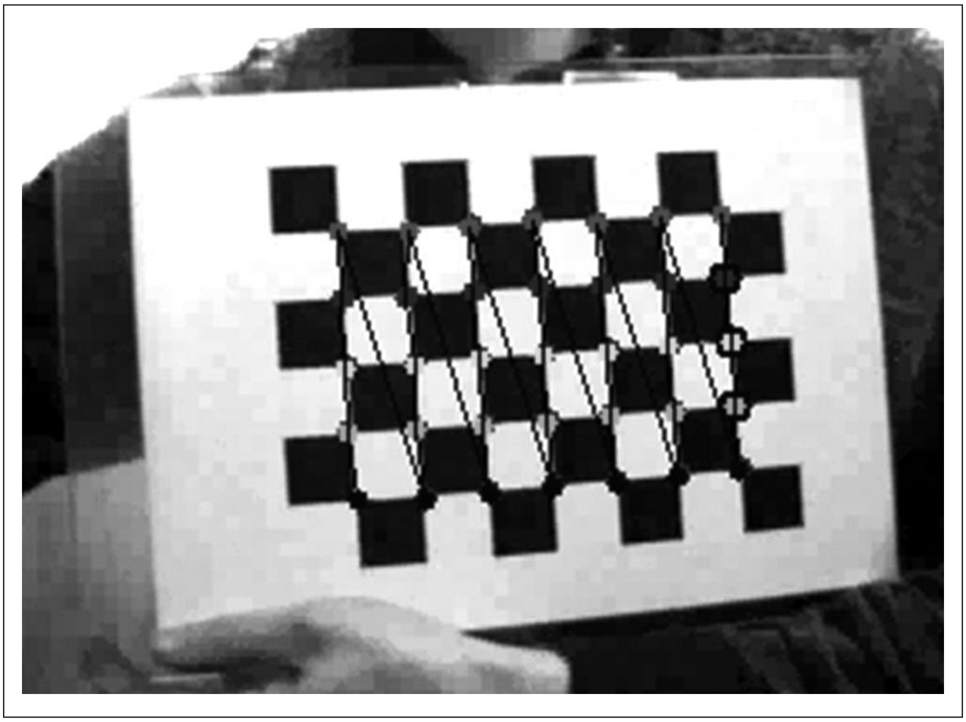


Figure 11-10. Result of cvDrawChessboardCorners(); once you find the corners using cvFindChessboardCorners(), you can project where these corners were found (small circles on corners) and in what order they belong (as indicated by the lines between circles)

We now turn to what a planar object can do for us. Points on a plane undergo perspective transform when viewed through a pinhole or lens. The parameters for this transform are contained in a 3-by-3 *homography* matrix, which we describe next.

Homography

In computer vision, we define *planar homography* as a projective mapping from one plane to another.* Thus, the mapping of points on a two-dimensional planar surface to

* The term "homography" has different meanings in different sciences; for example, it has a somewhat more general meaning in mathematics. The homographies of greatest interest in computer vision are a subset of the other, more general, meanings of the term.

the imager of our camera is an example of planar homography. It is possible to express this mapping in terms of matrix multiplication if we use homogeneous coordinates to express both the viewed point *Q* and the point *q* on the imager to which *Q* is mapped. If we define:

$$\tilde{Q} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix}^T$$
$$\tilde{q} = \begin{bmatrix} x & y & 1 \end{bmatrix}^T$$

then we can express the action of the homography simply as:

$$\tilde{q} = sH\tilde{Q}$$

Here we have introduced the parameter *s*, which is an arbitrary scale factor (intended to make explicit that the homography is defined only up to that factor). It is conventionally factored out of *H*, and we'll stick with that convention here.

With a little geometry and some matrix algebra, we can solve for this transformation matrix. The most important observation is that *H* has two parts: the physical transformation, which essentially locates the object plane we are viewing; and the projection, which introduces the camera intrinsics matrix. See Figure 11-11.

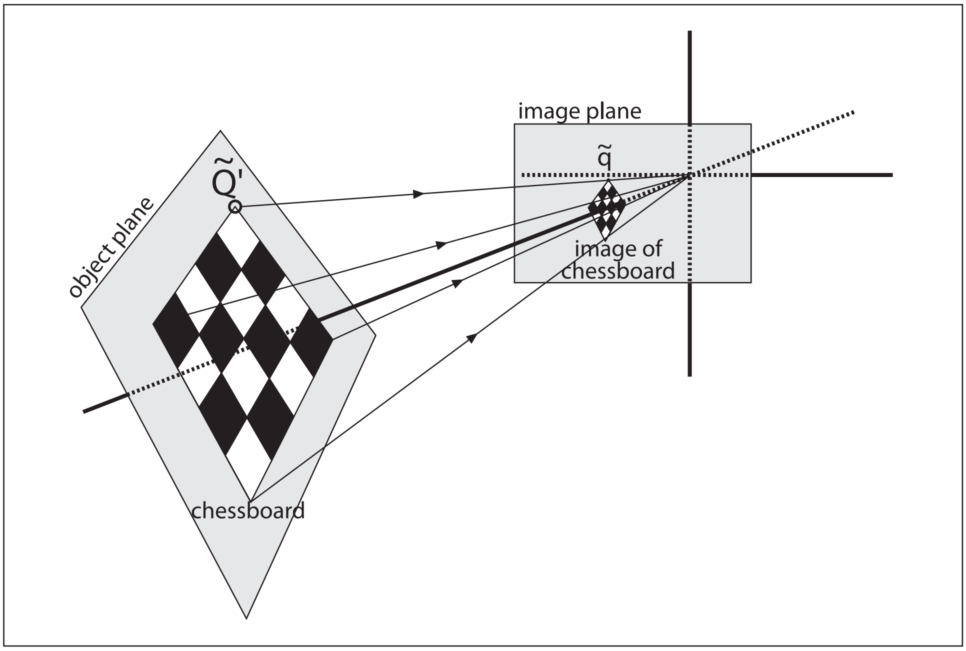


Figure 11-11. View of a planar object as described by homography: a mapping—from the object plane to the image plane—that simultaneously comprehends the relative locations of those two planes as well as the camera projection matrix

The physical transformation part is the sum of the effects of some rotation R and some translation $\mathbf{t}$ that relate the plane we are viewing to the image plane. Because we are working in homogeneous coordinates, we can combine these within a single matrix as follows:*

$$W = \begin{bmatrix} R & \mathbf{t} \end{bmatrix}$$

Then, the action of the camera matrix M (which we already know how to express in projective coordinates) is multiplied by $W\tilde{Q}$; this yields:

$$\tilde{q} = sMW\tilde{Q}, \quad \text{where} \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

It would seem that we are done. However, it turns out that in practice our interest is not the coordinate $\tilde{Q}$, which is defined for all of space, but rather a coordinate $\tilde{Q}'$, which is defined only on the plane we are looking at. This allows for a slight simplification.

Without loss of generality, we can choose to define the object plane so that $Z = 0$. We do this because, if we also break up the rotation matrix into three 3-by-1 columns (i.e., $R = [r_1\ r_2\ r_3]$), then one of those columns is not needed. In particular:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = sM\begin{bmatrix} r_1 & r_2 & r_3 & \mathbf{t} \end{bmatrix}\begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = sM\begin{bmatrix} r_1 & r_2 & \mathbf{t} \end{bmatrix}\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

The homography matrix H that maps a planar object's points onto the imager is then described completely by $H = sM[r_1\ r_2\ \mathbf{t}]$, where:

$$\tilde{q} = sH\tilde{Q}'$$

Observe that H is now a 3-by-3 matrix.

OpenCV uses the preceding equations to compute the homography matrix. It uses multiple images of the same object to compute both the individual translations and rotations for each view as well as the intrinsics (which are the same for all views). As we have discussed, rotation is described by three angles and translation is defined by three offsets; hence there are six unknowns for each view. This is OK, because a known planar object (such as our chessboard) gives us eight equations—that is, the mapping of a square into a quadrilateral can be described by four (x, y) points. Each new frame gives us eight equations at the cost of six new extrinsic unknowns, so given enough images we should be able to compute any number of intrinsic unknowns (more on this shortly).

* Here $W = [R\ \mathbf{t}]$ is a 3-by-4 matrix whose first three columns comprise the nine entries of R and whose last column consists of the three-component vector $\mathbf{t}$.

The homography matrix H relates the positions of the points on a source image plane to the points on the destination image plane (usually the imager plane) by the following simple equations:

$$p_{\text{dst}} = Hp_{\text{src}}, \quad p_{\text{src}} = H^{-1} p_{\text{dst}}$$

$$p_{\text{dst}} = \begin{bmatrix} x_{\text{dst}} \\ y_{\text{dst}} \\ 1 \end{bmatrix}, \quad p_{\text{src}} = \begin{bmatrix} x_{\text{src}} \\ y_{\text{src}} \\ 1 \end{bmatrix}$$

Notice that we can compute H without knowing anything about the camera intrinsics. In fact, computing multiple homographies from multiple views is the method OpenCV uses to solve for the camera intrinsics, as we'll see.

OpenCV provides us with a handy function, `cvFindHomography()`, which takes a list of correspondences and returns the homography matrix that best describes those correspondences. We need a minimum of four points to solve for H, but we can supply many more if we have them* (as we will with any chessboard bigger than 3-by-3). Using more points is beneficial, because invariably there will be noise and other inconsistencies whose effect we would like to minimize.

```
void cvFindHomography(
    const CvMat* src_points,
    const CvMat* dst_points,
    CvMat*       homography
);
```

The input arrays `src_points` and `dst_points` can be either N-by-2 matrices or N-by-3 matrices. In the former case the points are pixel coordinates, and in the latter they are expected to be homogeneous coordinates. The final argument, `homography`, is just a 3-by-3 matrix to be filled by the function in such a way that the back-projection error is minimized. Because there are only eight free parameters in the homography matrix, we chose a normalization where $H_{33} = 1$. Scaling the homography could be applied to the ninth homography parameter, but usually scaling is instead done by multiplying the entire homography matrix by a scale factor.

Camera Calibration

We finally arrive at camera calibration for camera intrinsics and distortion parameters. In this section we'll learn how to compute these values using `cvCalibrateCamera2()` and also how to use these models to correct distortions in the images that the calibrated camera would have otherwise produced. First we say a little more about how many views of a chessboard are necessary in order to solve for the intrinsics and distortion. Then we'll offer a high-level overview of how OpenCV actually solves this system before moving on to the code that makes it all easy to do.

* Of course, an exact solution is guaranteed only when there are four correspondences. If more are provided, then what's computed is a solution that is optimal in the sense of least-squares error.

How many chess corners for how many parameters?

It will prove instructive to review our unknowns. That is, how many parameters are we attempting to solve for through calibration? In the OpenCV case, we have four *intrinsic* parameters (f_x, f_y, c_x, c_y,) and five *distortion* parameters: three radial (k_1, k_2, k_3) and two tangential (p_1, p_2). Intrinsic parameters are directly tied to the 3D geometry (and hence the extrinsic parameters) of where the chessboard is in space; distortion parameters are tied to the 2D geometry of how the pattern of points gets distorted, so we deal with the constraints on these two classes of parameters separately. Three corner points in a known pattern yielding six pieces of information are (in principle) all that is needed to solve for our five distortion parameters (of course, we use much more for robustness). Thus, one view of a chessboard is all that we need to compute our distortion parameters. The same chessboard view could also be used in our intrinsics computation, which we consider next, starting with the extrinsic parameters. For the *extrinsic* parameters we'll need to know where the chessboard is. This will require three rotation parameters (ψ, ϕ, θ) and three translation parameters (T_x, T_y, T_z) for a total of six per view of the chessboard, because in each image the chessboard will move. Together, the four intrinsic and six extrinsic parameters make for ten altogether that we must solve for each view.

Let's say we have N corners and K images of the chessboard (in different positions). How many views and corners must we see so that there will be enough constraints to solve for all these parameters?

- K images of the chessboard provide $2NK$ constraints (we use the multiplier 2 because each point on the image has both an x and a y coordinate).
- Ignoring the distortion parameters for the moment, we have 4 intrinsic parameters and $6K$ extrinsic parameters (since we need to find the 6 parameters of the chessboard location in each of the K views).
- Solving then requires that $2NK \geq 6K + 4$ hold (or, equivalently, $(N - 3)\ K \geq 2$).

It seems that if $N = 5$ then we need only $K = 1$ image, but watch out! For us, K (the number of images) must be more than 1. The reason for requiring $K > 1$ is that we're using chessboards for calibration to fit a homography matrix for each of the K views. As discussed previously, a homography can yield at most eight parameters from four (x, y) pairs. This is because only four points are needed to express everything that a planar perspective view can do: it can stretch a square in four different directions at once, turning it into any quadrilateral (see the perspective images in Chapter 6). So, no matter how many corners we detect on a plane, we only get four corners' worth of information. Per chessboard view, then, the equation can give us only four corners of information or $(4 - 3)\ K > 1$, which means $K > 1$. This implies that two views of a 3-by-3 chessboard (counting only internal corners) are the minimum that could solve our calibration problem. Consideration for noise and numerical stability is typically what requires the collection of more images of a larger chessboard. In practice, for high-quality results, you'll need at least ten images of a 7-by-8 or larger chessboard (and that's only if you move the chessboard enough between images to obtain a "rich" set of views).

What's under the hood?

This subsection is for those who want to go deeper; it can be safely skipped if you just want to call the calibration functions. If you are still with us, the question remains: how is all this mathematics used for calibration? Although there are many ways to solve for the camera parameters, OpenCV chose one that works well on planar objects. The algorithm OpenCV uses to solve for the focal lengths and offsets is based on Zhang's method [Zhang00], but OpenCV uses a different method based on Brown [Brown71] to solve for the distortion parameters.

To get started, we pretend that there is no distortion in the camera while solving for the other calibration parameters. For each view of the chessboard, we collect a homography H as described previously. We'll write H out as column vectors, $H = [h_1\ h_2\ h_3]$, where each h is a 3-by-1 vector. Then, in view of the preceding homography discussion, we can set H equal to the camera intrinsics matrix M multiplied by a combination of the first two rotation matrix columns, r_1 and r_2, and the translation vector $\mathbf{t}$; after including the scale factor s, this yields:

$$H = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} = sM \begin{bmatrix} r_1 & r_2 & \mathbf{t} \end{bmatrix}$$

Reading off these equations, we have:

$$h_1 = sMr_1 \quad \text{or} \quad r_1 = \lambda M^{-1} h_1$$

$$h_2 = sMr_2 \quad \text{or} \quad r_2 = \lambda M^{-1} h_2$$

$$h_3 = sMt \quad \text{or} \quad t = \lambda M^{-1} h_3$$

Here, $\lambda = 1/s$.

The rotation vectors are orthogonal to each other by construction, and since the scale is extracted it follows that r_1 and r_2 are orthonormal. Orthonormal implies two things: the rotation vector's dot product is 0, and the vectors' magnitudes are equal. Starting with the dot product, we have:

$$r_1^{\mathrm{T}} r_2 = 0$$

For any vectors a and b we have $(ab)^{\mathrm{T}} = b^{\mathrm{T}} a^{\mathrm{T}}$, so we can substitute for r_1 and r_2 to derive our first constraint:

$$h_1^{\mathrm{T}} M^{-\mathrm{T}} M^{-1} h_2 = 0$$

where $A^{-\mathrm{T}}$ is shorthand for $(A^{-1})^{\mathrm{T}}$. We also know that the magnitudes of the rotation vectors are equal:

$$\|r_1\| = \|r_2\| \quad \text{or} \quad r_1^{\mathrm{T}} r_1 = r_2^{\mathrm{T}} r_2$$

Substituting for r_1 and r_2 yields our second constraint:

$$h_1^{\mathrm{T}} M^{-\mathrm{T}} M^{-1} h_1 = h_2^{\mathrm{T}} M^{-\mathrm{T}} M^{-1} h_2$$

To make things easier, we set $B = M^{-T}M^{-1}$. Writing this out, we have:

$$B = M^{-T}M^{-1} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{bmatrix}$$

It so happens that this matrix B has a general closed-form solution:

$$B = \begin{bmatrix} \frac{1}{f_x^2} & 0 & \frac{-c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & \frac{-c_y}{f_y^2} \\ \frac{-c_x}{f_x^2} & \frac{-c_y}{f_y^2} & \frac{c_x^2}{f_x^2} + \frac{c_y^2}{f_y^2} + 1 \end{bmatrix}$$

Using the B-matrix, both constraints have the general form $h_i^T B h_j$ in them. Let's multiply this out to see what the components are. Because B is symmetric, it can be written as one six-dimensional vector dot product. Arranging the necessary elements of B into the new vector b, we have:

$$h_i^T B h_j = v_{ij}^T b = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{i3} \end{bmatrix}^T \begin{bmatrix} B_{11} \\ B_{12} \\ B_{22} \\ B_{13} \\ B_{23} \\ B_{33} \end{bmatrix}^T$$

Using this definition for v_{ij}^T, our two constraints may now be written as:

$$\begin{bmatrix} v_{12}^T \\ (v_{11} - v_{22})^T \end{bmatrix} b = 0$$

If we collect K images of chessboards together, then we can stack K of these equations together:

$$Vb = 0$$

where V is a $2K$-by-6 matrix. As before, if $K \geq 2$ then this equation can be solved for our $b = [B_{11}, B_{12}, B_{22}, B_{13}, B_{23}, B_{33}]^T$. The camera intrinsics are then pulled directly out of our closed-form solution for the B-matrix:

$$\begin{aligned}
f_x &= \sqrt{\lambda / B_{11}} \\
f_y &= \sqrt{\lambda B_{11} / (B_{11}B_{22} - B_{12}^2)} \\
c_x &= -B_{13} f_x^2 / \lambda \\
c_y &= (B_{12}B_{13} - B_{11}B_{23}) / (B_{11}B_{22} - B_{12}^2)
\end{aligned}$$

where:

$$\lambda = B_{33} - (B_{13}^2 + c_y(B_{12}B_{13} - B_{11}B_{23})) / B_{11}$$

The extrinsics (rotation and translation) are then computed from the equations we read off of the homography condition:

$$\begin{aligned}
r_1 &= \lambda M^{-1} h_1 \\
r_2 &= \lambda M^{-1} h_2 \\
r_3 &= r_1 \times r_2 \\
t &= \lambda M^{-1} h_3
\end{aligned}$$

Here the scaling parameter is determined from the orthonormality condition $\lambda = 1/\|M^{-1}h_1\|$.

Some care is required because, when we solve using real data and put the *r*-vectors together ($R = [r_1\ r_2\ r_3]$), we will not end up with an exact rotation matrix for which $R^{\mathrm{T}}R = RR^{\mathrm{T}} = I$ holds.

To get around this problem, the usual trick is to take the singular value decomposition (SVD) of *R*. As discussed in Chapter 3, SVD is a method of factoring a matrix into two orthonormal matrices, *U* and *V*, and a middle matrix *D* of scale values on its diagonal. This allows us to turn *R* into $R = UDV^{\mathrm{T}}$. Because *R* is itself orthonormal, the matrix *D* must be the identity matrix *I* such that $R = UIV^{\mathrm{T}}$. We can thus "coerce" our computed *R* into being a rotation matrix by taking *R*'s singular value decomposition, setting its *D* matrix to the identity matrix, and multiplying by the SVD again to yield our new, conforming rotation matrix *R′*.

Despite all this work, we have not yet dealt with lens distortions. We use the camera intrinsics found previously—together with the distortion parameters set to 0—for our initial guess to start solving a larger system of equations.

The points we "perceive" on the image are really in the wrong place owing to distortion. Let (x_p, y_p) be the point's location if the pinhole camera were perfect and let (x_d, y_d) be its distorted location; then:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \begin{bmatrix} f_x X^W / Z^W + c_x \\ f_y X^W / Z^W + c_y \end{bmatrix}$$

We use the results of the calibration without distortion via the following substitution:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \begin{bmatrix} 2p_1 x_d y_d + p_2 (r^2 + 2x_d^2) \\ p_1 (r^2 + 2y_d^2) + 2p_2 x_d y_d \end{bmatrix}$$

A large list of these equations are collected and solved to find the distortion parameters, after which the intrinsics and extrinsics are reestimated. That's the heavy lifting that the single function cvCalibrateCamera2()[*] does for you!

Calibration function

Once we have the corners for several images, we can call cvCalibrateCamera2(). This routine will do the number crunching and give us the information we want. In particular, the results we receive are the *camera intrinsics matrix*, the *distortion coefficients*, the *rotation vectors*, and the *translation vectors*. The first two of these constitute the intrinsic parameters of the camera, and the latter two are the extrinsic measurements that tell us where the objects (i.e., the chessboards) were found and what their orientations were. The distortion coefficients (k_1, k_2, p_1, p_2, and k_3)[†] are the coefficients from the radial and tangential distortion equations we encountered earlier; they help us when we want to correct that distortion away. The camera intrinsic matrix is perhaps the most interesting final result, because it is what allows us to transform from 3D coordinates to the image's 2D coordinates. We can also use the camera matrix to do the reverse operation, but in this case we can only compute a line in the three-dimensional world to which a given image point must correspond. We will return to this shortly.

Let's now examine the camera calibration routine itself.

```
void cvCalibrateCamera2(
  CvMat*    object_points,
  CvMat*    image_points,
  int*      point_counts,
  CvSize    image_size,
  CvMat*    intrinsic_matrix,
  CvMat*    distortion_coeffs,
  CvMat*    rotation_vectors    = NULL,
  CvMat*    translation_vectors = NULL,
  int       flags               = 0
);
```

When calling cvCalibrateCamera2(), there are many arguments to keep straight. Yet we've covered (almost) all of them already, so hopefully they'll make sense.

* The cvCalibrateCamera2() function is used internally in the stereo calibration functions we will see in Chapter 12. For stereo calibration, we'll be calibrating two cameras at the same time and will be looking to relate them together through a rotation matrix and a translation vector.

† The third radial distortion component k_3 comes last because it was a late addition to OpenCV to allow better correction to highly distorted fish eye type lenses and should only be used in such cases. We will see momentarily that k_3 can be set to 0 by first initializing it to 0 and then setting the flag to CV_CALIB_FIX_K3.

The first argument is the object_points, which is an N-by-3 matrix containing the physical coordinates of each of the K points on each of the M images of the object (i.e., $N = K \times M$). These points are located in the coordinate frame attached to the object.* This argument is a little more subtle than it appears in that your manner of describing the points on the object will implicitly define your physical units and the structure of your coordinate system hereafter. In the case of a chessboard, for example, you might define the coordinates such that all of the points on the chessboard had a z-value of 0 while the x- and y-coordinates are measured in centimeters. Had you chosen inches, all computed parameters would then (implicitly) be in inches. Similarly if you had chosen all the x-coordinates (rather than the z-coordinates) to be 0, then the implied location of the chessboards relative to the camera would be largely in the x-direction rather than the z-direction. The squares define one unit, so that if, for example, your squares are 90 mm on each side, your camera world, object and camera coordinate units would be in mm/90. In principle you can use an object other than a chessboard, so it is not really necessary that all of the object points lie on a plane, but this is usually the easiest way to calibrate a camera.† In the simplest case, we simply define each square of the chessboard to be of dimension one "unit" so that the coordinates of the corners on the chessboard are just integer corner rows and columns. Defining S_{width} as the number of squares across the width of the chessboard and S_{height} as the number of squares over the height:

$$(0,0),(0,1),(0,2),\ldots,(1,0),(2,0),\ldots,(1,1),\ldots,(S_{\text{width}}-1,S_{\text{height}}-1)$$

The second argument is the image_points, which is an N-by-2 matrix containing the pixel coordinates of all the points supplied in object_points. If you are performing a calibration using a chessboard, then this argument consists simply of the return values for the M calls to cvFindChessboardCorners() but now rearranged into a slightly different format.

The argument point_counts indicates the number of points in each image; this is supplied as an M-by-1 matrix. The image_size is just the size, in pixels, of the images from which the image points were extracted (e.g., those images of yourself waving a chessboard around).

The next two arguments, intrinsic_matrix and distortion_coeffs, constitute the intrinsic parameters of the camera. These arguments can be both outputs (filling them in is the main reason for calibration) and inputs. When used as inputs, the values in these matrices when the function is called will affect the computed result. Which of these matrices will be used as input will depend on the flags parameter; see the following discussion. As we discussed earlier, the intrinsic matrix completely specifies the behavior

* Of course, it's normally the same object in every image, so the N points described are actually M repeated listings of the locations of the K points on a single object.

† At the time of this writing, automatic initialization of the intrinsic matrix before the optimization algorithm runs has been implemented only for planar calibration objects. This means that if you have a nonplanar object then you must provide a starting guess for the principal point and focal lengths (see CV_CALIB_USE_INTRINSIC_GUESS to follow).

of the camera in our ideal camera model, while the distortion coefficients characterize much of the camera's nonideal behavior. The camera matrix is always 3-by-3 and the distortion coefficients always number five, so the `distortion_coeffs` argument should be a pointer to a 5-by-1 matrix (they will be recorded in the order k_1, k_2, p_1, p_2, k_3).

Whereas the previous two arguments summarized the camera's intrinsic information, the next two summarize the extrinsic information. That is, they tell us where the calibration objects (e.g., the chessboards) were located relative to the camera in each picture. The locations of the objects are specified by a rotation and a translation.* The rotations, `rotation_vectors`, are defined by *M* three-component vectors arranged into an *M*-by-3 matrix (where *M* is the number of images). Be careful, these are not in the form of the 3-by-3 rotation matrix we discussed previously; rather, each vector represents an axis in three-dimensional space in the camera coordinate system around which the chessboard was rotated and where the length or magnitude of the vector encodes the counterclockwise angle of the rotation. Each of these rotation vectors can be converted to a 3-by-3 rotation matrix by calling `cvRodrigues2()`, which is described in its own section to follow. The translations, `translation_vectors`, are similarly arranged into a second *M*-by-3 matrix, again in the camera coordinate system. As stated before, the units of the camera coordinate system are exactly those assumed for the chessboard. That is, if a chessboard square is 1 inch by 1 inch, the units are inches.

Finding parameters through optimization can be somewhat of an art. Sometimes trying to solve for all parameters at once can produce inaccurate or divergent results if your initial starting position in parameter space is far from the actual solution. Thus, it is often better to "sneak up" on the solution by getting close to a good parameter starting position in stages. For this reason, we often hold some parameters fixed, solve for other parameters, then hold the other parameters fixed and solve for the original and so on. Finally, when we think all of our parameters are close to the actual solution, we use our close parameter setting as the starting point and solve for everything at once. OpenCV allows you this control through the `flags` setting. The `flags` argument allows for some finer control of exactly how the calibration will be performed. The following values may be combined together with a Boolean OR operation as needed.

`CV_CALIB_USE_INTRINSIC_GUESS`

Normally the intrinsic matrix is computed by `cvCalibrateCamera2()` with no additional information. In particular, the initial values of the parameters c_x and c_y (the image center) are taken directly from the `image_size` argument. If this argument is set, then `intrinsic_matrix` is assumed to contain valid values that will be used as an initial guess to be further optimized by `cvCalibrateCamera2()`.

* You can envision the chessboard's location as being expressed by (1) "creating" a chessboard at the origin of your camera coordinates, (2) rotating that chessboard by some amount around some axis, and (3) moving that oriented chessboard to a particular place. For those who have experience with systems like OpenGL, this should be a familiar construction.

CV_CALIB_FIX_PRINCIPAL_POINT

This flag can be used with or without CV_CALIB_USE_INTRINSIC_GUESS. If used without, then the principle point is fixed at the center of the image; if used with, then the principle point is fixed at the supplied initial value in the intrinsic_matrix.

CV_CALIB_FIX_ASPECT_RATIO

If this flag is set, then the optimization procedure will only vary f_x and f_y together and will keep their ratio fixed to whatever value is set in the intrinsic_matrix when the calibration routine is called. (If the CV_CALIB_USE_INTRINSIC_GUESS flag is not also set, then the values of f_x and f_y in intrinsic_matrix can be any arbitrary values and only their ratio will be considered relevant.)

CV_CALIB_FIX_FOCAL_LENGTH

This flag causes the optimization routine to just use the f_x and f_y that were passed in in the intrinsic_matrix.

CV_CALIB_FIX_K1, CV_CALIB_FIX_K2 and CV_CALIB_FIX_K3

Fix the radial distortion parameters k_1, k_2, and k_3. The radial parameters may be set in any combination by adding these flags together. In general, the last parameter should be fixed to 0 unless you are using a fish-eye lens.

CV_CALIB_ZERO_TANGENT_DIST:

This flag is important for calibrating high-end cameras which, as a result of precision manufacturing, have very little tangential distortion. Trying to fit parameters that are near 0 can lead to noisy spurious values and to problems of numerical stability. Setting this flag turns off fitting the tangential distortion parameters p_1 and p_2, which are thereby both set to 0.

Computing extrinsics only

In some cases you will already have the intrinsic parameters of the camera and therefore need only to compute the location of the object(s) being viewed. This scenario clearly differs from the usual camera calibration, but it is nonetheless a useful task to be able to perform.

```
void cvFindExtrinsicCameraParams2(
    const CvMat* object_points,
    const CvMat* image_points,
    const CvMat* intrinsic_matrix,
    const CvMat* distortion_coeffs,
    CvMat*       rotation_vector,
    CvMat*       translation_vector
);
```

The arguments to cvFindExtrinsicCameraParams2() are identical to the corresponding arguments for cvCalibrateCamera2() with the exception that the intrinsic matrix and the distortion coefficients are being supplied rather than computed. The rotation output is in the form of a 1-by-3 or 3-by-1 rotation_vector that represents the 3D axis around which the chessboard or points were rotated, and the vector magnitude or length represents the counterclockwise angle of rotation. This rotation vector can be converted into the 3-by-3 rotation matrix we've discussed before via the `cvRodrigues2()` function. The translation vector is the offset in camera coordinates to where the chessboard origin is located.

Undistortion

As we have alluded to already, there are two things that one often wants to do with a calibrated camera. The first is to correct for distortion effects, and the second is to construct three-dimensional representations of the images it receives. Let's take a moment to look at the first of these before diving into the more complicated second task in Chapter 12.

OpenCV provides us with a ready-to-use undistortion algorithm that takes a raw image and the distortion coefficients from `cvCalibrateCamera2()` and produces a corrected image (see Figure 11-12). We can access this algorithm either through the function `cvUndistort2()`, which does everything we need in one shot, or through the pair of routines `cvInitUndistortMap()` and `cvRemap()`, which allow us to handle things a little more efficiently for video or other situations where we have many images from the same camera.*

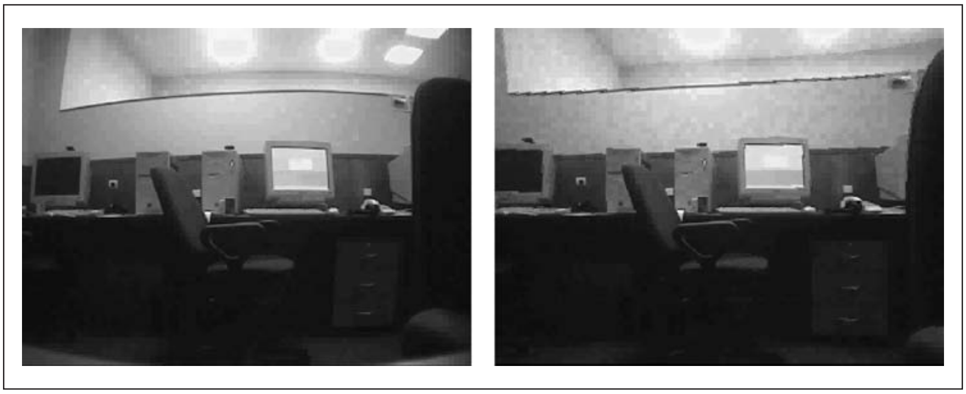


Figure 11-12. Camera image before undistortion (left) and after undistortion (right)

The basic method is to compute a *distortion map*, which is then used to correct the image. The function `cvInitUndistortMap()` computes the distortion map, and `cvRemap()` can be used to apply this map to an arbitrary image.† The function `cvUndistort2()` does one after the other in a single call. However, computing the distortion map is a time-consuming operation, so it's not very smart to keep calling `cvUndistort2()` if the distortion map is not changing. Finally, if we just have a list of 2D points, we can call the function `cvUndistortPoints()` to transform them from their original coordinates to their undistorted coordinates.

* We should take a moment to clearly make a distinction here between *undistortion*, which mathematically removes lens distortion, and *rectification*, which mathematically aligns the images with respect to each other.

† We first encountered `cvRemap()` in the context of image transformations (Chapter 6).

```
// Undistort images
void cvInitUndistortMap(
  const CvMat*   intrinsic_matrix,
  const CvMat*   distortion_coeffs,
  cvArr*         mapx,
  cvArr*         mapy
);
void cvUndistort2(
  const CvArr*   src,
  CvArr*         dst,
  const cvMat*   intrinsic_matrix,
  const cvMat*   distortion_coeffs
);
// Undistort a list of 2D points only
void cvUndistortPoints(
  const CvMat* _src,
  CvMat*       dst,
  const CvMat*  intrinsic_matrix,
  const CvMat*  distortion_coeffs,
  const CvMat*  R  = 0,
  const CvMat*  Mr = 0;
);
```

The function `cvInitUndistortMap()` computes the distortion map, which relates each point in the image to the location where that point is mapped. The first two arguments are the camera intrinsic matrix and the distortion coefficients, both in the expected form as received from `cvCalibrateCamera2()`. The resulting distortion map is represented by two separate 32-bit, single-channel arrays: the first gives the *x*-value to which a given point is to be mapped and the second gives the *y*-value. You might be wondering why we don't just use a single two-channel array instead. The reason is so that the results from `cvUnitUndistortMap()` can be passed directly to `cvRemap()`.

The function `cvUndistort2()` does all this in a single pass. It takes your initial (distorted image) as well as the camera's intrinsic matrix and distortion coefficients, and then outputs an undistorted image of the same size. As mentioned previously, `cvUndistortPoints()` is used if you just have a list of 2D point coordinates from the original image and you want to compute their associated undistorted point coordinates. It has two extra parameters that relate to its use in stereo rectification, discussed in Chapter 12. These parameters are `R`, the rotation matrix between the two cameras, and `Mr`, the camera intrinsic matrix of the rectified camera (only really used when you have two cameras as per Chapter 12). The rectified camera matrix `Mr` can have dimensions of 3-by-3 or 3-by-4 deriving from the first three or four columns of `cvStereoRectify()`'s return value for camera matrices `P1` or `P2` (for the left or right camera; see Chapter 12). These parameters are by default NULL, which the function interprets as identity matrices.

Putting Calibration All Together

OK, now it's time to put all of this together in an example. We'll present a program that performs the following tasks: it looks for chessboards of the dimensions that the user specified, grabs as many full images (i.e., those in which it can find all the chessboard corners) as the user requested, and computes the camera intrinsics and distortion parameters. Finally, the program enters a display mode whereby an undistorted version of the camera image can be viewed; see Example 11-1. When using this algorithm, you'll want to substantially change the chessboard views between successful captures. Otherwise, the matrices of points used to solve for calibration parameters may form an ill-conditioned (rank deficient) matrix and you will end up with either a bad solution or no solution at all.

Example 11-1. Reading a chessboard's width and height, reading and collecting the requested number of views, and calibrating the camera

```
// calib.cpp
// Calling convention:
// calib board_w board_h number_of_views
//
// Hit 'p' to pause/unpause, ESC to quit
//
#include <cv.h>
#include <highgui.h>
#include <stdio.h>
#include <stdlib.h>

int n_boards = 0; //Will be set by input list
const int board_dt = 20; //Wait 20 frames per chessboard view
int board_w;
int board_h;

int main(int argc, char* argv[]) {

  if(argc != 4){
    printf("ERROR: Wrong number of input parameters\n");
    return -1;
  }
  board_w  = atoi(argv[1]);
  board_h  = atoi(argv[2]);
  n_boards = atoi(argv[3]);
  int board_n  = board_w * board_h;
  CvSize board_sz = cvSize( board_w, board_h );
  CvCapture* capture = cvCreateCameraCapture( 0 );
  assert( capture );

  cvNamedWindow( "Calibration" );
  //ALLOCATE STORAGE
  CvMat* image_points      = cvCreateMat(n_boards*board_n,2,CV_32FC1);
  CvMat* object_points     = cvCreateMat(n_boards*board_n,3,CV_32FC1);
  CvMat* point_counts      = cvCreateMat(n_boards,1,CV_32SC1);
  CvMat* intrinsic_matrix  = cvCreateMat(3,3,CV_32FC1);
  CvMat* distortion_coeffs = cvCreateMat(5,1,CV_32FC1);

  CvPoint2D32f* corners = new CvPoint2D32f[ board_n ];
  int corner_count;
  int successes = 0;
  int step, frame = 0;
```

Example 11-1. Reading a chessboard's width and height, reading and collecting the requested number of views, and calibrating the camera (continued)

```
IplImage *image = cvQueryFrame( capture );
IplImage *gray_image = cvCreateImage(cvGetSize(image),8,1);//subpixel

// CAPTURE CORNER VIEWS LOOP UNTIL WE'VE GOT n_boards
// SUCCESSFUL CAPTURES (ALL CORNERS ON THE BOARD ARE FOUND)
//
while(successes < n_boards) {
  //Skip every board_dt frames to allow user to move chessboard
  if(frame++ % board_dt == 0) {
     //Find chessboard corners:
     int found = cvFindChessboardCorners(
              image, board_sz, corners, &corner_count,
              CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS
     );

     //Get Subpixel accuracy on those corners
     cvCvtColor(image, gray_image, CV_BGR2GRAY);
     cvFindCornerSubPix(gray_image, corners, corner_count,
                cvSize(11,11),cvSize(-1,-1), cvTermCriteria(
                CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 30, 0.1 ));

     //Draw it
     cvDrawChessboardCorners(image, board_sz, corners,
                corner_count, found);
     cvShowImage( "Calibration", image );

     // If we got a good board, add it to our data
     if( corner_count == board_n ) {
        step = successes*board_n;
        for( int i=step, j=0; j<board_n; ++i,++j ) {
           CV_MAT_ELEM(*image_points, float,i,0) = corners[j].x;
           CV_MAT_ELEM(*image_points, float,i,1) = corners[j].y;
           CV_MAT_ELEM(*object_points,float,i,0) = j/board_w;
           CV_MAT_ELEM(*object_points,float,i,1) = j%board_w;
           CV_MAT_ELEM(*object_points,float,i,2) = 0.0f;
        }
        CV_MAT_ELEM(*point_counts, int,successes,0) = board_n;
        successes++;
     }
  } //end skip board_dt between chessboard capture

  //Handle pause/unpause and ESC
  int c = cvWaitKey(15);
  if(c == 'p'){
     c = 0;
     while(c != 'p' && c != 27){
          c = cvWaitKey(250);
     }
   }
   if(c == 27)
      return 0;
```

Example 11-1. Reading a chessboard's width and height, reading and collecting the requested number of views, and calibrating the camera (continued)

```
    image = cvQueryFrame( capture ); //Get next image
  } //END COLLECTION WHILE LOOP.

  //ALLOCATE MATRICES ACCORDING TO HOW MANY CHESSBOARDS FOUND
  CvMat* object_points2  = cvCreateMat(successes*board_n,3,CV_32FC1);
  CvMat* image_points2   = cvCreateMat(successes*board_n,2,CV_32FC1);
  CvMat* point_counts2   = cvCreateMat(successes,1,CV_32SC1);
  //TRANSFER THE POINTS INTO THE CORRECT SIZE MATRICES
  //Below, we write out the details in the next two loops. We could
  //instead have written:
  //image_points->rows = object_points->rows  = \
  //successes*board_n; point_counts->rows = successes;
  //
  for(int i = 0; i<successes*board_n; ++i) {
      CV_MAT_ELEM( *image_points2, float, i, 0) =
             CV_MAT_ELEM( *image_points, float, i, 0);
      CV_MAT_ELEM( *image_points2, float,i,1) =
             CV_MAT_ELEM( *image_points, float, i, 1);
      CV_MAT_ELEM(*object_points2, float, i, 0) =
             CV_MAT_ELEM( *object_points, float, i, 0) ;
      CV_MAT_ELEM( *object_points2, float, i, 1) =
             CV_MAT_ELEM( *object_points, float, i, 1) ;
      CV_MAT_ELEM( *object_points2, float, i, 2) =
             CV_MAT_ELEM( *object_points, float, i, 2) ;
  }
  for(int i=0; i<successes; ++i){ //These are all the same number
    CV_MAT_ELEM( *point_counts2, int, i, 0) =
             CV_MAT_ELEM( *point_counts, int, i, 0);
  }
  cvReleaseMat(&object_points);
  cvReleaseMat(&image_points);
  cvReleaseMat(&point_counts);

  // At this point we have all of the chessboard corners we need.
  // Initialize the intrinsic matrix such that the two focal
  // lengths have a ratio of 1.0
  //
  CV_MAT_ELEM( *intrinsic_matrix, float, 0, 0 ) = 1.0f;
  CV_MAT_ELEM( *intrinsic_matrix, float, 1, 1 ) = 1.0f;

  //CALIBRATE THE CAMERA!
  cvCalibrateCamera2(
      object_points2, image_points2,
      point_counts2,  cvGetSize( image ),
      intrinsic_matrix, distortion_coeffs,
      NULL, NULL,0  //CV_CALIB_FIX_ASPECT_RATIO
  );

  // SAVE THE INTRINSICS AND DISTORTIONS
  cvSave("Intrinsics.xml",intrinsic_matrix);
  cvSave("Distortion.xml",distortion_coeffs);
```

Example 11-1. Reading a chessboard's width and height, reading and collecting the requested number of views, and calibrating the camera (continued)

```
  // EXAMPLE OF LOADING THESE MATRICES BACK IN:
  CvMat *intrinsic = (CvMat*)cvLoad("Intrinsics.xml");
  CvMat *distortion = (CvMat*)cvLoad("Distortion.xml");

  // Build the undistort map that we will use for all
  // subsequent frames.
  //
  IplImage* mapx = cvCreateImage( cvGetSize(image), IPL_DEPTH_32F, 1 );
  IplImage* mapy = cvCreateImage( cvGetSize(image), IPL_DEPTH_32F, 1 );
  cvInitUndistortMap(
    intrinsic,
    distortion,
    mapx,
    mapy
  );
  // Just run the camera to the screen, now showing the raw and
  // the undistorted image.
  //
  cvNamedWindow( "Undistort" );
  while(image) {
    IplImage *t = cvCloneImage(image);
    cvShowImage( "Calibration", image ); // Show raw image
    cvRemap( t, image, mapx, mapy );     // Undistort image
    cvReleaseImage(&t);
    cvShowImage("Undistort", image);     // Show corrected image

    //Handle pause/unpause and ESC
    int c = cvWaitKey(15);
    if(c == 'p') {
       c = 0;
       while(c != 'p' && c != 27) {
            c = cvWaitKey(250);
       }
    }
    if(c == 27)
        break;
    image = cvQueryFrame( capture );
  }

  return 0;
}
```

Rodrigues Transform

When dealing with three-dimensional spaces, one most often represents rotations in that space by 3-by-3 matrices. This representation is usually the most convenient because multiplication of a vector by this matrix is equivalent to rotating the vector in some way. The downside is that it can be difficult to intuit just what 3-by-3 matrix goes with what rotation. An alternate and somewhat easier-to-visualize* representation for a rotation is in the form of a vector about which the rotation operates together with a single angle. In this case it is standard practice to use only a single vector whose direction encodes the direction of the axis to be rotated around and to use the size of the vector to encode the amount of rotation in a counterclockwise direction. This is easily done because the direction can be equally well represented by a vector of any magnitude; hence we can choose the magnitude of our vector to be equal to the magnitude of the rotation. The relationship between these two representations, the matrix and the vector, is captured by the Rodrigues transform.† Let r be the three-dimensional vector $r = [r_x\ r_y\ r_z]$; this vector implicitly defines θ, the magnitude of the rotation by the length (or magnitude) of r. We can then convert from this axis-magnitude representation to a rotation matrix R as follows:

$$R = \cos(\theta)\cdot I + (1-\cos(\theta))\cdot rr^{\mathrm{T}} + \sin(\theta)\cdot\begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ r_y & r_x & 0 \end{bmatrix}$$

We can also go from a rotation matrix back to the axis-magnitude representation by using:

$$\sin(\theta)\cdot\begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ r_y & r_x & 0 \end{bmatrix} = \frac{(R - R^{\mathrm{T}})}{2}$$

Thus we find ourselves in the situation of having one representation (the matrix representation) that is most convenient for computation and another representation (the Rodrigues representation) that is a little easier on the brain. OpenCV provides us with a function for converting from either representation to the other.

```
void  cvRodrigues2(
  const CvMat*  src,
  CvMat*        dst,
  CvMat*        jacobian = NULL
);
```

Suppose we have the vector r and need the corresponding rotation matrix representation R; we set `src` to be the 3-by-1 vector r and `dst` to be the 3-by-3 rotation matrix R. Conversely, we can set `src` to be a 3-by-3 rotation matrix R and `dst` to be a 3-by-1 vector r. In either case, `cvRodrigues2()` will do the right thing. The final argument is optional. If `jacobian` is not `NULL`, then it should be a pointer to a 3-by-9 or a 9-by-3 matrix that will

* This "easier" representation is not just for humans. Rotation in 3D space has only three components. For numerical optimization procedures, it is more efficient to deal with the three components of the Rodrigues representation than with the nine components of a 3-by-3 rotation matrix.

† Rodrigues was a 19th-century French mathematician.

be filled with the partial derivatives of the output array components with respect to the input array components. The `jacobian` outputs are mainly used for the internal optimization algorithms of `cvFindExtrinsicCameraParameters2()` and `cvCalibrateCamera2()`; your use of the `jacobian` function will mostly be limited to converting the outputs of `cvFindExtrinsicCameraParameters2()` and `cvCalibrateCamera2()` from the Rodrigues format of 1-by-3 or 3-by-1 axis-angle vectors to rotation matrices. For this, you can leave `jacobian` set to `NULL`.

Exercises

1. Use Figure 11-2 to derive the equations $x = f_x \cdot (X/Z) + c_x$ and $y - f_y \cdot (Y/Z) + c_y$ using similar triangles with a center-position offset.
2. Will errors in estimating the true center location (c_x, c_y) affect the estimation of other parameters such as focus?

   Hint: See the $q = MQ$ equation.
3. Draw an image of a square:
   a. Under radial distortion.
   b. Under tangential distortion.
   c. Under both distortions.
4. Refer to Figure 11-13. For perspective views, explain the following.
   a. Where does the "line at infinity" come from?
   b. Why do parallel lines on the object plane converge to a point on the image plane?
   c. Assume that the object and image planes are perpendicular to one another. On the object plane, starting at a point p_1, move 10 units directly away from the image plane to p_2. What is the corresponding movement distance on the image plane?
5. Figure 11-3 shows the outward-bulging "barrel distortion" effect of radial distortion, which is especially evident in the left panel of Figure 11-12. Could some lenses generate an inward-bending effect? How would this be possible?
6. Using a cheap web camera or cell phone, create examples of radial and tangential distortion in images of concentric squares or chessboards.
7. Calibrate the camera in exercise 6. Display the pictures before and after undistortion.
8. Experiment with numerical stability and noise by collecting many images of chessboards and doing a "good" calibration on all of them. Then see how the calibration parameters change as you reduce the number of chessboard images. Graph your results: camera parameters as a function of number of chessboard images.

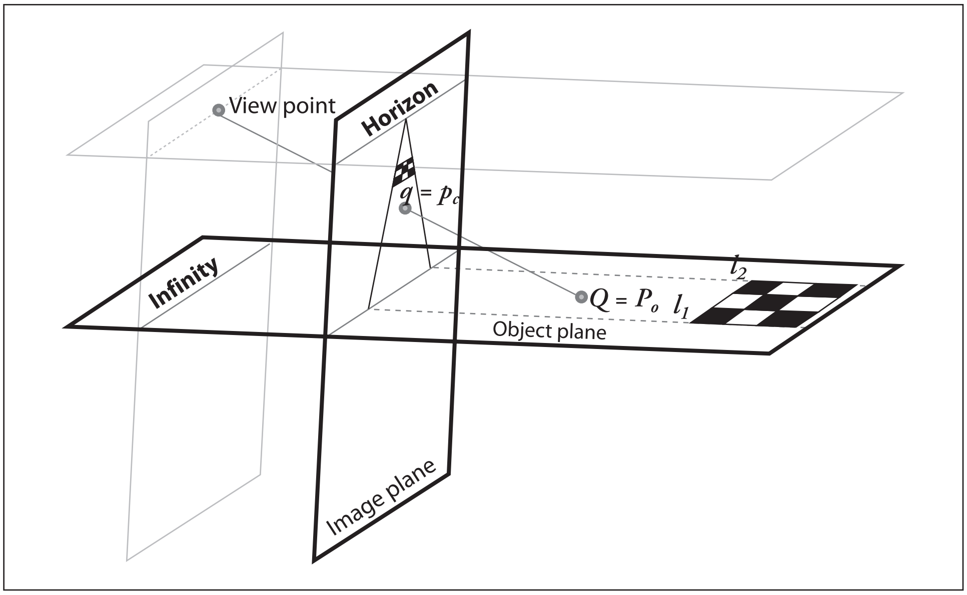


Figure 11-13. Homography diagram showing intersection of the object plane with the image plane and a viewpoint representing the center of projection

9. With reference to exercise 8, how do calibration parameters change when you use (say) 10 images of a 3-by-5, a 4-by-6, and a 5-by-7 chessboard? Graph the results.

10. High-end cameras typically have systems of lens that correct physically for distortions in the image. What might happen if you nevertheless use a multiterm distortion model for such a camera?

    Hint: This condition is known as *overfitting*.

11. *Three-dimensional joystick trick*. Calibrate a camera. Using video, wave a chessboard around and use `cvFindExtrinsicCameraParams2()` as a 3D joystick. Remember that `cvFindExtrinsicCameraParams2()` outputs rotation as a 3-by-1 or 1-by-3 vector axis of rotation, where the magnitude of the vector represents the counterclockwise angle of rotation along with a 3D translation vector.

    a. Output the chessboard's axis and angle of the rotation along with where it is (i.e., the translation) in real time as you move the chessboard around. Handle cases where the chessboard is not in view.

    b. Use `cvRodrigues2()` to translate the output of `cvFindExtrinsicCameraParams2()` into a 3-by-3 rotation matrix and a translation vector. Use this to animate a simple 3D stick figure of an airplane rendered back into the image in real time as you move the chessboard in view of the video camera.

CHAPTER 12

Projection and 3D Vision

In this chapter we'll move into three-dimensional vision, first with projections and then with multicamera stereo depth perception. To do this, we'll have to carry along some of the concepts from Chapter 11. We'll need the *camera instrinsics* matrix *M*, the *distortion coefficients*, the rotation matrix *R*, the translation vector *T,* and especially the *homography matrix H*.

We'll start by discussing projection into the 3D world using a calibrated camera and reviewing affine and projective transforms (which we first encountered in Chapter 6); then we'll move on to an example of how to get a bird's-eye view of a ground plane.* We'll also discuss POSIT, an algorithm that allows us to find the 3D pose (position and rotation) of a known 3D object in an image.

We will then move into the three-dimensional geometry of multiple images. In general, there is no reliable way to do calibration or to extract 3D information without multiple images. The most obvious case in which we use multiple images to reconstruct a three-dimensional scene is *stereo vision*. In stereo vision, features in two (or more) images taken at the same time from separate cameras are matched with the corresponding features in the other images, and the differences are analyzed to yield depth information. Another case is *structure from motion*. In this case we may have only a single camera, but we have multiple images taken at different times and from different places. In the former case we are primarily interested in *disparity effects* (triangulation) as a means of computing distance. In the latter, we compute something called the *fundamental matrix* (relates two different views together) as the source of our scene understanding. Let's get started with projection.

Projections

Once we have calibrated the camera (see Chapter 11), it is possible to unambiguously project points in the physical world to points in the image. This means that, given a location in the three-dimensional physical coordinate frame attached to the camera, we

* This is a recurrent problem in robotics as well as many other vision applications.

can compute where on the imager, in pixel coordinates, an external 3D point should appear. This transformation is accomplished by the OpenCV routine cvProjectPoints2().

```
void cvProjectPoints2(
    const CvMat* object_points,
    const CvMat* rotation_vector,
    const CvMat* translation_vector,
    const CvMat* intrinsic_matrix,
    const CvMat* distortion_coeffs,
    CvMat*       image_points,
    CvMat*       dpdrot          = NULL,
    CvMat*       dpdt            = NULL,
    CvMat*       dpdf            = NULL,
    CvMat*       dpdc            = NULL,
    CvMat*       dpddist         = NULL,
    double       aspectRatio     = 0
);
```

At first glance the number of arguments might be a little intimidating, but in fact this is a simple function to use. The cvProjectPoints2() routine was designed to accommodate the (very common) circumstance where the points you want to project are located on some rigid body. In this case, it is natural to represent the points not as just a list of locations in the camera coordinate system but rather as a list of locations in the object's own body centered coordinate system; then we can add a rotation and a translation to specify the relationship between the object coordinates and the camera's coordinate system. In fact, cvProjectPoints2() is used internally in cvCalibrateCamera2(), and of course this is the way cvCalibrateCamera2() organizes its own internal operation. All of the optional arguments are primarily there for use by cvCalibrateCamera2(), but sophisticated users might find them handy for their own purposes as well.

The first argument, object_points, is the list of points you want projected; it is just an N-by-3 matrix containing the point locations. You can give these in the object's own local coordinate system and then provide the 3-by-1 matrices rotation_vector* and translation_vector to relate the two coordinates. If in your particular context it is easier to work directly in the camera coordinates, then you can just give object_points in that system and set both rotation_vector and translation_vector to contain 0s.†

The intrinsic_matrix and distortion_coeffs are just the camera intrinsic information and the distortion coefficients that come from cvCalibrateCamera2() discussed in Chapter 11. The image_points argument is an N-by-2 matrix into which the results of the computation will be written.

Finally, the long list of optional arguments dpdrot, dpdt, dpdf, dpdc, and dpddist are all Jacobian matrices of partial derivatives. These matrices relate the image points to each of the different input parameters. In particular: dpdrot is an N-by-3 matrix of partial derivatives of image points with respect to components of the rotation vector; dpdt is an

* The "rotation vector" is in the usual Rodrigues representation.

† Remember that this rotation vector is an axis-angle representation of the rotation, so being set to all 0s means it has zero magnitude and thus "no rotation".

N-by-3 matrix of partial derivatives of image points with respect to components of the translation vector; `dpdf` is an N-by-2 matrix of partial derivatives of image points with respect to f_x and f_y; `dpdc` is an N-by-2 matrix of partial derivatives of image points with respect to c_x and c_y; and `dpddist` is an N-by-4 matrix of partial derivatives of image points with respect to the distortion coefficients. In most cases, you will just leave these as `NULL`, in which case they will not be computed. The last parameter, aspectRatio, is also optional; it is used for derivatives only when the aspect ratio is fixed in `cvCalibrateCamera2()` or `cvStereoCalibrate()`. If this parameter is not 0 then the derivatives `dpdf` are adjusted.

Affine and Perspective Transformations

Two transformations that come up often in the OpenCV routines we have discussed—as well as in other applications you might write yourself—are the affine and perspective transformations. We first encountered these in Chapter 6. As implemented in OpenCV, these routines affect either lists of points or entire images, and they map points on one location in the image to a different location, often performing subpixel interpolation along the way. You may recall that an affine transform can produce any parallelogram from a rectangle; the perspective transform is more general and can produce any trapezoid from a rectangle.

The *perspective transformation* is closely related to the *perspective projection*. Recall that the perspective projection maps points in the three-dimensional physical world onto points on the two-dimensional image plane along a set of projection lines that all meet at a single point called *the center of projection*. The perspective transformation, which is a specific kind of *homography*,* relates two different images that are alternative projections of the same three-dimensional object onto two different *projective planes* (and thus, for nondegenerate configurations such as the plane physically intersecting the 3D object, typically to two different centers of projection).

These projective transformation-related functions were discussed in detail in Chapter 6; for convenience, we summarize them here in Table 12-1.

Table 12-1. Affine and perspective transform functions

Function	Use
`cvTransform()`	Affine transform a list of points
`cvWarpAffine()`	Affine transform a whole image
`cvGetAffineTransform()`	Fill in affine transform matrix parameters
`cv2DRotationMatrix()`	Fill in affine transform matrix parameters
`cvGetQuadrangleSubPix()`	Low-overhead whole image affine transform
`cvPerspectiveTransform()`	Perspective transform a list of points
`cvWarpPerspective()`	Perspective transform a whole image
`cvGetPerspectiveTransform()`	Fill in perspective transform matrix parameters

* Recall from Chapter 11 that this special kind of homography is known as *planar homography*.

Bird's-Eye View Transform Example

A common task in robotic navigation, typically used for planning purposes, is to convert the robot's camera view of the scene into a top-down "bird's-eye" view. In Figure 12-1, a robot's view of a scene is turned into a bird's-eye view so that it can be subsequently overlaid with an alternative representation of the world created from scanning laser range finders. Using what we've learned so far, we'll look in detail about how to use our calibrated camera to compute such a view.

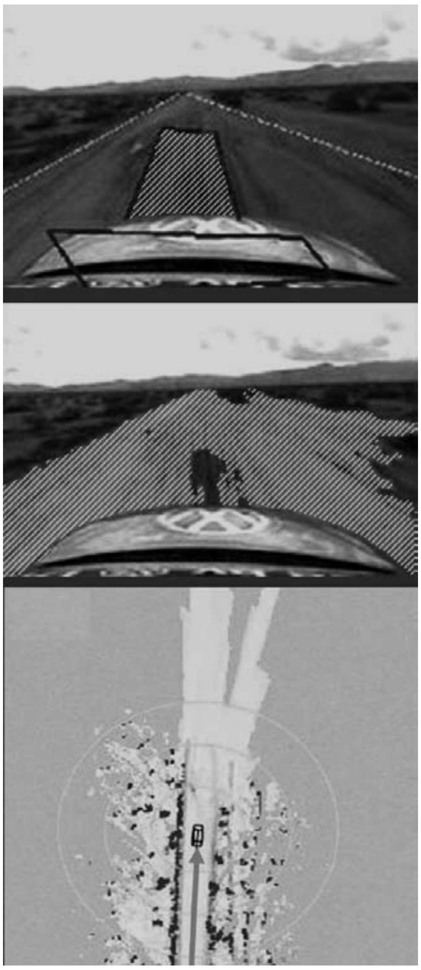


Figure 12-1. Bird's-eye view: A camera on a robot car looks out at a road scene where laser range finders have identified a region of "road" in front of the car and marked it with a box (top); vision algorithms have segmented the flat, roadlike areas (center); the segmented road areas are converted to a bird's-eye view and merged with the bird's-eye view laser map (bottom)

To get a bird's-eye view,[*] we'll need our camera intrinsics and distortion matrices from the calibration routine. Just for the sake of variety, we'll read the files from disk. We put a chessboard on the floor and use that to obtain a ground plane image for a robot cart; we then remap that image into a bird's-eye view. The algorithm runs as follows.

1. Read the intrinsics and distortion models for the camera.
2. Find a known object on the ground plane (in this case, a chessboard). Get at least four points at subpixel accuracy.
3. Enter the found points into `cvGetPerspectiveTransform()` (see Chapter 6) to compute the homography matrix *H* for the ground plane view.
4. Use `cvWarpPerspective( )` (again, see Chapter 6) with the flags `CV_INTER_LINEAR + CV_WARP_INVERSE_MAP + CV_WARP_FILL_OUTLIERS` to obtain a frontal parallel (bird's-eye) view of the ground plane.

Example 12-1 shows the full working code for bird's-eye view.

Example 12-1. Bird's-eye view

```
//Call:
//  birds-eye board_w board_h instrinics distortion image_file
// ADJUST VIEW HEIGHT using keys ‘u’ up, ‘d’ down. ESC to quit.
//

int main(int argc, char* argv[]) {
  if(argc != 6) return -1;

  // INPUT PARAMETERS:
  //
  int       board_w    = atoi(argv[1]);
  int       board_h    = atoi(argv[2]);
  int       board_n    = board_w * board_h;
  CvSize    board_sz   = cvSize( board_w, board_h );
  CvMat*    intrinsic  = (CvMat*)cvLoad(argv[3]);
  CvMat*    distortion = (CvMat*)cvLoad(argv[4]);
  IplImage* image      = 0;
  IplImage* gray_image = 0;
  if( (image = cvLoadImage(argv[5])) == 0 ) {
    printf(“Error: Couldn’t load %s\n”,argv[5]);
    return -1;
  }
  gray_image = cvCreateImage( cvGetSize(image), 8, 1 );
  cvCvtColor(image, gray_image, CV_BGR2GRAY );

  // UNDISTORT OUR IMAGE
  //
  IplImage* mapx = cvCreateImage( cvGetSize(image), IPL_DEPTH_32F, 1 );
  IplImage* mapy = cvCreateImage( cvGetSize(image), IPL_DEPTH_32F, 1 );
```

[*] The bird's-eye view technique also works for transforming perspective views of any plane (e.g., a wall or ceiling) into frontal parallel views.

Example 12-1. Bird's-eye view (continued)

```
//This initializes rectification matrices
//
cvInitUndistortMap(
  intrinsic,
  distortion,
  mapx,
  mapy
);
IplImage *t = cvCloneImage(image);

// Rectify our image
//
cvRemap( t, image, mapx, mapy );

// GET THE CHESSBOARD ON THE PLANE
//
cvNamedWindow("Chessboard");
CvPoint2D32f* corners = new CvPoint2D32f[ board_n ];
int corner_count = 0;
int found = cvFindChessboardCorners(
  image,
  board_sz,
  corners,
  &corner_count,
  CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS
);
if(!found){
  printf("Couldn't aquire chessboard on %s, "
    "only found %d of %d corners\n",
    argv[5],corner_count,board_n
  );
  return -1;
}
//Get Subpixel accuracy on those corners:
cvFindCornerSubPix(
  gray_image,
  corners,
  corner_count,
  cvSize(11,11),
  cvSize(-1,-1),
  cvTermCriteria( CV_TERMCRIT_EPS | CV_TERMCRIT_ITER, 30, 0.1 )
);

//GET THE IMAGE AND OBJECT POINTS:
// We will choose chessboard object points as (r,c):
// (0,0), (board_w-1,0), (0,board_h-1), (board_w-1,board_h-1).
//
CvPoint2D32f objPts[4], imgPts[4];
objPts[0].x = 0;         objPts[0].y = 0;
objPts[1].x = board_w-1; objPts[1].y = 0;
objPts[2].x = 0;         objPts[2].y = board_h-1;
objPts[3].x = board_w-1; objPts[3].y = board_h-1;
imgPts[0]   = corners[0];
```

Example 12-1. Bird's-eye view (continued)

```
imgPts[1]   = corners[board_w-1];
imgPts[2]   = corners[(board_h-1)*board_w];
imgPts[3]   = corners[(board_h-1)*board_w + board_w-1];

// DRAW THE POINTS in order: B,G,R,YELLOW
//
cvCircle( image, cvPointFrom32f(imgPts[0]), 9, CV_RGB(0,0,255),   3);
cvCircle( image, cvPointFrom32f(imgPts[1]), 9, CV_RGB(0,255,0),   3);
cvCircle( image, cvPointFrom32f(imgPts[2]), 9, CV_RGB(255,0,0),   3);
cvCircle( image, cvPointFrom32f(imgPts[3]), 9, CV_RGB(255,255,0), 3);

// DRAW THE FOUND CHESSBOARD
//
cvDrawChessboardCorners(
  image,
  board_sz,
  corners,
  corner_count,
  found
);
cvShowImage( “Chessboard”, image );

// FIND THE HOMOGRAPHY
//
CvMat *H = cvCreateMat( 3, 3, CV_32F);
cvGetPerspectiveTransform( objPts, imgPts, H);

// LET THE USER ADJUST THE Z HEIGHT OF THE VIEW
//
float Z = 25;
int key = 0;
IplImage *birds_image = cvCloneImage(image);
cvNamedWindow(“Birds_Eye”);

// LOOP TO ALLOW USER TO PLAY WITH HEIGHT:
//
// escape key stops
//
while(key != 27) {
  // Set the height
  //
  CV_MAT_ELEM(*H,float,2,2) = Z;

  // COMPUTE THE FRONTAL PARALLEL OR BIRD’S-EYE VIEW:
  // USING HOMOGRAPHY TO REMAP THE VIEW
  //
  cvWarpPerspective(
    image,
    birds_image,
    H,
    CV_INTER_LINEAR | CV_WARP_INVERSE_MAP | CV_WARP_FILL_OUTLIERS
  );
  cvShowImage( “Birds_Eye”, birds_image );
```

Example 12-1. Bird's-eye view (continued)

```
    key = cvWaitKey();
    if(key == 'u') Z += 0.5;
    if(key == 'd') Z -= 0.5;
  }

  cvSave("H.xml",H); //We can reuse H for the same camera mounting
  return 0;
}
```

Once we have the homography matrix and the height parameter set as we wish, we could then remove the chessboard and drive the cart around, making a bird's-eye view video of the path, but we'll leave that as an exercise for the reader. Figure 12-2 shows the input at left and output at right for the bird's-eye view code.

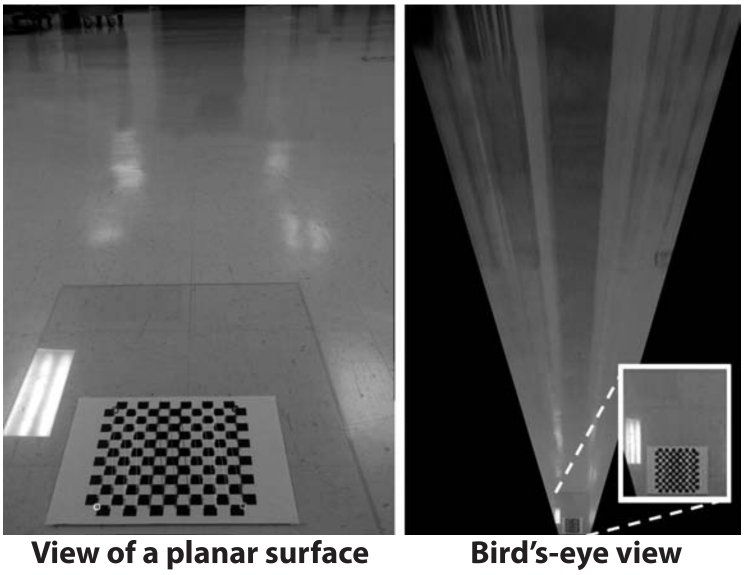


Figure 12-2. Bird's-eye view example

POSIT: 3D Pose Estimation

Before moving on to stereo vision, we should visit a useful algorithm that can estimate the positions of known objects in three dimensions. POSIT (aka "Pose from Orthography and Scaling with Iteration") is an algorithm originally proposed in 1992 for computing the pose (the position *T* and orientation *R* described by six parameters [DeMenthon92]) of a 3D object whose exact dimensions are known. To compute this pose, we must find on the image the corresponding locations of at least four non-coplanar points on the surface of that object. The first part of the algorithm, *pose from orthography and scaling*

(POS), assumes that the points on the object are all at effectively the same depth* and that size variations from the original model are due solely to scaling with distance from the camera. In this case there is a closed-form solution for that object's 3D pose based on scaling. The assumption that the object points are all at the same depth effectively means that the object is far enough away from the camera that we can neglect any internal depth differences within the object; this assumption is known as the *weak-perspective* approximation.

Given that we know the camera intrinsics, we can find the perspective scaling of our known object and thus compute its approximate pose. This computation will not be very accurate, but we can then project where our four observed points would go if the true 3D object were at the pose we calculated through POS. We then start all over again with these new point positions as the inputs to the POS algorithm. This process typically converges within four or five iterations to the true object pose—hence the name "POS algorithm *with iteration*". Remember, though, that all of this assumes that the internal depth of the object is in fact small compared to the distance away from the camera. If this assumption is not true, then the algorithm will either not converge or will converge to a "bad pose". The OpenCV implementation of this algorithm will allow us to track more than four (non-coplanar) points on the object to improve pose estimation accuracy.

The POSIT algorithm in OpenCV has three associated functions: one to allocate a data structure for the pose of an individual object, one to de-allocate the same data structure, and one to actually implement the algorithm.

```
CvPOSITObject* cvCreatePOSITObject(
    CvPoint3D32f* points,
    int           point_count
);
void cvReleasePOSITObject(
    CvPOSITObject** posit_object
);
```

The `cvCreatePOSITObject()` routine just takes `points` (a set of three-dimensional points) and `point_count` (an integer indicating the number of points) and returns a pointer to an allocated POSIT object structure. Then `cvReleasePOSITObject()` takes a pointer to such a structure pointer and de-allocates it (setting the pointer to `NULL` in the process).

```
void cvPOSIT(
    CvPOSITObject* posit_object,
    CvPoint2D32f*  image_points,
    double         focal_length,
    CvTermCriteria criteria,
    float*         rotation_matrix,
    float*         translation_vector
);
```

* The construction finds a reference plane through the object that is parallel to the image plane; this plane through the object then has a single distance *Z* from the image plane. The 3D points on the object are first projected to this plane through the object and then projected onto the image plane using perspective projection. The result is scaled orthographic projection, and it makes relating object size to depth particularly easy.

Now, on to the POSIT function itself. The argument list to `cvPOSIT()` is different stylistically than most of the other functions we have seen in that it uses the "old style" arguments common in earlier versions of OpenCV.* Here `posit_object` is just a pointer to the POSIT object that you are trying to track, and `image_points` is a list of the locations of the corresponding points in the image plane (notice that these are 32-bit values, thus allowing for subpixel locations). The current implementation of `cvPOSIT()` assumes square pixels and thus allows only a single value for the `focal_length` parameter instead of one in the *x* and one in the *y* directions. Because `cvPOSIT()` is an iterative algorithm, it requires a termination criteria: `criteria` is of the usual form and indicates when the fit is "good enough". The final two parameters, `rotation_matrix` and `translation_vector`, are analogous to the same arguments in earlier routines; observe, however, that these are pointers to `float` and so are just the data part of the matrices you would obtain from calling (for example) `cvCalibrateCamera2()`. In this case, given a matrix *M*, you would want to use something like `M->data.fl` as an argument to `cvPOSIT()`.

When using POSIT, keep in mind that the algorithm does not benefit from additional surface points that are coplanar with other points already on the surface. Any point lying on a plane defined by three other points will not contribute anything useful to the algorithm. In fact, extra coplanar points can cause degeneracies that hurt the algorithm's performance. Extra non-coplanar points will help the algorithm. Figure 12-3 shows the POSIT algorithm in use with a toy plane [Tanguay00]. The plane has marking lines on it, which are used to define four non-coplanar points. These points were fed into `cvPOSIT()`, and the resulting `rotation_matrix` and `translation_vector` are used to control a flight simulator.

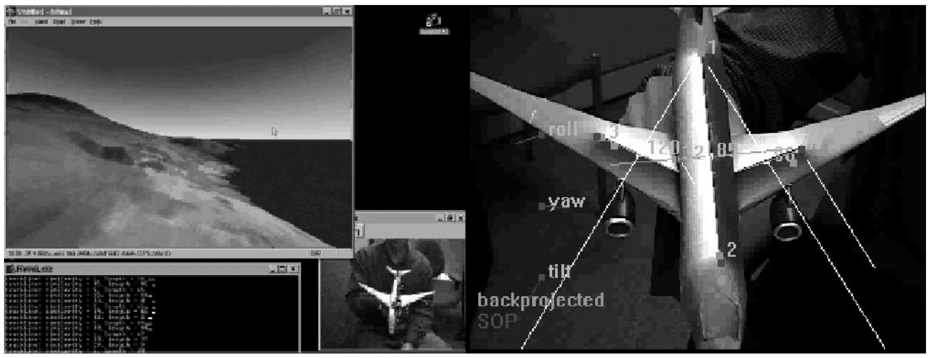


Figure 12-3. POSIT algorithm in use: four non-coplanar points on a toy jet are used to control a flight simulator

* You might have noticed that many function names end in "2". More often than not, this is because the function in the current release in the library has been modified from its older incarnation to use the newer style of arguments.

Stereo Imaging

Now we are in a position to address *stereo imaging*.* We all are familiar with the stereo imaging capability that our eyes give us. To what degree can we emulate this capability in computational systems? Computers accomplish this task by finding correspondences between points that are seen by one imager and the same points as seen by the other imager. With such correspondences and a known baseline separation between cameras, we can compute the 3D location of the points. Although the search for corresponding points can be computationally expensive, we can use our knowledge of the geometry of the system to narrow down the search space as much as possible. In practice, stereo imaging involves four steps when using two cameras.

1. Mathematically remove radial and tangential lens distortion; this is called *undistortion* and is detailed in Chapter 11. The outputs of this step are undistorted images.
2. Adjust for the angles and distances between cameras, a process called *rectification*. The outputs of this step are images that are row-aligned† and rectified.
3. Find the same features in the left and right‡ camera views, a process known as *correspondence*. The output of this step is a *disparity* map, where the disparities are the differences in *x*-coordinates on the image planes of the same feature viewed in the left and right cameras: $x^l - x^r$.
4. If we know the geometric arrangement of the cameras, then we can turn the disparity map into distances by *triangulation*. This step is called *reprojection*, and the output is a depth map.

We start with the last step to motivate the first three.

Triangulation

Assume that we have a perfectly undistorted, aligned, and measured stereo rig as shown in Figure 12-4: two cameras whose image planes are exactly coplanar with each other, with exactly parallel optical axes (the optical axis is the ray from the center of projection *O* through the principal point *c* and is also known as the *principal ray*§) that are a known distance apart, and with equal focal lengths $f_l = f_r$. Also, assume for now that the *principal points* c_x^{left} and c_x^{right} have been calibrated to have the same pixel coordinates in their respective left and right images. Please don't confuse these principal points with the center of the image. A principal point is where the principal ray intersects the imaging

* Here we give just a high-level understanding. For details, we recommend the following texts: Trucco and Verri [Trucco98], Hartley and Zisserman [Hartley06], Forsyth and Ponce [Forsyth03], and Shapiro and Stockman [Shapiro02]. The stereo rectification sections of these books will give you the background to tackle the original papers cited in this chapter.

† By "row-aligned" we mean that the two image planes are coplanar and that the image rows are exactly aligned (in the same direction and having the same *y*-coordinates).

‡ Every time we refer to left and right cameras you can also use vertically oriented up and down cameras, where disparities are in the *y*-direction rather than the *x*-direction.

§ Two parallel principal rays are said to intersect at infinity.

plane. This intersection depends on the optical axis of the lens. As we saw in Chapter 11, the image plane is rarely aligned exactly with the lens and so the center of the imager is almost never exactly aligned with the principal point.

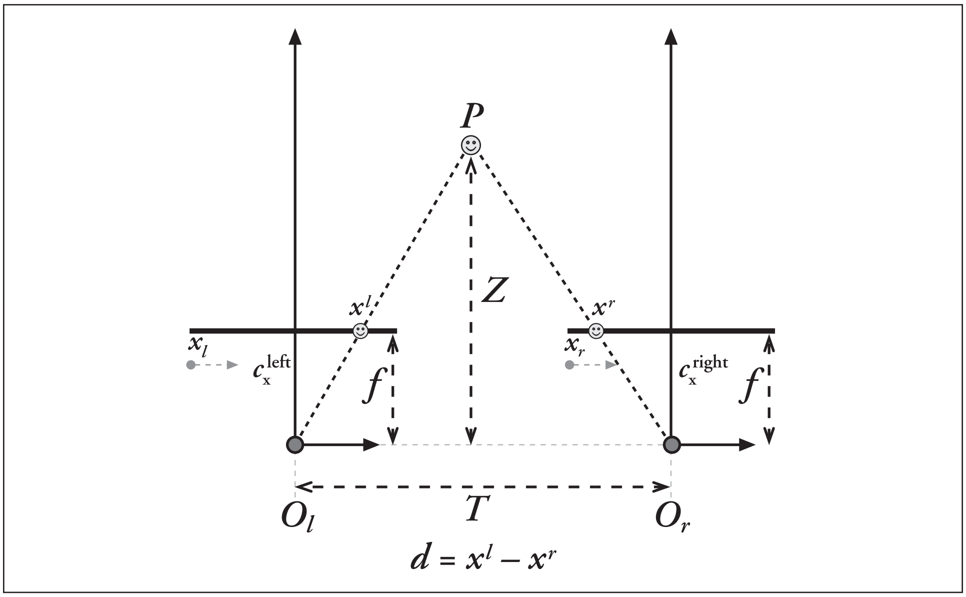


Figure 12-4. With a perfectly undistorted, aligned stereo rig and known correspondence, the depth Z can be found by similar triangles; the principal rays of the imagers begin at the centers of projection O_l and O_r and extend through the principal points of the two image planes at c_l and c_r

Moving on, let's further assume the images are row-aligned and that every pixel row of one camera aligns exactly with the corresponding row in the other camera.* We will call such a camera arrangement *frontal parallel*. We will also assume that we can find a point P in the physical world in the left and the right image views at p_l and p_r, which will have the respective horizontal coordinates x^l and x^r.

In this simplified case, taking x^l and x^r to be the horizontal positions of the points in the left and right imager (respectively) allows us to show that the depth is inversely proportional to the disparity between these views, where the disparity is defined simply by $d = x^l - x^r$. This situation is shown in Figure 12-4, where we can easily derive the depth Z by using similar triangles. Referring to the figure, we have:†

* This makes for quite a few assumptions, but we are just looking at the basics right now. Remember that the process of rectification (to which we will return shortly) is how we get things done mathematically when these assumptions are not physically true. Similarly, in the next sentence we will temporarily "assume away" the correspondence problem.

† This formula is predicated on the principal rays intersecting at infinity. However, as you will see in "Calibrated Stereo Rectification" (later in this chapter), we derive stereo rectification relative to the principal points c_x^{left} and c_x^{right}. In our derivation, if the principal rays intersect at infinity then the principal points have the same coordinates and so the formula for depth holds as is. However, if the principal rays intersect at a finite distance then the principal points will not be equal and so the equation for depth becomes $Z = fT / (d - (c_x^{\text{left}} - c_x^{\text{right}}))$.

$$\frac{T-(x^l-x^r)}{Z-f}=\frac{T}{Z} \quad\Rightarrow\quad Z=\frac{fT}{x^l-x^r}$$

Since depth is inversely proportional to disparity, there is obviously a nonlinear relationship between these two terms. When disparity is near 0, small disparity differences make for large depth differences. When disparity is large, small disparity differences do not change the depth by much. The consequence is that stereo vision systems have high depth resolution only for objects relatively near the camera, as Figure 12-5 makes clear.

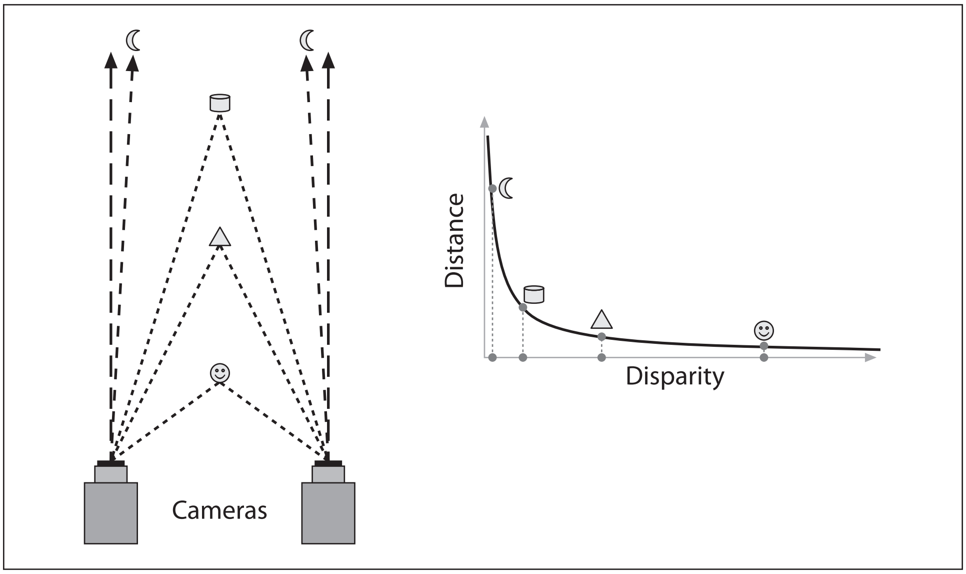


Figure 12-5. Depth and disparity are inversely related, so fine-grain depth measurement is restricted to nearby objects

We have already seen many coordinate systems in the discussion of calibration in Chapter 11. Figure 12-6 shows the 2D and 3D coordinate systems used in OpenCV for stereo vision. Note that it is a right-handed coordinate system: if you point your right index finger in the direction of *X* and bend your right middle finger in the direction of *Y*, then your thumb will point in the direction of the principal ray. The left and right imager pixels have image origins at upper left in the image, and pixels are denoted by coordinates (x_l, y_l) and (x_r, y_r), respectively. The center of projection are at O_l and O_r with principal rays intersecting the image plane at the principal point (not the center) (c_x, c_y). After mathematical rectification, the cameras are row-aligned (coplanar and horizontally aligned), displaced from one another by *T*, and of the same focal length *f*.

With this arrangement it is relatively easily to solve for distance. Now we must spend some energy on understanding how we can map a real-world camera setup into a geometry that resembles this ideal arrangement. In the real world, cameras will almost never be exactly aligned in the frontal parallel configuration depicted in Figure 12-4. Instead,

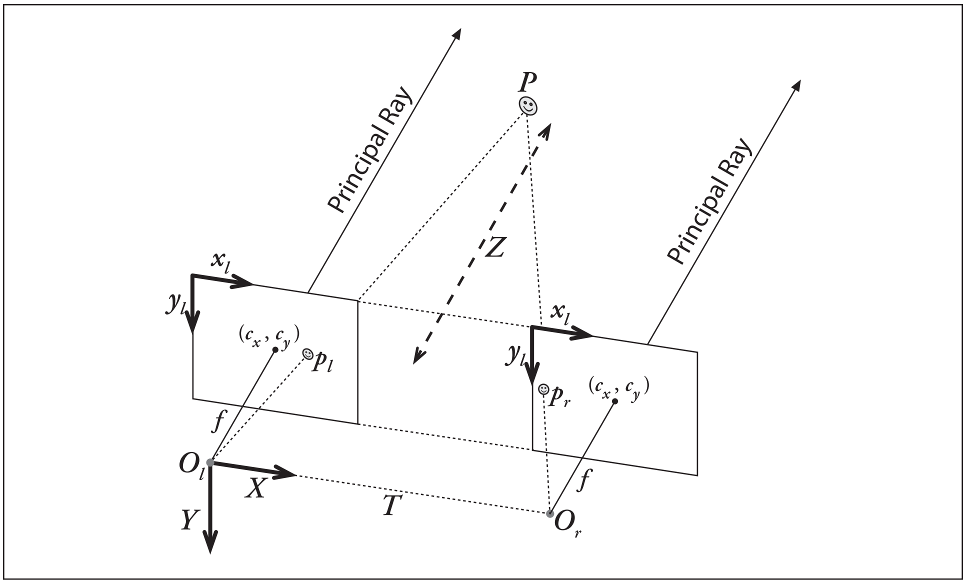


Figure 12-6. Stereo coordinate system used by OpenCV for undistorted rectified cameras: the pixel coordinates are relative to the upper left corner of the image, and the two planes are row-aligned; the camera coordinates are relative to the left camera's center of projection

we will mathematically find image projections and distortion maps that will rectify the left and right images into a frontal parallel arrangement. When designing your stereo rig, it is best to arrange the cameras approximately frontal parallel and as close to horizontally aligned as possible. This physical alignment will make the mathematical tranformations more tractable. If you don't align the cameras at least approximately, then the resulting mathematical alignment can produce extreme image distortions and so reduce or eliminate the stereo overlap area of the resulting images.* For good results, you'll also need synchronized cameras. If they don't capture their images at the exact same time, then you will have problems if anything is moving in the scene (including the cameras themselves). If you do not have synchronized cameras, you will be limited to using stereo with stationary cameras viewing static scenes.

Figure 12-7 depicts the real situation between two cameras and the mathematical alignment we want to achieve. To perform this mathematical alignment, we need to learn more about the geometry of two cameras viewing a scene. Once we have that geometry defined and some terminology and notation to describe it, we can return to the problem of alignment.

* The exception to this advice is that for applications where we want more resolution at close range; in this case, we tilt the cameras slightly in toward each other so that their principal rays intersect at a finite distance. After mathematical alignment, the effect of such inward verging cameras is to introduce an x-offset that is subtracted from the disparity. This may result in negative disparities, but we can thus gain finer depth resolution at the nearby depths of interest.

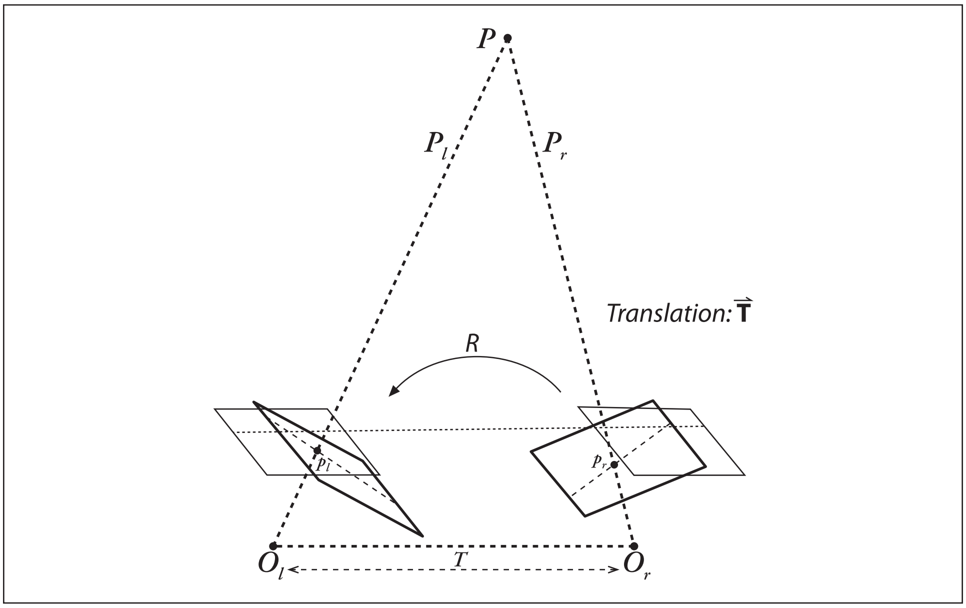


Figure 12-7. Our goal will be to mathematically (rather than physically) align the two cameras into one viewing plane so that pixel rows between the cameras are exactly aligned with each other

Epipolar Geometry

The basic geometry of a stereo imaging system is referred to as *epipolar geometry*. In essence, this geometry combines two pinhole models (one for each camera*) and some interesting new points called the *epipoles* (see Figure 12-8). Before explaining what these epipoles are good for, we will start by taking a moment to define them clearly and to add some new related terminology. When we are done, we will have a concise understanding of this overall geometry and will also find that we can narrow down considerably the possible locations of corresponding points on the two stereo cameras. This added discovery will be important to practical stereo implementations.

For each camera there is now a separate center of projection, O_l and O_r, and a pair of corresponding projective planes, Π_l and Π_r. The point P in the physical world has a projection onto each of the projective planes that we can label p_l and p_r. The new points of interest are the epipoles. An epipole e_l (resp. e_r) on image plane Π_l (resp. Π_r) is defined as the image of the center of projection of the other camera O_r (resp. O_l). The plane in space formed by the actual viewed point P and the two epipoles e_l and e_r (or, equivalently, through the two centers of projection O_r and O_l) is called the *epipolar plane*, and the lines p_le_l and p_re_r (from the points of projection to the corresponding epipolar points) are called the *epipolar lines*.†

* Since we are actually dealing with real lenses and not pinhole cameras, it is important that the two images be undistorted; see Chapter 11.

† You can see why the epipoles did not come up before: as the planes approach being perfectly parallel, the epipoles head out toward infinity!

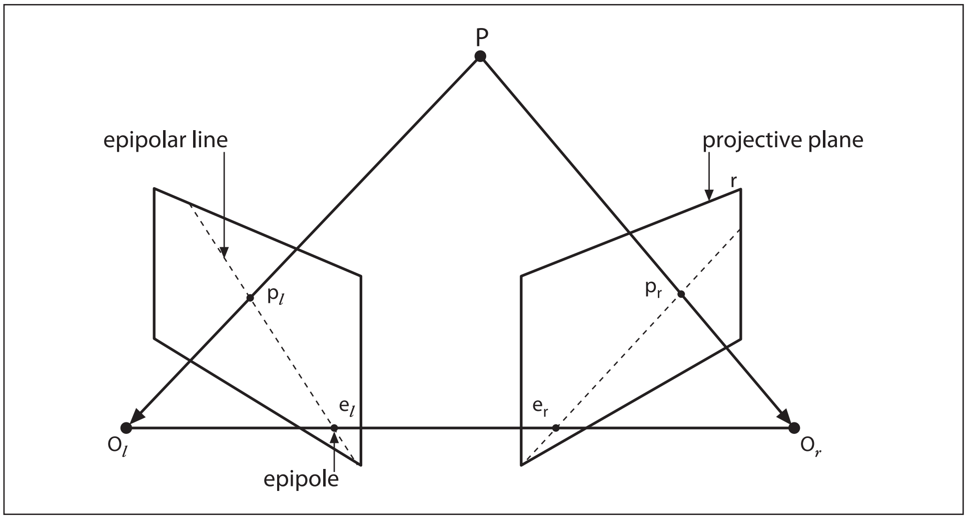


Figure 12-8. The epipolar plane is defined by the observed point P and the two centers of projection, O_l and O_r; the epipoles are located at the point of intersection of the line joining the centers of projection and the two projective planes

To understand the utility of the epipoles we first recall that, when we see a point in the physical world projected onto our right (or left) image plane, that point could actually be located anywhere along a entire line of points formed by the ray going from O_r out through p_r (or O_l through p_l) because, with just that single camera, we do not know the distance to the point we are viewing. More specifically, take for example the point P as seen by the camera on the right. Because that camera sees only p_r (the projection of P onto Π_r), the actual point P could be located anywhere on the line defined by p_r and O_r. This line obviously contains P, but it contains a lot of other points, too. What is interesting, though, is to ask what that line looks like projected onto the left image plane Π_l; in fact, it is the epipolar line defined by p_l and e_l. To put that into English, the image of all of the possible locations of a *point* seen in one imager is the *line* that goes through the corresponding point and the epipolar point on the other imager.

We'll now summarize some facts about stereo camera epipolar geometry (and why we care).

- Every 3D point in view of the cameras is contained in an epipolar plane that intersects each image in an epipolar line.
- Given a feature in one image, its matching view in the other image *must* lie along the corresponding epipolar line. This is known as the *epipolar constraint*.
- The epipolar constraint means that the possible two-dimensional search for matching features across two imagers becomes a one-dimensional search along the epipolar lines once we know the epipolar geometry of the stereo rig. This is not only a vast computational savings, it also allows us to reject a lot of points that could otherwise lead to spurious correspondences.

- Order is preserved. If points *A* and *B* are visible in both images and occur horizontally in that order in one imager, then they occur horizontally in that order in the other imager.*

The Essential and Fundamental Matrices

You might think that the next step would be to introduce some OpenCV function that computes these epipolar lines for us, but we actually need two more ingredients before we can arrive at that point. These ingredients are the *essential matrix E* and the *fundamental matrix F*.† The matrix *E* contains information about the translation and rotation that relate the two cameras in physical space (see Figure 12-9), and *F* contains the same information as *E* in addition to information about the intrinsics of both cameras.‡ Because *F* embeds information about the intrinsic parameters, it relates the two cameras in pixel coordinates.

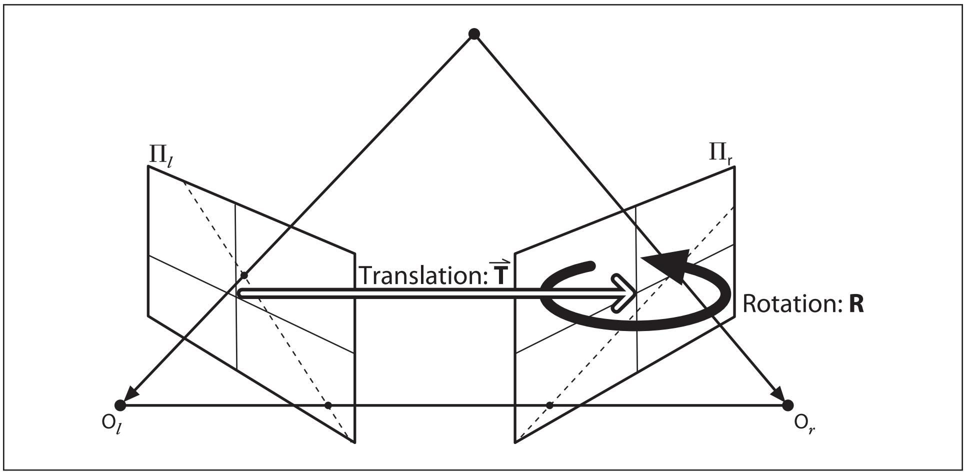


Figure 12-9. The essential geometry of stereo imaging is captured by the essential matrix E, which contains all of the information about the translation T and the rotation R, which describe the location of the second camera relative to the first in global coordinates

* Because of occlusions and areas of overlapping view, it is certainly possible that both cameras do not see the same points. Nevertheless, order is maintained. If points *A*, *B*, and *C* are arranged left to right on the left imager and if *B* is not seen on the right imager owing to occlusion, then the right imager will still see points *A* and *C* left to right.

† The next subsections are a bit mathy. If you do not like math then just skim over them; at least you'll have confidence that somewhere, someone understands all of this stuff. For simple applications, you can just use the machinery that OpenCV provides without the need for all of the details in these next few pages.

‡ The astute reader will recognize that *E* was described in almost the exact same words as the homography matrix *H* in the previous section. Although both are constructed from similar information, they are not the same matrix and should not be confused. An essential part of the definition of *H* is that we were considering a plane viewed by a camera and thus could relate one point in that plane to the point on the camera plane. The matrix *E* makes no such assumption and so will only be able to relate a point in one image to a line in the other.

Let's reinforce the differences between E and F. The essential matrix E is purely geometrical and knows nothing about imagers. It relates the location, in physical coordinates, of the point P as seen by the left camera to the location of the same point as seen by the right camera (i.e., it relates p_l to p_r). The fundamental matrix F relates the points on the image plane of one camera in image coordinates (pixels) to the points on the image plane of the other camera in image coordinates (for which we will use the notation q_l and q_r).

Essential matrix math

We will now submerge into some math so we can better understand the OpenCV function calls that do the hard work for our stereo geometry problems.

Given a point P, we would like to derive a relation which connects the observed locations p_l and p_r of P on the two imagers. This relationship will turn out to serve as the definition of the essential matrix. We begin by considering the relationship between p_l and p_r, the physical locations of the point we are viewing in the coordinates of the two cameras. These can be related by using epipolar geometry, as we have already seen.*

Let's pick one set of coordinates, left or right, to work in and do our calculations there. Either one is just as good, but we'll choose the coordinates centered on O_l of the left camera. In these coordinates, the location of the observed point is P_l and the origin of the other camera is located at T. The point P as seen by the right camera is P_r in that camera's coordinates, where $P_r = R(P_l - T)$. The key step is the introduction of the epipolar plane, which we already know relates all of these things. We could, of course, represent a plane any number of ways, but for our purpose it is most helpful to recall that the equation for all points **x** on a plane with normal vector **n** and passing through point **a** obey the following constraint:

$$(\mathbf{x}-\mathbf{a})\cdot\mathbf{n}=0$$

Recall that the epipolar plane contains the vectors P_l and T; thus, if we had a vector (e.g., $P_l \times T$) perpendicular to both,† then we could use that for **n** in our plane equation. Thus an equation for all possible points P_l through the point T and containing both vectors would be:‡

$$(P_l - T)^{\mathrm{T}}(T \times P_l) = 0$$

Remember that our goal was to relate q_l and q_r by first relating P_l and P_r. We draw P_r into the picture via our equality $P_r = R(P_l - T)$, which we can conveniently rewrite as $(P_l - T) = R^{-1} P_r$. Making this substitution and using that $R^{\mathrm{T}} = R^{-1}$ yields:

* Please do not confuse p_l and p_r, which are points on the projective image planes, with p_l and p_r, which are the locations of the point P in the coordinate frames of the two cameras.

† The cross product of vectors produces a third vector orthogonal to the first two. The direction is defined by the "right hand rule": if you point in the direction a and bend your middle finger in the direction b, then the cross product $a \times b$ points perpendicular to a and b in the direction of your thumb.

‡ Here we have replaced the dot product with matrix multiplication by the transpose of the normal vector.

$$(R^{\mathrm{T}}P_r)^{\mathrm{T}}(T \times P_l) = 0$$

It is always possible to rewrite a cross product as a (somewhat bulky) matrix multiplication. We thus define the matrix *S* such that:

$$T \times P_l = SP_l \quad \Rightarrow \quad S = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix}$$

This leads to our first result. Making this substitution for the cross product gives:

$$(P_r)^{\mathrm{T}} RSP_l = 0$$

This product *RS* is what we define to be the essential matrix *E*, which leads to the compact equation:

$$(P_r)^{\mathrm{T}} EP_l = 0$$

Of course, what we really wanted was a relation between the points as we observe them on the imagers, but this is just a step away. We can simply substitute using the projection equations $p_l = f_l P_l / Z_l$ and $p_r = f_r P_r / Z_r$ and then divide the whole thing by $Z_l Z_r / f_l f_r$ to obtain our final result:

$$p_r^{\mathrm{T}} Ep_l = 0$$

This might look at first like it completely specifies one of the *p*-terms if the other is given, but *E* turns out to be a rank-deficient matrix* (the 3-by-3 essential matrix has rank 2) and so this actually ends up being an equation for a line. There are five parameters in the essential matrix—three for rotation and two for the direction of translation (scale is not set)—along with two other constraints. The two additional constraints on the essential matrix are: (1) the determinant is 0 because it is rank-deficient (a 3-by-3 matrix of rank 2); and (2) its two nonzero singular values are equal because the matrix *S* is skew-symmetric and *R* is a rotation matrix. This yields a total of seven constraints. Note again that *E* contains nothing intrinsic to the cameras in *E*; thus, it relates points to each other in physical or camera coordinates, not pixel coordinates.

Fundamental matrix math

The matrix *E* contains all of the information about the geometry of the two cameras relative to one another but no information about the cameras themselves. In practice, we are usually interested in pixel coordinates. In order to find a relationship between a pixel in one image and the corresponding epipolar line in the other image, we will have

* For a square *n*-by-*n* matrix like *E*, *rank deficient* essentially means that there are fewer than *n* nonzero eigenvalues. As a result, a system of linear equations specified by a rank-deficient matrix does not have a unique solution. If the rank (number of nonzero eigenvalues) is *n* – 1 then there will be a line formed by a set of points all of which satisfy the system of equations. A system specified by a matrix of rank *n* – 2 will form a plane, and so forth.

to introduce intrinsic information about the two cameras. To do this, for *p* (the pixel coordinate) we substitute *q* and the camera intrinsics matrix that relates them. Recall that $q = Mp$ (where *M* is the camera intrinsics matrix) or, equivalently, $p = M^{-1} q$. Hence our equation for *E* becomes:

$$q_r^{\mathrm{T}} (M_r^{-1})^{\mathrm{T}} E M_l^{-1} q_l = 0$$

Though this looks like a bit of a mess, we clean it up by defining the fundamental matrix *F* as:

$$F = (M_r^{-1})^{\mathrm{T}} E M_l^{-1}$$

so that

$$q_r^{\mathrm{T}} F q_l = 0$$

In a nutshell: the fundamental matrix *F* is just like the essential matrix *E*, except that *F* operates in image pixel coordinates whereas *E* operates in physical coordinates.* Just like *E*, the fundamental matrix *F* is of rank 2. The fundamental matrix has seven parameters, two for each epipole and three for the homography that relates the two image planes (the scale aspect is missing from the usual four parameters).

How OpenCV handles all of this

We can compute *F*, in a manner analogous to computing the image homography in the previous section, by providing a number of known correspondences. In this case, we don't even have to calibrate the cameras separately because we can solve directly for *F*, which contains implicitly the fundamental matrices for both cameras. The routine that does all of this for us is called `cvFindFundamentalMat()`.

```
int cvFindFundamentalMat(
  const CvMat* points1,
  const CvMat* points2,
  CvMat*       fundamental_matrix,
  int          method            = CV_FM_RANSAC,
  double       param1            = 1.0,
  double       param2            = 0.99,
  CvMat*       status            = NULL
);
```

The first two arguments are *N*-by-2 or *N*-by-3† floating-point (single- or double-precision) matrices containing the corresponding *N* points that you have collected (they can also be *N*-by-1 multichannel matrices with two or three channels). The result is

* Note the equation that relates the fundamental matrix to the essential matrix. If we have rectified images and we normalize the points by dividing by the focal lengths, then the intrinsic matrix *M* becomes the identity matrix and $F = E$.

† You might be wondering what the *N*-by-3 or three-channel matrix is for. The algorithm will deal just fine with actual 3D points (x, y, z) measured on the calibration object. Three-dimensional points will end up being scaled to $(x/z, y/z)$, or you could enter 2D points in homogeneous coordinates $(x, y, 1)$, which will be treated in the same way. If you enter $(x, y, 0)$ then the algorithm will just ignore the 0. Using actual 3D points would be rare because usually you have only the 2D points detected on the calibration object.

fundamental_matrix, which should be a 3-by-3 matrix of the same precision as the points (in a special case the dimensions may be 9-by-3; see below).

The next argument determines the method to be used in computing the fundamental matrix from the corresponding points, and it can take one of four values. For each value there are particular restrictions on the number of points required (or allowed) in points1 and points2, as shown in Table 12-2.

Table 12-2. Restrictions on argument for method in cvFindFundamentalMat()

Value of method	Number of points	Algorithm
CV_FM_7POINT	$N = 7$	7-point algorithm
CV_FM_8POINT	$N \geq 8$	8-point algorithm
CV_FM_RANSAC	$N \geq 8$	RANSAC algorithm
CV_FM_LMEDS	$N \geq 8$	LMedS algorithm

The 7-point algorithm uses exactly seven points, and it uses the fact that the matrix F must be of rank 2 to fully constrain the matrix. The advantage of this constraint is that F is then always exactly of rank 2 and so cannot have one very small eigenvalue that is not quite 0. The disadvantage is that this constraint is not absolutely unique and so three different matrices might be returned (this is the case in which you should make fundamental_matrix a 9-by-3 matrix, so that all three returns can be accommodated). The 8-point algorithm just solves F as a linear system of equations. If more than eight points are provided then a linear least-squares error is minimized across all points. The problem with both the 7-point and 8-point algorithms is that they are extremely sensitive to outliers (even if you have many more than eight points in the 8-point algorithm). This is addressed by the RANSAC and LMedS algorithms, which are generally classified as robust methods because they have some capacity to recognize and remove outliers.*
For both methods, it is desirable to have many more than the minimal eight points.

The next two arguments are parameters used only by RANSAC and LMedS. The first, param1, is the maximum distance from a point to the epipolar line (in pixels) beyond which the point is considered an outlier. The second parameter, param2, is the desired confidence (between 0 and 1), which essentially tells the algorithm how many times to iterate.

The final argument, status, is optional; if used, it should be an N-by-1 matrix of type CV_8UC1, where N is the same as the length of points1 and points2. If this matrix is non-NULL, then RANSAC and LMedS will use it to store information about which points were ultimately considered outliers and which points were not. In particular, the appropriate

* The inner workings of RANSAC and LMedS are beyond the scope of this book, but the basic idea of RANSAC is to solve the problem many times using a random subset of the points and then take the particular solution closest to the average or the median solution. LMedS takes a subset of points, estimates a solution, then adds from the remaining points only those points that are "consistent" with that solution. You do this many times, take the set of points that fits the best, and throw away the others as "outliers". For more information, consult the original papers: Fischler and Bolles [Fischler81] for RANSAC; Rousseeuw [Rousseeuw84] for least median squares; and Inui, Kaneko, and Igarashi [Inui03] for line fitting using LMedS.

entry will be set to 0 if the point was decided to be an outlier and to 1 otherwise. For the other two methods, if this array is present then all values will be set to 1.

The return value of `cvFindFundamentalMat()` is an integer indicating the number of matrices found. It will be either 1 or 0 for all methods other than the 7-point algorithm, where it can also be 3. If the value is 0 then no matrix could be computed. The sample code from the OpenCV manual, shown in Example 12-2, makes this clear.

Example 12-2. Computing the fundamental matrix using RANSAC

```
int point_count = 100;
CvMat* points1;
CvMat* points2;
CvMat* status;
CvMat* fundamental_matrix;

points1 = cvCreateMat(1,point_count,CV_32FC2);
points2 = cvCreateMat(1,point_count,CV_32FC2);
status = cvCreateMat(1,point_count,CV_8UC1);

/* Fill the points here ... */
for( int i = 0; i < point_count; i++ )
{
    points1->data.fl[i*2]   = <x1,i>;  //These are points such as found
    points1->data.fl[i*2+1] = <y1,i>;  // on the chessboard calibration
    points2->data.fl[i*2]   = <x2,i>;  // pattern.
    points2->data.fl[i*2+1] = <y2,i>;
}

fundamental_matrix = cvCreateMat(3,3,CV_32FC1);
int fm_count = cvFindFundamentalMat( points1, points2,
                                     fundamental_matrix,
                                     CV_FM_RANSAC,1.0,0.99,status );
```

One word of warning—related to the possibility of returning 0—is that these algorithms can fail if the points supplied form *degenerate configurations*. These degenerate configurations arise when the points supplied provide less than the required amount of information, such as when one point appears more than once or when multiple points are collinear or coplanar with too many other points. It is important to always check the return value of `cvFindFundamentalMat()`.

Computing Epipolar Lines

Now that we have the fundamental matrix, we want to be able to compute epipolar lines. The OpenCV function `cvComputeCorrespondEpilines()` computes, for a list of points in one image, the epipolar lines in the other image. Recall that, for any given point in one image, there is a different corresponding epipolar line in the other image. Each computed line is encoded in the form of a vector of three points (a, b, c) such that the epipolar line is defined by the equation:

$$ax + by + c = 0$$

To compute these epipolar lines, the function requires the fundamental matrix that we computed with `cvFindFundamentalMat()`.

```
void cvComputeCorrespondEpilines(
  const CvMat* points,
  int          which_image,
  const CvMat* fundamental_matrix,
  CvMat*       correspondent_lines
);
```

Here the first argument, `points`, is the usual *N*-by-2 or *N*-by-3[*] array of points (which may be an *N*-by-1 multichannel array with two or three channels). The argument `which_image` must be either 1 or 2, and indicates which image the points are defined on (relative to the `points1` and `points2` arrays in `cvFindFundamentalMat()`), Of course, `fundamental_matrix` is the 3-by-3 matrix returned by `cvFindFundamentalMat()`. Finally, `correspondent_lines` is an *N*-by-3 array of floating-point numbers to which the result lines will be written. It is easy to see that the line equation $ax + by = c = 0$ is independent of the overall normalization of the parameters a, b, and c. By default they are normalized so that $a^2 + b^2 = 1$.

Stereo Calibration

We've built up a lot of theory and machinery behind cameras and 3D points that we can now put to use. This section will cover stereo calibration, and the next section will cover stereo rectification. *Stereo calibration* is the process of computing the geometrical relationship between the two cameras in space. In contrast, *stereo rectification* is the process of "correcting" the individual images so that they appear as if they had been taken by two cameras with row-aligned image planes (review Figures 12-4 and 12-7). With such a rectification, the optical axes (or principal rays) of the two cameras are parallel and so we say that they intersect at infinity. We could, of course, calibrate the two camera images to be in many other configurations, but here (and in OpenCV) we focus on the more common and simpler case of setting the principal rays to intersect at infinity.

Stereo calibration depends on finding the rotation matrix R and translation vector T between the two cameras, as depicted in Figure 12-9. Both R and T are calculated by the function `cvStereoCalibrate()`, which is similar to `cvCalibrateCamera2()` that we saw in Chapter 11 except that we now have two cameras and our new function can compute (or make use of any prior computation of) the camera, distortion, essential, or fundamental matrices. The other main difference between stereo and single-camera calibration is that, in `cvCalibrateCamera2()`, we ended up with a list of rotation and translation vectors between the camera and the chessboard views. In `cvStereoCalibrate()`, we seek a single rotation matrix and translation vector that relate the right camera to the left camera.

We've already shown how to compute the essential and fundamental matrices. But how do we compute R and T between the left and right cameras? For any given 3D point P in object coordinates, we can separately use single-camera calibration for the two cameras

[*] See the footnote on page 424.

to put P in the camera coordinates $P_l = R_l P + T_l$ and $P_r = R_r P + T_r$ for the left and right cameras, respectively. It is also evident from Figure 12-9 that the two views of P (from the two cameras) are related by $P_l = R^T(P_r - T)$,* where R and T are, respectively, the rotation matrix and translation vector between the cameras. Taking these three equations and solving for the rotation and translation separately yields the following simple relations:†

$$R = R_r(R_l)^{\mathrm{T}}$$

$$T = T_r - RT_l$$

Given many joint views of chessboard corners, `cvStereoCalibrate()` uses `cvCalibrateCamera2()` to solve for rotation and translation parameters of the chessboard views for each camera separately (see the discussion in the "What's under the hood?" subsection of Chapter 11 to recall how this is done). It then plugs these left and right rotation and translation solutions into the equations just displayed to solve for the rotation and translation parameters between the two cameras. Because of image noise and rounding errors, each chessboard pair results in slightly different values for R and T. The `cvStereoCalibrate()` routine then takes the median values for the R and T parameters as the initial approximation of the true solution and then runs a robust Levenberg-Marquardt iterative algorithm to find the (local) minimum of the reprojection error of the chessboard corners for both camera views, and the solution for R and T is returned. To be clear on what stereo calibration gives you: the rotation matrix will put the right camera in the same plane as the left camera; this makes the two image planes coplanar but not row-aligned (we'll see how row-alignment is accomplished in the Stereo Rectification section below).

The function `cvStereoCalibrate()` has a lot of parameters, but they are all fairly straightforward and many are the same as for `cvCalibrateCamera2()` in Chapter 11.

```
bool cvStereoCalibrate(
    const CvMat*   objectPoints,
    const CvMat*   imagePoints1,
    const CvMat*   imagePoints2,
    const CvMat*   npoints,
    CvMat*         cameraMatrix1,
    CvMat*         distCoeffs1,
    CvMat*         cameraMatrix2,
    CvMat*         distCoeffs2,
    CvSize         imageSize,
    CvMat*         R,
    CvMat*         T,
    CvMat*         E,
    CvMat*         F,
```

* Let's be careful about what these terms mean: P_l and P_r denote the locations of the 3D point P from the coordinate system of the left and right cameras respectively; R_l and T_l (resp., R_r and T_r) denote the rotation and translation vectors from the camera to the 3D point for the left (resp. right) camera; and R and T are the rotation and translation that bring the right-camera coordinate system into the left.

† The left and right cameras can be reversed in these equations either by reversing the subscripts in both equations or by reversing the subscripts and dropping the transpose of R in the translation equation only.

```
        CvTermCriteria termCrit,
        int            flags=CV_CALIB_FIX_INTRINSIC
    );
```

The first parameter, objectPoints, is an N-by-3 matrix containing the physical coordinates of each of the K points on each of the M images of the 3D object such that $N = K \times M$. When using chessboards as the 3D object, these points are located in the coordinate frame attached to the object—setting, say, the upper left corner of the chessboard as the origin (and usually choosing the Z-coordinate of the points on the chessboard plane to be 0), but any known 3D points may be used as discussed with cvCalibrateCamera2().

We now have two cameras, denoted by "1" and "2" appended to the appropriate parameter names.* Thus we have imagePoints1 and imagePoints2, which are N-by-2 matrices containing the left and right pixel coordinates (respectively) of all of the object reference points supplied in objectPoints. If you performed calibration using a chessboard for the two cameras, then imagePoints1 and imagePoints2 are just the respective returned values for the M calls to cvFindChessboardCorners() for the left and right camera views.

The argument npoints contains the number of points in each image supplied as an M-by-1 matrix.

The parameters cameraMatrix1 and cameraMatrix2 are the 3-by-3 camera matrices, and distCoeffs1 and distCoeffs2 are the 5-by-1 distortion matrices for cameras 1 and 2, respectively. Remember that, in these matrices, the first two radial parameters come first; these are followed by the two tangential parameters and finally the third radial parameter (see the discussion in Chapter 11 on distortion coefficients). The third radial distortion parameter is last because it was added later in OpenCV's development; it is mainly used for wide-angle (fish-eye) camera lenses. The use of these camera intrinsics is controlled by the flags parameter. If flags is set to CV_CALIB_FIX_INTRINSIC, then these matrices are used as is in the calibration process. If flags is set to CV_CALIB_USE_INTRINSIC_GUESS, then these matrices are used as a starting point to optimize further the intrinsic and distortion parameters for each camera and will be set to the refined values on return from cvStereoCalibrate(). You may additively combine other settings of flags that have possible values that are exactly the same as for cvCalibrateCamera2(), in which case these parameters will be computed from scratch in cvStereoCalibrate(). That is, you can compute the intrinsic, extrinsic, and stereo parameters in a single pass using cvStereoCalibrate().†

The parameter imageSize is the image size in pixels. It is used only if you are refining or computing intrinsic parameters, as when flags is not equal to CV_CALIB_FIX_INTRINSIC.

* For simplicity, think of "1" as denoting the left camera and "2" as denoting the right camera. You can interchange these as long as you consistently treat the resulting rotation and translation solutions in the opposite fashion to the text discussion. The most important thing is to physically align the cameras so that their scan lines approximately match in order to achieve good calibration results.

† Be careful: Trying to solve for too many parameters at once will sometimes cause the solution to diverge to nonsense values. Solving systems of equations is something of an art, and you must verify your results. You can see some of these considerations in the calibration and rectification code example, where we check our calibration results by using the epipolar constraint.

The terms *R* and *T* are output parameters that are filled on function return with the rotation matrix and translation vector (relating the right camera to the left camera) that we seek. The parameters *E* and *F* are optional. If they are not set to NULL, then `cvStereoCalibrate()` will calculate and fill these 3-by-3 essential and fundamental matrices. We have seen `termCrit` many times before. It sets the internal optimization either to terminate after a certain number of iterations or to stop when the computed parameters change by less than the threshold indicated in the `termCrit` structure. A typical argument for this function is `cvTermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 100, 1e-5)`.

Finally, we've already discussed the `flags` parameter somewhat. If you've calibrated both cameras and are sure of the result, then you can "hard set" the previous single-camera calibration results by using `CV_CALIB_FIX_INTRINSIC`. If you think the two cameras' initial calibrations were OK but not great, you can use it to refine the intrinsic and distortion parameters by setting `flags` to `CV_CALIB_USE_INTRINSIC_GUESS`. If the cameras have not been individually calibrated, you can use the same settings as we used for the `flags` parameter in `cvCalibrateCamera2()` in Chapter 11.

Once we have either the rotation and translation values (*R, T*) or the fundamental matrix *F*, we may use these results to rectify the two stereo images so that the epipolar lines are arranged along image rows and the scan lines are the same across both images. Although *R* and *T* don't define a unique stereo rectification, we'll see how to use these terms (together with other constraints) in the next section.

Stereo Rectification

It is easiest to compute the stereo disparity when the two image planes align exactly (as shown in Figure 12-4). Unfortunately, as discussed previously, a perfectly aligned configuration is rare with a real stereo system, since the two cameras almost never have exactly coplanar, row-aligned imaging planes. Figure 12-7 shows the goal of stereo rectification: We want to reproject the image planes of our two cameras so that they reside in the exact same plane, with image rows perfectly aligned into a frontal parallel configuration. How we choose the specific plane in which to mathematically align the cameras depends on the algorithm being used. In what follows we discuss two cases addressed by OpenCV.

We want the image rows between the two cameras to be aligned after rectification so that stereo correspondence (finding the same point in the two different camera views) will be more reliable and computationally tractable. Note that reliability and computational efficiency are both enhanced by having to search only one row for a match with a point in the other image. The result of aligning horizontal rows within a common image plane containing each image is that the epipoles themselves are then located at infinity. That is, the image of the center of projection in one image is parallel to the other image plane. But because there are an infinite number of possible frontal parallel planes to choose from, we will need to add more constraints. These include maximizing view overlap and/or minimizing distortion, choices that are made by the algorithms discussed in what follows.

The result of the process of aligning the two image planes will be eight terms, four each for the left and the right cameras. For each camera we'll get a distortion vector `distCoeffs` , a rotation matrix R_{rect} (to apply to the image), and the rectified and unrectified camera matrices (M_{rect} and M, respectively). From these terms, we can make a map, using `cvInitUndistortRectifyMap()` (to be discussed shortly), of where to interpolate pixels from the original image in order to create a new rectified image.*

There are many ways to compute our rectification terms, of which OpenCV implements two: (1) Hartley's algorithm [Hartley98], which can yield uncalibrated stereo using just the fundamental matrix; and (2) Bouguet's algorithm,† which uses the rotation and translation parameters from two calibrated cameras. Hartley's algorithm can be used to derive structure from motion recorded by a single camera but may (when stereo rectified) produce more distorted images than Bouguet's calibrated algorithm. In situations where you can employ calibration patterns—such as on a robot arm or for security camera installations—Bouguet's algorithm is the natural one to use.

Uncalibrated stereo rectification: Hartley's algorithm

Hartley's algorithm attempts to find homographies that map the epipoles to infinity while minimizing the computed disparities between the two stereo images; it does this simply by matching points between two image pairs. Thus, we bypass having to compute the camera intrinsics for the two cameras because such intrinsic information is implicitly contained in the point matches. Hence we need only compute the fundamental matrix, which can be obtained from any matched set of seven or more points between the two views of the scene via `cvFindFundamentalMat()` as already described. Alternatively, the fundamental matrix can be computed from `cvStereoCalibrate()`.

The advantage of Hartley's algorithm is that online stereo calibration can be performed simply by observing points in the scene. The disadvantage is that we have no sense of image scale. For example, if we used a chessboard for generating point matches then we would not be able to tell if the chessboard were 100 meters on each side and far away or 100 centimeters on each side and nearby. Neither do we explicitly learn the intrinsic camera matrix, without which the cameras might have different focal lengths, skewed pixels, different centers of projection, and/or different principal points. As a result, we can determine 3D object reconstruction only up to a projective transform. What this means is that different scales or projections of an object can appear the same to us (i.e., the feature points have the same 2D coordinates even though the 3D objects differ). Both of these issues are illustrated in Figure 12-10.

* Stereo rectification of an image in OpenCV is possible only when the epipole is outside of the image rectangle. Hence this rectification algorithm may not work with stereo configurations that are characterized by either a very wide baseline or when the cameras point towards each other too much.

† The Bouguet algorithm is a completion and simplification of the method first presented by Tsai [Tsai87] and Zhang [Zhang99; Zhang00]. Jean-Yves Bouguet never published this algorithm beyond its well-known implementation in his Camera Calibration Toolbox Matlab.

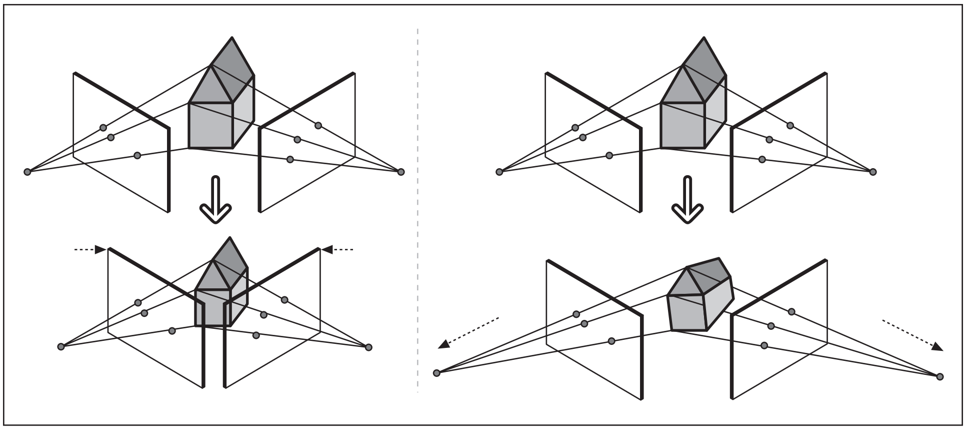


Figure 12-10. Stereo reconstruction ambiguity: if we do not know object size, then different size objects can appear the same depending on their distance from the camera (left); if we don't know the camera instrinsics, then different projections can appear the same—for example, by having different focal lengths and principal points

Assuming we have the fundamental matrix F, which required seven or more points to compute, Hartley's algorithm proceeds as follows (see Hartley's original paper [Hartley98] for more details).

1. We use the fundamental matrix to compute the two epipoles via the relations $Fe_l = 0$ and $(e_r)^T F = 0$ for the left and right epipoles, respectively.
2. We seek a first homography H_r, which will map the right epipole to the 2D homogeneous point at infinity $(1, 0, 0)^T$. Since a homography has seven constraints (scale is missing), and we use three to do the mapping to infinity, we have 4 degrees of freedom left in which to choose our H_r. These 4 degrees of freedom are mostly freedom to make a mess since most choices of H_r will result in highly distorted images. To find a good H_r, we choose a point in the image where we want minimal distortion to happen, allowing only rigid rotation and translation not shearing there. A reasonable choice for such a point is the image origin and we'll further assume that the epipole $(e_r)^T = (f, 0, 1)$ lies on the x-axis (a rotation matrix will accomplish this below). Given these coordinates, the matrix

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1/k & 0 & 1 \end{pmatrix}$$

   will take such an epipole to infinity.
3. For a selected point of interest in the right image (we chose the origin), we compute the translation T that will take that point to the image origin (0 in our case) and the rotation R that will take the epipole to $(e_r)^T = (f, 0, 1)$. The homography we want will then be $H_r = GRT$.

4. We next search for a matching homography H_l that will send the left epipole to infinity and align the rows of the two images. Sending the left epipole to infinity is easily done by using up three constraints as in step 2. To align the rows, we just use the fact that aligning the rows minimizes the total distance between all matching points between the two images. That is, we find the H_l that minimizes the total disparity in left-right matching points $\sum_i d(H_l\, p_i^l, H_r p_i^r)$. These two homographies define the stereo rectification.

Although the details of this algorithm are a bit tricky, cvStereoRectify Uncalibrated() does all the hard work for us. The function is a bit misnamed because it does not rectify uncalibrated stereo images; rather, it computes homographies that may be used for rectification. The algorithm call is

```
int cvStereoRectifyUncalibrated(
    const CvMat* points1,
    const CvMat* points2,
    const CvMat* F,
          CvSize imageSize,
          CvMat* Hl,
          CvMat* Hr,
          double threshold
);
```

In cvStereoRectifyUncalibrated(), the algorithm takes as input an array of 2-by-K corresponding points between the left and right images in the arrays points1 and points2. The fundamental matrix we calculated above is passed as the array F. We are familiar with imageSize, which just describes the width and height of the images that were used during calibration. Our return rectifying homographies are returned in the function variables Hl and Hr. Finally, if the distance from points to their corresponding epilines exceeds a set threshold, the corresponding point is eliminated by the algorithm.*

If our cameras have roughly the same parameters and are set up in an approximately horizontally aligned frontal parallel configuration, then our eventual rectified outputs from Hartley's algorithm will look very much like the calibrated case described next. If we know the size or the 3D geometry of objects in the scene, we can obtain the same results as the calibrated case.

Calibrated stereo rectification: Bouguet's algorithm

Given the rotation matrix and translation (R, T) between the stereo images, Bouguet's algorithm for stereo rectification simply attempts to minimize the amount of change reprojection produces for each of the two images (and thereby minimize the resulting reprojection distortions) while maximizing common viewing area.

To minimize image reprojection distortion, the rotation matrix R that rotates the right camera's image plane into the left camera's image plane is split in half between the two

* Hartley's algorithm works best for images that have been rectified previously by single-camera calibration. It won't work at all for images with high distortion. It is rather ironic that our "calibration-free" routine works only for undistorted image inputs whose parameters are typically derived from prior calibration. For another uncalibrated 3D approach, see Pollefeys [Pollefeys99a].

cameras; we call the two resulting rotation matrixes r_l and r_r for the left and right camera, respectively. Each camera rotates half a rotation, so their principal rays each end up parallel to the vector sum of where their original principal rays had been pointing. As we have noted, such a rotation puts the cameras into coplanar alignment but not into row alignment. To compute the R_{rect} that will take the left camera's epipole to infinity and align the epipolar lines horizontally, we create a rotation matrix by starting with the direction of the epipole e_1 itself. Taking the principal point (c_x, c_y) as the left image's origin, the (unit normalized) direction of the epipole is directly along the translation vector between the two cameras' centers of projection:

$$e_1 = \frac{T}{\|T\|}$$

The next vector, e_2, must be orthogonal to e_1 but is otherwise unconstrained. For e_2, choosing a direction orthogonal to the principal ray (which will tend to be along the image plane) is a good choice. This is accomplished by using the cross product of e_1 with the direction of the principal ray and then normalizing so that we've got another unit vector:

$$e_2 = \frac{[-T_y \; T_x \; 0]^{\mathrm{T}}}{\sqrt{T_x^2 + T_y^2}}$$

The third vector is just orthogonal to e_1 and e_2; it can be found using the cross product:

$$e_3 = e_1 \times e_2$$

Our matrix that takes the epipole in the left camera to infinity is then:

$$R_{\text{rect}} = \begin{bmatrix} (e_1)^{\mathrm{T}} \\ (e_2)^{\mathrm{T}} \\ (e_3)^{\mathrm{T}} \end{bmatrix}$$

This matrix rotates the left camera about the center of projection so that the epipolar lines become horizontal and the epipoles are at infinity. The row alignment of the two cameras is then achieved by setting:

$$R_l = R_{\text{rect}} r_l$$

$$R_r = R_{\text{rect}} r_r$$

We will also compute the rectified left and right camera matrices M_{rect_l} and M_{rect_r} but return them combined with projection matrices P_l and P_r:

$$P_l = M_{\text{rect}\_l} P_l' = \begin{bmatrix} f_{x\_l} & \alpha_l & c_{x\_l} \\ 0 & f_{y\_l} & c_{y\_l} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

and

$$P_r = M_{\text{rect}\_r} P'_r = \begin{bmatrix} f_{x\_r} & \alpha_r & c_{x\_r} \\ 0 & f_{y\_r} & c_{y\_r} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

(here α_l and α_r allow for a pixel skew factor that in modern cameras is almost always 0). The projection matrices take a 3D point in homogeneous coordinates to a 2D point in homogeneous coordinates as follows:

$$P\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

where the screen coordinates can be calculated as (*x/w, y/w*). Points in two dimensions can also then be reprojected into three dimensions given their screen coordinates and the camera intrinsics matrix. The reprojection matrix is:

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & -1/T_x & (c_x - c'_x)/T_x \end{bmatrix}$$

Here the parameters are from the left image except for c'_x, which is the principal point *x* coordinate in the right image. If the principal rays intersect at infinity, then $c_x = c'_x$ and the term in the lower right corner is 0. Given a two-dimensional homogeneous point and its associated disparity *d*, we can project the point into three dimensions using:

$$Q\begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

The 3D coordinates are then (*X/W, Y/W, Z/W*).

Applying the Bouguet rectification method just described yields our ideal stereo configuration as per Figure 12-4. New image centers and new image bounds are then chosen for the rotated images so as to maximize the overlapping viewing area. Mainly this just sets a uniform camera center and a common maximal height and width of the two image areas as the new stereo viewing planes.

```
void cvStereoRectify(
    const CvMat* cameraMatrix1,
    const CvMat* cameraMatrix2,
    const CvMat* distCoeffs1,
    const CvMat* distCoeffs2,
          CvSize imageSize,
    const CvMat* R,
    const CvMat* T,
          CvMat* Rl,
          CvMat* Rr,
          CvMat* Pl,
          CvMat* Pr,
          CvMat* Q=0,
          int    flags=CV_CALIB_ZERO_DISPARITY
);
```

For `cvStereoRectify()`,[*] we input the familiar original camera matrices and distortion vectors returned by `cvStereoCalibrate()`. These are followed by `imageSize`, the size of the chessboard images used to perform the calibration. We also pass in the rotation matrix R and translation vector T between the right and left cameras that was also returned by `cvStereoCalibrate()`.

Return parameters are `Rl` and `Rr`, the 3-by-3 row-aligned rectification rotations for the left and right image planes as derived in the preceding equations. Similarly, we get back the 3-by-4 left and right projection equations `Pl` and `Pr`. An optional return parameter is `Q`, the 4-by-4 reprojection matrix described previously.

The `flags` parameter is defaulted to set disparity at infinity, the normal case as per Figure 12-4. Unsetting `flags` means that we want the cameras verging toward each other (i.e., slightly "cross-eyed") so that zero disparity occurs at a finite distance (this might be necessary for greater depth resolution in the proximity of that particular distance).

If the `flags` parameter was not set to `CV_CALIB_ZERO_DISPARITY`, then we must be more careful about how we achieve our rectified system. Recall that we rectified our system relative to the principal points (c_x, c_y) in the left and right cameras. Thus, our measurements in Figure 12-4 must also be relative to these positions. Basically, we have to modify the distances so that $\tilde{x}^r = x^r - c_x^{\text{right}}$ and $\tilde{x}^l = x^l - c_x^{\text{left}}$. When disparity has been set to infinity, we have $c_x^{\text{left}} = c_x^{\text{right}}$ (i.e., when `CV_CALIB_ZERO_DISPARITY` is passed to `cvStereoRectify()`), and we can pass plain pixel coordinates (or disparity) to the formula for depth. But if `cvStereoRectify()` is called without `CV_CALIB_ZERO_DISPARITY` then $c_x^{\text{left}} \neq c_x^{\text{right}}$ in general. Therefore, even though the formula $Z = fT/(x^l - x^r)$ remains the same, one should keep in mind that x^l and x^r are not counted from the image center but rather from the respective principal points c_x^{left} and c_x^{right}, which could differ from x^l and x^r. Hence, if you computed disparity $d = x^l - x^r$ then it should be adjusted before computing Z: $Z\, fT/(d - (c_x^{\text{left}} - c_x^{\text{right}}))$.

Rectification map

Once we have our stereo calibration terms, we can pre-compute left and right rectification lookup maps for the left and right camera views using separate calls to `cvInitUndistort`

[*] Again, `cvStereoRectify()` is a bit of a misnomer because the function computes the terms that we can use for rectification but doesn't actually rectify the stereo images.

RectifyMap(). As with any image-to-image mapping function, a forward mapping (in which we just compute where pixels go from the source image to the destination image) will not, owing to floating-point destination locations, hit all the pixel locations in the destination image, which thus will look like Swiss cheese. So instead we work backward: for each integer pixel location in the destination image, we look up what floating-point coordinate it came from in the source image and then interpolate from its surrounding source pixels a value to use in that integer destination location. This source lookup typically uses bilinear interpolation, which we encountered with cvRemap() in Chapter 6.

The process of rectification is illustrated in Figure 12-11. As shown by the equation flow in that figure, the actual rectification process proceeds backward from (c) to (a) in a process known as reverse mapping. For each integer pixel in the rectified image (c), we find its coordinates in the undistorted image (b) and use those to look up the actual (floating-point) coordinates in the raw image (a). The floating-point coordinate pixel value is then interpolated from the nearby integer pixel locations in the original source image, and that value is used to fill in the rectified integer pixel location in the destination image (c). After the rectified image is filled in, it is typically cropped to emphasize the overlapping areas between the left and right images.

The function that implements the math depicted in Figure 12-11 is called cvInitUndistort RectifyMap(). We call this function twice, once for the left and once for the right image of stereo pair.

```
void cvInitUndistortRectifyMap(
   const CvMat* M,
   const CvMat* distCoeffs,
   const CvMat* Rrect,
   const CvMat* Mrect,
   CvArr*       mapx,
   CvArr*       mapy
);
```

The cvInitUndistortRectifyMap() function takes as input the 3-by-3 camera matrix M, the rectified 3-by-3 camera matrix Mrect, the 3-by-3 rotation matrix Rrect, and the 5-by-1 camera distortion parameters in distCoeffs.

If we calibrated our stereo cameras using cvStereoRectify(), then we can read our input to cvInitUndistortRectifyMap() straight out of cvStereoRectify() using first the left parameters to rectify the left camera and then the right parameters to rectify the right camera. For Rrect, use Rl or Rr from cvStereoRectify(); for M, use cameraMatrix1 or cameraMatrix2. For Mrect we could use the first three columns of the 3-by-4 Pl or Pr from cvStereoRectify(), but as a convenience the function allows us to pass Pl or Pr directly and it will read Mrect from them.

If, on the other hand, we used cvStereoRectifyUncalibrated() to calibrate our stereo cameras, then we must preprocess the homography a bit. Although we could—in principle and in practice—rectify stereo without using the camera intrinsics, OpenCV does not have a function for doing this directly. If we do not have Mrect from some prior calibration, the proper procedure is to set Mrect equal to M. Then, for Rrect in

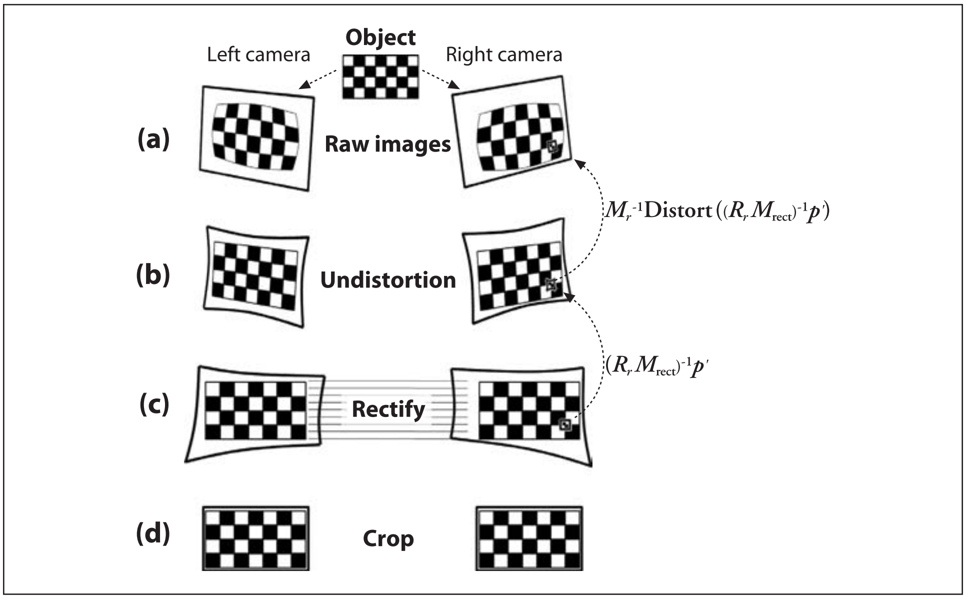

Object
Left camera
Right camera
(a)
Raw images
M_r^{-1}Distort $((R_r M_{\text{rect}})^{-1} p')$
(b)
Undistortion
$(R_r M_{\text{rect}})^{-1} p'$
(c)
Rectify
(d)
Crop

Figure 12-11. Stereo rectification: for the left and right camera, the raw image (a) is undistorted (b) and rectified (c) and finally cropped (d) to focus on overlapping areas between the two cameras; the rectification computation actually works backward from (c) to (a)

cvInitUndistortRectifyMap(), we need to compute $R_{\text{rect\_}l} = M_{\text{rect\_}l}^{-1} H_l M_l$ (or just $R_{\text{rect\_}l} = M_l^{-1} H_l M_l$ if $M_{\text{rect\_}l}^{-1}$ is unavailable) and $R_{\text{rect\_}r} = M_{\text{rect\_}r}^{-1} H_r M_r$ (or just $R_{\text{rect\_}r} = M_r^{-1} H_r M_r$ if $M_{\text{rect\_}r}^{-1}$ is unavailable) for the left and the right rectification, respectively. Finally, we will also need the distortion coefficients for each camera to fill in the 5-by-1 distCoeffs parameters.

The function cvInitUndistortRectifyMap() returns lookup maps mapx and mapy as output. These maps indicate from where we should interpolate source pixels for each pixel of the destination image; the maps can then be plugged directly into cvRemap(), a function we first saw in Chapter 6. As we mentioned, the function cvInitUndistortRectifyMap() is called separately for the left and the right cameras so that we can obtain their distinct mapx and mapy remapping parameters. The function cvRemap() may then be called, using the left and then the right maps each time we have new left and right stereo images to rectify. Figure 12-12 shows the results of stereo undistortion and rectification of a stereo pair of images. Note how feature points become horizontally aligned in the undistorted rectified images.

Stereo Correspondence

Stereo correspondence—matching a 3D point in the two different camera views—can be computed only over the visual areas in which the views of the two cameras overlap. Once again, this is one reason why you will tend to get better results if you arrange your cameras to be as nearly frontal parallel as possible (at least until you become expert at stereo vision). Then, once we know the physical coordinates of the cameras or the sizes

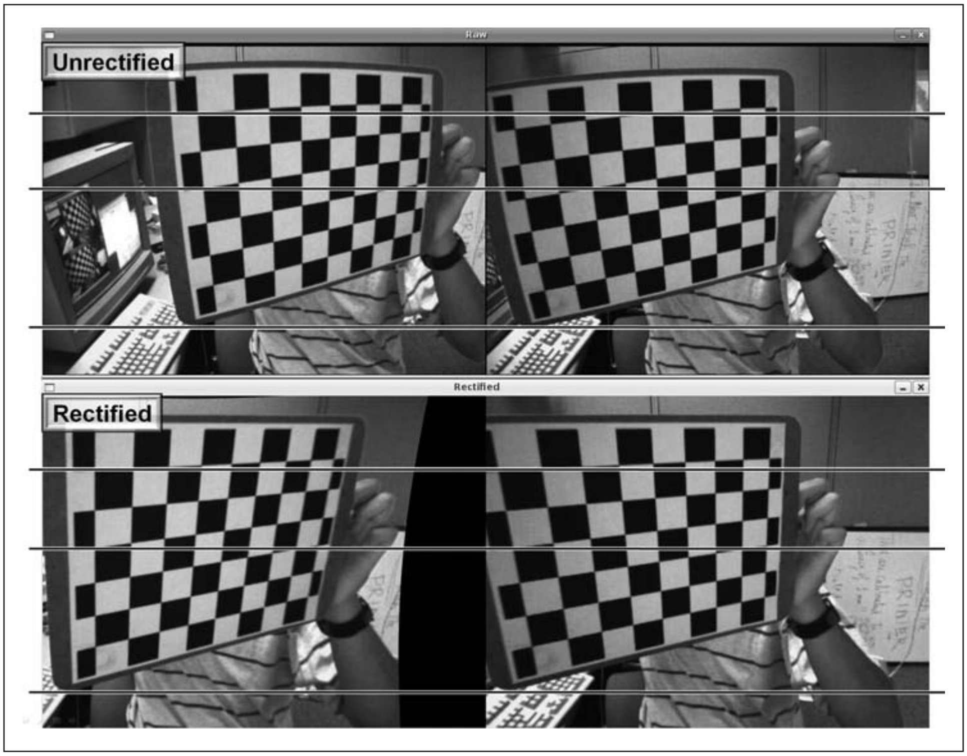


Figure 12-12. Stereo rectification: original left and right image pair (upper panels) and the stereo rectified left and right image pair (lower panels); note that the barrel distortion (in top of chessboard patterns) has been corrected and the scan lines are aligned in the rectified images

of objects in the scene, we can derive depth measurements from the triangulated disparity measures $d = x^l - x^r$ (or $d = x^l - x^r - (c_x^{\text{left}} - c_x^{\text{right}})$ if the principal rays intersect at a finite distance) between the corresponding points in the two different camera views. Without such physical information, we can compute depth only up to a scale factor. If we don't have the camera instrinsics, as when using Hartley's algorithm, we can compute point locations only up to a projective transform (review Figure 12-10).

OpenCV implements a fast and effective block-matching stereo algorithm, `cvFindStereoCorrespondenceBM()`, that is similar to the one developed by Kurt Konolige [Konolige97]; it works by using small "sum of absolute difference" (SAD) windows to find matching points between the left and right stereo rectified images.* This algorithm finds only strongly matching (high-texture) points between the two images. Thus, in a highly textured scene such as might occur outdoors in a forest, every pixel might have computed depth. In a very low-textured scene, such as an indoor hallway, very few points might register depth. There are three stages to the block-matching stereo correspondence algorithm, which works on undistorted, rectified stereo image pairs:

* This algorithm is available in an FPGA stereo hardware system from Videre (see [Videre]).

1. Prefiltering to normalize image brightness and enhance texture.
2. Correspondence search along horizontal epipolar lines using an SAD window.
3. Postfiltering to eliminate bad correspondence matches.

In the prefiltering step, the input images are normalized to reduce lighting differences and to enhance image texture. This is done by running a window—of size 5-by-5, 7-by-7 (the default), . . ., 21-by-21 (the maximum)—over the image. The center pixel I_c under the window is replaced by $\min[\max(I_c - \bar{I}, -I_{\text{cap}}), I_{\text{cap}}]$, where $\bar{I}$ is the average value in the window and I_{cap} is a positive numeric limit whose default value is 30. This method is invoked by a `CV_NORMALIZED_RESPONSE` flag. The other possible flag is `CV_LAPLACIAN_OF_GAUSSIAN`, which runs a peak detector over a smoothed version of the image.

Correspondence is computed by a sliding SAD window. For each feature in the left image, we search the corresponding row in the right image for a best match. After rectification, each row is an epipolar line, so the matching location in the right image must be along the same row (same y-coordinate) as in the left image; this matching location can be found if the feature has enough texture to be detectable and if it is not occluded in the right camera's view (see Figure 12-16). If the left feature pixel coordinate is at (x_0, y_0) then, for a horizontal frontal parallel camera arrangement, the match (if any) must be found on the same row and at, or to the left of, x_0; see Figure 12-13. For frontal parallel cameras, x_0 is at zero disparity and larger disparities are to the left. For cameras that are angled toward each other, the match may occur at negative disparities (to the right of x_0). The first parameter that controls matching search is `minDisparity`, which is where the matching search should start. The default for `minDisparity` is 0. The disparity search is then carried out over `numberOfDisparities` counted in pixels (the default is 64 pixels). Disparities have discrete, subpixel resolution that is set by the parameter `subPixelDisparities` (the default is 16 subdisparities per pixel). Reducing the number of disparities to be searched can help cut down computation time by limiting the length of a search for a matching point along an epipolar line. Remember that large disparities represent closer distances.

Setting the minimum disparity and the number of disparities to be searched establishes the *horopter*, the 3D volume that is covered by the search range of the stereo algorithm. Figure 12-14 shows disparity search limits of five pixels starting at three different disparity limits: 20, 17, and 16. Each disparity limit defines a plane at a fixed depth from the cameras (see Figure 12-15). As shown in Figure 12-14, each disparity limit—together with the number of disparities—sets a different horopter at which depth can be detected. Outside of this range, depth will not be found and will represent a "hole" in the depth map where depth is not known. Horopters can be made larger by decreasing the baseline distance T between the cameras, by making the focal length smaller, by increasing the stereo disparity search range, or by increasing the pixel width.

Correspondence within the horopter has one in-built constraint, called the *order constraint*, which simply states that the order of the features cannot change from the left view to the right. There may be *missing* features—where, owing to occlusion and noise,

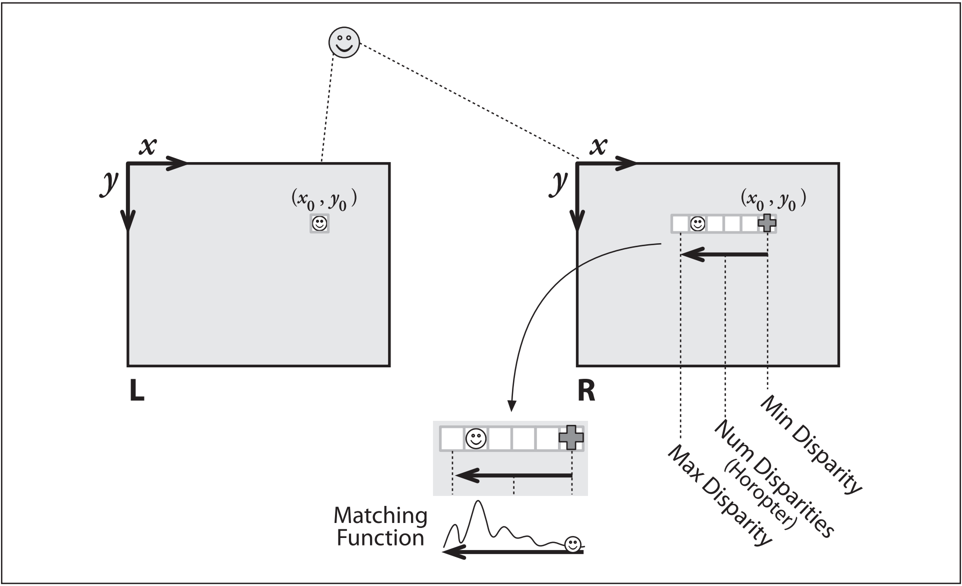


Figure 12-13. Any right-image match of a left-image feature must occur on the same row and at (or to the left of) the same coordinate point, where the match search starts at the minDisparity point (here, 0) and moves to the left for the set number of disparities; the characteristic matching function of window-based feature matching is shown in the lower part of the figure

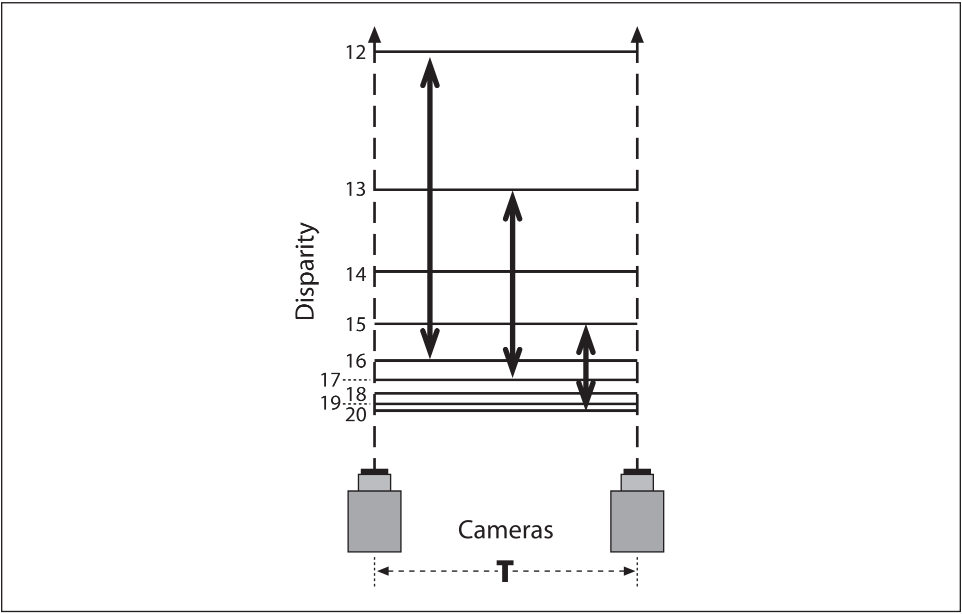


Figure 12-14. Each line represents a plane of constant disparity in integer pixels from 20 to 12; a disparity search range of five pixels will cover different horopter ranges, as shown by the vertical arrows, and different maximal disparity limits establish different horopters

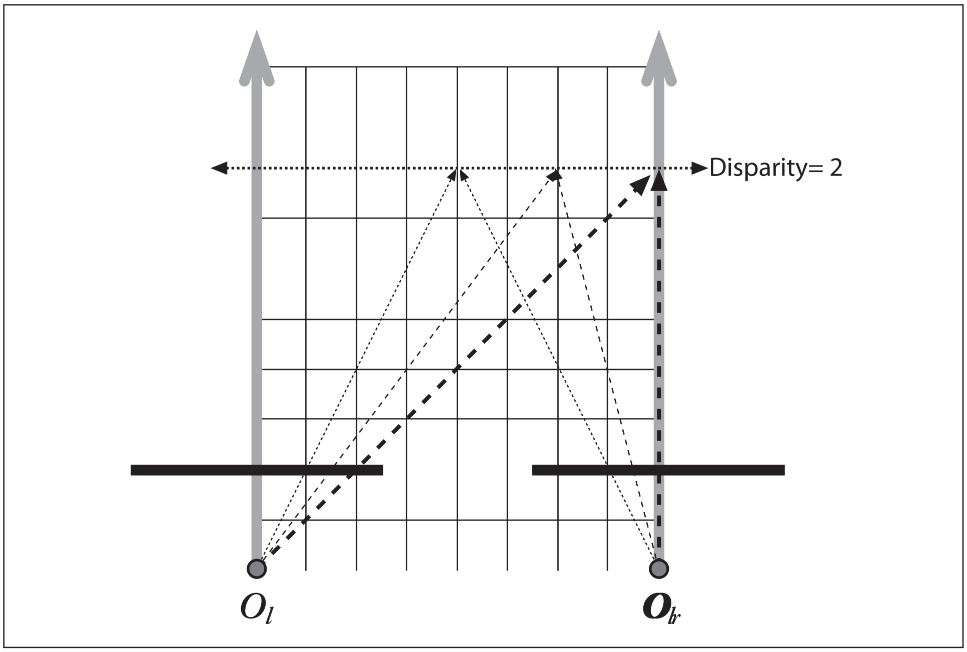


Figure 12-15. A fixed disparity forms a plane of fixed distance from the cameras

some features found on the left cannot be found on the right—but the ordering of those features that are found remains the same. Similarly, there may be many features on the right that were not identified on the left (these are called *insertions*), but insertions do not change the *order* of features although they may spread those features out. The procedure illustrated in Figure 12-16 reflects the ordering constraint when matching features on a horizontal scan line.

Given the smallest allowed disparity increment Δd, we can determine smallest achievable depth range resolution ΔZ by using the formula:

$$\Delta Z = \frac{Z^2}{fT} \Delta d$$

It is useful to keep this formula in mind so that you know what kind of depth resolution to expect from your stereo rig.

After correspondence, we turn to postfiltering. The lower part of Figure 12-13 shows a typical matching function response as a feature is "swept" from the minimum disparity out to maximum disparity. Note that matches often have the characteristic of a strong central peak surrounded by side lobes. Once we have candidate feature correspondences between the two views, postfiltering is used to prevent false matches. OpenCV makes use of the matching function pattern via a `uniquenessRatio` parameter (whose default value is 12) that filters out matches, where `uniquenessRatio > (match_val–min_match)/min_match`.

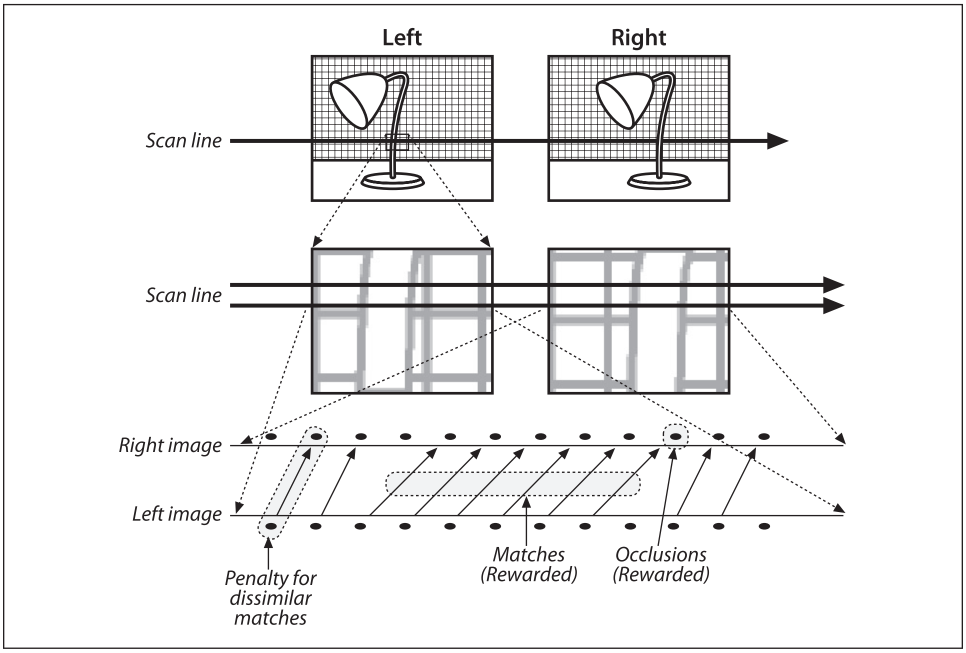


Figure 12-16. Stereo correspondence starts by assigning point matches between corresponding rows in the left and right images: left and right images of a lamp (upper panel); an enlargement of a single scan line (middle panel); visualization of the correspondences assigned (lower panel).

To make sure that there is enough texture to overcome random noise during matching, OpenCV also employs a `textureThreshold`. This is just a limit on the SAD window response such that no match is considered whose response is below the `textureThreshold` (the default value is 12). Finally, block-based matching has problems near the boundaries of objects because the matching window catches the foreground on one side and the background on the other side. This results in a local region of large and small disparities that we call *speckle*. To prevent these borderline matches, we can set a speckle detector over a speckle window (ranging in size from 5-by-5 up to 21-by-21) by setting `speckleWindowSize`, which has a default setting of 9 for a 9-by-9 window. Within the speckle window, as long as the minimum and maximum detected disparities are within `speckleRange`, the match is allowed (the default range is set to 4).

Stereo vision is becoming crucial to surveillance systems, navigation, and robotics, and such systems can have demanding real-time performance requirements. Thus, the stereo correspondence routines are designed to run fast. Therefore, we can't keep allocating all the internal scratch buffers that the correspondence routine needs each time we call `cvFindStereoCorrespondenceBM()`.

The block-matching parameters and the internal scratch buffers are kept in a data structure named `CvStereoBMState`:

```
typedef struct CvStereoBMState {
  //pre filters (normalize input images):
    int       preFilterType;
    int       preFilterSize;//for 5x5 up to 21x21
    int       preFilterCap;
  //correspondence using Sum of Absolute Difference (SAD):
    int       SADWindowSize; // Could be 5x5,7x7, ..., 21x21
    int       minDisparity;
    int       numberOfDisparities;//Number of pixels to search
  //post filters (knock out bad matches):
    int       textureThreshold; //minimum allowed
    float     uniquenessRatio;// Filter out if:
                              // [ match_val - min_match <
                              // uniqRatio*min_match ]
                              // over the corr window area
    int       speckleWindowSize;//Disparity variation window
    int       speckleRange;//Acceptable range of variation in window
   // temporary buffers
    CvMat* preFilteredImg0;
    CvMat* preFilteredImg1;
    CvMat* slidingSumBuf;
} CvStereoBMState;
```

The state structure is allocated and returned by the function `cvCreateStereoBMState()`. This function takes the parameter `preset`, which can be set to any one of the following.

`CV_STEREO_BM_BASIC`
: Sets all parameters to their default values

`CV_STEREO_BM_FISH_EYE`
: Sets parameters for dealing with wide-angle lenses

`CV_STEREO_BM_NARROW`
: Sets parameters for stereo cameras with narrow field of view

This function also takes the optional parameter `numberOfDisparities`; if nonzero, it overrides the default value from the preset. Here is the specification:

```
CvStereoBMState* cvCreateStereoBMState(
    int presetFlag=CV_STEREO_BM_BASIC,
    int numberOfDisparities=0
);
```

The state structure, `CvStereoBMState{}`, is released by calling

```
void cvReleaseBMState(
    CvStereoBMState **BMState
);
```

Any stereo correspondence parameters can be adjusted at any time between `cvFindStereoCorrespondenceBM` calls by directly assigning new values of the state structure fields. The correspondence function will take care of allocating/reallocating the internal buffers as needed.

Finally, `cvFindStereoCorrespondenceBM()` takes in rectified image pairs and outputs a disparity map given its state structure:

```
void cvFindStereoCorrespondenceBM(
    const CvArr     *leftImage,
    const CvArr     *rightImage,
    CvArr           *disparityResult,
    CvStereoBMState *BMState
);
```

Stereo Calibration, Rectification, and Correspondence Code

Let's put this all together with code in an example program that will read in a number of chessboard patterns from a file called *list.txt*. This file contains a list of alternating left and right stereo (chessboard) image pairs, which are used to calibrate the cameras and then rectify the images. Note once again that we're assuming you've arranged the cameras so that their image scan lines are roughly physically aligned and such that each camera has essentially the same field of view. This will help avoid the problem of the epipole being within the image* and will also tend to maximize the area of stereo overlap while minimizing the distortion from reprojection.

In the code (Example 12-3), we first read in the left and right image pairs, find the chessboard corners to subpixel accuracy, and set object and image points for the images where all the chessboards could be found. This process may optionally be displayed. Given this list of found points on the found good chessboard images, the code calls `cvStereoCalibrate()` to calibrate the camera. This calibration gives us the camera matrix `_M` and the distortion vector `_D` for the two cameras; it also yields the rotation matrix `_R`, the translation vector `_T`, the essential matrix `_E`, and the fundamental matrix `_F`.

Next comes a little interlude where the accuracy of calibration is assessed by checking how nearly the points in one image lie on the epipolar lines of the other image. To do this, we undistort the original points using `cvUndistortPoints()` (see Chapter 11), compute the epilines using `cvComputeCorrespondEpilines()`, and then compute the dot product of the points with the lines (in the ideal case, these dot products would all be 0). The accumulated absolute distance forms the error.

The code then optionally moves on to computing the rectification maps using the uncalibrated (Hartley) method `cvStereoRectifyUncalibrated()` or the calibrated (Bouguet) method `cvStereoRectify()`. If uncalibrated rectification is used, the code further allows for either computing the needed fundamental matrix from scratch or for just using the fundamental matrix from the stereo calibration. The rectified images are then computed using `cvRemap()`. In our example, lines are drawn across the image pairs to aid in seeing how well the rectified images are aligned. An example result is shown in Figure 12-12, where we can see that the barrel distortion in the original images is largely corrected from top to bottom and that the images are aligned by horizontal scan lines.

Finally, if we rectified the images then we initialize the block-matching state (internal allocations and parameters) using `cvCreateBMState()`. We can then compute the disparity maps by using `cvFindStereoCorrespondenceBM()`. Our code example allows you to use either horizontally aligned (left-right) or vertically aligned (top-bottom) cameras; note,

* OpenCV does not (yet) deal with the case of rectifying stereo images when the epipole is within the image frame. See, for example, Pollefeys, Koch, and Gool [Pollefeys99b] for a discussion of this case.

however, that for the vertically aligned case the function `cvFindStereoCorrespondenceBM()` can compute disparity only for the case of uncalibrated rectification unless you add code to transpose the images yourself. For horizontal camera arrangements, `cvFindStereoCorrespondenceBM()` can find disparity for calibrated or for uncalibrated rectified stereo image pairs. (See Figure 12-17 in the next section for example disparity results.)

Example 12-3. Stereo calibration, rectification, and correspondence

```
#include "cv.h"
#include "cxmisc.h"
#include "highgui.h"
#include "cvaux.h"
#include <vector>
#include <string>
#include <algorithm>
#include <stdio.h>
#include <ctype.h>

using namespace std;

//
// Given a list of chessboard images, the number of corners (nx, ny)
// on the chessboards, and a flag called useCalibrated (0 for Hartley
// or 1 for Bouguet stereo methods). Calibrate the cameras and display the
// rectified results along with the computed disparity images.
//
static void
StereoCalib(const char* imageList, int nx, int ny, int useUncalibrated)
{
    int displayCorners = 0;
    int showUndistorted = 1;
    bool isVerticalStereo = false;//OpenCV can handle left-right
                                      //or up-down camera arrangements
    const int maxScale = 1;
    const float squareSize = 1.f; //Set this to your actual square size
    FILE* f = fopen(imageList, "rt");
    int i, j, lr, nframes, n = nx*ny, N = 0;
    vector<string> imageNames[2];
    vector<CvPoint3D32f> objectPoints;
    vector<CvPoint2D32f> points[2];
    vector<int> npoints;
    vector<uchar> active[2];
    vector<CvPoint2D32f> temp(n);
    CvSize imageSize = {0,0};
    // ARRAY AND VECTOR STORAGE:
    double M1[3][3], M2[3][3], D1[5], D2[5];
    double R[3][3], T[3], E[3][3], F[3][3];
    CvMat _M1 = cvMat(3, 3, CV_64F, M1 );
    CvMat _M2 = cvMat(3, 3, CV_64F, M2 );
    CvMat _D1 = cvMat(1, 5, CV_64F, D1 );
    CvMat _D2 = cvMat(1, 5, CV_64F, D2 );
    CvMat _R = cvMat(3, 3, CV_64F, R );
    CvMat _T = cvMat(3, 1, CV_64F, T );
    CvMat _E = cvMat(3, 3, CV_64F, E );
```

Example 12-3. Stereo calibration, rectification, and correspondence (continued)

```
    CvMat _F = cvMat(3, 3, CV_64F, F );
    if( displayCorners )
        cvNamedWindow( "corners", 1 );
// READ IN THE LIST OF CHESSBOARDS:
    if( !f )
    {
        fprintf(stderr, "can not open file %s\n", imageList );
        return;
    }
    for(i=0;;i++)
    {
        char buf[1024];
        int count = 0, result=0;
        lr = i % 2;
        vector<CvPoint2D32f>& pts = points[lr];
        if( !fgets( buf, sizeof(buf)-3, f ))
            break;
        size_t len = strlen(buf);
        while( len > 0 && isspace(buf[len-1]))
            buf[--len] = '\0';
        if( buf[0] == '#')
            continue;
        IplImage* img = cvLoadImage( buf, 0 );
        if( !img )
            break;
        imageSize = cvGetSize(img);
        imageNames[lr].push_back(buf);
    //FIND CHESSBOARDS AND CORNERS THEREIN:
        for( int s = 1; s <= maxScale; s++ )
        {
            IplImage* timg = img;
            if( s > 1 )
            {
                timg = cvCreateImage(cvSize(img->width*s,img->height*s),
                    img->depth, img->nChannels );
                cvResize( img, timg, CV_INTER_CUBIC );
            }
            result = cvFindChessboardCorners( timg, cvSize(nx, ny),
                &temp[0], &count,
                CV_CALIB_CB_ADAPTIVE_THRESH |
                CV_CALIB_CB_NORMALIZE_IMAGE);
            if( timg != img )
                cvReleaseImage( &timg );
            if( result || s == maxScale )
                for( j = 0; j < count; j++ )
            {
                temp[j].x /= s;
                temp[j].y /= s;
            }
            if( result )
                break;
        }
        if( displayCorners )
```

Example 12-3. Stereo calibration, rectification, and correspondence (continued)

```
        {
            printf("%s\n", buf);
            IplImage* cimg = cvCreateImage( imageSize, 8, 3 );
            cvCvtColor( img, cimg, CV_GRAY2BGR );
            cvDrawChessboardCorners( cimg, cvSize(nx, ny), &temp[0],
                count, result );
            cvShowImage( "corners", cimg );
            cvReleaseImage( &cimg );
            if( cvWaitKey(0) == 27 ) //Allow ESC to quit
                exit(-1);
        }
        else
            putchar('.');
        N = pts.size();
        pts.resize(N + n, cvPoint2D32f(0,0));
        active[lr].push_back((uchar)result);
    //assert( result != 0 );
        if( result )
        {
         //Calibration will suffer without subpixel interpolation
            cvFindCornerSubPix( img, &temp[0], count,
                cvSize(11, 11), cvSize(-1,-1),
                cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,
                30, 0.01) );
            copy( temp.begin(), temp.end(), pts.begin() + N );
        }
        cvReleaseImage( &img );
    }
    fclose(f);
    printf("\n");
// HARVEST CHESSBOARD 3D OBJECT POINT LIST:
    nframes = active[0].size();//Number of good chessboads found
    objectPoints.resize(nframes*n);
    for( i = 0; i < ny; i++ )
        for( j = 0; j < nx; j++ )
        objectPoints[i*nx + j] =
        cvPoint3D32f(i*squareSize, j*squareSize, 0);
    for( i = 1; i < nframes; i++ )
        copy( objectPoints.begin(), objectPoints.begin() + n,
        objectPoints.begin() + i*n );
    npoints.resize(nframes,n);
    N = nframes*n;
    CvMat _objectPoints = cvMat(1, N, CV_32FC3, &objectPoints[0] );
    CvMat _imagePoints1 = cvMat(1, N, CV_32FC2, &points[0][0] );
    CvMat _imagePoints2 = cvMat(1, N, CV_32FC2, &points[1][0] );
    CvMat _npoints = cvMat(1, npoints.size(), CV_32S, &npoints[0] );
    cvSetIdentity(&_M1);
    cvSetIdentity(&_M2);
    cvZero(&_D1);
    cvZero(&_D2);

// CALIBRATE THE STEREO CAMERAS
    printf("Running stereo calibration ...");
```

Example 12-3. Stereo calibration, rectification, and correspondence (continued)

```
    fflush(stdout);
    cvStereoCalibrate( &_objectPoints, &_imagePoints1,
        &_imagePoints2, &_npoints,
        &_M1, &_D1, &_M2, &_D2,
        imageSize, &_R, &_T, &_E, &_F,
        cvTermCriteria(CV_TERMCRIT_ITER+
        CV_TERMCRIT_EPS, 100, 1e-5),
        CV_CALIB_FIX_ASPECT_RATIO +
        CV_CALIB_ZERO_TANGENT_DIST +
        CV_CALIB_SAME_FOCAL_LENGTH );
    printf(" done\n");
// CALIBRATION QUALITY CHECK
// because the output fundamental matrix implicitly
// includes all the output information,
// we can check the quality of calibration using the
// epipolar geometry constraint: m2^t*F*m1=0
    vector<CvPoint3D32f> lines[2];
    points[0].resize(N);
    points[1].resize(N);
    _imagePoints1 = cvMat(1, N, CV_32FC2, &points[0][0] );
    _imagePoints2 = cvMat(1, N, CV_32FC2, &points[1][0] );
    lines[0].resize(N);
    lines[1].resize(N);
    CvMat _L1 = cvMat(1, N, CV_32FC3, &lines[0][0]);
    CvMat _L2 = cvMat(1, N, CV_32FC3, &lines[1][0]);
//Always work in undistorted space
    cvUndistortPoints( &_imagePoints1, &_imagePoints1,
        &_M1, &_D1, 0, &_M1 );
    cvUndistortPoints( &_imagePoints2, &_imagePoints2,
        &_M2, &_D2, 0, &_M2 );
    cvComputeCorrespondEpilines( &_imagePoints1, 1, &_F, &_L1 );
    cvComputeCorrespondEpilines( &_imagePoints2, 2, &_F, &_L2 );
    double avgErr = 0;
    for( i = 0; i < N; i++ )
    {
        double err = fabs(points[0][i].x*lines[1][i].x +
            points[0][i].y*lines[1][i].y + lines[1][i].z)
            + fabs(points[1][i].x*lines[0][i].x +
            points[1][i].y*lines[0][i].y + lines[0][i].z);
        avgErr += err;
    }
    printf( "avg err = %g\n", avgErr/(nframes*n) );
//COMPUTE AND DISPLAY RECTIFICATION
    if( showUndistorted )
    {
        CvMat* mx1 = cvCreateMat( imageSize.height,
            imageSize.width, CV_32F );
        CvMat* my1 = cvCreateMat( imageSize.height,
            imageSize.width, CV_32F );
        CvMat* mx2 = cvCreateMat( imageSize.height,
            imageSize.width, CV_32F );
        CvMat* my2 = cvCreateMat( imageSize.height,
```

Example 12-3. Stereo calibration, rectification, and correspondence (continued)

```
            imageSize.width, CV_32F );
        CvMat* img1r = cvCreateMat( imageSize.height,
            imageSize.width, CV_8U );
        CvMat* img2r = cvCreateMat( imageSize.height,
            imageSize.width, CV_8U );
        CvMat* disp = cvCreateMat( imageSize.height,
            imageSize.width, CV_16S );
        CvMat* vdisp = cvCreateMat( imageSize.height,
            imageSize.width, CV_8U );
        CvMat* pair;
        double R1[3][3], R2[3][3], P1[3][4], P2[3][4];
        CvMat _R1 = cvMat(3, 3, CV_64F, R1);
        CvMat _R2 = cvMat(3, 3, CV_64F, R2);
// IF BY CALIBRATED (BOUGUET'S METHOD)
        if( useUncalibrated == 0 )
        {
            CvMat _P1 = cvMat(3, 4, CV_64F, P1);
            CvMat _P2 = cvMat(3, 4, CV_64F, P2);
            cvStereoRectify( &_M1, &_M2, &_D1, &_D2, imageSize,
                &_R, &_T,
                &_R1, &_R2, &_P1, &_P2, 0,
                0/*CV_CALIB_ZERO_DISPARITY*/ );
            isVerticalStereo = fabs(P2[1][3]) > fabs(P2[0][3]);
    //Precompute maps for cvRemap()
            cvInitUndistortRectifyMap(&_M1,&_D1,&_R1,&_P1,mx1,my1);
            cvInitUndistortRectifyMap(&_M2,&_D2,&_R2,&_P2,mx2,my2);
        }
//OR ELSE HARTLEY'S METHOD
        else if( useUncalibrated == 1 || useUncalibrated == 2 )
     // use intrinsic parameters of each camera, but
     // compute the rectification transformation directly
     // from the fundamental matrix
        {
            double H1[3][3], H2[3][3], iM[3][3];
            CvMat _H1 = cvMat(3, 3, CV_64F, H1);
            CvMat _H2 = cvMat(3, 3, CV_64F, H2);
            CvMat _iM = cvMat(3, 3, CV_64F, iM);
    //Just to show you could have independently used F
            if( useUncalibrated == 2 )
                cvFindFundamentalMat( &_imagePoints1,
                &_imagePoints2, &_F);
            cvStereoRectifyUncalibrated( &_imagePoints1,
                &_imagePoints2, &_F,
                imageSize,
                &_H1, &_H2, 3);
            cvInvert(&_M1, &_iM);
            cvMatMul(&_H1, &_M1, &_R1);
            cvMatMul(&_iM, &_R1, &_R1);
            cvInvert(&_M2, &_iM);
            cvMatMul(&_H2, &_M2, &_R2);
            cvMatMul(&_iM, &_R2, &_R2);
    //Precompute map for cvRemap()
```

Example 12-3. Stereo calibration, rectification, and correspondence (continued)

```
            cvInitUndistortRectifyMap(&_M1,&_D1,&_R1,&_M1,mx1,my1);

            cvInitUndistortRectifyMap(&_M2,&_D1,&_R2,&_M2,mx2,my2);
        }
        else
            assert(0);
        cvNamedWindow( "rectified", 1 );
// RECTIFY THE IMAGES AND FIND DISPARITY MAPS
        if( !isVerticalStereo )
            pair = cvCreateMat( imageSize.height, imageSize.width*2,
            CV_8UC3 );
        else
            pair = cvCreateMat( imageSize.height*2, imageSize.width,
            CV_8UC3 );
//Setup for finding stereo correspondences
        CvStereoBMState *BMState = cvCreateStereoBMState();
        assert(BMState != 0);
        BMState->preFilterSize=41;
        BMState->preFilterCap=31;
        BMState->SADWindowSize=41;
        BMState->minDisparity=-64;
        BMState->numberOfDisparities=128;
        BMState->textureThreshold=10;
        BMState->uniquenessRatio=15;
        for( i = 0; i < nframes; i++ )
        {
            IplImage* img1=cvLoadImage(imageNames[0][i].c_str(),0);
            IplImage* img2=cvLoadImage(imageNames[1][i].c_str(),0);
            if( img1 && img2 )
            {
                CvMat part;
                cvRemap( img1, img1r, mx1, my1 );
                cvRemap( img2, img2r, mx2, my2 );
                if( !isVerticalStereo || useUncalibrated != 0 )
                {
              // When the stereo camera is oriented vertically,
              // useUncalibrated==0 does not transpose the
              // image, so the epipolar lines in the rectified
              // images are vertical. Stereo correspondence
              // function does not support such a case.
                    cvFindStereoCorrespondenceBM( img1r, img2r, disp,
                        BMState);
                    cvNormalize( disp, vdisp, 0, 256, CV_MINMAX );
                    cvNamedWindow( "disparity" );
                    cvShowImage( "disparity", vdisp );
                }
                if( !isVerticalStereo )
                {
                    cvGetCols( pair, &part, 0, imageSize.width );
                    cvCvtColor( img1r, &part, CV_GRAY2BGR );
                    cvGetCols( pair, &part, imageSize.width,
                        imageSize.width*2 );
```

Example 12-3. Stereo calibration, rectification, and correspondence (continued)

```
                    cvCvtColor( img2r, &part, CV_GRAY2BGR );
                    for( j = 0; j < imageSize.height; j += 16 )
                        cvLine( pair, cvPoint(0,j),
                        cvPoint(imageSize.width*2,j),
                        CV_RGB(0,255,0));
                }
                else
                {
                    cvGetRows( pair, &part, 0, imageSize.height );
                    cvCvtColor( img1r, &part, CV_GRAY2BGR );
                    cvGetRows( pair, &part, imageSize.height,
                        imageSize.height*2 );
                    cvCvtColor( img2r, &part, CV_GRAY2BGR );
                    for( j = 0; j < imageSize.width; j += 16 )
                        cvLine( pair, cvPoint(j,0),
                        cvPoint(j,imageSize.height*2),
                        CV_RGB(0,255,0));
                }
                cvShowImage( "rectified", pair );
                if( cvWaitKey() == 27 )
                    break;
            }
            cvReleaseImage( &img1 );
            cvReleaseImage( &img2 );
        }
        cvReleaseStereoBMState(&BMState);
        cvReleaseMat( &mx1 );
        cvReleaseMat( &my1 );
        cvReleaseMat( &mx2 );
        cvReleaseMat( &my2 );
        cvReleaseMat( &img1r );
        cvReleaseMat( &img2r );
        cvReleaseMat( &disp );
    }
}
int main(void)
{
    StereoCalib("list.txt", 9, 6, 1);
    return 0;
}
```

Depth Maps from 3D Reprojection

Many algorithms will just use the disparity map directly—for example, to detect whether or not objects are on (stick out from) a table. But for 3D shape matching, 3D model learning, robot grasping, and so on, we need the actual 3D reconstruction or depth map. Fortunately, all the stereo machinery we've built up so far makes this easy. Recall the 4-by-4 reprojection matrix Q introduced in the section on calibrated stereo rectification. Also recall that, given the disparity d and a 2D point (x, y), we can derive the 3D depth using

$$Q\begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix}$$

where the 3D coordinates are then (X/W, Y/W, Z/W). Remarkably, Q encodes whether or not the cameras' lines of sight were converging (cross eyed) as well as the camera baseline and the principal points in both images. As a result, we need not explicitly account for converging or frontal parallel cameras and may instead simply extract depth by matrix multiplication. OpenCV has two functions that do this for us. The first, which you are already familiar with, operates on an array of points and their associated disparities. It's called `cvPerspectiveTransform`:

```
void cvPerspectiveTransform(
    const CvArr  *pointsXYD,
    CvArr* result3DPoints,
    const CvMat  *Q
);
```

The second (and new) function `cvReprojectImageTo3D()` operates on whole images:

```
void cvReprojectImageTo3D(
    CvArr *disparityImage,
    CvArr *result3DImage,
    CvArr *Q
);
```

This routine takes a single-channel `disparityImage` and transforms each pixel's (x, y) coordinates along with that pixel's disparity (i.e., a vector $[x\ y\ d]^T$) to the corresponding 3D point (X/W, Y/W, Z/W) by using the 4-by-4 reprojection matrix Q. The output is a three-channel floating-point (or a 16-bit integer) image of the same size as the input.

Of course, both functions let you pass an arbitrary perspective transformation (e.g., the canonical one) computed by `cvStereoRectify` or a superposition of that and the arbitrary 3D rotation, translation, et cetera.

The results of `cvReprojectImageTo3D()` on an image of a mug and chair are shown in Figure 12-17.

Structure from Motion

Structure from motion is an important topic in mobile robotics as well as in the analysis of more general video imagery such as might come from a handheld camcorder. The topic of structure from motion is a broad one, and a great deal of research has been done in this field. However, much can be accomplished by making one simple observation: In a static scene, an image taken by a camera that has moved is no different than an image taken by a second camera. Thus all of our intuition, as well as our mathematical and algorithmic machinery, is immediately portable to this situation. Of course, the descriptor

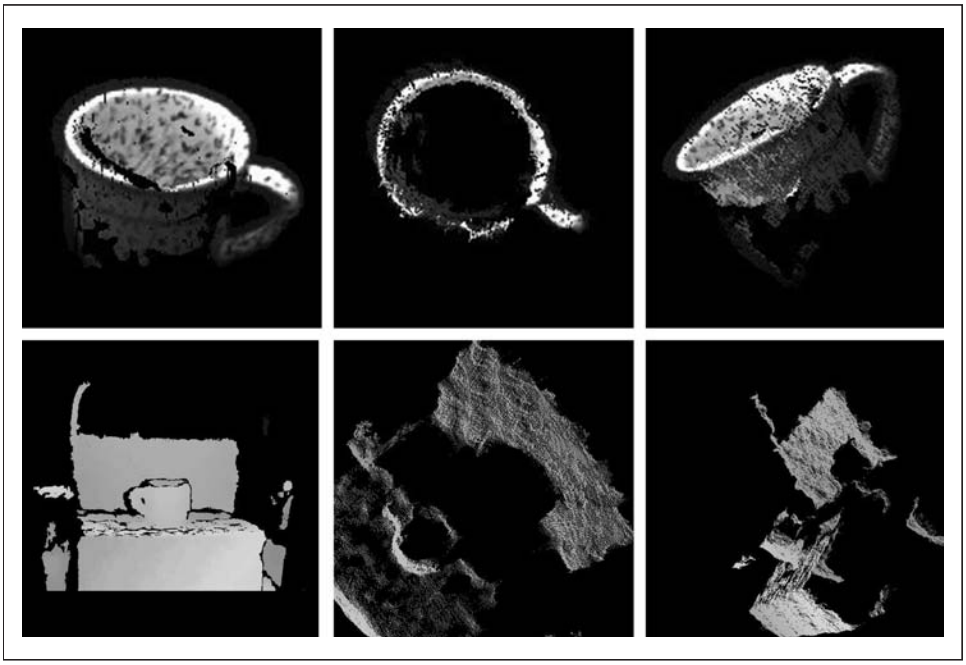


Figure 12-17. Example output of depth maps (for a mug and a chair) computed using cvFindStereo-CorrespondenceBM() and cvReprojectImageTo3D() (image courtesy of Willow Garage)

"static" is crucial, but in many practical situations the scene is either static or sufficiently static that the few moved points can be treated as outliers by robust fitting methods.

Consider the case of a camera moving through a building. If the environment is relatively rich in recognizable features, as might be found with optical flow techniques such as `cvCalcOpticalFlowPyrLK()`, then we should be able to compute correspondences between enough points—from frame to frame—to reconstruct not only the trajectory of the camera (this information is encoded in the essential matrix E, which can be computed from the fundamental matrix F and the camera intrinsics matrix M) but also, indirectly, the overall three-dimensional structure of the building and the locations of all the aforementioned features in that building. The `cvStereoRectifyUncalibrated()` routine requires only the fundamental matrix in order to compute the basic structure of a scene up to a scale factor.

Fitting Lines in Two and Three Dimensions

A final topic of interest in this chapter is that of general line fitting. This can arise for many reasons and in a many contexts. We have chosen to discuss it here because one especially frequent context in which line fitting arises is that of analyzing points in three dimensions (although the function described here can also fit lines in two dimensions). Line-fitting algorithms generally use statistically robust techniques [Inui03, Meer91, Rousseeuw87]. The OpenCV line-fitting algorithm cvFitLine() can be used whenever line fitting is needed.

```
void  cvFitLine(
  const CvArr* points,
  int          dist_type,
  double       param,
  double       reps,
  double       aeps,
  float*       line
);
```

The array points can be an N-by-2 or N-by-3 matrix of floating-point values (accommodating points in two or three dimensions), or it can be a sequence of cvPointXXX structures.* The argument dist_type indicates the distance metric that is to be minimized across all of the points (see Table 12-3).

Table 12-3. Metrics used for computing dist_type values

Value of dist_type	Metric
CV_DIST_L2	$\rho(r)=\frac{r^2}{2}$
CV_DIST_L1	$\rho(r)=r$
CV_DIST_L12	$\rho(r)=\left[\sqrt{1+\frac{r^2}{2}}-1\right]$
CV_DIST_FAIR	$\rho(r)=C^2\left[\frac{r}{C}-\log\left(1+\frac{r}{C}\right)\right],\ C=1.3998$
CV_DIST_WELSCH	$\rho(r)=\frac{C^2}{2}\left[1-\exp\left(\frac{r}{c}\right)^2\right],\ C=2.9846$
CV_DIST_HUBER	$\rho(r)=\begin{cases} r^2/2 & r<C \\ C(r-C/2) & r\geq C \end{cases}\quad C=1.345$

The parameter param is used to set the parameter C listed in Table 12-3. This can be left set to 0, in which case the listed value from the table will be selected. We'll get back to reps and aeps after describing line.

The argument line is the location at which the result is stored. If points is an N-by-2 array, then line should be a pointer to an array of four floating-point numbers (e.g., float array[4]). If points is an N-by-3 array, then line should be a pointer to an array of six floating-point numbers (e.g., float array[6]). In the former case, the return values will be (v_x, v_y, x_0, y_0), where (v_x, v_y) is a normalized vector parallel to the fitted line and (x_0, y_0)

* Here XXX is used as a placeholder for anything like 2D32f or 3D64f.

is a point on that line. Similarly, in the latter (three-dimensional) case, the return values will be (v_x, v_y, v_z, x_0, y_0, z_0), where (v_x, v_y, v_z) is a normalized vector parallel to the fitted line and (x_0, y_0, z_0) is a point on that line. Given this line representation, the estimation accuracy parameters `reps` and `aeps` are as follows: `reps` is the requested accuracy of `x0, y0[, z0]` estimates and `aeps` is the requested angular accuracy for `vx, vy[, vz]`. The OpenCV documentation recommends values of 0.01 for both accuracy values.

`cvFitLine()` can fit lines in two or three dimensions. Since line fitting in two dimensions is commonly needed and since three-dimensional techniques are of growing importance in OpenCV (see Chapter 14), we will end with a program for line fitting, shown in Example 12-4.[*] In this code we first synthesize some 2D points noisily around a line, then add some random points that have nothing to do with the line (called *outlier* points), and finally fit a line to the points and display it. The `cvFitLine()` routine is good at ignoring the outlier points; this is important in real applications, where some measurements might be corrupted by high noise, sensor failure, and so on.

Example 12-4. Two-dimensional line fitting

```
#include “cv.h”
#include “highgui.h”
#include <math.h>

int main( int argc, char** argv )
{
  IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
  CvRNG rng = cvRNG(-1);

  cvNamedWindow( “fitline”, 1 );

  for(;;) {

    char key;
    int i;
    int count    = cvRandInt(&rng)%100 + 1;
    int outliers = count/5;
    float a      = cvRandReal(&rng)*200;
    float b      = cvRandReal(&rng)*40;
    float angle  = cvRandReal(&rng)*CV_PI;
    float cos_a  = cos(angle);
    float sin_a  = sin(angle);
    CvPoint pt1, pt2;
    CvPoint* points = (CvPoint*)malloc( count * sizeof(points[0]));
    CvMat pointMat = cvMat( 1, count, CV_32SC2, points );
    float line[4];
    float d, t;

    b = MIN(a*0.3, b);

    // generate some points that are close to the line
    //
```

[*] Thanks to Vadim Pisarevsky for generating this example.

Example 12-4. Two-dimensional line fitting (continued)

```
    for( i = 0; i < count - outliers; i++ ) {
        float x = (cvRandReal(&rng)*2-1)*a;
        float y = (cvRandReal(&rng)*2-1)*b;
        points[i].x = cvRound(x*cos_a - y*sin_a + img->width/2);
        points[i].y = cvRound(x*sin_a + y*cos_a + img->height/2);
    }

    // generate “completely off” points
    //
    for( ; i < count; i++ ) {
        points[i].x = cvRandInt(&rng) % img->width;
        points[i].y = cvRandInt(&rng) % img->height;
    }

    // find the optimal line
    //
    cvFitLine( &pointMat, CV_DIST_L1, 1, 0.001, 0.001, line );
    cvZero( img );

    // draw the points
    //
    for( i = 0; i < count; i++ )
      cvCircle(
        img,
        points[i],
        2,
        (i < count – outliers) ? CV_RGB(255, 0, 0) : CV_RGB(255,255,0),
        CV_FILLED, CV_AA,
        0
      );

    // ... and the line long enough to cross the whole image
    d = sqrt((double)line[0]*line[0] + (double)line[1]*line[1]);
    line[0] /= d;
    line[1] /= d;
    t = (float)(img->width + img->height);
    pt1.x = cvRound(line[2] - line[0]*t);
    pt1.y = cvRound(line[3] - line[1]*t);
    pt2.x = cvRound(line[2] + line[0]*t);
    pt2.y = cvRound(line[3] + line[1]*t);
    cvLine( img, pt1, pt2, CV_RGB(0,255,0), 3, CV_AA, 0 );

    cvShowImage( “fitline”, img );

    key = (char) cvWaitKey(0);
    if( key == 27 || key == ‘q’ || key == ‘Q’ ) // ‘ESC’
        break;
    free( points );
  }

  cvDestroyWindow( “fitline” );
  return 0;
}
```

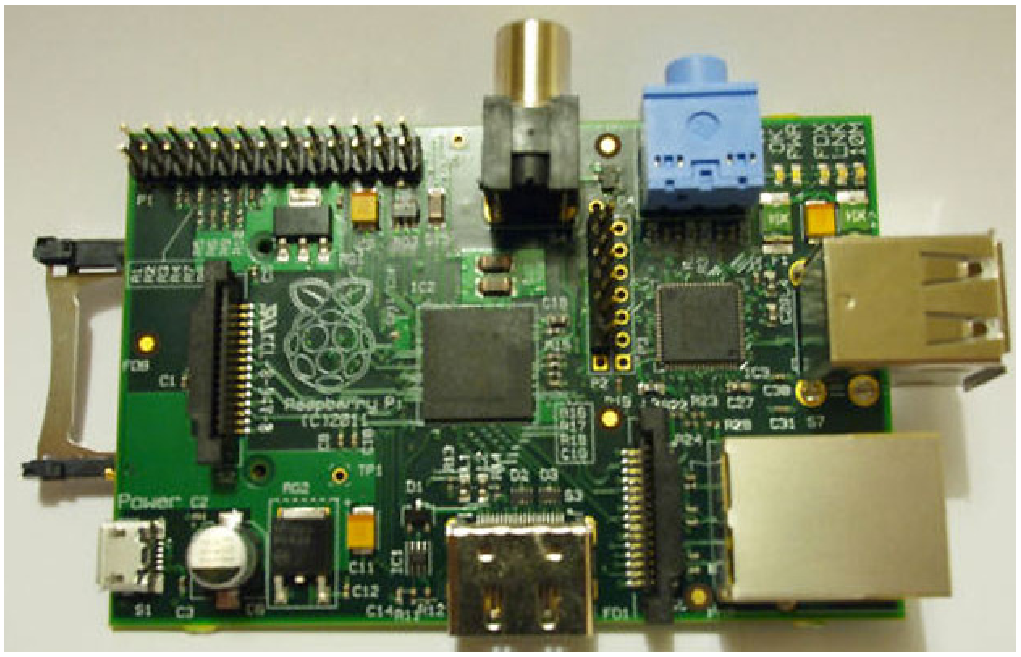
Exercises

1. Calibrate a camera using `cvCalibrateCamera2()` and at least 15 images of chessboards. Then use `cvProjectPoints2()` to project an arrow orthogonal to the chessboards (the surface normal) into each of the chessboard images using the rotation and translation vectors from the camera calibration.
2. *Three-dimensional joystick*. Use a simple known object with at least four measured, non-coplanar, trackable feature points as input into the POSIT algorithm. Use the object as a 3D joystick to move a little stick figure in the image.
3. In the text's bird's-eye view example, with a camera above the plane looking out horizontally along the plane, we saw that the homography of the ground plane had a horizon line beyond which the homography wasn't valid. How can an infinite plane have a horizon? Why doesn't it just appear to go on forever?

   Hint: Draw lines to an equally spaced series of points on the plane going out away from the camera. How does the angle from the camera to each next point on the plane change from the angle to the point before?
4. Implement a bird's-eye view in a video camera looking at the ground plane. Run it in real time and explore what happens as you move objects around in the normal image versus the bird's-eye view image.
5. Set up two cameras or a single camera that you move between taking two images.
   a. Compute, store, and examine the fundamental matrix.
   b. Repeat the calculation of the fundamental matrix several times. How stable is the computation?
6. If you had a calibrated stereo camera and were tracking moving points in both cameras, can you think of a way of using the fundamental matrix to find tracking errors?
7. Compute and draw epipolar lines on two cameras set up to do stereo.
8. Set up two video cameras, implement stereo rectification and experiment with depth accuracy.
   a. What happens when you bring a mirror into the scene?
   b. Vary the amount of texture in the scene and report the results.
   c. Try different disparity methods and report on the results.
9. Set up stereo cameras and wear something that is textured over one of your arms. Fit a line to your arm using all the `dist_type` methods. Compare the accuracy and reliability of the different methods.

Quick Start Guide

The Raspberry Pi – Single Board Computer



Source: Raspberry Pi & Wiki

Chapter 1: RPi Hardware Basic Setup

Typical Hardware You Will Need

While the RPi can be used without any additional hardware (except perhaps a power supply of some kind), it won't be much use as a general computer. As with any normal PC, it is likely you will need some additional hardware.

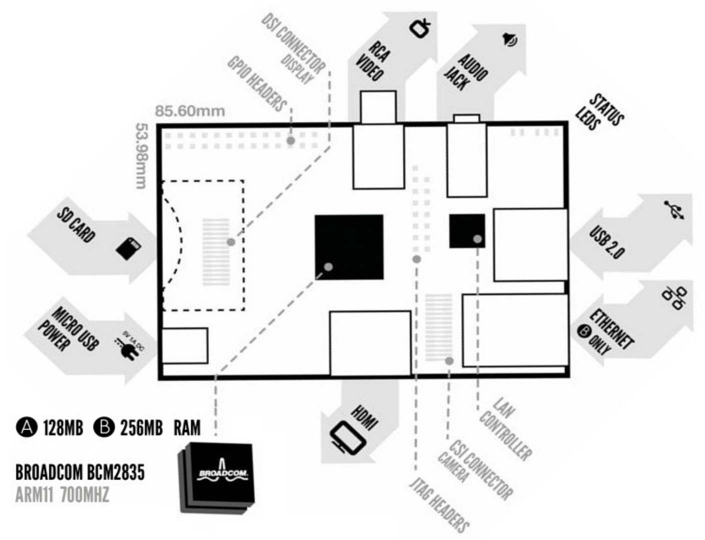
The following are more or less essential:

- Raspberry Pi board
- Prepared Operating System SD Card
- USB keyboard
- Display (with HDMI, DVI, Composite or SCART input)
- Power Supply
- Cables

Highly suggested extras include:

- USB mouse
- Internet connectivity - a USB WiFi adaptor (Model A/B) or a LAN cable (Model B)
- Powered USB Hub
- Case

Connecting Together



You can use the diagram to connect everything together, or use the following instructions:

1. Plug the preloaded SD Card into the Pi.
2. Plug the USB keyboard and mouse into the Pi, perhaps via a USB Hub. Connect the Hub to power, if necessary.

3. Plug the video cable into the screen (TV) and into the Pi.
4. Plug your extras into the Pi (USB WiFi, Ethernet cable, hard drive etc.). This is where you may really need a USB Hub.
5. Ensure that your USB Hub (if any) and screen are working.
6. Plug the power source into the main socket.
7. With your screen on, plug the other end of the power source into the Pi.
8. The Pi should boot up and display messages on the screen.

It is always recommended to connect the MicroUSB Power to the unit last (while most connections can be made live, it is best practice to connect items such as displays/h/w pin connections with the power turned off).

The RPi may take a long time to boot when powered-on for the first time, so be patient!

Prepared Operating System SD Card

As the RPi has no internal storage or built-in operating system it requires an SD-Card that is set up to boot the RPi.

- You can create your own preloaded card using any suitable SD card you have. Be sure to backup any existing data on the card.
- Preloaded SD cards will be available from the RPi Shop.

This guide will assume you have a preloaded SD card.

Keyboard & Mouse

Most standard USB keyboards and mice will work with the RPi. Wireless keyboard/mice should also function, and only require a single USB port for an RF dongle. In order to use a Bluetooth keyboard or mouse you would need to use a Bluetooth dongle, which again uses a single port.

Remember that the Model A has a single USB port and the Model B only has two (typically a keyboard and mouse will use a USB port each).

Display

There are two main connection options for the RPi display, *HDMI* (high definition) and *Composite* (low definition).

- HD TVs and most LCD Monitors can be connected using a full-size 'male' HDMI cable, and with an inexpensive adaptor if DVI is used. HDMI versions 1.3 and 1.4 are supported, and a version 1.4 cable is recommended. The RPi outputs audio and video via HMDI, but does not support HDMI input.
- Older TVs can be connected using Composite (a yellow-to-yellow cable) or via SCART (using a Composite to SCART adaptor). PAL and NTSC TVs are supported. When using composite video, audio is available from a 3.5mm (1/8 inch) socket, and can be sent to your TV, to headphones, or to an amplifier. To send audio your TV,

you will need a cable which adapts from 3.5mm to double (red and white) RCA connectors.

Note: There is no VGA output available, so older VGA monitors will require an expensive adaptor.

Using an HDMI to DVI-D (digital) adaptor plus a DVI to VGA adaptor will not work. HDMI does not supply the DVI-A (analogue) needed to convert to VGA - converting an HDMI or DVI-D source to VGA (or component) needs an active converter. (It can work out cheaper to buy a new monitor.) The lack of VGA has been acknowledged as a priority issue.

Power Supply

The unit uses a Micro USB connection to power itself (only the power pins are connected - so it will not transfer data over this connection). A standard modern phone charger with a micro-USB connector will do, but needs to produce at least 700mA at 5 volts. Check your power supply's ratings carefully. Suitable mains adaptors will be available from the RPi Shop and are recommended if you are unsure what to use.

You can use a range of other power sources (assuming they are able to provide enough current ~700mA):

- Computer USB Port or powered USB hub (will depend on power output)
- Special wall warts with USB ports
- Mobile Phone Backup Battery (will depend on power output) (in theory - needs confirmation)

To use the above, you'll need a USB A 'male' to USB micro 'male' cable - these are often shipped as data cables with MP3 players.

Cables

You will probably need a number of cables in order to connect your RPi up.

1. Micro-B USB Power Cable
2. HDMI-A or Composite cable, plus DVI adaptor or SCART adaptor if required, to connect your RPi to the Display/Monitor/TV of your choice.
3. Audio cable, this is not needed if you use a HDMI TV/monitor.
4. Ethernet/LAN Cable

Additional Peripherals

You may decide you want to use various other devices with your RPi, such as Flash Drives/Portable Hard Drives, Speakers etc.

Internet Connectivity

This may be an Ethernet/LAN cable (standard RJ45 connector) or a USB WiFi adaptor. The RPi ethernet port is auto-sensing which means that it may be connected to a router or directly to another computer (without the need for a crossover cable).

USB-Hub

In order to connect additional devices to the RPi, you may want to obtain a USB Hub, which will allow multiple devices to be used.

It is recommended that a **powered** hub is used - this will provide any additional power to the devices without affecting the RPi itself.

USB version 2.0 is recommended. USB version 1.1 is fine for keyboards and mice, but may not be fast enough for other accessories.

Case

Since the RPi is supplied without a case, it will be important to ensure that you do not use it in places where it will come into contact with conductive metal or liquids, unless suitably protected.

Expansion & Low Level Peripherals

If you plan on making use of the low level interfaces available on the RPi, then ensure you have suitable header pins for the GPIO (and if required JTAG) suitable for your needs.

Also if you have a particular low-level project in mind, then ensure you design in suitable protection circuits to keep your RPi safe.

Chapter 2: RPi Advanced Setup

Finding hardware and setting up

You'll need a preloaded SD card, USB keyboard, TV/Monitor (with HDMI/ DVI/ Composite/ SCART input), and power supply (USB charger or a USB port from a powered USB Hub or another computer).

You'll likely also want a USB mouse, a case, and a USB Hub (a necessity for Model A). A powered USB Hub will reduce the demand on the RPi. To connect to the Internet, you'll need either an Ethernet/LAN cable (Model B) or a USB WiFi adaptor (either model).

When setting up, it is advisable to connect the power after everything else is ready.

Serial connection

The Serial Port is a simple and uncomplicated method to connect to the Raspberry Pi. The communication depends on byte wise data transmission, is easy to setup and is generally available even before boot time.

First interaction with the board

Connect the serial cable to the COM port in the Raspberry Pi, and connect the other end to the COM port or USB Serial Adapter in the computer.

Serial Parameters

The following parameters are needed to connect to the Raspberry. All parameters except **Port_Name** and **Speed** are default values and may not need to be set.

- **Port_Name**: Linux automatically assigns different names for different types of serial connectors. Choose your option:
  - Standard Serial Port: ttyS0 ... ttySn
  - USB Serial Port Adapter: ttyUSB0 ... ttyUSBn
- **Speed**: 115200
- Bits: 8
- Parity: None
- Stop Bits: 1
- Flow Control: None

The Serial Port is generally usable by the users in the group **dialout**. To add oneself to the group **dialout** the the following command needs to be executed with **root** privileges:

```
$useradd -G {dialout} your_name
```

- **Super Easy Way Using GNU Screen**

Enter the command below into a terminal window

```
screen Port_Name 115200
```

- **Super Easy Way Using Minicom**

Run minicom with the following parameters:

```
minicom -b 115200 -o -D Port_Name
```

- **GUI method with GtkTerm**

Start *GtkTerm*, select Configuration->Port and enter the values above in the labelled fields.

- **Windows Users**

Windows Users above Windows XP must download putty or a comparable terminal program. Users of XP and below can choose between using *putty* and *Hyperterminal.*

First Dialog

If you get the prompt below, you are connected to the Raspberry Pi shell!

```
prompt> #
```

First command you might want try is "help":

```
prompt> # help
```

If you get some output, you are correctly connected to the Raspberry Pi! Congratulations!

SD card setup

Now we want to use an SD card to install some GNU/Linux distro in it and get more space for our stuff. You can use either an SD or SDHC card. In the latter case of course take care that your PC card reader also supports SDHC. Be aware that you are not dealing with an x86 processor, but instead a completely different architecture called ARM, so don't forget to install the ARM port for the distro you are planning to use.

Formatting the SD card via the mkcard.txt script

1. Download **mkcard.txt** .
2. $ chmod +x mkcard.txt

3. $ ./mkcard.txt /dev/sd*x*, where *x* is the letter of the card. You can find this by inserting your card and then running dmesg | tail. You should see the messages about the device being mounted in the log. Mine mounts as **sdc**.

Once run, your card should be formatted.

Formatting the SD card via fdisk "Expert mode"

First, lets clear the partition table:

```
==========================================================================
=====
$ sudo fdisk /dev/sdb

Command (m for help): o
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that, of course, the previous
content won't be recoverable.

Warning: invalid flag 0x0000 of partition table 4 will be corrected by
w(rite)
==========================================================================
=====
```

Print card info:

```
==========================================================================
=====
Command (m for help): p

Disk /dev/sdb: 128 MB, 128450560 bytes
....
==========================================================================
=====
```

Note card size in bytes. Needed later below.

Then go into "Expert mode":

```
==========================================================================
=====
Command (m for help): x
==========================================================================
=====
```

Now we want to set the geometry to 255 heads, 63 sectors and calculate the number of cylinders required for the particular SD/MMC card:

```
===========================================================================
=====
Expert command (m for help): h
Number of heads (1-256, default 4): 255

Expert command (m for help): s
Number of sectors (1-63, default 62): 63
Warning: setting sector offset for DOS compatiblity
===========================================================================
=====
```

NOTE: Be especially careful in the next step. First calculate the number of cylinders as follows:

- B = Card size in bytes (you got it before, in the second step when you printed the info out)
- C = Number of cylinders

```
C=B/255/63/512
```

When you get the number, you round it DOWN. Thus, if you got 108.8 you'll be using 108 cylinders.

```
===========================================================================
=====
Expert command (m for help): c
Number of cylinders (1-1048576, default 1011): 15
===========================================================================
=====
```

In this case 128MB card is used (reported as 128450560 bytes by fdisk above), thus 128450560 / 255 / 63 / 512 = 15.6 rounded down to 15 cylinders. Numbers there are 255 heads, 63 sectors, 512 bytes per sector.

So far so good, now we want to create two partitions. One for the boot image, one for our distro. Create the FAT32 partition for booting and transferring files from Windows. Mark it as bootable.

```
===========================================================================
=====
Expert command (m for help): r
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
```

```
First cylinder (1-245, default 1): (press Enter)
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-245, default 245): +50

Command (m for help): t
Selected partition 1
Hex code (type L to list codes): c
Changed system type of partition 1 to c (W95 FAT32 (LBA))

Command (m for help): a
Partition number (1-4): 1
============================================================================
=====
```

Create the Linux partition for the root file system.

```
============================================================================
=====
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 2
First cylinder (52-245, default 52): (press Enter)
Using default value 52
Last  cylinder  or  +size  or  +sizeM  or  +sizeK  (52-245,  default  245):(press
Enter)
Using default value 245
============================================================================
=====
```

Print and save the new partition records.

```
============================================================================
=====
Command (m for help): p

Disk /dev/sdc: 2021 MB, 2021654528 bytes
255 heads, 63 sectors/track, 245 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

  Device Boot       Start          End       Blocks    Id   System
/dev/sdc1    *           1           51       409626     c   W95 FAT32 (LBA)
/dev/sdc2               52          245      1558305    83   Linux

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
```

```
WARNING: Re-reading the partition table failed with error 16: Device or
resource busy. The kernel still uses the old table. The new table will be
used at the next reboot.

WARNING: If you have created or modified any DOS 6.x partitions, please see
the fdisk manual page for additional information.
Syncing disks.
===========================================================================
=====
```

Now we've got both partitions, next step is formatting them.

NOTE: If the partitions (/dev/sdc1 and /dev/sdc2) does not exist, you should unplug the card and plug it back in. Linux will now be able to detect the new partitions.

```
===========================================================================
=====
$ sudo mkfs.msdos -F 32 /dev/sdc1 -n LABEL
mkfs.msdos 2.11 (12 Mar 2005)

$ sudo mkfs.ext3 /dev/sdc2
mke2fs 1.40-WIP (14-Nov-2006)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
195072 inodes, 389576 blocks
19478 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=402653184
12 block groups
32768 blocks per group, 32768 fragments per group
16256 inodes per group
Superblock backups stored on blocks:
       32768, 98304, 163840, 229376, 294912

Writing inode tables: done
Creating journal (8192 blocks): done
Writing superblocks and filesystem accounting information:
===========================================================================
=====
```

All done!

NOTE: For convenience, you can add the -L option to the mkfs.ext3 command to assign a volume label to the new ext3 filesystem. If you do that, the new (automatic) mount point under /media when you insert that SD card into some Linux hosts will be based on that label. If there's no label, the new mount point will most likely be a long hex string, so assigning a label makes manual mounting on the host more convenient.

Setting up the boot partition

The boot partition must contain:

- bootcode.bin : 2nd stage bootloader, starts with SDRAM disabled
- loader.bin : 3rd stage bootloader, starts with SDRAM enabled
- start.elf: The GPU binary firmware image, provided by the foundation.
- kernel.img: The OS kernel to load on the ARM processor. Normally this is Linux - see instructions for compiling a kernel.
- cmdline.txt: Parameters passed to the kernel on boot.

Optional files:

- config.txt: A configuration file read by the GPU. Use this to override set the video mode, alter system clock speeds, voltages, etc.
- vlls directory: Additional GPU code, e.g. extra codec's. Not present in the initial release.

Additional files supplied by the foundation

These files are also present on the SD cards supplied by the foundation.

Additional kernels. Rename over kernel.img to use them (ensure you have a backup of the original kernel.img first!):

- kernel_emergency.img : kernel with busybox rootfs. You can use this to repair the main linux partition using e2fsck if the linux partition gets corrupted.

Additional GPU firmware images, rename over start.elf to use them:

- arm128_start.elf : 128M ARM, 128M GPU split (use this for heavy 3D work, possibly also required for some video decoding)
- arm192_start.elf : 192M ARM, 64M GPU split (this is the default)
- arm224_start.elf : 224M ARM, 32M GPU split (use this for Linux only with no 3D or video processing. It's enough for the 1080p frame buffer, but not much else)

Writing the image into the SDcard and finally booting GNU/Linux

The easiest way to do this is to use PiCard. It even saves you from some hassles explained above. You will need your SD card + reader and a Linux pc to use PiCard. After that, just plug the card into your Rpi.

Setting up the boot args

Wire up your Raspberry Pi and power it up

As explained in Chapter 1

SD Card Cloning/Backup

```
Note: Update these instructions if required once they've been tried
```

From windows you can copy the full SD-Card by using Win32DiskImager. Alternatively, you can use the following instructions;

```
Note:
Many built-in SD card readers do not work, so if you have problems
use an external SD-USB adapter for this.
```

Required Software Setup

- download a windows utility dd.exe from *http://www.chrysocome.net/dd*
- rename it windd.exe

(This executable can to write to your harddisk so exercise caution using it!)

- make a copy named dd-removable.exe

(That executable refuses to write to your hard disk as it is named dd-removable As long as you use dd-removable.exe you cannot lose your hard disk)

- Connect an SD card to the computer
- run "dd-removable –list"

Should give something like this:

```
rawwrite dd for windows version 0.6beta3.
Written by John Newbigin <jn@it.swin.edu.au>
This program is covered by terms of the GPL Version 2.

NT Block Device Objects
\\?\Device\Harddisk1\Partition0
link to \\?\Device\Harddisk1\DR8
Removable media other than floppy. Block size = 512
size is 4075290624 bytes
```

This "\\?\Device\Harddisk1\Partition0" is the part you need.

Reading an image from the SD Card

BEWARE: DO THIS WRONG AND YOU CAN LOSE YOUR HARDDISK!!!

Obviously, you can NOT use 'dd-removable' to read an image as that executable refuses to write to your hard disk (so extra care is required here as you use 'windd').

- To **read** an SD-card image from the SD-card use:

```
windd bs=1M if=\\?\Device\Harddisk1\Partition0 of=THE_IMAGE_READ -size
Your disk name ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Copying an image to the SD Card

BEWARE: DO THIS WRONG AND YOU CAN LOSE YOUR HARDDISK!!!

- To **copy** an image named "THEIMAGE" to the SD-card do this:

```
dd-removable bs=1M if=THEIMAGE of=\\?\Device\Harddisk1\Partition0
                   Your disk name ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Software Development/Proving

A supported platform for the Raspberry is Qt , which is already being worked on. C/C++ is supported through a gcc cross-compiling tool chain.

After compiling, using QEMU and a Linux VM would be one way of testing your apps. This also works on Windows. Search the forum for the readymade ARM images. The choice of programming languages, IDEs and other tools ON the R-Pi is only determined by:

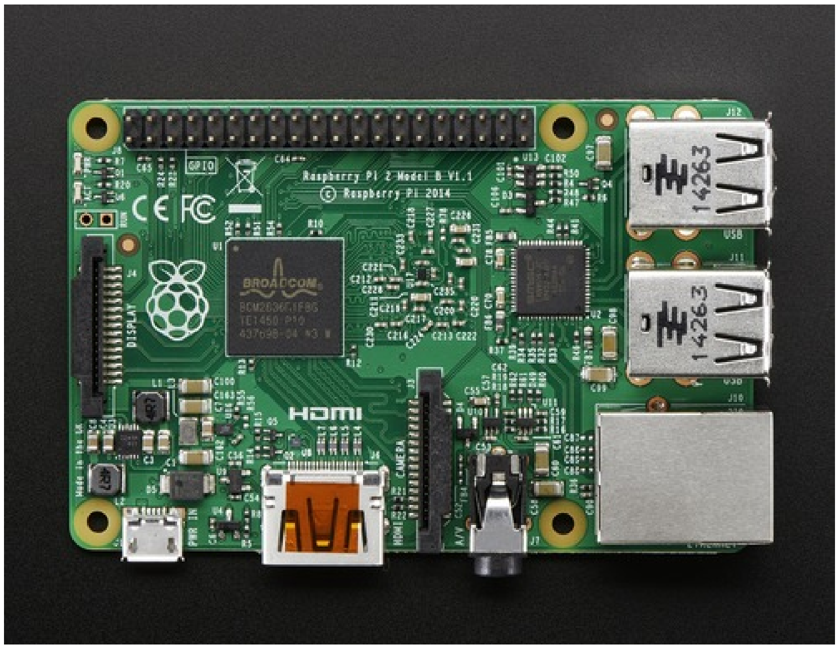
- The operating system compatibility (at the moment the specific Linux distro used)
- The status of the respective ARM package repositories and their binary compatibility
- The possibility to build other software + its dependencies for the R-Pi from sources.

For more guides and projects involving the Raspberry Pi, see RPi Projects (http://elinux.org/RPi_Projects).

adafruit learning system

Introducing the Raspberry Pi 2 - Model B

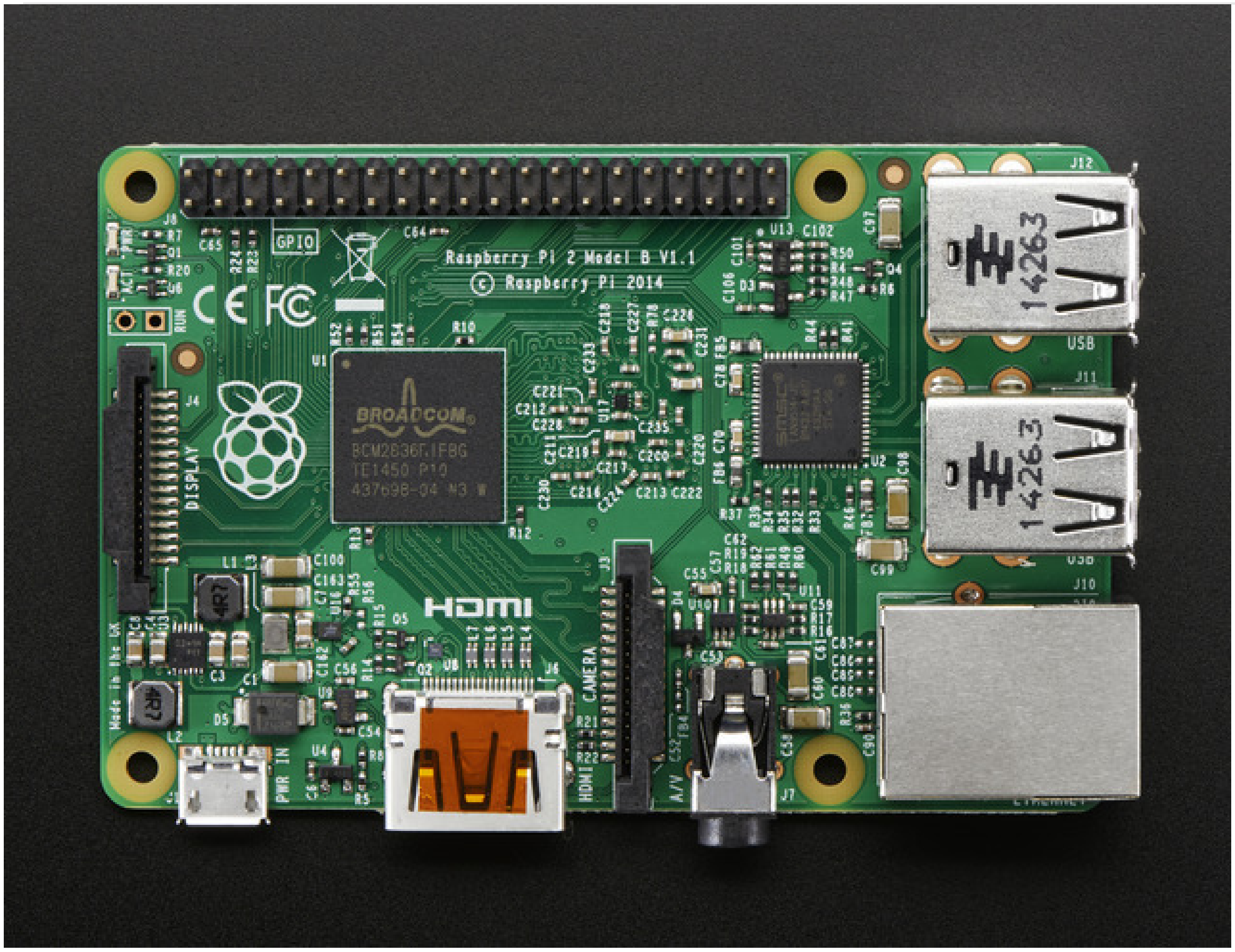
Created by lady ada



Last updated on 2015-02-09 03:00:41 PM EST

Guide Contents

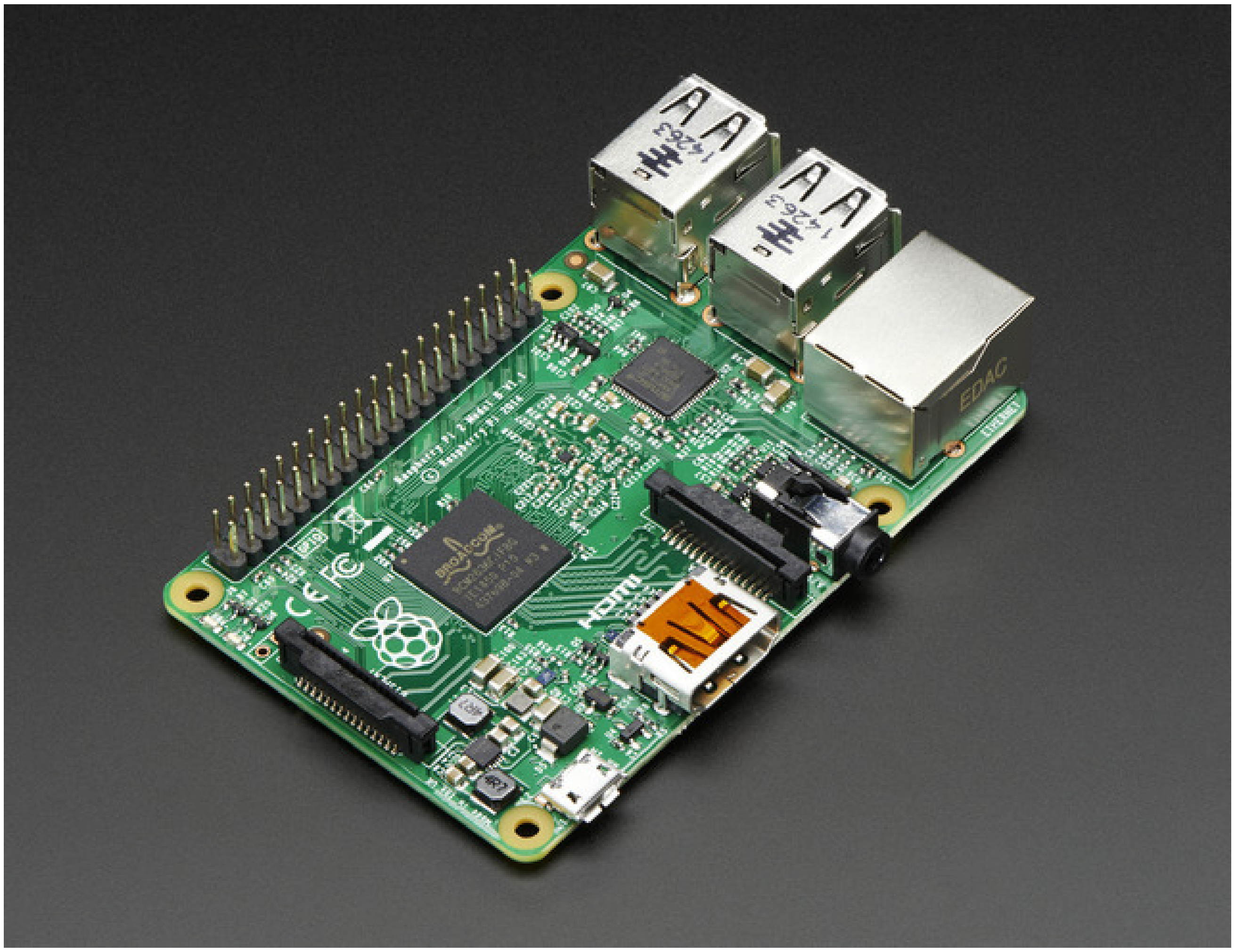
Overview



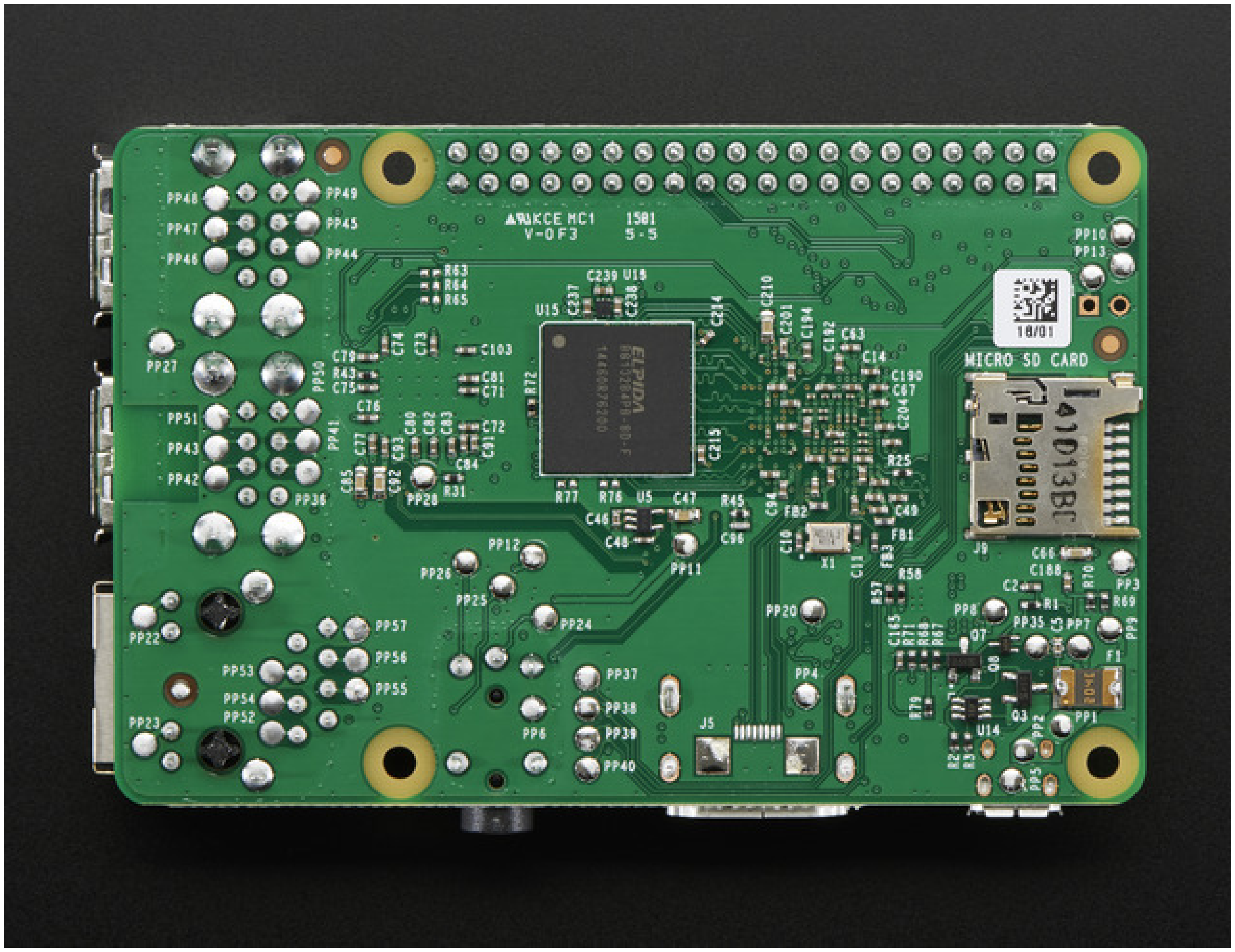
Didn't think the Raspberry Pi could get any better? You're in for a big surprise! The Raspberry Pi 2 Model B is out and it's amazing! With an upgraded ARMv7 multicore processor, and a full Gigabyte of RAM, this pocket computer has moved from being a 'toy computer' to a real desktop PC

The big upgrade is a move from the BCM2835 (single core ARMv6) to BCM2836 (quad core ARMv7). The upgrade in processor types means you will see ~2x performance increase just on processor-upgrade only. For software that can take advantage of multiple-core processors, you can expect 4x performance on average and for really multi-thread-friendly code, up to 7.5x increase in speed!

That's not even taking into account the 1 Gig of RAM, which will greatly improve games and web-browser performance!



Best of all, the Pi 2 keeps the same shape, connectors and mounting holes as the Raspberry Pi B+. That means that all of your HATs and other plug-in daughterboards will work just fine. 99% of cases and accessories will be fully compatible with both versions



Please note: The new processor on the Pi 2 means that you will need to update your existing SD card or create a new SD card with your operating system (Raspbian, Arch, XBMC, NooBs, etc) because you cannot plug in olderscards from a Pi 1 into a Pi 2 without upgrading with **sudo apt-get upgrade** on the Pi 1 first.

Also, any precompiled software will not work at full speed (although supposedly the processor will be able to run it). Still, you'll likely want to have it recompiled for the new processor! For many people, this isn't a big deal, but if you have a pre-created Pi 1 Model A+B+ card image, just be aware it won't work without performing an 'sudo apt-get upgrade' on the *older Pi 1* before installing on the Pi 2!

What to watch out for!

Watch out, the Raspberry Pi 2 Model B is VERY different from the Raspberry Pi Model B - check for that "2" when checking accessories and compatibility!

The Raspberry Pi 2 Model B looks *a lot* like a Raspberry Pi Model B+! Look for the chip on the bottom to identify the Pi 2

What hasn't changed

The basic form-factor of the Pi 2 Model B is nearly 100% the same as the Pi model B+

- The shape and size of the PCB is the same
- The 4 mounting holes are in the same location and are the same size
- The USB, Ethernet, A/V, HDMI, micro SD and microUSB connectors are int the same exact locations and are the same size
- The Camera, Display and 40-pin GPIO connectors are in the same exact locations and are the same size

The physical changes include:

- Processor chip is larger, has moved slightly
- The RAM is now soldered onto the bottom of the board (no longer PoP)
- Other components and chips are moved around slightly to make space for the larger processor and RAM chip on bottom

This means that 99% of cases designed for the Raspberry Pi Model B+ will work with the Raspberry Pi 2 Model B. This includes the Adafruit B+ Pi cases (http://adafru.it/2258)

One exception is some Pibow cases which have a layer that has cutouts for the specific location of the processor. (http://adafru.it/epX)Pimoroni has informed us that they will have a new case design that is compatible with both. Check the description of any case to make sure it is compatible with both **Raspberry Pi Model B+** *and* **Raspberry Pi 2 Model B**

What has changed, watch out!

New Processor

The processor has completely changed on the new RaspberryPi 2, instead of a ARM v6 core chip (arm6l) the BCM2836 has been upgraded to an ARM v7 core which is a much more powerful core

However, your existing Raspberry Pi SD card images may not work because the firmware and kernel must be recompiled/adapted for the new processor.

If you have a Raspberry Pi 2, and you are trying to upgrade your existing SD card, you will need to upgrade your installation. To do that, log into your Pi 1 and at a console or terminal type in **sudo apt-get upgrade** to perform the upgrade procedure. You'll need your Pi to be on the Internet to do this. Once upgraded, the card will work on both Pi 1 and Pi 2 computers

If you have any pre-compiled binaries that you are downloading, those will need updating too, in order to take advantage of the speed increase. Anything where you have access to source code can be recompiled and ought to work just fine. Supposedly any ARMv6 software is forwards compatible with ARMv7 but we haven't tested it for sure yet

Power Draw

Quad-core ARMv7 processor means higher current draw.

Just having the Pi 2 Model B running idle (no HDMI, graphics, ethernet or wifi just console cable) the Pi 2 draws **200mA**

With WiFi running, that **adds another 170mA**

If you have Ethernet instead, that **adds about 40mA**

When doing the heaviest computational tasks, we added about 200-250mA more current. So if you're really using your Pi 2 and you have a WiFi dongle, **expect to need 650mA @ 5V, at least**. More if you have stuff connected to the GPIO connector, other USB devices, Ethernet as well, etc!

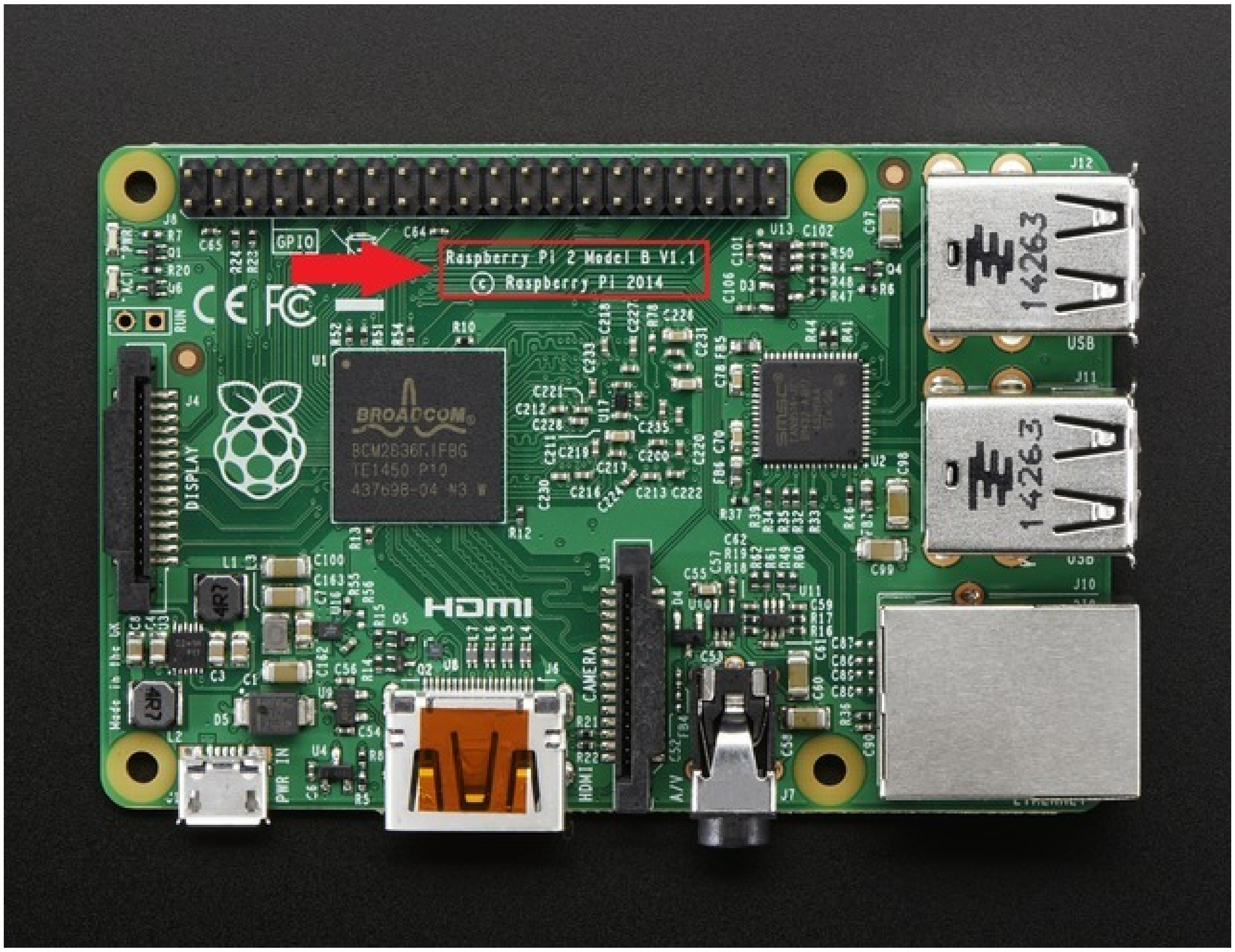
If you're still running with a cheap 5V 700mA power supply, we really recommend upgrading to a 5V @ 2000mA!

How to tell if you have a Pi 2

Since the Raspberry Pi 2 Model B looks *a lot* like the Raspberry Pi Model B+ you'll want to get good at identifying which you have.

PCB Silkscreen Name

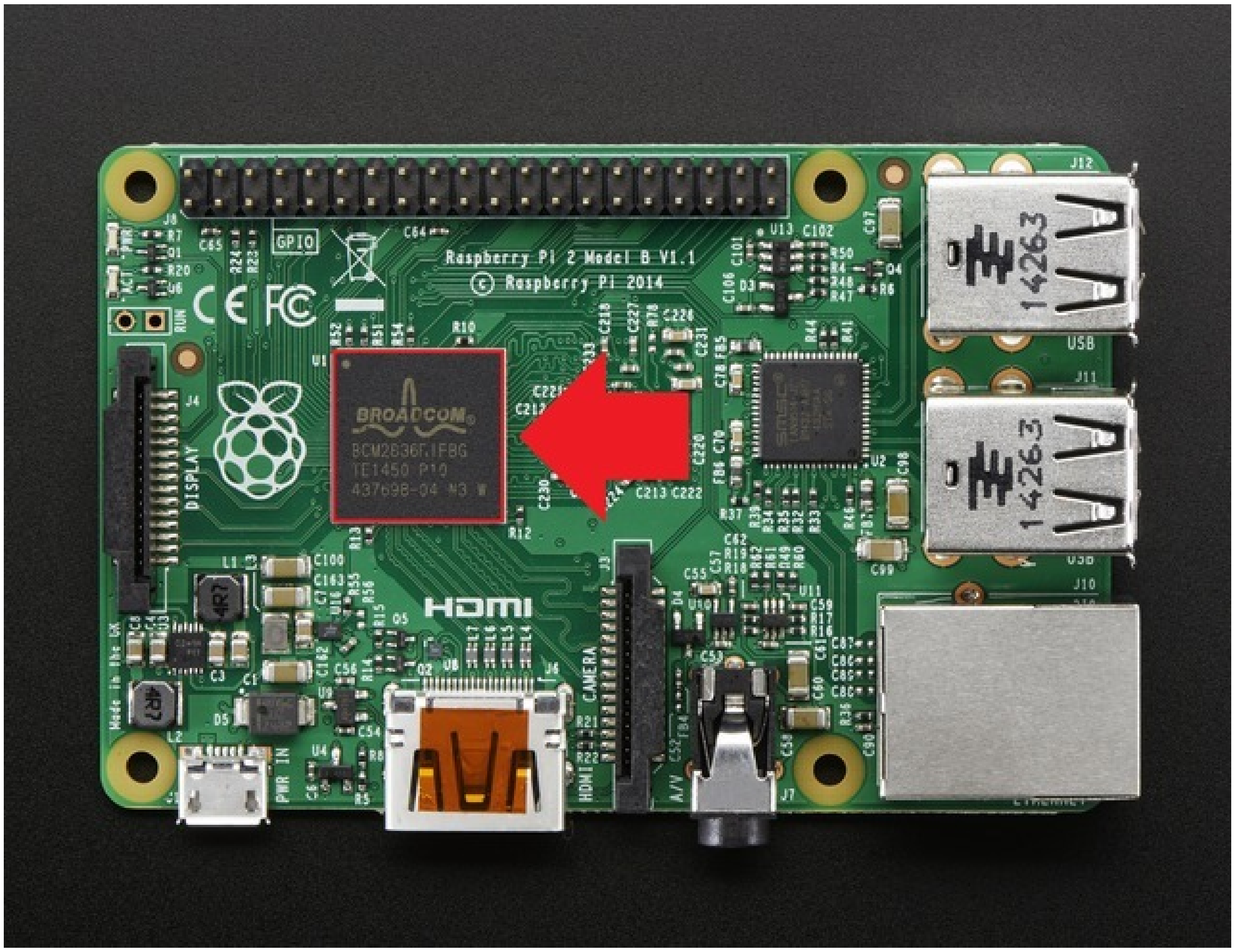
First up, you can look for the name which is near the GPIO connector:



Look for the "Raspberry Pi 2" text and you're golden!

Broadcom Logo on Processor

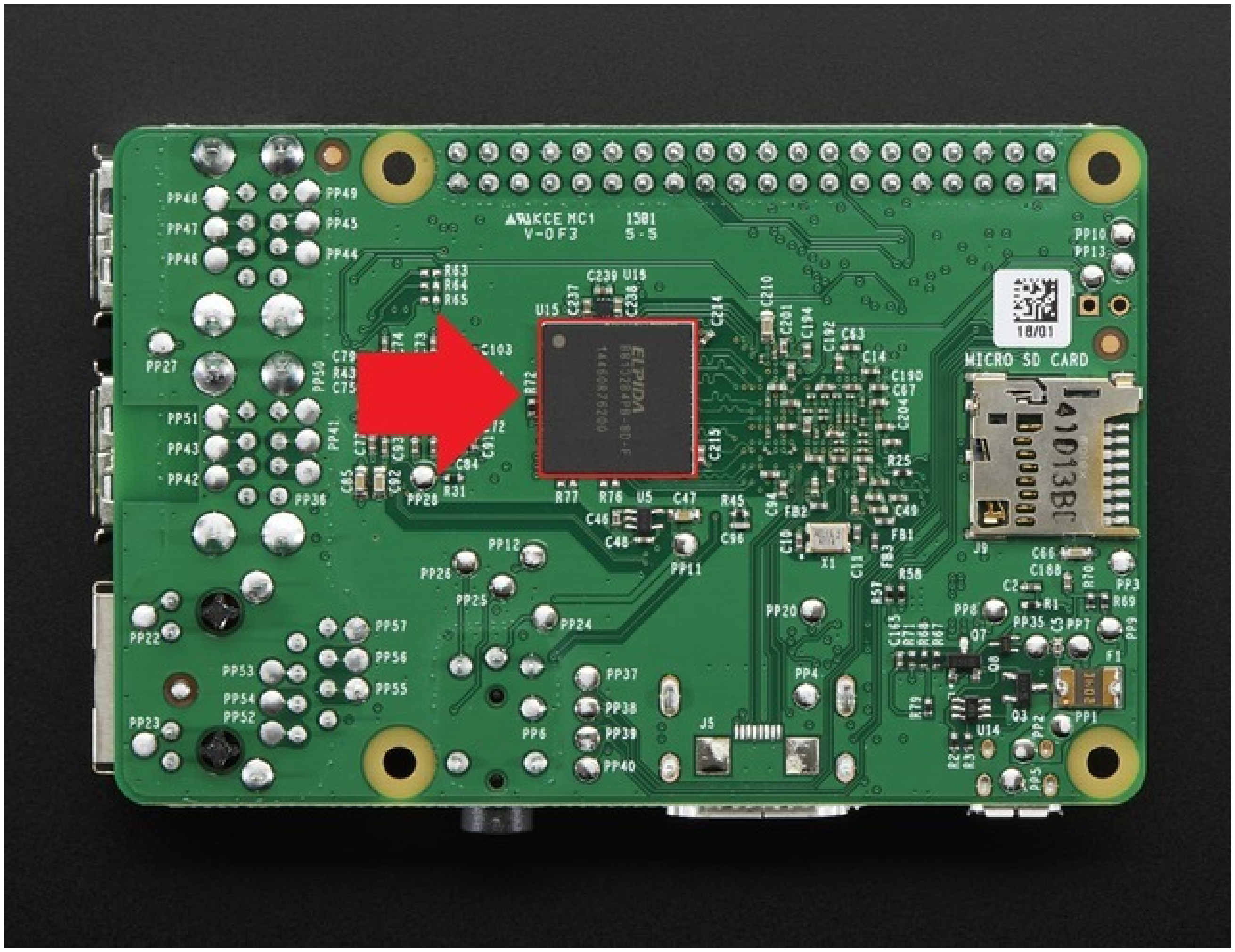
Since the Pi 2 does not use 'package-on-package', assembling the RAM on top of the processor, you can easily spot the Broadcom logo on top of the quad processor



If you don't see the logo, or you see something like "Samsung" or "Hynix" it's a B+

RAM chip on bottom

The Pi 2 has a RAM chip that is soldered onto the bottom of the Raspberry Pi's circuit board. The B+ does not have one at all, the RAM chip is soldered directly on the processor. So just look for a black square chip on the bottom of the PCB. The naming and logo on the RAM may vary depending on what company supplied the memory.



Benchmarks & Performance Improvements

The big reason to upgrade to a Pi 2 Model B over a classic Raspberry Pi Model B+ is the big boost in performance

The Pi 2 has 4 processors in one chip (the B+ has only one), an ARMv7 core vs an ARMv6, and 1 Gig of RAM vs 512 MB for the model B and B+

Those 3 improvements translate to pretty big performance increases!

OK but how *much* faster is the Pi 2 vs the Model B+? While it strongly depends on what you're doing, you should see *at least* 85% improvement (single-core processes that just depend on the ARMv7 vs ARMv6 upgrade. For anything that can take advantage of multi-core processors, you can see up to 7x increase in speed!

Using the Pi as a computer feels fast and 'desktop like' - not sluggish! Particularly for developers, compiling code on the Pi 2 is 4x faster and the extra RAM helps a lot too, so most programs can now be compiled directly on the Pi. We still recommend our Pi Kernel-O-Matic for cross-compiling kernels since you need a lot of space & RAM (http://adafru.it/epp)

Compared to other Single-Board-Computers

You can see how the Pi 2 compares to the Arduino Yun / Beaglebone Black / Intel Galileo by checking out this earlier comparison guide (we'll be updating the guide shortly to add the Pi 2 numbers!) (http://adafru.it/eq1)

We provide nbench numbers below that you can compare to the other computers until we update that tutorial...

nbench on Pi 2 @ 900MHz

```
TEST                : Iterations/sec.  : Old Index   : New Index
                    :                  : Pentium 90* : AMD K6/233*
--------------------:------------------:-------------:------------
NUMERIC SORT        :          444.24  :      11.39  :       3.74
STRING SORT         :          36.251  :      16.20  :       2.51
BITFIELD            :      1.2604e+08  :      21.62  :       4.52
FP EMULATION        :          69.824  :      33.50  :       7.73
FOURIER             :          4728.6  :       5.38  :       3.02
ASSIGNMENT          :          6.7648  :      25.74  :       6.68
IDEA                :          1297.9  :      19.85  :       5.89
HUFFMAN             :           654.5  :      18.15  :       5.80
NEURAL NET          :          6.2233  :      10.00  :       4.21
LU DECOMPOSITION    :          228.32  :      11.83  :       8.54
==========================ORIGINAL BYTEMARK RESULTS==========================
INTEGER INDEX       : 19.909
FLOATING-POINT INDEX: 8.599
Baseline (MSDOS*)   : Pentium* 90, 256 KB L2-cache, Watcom* compiler 10.0
==============================LINUX DATA BELOW===============================
CPU                 : 4 CPU ARMv7 Processor rev 5 (v7l)
L2 Cache            :
OS                  : Linux 3.18.5-v7+
C compiler          : gcc version 4.6.3 (Debian 4.6.3-14+rpi1)
libc                : libc-2.13.so
MEMORY INDEX        : 4.228
INTEGER INDEX       : 5.607
FLOATING-POINT INDEX: 4.769
Baseline (LINUX)    : AMD K6/233*, 512 KB L2-cache, gcc 2.7.2.3, libc-5.4.38
```

nbench @ 950MHz

```
TEST                : Iterations/sec.  : Old Index   : New Index
                    :                  : Pentium 90* : AMD K6/233*
--------------------:------------------:-------------:------------
NUMERIC SORT        :          481.57  :      12.35  :       4.06
STRING SORT         :            37.6  :      16.80  :       2.60
BITFIELD            :      1.1826e+08  :      20.29  :       4.24
FP EMULATION        :            87.4  :      41.94  :       9.68
FOURIER             :            5126  :       5.83  :       3.27
ASSIGNMENT          :          7.6138  :      28.97  :       7.51
IDEA                :          1450.7  :      22.19  :       6.59
HUFFMAN             :          705.88  :      19.57  :       6.25
NEURAL NET          :          6.3669  :      10.23  :       4.30
LU DECOMPOSITION    :          242.49  :      12.56  :       9.07
==========================ORIGINAL BYTEMARK RESULTS==========================
INTEGER INDEX       : 21.639
FLOATING-POINT INDEX: 9.082
Baseline (MSDOS*)   : Pentium* 90, 256 KB L2-cache, Watcom* compiler 10.0
==============================LINUX DATA BELOW===============================
CPU                 : 4 CPU ARMv7 Processor rev 5 (v7l)
L2 Cache            :
OS                  : Linux 3.18.1-v7+
C compiler          : gcc-4.7
libc                : libc-2.13.so
MEMORY INDEX        : 4.359
INTEGER INDEX       : 6.341
FLOATING-POINT INDEX: 5.037
Baseline (LINUX)    : AMD K6/233*, 512 KB L2-cache, gcc 2.7.2.3, libc-5.4.38
* Trademarks are property of their respective holder.
```

For comparison-geeks, note that if you overclock the Pi 2 to 900-1000 MHz it's essentially the same processing speed as a BeagleBone Black (also an ARMv7), but with the improved Floating Point capabilities. There's a lot of reasons to go with a BBB vs Pi2 so please note it's not that the Pi 2 is a 'replacement' for the BBB!

Sysbench tests (Compared to Pi B+)

Sysbench is a linux program that can do raw computational tests. It's a pure-math test, but will tell you the 'upper bound' for speed and is good for general comparison.

Running on a B+ @ 700MHz with one thread, we get:

```
Running the test with following options:
Number of threads: 1

Doing CPU performance benchmark

Threads started!
Done.

Maximum prime number checked in CPU test: 10000


Test execution summary:
    total time:                          523.7819s
    total number of events:              10000
    total time taken by event execution: 523.7231
    per-request statistics:
         min:                            51.99ms
         avg:                            52.37ms
         max:                            54.81ms
         approx.  95 percentile:         53.54ms

Threads fairness:
    events (avg/stddev):           10000.0000/0.00
    execution time (avg/stddev):   523.7231/0.00
```

And for 4 threads:

```
Running the test with following options:
Number of threads: 4

Doing CPU performance benchmark

Threads started!
Done.

Maximum prime number checked in CPU test: 10000


Test execution summary:
    total time:                          523.1061s
    total number of events:              10000
    total time taken by event execution: 2091.9841
    per-request statistics:
         min:                            162.66ms
         avg:                            209.20ms
         max:                            252.29ms
         approx.  95 percentile:         232.33ms

Threads fairness:
    events (avg/stddev):           2500.0000/1.22
    execution time (avg/stddev):   522.9960/0.04
```

Note that *both tests* take 523 seconds, because the B+ is a single-core processor, there is no improvement for having 4 threads vs 1 (all 4 threads are one one processor)

In comparison, the Pi 2 at 900 MHz has for a single thread:

```
Running the test with following options:
Number of threads: 1

Doing CPU performance benchmark

Threads started!
Done.

Maximum prime number checked in CPU test: 10000


Test execution summary:
    total time:                          298.6816s
    total number of events:              10000
    total time taken by event execution: 298.6632
    per-request statistics:
         min:                            29.64ms
         avg:                            29.87ms
         max:                            44.60ms
         approx.  95 percentile:         32.14ms

Threads fairness:
    events (avg/stddev):           10000.0000/0.00
    execution time (avg/stddev):   298.6632/0.00
```

298 seconds vs 523 for a single thread, so even without taking advantage of multicore, there's a 523/298 = 75% increase. That's nearly double just by having a ARMv7 doing the computation

If running with 4 threads, one on each processor, we see another big improvement

```
Number of threads: 4

Doing CPU performance benchmark

Threads started!
Done.

Maximum prime number checked in CPU test: 10000


Test execution summary:
    total time:                          76.1168s
    total number of events:              10000
    total time taken by event execution: 304.4156
    per-request statistics:
         min:                                  29.65ms
         avg:                                  30.44ms
         max:                                  63.32ms
         approx.  95 percentile:               34.97ms

Threads fairness:
    events (avg/stddev):           2500.0000/7.38
    execution time (avg/stddev):   76.1039/0.01
```

because we could split the work over 4 cores, we sped up 4x to 76 seconds.

Compared to a model B+, the Pi 2 is at most 7x faster when using multi-threaded/core computation!

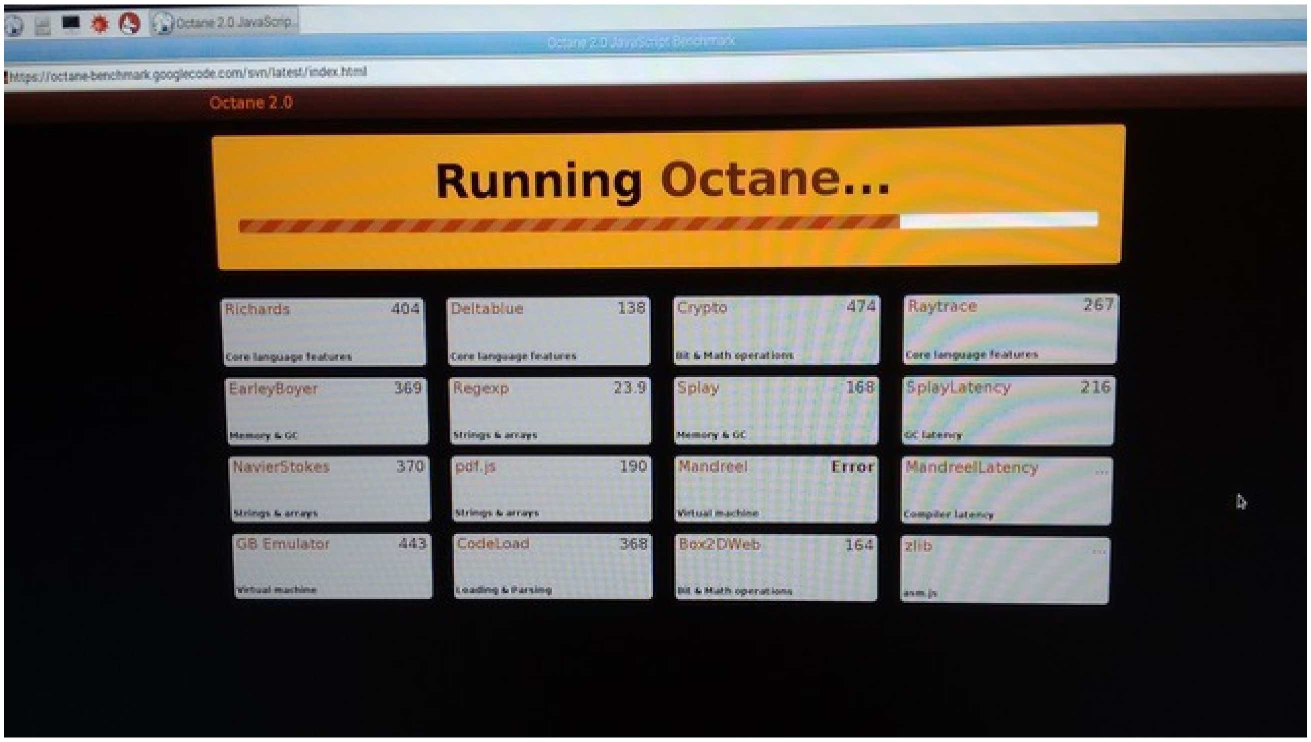
Web performance (Compared to Pi B+)

When using the Pi 2 for desktop use such as running Scratch, minecraft, or web-browsing, it feels much faster. But, that's pretty subjective and we wanted to have some Real Numbers for comparison so we ran a few web-browser Javascript tests.

Javascript is fairly processor-intensive and runs a huge amount of the interactivity of websites, so speedy Javascript will translate directly to speedy browsing!

The first test we did is called Octane, you can run it by visiting here (http://adafru.it/epY) - it runs in your webbrowser and does a series of tests.

On a B+, we actually couldn't get the test to finish without crashing, but before it crashed we got the following:

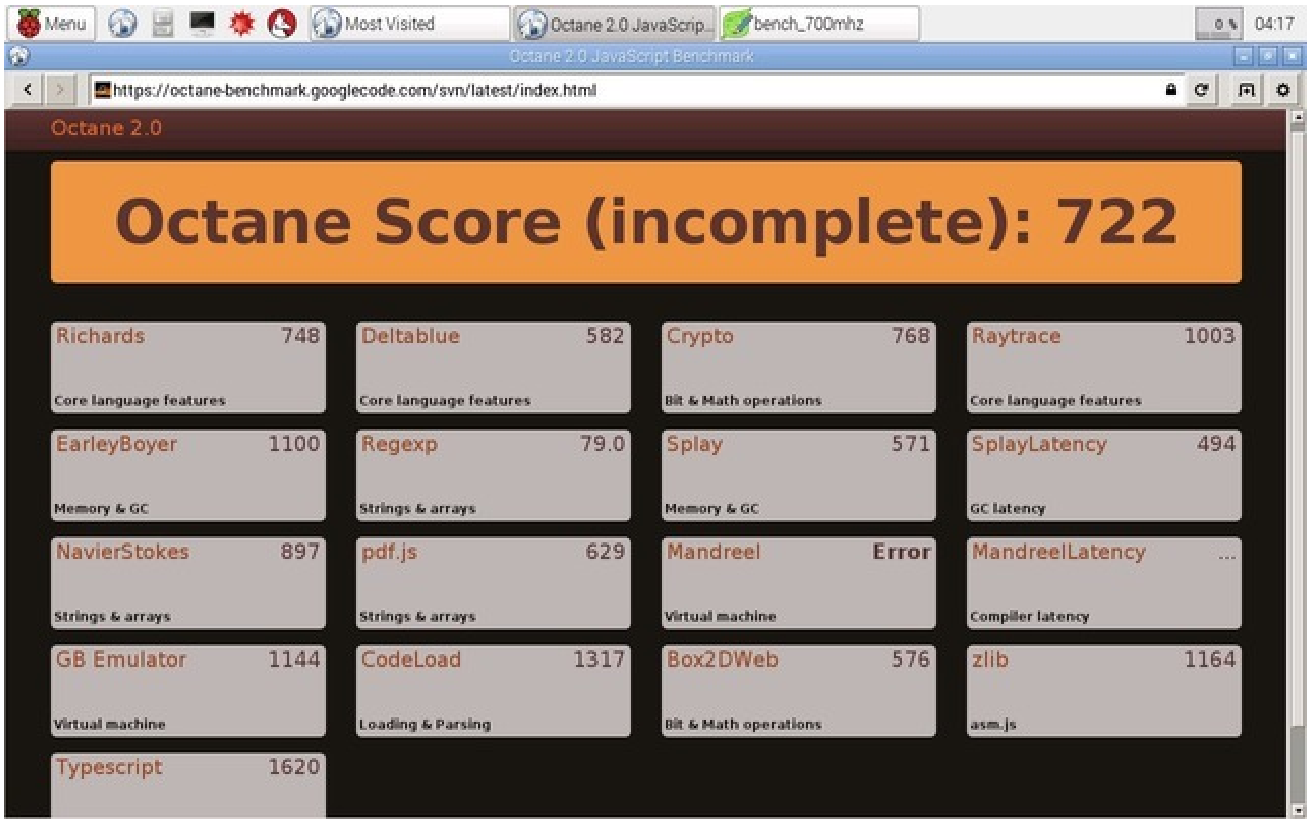

https://octane-benchmark.googlecode.com/svn/latest/index.html

Octane 2.0

Running Octane...

Richards 404 Core language features	Deltablue 138 Core language features	Crypto 474 Bit & Math operations	Raytrace 267 Core language features
EarleyBoyer 369 Memory & GC	Regexp 23.9 Strings & arrays	Splay 168 Memory & GC	SplayLatency 216 GC latency
NavierStokes 370 Strings & arrays	pdf.js 190 Strings & arrays	Mandreel Error Virtual machine	MandreelLatency ... Compiler latency
GB Emulator 443 Virtual machine	CodeLoad 368 Loading & Parsing	Box2DWeb 164 Bit & Math operations	zlib ... asm.js

Compare to the Pi 2 which did at least finish and gave us these numbers:


https://octane-benchmark.googlecode.com/svn/latest/index.html

Octane 2.0

Octane Score (incomplete): 722

Richards 748 Core language features	Deltablue 582 Core language features	Crypto 768 Bit & Math operations	Raytrace 1003 Core language features
EarleyBoyer 1100 Memory & GC	Regexp 79.0 Strings & arrays	Splay 571 Memory & GC	SplayLatency 494 GC latency
NavierStokes 897 Strings & arrays	pdf.js 629 Strings & arrays	Mandreel Error Virtual machine	MandreelLatency ... Compiler latency
GB Emulator 1144 Virtual machine	CodeLoad 1317 Loading & Parsing	Box2DWeb 576 Bit & Math operations	zlib 1164 asm.js
Typescript 1620			

Higher numbers are **better** in this case

You can tell that depending on the tests, the Pi 2 is at least 2x as fast, and in most cases is 4x as fast.

SunSpider (Compared to Pi B+)

Another test you can run is called SunSpider (http://adafru.it/epZ), it's also a Javascript benchmarker. Here's the results from running it on a B+

```
============================================
RESULTS (means and 95% confidence intervals)
--------------------------------------------
Total:                 9477.4ms +/- 0.4%
--------------------------------------------

  3d:                  1657.4ms +/- 0.9%
    cube:               552.5ms +/- 0.5%
    morph:              316.1ms +/- 0.7%
    raytrace:           788.8ms +/- 1.8%

  access:               482.1ms +/- 0.6%
    binary-trees:        80.6ms +/- 1.6%
    fannkuch:           203.0ms +/- 1.3%
    nbody:              133.7ms +/- 0.8%
    nsieve:              64.8ms +/- 3.2%

  bitops:               225.2ms +/- 0.3%
    3bit-bits-in-byte:   20.1ms +/- 1.1%
    bits-in-byte:        38.5ms +/- 2.4%
    bitwise-and:         51.1ms +/- 1.5%
    nsieve-bits:        115.5ms +/- 0.7%

  controlflow:           74.0ms +/- 1.2%
    recursive:           74.0ms +/- 1.2%

  crypto:               647.4ms +/- 2.9%
    aes:                337.7ms +/- 2.0%
    md5:                171.7ms +/- 5.4%
    sha1:               138.0ms +/- 3.6%

  date:                1503.9ms +/- 0.6%
    format-tofte:       784.9ms +/- 0.8%
    format-xparb:       719.0ms +/- 0.8%

  math:                 431.4ms +/- 1.7%
    cordic:             104.7ms +/- 2.0%
    partial-sums:       238.2ms +/- 2.4%
    spectral-norm:       88.5ms +/- 2.6%

  regexp:              174.8ms +/- 1.2%
    dna:               174.8ms +/- 1.2%

  string:             4281.2ms +/- 0.5%
    base64:            208.6ms +/- 1.6%
    fasta:             466.5ms +/- 3.7%
    tagcloud:          711.4ms +/- 0.9%
    unpack-code:      2436.8ms +/- 0.4%
    validate-input:    457.9ms +/- 0.9%
```

And running on a Pi 2:

```
============================================
RESULTS (means and 95% confidence intervals)
--------------------------------------------
Total:              2476.9ms +/- 0.7%
--------------------------------------------

  3d:                  499.8ms +/- 2.6%
    cube:              141.7ms +/- 1.4%
    morph:             150.5ms +/- 6.1%
    raytrace:          207.6ms +/- 2.1%

  access:              190.6ms +/- 1.0%
    binary-trees:       24.8ms +/- 1.2%
    fannkuch:           90.8ms +/- 1.0%
    nbody:              48.1ms +/- 2.2%
    nsieve:             26.9ms +/- 3.9%

  bitops:              100.6ms +/- 1.3%
    3bit-bits-in-byte:   8.4ms +/- 4.4%
    bits-in-byte:       19.4ms +/- 1.9%
    bitwise-and:        25.8ms +/- 1.2%
    nsieve-bits:        47.0ms +/- 2.0%

  controlflow:          25.6ms +/- 3.0%
    recursive:          25.6ms +/- 3.0%

  crypto:              194.5ms +/- 1.3%
    aes:                99.6ms +/- 0.7%
    md5:                52.3ms +/- 4.2%
    sha1:               42.6ms +/- 0.9%
```

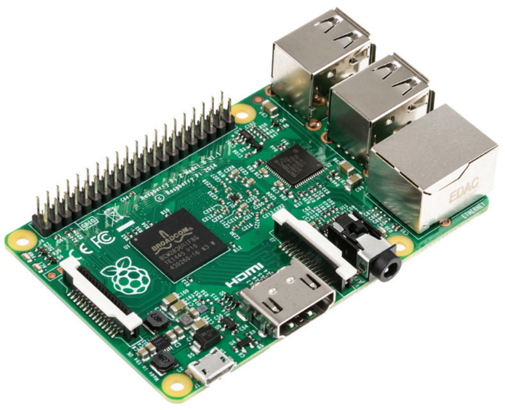
```
date:              303.5ms +/- 1.2%
  format-tofte:      154.4ms +/- 0.8%
  format-xparb:      149.1ms +/- 1.7%

math:              141.5ms +/- 1.0%
  cordic:            39.1ms +/- 1.0%
  partial-sums:      69.9ms +/- 1.0%
  spectral-norm:     32.5ms +/- 1.6%

regexp:             90.0ms +/- 0.7%
  dna:               90.0ms +/- 0.7%

string:            930.8ms +/- 0.7%
  base64:            57.6ms +/- 1.0%
  fasta:            141.2ms +/- 0.7%
  tagcloud:         160.7ms +/- 0.6%
  unpack-code:      460.6ms +/- 1.0%
  validate-input:   110.7ms +/- 1.0%
```

In this case, lower numbers are better. Again, you can see that all tests are at least 2x faster on a Pi 2 vs a B+ and most are about 4x faster!

Other tests!

OK we'll be doing more tests, but one thing we did get going was playing around with emulators. Of course the Pi 2 is much speedier than the B+ and by overclocking to 900 MHz we could run pcsx (playstation 1 emulator) and Crash Bandicoot at full speed with HDMI audio! Simply download, build and run as per this tutorial (http://adafru.it/eq0).

Raspberry Pi



Raspberry Pi 2, Model B

Product Name	**Raspberry Pi 2, Model B**
Product Description	The Raspberry Pi 2 delivers 6 times the processing capacity of previous models. This second generation Raspberry Pi has an upgraded Broadcom BCM2836 processor, which is a powerful ARM Cortex-A7 based quad-core processor that runs at 900MHz. The board also features an increase in memory capacity to 1Gbyte.
RS Part Number	**832-6274**

Specifications

Chip	Broadcom BCM2836 SoC
Core architecture	Quad-core ARM Cortex-A7
CPU	900 MHz
GPU	Dual Core VideoCore IV® Multimedia Co-Processor
	Provides Open GL ES 2.0, hardware-accelerated OpenVG, and 1080p30 H.264 high-profile decode
	Capable of 1Gpixel/s, 1.5Gtexel/s or 24GFLOPs with texture filtering and DMA infrastructure
Memory	1GB LPDDR2
Operating System	Boots from Micro SD card, running a version of the Linux operating system
Dimensions	85 x 56 x 17mm
Power	Micro USB socket 5V, 2A

Connectors:

Ethernet	10/100 BaseT Ethernet socket
Video Output	HDMI (rev 1.3 & 1.4)
	Composite RCA (PAL and NTSC)
Audio Output	3.5mm jack, HDMI
USB	4 x USB 2.0 Connector
GPIO Connector	40-pin 2.54 mm (100 mil) expansion header: 2x20 strip
	Providing 27 GPIO pins as well as +3.3 V, +5 V and GND supply lines
Camera Connector	15-pin MIPI Camera Serial Interface (CSI-2)
JTAG	Not populated
Display Connector	Display Serial Interface (DSI) 15 way flat flex cable connector with two data lanes and a clock lane
Memory Card Slot	Micro SDIO

RS

RaspiCam Documentation

July 2013

This document describes the use of the three Raspberry Pi camera applications as of July 2013.

There are three applications provided: `raspistill`, `raspivid` and `raspistillyuv`. Both `raspistill` and `raspistillyuv` are very similar and are intended for capturing images, while `raspivid` is for capturing video.

All the applications are command-line driven, written to take advantage of the mmal API which runs over OpenMAX. The mmal API provides an easier to use system than that presented by OpenMAX. Note that mmal is a Broadcom specific API used only on Videocore 4 systems.

The applications use up to four OpenMAX(mmal) components - `camera`, `preview`, `encoder` and `null_sink`. All applications use the camera component: `raspistill` uses the Image Encode component, `raspivid` uses the Video Encode component, and `raspistillyuv` does not use an encoder, and sends its YUV or RGB output direct from camera component to file.

The preview display is optional, but can be used full screen or directed to a specific rectangular area on the display. If preview is disabled, the `null_sink` component is used to 'absorb' the preview frames. It is necessary for the camera to produce preview frames even if not required for display, as they are used for calculating exposure and white balance settings.

In addition it is possible to omit the filename option, in which case the preview is displayed but no file is written, or to redirect all output to stdout. Command line help is available by typing just the application name in on the command line.

Setting up the camera hardware

Please note that camera modules are static-sensitive. Earth yourself prior to handling the PCB: a sink tap/faucet or similar should suffice if you don't have an earthing strap.

The camera board attaches to the Raspberry Pi via a 15-way ribbon cable. There are only two connections to make: the ribbon cable need to be attached to the camera PCB and the Raspberry Pi itself. You need to get it the right way round, or the camera will not work. On the camera PCB, the blue backing on the cable should be facing away from the PCB, and on the Raspberry Pi it should be facing towards the Ethernet connection (or where the Ethernet connector would be if you are using a model A).

Although the connectors on the PCB and the Pi are different, they work in a similar way. On the Raspberry Pi, pull up the tabs on each end of the connector. It should slide up easily, and be able to pivot around slightly. Fully insert the ribbon cable into the slot, ensuring it is straight, then gently press down the tabs to clip it into place. The camera PCB itself also requires you to pull the tabs away from the board, gently insert the cable, then push the tabs back. The PCB connector is a little more awkward than the one on the Pi itself. You can watch a video showing you how to attach the connectors at www.raspberrypi.org/archives/3890 (scroll down for the video).

Setting up the Camera software

Execute the following instructions on the command line to download and install the latest kernel, GPU firmware and applications. You will need an internet connection for this to work correctly.

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

Now you need to enable camera support, using the `raspi-config` program you will have used when you first set up your Raspberry Pi.

```
sudo raspi-config
```

Use the cursor keys to move to the camera option and select *enable*. On exiting `raspi-config` it will ask to reboot. The *enable* option will ensure that on reboot the correct GPU firmware will be running (with the camera driver and tuning), and the GPU memory split is sufficient to allow the camera to acquire enough memory to run correctly.

To test that the system is installed and working, try the following command:

```
raspistill -v -o test.jpg
```

The display should show a 5-second preview from the camera and then take a picture, saved to the file test.jpg, while displaying various informational messages.

Troubleshooting

If the camera is not working correctly, there are number of things to try.

- Are the ribbon connectors all firmly seated and the right way round? They must be straight in their sockets.
- Is the camera module connector firmly attached to the camera PCB? This is the connection from the smaller black camera module itself to the camera PCB. Sometimes this connection can come loose. Using a fingernail, flip up the connector on the PCB, then reseat it with gentle pressure, it engages with a very slight click.
- Have `sudo apt-get update` and `sudo apt-get upgrade` been run?
- Has `raspi-config` been run and the camera enabled?

If things are still not working, try the following:

Error : raspistill/raspivid not found. This probably means your update/upgrade failed in some way. Try it again.

Error : ENOMEM displayed. Camera is not starting up. Check all connections again.

Error : ENOSPC displayed. Camera is probably running out of GPU memory. Check `config.txt` in the `/boot/` folder. The `gpu_mem` option should be at least 128.

If, after all the above, the camera is still not working, it may have a defect (most likely because it has suffered static shock). Try posting on the Raspberry Pi forum in the camera board section to see if there is any more help available there.

Common Command line Options

Preview Window

`--preview, -p` Preview window settings <'x,y,w,h'>

Allows the user to define the size and location on the screen that the preview window will be placed. Note this will be superimposed over the top of any other windows/graphics.

`--fullscreen, -f` Fullscreen preview mode

Forces the preview window to use the whole screen. Note that the aspect ratio of the incoming image will be retained, so there may be bars on some edges.

`--nopreview, -n,` Do not display a preview window

Disables the preview window completely. Note that even though the preview is disabled, the camera will still be producing frames, so will be using power.

`--opacity, -op` Set preview window opacity

Sets the opacity of the preview windows. 0 = invisible, 255 = fully opaque.

Camera Control Options

`--sharpness, -sh`   Set image sharpness (-100 to 100)

Set the sharpness of the image, 0 is the default.

`--contrast, -co`   Set image contrast (-100 to 100)

Set the contrast of the image, 0 is the default

`--brightness, -br`   Set image brightness (0 to 100)

Set the brightness of the image, 50 is the default. 0 is black, 100 is white.

`--saturation, -sa`   Set image saturation (-100 to 100)

Set the colour saturation of the image. 0 is the default.

`--ISO, -ISO`   Set capture ISO

Sets the ISO to be used for captures. Range is 100 to 800.

`--vstab, -vs`   Turn on video stabilization

In video mode only, turn on video stabilization.

`--ev, -ev`   Set EV compensation

Set the EV compensation of the image. Range is -10 to +10, default is 0.

`--exposure, -ex` Set exposure mode

Possible options are:

off	
auto	Use automatic exposure mode
night	Select setting for night shooting
nightpreview	
backlight	Select setting for back-lit subject
spotlight	
sports	Select setting for sports (fast shutter etc.)
snow	Select setting optimized for snowy scenery
beach	Select setting optimized for beach
verylong	Select setting for long exposures
fixedfps	Constrain fps to a fixed value
antishake	Antishake mode
fireworks	Select setting optimized for fireworks

Note that not all of these settings may be implemented, depending on camera tuning.

`--awb, -awb` Set automatic white balance (AWB)

off	Turn off white balance calculation
auto	Automatic mode (default)
sun	Sunny mode
cloudshade	Cloudy mode
tungsten	Tungsten lighting mode
fluorescent	Fluorescent lighting mode
incandescent	Incandescent lighting mode
flash	Flash mode
horizon	Horizon mode

`--imxfx, -ifx` Set image effect

none	No effect
negative	Produces a negative image
solarise	Solarise the image
whiteboard	Whiteboard effect
blackboard	Blackboard effect
sketch	Sketch-style effect
denoise	Denoise the image
emboss	Embossed effect
oilpaint	Oil paint-style effect
hatch	Cross-hatch sketch style
gpen	Graphite sketch style
pastel	Pastel effect
watercolour	Watercolour effect
film	Grainy film effect
blur	Blur the image
saturation	Colour-saturate the image
colourswap	Not fully implemented
washedout	Not fully implemented
posterise	Not fully implemented
colourpoint	Not fully implemented
colourbalance	Not fully implemented
cartoon	Not fully implemented

`--colfx, -cfx` Set colour effect <U:V>

The supplied U and V parameters (range 0 to 255) are applied to the U and Y channels of the image. For example, --colfx 128:128 should result in a monochrome image.

`--metering, -mm` Set metering mode

Specify the metering mode used for the preview and capture.

average	Average the whole frame for metering
spot	Spot metering
backlit	Assume a backlit image
matrix	Matrix metering

`--rotation, -rot` Set image rotation (0-359)

Sets the rotation of the image in viewfinder and resulting image. This can take any value from 0 upwards, but due to hardware constraints only 0, 90, 180 and 270-degree rotations are supported.

`--hflip, -hf` Set horizontal flip

Flips the preview and saved image horizontally.

`--vflip, -vf` Set vertical flip

Flips the preview and saved image vertically.

`--roi, -roi` Set sensor region of interest

Allows the specification of the area of the sensor to be used as the source for the preview and capture. This is defined as x,y for the top left corner, and a width and height, all values in normalised coordinates (0.0-1.0). So to set a ROI at half way across and down the sensor, and an width and height of a quarter of the sensor use :

```
-roi 0.5,0.5,0.25,0.25
```

Application-specific settings

raspistill

`--width, -w`	Set image width <size>
`--height, -h`	Set image height <size>
`--quality, -q`	Set jpeg quality <0 to 100>

Quality 100 is almost completely uncompressed. 75 is a good all-round value.

`--raw, -r`	Add raw Bayer data to jpeg metadata

This option inserts the raw Bayer data from the camera in to the JPEG metadata.

`--output -o`	Output filename <filename>

Specify the output filename. If not specified, no file is saved. If the filename is '-', then all output is sent to stdout.

`--verbose, -v`	Output verbose information during run

Outputs debugging/information messages during the program run.

`--timeout, -t`	Time before capture and shut down

The program will run for this length of time, then take the capture (if output is specified). If not specified, this is set to 5 seconds.

`--timelapse, -tl`   Timelapse mode.

The specific value is the time between shots in milliseconds. Note you should specify %04d at the point in the filename where you want a frame count number to appear. For example:

```
-t 30000 -tl 2000 -o image%04d.jpg
```

will produce a capture every 2 seconds over a total period of 30s, named image1.jpg, image0002.jpg...image0015.jpg. Note that the %04d indicates a four-digit number with leading zeros added to pad to the required number of digits. So, for example, %08d would result in an eight-digit number.

`--thumb, -th`   Set thumbnail parameters (x:y:quality)

Allows specification of the thumbnail image inserted in to the JPEG file. If not specified, defaults are a size of 64x48 at quality 35.

`--demo, -d`   Run a demo mode <milliseconds>

This options cycles through range of camera options, and no capture is done. The demo will end at the end of the timeout period, irrespective of whether all the options have been cycled. The time between cycles should be specified as a millisecond value.

`--encoding, -e`   Encoding to use as output file

Valid options are jpg, bmp, gif and png. Note that unaccelerated image types (gif, png, bmp) will take much longer to save than jpg, which is hardware accelerated. Also note that the filename suffix is completely ignored when encoding a file.

`--exif, -x`	EXIF tag to apply to captures (format as 'key=value')

Allows the insertion of specific EXIF tags into the JPEG image. You can have up to 32 EXIF tge entries. This is useful for things like adding GPS metadata. For example, to set the longitude:

```
    --exif GPS.GPSLongitude=5/1,10/1,15/100
```

would set the longitude to 5degs, 10 minutes, 15 seconds. See EXIF documentation for more details on the range of tags available; the supported tags are as follows:

IFD0.< or
IFD1.<
ImageWidth, ImageLength, BitsPerSample, Compression, PhotometricInterpretation, ImageDescription, Make, Model, StripOffsets, Orientation, SamplesPerPixel, RowsPerString, StripByteCounts, Xresolution, Yresolution, PlanarConfiguration, ResolutionUnit, TransferFunction, Software, DateTime, Artist, WhitePoint, PrimaryChromaticities, JPEGInterchangeFormat, JPEGInterchangeFormatLength, YcbCrCoefficients, YcbCrSubSampling, YcbCrPositioning, ReferenceBlackWhite, Copyright>

EXIF.<
ExposureTime, FNumber, ExposureProgram, SpectralSensitivity, aISOSpeedRatings, OECF, ExifVersion, DateTimeOriginal, DateTimeDigitized, ComponentsConfiguration, CompressedBitsPerPixel, ShutterSpeedValue, ApertureValue, BrightnessValue, ExposureBiasValue, MaxApertureValue, SubjectDistance, MeteringMode, LightSource, Flash, FocalLength, SubjectArea, MakerNote, UserComment, SubSecTime, SubSecTimeOriginal, SubSecTimeDigitized, FlashpixVersion, ColorSpace, PixelXDimension, PixelYDimension, RelatedSoundFile, FlashEnergy, SpacialFrequencyResponse, FocalPlaneXResolution,

FocalPlaneYResolution, FocalPlaneResolutionUnit, SubjectLocation, ExposureIndex, SensingMethod, FileSource, SceneType, CFAPattern, CustomRendered, ExposureMode,WhiteBalance, DigitalZoomRatio, FocalLengthIn35mmFilm, SceneCaptureType, GainControl, Contrast, Saturation, Sharpness, DeviceSettingDescription, SubjectDistanceRange, ImageUniqueID>

GPS.<
GPSVersionID, GPSLatitudeRef, GPSLatitude, GPSLongitudeRef, GPSLongitude, GPSAltitudeRef, GPSAltitude, GPSTimeStamp, GPSSatellites, GPSStatus, GPSMeasureMode, GPSDOP, GPSSpeedRef, GPSSpeed, GPSTrackRef, GPSTrack, GPSImgDirectionRef, GPSImgDirection, GPSMapDatum, GPSDestLatitudeRef, GPSDestLatitude, GPSDestLongitudeRef, GPSDestLongitude, GPSDestBearingRef, GPSDestBearing, GPSDestDistanceRef, GPSDestDistance, GPSProcessingMethod, GPSAreaInformation, GPSDateStamp, GPSDifferential>

EINT.<
InteroperabilityIndex, InteroperabilityVersion, RelatedImageFileFormat, RelatedImageWidth, RelatedImageLength>

Note that a small subset of these tags will be set automatically by the camera system, but will be overridden by any exif options on the command line.

`--fullpreview, -fp` Full Preview mode

This runs the preview windows using the full resolution capture mode. Maximum frames per second in this mode is 15fps and the preview will have the same field of view as the capture. Captures should happen more quickly as no mode change should be required. This feature is currently under development.

raspistillyuv

Many of the options for `raspistillyuv` are the same as those for `raspistill`. This section shows the differences.

Unsupported Options:

`--exif, --encoding, --thumb, --raw, --quality`

Extra Options:

`--rgb, -rgb`	Save uncompressed data as RGB888

This option forces the image to be saved as RGB data with 8 bits per channel, rather than YUV420.

Note that the image buffers saved in `raspistillyuv` are padded to a horizontal size divisible by 16 (so there may be unused bytes at the end of each line to made the width divisible by 16). Buffers are also padded vertically to be divisible by 16, and in the YUV mode, each plane of Y,U,V is padded in this way.

raspivid

`--width, -w`	Set image width <size>

Width of resulting video. This should be between 64 and 1920.

`--height, -h`	Set image height <size>

Height of resulting video. This should be between 64 and 1080.

`--bitrate, -b`	Set bitrate

Use bits per second, so 10MBits/s would be -b 10000000. For H264, 1080p a high quality bitrate would be 15Mbits/s or more.

`--output, -o` Output filename <filename>.

Specify the output filename. If not specified, no file is saved. If the filename is '-', then all output is sent to stdout.

`--verbose, -v` Output verbose information during run

Outputs debugging/information messages during the program run.

`--timeout, -t` Time before capture and shut down

The program will run for this length of time, then take the capture (if output is specified). If not specified, this is set to five seconds. Setting 0 will mean the application will run continuously until stopped with Ctrl-C.

`--demo, -d` Run a demo mode <milliseconds>

This option cycles through range of camera options, no capture is done, the demo will end at the end of the timeout period, irrespective of whether all the options have been cycled. The time between cycles should be specified as a millisecond value.

`--framerate, -fps` Specify the frames per second to record

At present, the minimum frame rate allowed is 2fps, the maximum is 30fps. This is likely to change in the future.

`--penc, -e` Display preview image *after* encoding

Switch on an option to display the preview after compression. This will show any compression artefacts in the preview window. In normal operation, the preview will show the camera output prior to being compressed. This option is not guaranteed to work in future releases.

`--intra, -g`	Specify the intra refresh period (key frame rate/GoP)

Sets the intra refresh period (GoP) rate for the recorded video. H.264 video uses a complete frame (I-frame) every intra refresh period from which subsequent frames are based. This options specifies the numbers of frames between each I-frame. Larger numbers here will reduce the size of the resulting video, smaller numbers make the stream more robust to error.

Examples

Still captures

By default, captures are done at the highest resolution supported by the sensor. This can be changed using the -w and -h command line options.

Taking a default capture after two seconds (note times are specified in milliseconds) on viewfinder, saving in image.jpg

```
raspistill -t 2000 -o image.jpg
```

Take a capture at a different resolution

```
raspistill -t 2000 -o image.jpg -w 640
-h 480
```

Now reduce the quality considerably to reduce file size

```
raspistill -t 2000 -o image.jpg -q 5
```

Force the preview to appear at coordinate 100,100, with width 300 and height 200 pixels.

```
raspistill -t 2000 -o image.jpg -p
100,100,300,200
```

Disable preview entirely.

```
raspistill -t 2000 -o image.jpg -n
```

Save the image as a png file (lossless compression, but slower than JPEG). Note that the filename suffix is ignored when choosing the image encoding.

```
raspistill -t 2000 -o image.png —e png
```

Add some EXIF information to the JPEG. This sets the Artist tag name to Mooncake, and the GPS altitude to 123.5m. Note that if setting GPS tags you should set as a minimum GPSLatitude, GPSLatitudeRef, GPSLongitude, GPSLongitudeRef, GPSAltitude and GPSAltitudeRef.

```
raspistill -t 2000 -o image.jpg -x
IFDO.Artist=Mooncake -x
GPS.GPSAltitude=1235/10
```

Set an emboss style image effect.

```
raspistill -t 2000 -o image.jpg -ifx emboss
```

Set the U and V channels of the YUV image to specific values (128:128 produces a greyscale image)

```
raspistill -t 2000 -o image.jpg -cfx
128:128
```

Run preview ONLY for two seconds, no saved image.

```
raspistill -t 2000
```

Take timelapse picture, one every 10 seconds for 10 minutes (10 minutes = 600000ms), named image_number_1_today.jpg, image_number_2_today.jpg onwards.

```
    raspistill -t 600000 -tl 10000 -o
    image_num_%d_today.jpg
```

Take a picture and send image data to stdout

```
    raspistill -t 2000 -o -
```

Take a picture and send image data to file

```
    raspistill -t 2000 -o - > my_file.jpg
```

Video Captures

Image size and preview settings are the same as for stills capture. Default size for video recording is 1080p (1920x1080)

Record a 5s clip with default settings (1080p30)

```
    raspivid -t 5000 -o video.h264
```

Record a 5s clip at a specified bitrate (3.5MBits/s)

```
    raspivid -t 5000 -o video.h264 -b 3500000
```

Record a 5s clip at a specified framerate (5fps)

```
    raspivid -t 5000 -o video.h264 -f 5
```

Encode a 5s camera stream and send image data to stdout

```
    raspivid -t 5000 -o -
```

Encode a 5s camera stream and send image data to file

```
    raspivid -t 5000 -o - > my_file.h264
```