
Universidade de Aveiro
2008
Departamento de Electrónica, Telecomunicações e Informática

PEDRO MIGUEL RIBEIRO CALEIRO

Configuração de sistemas de visão robótica

Universidade de Aveiro
2008
Departamento de Electrónica, Telecomunicações e Informática

PEDRO MIGUEL RIBEIRO CALEIRO

Configuração de sistemas de visão robótica

dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Doutor António José Ribeiro Neves, Professor Auxiliar Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor Armando José Formoso de Pinho, Professor Associado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Dedico este trabalho à minha namorada Inês, pelo apoio e motivação que sempre me deu.

o júri

presidente

Doutor Tomás António Mendes Oliveira e Silva
Professor Associado da Universidade de Aveiro

Doutor Armando José Formoso de Pinho
Professor Associado da Universidade de Aveiro

Doutor Luís Paulo Gonçalves dos Reis
Professor Auxiliar do Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto

Doutor António José Ribeiro Neves
Professor Auxiliar Convidado da Universidade de Aveiro

agradecimentos

Agradeço aos meus orientadores Dr. António Neves e Dr. Armando Pinho a paciência e ajuda que me prestaram.

palavras-chave Visão robótica, processamento de imagem, futebol robótico, espaços de cores, inteligência artificial, câmaras digitais.

resumo Esta dissertação apresenta o trabalho desenvolvido na criação de uma aplicação para calibração da visão robótica de robôs jogadores de futebol do projecto CAMBADA da Universidade de Aveiro. Paralelamente à aplicação foi desenvolvida uma biblioteca de apoio à comunicação com periféricos de captura de vídeo, que também é discutida aqui.
Com vista a resolver alguns problemas subjacentes ao reconhecimento de objectos através de informação de cor, é apresentado um estudo sobre a influência da utilização de diversas representações da cor neste mesmo reconhecimento.

keywords

Robotic vision, image processing, robotic soccer, color spaces, artificial intelligence, digital cameras.

abstract

This thesis presents the work developed in the creation of an application for the callibration of robotic vision in soccer playing robots of the CAMBADA project at the University of Aveiro. Simultaneously to the application, a support library to communicate with video capture peripherals was also developed and is also discussed.
Aiming to resolve some of the underlying problems with color information based object recognition, a study is presented about the influence of several color representations in object recognition.

Conteúdo

1 Introdução

Robôs autónomos são máquinas dotadas de sensores e pré-programadas para cumprirem um conjunto designado de tarefas sem intervenção humana, recorrendo apenas a estímulos captados do ambiente em seu redor, através desses mesmos sensores, podendo por vezes comunicar entre si. Dentro das tarefas que estes podem realizar, contam-se resolver problemas como chegar a um determinado ponto num labirinto, encontrando o caminho e desviando-se dos obstáculos (Micro-Rato [13]); condução autónoma, em que o robô faz um percurso obedecendo a regras de trânsito (CARL [14]); ou, no caso para o qual este trabalho foi desenvolvido, jogar futebol (CAMBADA [15]).

Existem vários tipos de sensores que permitem a um robô adquirir informação do meio ambiente para se orientar (infra-vermelhos, ultra-sons, ondas rádio, câmaras, ...). Neste trabalho, explora-se a habilidade de um robô reconhecer objectos através das suas cores, informação esta que é obtida através de uma ou mais câmaras. Os espaços em que estes robôs vão operar são normalmente *color-coded*, isto é, cada objecto relevante tem uma ou várias cores identificativas diferentes, que têm que ser ensinadas aos robôs para que eles actuem correctamente. Estes espaços também costumam ter a iluminação controlada, para que as cores obtidas pelos robôs não variem. Apesar deste controlo, sombras, ligeiras alterações de iluminação ou mesmo problemas associados aos próprios sensores cromáticos dos robôs, fazem com que uma cor que o robô conseguia previamente identificar correctamente passe a ser ignorada ou vice-versa, o que obviamente não é desejado.

1.1 Objectivos

A principal contribuição deste trabalho foi o desenvolvimento de uma aplicação em *GTK* para a calibração da visão de robôs autónomos (projecto CAMBADA), com vista a substituir o *software* já existente que continha um conjunto de problemas:

- Baseado em opções por linha de comandos e em atalhos por teclado. Estes factos tornavam-na apenas passível de ser utilizada por alguém que a conhecesse de antemão, por ser pouco intuitiva.
- Necessitava de ser terminado e iniciado de novo para cada ficheiro de configuração que se pretendesse alterar.

- Falta de suporte para calibração manual de balanço de brancos integrado. Esta era feita através de outra aplicação.
- Falta de suporte para gravação dos parâmetros da câmara usados na calibração.
- Poucas definições da câmara disponíveis para ajuste.
- Falta de suporte para alteração dos *framerates* e resoluções durante a calibração.
- Opção de *Undo* limitada apenas à última selecção de cor.
- Limitado a *webcams* com um *chip Philips* com que se pudesse comunicar através do *driver* '*PWC*'.

Para a comunicação com uma gama mais variada de modelos de câmaras foi desenvolvida uma biblioteca com uma *API* uniforme para comunicação com câmaras *Firewire*, *PWC* e *Video4Linux*. Deste modo as aplicações que usam esta biblioteca podem abstrair-se do *hardware* que se encontra a ser usado.

O *software* já existente usava o espaço de cor *YMP* (uma transformação do espaço para representação polar através de luminância, módulo e fase) para representar a informação adquirida pelos sensores das câmaras e para os algoritmos de detecção de objectos. Este espaço de cores apresenta um conjunto de problemas:

- A detecção é facilmente afectada por variações na iluminação dos objectos, quer devidas ao ambiente, quer devidas aos sensores. Os robôs podem deixar de detectar objectos devido a este problema.
- As câmaras têm que ser pré-calibradas para as condições existentes de iluminação. Os seus parâmetros (ganho, tempo de exposição, balanço de brancos) têm que ser ajustados previamente.
- Não se podem utilizar os modos de ajuste automático de parâmetros da câmara.
- A detecção de objectos baseia-se apenas na informação de cor. Também se poderia basear na informação da forma do objecto e de movimento do mesmo.
- Não se sabe se este espaço de cor é o mais adequado à detecção de objectos. Existem outros espaços de cor não testados, como o IHSL que se baseia na percepção humana da cor, o conhecido *RGB* que se baseia nas propriedades aditivas da cor, etc...

Numa tentativa de resolver os problemas apresentados, foi efectuado um estudo sobre as vantagens e desvantagens de cada espaço de cores existente para efeitos de reconhecimento de objectos apenas através da informação de cor.

2 Câmaras digitais

As câmaras digitais são usadas pelos robôs para obter informação cromática sobre o meio ambiente. Estas são constituídas basicamente por uma lente, um sensor de cor (CCD - *Charged-Coupled Device* ou CMOS - *Complementary Metal–Oxide–Semiconductor*), *software* de tratamento de imagem na própria câmara e *hardware* também para tratamento de imagem, compressão, processamento e envio das imagens capturadas (que são designadas de *frames*).

2.1 Sensor CCD

Quase todas as cores do espectro da luz visível podem ser reproduzidas adicionando partes distintas de luz vermelha, verde e azul. O sensor CCD aproveita esta propriedade da luz, consistindo numa grelha de condensadores que são carregados por um elemento fotossensível, que por sua vez se encontra coberto por um filtro vermelho, verde ou azul que apenas deixa passar uma das componentes da cor, sendo que absorve todas as outras. Sempre que a câmara deseja obter uma imagem do mundo exterior, esta grelha de condensadores é descarregada por um circuito de controlo e a carga de cada um é medida.

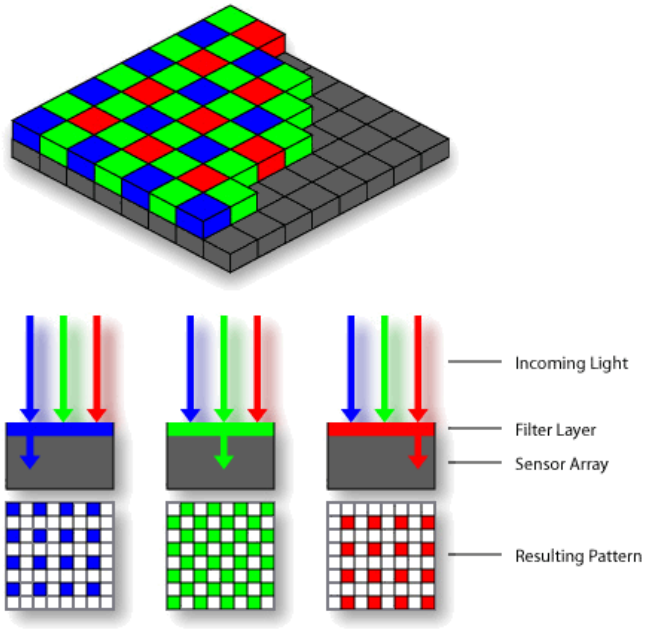

Figura 2.1: Array de Bayer [7]

Como a luz verde apresenta um maior peso na informação de luminosidade e como o olho humano é mais sensível à informação de luz do que de cor, normalmente o número de filtros que deixam passar e detectam a luz verde é o dobro dos outros para melhor quantizar a informação sobre a intensidade luminosa, visto que os filtros vermelhos e azuis absorvem-na mais que os verdes.

Esta disposição de filtros sobre os condensadores é designada por *Array de Bayer* (Figura 2.1). Nela, cada 4 filtros/condensadores correspondem a 1 *pixel* na imagem obtida final. A informação fornecida pelos 4 condensadores é processada algebricamente pelo *hardware* da câmara para encontrar o valor do *pixel*.

Normalmente, a informação digital sobre a cor de uma imagem é armazenada através da representação (ou espaço de cor) *RGB* (vermelho-verde-azul), visto ser uma forma de representação intuitiva e que segue as propriedades aditivas destas três componentes da cor.

2.2 Transmissão de imagem

Para reduzir largura de banda na altura de transmitir a informação de cor, obtida pelo sensor, para a máquina que esteja a controlar a câmara (*host*), torna-se necessário por vezes diminuir o tamanho de cada imagem adquirida através de compressão de dados. O método mais usado é representar a imagem através do espaço de cor *YUV*.

Ao contrário da *RGB*, o espaço *YUV* representa-a com base na sua informação de luminosidade e de crominância (cor). Visto que o olho humano é mais sensível à luminosidade, a crominância é representada através de menos bits do que a informação de luminosidade. Assim se consegue a compressão.

Algumas outras câmaras comprimem a imagem ainda mais, usando, por exemplo, o formato de compressão *Motion JPEG*.

2.3 Captura e processamento de imagem

De modo a que as imagens enviadas da câmara para o *host* sejam perceptíveis, esta obtém-nas de acordo com alguns parâmetros e processa-as antes de as enviar. Estes parâmetros podem ser ajustados manualmente pelo utilizador ou este pode deixar que a câmara os ajuste (modos automáticos).

2.3.1 Ganho (*Gain*)

Esta função da câmara é normalmente implementada por *hardware*. A carga extraída de cada um dos condensadores do sensor é amplificada por *hardware* antes de ser quantizada. Isto tem o efeito de tornar as imagens mais claras, mas também mais ruidosas, visto que o ruído também é amplificado. Normalmente, o ganho pode ser colocado em modo automático, ou seja, a própria câmara regula a quantidade de ganho a aplicar através de um algoritmo implementado por *software* que avalia a claridade de cada imagem precedente antes de regular o ganho da seguinte.

2.3.2 Diafragma (*Shutter*) ou Tempo de exposição (*Exposure*)

Estas funções são equivalentes e logo não existem em simultâneo na mesma câmara, sendo também implementadas por *hardware*. A câmara regula o tempo que o diafragma, responsável por deixar entrar a luz para o sensor, se encontra aberto. Este tempo é limitado pela velocidade de transmissão das imagens adquiridas. Por exemplo, se a velocidade for de 15 imagens por segundo, o diafragma apenas pode estar aberto 1/15 segundo. Quanto mais tempo estiver aberto, mais clara fica a imagem obtida, mas ao contrário do ganho, este processo baixa a SNR (relação sinal ruído) da imagem. O tempo de abertura pode também por vezes ser posto em modo automático, no qual o *software* da câmara avalia, tal como para o ganho, o tempo que deve ser usado para obter a *frame* seguinte, baseado na claridade da anterior. Normalmente, a câmara dá prioridade ao *shutter* em relação ao *gain* na altura de tornar as imagens mais claras, de modo a não torná-las ruidosas.

2.3.3 Claridade (*Brightness*)

Esta função é implementada apenas pelo *software* da câmara. Para cada *pixel* é adicionado um valor fixo de claridade o que tem o efeito de torná-la mais clara (valor positivo) ou mais escura (valor negativo). Em casos extremos, este processo tem o efeito de tornar a imagem ou completamente preta ou completamente branca. Este parâmetro da câmara não pode normalmente ser colocado em modo automático.

2.3.4 Contraste (*Contrast*)

Esta função também é efectuada puramente por *software*. Para cada *pixel*, subtrai-se-lhe metade do máximo valor possível, multiplica-se o resultado pelo valor de contraste escolhido e volta-se a somar metade do valor máximo possível. Esta operação tem o efeito de aproximar as cores claras do branco e as escuras do preto. Num caso extremo, a imagem ficaria branca e preta. Esta função serve para se poder distinguir melhor as cores. Tal como a função de *brightness*, esta também não costuma ter um modo automático controlado pela câmara.

2.3.5 Gama (γ)

Esta função pode ser implementada quer pelo *hardware* ou *software* da câmara. É normalmente usada para que a mesma informação de imagem apresente o mesmo aspecto visual em monitores diferentes. Para cada *pixel* efectua-se a operação $Pc = Po^{\frac{1}{\gamma}}$, onde Pc e Po são os *pixels* corrigido e original respectivamente, e γ é o valor de correcção. O efeito desta transformação é de aclarar ou escurecer mais os *pixels* escuros do que os claros. Esta função não costuma ter modo automático.

2.3.6 Saturação (*Saturation*)

A saturação de uma cor refere-se ao quão mais vibrante esta é. Uma cor pouco saturada aproxima-se do cinzento, enquanto se for muito saturada torna-se brilhante (Figura 2.2). Os algoritmos, implementados por *software*, variam de câmara para câmara. O ajuste desta definição costuma ser apenas manual.

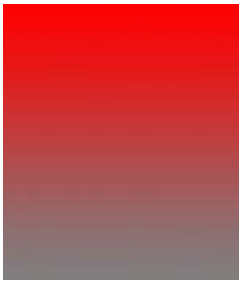

Figura 2.2: Escala de saturação da cor vermelha.

2.3.7 Coloração (*Hue*)

Algumas vezes esta função é erroneamente trocada com a saturação. Alterar este parâmetro na câmara tem o efeito de alterar completamente as cores de uma imagem (Fig. 2.3) e, como tal, não é usado por defeito pela câmara e também não pode ser posto em modo automático.

Figura 2.3: Diferentes colorações de uma imagem

2.3.8 Balanço de brancos (*White Balance*)

Esta trata-se de uma das funções mais importantes da câmara. Costuma ser implementada por *software*.

Diferentes fontes de iluminação dão diferentes tonalidades a uma imagem capturada que não seja tratada. Por exemplo, a luz do sol dá uma tonalidade azul às imagens, enquanto que a luz duma lâmpada comum (tungsténio) dá-lhes uma tonalidade vermelha (Figuras 2.4 e 2.5).

Figura 2.4: Iluminação por lâmpadas comuns

Para compensar este problema, a câmara pode ajustar as tonalidades vermelha e azul (e por vezes até a verde, mas não é comum) da imagem de forma a que esta tenha um aspecto 'natural'. Esta compensação pode ser feita automaticamente pela câmara ou manualmente pelo utilizador.

Quando se deixa que a câmara ajuste estes parâmetros, esta pode usar os mais variados algoritmos para descobrir os valores a usar. Os mais comuns são tentar que a próxima imagem tenha componentes iguais de azul, verde e vermelho com uma análise baseada na imagem anterior ou então tentar descobrir um objecto na imagem que se saiba ser branco e tentar que este fique branco, através do processamento, na próxima imagem.

Figura 2.5: Iluminação ambiente

3 Espaços de cor

Espaços de cor são diferentes formas de representar uma determinada cor digitalmente. Existem variados espaços, cada um com funções e características diferentes.

3.1 *RGB*

Neste espaço, a cor é dividida nas suas componentes vermelha (R), verde (G) e azul (B). Este espaço de cor tem em conta as propriedades aditivas destas três componentes da cor (Fig. 3.1). Somando as três em partes distintas pode-se obter quase qualquer cor do espectro visível.

Figura 3.1: Propriedades aditivas do espaço *RGB*

O espaço *RGB* é bastante usado em todo o tipo de aplicações digitais que envolvam o uso de cor (armazenamento de imagens, descrição de cores em páginas *web*, etc...). O uso do espaço *RGB* como o *standard* para apresentação de cor na *Internet* tem as suas raízes no *standard RCA Color-TV* de 1953 e no uso por *Edwin Land* deste formato nas câmaras *Land* / *Polaroid*.

3.2 *YUV*

Este espaço representa a informação de determinada cor com base na sua luminosidade (Y) e componentes cromáticas (U, também por vezes denominado Cb e V, também por vezes denominado Cr). A luminosidade pode ser visualizada como sendo uma escala cinza que se dirige do preto ao branco. As componentes cromáticas encontram-se representadas na Figura 3.2, com

gamas que vão de -0,5 a 0,5.

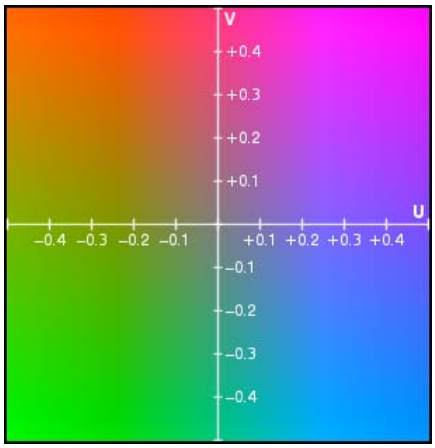

Figura 3.2: Representação do espaço *YUV*

A vantagem na utilização deste espaço, como já referido, prende-se ao facto da visão humana distinguir melhor a componente Y do que as componentes UV entre *pixels* numa imagem. Logo, as componentes UV podem ser sub-amostradas, ou seja, podem ser escaladas de modo a que o seu armazenamento ocupe menos espaço ou então podem ser reutilizadas para vários *pixels* em conjunto, o que produz o mesmo efeito. Esta operação remove alguma informação original sobre as cores, mas assume-se que um ser humano não conseguirá distinguir a diferença, pelo que se trata de um bom método de compressão (compressão perceptual).

Existem vários tipos de formatos *YUV*. As diferenças entre cada um deles prende-se com a quantidade de componentes U e V que são armazenadas em relação às componentes Y (a Figura 3.3 mostra alguns exemplos).

3.2.1 *YUV* 4:4:4

Cada um dos três canais é amostrado as mesmas vezes, logo cada *pixel* da imagem contém a informação de cor completa sem perdas.

Mapeamento:

$Y_0\ U_0\ V_0\ Y_1\ U_1\ V_1\ Y_2\ U_2\ V_2\ Y_3\ U_3\ V_3$

fica mapeado nos seguintes *pixels*:

$[Y_0\ U_0\ V_0]\ [Y_1\ U_1\ V_1]\ [Y_2\ U_2\ V_2]\ [Y_3\ U_3\ V_3]$.

Este espaço é usado, por exemplo, para armazenamento intermediário em aplicações de pós-produção cinematográfica, visto que armazena a informação sem perdas.

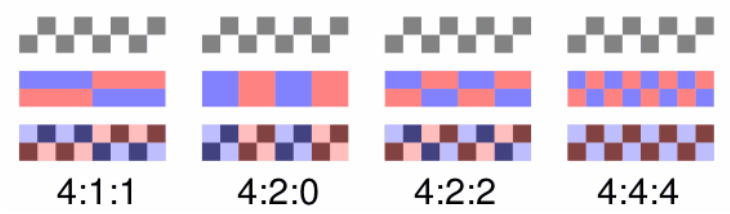

Figura 3.3: Representação de modos diferentes do espaço *YUV [4]*.

3.2.2 *YUV* 4:2:2

Cada um dos canais U e V é amostrado metade das vezes do que o canal Y, logo a resolução horizontal é metade da do *YUV* 4:4:4.

Mapeamento:

$Y_0\ U_0\ Y_1\ V_1\ Y_2\ U_2\ Y_3\ V_3$

fica mapeado nos seguintes *pixels*:

$[Y_0\ U_0\ V_1]\ [Y_1\ U_0\ V_1]\ [Y_2\ U_2\ V_3]\ [Y_3\ U_2\ V_3]$

Formatos vídeo que usam este espaço:

- *Digital betacam*
- DVCPRO50
- D-9
- CCIR 601 / *Serial digital interface* / D1

3.2.3 *YUV* 4:1:1

A resolução de cor horizontal (U e V) é dividida por um quarto. Vídeo neste formato usa 6 bytes para armazenar cada *macropixel* (um quadrado de *pixels* 2x2).

Mapeamento:

$Y_0\ U_0\ Y_1\ Y_2\ V_2\ Y_3$

fica mapeado nos seguintes *pixels*:

$[Y_0\ U_0\ V_2]\ [Y_1\ U_0\ V_2]\ [Y_2\ U_0\ V_2]\ [Y_3\ U_0\ V_2]$

Formatos vídeo que usam este espaço:

- DVCPRO
- *NTSC* DV e DVCAM
- D-7

3.2.4 *YUV* 4:2:0

A designação 4:2:0 não significa que não existe informação U e V, significa apenas que em cada linha apenas uma das informações de cor é armazenada (U ou V). O canal cromático armazenado vai variando de linha para linha. Assim, o rácio é 4:0:2 numa linha, 4:2:0 na seguinte e assim em diante. Isto leva a que a resolução de cor seja metade da resolução de luminosidade (Y) quer na horizontal, quer na vertical. Vídeo neste formato, tal como no *YUV* 4:1:1, usa 6 bytes para armazenar cada *macropixel* (um quadrado de *pixels* 2x2).

Mapeamento:

Yo0 Uo0 Yo1 Yo2 Uo2 Yo3

Ye0 Ve0 Ye1 Ye2 Ve2 Ye3

ficam mapeados nos seguintes *pixels*:

[Yo0 Uo0 Ve0] [Yo1 Uo0 Ve0] [Yo2 Uo2 Ve2] [Yo3 Uo2 Ve2]

[Ye0 Uo0 Ve0] [Ye1 Uo0 Ve0] [Ye2 Uo2 Ve2] [Ye3 Uo2 Ve2].

A qualidade deste método de codificação é muito próxima do *YUV* 4:1:1 e é usado nos seguintes formatos:

- *DVD* e outras implementações '*Main Profile MPEG-2*'
- *PAL DV* e *DVCAM*
- *HDV*
- implementações mais comuns de *JPEG* e M*JPEG*

3.2.5 *YUV* Planar

Um formato *YUV* designa-se por planar quando a informação sobre os três canais é codificada em três blocos separadas, em vez de estar misturada (*interleaved*). Logo temos primeiro toda a informação Y codificada em sequência, depois outra sequência com a informação U e por fim a informação V. Esta representação pode ser aplicada a qualquer um dos rácios *YUV*.

3.2.6 Conversão de/para *RGB*

RGB para *YUV [16]*:

Y = (0.257 * R) + (0.504 * G) + (0.098 * B) + 16

V = (0.439 * R) - (0.368 * G) - (0.071 * B) + 128

U = -(0.148 * R) - (0.291 * G) + (0.439 * B) + 128

YUV para *RGB [16]*:

B = 1.164(Y - 16) + 2.018(U - 128)

G = 1.164(Y - 16) - 0.813(V - 128) - 0.391(U - 128)

R = 1.164(Y - 16) + 1.596(V – 128)

Assume-se que Y, U, V, R, G e B estão compreendidos entre 0 e 255, sendo necessário verificar que os valores continuam dentro desta gama após conversão.

3.3 *HSV*

Esta representação de cor, também chamada de HSB, divide a informação em coloração (H), saturação (S) e valor (V). Este espaço de cor pode ser representado através dum cilindro. A coloração varia na circunferência exterior deste cilindro (Fig. 3.4). A saturação varia com a distância ao eixo do cilindro. O valor (V) é a componente vertical deste.

Os artistas costumam preferir esta representação de cor porque é mais análoga ao modo como os humanos percebem a cor. Enquanto o *RGB* usa as propriedades aditivas da cor para a representar, este método usa algo mais familiar: “que cor é?”; “quão vibrante é?”; “é escura ou clara?”.

Exemplos de aplicações que usam o *HSV*:

- Seleccionador de cores de sistema no *Apple Mac OS X* (tem um disco de cor para H e S, e uma barra para o V)
- *GIMP*
- *Xara X*
- *Paint.net* (tem um disco de cor para H e S, e uma barra para o V)

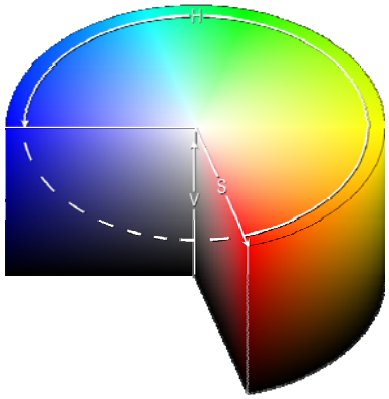

Figura 3.4: Representação do espaço *HSV*.

3.3.1 Conversão de/para *RGB*

RGB para *HSV [5] [34]*:

$$H = \begin{cases} indefinido \xrightarrow{se} MAX = MIN \\ 60 \times \dfrac{G-B}{MAX-MIN} + 0 \xrightarrow{se} MAX = R \wedge G \geq B \\ 60 \times \dfrac{G-B}{MAX-MIN} + 360 \xrightarrow{se} MAX = R \wedge G < B \\ 60 \times \dfrac{G-B}{MAX-MIN} + 120 \xrightarrow{se} MAX = G \\ 60 \times \dfrac{G-B}{MAX-MIN} + 240 \xrightarrow{se} MAX = B \end{cases}$$

$$S = \begin{cases} 0 \xrightarrow{if} MAX = 0 \\ 1 - \dfrac{MIN}{MAX} \xrightarrow{se} outros \end{cases}$$

$$V = MAX$$

Onde MIN e MAX são os valores mínimos e máximos de entre os valores de R, G e B para uma certa cor, e estes estão compreendidos entre 0,0 e 1,0. H varia de 0 a 359 (graus). S e V variam de 0,0 a 1,0. Normalmente, usa-se H = 0 quando o valor deste é indefinido.

HSV para *RGB [5] [34]*:

$$Hi = \frac{H}{60} \bmod 6$$
$$f = \frac{H}{60} - Hi$$
$$p = V(1 - S)$$
$$q = V(1 - fS)$$
$$t = V(1 - (1 - f)S)$$
$$\begin{cases} R = v, G = t, B = p \xrightarrow{se} H = 0 \\ R = q, G = V, B = p \xrightarrow{se} H = 1 \\ R = p, G = V, B = t \xrightarrow{se} H = 2 \\ R = p, G = q, B = V \xrightarrow{se} H = 3 \\ R = t, G = p, B = V \xrightarrow{se} H = 4 \\ R = V, G = p, B = q \xrightarrow{se} H = 5 \end{cases}$$

3.4 *HLS*

Este espaço de cor, também por vezes designado de *HSI* ou *HSL*, divide a informação de cor em três componentes, coloração (H), saturação (S) e luminância (L).

O *HLS* é idêntico ao *HSV*, mas separa melhor a noção de saturação e luminância, pelo que é melhor para artistas. No *HLS*, a saturação vai desde a cor totalmente saturada até ao cinza. No *HSV* vai desde a cor totalmente saturada até ao branco, o que não é muito intuitivo. A luminância no *HSL* abrange toda a gama desde a coloração escolhida desde o preto até ao branco. No *HSV*, o 'valor' vai apenas até metade dessa gama, desde o preto até à coloração escolhida (Fig. 3.5).

Exemplos de aplicações que usam o *HLS*:

- A especificação CSS3
- *Inkscape* (a partir da versão 0.42)
- *Macromedia Studio*
- Seleccionador de cores de sistema no *Microsoft windows* (incluindo *Microsoft Paint*)
- *Paint Shop Pro*

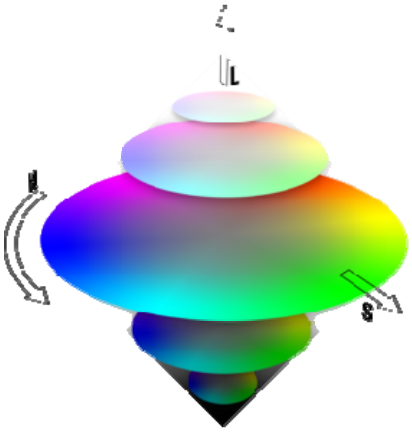

Figura 3.5: Representação do espaço *HLS*.

3.4.1 Conversão de/para *RGB*

RGB para *HLS [3]*:

$$H = \begin{cases} indefinido \xrightarrow{se} MAX = MIN \\ 60º \times \dfrac{G-B}{MAX-MIN} \xrightarrow{se} MAX = R \\ 60º \times \dfrac{B-R}{MAX-MIN} + 120º \xrightarrow{se} MAX = G \\ 60º \times \dfrac{R-G}{MAX-MIN} + 240º \xrightarrow{se} MAX = B \end{cases}$$

$$S = \begin{cases} 0 \xrightarrow{se} MIN = MAX \\ \dfrac{MAX-MIN}{MAX+MIN} = \dfrac{MAX-MIN}{2L} \xrightarrow{se} 0 < L \le \dfrac{1}{2} \\ \dfrac{MAX-MIN}{2-(MAX+MIN)} = \dfrac{MAX-MIN}{2-2L} \xrightarrow{se} L > \dfrac{1}{2} \end{cases}$$

$$L = \frac{1}{2}(MAX + MIN)$$

MIN e MAX são o valor mínimo e máximo entre os valores de R, G e B para uma certa cor, que estão compreendidos entre 0,0 e 1,0. Mais uma vez usa-se H = 0 quando o valor deste é indefinido. H varia entre 0 e 359 e L e S variam entre 0,0 e 1,0.

HLS para *RGB [3]*:

$$temp2 = \begin{cases} L \times (1.0 + S) \xrightarrow{se} L < 0.5 \\ L + S - L \times S \xrightarrow{se} L \geq 0.5 \end{cases}$$

$$temp1 = 2.0 \times L - temp2$$

$$H_k = \frac{H}{360}$$

$$temp3_R = H_k + \frac{1}{3}$$

$$temp3_G = H_k$$

$$temp3_B = H_k - \frac{1}{3}$$

$$temp3_C = \begin{cases} temp3_C + 1.0 \xrightarrow{se} temp3_C < 0, C \in \{R, G, B\} \\ temp3_C - 1.0 \xrightarrow{se} temp3_C > 1, C \in \{R, G, B\} \end{cases}$$

para cada cor C = R, G, B:

$$color_C = \begin{cases} temp1 + (temp2 - temp1) \times 6.0 \times temp3_C \xrightarrow{se} temp3_C < \frac{1}{6} \\ temp2 \xrightarrow{se} \frac{1}{6} \leq temp3_C < \frac{1}{2} \\ temp1 + (temp2 - temp1) \times \left(\frac{2}{3} - temp3_C \right) \times 6.0 \xrightarrow{se} \frac{1}{2} \leq temp3_C < \frac{2}{3} \\ temp1 \xrightarrow{se} other \end{cases}$$

3.5 *IHLS*

Este formato é exactamente igual ao *HLS*, mas a informação de coloração (H) é obtida de modo diferente pelo que é designada de *Improved Hue*, ou seja Coloração Melhorada.

Esta alteração no canal H também produz mudanças no canal S, visto que os dois estão relacionados. Enquanto no *HLS*, o canal S apresenta bastante ruído, no espaço *IHLS* tal não acontece, porque neste espaço a saturação não é normalizada em função da luminosidade [27], o que torna este formato mais adequado para analisar estatisticamente a informação de cor numa imagem.

3.5.1 Conversão de *RGB[28]*

$$s = \max(R,G,B) - \min(R,G,B)$$
$$y = 0.2125R + 0.7154G + 0.0721B$$
$$c_{r1} = R - \frac{G+B}{2}, c_{r2} = \frac{\sqrt{3}}{2}(B-G)$$
$$c_r = \sqrt{{c_{r1}}^2 + {c_{r2}}^2}$$
$$\Theta^H = \begin{cases} indefinido \xrightarrow{se} c_r = 0 \\ \arccos\left(\frac{c_{r1}}{c}\right) \xrightarrow{se} c_r \neq 0 \wedge c_{r2} \leq 0 \\ 360^o - \arccos\left(\frac{c_{r1}}{c}\right) \xrightarrow{se} c_r \neq 0 \wedge c_{r2} > 0 \end{cases}$$

onde 's' corresponde a S, 'y' corresponde a L e 'θ^H' corresponde a IH. R, G e B são valores normalizados, ou seja, a sua soma é igual a 1,0.

3.5.2 Conversão para *RGB[26]*

Podemos converter de *IHLS* para *RGB* a partir da seguinte equação. Nem todos os valores de *IHLS* representam valores válidos quando convertidos para *RGB*.

$$C = \frac{\sqrt{3}S}{2\sin(120^o - H^*)}$$
$$H^* = H - k \times 60^o \xrightarrow{onde} k \in \{1,2,3,4,5\}$$
$$C_1 = C \times \cos(H)$$
$$C_2 = -C \times \sin(H)$$
$$C_1 = C_2 = 0 \xrightarrow{quando} hue \quad indefinido$$
$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0000 & 0.7875 & 0.3714 \\ 1.0000 & -0.2125 & -0.2059 \\ 1.0000 & -0.2125 & 0.9488 \end{bmatrix} \begin{bmatrix} L \\ C_1 \\ C_2 \end{bmatrix}$$

3.6 *YIQ*

Este é um espaço de cor que era originalmente usado nas transmissões televisivas no formato *NTSC* (que agora usa *YUV*). Y significa luminosidade, I e Q são a crominância (Fig. 3.6). IQ no *YIQ* e UV no *YUV* podem ser pensados como diferentes sistemas de coordenadas no mesmo plano

de crominância (cor).

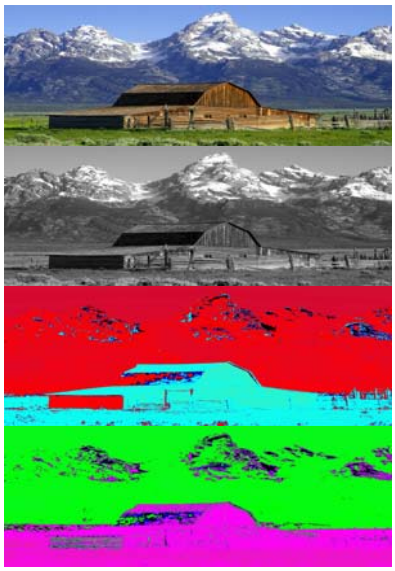

Figura 3.6: Uma imagem representada através das suas componentes Y, I e Q.

Este espaço pretende tirar partido das propriedades como a visão humana regista a cor. O olho é mais sensível em variações de cor de laranja ao azul (I) do que do púrpura ao verde (Q). Logo, pode-se representar a componente Q utilizando menos espaço que a I. Tal como no *YUV*, este método comprime a imagem.

Esta representação é por vezes utilizada em transformações de processamento de imagem. Por exemplo, aplicando uma equalização de histograma directamente nos canais *RGB*, alteraria as cores em relação umas às outras, resultando numa imagem com cores sem sentido. Em vez disso, a equalização é aplicada ao canal Y que apenas normaliza a informação de luminosidade da imagem. Esta transformação pode ser efectuada em qualquer representação que tenha a informação de luminância separada dos outros canais.

3.7 *XYZ*

Este espaço (Fig. 3.7), também conhecido por CIE 1931 *(Commission Internationale de l'Éclairage 1931)*, é um espaço baseado em medidas efectuadas directamente no olho humano. Foi baseado numa série de experiências realizadas do final dos anos 20 por *David Wright* e *John Guild*. X, Y e Z representam, aproximadamente, o mesmo que R, G e B (vermelho, verde e azul).

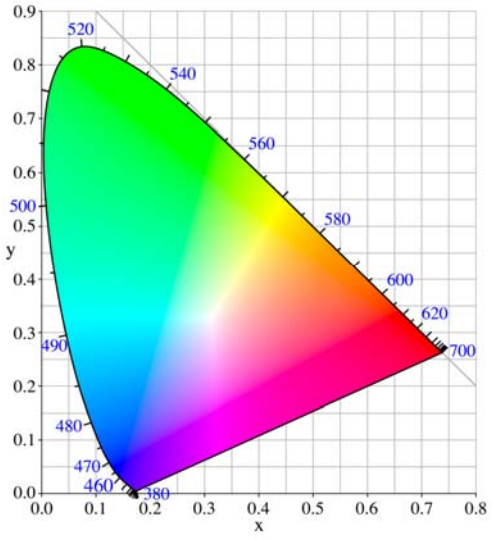

Figura 3.7: Representação do espaço *XYZ* normalizado (*xyz*).

3.7.1 Conversão de *RGB*

Podemos transformar as componentes *RGB* em *XYZ* através da matriz de transformação seguinte (assumindo que a cor branca corresponde às componentes [1.0; 1.0; 1.0] em *RGB* normalizado):

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.49 & 0.17697 & 0.0 \\ 0.31 & 0.8124 & 0.01 \\ 0.2 & 0.01063 & 0.99 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Esta transformação, altera as quantidades de vermelho, verde e azul de modo a que passem a estar directamente relacionadas com as tonalidades das mesmas cores detectadas pelo olho humano. Estas são diferentes, por exemplo, das emitidas por um monitor.

3.8 *Lab* e *Luv*

Lab é abreviatura de dois espaços de cor conhecidos como *CIE Lab* e *Hunter lab*. Ambos são baseados no espaço CIE 1931 *XYZ*. No entanto, o espaço *CIE* é calculado usando raízes cúbicas, enquanto que o espaço *Hunter* usa raízes quadradas. Vamos focar-nos no espaço *CIE Lab*, visto que o *Hunter* é pouco usado.

O intento deste espaço é produzir uma representação perceptualmente mais linear que outros

espaços de cor existentes. Perceptualmente linear significa que uma variação numa certa quantidade de um parâmetro deverá produzir uma variação com a mesma 'importância visual' na imagem. Quando se armazenam valores de precisão limitada, isto pode melhorar a reprodução dos tons. Este espaço define 'cor' exactamente, ao contrário do *RGB* que pode ser considerado uma 'receita' para misturar luz.

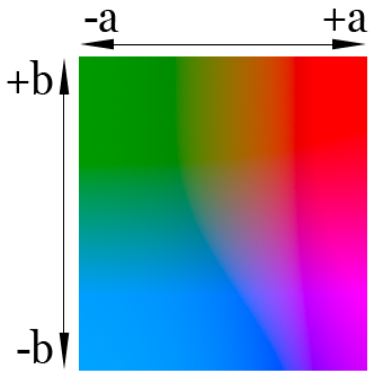

Figura 3.8: Componentes a e b do espaço de cor *Lab*.

Os três parâmetros (Fig. 3.8) neste modelo representam luminosidade (L), a posição da cor entre o magenta e o verde (a) e a posição da cor entre o amarelo e o azul (b).

O espaço *Luv*, também conhecido por *CIE Luv* é bastante parecido com o *Lab*, no aspecto em que uma variação de quantidade num dos canais de cor produz uma variação que tem a mesma variação perceptual na imagem. Mas as componentes de cor, u e v (crominâncias) são diferentes das componentes a e b.

Ambos os espaços são derivados do espaço *XYZ* que é um espaço de cor que necessita duma referência (o valor da cor branca) para ser representado. Logo, ambos também precisam de valores predefinidos do que é a cor branca nas suas representações.

4 Aplicações desenvolvidas

Neste trabalho foram desenvolvidas duas aplicações que designámos por '*LiveColorGUI*' e '*LiveColorGUIuc*' e uma biblioteca de suporte à comunicação com as câmaras designada de '*libunicam*'.

4.1 *LiveColorGUI* e *LiveColorGUIuc*

4.1.1 Descrição

Ambas as aplicações foram desenvolvidas como um substituto para a aplicação existente (*LiveColorConfig*). Possuem uma interface gráfica construída com *GTK* (*Gimp toolkit*), uma biblioteca de funções para construir aplicações gráficas em C. A diferença entre as duas reside no facto que a primeira acede ao *hardware* directamente, enquanto a segunda foi desenvolvida com base na '*libunicam*' (documentada em anexo a esta dissertação). Quando o *LiveColorGUIuc* atingiu um estado estável, passou a ser a única aplicação a ser desenvolvida, pelo que se encontra mais avançada e será a aplicação descrita neste documento.

4.1.2 Bibliotecas utilizadas

SDL

A biblioteca de funções *SDL* (*Simple DirectMedia Layer* - http://www.libsdl.org/index.php) fornece ao programador ferramentas para desenvolver aplicações multimédia em C ou C++ (ou noutras linguagens de programação através de '*wrappers*'). Com o *SDL*, podemo-nos abstrair do sistema operativo e do *hardware* e receber informação do teclado e rato, assim como programar aplicações gráficas 2D (de uma só janela), rotinas de som e aceder ao armazenamento em massa da máquina de uma forma intuitiva e multi-plataforma. Não permite, no entanto, a programação de interfaces para comunicação com o utilizador (com botões, caixas de texto, etc...).

Esta biblioteca foi desenvolvida por Sam Lantinga para ser usada na programação de jogos para *Linux*, mas hoje em dia é usada para todo o tipo de aplicações, em vários sistemas operativos e conta com a ajuda de vários programadores que a desenvolvem e documentam.

GTK+

A biblioteca *GTK* (*Gimp Toolkit* - http://www.gtk.org/) trata-se duma biblioteca para desenhar *GUI*s (*Graphical User Interfaces*), ou seja, permite ao programador desenhar aplicações gráficas através do uso de *widgets* (botões, listas, caixas de texto, menus, etc...). Esta biblioteca, tal como a *SDL*, é multi-plataforma, pelo que código programado para um sistema operativo corre em todos que sejam suportados pela biblioteca com poucas ou nenhumas alterações.

O desenho das interfaces é feito dando à biblioteca informação sobre a posição relativa das *widgets* e esta encarrega-se de 'arrumar' tudo nos espaços correctos. Não podem ser usadas posições absolutas, visto que o tamanho das letras e controlos (*widgets*) variam de um sistema operativo para o outro. Logo, uma aplicação que parecesse 'bem' num sistema, podia aparecer totalmente 'quebrada' noutro se não se usassem posições relativas.

Como o nome indica, o *GTK* começou a ser desenvolvido para a programação da conhecida aplicação de processamento de imagem *Gimp*, sendo hoje desenvolvida por um grande número de pessoas e sendo usada por defeito no também conhecido ambiente gráfico *Gnome*.

4.1.3 Linguagens utilizadas

A linguagem de programação 'C' é uma linguagem por procedimentos, cujo código é estaticamente compilado (traduzido e optimizado para código nativo à máquina) para objectos. Estes objectos são depois '*linkados*', ou seja, são agrupados de forma a constituírem a aplicação final.

Trata-se de uma das linguagens de programação mais em uso de momento, apesar de não suportar o paradigma de programação por objectos. Foi desenvolvida no início dos anos 70 por *Dennis Ritchie* para uso no sistema operativo *UNIX*.

Hoje em dia existe um grande número de compiladores para 'C', tendo sido utilizado no desenvolvimento das aplicações o GCC (*Gnu C Compiler*), que é o compilador usado por defeito em distribuições *GNU/Linux*. Este também serve como *linker*.

4.1.4 Desenvolvimento

Execução linear vs Execução por eventos:

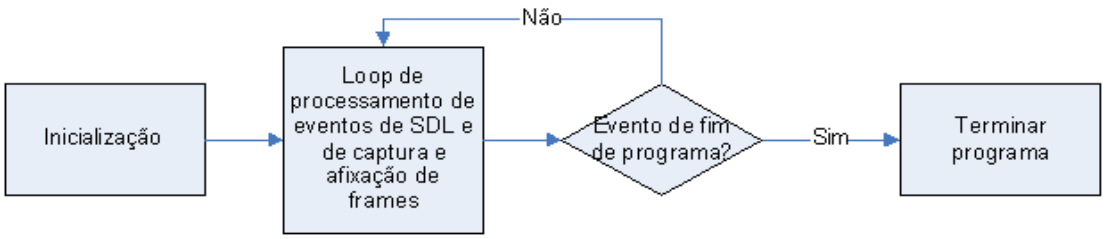

Figura 4.1: Execução linear do *LiveColorConfig.*

Esta aplicação começou a ser desenvolvida tendo por base o código do programa *LiveColorConfig.* Esta tinha um modelo de execução linear (Fig. 4.1), em que o código era inicializado até chegar a um *loop*. Neste *loop* era capturada uma *frame* da câmara, processada e apresentada no ecrã através da biblioteca *SDL*. Também neste *loop* eram processadas quaisquer teclas premidas durante a execução do programa. Quando o programa recebesse o sinal de janela de *SDL* fechada ou de que a tecla 'q' (*Quit*) tinha sido carregada, o *loop* era terminado e o programa acabava.

A biblioteca *GTK* baseia-se num modelo de execução por eventos, ou seja, o código do programa apenas é executado quando ocorre um evento relevante à sua execução. Neste contexto, o primeiro passo ao planear a aplicação, foi substituir o modelo de execução linear, por um modelo de execução por eventos (Fig. 4.2).

O código do *loop* de execução principal foi movido para uma função a ser chamada sempre que o sistema esteja desocupado, ou seja, sempre que não haja outro evento a ser processado. Passou a ter que existir uma função de término do programa, chamada quando o utilizador deseja saír da aplicação, uma vez que deixou de existir um *loop* principal.

Todo o código chamado quando o utilizador prime uma tecla foi movido para funções separadas. Isto deve-se à necessidade de manter a funcionalidade das teclas (para que utilizadores habituados à aplicação antiga possam efectuar a transição facilmente) e ao facto de passarem a haver *widgets* (botões, caixas, etc...) na janela *GTK* que precisam de chamar o mesmo código.

A Figura 4.2 representa o novo modelo de execução por eventos.

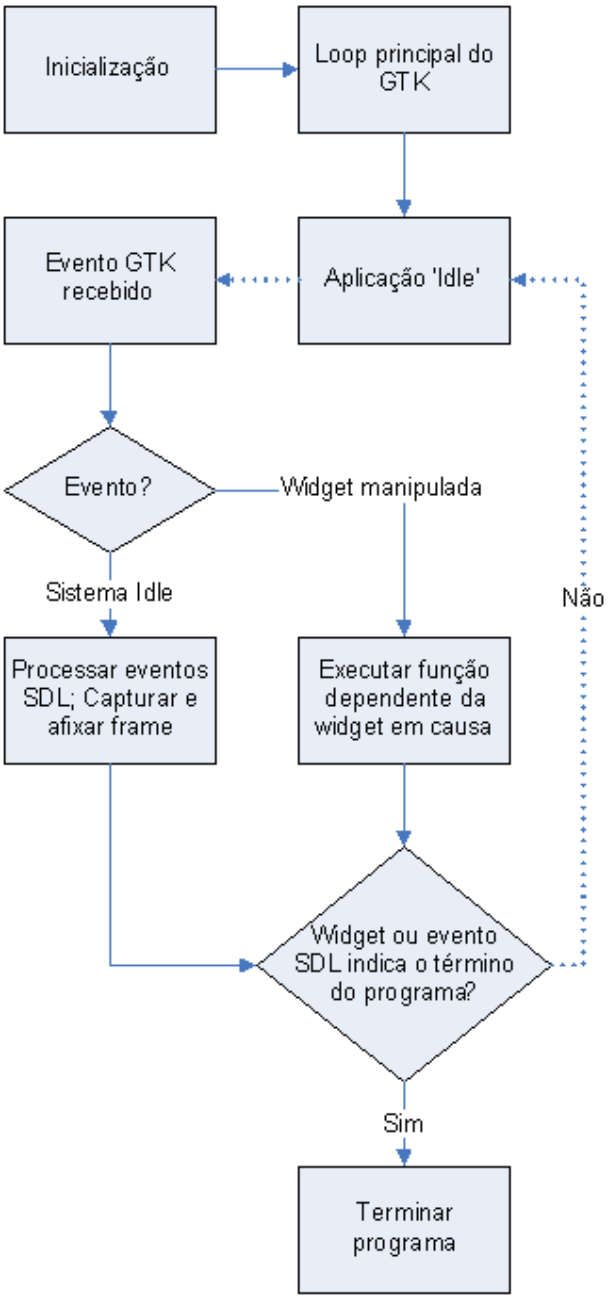

Figura 4.2: Sistema por eventos do *GTK*.

4.1.5 Variáveis globais

A maior parte das funções da aplicação necessitam de partilhar variáveis. A existência de variáveis globais é uma má prática, porque leva ocasionalmente a conflitos entre módulos e pode permitir (dependendo do compilador) que uma fuga de memória escreva por cima de um espaço de

código, tornando o *debugging* muito difícil.

Uma das tarefas que levou mais tempo foi passar todas as variáveis do programa, ou para variáveis locais, ou para uma estrutura. Esta estrutura é passada como argumento a todas as funções do programa que dela necessitem, eliminando assim a necessidade de utilizar variáveis globais.

4.1.6 Primeiras *widgets*

Ao início procedeu-se à implementação de apenas botões na janela principal (Fig. 4.3), sendo o objectivo destes reproduzir a funcionalidade já existente na aplicação original através do uso de teclas. A implementação dos botões serviu sobretudo para familiarização com a *API* da biblioteca *GTK* (como organizar *widgets*, capturar sinais, ...). Numa primeira fase, foram acrescentadas todas as funções passíveis de serem implementadas através de botões.

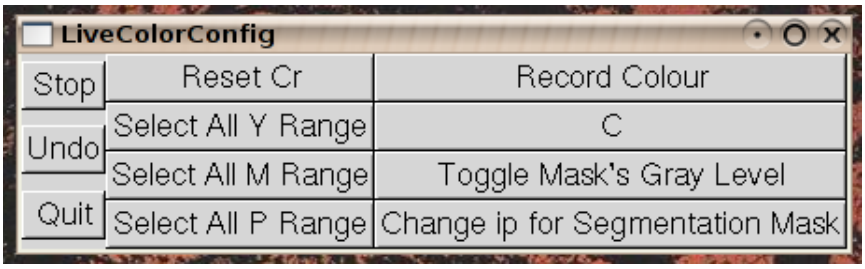

Figura 4.1: Primeira versão do *LiveColorConfig*.

O passo seguinte foi acrescentar um *checkbox* e *scales* (botões deslizantes para seleccionar valores) para se poder ajustar o balanço de brancos da câmara, assim como alternar entre o seu modo manual e automático (Fig. 4.4). Também se acrescentou um conjunto de *labels* que mostram a gama de cor (*YMP*) correntemente seleccionada, e funcionalidade para copiar a mesma para o *clipboard*. Antes desta alteração era sempre necessário recorrer à janela do terminal para se obter esta informação. Uma *combobox* (caixa de escolha) para se poder seleccionar que câmara ligar entre uma lista predefinida de câmaras (identificadas pelo seu nome de dispositivo) foi a última alteração neste estágio. Os primeiros ícones para botões foram acrescentados para dar um aspecto mais *user-friendly* à aplicação. Esta apresenta um aspecto completamente diferente da versão anterior devido a ter passado a ser compilada com a versão 2.x do *GTK*, ao passo que a primeira tinha sido com a versão 1.2.

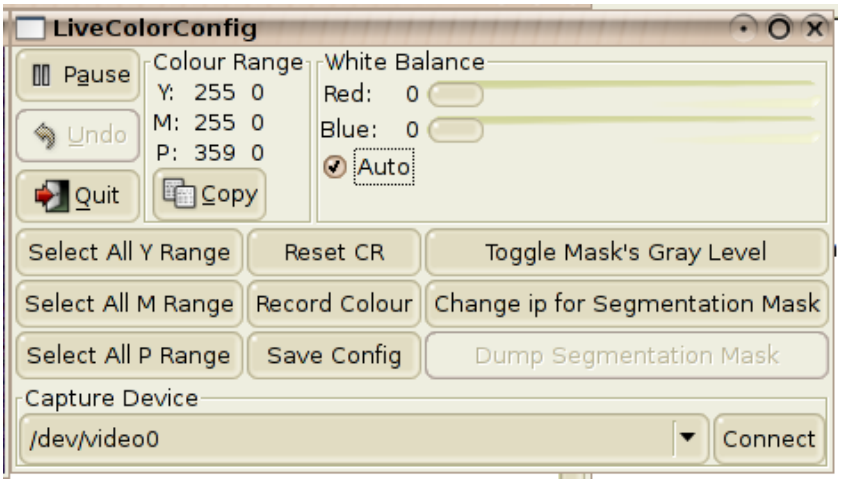

Figura 4.2: *LiveColorConfig* com *GTK*+ 2.0.

4.1.7 *Undo*

O objectivo principal nesta altura era permitir passar a usar a aplicação já existente de um modo mais intuitivo. O primeiro melhoramento em relação à funcionalidade foi a alteração da função '*Undo*'.

Esta função serve para o utilizador poder voltar atrás na sua acção, após ter seleccionado uma cor que não desejaria. A função original apenas permitia voltar atrás uma única selecção, sendo que o estado anterior permanecia guardado numa única variável. A nova função passou a contar com uma lista circular, que permite guardar X estados anteriores (por defeito 30). Quando o utilizador realiza mais do que X selecções de cor, a selecção mais antiga na lista é trocada pela mais recente e deixa de ser possível voltar atrás até à mesma. Esta nova funcionalidade permite que um utilizador possa experimentar selecções que normalmente não faria ou se possa enganar mais sem ter que anular todo o trabalho já efectuado.

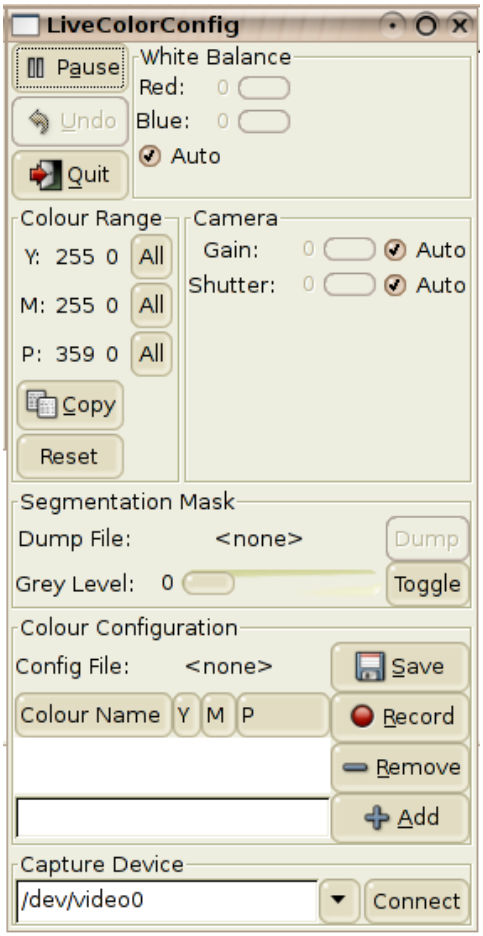

Figura 4.3: Nova versão da aplicação, com a maior parte dos botões removidos.

4.1.8 Máscara de Segmentação (*Segmentation Mask*)

Quando um utilizador selecciona uma gama de cores, esta é representada visualmente na janela de captura através de uma máscara. Na aplicação original esta máscara estava definida como sendo sempre observada através da cor preta ou branca (alterável através de uma tecla ou de um botão de troca). Contudo, existe um problema nesta representação, quando a própria cor que se deseja seleccionar, ou o próprio ambiente, não permitem que a máscara fique visível. A alteração lógica foi poder-se alterar a cor desta, seleccionando um valor dentro da escala cinza (branco a preto, passando pelos vários cinzentos). No entanto, para se poder alterar este valor tornou-se necessário acrescentar um *slider* (barra de ajuste) à aplicação (Fig. 4.5). Manteve-se contudo a função de *toggle* através de um botão. Este varia a cor da máscara para a cor oposta em relação ao valor médio da escala, ou seja, se a cor antes do *toggle* for branca, passa a ser preta e vice-versa, como já acontecia antes.

4.1.9 Mais *widgets*

Para que a nova aplicação fornecesse toda a funcionalidade existente no programa original de um modo visual foi necessário acrescentar mais *widgets* à aplicação. Como explicado no parágrafo anterior, foi adicionada mais uma *scale* para seleccionar o valor de escala cinza da máscara de segmentação. Também foram adicionadas *scales* e *checkboxes* para o controlo do ganho e tempo de exposição (Fig. 4.6). Algumas das funções chamadas através do premir de teclas passaram a depender do *GTK*, para que a janela fosse actualizada correctamente (e.g.: premir a tecla de aumentar ganho, fazer subir a *scale* de ganho).

A nova estrutura da janela passou a não basear-se numa representação das funções através de botões, que até à altura era a mais lógica para reproduzir as funções invocadas através do teclado, mas sim numa divisão mais lógica das funções de acordo com grupos funcionais.

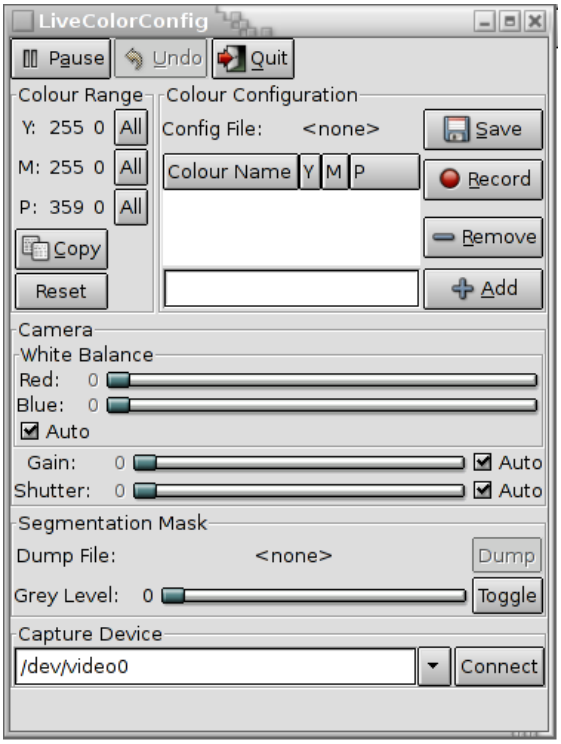

Figura 4.4: Exemplo de um *theme* do *GTK* v2.x.

A alteração mais profunda, foi a do sistema usado para construir os ficheiros de configuração de cores, como se descreve em seguida.

4.1.10 Ficheiros de configuração

Estes ficheiros contêm uma lista de gamas de cor (no espaço *YMP*) que permitem a um robô

identificar objectos. Cada entrada nesta lista consiste no nome da cor e nos seus limites Y,M,P inferiores e superiores.

Os ficheiros de configuração na aplicação original eram construídos do seguinte modo. Existia já uma lista de cores 'embutida' no próprio ficheiro a modificar. O ficheiro era lido pela aplicação e o utilizador ia escolhendo gamas de cores, premindo uma tecla para gravar a cor em memória, gravar a memória para o ficheiro, ou então passar para a cor seguinte.

Embora esta funcionalidade tenha sido preservada pelos motivos já apresentados, foi acrescentada à aplicação uma lista visível de cores que podem ser calibradas (o que levou imenso tempo devido ao modelo usado pelo *GTK* para implementar este tipo de listas visualmente). Tal como na aplicação original, a presente selecção pode ser gravada na lista em memória (*Record*) ou todo o conteúdo da memória gravado em ficheiro (*Save*). Em adição a estas funções já existentes, foram acrescentadas duas novas, *Add* e *Remove*. Estas permitem acrescentar ou remover cores da lista sem que se tenha que manipular o ficheiro de configuração com outro programa (editor de texto). Foi necessário implementar código para manipular as estruturas existentes em memória para isto ser conseguido.

4.1.11 Ajustar a apresentação

Devido à constante implementação de novas funções, tornou-se necessário estar constantemente a alterar a posição das *widgets* dentro da janela da aplicação, para que esta apresentasse sempre um aspecto coerente e de modo a implementar o máximo de funcionalidade no mínimo de espaço ocupado no ecrã, tendo em conta que a janela da aplicação tem que ser apresentada junto com a janela de captura e que não convém ao utilizador que apenas uma delas esteja visível.

Uma das características das aplicações feitas com *GTK* 2.x é adoptarem um tema ou *skin* escolhido pelo utilizador para todo o seu ambiente de trabalho, tornando mais fácil a adaptação à aplicação devido a estar familiarizado com o seu aspecto.

Todos os parâmetros passados à aplicação pela linha de comandos são tomados em conta na altura de desenhar a janela, visto que a aplicação ainda pode ser controlada através destes.

4.1.12 Novas câmaras

A aplicação existente estava preparada para comunicar apenas com câmaras suportadas pelo *driver PWC* (*Philips Webcam*). Como a nova aplicação deveria suportar também câmaras *Firewire* (*IEEE1394*), todas as funções de captura e controlo de parâmetros das câmaras tiveram que ser

reescritas para poderem funcionar com ambos tipos de *drivers* em conjunto (*PWC* e *libdc1394*). Devido à diferente natureza de cada um, foram acrescentadas novas funções para compatibilizar as funções de ambos e permitir ao resto do código ignorar as diferenças.

Com as câmaras compatíveis com o *driver PWC* apenas se capturavam *frames* representadas no formato '*YUV* 4:2:0 Planar'. Estas eram apresentadas na janela *SDL* através duma função de conversão existente na própria biblioteca (*SDLYuvOverlay*). As câmaras *Firewire* codificam as *frames* nos formatos *YUV* 4:4:4, 4:2:2, 4:1:1, todos não planares. Para se poderem exibir todos os tipos de formatos na janela *SDL*, tiveram que ser criadas funções de conversão dos vários formatos *YUV* para *RGB* (24-bits). Por sua vez, os dados já convertidos para *RGB* são escritos directamente no *buffer* gráfico da janela *SDL*.

4.1.13 Selecção da câmara

Até ao momento, era necessário indicar o dispositivo ao qual estava associado a câmara, pela linha de comandos, para se poder seleccionar a câmara a configurar. Não apenas isto, mas um modo de escolher qual câmara *Firewire* a usar nem sequer estava disponível. Sempre que se quisesse seleccionar uma câmara diferente, tinha que se terminar o programa e voltar a chamá-lo. Por outro lado, para seleccionar a resolução (o tamanho das imagens capturadas) e o *framerate* (o ritmo a que cada *frame* é enviada pela câmara) também era necessário passar estes argumentos pela linha de comandos. Para acabar com esta limitação, foi adicionada uma forma gráfica para se controlar estes parâmetros (Fig. 4.7) e o código foi alterado para se poder alterá-los e voltar a ligar à mesma ou outra câmara sem se ter que re/executar o programa.

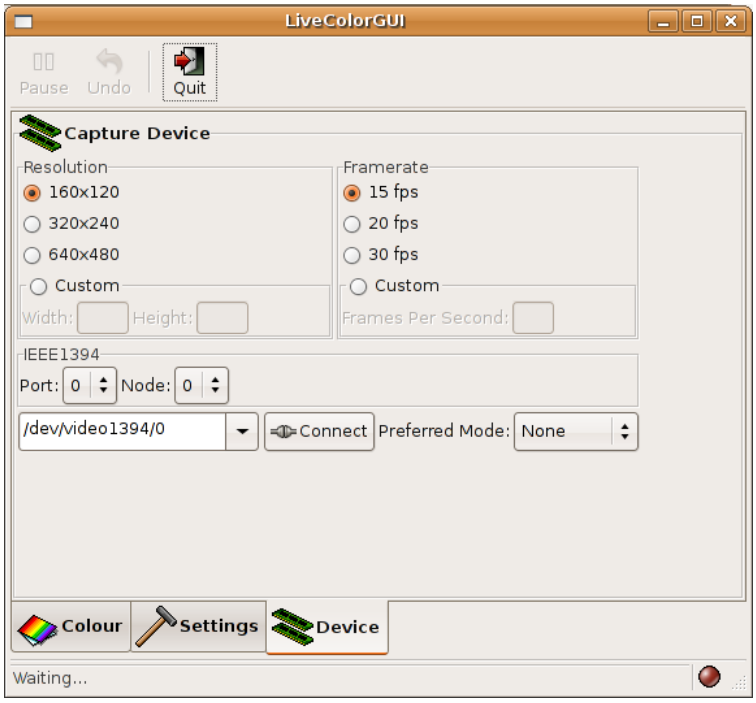

Figura 4.5: Interface de ligação às câmaras.

A maior dificuldade na implementação foi a reinicialização das variáveis e dos *buffers* do programa.

Esta adição tornou a janela demasiado grande pelo que foi necessário voltar a formular a estrutura desta.

4.1.14 Tabulações e Ícones

A solução para encolher a janela foi dividi-la em 3 páginas seleccionáveis por tabulações, sendo uma página para a configuração de cores, uma para o ajuste de parâmetros da câmara e outra para a escolha da câmara a ligar e os seus parâmetros de ligação. A outra opção seria dividir a aplicação por várias janelas, mas isto torná-la-ia demasiado confusa para utilizar confortavelmente.

Foram também acrescentados ícones informativos em cada uma das tabulações e *frames* (caixas, representadas por um rectângulo com um título, que englobam outras *widgets*). Os botões que precisam de ser acedidos com frequência foram colocados numa *toolbar* (barra de botões no cimo da janela). Foi acrescentada uma barra de *status* no fundo da janela para apresentar mensagens informativas, assim como um ícone para informar sobre o estado da ligação da câmara.

4.1.15 Balanço de brancos por *software*

Normalmente, as câmaras dispõem de um ajuste automático das cores vermelha e azul que corrigem a aberração cromática induzida por diferentes tipos de iluminação. O problema é que este ajuste automático pode interferir com a detecção correcta das cores pretendidas.

Para resolver este problema, há que colocar a correcção de brancos da câmara em modo manual, com valores de correcção azul e vermelha a serem definidos pelo utilizador. Para encontrar estes valores já existia uma aplicação disponível (*AdaptiveWB*). O algoritmo desta foi importado para a aplicação, acelerado e melhorado para se poder escolher a área da imagem (que convém ser branca para se obter bons resultados) a usar como referência para a calibração. O algoritmo funciona alterando os valores correctivos de azul, vermelho (e opcionalmente de ganho) até que a cor média dos pontos constituintes da área seleccionada estejam dentro duma gama definida como branco. Quanto mais longe a correcção estiver de alcançar esta cor, mais rápido os valores de azul/vermelho são incrementados/decrementados, para acelerar a correcção.

4.1.16 Divisão do programa em dois

Houve uma altura no desenvolvimento do código do programa em que as funções para controlo e compatibilidade entre câmaras se estavam a tornar demasiado grandes e confusas, principalmente se se voltasse a ter que acrescentar suporte para um novo *driver*. Para solucionar este problema, foi iniciado o desenvolvimento de uma biblioteca de funções dirigida apenas ao controlo das câmaras, captura e conversão de *frames* e de abstracção em relação ao *hardware*. Visto que ao início a biblioteca não estava ainda num estado funcional, foi necessário dividir a aplicação em duas, partindo do mesmo código: uma usando a biblioteca (*LiveColorGUIuc*), que passou a ser a aplicação activamente desenvolvida; e a original que não usa a biblioteca (*LiveColorGUI*), que ficou num estado parado de desenvolvimento, sendo apenas alterada para correcções pontuais, visto que estava a ser usada para a calibração dos robôs no projecto CAMBADA.

Esta dissertação passa agora a descrever apenas o desenvolvimento da aplicação *LiveColorGUIuc*, ignorando a original.

4.1.17 Detecção automática

Com o desenvolvimento da *libunicam*, passou a ser possível detectar as câmaras ligadas ao computador e os seus modos de vídeo disponíveis. Até à altura era necessário escolher o *device* a que estaria ligada a câmara (*USB*) ou então a sua porta e nó (*Firewire*). Também era necessário

escolher a resolução e o modo *YUV* com que a câmara enviaria as *frames* para o computador, caso houvesse vários à escolha. A autodetecção de câmaras e modos de captura veio simplificar e acelerar o tempo de ligação ao *device* pretendido.

As capacidades da câmara também passaram a ser detectadas automaticamente. Assim sendo, o programa apenas apresenta controlos para as propriedades que o *driver* da câmara indique como estando disponíveis. Além disso, o programa agora também conhece os valores mínimos e máximos das mesmas, visto que estes diferem de *driver* para *driver*.

4.1.18 *GDK* vs *SDL*

Uma das desvantagens de usar a biblioteca gráfica *SDL* para a apresentação da captura, é ter que a apresentar numa janela em separado. De modo a poder-se integrar a captura na janela principal, foram estudadas várias opções como integrar a própria janela *SDL* na janela principal, usar *OpenGL*, outras bibliotecas gráficas, ou então o *GDK* (*Gimp Drawing Kit*) que é uma biblioteca interligada com a *GTK*, que a usa por natureza para todas as funções gráficas.

Usar bibliotecas externas foi posto de parte, visto que tornariam o espaço ocupado por dependências maior, o que não é desejável em algo com o espaço para programas tão limitado como num robô.

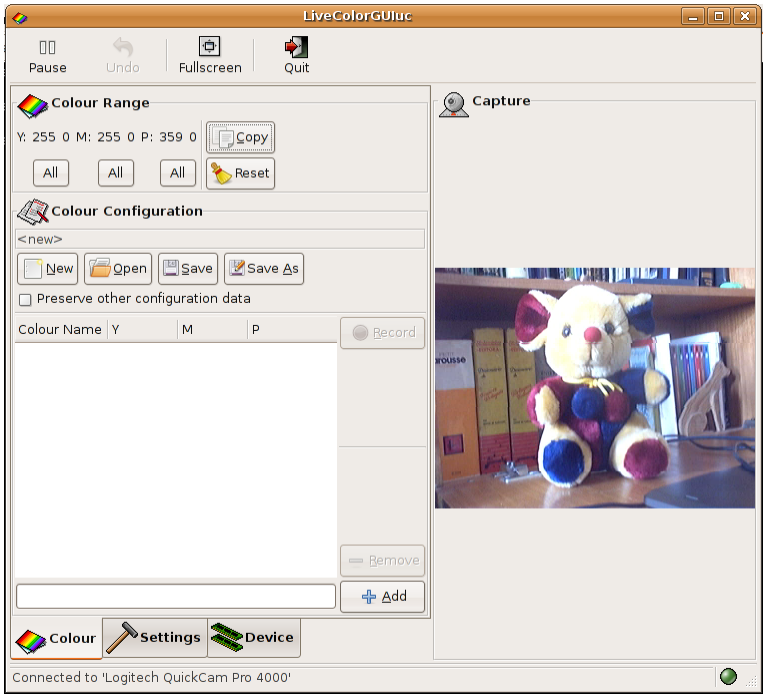

Figura 4.6: *LiveColorGUIuc* com apresentação de captura por *GDK*.

Ao início pensou-se que o uso da biblioteca *GDK* não seria aceitável, porque parecia que não alcançaria *framerate*s suficientemente rápidos para apresentar a captura sem saltar *frames*. Ao implementar as rotinas de desenho, verificou-se que tal não era verdade e que o *GDK* se comporta tão bem como o *SDL* (Fig. 4.8). No entanto, houve alguns desafios de implementação, particularmente na adaptação das rotinas já existentes para funcionar com uma janela *SDL*.

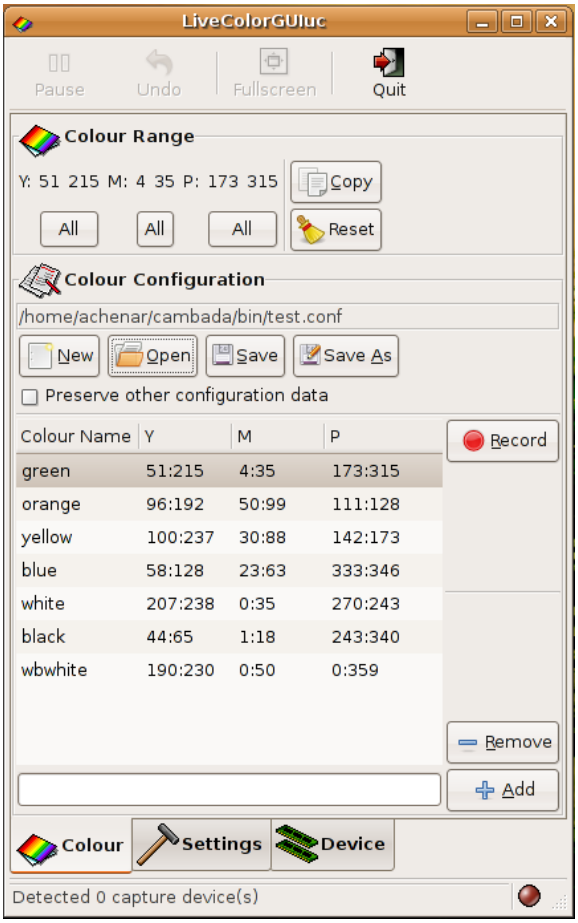

Figura 4.7: Edição de configurações.

Visto que a captura apresentada através do *GDK* se encontra integrada na janela principal é preciso sempre que necessário converter coordenadas da janela principal para o espaço de desenho *GDK*. Em adição a este facto, quando um utilizador, por exemplo, carrega na captura com o rato, mas larga o botão fora da janela (a seleccionar uma zona de calibração de balanço de brancos por exemplo) é necessário que o programa saiba deste evento. Portanto, foi necessário ligar uma rotina intermédia entre o sistema *X-Windows* (o sistema de janelas do *Linux*) e a aplicação, que processa eventos que interessem, mas por defeito não fossem destinados à aplicação.

No entanto, ao usar-se a visualização através de *GDK*, as teclas de atalho, que foram

mantidas desde a versão original do programa, deixam de estar disponíveis. A única maneira para usá-las é usando uma janela *SDL* e seleccionando-a.

Na Figura 4.8 podemos observar que através da biblioteca '*libunicam*' se pode detectar vários dispositivos de captura de imagem ligados ao computador, apresentados para escolha numa *combobox*.

4.1.19 Editor de configurações

Para tornar o programa num editor completo de ficheiros de configuração e evitar que o utilizador tenha que recorrer ao auxílio de programas externos, foram implementadas opções para 'Criar Novo', 'Abrir', 'Gravar' e Gravar Como' às que já tinham sido criadas previamente com vista à manipulação de ficheiros de configuração de cor. Estas novas opções (Fig. 4.9) permitem que o utilizador possa usar apenas este programa para editar todos os seus ficheiros de configuração e evita também que se tenha que passar o nome do ficheiro de configuração a editar pela linha de comandos, o que faz com que se possa configurar várias câmaras sem nunca re-executar o programa, perdendo-se assim menos tempo.

Este editor mantém um registo sobre se foram feitas alterações à configuração em memória e caso o utilizador tente usar uma operação que destrua as modificações sem serem previamente gravadas, apresenta-lhe uma opção para o efeito.

4.1.20 Ficheiro de configuração de parâmetros da câmara

Com vista a ser mais fácil configurar as câmaras com os parâmetros encontrados com o programa, adicionou-se uma opção para exportar as propriedades da câmara para um ficheiro desenhado para o efeito. Este ficheiro pode depois ser importado por outras aplicações.

Descrição do formato do ficheiro:

- id i - onde 'i' é 'f' (câmara frontal) ou 'o' (câmara omnidireccional)
- wb r:b – onde 'r' e 'b' são os valores dos parâmetros de ganho vermelho e azul no balanço de brancos, respectivamente.
- exposure e – onde 'e' é o valor do parâmetro *exposure* (exposição).
- gain g – onde 'g' é o valor do parâmetro *gain* (ganho).
- fps f – onde 'f' são o número de *frames* transmitidas pela câmara por segundo.
- mode m – onde 'm' é o modo de resolução da câmara. Este parâmetro apenas é

válido para câmaras *Firewire*. m = <0 – 160x120 *YUV*444> | <1 – 320x240 *YUV*422> | <3 – 640x480 *YUV*422> | <4 – 640x480 *YUV*411>.

Este ficheiro é gravado como um ficheiro de texto normal e pode ser modificado com qualquer editor.

4.1.21 *Fullscreen*

Para permitir ao utilizador visualizar a captura usando a extensão completa do seu monitor, foi adicionado um modo de visualização em *Fullscreen* (ecrã inteiro). Este modo amplia a imagem, multiplicando as suas dimensões por um número inteiro e escolhendo o modo gráfico disponível em que a imagem já ampliada ocupe mais espaço no mesmo. Multiplicar as dimensões por um inteiro, tornou mais fácil a programação das rotinas de desenho, assim como todas as outras rotinas de manipulação de imagem (selecção de cor, balanço de brancos, etc...).

O algoritmo que escolhe a resolução do ecrã a usar funciona da seguinte forma. Primeiro obtém-se uma lista das resoluções disponíveis. Depois para cada resolução obtém-se um factor de *zoom* (inteiro) máximo que amplie as *frames* obtidas de modo a que estas caibam no ecrã. Para cada resolução então verifica-se a percentagem do ecrã preenchida com o *zoom* das *frames* e a resolução que obtiver maior percentagem é a usada.

Esta opção é útil, quando a resolução da câmara é baixa e se pretende escolher uma cor, visto que actua como '*zoom*'.

4.1.22 *Tooltips*

Para ajudar o utilizador a compreender algumas das opções do programa, algumas *widgets* apresentam uma nota que aparece em *popup* passados uns segundos, caso se mantenha o cursor por cima desta, e que apresenta uma descrição da função. Embora não estejam implementadas para todas as funções do programa (algumas *widgets* não podem ter *tooltips* por particularidades relacionadas com o próprio *GTK*), estas informações fornecem uma ajuda valiosa sem que o utilizador tenha que recorrer ao manual do programa.

4.2 *libunicam*

4.2.1 Descrição

A *libunicam* trata-se de uma biblioteca de rotinas desenhada para interagir com diversos

tipos de câmaras, unificando as diferentes formas de comunicar com elas em apenas uma *API* comum a todas. Esta biblioteca serve como uma camada de abstracção entre as aplicações e os diferentes *drivers*, para que estas não se precisem de preocupar com a manutenção do código de acesso às câmaras.

4.2.2 Detecção de câmaras

Devido aos diferentes tipos de câmaras e às diferentes formas de estabelecer a comunicação com cada uma (por exemplo, nas câmaras *USB* é preciso abrir um *device* enquanto que com as câmaras *Firewire* a comunicação estabelece-se através de uma biblioteca de funções de controlo), foi preciso encontrar um método que permitisse às aplicações ligarem-se às câmaras abstraindo-se de pormenores.

O método encontrado foi fornecer às aplicações uma lista de câmaras compatíveis encontradas (e uma descrição das mesmas) e deixar a aplicação ligar-se a qualquer uma delas apenas através de um número, ou índice na lista de câmaras.

Mesmo assim, por motivos de compatibilidade, mantêm-se funções que permitem aceder a uma câmara através de um *device* (caso *USB*) ou através de uma porta e nó (caso *Firewire*) para mais rápida transição do código das aplicações já existentes.

4.2.3 Ajuste de parâmetros

Devido à diferente natureza de cada câmara, é necessário que a biblioteca informe a aplicação sobre quais os parâmetros ajustáveis na câmara e se a câmara possui funções automáticas, etc... Além disso, também é necessário que as aplicações possam saber a gama de valores válida para cada um dos parâmetros ajustáveis. Logo, a biblioteca dispõe de funções não só de ajuste, mas também informativas para que a aplicação saiba o que pode ou não fazer com a câmara.

Além dos parâmetros básicos comuns à maior parte das câmaras, como, por exemplo, ganho, exposição, contraste..., que são designados na biblioteca por '*features*', também existem por vezes parâmetros que só podem ter dois estados, por exemplo ligar e desligar um LED, que são designados por *switches*. Ainda existem funcionalidades da câmara que podem ser chamadas, como, por exemplo, gravar as suas propriedades na memória interna não volátil: estas são designadas de '*functions*'. A biblioteca implementa funções para detecção e ajuste de todas estas funcionalidades mencionadas.

4.2.4 Captura de *frames*

Como a biblioteca foi desenhada para funcionar com código já existente para o projecto CAMBADA, que trabalha exclusivamente com *frames* em modos *YUV*, toda a captura de *frames* é convertida para o formato *YUV* planar que permita efectuar a conversão sem perdas. Portanto, todos os modos *YUV* são convertidos para o seu correspondente planar se necessário e os modos *RGB* são convertidos para *YUV* 4:4:4 para que não se perca informação de cor.

A captura pode ser efectuada em modo *blocking* ou *polling*. O primeiro espera que a captura da próxima *frame* acabe antes de devolver controlo à aplicação que invocou a função. O segundo devolve imediatamente controlo à aplicação e notifica-a se foi alguma *frame* capturada ou não. Este último modo permite que a aplicação possa trabalhar em outros aspectos enquanto espera pela próxima *frame* ou então permite implementar uma função de *time-out* caso não cheguem *frames*.

4.2.5 Parâmetros automáticos e optimização

Para tornar mais eficiente a comunicação entre as aplicações e as câmaras, a biblioteca mantém tabelas internas com informação sobre os parâmetros das câmaras. Estas servem para acelerar a comunicação quando a aplicação tenta ler um valor da câmara, porque devolvem o valor guardado em memória em vez de o pedirem à câmara, que pode ser lento principalmente quando é feito muitas vezes num curto espaço de tempo. Esta tabela mantém um registo de quando é preciso mesmo ler o valor da câmara ou quando se pode confiar no valor guardado para que não ocorram discrepâncias entre ambos.

Além de servirem para optimização, quando uma das funções da câmara é colocada em modo automático (ganho por exemplo) a biblioteca precisa de saber em que valor colocar a mesma quando voltar ao modo manual e vai procurar este às tabelas. Por exemplo, o *driver* da biblioteca *PWC* exige que se lhe passe o valor '-1' como parâmetro para colocar uma função da câmara em automático. Se a aplicação depois quisesse colocar a câmara em manual evocando 'desligar modo automático', a biblioteca não saberia que valor passar ao *driver* sem as tabelas, porque se tentasse ler algum valor da câmara, leria '-1'.

4.2.6 Implementação de *drivers*

A comunicação com cada *driver* tem as suas próprias funções. A biblioteca define que funções devem ser usadas de acordo com a câmara com a qual estabelece comunicação. De momento existem funções de comunicação implementadas para a comunicação com os *drivers PWC*, genérico *Firewire* e *Video4Linux* v1. Estes permitem a comunicação com a maior parte das

câmaras compatíveis com *Linux*, embora o *driver Video4Linux* não permita controlar todas as funções das câmaras devido à natureza do mesmo, que foi desenvolvido primariamente como um meio de comunicação com placas de captura *TV*, pelo que não permite o ajuste de alguns parâmetros de que estas não dispõem (balanço de brancos por exemplo).

Para implementar a comunicação com um novo *driver* é necessário escrever toda a camada de comunicação com o mesmo, sendo que as funções de optimização e emulação de características das câmaras não precisam de ser alteradas, visto estarem separadas no próprio código da biblioteca.

4.2.7 Funções emuladas por *software*

Algumas câmaras ou *drivers* não dispõem de ajuste de parâmetros importantes, como, por exemplo, ganho ou balanço de brancos. Estes parâmetros podem ser emulados por *software*, sendo que tal despende um maior esforço por parte do *CPU*. A aplicação que comunica com a biblioteca não consegue fazer distinção entre uma função emulada ou uma que esteja presente na própria câmara. Estas funções são implementadas, ajustando cada *frame* por *software* após esta ser capturada.

Apenas para exemplo, encontra-se implementado o balanço de brancos por *software* para câmaras que usem o *driver Video4Linux*, visto que este não dispõe de ajuste do mesmo. Isto permite que alguém com uma câmara que funcione com *V4L* (praticamente todas que funcionam em *Linux*) possa usar o programa *LiveColorGUIuc* e não tenha que recorrer a câmaras mais dispendiosas.

4.3 *LiveColorGUIcs*

4.3.1 Descrição

Esta aplicação consiste numa ligeira modificação do *LiveColorGUIuc*, de modo a que se possam seleccionar cores usando diferentes representações além da *YMP* que é a usada pelos robôs do CAMBADA e portanto usada por defeito para a calibração da visão.

Esta alteração tem como objectivo poder-se testar o comportamento dos vários espaços de cor quando as condições de iluminação são ligeiramente alteradas, para se poder verificar qual o mais resistente às mesmas, ou seja, o que permite que os objectos continuem a ser detectados correctamente.

4.3.2 Desenvolvimento

Na aplicação original, a imagem capturada pelas câmaras encontra-se codificada sempre num espaço *YUV*. Este é em seguida transformado para um espaço *YMP* e todo o processamento é feito com base nesta última representação da imagem.

Nesta aplicação, a imagem em *YUV* é imediatamente transformada para uma representação em *RGB*, que em seguida é convertida para o espaço de cor desejado. Todas as funções dependentes dos espaços *YUV* e *YMP* (apresentação de *frames*, desenho da máscara de segmentação, selecção de cor...) tiveram que ser alteradas para passarem a depender do espaço *RGB* e dinamicamente do espaço de cor seleccionado para teste.

Um problema na conversão de *RGB* para outros espaços reside no facto da bibliografia sobre o assunto por vezes não fazer referência às gamas de valores esperadas antes e após a conversão. Para resolver este assunto, foi desenvolvida uma pequena aplicação que simplesmente percorre toda a gama de valores *RGB* (de 0 a 255), converte a mesma para o espaço de cores pretendido e indica entre que gama ficou o espaço resultante. Este programa permite ajustar os resultados, com pequenas alterações nas funções de transformação, para que estes se encontrem também entre 0 e 255, visto que a aplicação *LiveColorGUIcs* usa *unsigned chars* para armazenar o mapa de cores.

5 Estudo de espaços de cor e segmentação de cor para reconhecimento de objectos em tempo real em aplicações robóticas

5.1 Luminância e Crominância

Normalmente as imagens são adquiridas por um robô utilizando uma câmara e são processadas em tempo real. Essas *frames* são enviadas da câmara para o computador e representadas num espaço de cor como o *RGB* ou *YUV*.

O espaço de cor *RGB* permite representar de modo exacto as propriedades aditivas das componentes vermelha, verde e azul da luz. Uma quantização adequada de cada uma destas 3 componentes pode representar qualquer cor no espectro de luz visível. Enquanto que esta é uma representação natural de cor, não é apropriada para aplicações em que é necessária uma distinção entre a cor e quão clara essa cor é, como em quantização de cor e reconhecimento de objectos.

Para ultrapassar esta limitação e tirar partido da percepção cromática do olho humano para o propósito de compressão de dados, foi criada a representação de cor *YUV*. Neste caso, a cor de um objecto e quão claro este é são componentes distintas: Y é a luminância de uma cor e UV são as suas duas componentes cromáticas. Mas a componente Y é quantificada de acordo com a percepção de cor nas células sensoriais na retina do olho humano. Se prestarmos atenção à formula de conversão de *RGB* para a componente Y, $Y = 0.299 * R + 0.587 * G + 0.114 * B$, podemos constatar claramente (Figuras 5.1 e 5.2) que esta componente contém ainda informação cromática, com efeitos indesejáveis em representação cromática precisa.

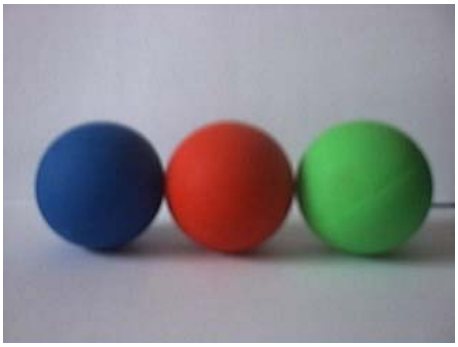

Figura 5.1: *Frame* capturada em *RGB*. Três bolas são mostradas. Azul, vermelha e verde (esquerda para a direita).

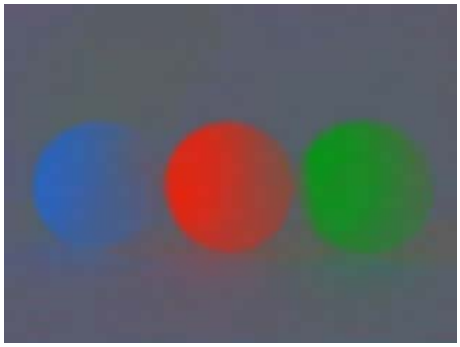

Figura 5.2: Mesma *frame* que 5.1) em *YUV* com uma componente Y alterada e constante.

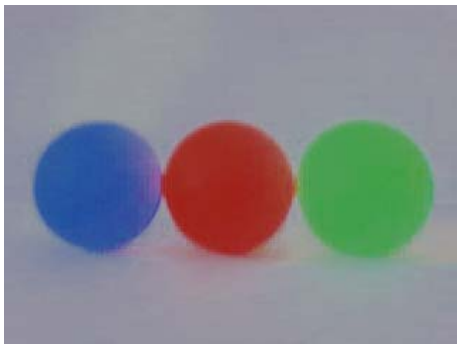

Figura 5.3: *Frame* 5.1) capturada em representação *HSV* com luminância constante (componente 'value').

A Figura 5.1) mostra a *frame* original em *RGB*. A Figura 5.2) é a mesma *frame* representada em *YUV*, onde a componente Y em cada *pixel* foi definida para um valor constante, de modo a que apenas as diferenças nas componentes de crominância UV sejam vistas. Existe claramente um esbatimento na cor das bolas que se mistura com o fundo acromático.

Este facto torna os objectos difíceis de identificar utilizando representações *YUV*. Outras

representações de cores com o mesmo problema são a *YMP* (uma distorção cilíndrica de *YUV*, onde MP são as componentes cromáticas em representação polar), a *YIQ* e a *IHLS* (discutidas mais à frente).

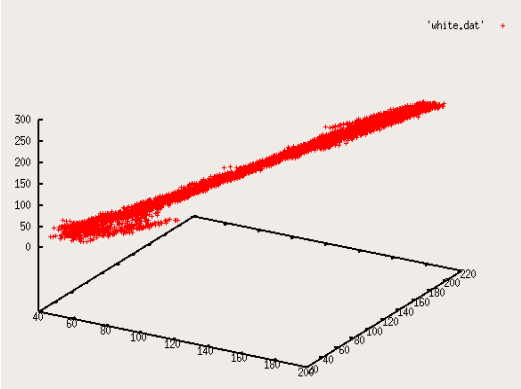

Figura 5.4: Distribuição linear de várias sombras incidentes num objecto branco, representada no espaço *RGB* (gráfico gerado em *gnuplot* com base na imagem dum objecto acromático).

5.2 Saturação e *Hue*

Numa tentativa de separar completamente a luminosidade das outras componentes da cor, foram criados novos espaços de cor. Os três que discutirei nesta dissertação são *HLS*, *HSV* e *IHLS*.

Nestes três espaços, as suas componentes de luminosidade correspondem a uma cor no eixo *RGB* acromático (R=G=B). Esta poderia ser uma vantagem em detecção de cor. Como uma demonstração, algumas amostras *RGB* foram tiradas de um objecto branco com sombras por cima. O resultado é o gráfico observado na Figura 5.4, onde podemos observar que ocorre uma variação linear de branco puro para preto puro, ao longo do eixo acromático.

Isto significa que um objecto acromático permanece acromático (tendo em conta condições optimizadas de luz e correcção de balanço branco) independentemente de quão brilhante está a ser iluminado. O ruído observado no gráfico é provavelmente devido a ruído sensorial na câmara e uma segunda fonte de luz não controlada.

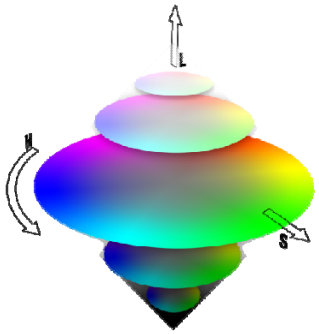

Figura 5.5: Representação bi-cónica do espaço de cor *HLS*.

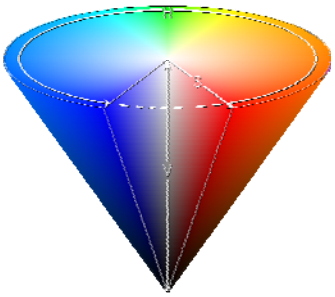

Figura 5.6: Representação cónica do espaço de cor *HSV*.

Na Figura 5.2 podemos ver a representação *HSV* da Figura 5.1 com o seu V (valor que corresponde à luminosidade em *HSV*) modificado para ser constante ao longo da *frame*.

O efeito observado no espaço *YUV*, a mistura de cores com fundo branco não ocorre. Então, quais são as diferenças entre *HSV*, *HLS* e *IHLS* que os tornam distintos? Se olharmos para as figuras 5.5 e 5.6, veremos uma representação dos espaços de cor *HLS* e *HSV* respectivamente.

Devido à sua natureza, há algumas combinações de representações de componentes que não assentam dentro dos próprios espaços. Para ultrapassar este problema, são normalmente projectadas num cilindro. Mas isto põe outro problema. Nos seus extremos de luminosidade (baixa luminosidade em *HSV*, e tanto baixa como alta em *HLS*) a informação de *hue* torna-se irrelevante, e pior, essas cores poderiam tornar-se altamente saturadas (um branco azulado, azul pouco saturado, poderia tornar-se um azul escuro e brilhante, azul altamente saturado) ao converter de *RGB* para *HLS* e ao contrário, por exemplo).

Em relação à Figura 5.9 podemos ver que quando a componente de valor (luminosidade) do *HSV* é novamente alterada para permanecer constante em toda a *frame*, as áreas escuras na Figura 5.7 original têm *hue* aleatório (devido a ruído) e tornam-se altamente saturadas. O mesmo efeito

ocorre em *HLS* (Fig. 5.10), mas desta vez também afecta áreas de grande luminosidade.

Figura 5.7: Cena em representação *RGB*.

Figura 5.8: Mesma cena que 5.7) em *IHLS*.

Figura 5.9: Mesma cena que 5.7) em *HSV*.

O *IHLS,* que foi um melhoramento em relação ao *HLS* proposto por *Allan Hanbury* e *Jean Serra* [27], tem o mesmo problema de componente de luminosidade que o *YUV*, como foi dito antes, e quando a luminosidade das cores é modificada para um valor constante, os mesmos valores saem da sua gama de representação (nem todas as combinações de valores de componentes em *IHLS* têm um significado quando convertidas para um outro espaço de cor).

A Figura 5.10, representada em *HLS* com uma componente de luminosidade constante, tem uma clara distorção (ruído) de *hue* e saturação, quando comparada à Figura 5.7 original em *RGB*. Este efeito é menos proeminente na Figura 5.9), representada em *HSV*, que parece ter esta distorção apenas em áreas de baixa luminosidade, correspondente à componente valor em *HSV*.

Em *IHLS* (Figura 5.8)), croma e luminosidade não são distintamente separadas tal como era esperado. Mas *HSV* tem um inconveniente, uma vez que considera branco como uma cor independente e não um efeito de luminosidade. Diferentes condições de luz e balanço de brancos incorrecto na câmara poderiam severamente distorcer a informação de croma (HS) como foi observado no pano em cima da mesa, que se tornou violeta.

Isto poderia ser resolvido com uma técnica de balanço de brancos adaptativa apropriada, mas de novo, isso poderia distorcer o *hue* ainda mais em condições de pouca luz. Alexandre R.J François e Gérard G.Medioni sugerem [29] que simplesmente se descarte informação de *hue* abaixo de um certo limiar de luminosidade, utilizando um algoritmo que ajusta o limiar de *frame* a *frame* para determinar este valor.

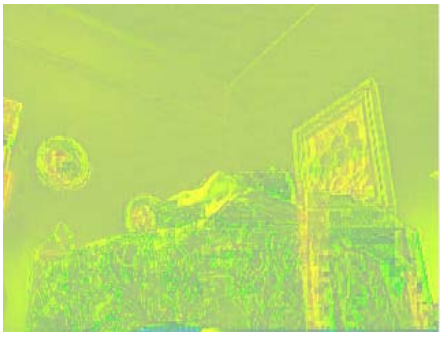

Figura 5.10: Mesma cena que na Fig. 5.7 em *HLS*.

5.3 Reflectividade de objectos e distorção de cores de *pixels*

A cor de um objecto é uma propriedade inerente, consistindo na reflexão de luz vinda de uma ou múltiplas fontes de luz para os sensores do destinatário (numa câmara ou nos olhos humanos, por exemplo). Dois objectos com a mesma cor podem reflectir cada uma das cores primárias (vermelho, verde e azul) em diferentes proporções apesar de parecerem possuir a mesma cor quando vistas a olho nu.

De novo em relação à Figura 5.3 podem ver-se as cores das bolas reflectidas no fundo branco e luz vermelha emitida a partir da bola vermelha para a azul e retransmitida para a câmara (que indica que a bola azul consegue reflectir luz vermelha, pelo que a cor da bola é independente da sua

capacidade de reflectir outras cores), que a percepciona como magenta, composta tanto por azul como por vermelho.

Em condições optimizadas (apenas uma fonte de luz de branco puro e um objecto de azul puro, ignorando o balanço de brancos), o gráfico representado na Figura 5.11) deveria ser uma linha recta ao longo do plano da componente azul e deveria convergir linear e abruptamente para o eixo acromático. Isso não acontece no gráfico (deveria convergir para R = G = B = 0 e R = G = B = 255) devido a luz de ambiente especular existente no ambiente de que foram retirados os dados.

Os dois pontos em que a cor converge para o eixo acromático (ou condições de luz ambiente, segunda fonte de luz ou balanço de brancos sob um ambiente incontrolado) variam de objecto para objecto (testados com diferentes objectos possuindo a mesma cor). A capacidade de identificar o ponto de conversão a níveis de alta luminosidade também varia de objecto para objecto. Não pode ser identificado no gráfico, devido a baixas propriedades de reflexão da própria bola.

O ponto de baixa luminosidade a que a informação de croma se torna irrelevante é facilmente observável.

Considerando a mesma informação em representação *HSV*, a *hue* e saturação da bola seriam bastante precisas, com a excepção de zonas de pouca incidência de luz (daí as duas linhas aparentemente paralelas no gráfico, uma acromática e uma de *hue* azul). A capacidade de um objecto reflectir uma dada componente de cor, poderia também relacionar-se com o ângulo de incidência de luz e o ângulo entre o objecto e o sensor. Para fazer uma analogia, imagine que está a olhar para um lago. Se olhássemos de cima, poderíamos provavelmente ver a cor "verdadeira" do lago (se a água não é límpida como cristais), mas se olhássemos de um ângulo horizontal próximo, poderíamos ver árvores ou o céu reflectidos na sua superfície e por isso, não a sua cor.

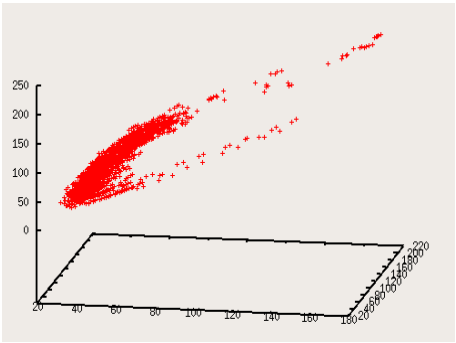

Figura 5.11: Informação dos *pixels* da bola azul representada em *RGB* (gráfico gerado em *gnuplot* através de várias imagens da bola azul).

A reflexão de cor em objectos é difícil de corrigir sem recorrer a reconhecimento de cor

e/ou movimento, que não se encontra na esfera deste estudo e é muito difícil de corrigir adequadamente utilizando apenas reconhecimento de cor. Uma opção seria utilizar visão stereo (duas câmaras paralelas) para tentar distinguir profundidade e correlacionar a informação das duas câmaras para produzir uma imagem sem reflexão.

Os *pixels* numa *frame* poderiam ser distorcidos pela compressão aplicada aos dados enviados pela câmara. Isto poderia resultar em imagens com blocos (especialmente em *standard*s de compressão *JPEG*, *Motion JPEG* e *MPEG*), especialmente em áreas de pouco pormenor. Poderiam haver outras causas (ruído, por exemplo, sensores avariados, refracção de luz no objecto, …) interferindo com a percepção verdadeira da cor de um *pixel* pelo sensor. Este efeito de distorção é mais acentuado em *pixels* adjacentes. Uma vez que poderia haver algo a influenciar a cor do *pixel* que não seja discernível na cena capturada pela câmara, ou demasiada informação para processar, um modelo estatístico poderia ser utilizado para eliminar ou pelo menos reduzir este efeito.

Tal como a reflexão da cor, o defeito da cor introduzido pelo ruído, sensores com problemas de funcionamento e compressão de cor e especialmente interpolação de cor pela câmara, é um problema que interfere com uma precisa segmentação da cor. A interpolação da informação de cor pela câmara leva a problemas de segmentação, uma vez que dois objectos (com cores distintas) poderiam partilhar alguns valores de cor do *pixel*. Informação das *frames* anteriores e /ou um filtro apropriado poderiam ser utilizados para reduzir estes defeitos de cor. A visão stereo poderia mais uma vez ser utilizada para bons efeitos.

5.4 Supressão de sombras baseada em componentes

Esta técnica consiste em ignorar informação numa das componentes da cor (luminosidade, por exemplo) e tentar segmentar os objectos apenas utilizando os restantes dois componentes (componentes de croma, por exemplo).

Se a crominância de um objecto fosse constante, independentemente das sombras (luz fraca) em cima dele, poderia ser possível ignorar a luminosidade, por exemplo, e descartar informação das sombras (supressão de sombras).

Tal como foi visto anteriormente, a transmissão de informação de crominância de objecto para objecto e a perda de informação de cor, devido a alta reflectividade de distorção cromática tanto em áreas de luminosidade muito alta como muito baixa, torna este feito impraticável sem pré-processamento de imagem apropriado, uma vez que a cor de um objecto, e por isso as suas sombras, não são constantes e são por sua vez influenciadas pelo ambiente e pelos parâmetros da

câmara, principalmente correcção de luz branca. Seria ainda possível de conseguir, apesar disso, com pré-processamento de imagem adequado, mas isto implicaria provavelmente alguma detecção de objectos baseada em formas.

5.5 Métodos de segmentação de cor

A segmentação consiste em detectar um objecto baseado na informação anterior que temos sobre o mesmo objecto. Este estudo foca-se na segmentação de objectos apenas por informação de cor, tal como existem outros métodos (forma, movimento, etc. …).

Dois métodos simples utilizados para armazenar cor são a utilização de uma lista de cores (uma lista de valores correspondentes às cores que queremos detectar) e definição de uma gama de cores (uma representação geométrica da informação da cor. Ex.: "todas as cores entre os valores A e B"). A Figura 5.12 representa uma cena onde a bola vermelha (a que está parcialmente escondida pela máscara de segmentação preta) foi seleccionada utilizando uma lista de cores.

Pode-se observar que a bola cor de laranja também foi detectada (*pixels* pretos) apesar de não ser propositado que isso acontecesse. Podemos concluir que tanto a bola vermelha, a bola cor de laranja, ou ambas, têm a sua informação de croma distorcida devido à natureza do próprio espaço *HSV* ou do ambiente. Esta perda de informação de croma acontece normalmente quando não há luz suficiente a iluminar o objecto. O mesmo fenómeno ocorre em espaço *HSL*, com a agravante de ter informação de croma distorcida em objectos altamente iluminados.

Figura 5.12: Segmentação baseada em croma usando uma lista de cores numa *frame HSV*.

Figura 5.13: Mesmo método de segmentação que o da Fig. 5.12 com balanço de brancos automático da câmara.

Referindo-nos à mesma imagem, como um exemplo, torna-se óbvio que a bola cor de laranja está a reflectir menos luz verde para os sensores numa área da sua superfície, que é pobremente iluminada, e que é por sua vez detectada como vermelha. Utilizar uma gama de cores para seleccionar as bolas poderia agravar este problema ainda mais, especialmente num espaço de cor onde croma e luminosidade não estão claramente separadas.

Deste modo, um balanço de brancos adequado reduz ligeiramente a distorção, como é visto na Figura 5.13, mas não se pode ter sempre a certeza de que a correcção do balanço de brancos, mesmo se calibrado manualmente, é preciso durante toda a duração da vida da aplicação robótica (devido a variadas condições de luz no ambiente). Mesmo se quiséssemos anexar um ponto de referência branco (um bocado de papel branco, por exemplo) ao campo de visão da câmara, não poderíamos ter a certeza de que as mesmas condições de luz seriam exactas para a cena inteira (devido a diferentes fontes de luz e reflexão de objectos).

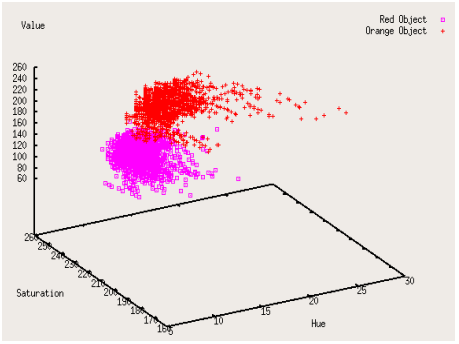

Figura 5.14: Gráfico que representa listas de cores de *pixels* (em espaço *HSV*) tiradas de dois objectos distintos. Intersecção de regiões é observada.

Um método que poderia ser utilizado para a segmentação consistiria em definir uma região de detecção, dentro do espaço de cor, formada por uma lista de cores de *pixel*. Se duas regiões se

interceptarem, (duas cores de objecto a serem segmentadas), a aplicação dentro do robô podia assumir os *pixels* dentro das regiões que se interceptam como dados não válidos, e simplesmente ignorá-los, ou podia assumir ambas as cores como estando presentes no mesmo *pixel* da *frame*.

Assumir os *pixels* como tendo duas (ou mais) cores em vez de ignorá-los, tem a vantagem de não descartar dados que poderiam mostrar-se de valor em detectar correctamente o objecto, especialmente quando a luz é fraca ou as sombras estão presentes no objecto, uma vez que luz fraca leva a regiões que se interceptam (em representações *HSV* & *HLS*).

Outra perspectiva ao tratar intersecções de regiões consistiria em estimar a verdadeira região a que o *pixel* corresponde, estudando a área circundante, mas esse método cairia no domínio do reconhecimento de formas. Essas intercepções da região de segmentação ocorrem facilmente em cores com componente de valor baixa em representação *HSV* (a Figura 5.14 tenta demonstrar este efeito de intercepção) e cores de baixa e alta luminosidade em representação *HLS*. Estas regiões de listas de *pixels* podiam também ser corrigidas de *frame* a *frame* para compensar o incorrecto balanço de brancos ou condições de iluminação variáveis.

Se esse método de segmentação não for adequado ou desejável para a tarefa a realizar, poderia ser criado um modelo estatístico baseado nos valores de *pixels* da *frame* inteira para cortar a informação de *hue* e saturação em espaços *HSV* ou *HLS* abaixo ou acima de um certo limiar e simplesmente ignorar a informação de croma além destes limiares. Isso evitaria detecção incorrecta de croma.

5.6 Conclusões do estudo

Com a falta de duas câmaras para visão stéreo é muito difícil determinar profundidade através de uma única *frame*. Para o conseguir deveria ser criada uma forma de corrigir a distorção de cor dos *pixels* causada através de reflexão, compressão e ruído, baseada unicamente na informação contida na *frame* única. Caso contrário, outra câmara devia ser utilizada. Este tipo de métodos de pré-processamento, junto com a informação de profundidade, melhoraria a capacidade de mais tarde segmentar cores com precisão.

O espaço de cor *HSV* parece ser o melhor espaço de cor a ser usado em aplicações robóticas, uma vez que separa claramente informação de luz e cromática e é a que menos distorce esta última.

Utilizar regiões de cor em vez de listas de cores de *pixels* é preferível para propósitos de segmentação, porque as regiões compensam ligeiramente mudanças das condições de luz enquanto que as listas não, ao passo que o balanço influencia claramente a detecção exacta da cor,

especialmente em condições de luz fraca onde regiões de segmentação normalmente têm uma tendência para se interceptar no espaço *HSV*.

A supressão de sombras baseada apenas em informação de cor é muito difícil de conseguir. A capacidade para executá-la depende fortemente das condições de iluminação e parâmetros de balanço de brancos.

Torna-se claro que mudar para reconhecimento de formas é provavelmente o próximo melhor passo se forem requeridas melhores técnicas de detecção de objectos e supressão de sombras.

6 Comentários e trabalho futuro

6.1 Optimização de funções de conversão

Neste momento, o código do programa que é mais *CPU intensive* é o código que converte entre espaços de cor, ou que efectua transformações nos mesmos. De forma a acelerar a execução da aplicação de forma apreciável, estas funções deveriam ser optimizadas. Tal produziria um impacto em todas as aplicações que dependam das funções, visto que estas se encontram em bibliotecas partilhadas.

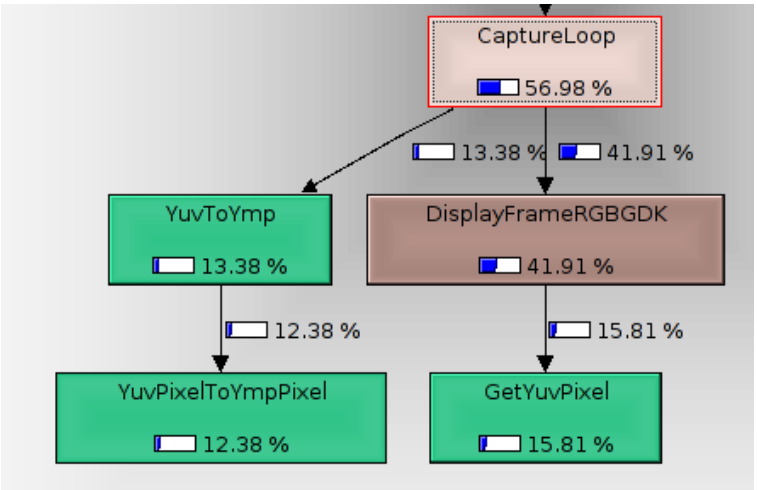

Figura 6.1: Análise do *profile* da função CaptureLoop().

Como se pode verificar através da análise do *profiler* "*Valgrind*" efectuada à função CaptureLoop() (Fig. 6.1), que é a função que usa mais tempo de *CPU* na aplicação *LiveColorGUIuc*, podemos verificar que as funções que necessitam de optimização, visto serem as que de momento necessitam de mais tempo de execução, são as funções de conversão e e manipulação do espaço *YUV*, e a função que apresenta a captura no ecrã (Display*FrameRGBGDK*()). Ao optimizar estas funções, poder-se-ia executar a aplicação em robôs com potência de processamento mais limitada.

6.2 Calibração automática

De momento, os robôs são calibrados antes do jogo e todos os parâmetros da câmara mantêm-se inalterados durante o mesmo. Basta uma ligeira alteração na iluminação para que os robôs deixem de detectar as cores correctamente.

Uma solução para este problema seria colocar uma cor de referência no próprio robô, que estivesse sempre dentro do campo de visão das câmaras, para que estas ajustassem os seus pârametros de modo a essa cor se manter inalterável. Isto levaria supostamente a que todas as outras cores permanecessem constantes.

Um método ligeiramente diferente poderia ser usar a cor verde do próprio campo de jogo como referência, visto que este é constantemente filmado pelas câmaras. Para um correcto uso deste tipo de referência, dever-se-ia ter um algoritmo que decidisse correctamente quais as zonas verdes do campo que não estejam a ser afectadas por sombras de objectos no mesmo, que distorceriam a calibração. Se se conseguisse detectar sombras no campo, também se poderia usar esta informação para corrigir todas as cores que estivessem a ser afectadas pelo mesmo problema, usando a relação entre o verde com e sem sombra como referência.

6.3 Detecção melhorada de objectos

Além da detecção de objectos através da informação de cor, que é usada de momento, estes também podem ser detectados adicionalmente através da forma ou do movimento, ou através duma junção de dois ou todos estes métodos.

Utilizando a detecção através da forma dos objectos, seria mais simples detectar alteração na cor dos mesmos. Um dos problemas que pode ocorrer, trata-se de apenas parte do objecto se encontrar no campo de visão da câmara. Quando tal acontece, poder-se-ia passar outra vez para a detecção apenas através da cor, usando a informação sobre a variação das cores previamente obtida com a junção de ambos os algoritmos, para compensar alterações.

O problema na detecção de movimento consiste do facto da câmara também se encontrar em movimento, pelo que seria necessário compensá-lo em cada imagem obtida. Tal não é trivial visto a velocidade do robô não ser constante e este poder colidir com outros objectos no campo de jogo, o que sem um método de obtenção de posicionamento do robô invalidaria qualquer algoritmo de compensação.

7 Conclusões

O objectivo de conceber uma aplicação com interface gráfica com vista à calibração de cores e parâmetros de câmara em aplicações de visão robótica foi atingido com sucesso.

A biblioteca elaborada para comunicação com câmaras vai permitir um desenvolvimento mais célere de aplicações que precisem de funcionalidade de captura de vídeo.

Com base no estudo efectuado, recomenda-se a utilização do espaço de cores *HSV* (como já foi mencionado) para uma melhor detecção de objectos baseada em informação de cor.

A Manual do *LiveColorGUI*

Pause

Pausa ou recomeça a captura de *frames*. Apenas é possível seleccionar gamas de cores quando a captura está parada.

Undo

Volta atrás na decisão de seleccionar cores. Pode-se, no máximo, reverter 30 escolhas de cor passadas.

Fullscreen

Permite visualizar a captura utilizando todo o ecrã, em vez de ser apenas numa janela.

Quit

Sair do programa.

Página '*Colour*'

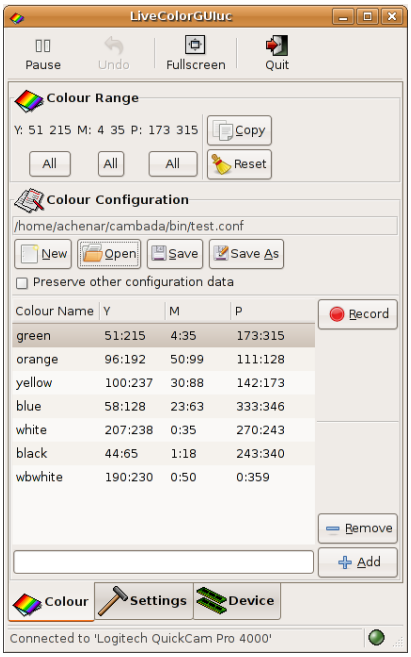

Figura A.1: Página '*Colour*'.

Copy

Copia a presente gama de cores para o '*clipboard*', para que se possa colar o conteúdo com outra aplicação.

Reset

De-selecciona todas as cores, ou seja, reinicializa a gama de cores.

All

Selecciona toda a gama dentro das propriedades Y, M ou P.

New

Cria um novo ficheiro de configuração de cores em memória.

Open

Abre um ficheiro de configuração de cores e lê o seu conteúdo para a memória.

Save As

Grava o conteúdo em memória sobre gamas de cores num ficheiro a escolher pelo utilizador.

Save

Grava o conteúdo em memória sobre gamas de cores no ficheiro já aberto.

Preserve other configuration data

Escolhe se se grava apenas a informação de cor no ficheiro e se ignora tudo o resto que lá exista (informação de sensores, etc...) ou se se mantém toda a outra informação.

Record

Grava a gama de cores presentemente escolhida, na entrada da tabela seleccionada.

Remove

Remove a entrada de cor seleccionada na tabela.

Add

Adiciona uma nova cor à tabela, cujo o nome será escrito na caixa à esquerda deste botão.

Página '*Settings*'

White Balance

As barras *'Red'* e *'Blue'* ajustam o ganho vermelho e azul respectivamente do balanço de brancos quando este está em modo manual. A caixa *'Auto'* alterna entre o modo automático e manual do balanço de brancos.

Calibrate

Calibra o balanço de brancos por *software*. Quando este botão está seleccionado é possível carregar na captura e arrastar o rato para seleccionar a área a utilizar para a calibração.

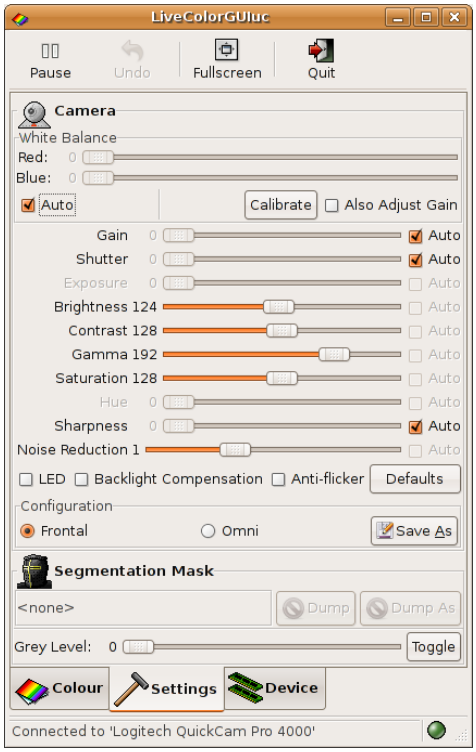

Figura A.2: Página '*Settings*.

Also adjust gain

Permite que a calibração por *software* de balanço de brancos também ajuste o ganho.

Gain

Ajuste manual do ganho da câmara. A caixa 'Auto' alterna entre modo manual e automático quando disponível.

Shutter

Ajuste manual da velocidade do diafragma da câmara. A caixa 'Auto' alterna entre modo

manual e automático, quando disponível.

Exposure

Ajuste manual do tempo de exposição da câmara. A caixa *'Auto'* alterna entre modo manual e automático quando disponível.

Brightness

Ajuste manual da luminosidade da câmara. A caixa *'Auto'* alterna entre modo manual e automático quando disponível.

Contrast

Ajuste manual do contraste da câmara. A caixa *'Auto'* alterna entre modo manual e automático quando disponível.

Gamma

Ajuste manual da correcção gama da câmara. A caixa *'Auto'* alterna entre modo manual e automático quando disponível.

Saturation

Ajuste manual do saturação da câmara. A caixa *'Auto'* alterna entre modo manual e automático quando disponível.

Hue

Ajuste manual da crominância da câmara. A caixa *'Auto'* alterna entre modo manual e automático quando disponível.

Sharpness

Ajuste manual do 'esbatimento' da câmara. A caixa *'Auto'* alterna entre modo manual e automático quando disponível.

Noise reduction

Ajuste manual da redução de ruído da câmara. A caixa *'Auto'* alterna entre modo manual e automático quando disponível.

LED

Ligar ou desligar o *LED* da câmara.

Backlight Compensation

Compensar a imagem quando se encontra uma fonte luminosa forte por detrás do objecto

filmado.

Anti-flicker

Reduzir o *flickering* nas *frames* capturadas, ou seja, reduz o efeito pelo qual parece que as imagens estão a 'piscar'.

Defaults

Restaurar os valores de fábrica para as definições da câmara.

Configuration

Configurar se a câmara é frontal ou omni-direccional e grava os valores presentemente definidos nas propriedades da câmara num ficheiro através do botão '*Save As*'.

Segmentation Mask

As opções *'Dump'* e *'Dump As'* gravam as coordenadas de todos os pontos do ecrã que caiem dentro da gama de cores presentemente seleccionada. A diferença entre estas duas opções é que uma grava no ficheiro de *dump* correntemente seleccionado e a outra pede o nome do ficheiro ao utilizador.

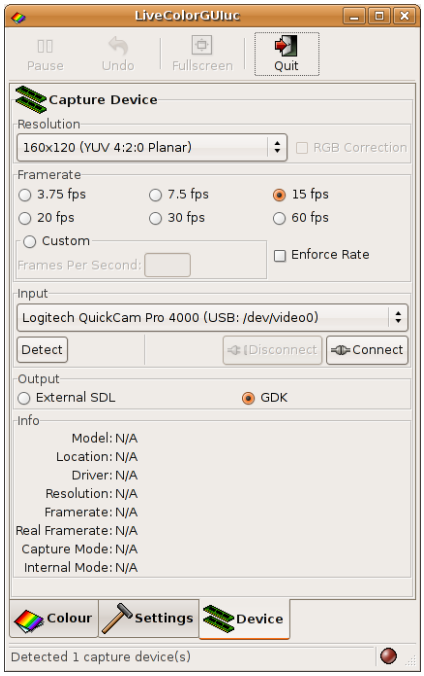

Figura A.3: Página '*Device*'.

A opção *grey level* permite ajustar o nível de cinzento que é apresentado na captura para representar os pontos que estão dentro da presente gama de cores. A opção '*Toggle*' varia

radicalmente este nível de cinzento (por exemplo preto para branco e vice-versa).

Página *'Device'*

Resolution

Permitir escolher a resolução das *frames* capturadas e o modo gráfico (eg: *YUV*) como as estas são armazenadas em memória. As resoluções apresentadas são baseadas na presente câmara detectada. (ver '*Input*')

RGB Correction

Alguns *drivers* identificam erroneamente os modos BGR como sendo *RGB*. Esta função troca os valores de vermelho e azul em memória para corrigir as *frames*.

Framerate

Escolher quantas *frames* por segundo serão enviadas pela câmara. A opção '*Custom*' permite seleccionar um valor não previsto pela aplicação.

Enforce Rate

Normalmente, a própria câmara encarrega-se de enviar o número certo de *frames* segundo o pedido. Quando tal não acontece, esta opção pode ser usada para que a própria aplicação peça o número certo de *frames* por segundo à câmara em vez de deixa-la controlar este parâmetro.

Input

Esta caixa permite seleccionar a qual câmara detectada a aplicação se deve ligar, carregando em seguida no botão *'Connect'*. Quando se selecciona uma câmara, as resoluções disponíveis para escolha são modificadas de acordo com as capacidades da câmara.

A opção *'Detect'* volta a procurar câmaras ligadas à máquina. A opção *'Disconnect'* faz com que a aplicação se desligue da câmara à qual se encontra presentemente ligada.

Output

Escolher como deve ser apresentada a captura. A opção '*SDL*' apresenta a captura numa janela (*SDL*) separada da aplicação principal. A opção '*GDK*' apresenta a captura integrada na aplicação principal.

Info

Esta caixa é apenas informativa.

Modelo – Nome da câmara.

Location – *Device* ou porta e nó onde a câmara se encontra representada no sistema operativo.

Driver – *Driver* interno da *libunicam* que está a ser usado.

Resolution – Resolução da captura.

Framerate – *Framerate* pedido.

Real Framerate – *Framerate* conseguido.

Capture Mode – Modo gráfico como as *frames* são enviadas para a aplicação.

Internal Mode – Modo gráfico com o qual as *frames* são armazenadas e processadas pela aplicação.

Teclas de atalho na janela *SDL*

Captura a decorrer:

Y – Seleccionar toda a gama Y (luminosidade)

M – Seleccionar toda a gama M

P – Seleccionar toda a gama P

S – Parar captura

G – *Toggle* do nível de cinzento da máscara de segmentação

R – Fazer *reset* à gama de cores

ESC – Sair do modo de ecrã inteiro

Q – Terminar aplicação

K – Gravar presente gama de cores na lista em memória

C – Gravar configuração de cores

N – Passar para a próxima entrada da lista de cores em memória

E – Aumentar ganho da câmara

D – Diminuir ganho da câmara

A – Alternar entre modos de ganho automático e manual da câmara

Z – Alternar entre modos de *shutter* automático e manual da câmara

V – Aumentar tempo de exposição da câmara

B – Diminuir tempo de exposição da câmara

Captura parada:

S – Resumir captura

U – *Undo* (voltar atrás nas últimas selecções de cor)

W – Escrever *dump* da máscara de segmentação em ficheiro

Q – Terminar a aplicação

G – *Toggle* do nível de cinzento da máscara de segmentação

B *libunicam API* Reference

Functions

int uc_cam_detect_all(uc_cam_info **ci);

int uc_cam_open_detected(uc_cam *cam, uc_cam_info *ci, int idx);
int uc_cam_open_USB(uc_cam *cam, char *device);
int uc_cam_open_Firewire(uc_cam *cam, int port, int node);
int uc_cam_close(uc_cam *cam);

int uc_cam_detect_modes(uc_cam *cam, uc_cam_mode **cammodes);
int uc_cam_max_resolution(uc_cam *cam, int *width, int *height);
int uc_cam_min_resolution(uc_cam *cam, int *width, int *height);
int uc_cam_setup_capture(uc_cam *cam, int width, int height, float fps, int prefmode);
int uc_cam_capture(uc_cam *cam, uChar *buf);
int uc_cam_capture_poll(uc_cam *cam, uChar *buf);

int uc_cam_has_feature(uc_cam *cam, int feat);
int uc_cam_set_feature(uc_cam *cam, int feat, int value);
int uc_cam_get_feature(uc_cam *cam, int feat, int *value);
int uc_cam_set_wb(uc_cam *cam, int blue, int red);
int uc_cam_get_wb(uc_cam *cam, int *blue, int *red);
int uc_cam_has_auto(uc_cam *cam, int feat);
int uc_cam_set_auto(uc_cam *cam, int feat, int state);
int uc_cam_get_auto(uc_cam *cam, int feat, int *state);
int uc_cam_feature_min(uc_cam *cam, int feat);
int uc_cam_feature_max(uc_cam *cam, int feat);

int uc_cam_has_switch(uc_cam *cam, int sw);
int uc_cam_set_switch(uc_cam *cam, int sw, int state);
int uc_cam_get_switch(uc_cam *cam, int sw, int *state);

int uc_cam_has_function(uc_cam *cam, int func);
int uc_cam_call_function(uc_cam *cam, int func);

char *uc_feature_name(int feat);
char *uc_driver_name(int driver);
char *uc_mode_name(int mode);
Drivers
Errors

Structures

uc_cam
uc_cam_info
uc_cam_mode
Buffer modes
Camera features
Camera switches
Camera functions
Software emulated features

Introduction

The goal of this library is allowing to have a common programming interface to interact with cameras that use different drivers and/or APIs. In addiction to this primary goal the library should convert different RGB/YUV formats to those required by the user.

Functions

int uc_cam_detect_all(uc_cam_info **ci);

Scans devices from /dev/video0 to /dev/video63 for USB and Parallel cameras and scans IEEE1394 ports and nodes for Firewire cameras. Returns found cameras.

Returns:

An allocated and initialized array of 'uc_cam_info' in 'ci' and the number of cameras detected or an error code.

Example:

```
int cam_cnt;
uc_cam_info *ci;

cam_cnt = uc_cam_detect_all(&ci)
...
free(ci);
```

int uc_cam_open_detected(uc_cam *cam, uc_cam_info *ci, int idx);

Opens a detected webcam from a initialized 'uc_cam_info *ci' array and an index 'idx' into the array.

Returns:

UC_ERR_NONE or error code.

Example:

```
int cam_cnt;
uc_cam_info *ci;
uc_cam cam;

cam_cnt = uc_cam_detect_all(&ci)
if(cam_cnt > 0)
    uc_cam_open_detected(&cam, ci, 0);
free(ci);
```

int uc_cam_open_USB(uc_cam *cam, char *device);

Opens a USB webcam from a given 'device' name. Automatically detects which driver to use.

Returns:

UC_ERR_NONE or error code.

Example:

```
uc_cam cam;

uc_cam_open_USB(&cam, "/dev/video0");
```

int uc_cam_open_Firewire(uc_cam *cam, int port, int node);

Opens a Firewire webcam from a given 'port' and 'node' location.

Returns:

UC_ERR_NONE or error code.

Example:

```
uc_cam cam;

uc_cam_open_Firewire(&cam, 0, 0);
```

int uc_cam_close(uc_cam *cam);

Shuts down a camera and closes its handler and related buffers.

Returns:

UC_ERR_NONE or error code.

Example:

```
uc_cam cam;

uc_cam_open_Firewire(&cam, 0, 0);
uc_cam_close(&cam);
```

int uc_cam_detect_modes(uc_cam *cam, uc_cam_mode **cammodes);

Detects available capture modes available for this camera.

Returns:

An allocated list of 'uc_cam_mode' in 'cammodes' and the number of modes available or error code.

Example:

```
uc_cam cam;
uc_cam_mode *cm;
int mode_cnt;

uc_cam_open_Firewire(&cam, 0, 0);
mode_cnt = uc_cam_detect_modes(&cam, &cm);
...
free(cm);
uc_cam_close(&cam);
```

int uc_cam_max_resolution(uc_cam *cam, int *width, int *height);

Detects the biggest resolution the camera supports.

Returns:

Resolution 'width' and 'height'. UC_ERR_NONE or error code.

Example:

```
int w, h;


uc_cam_max_resolution(&cam, &w, &h);
```

int uc_cam_min_resolution(uc_cam *cam, int *width, int *height);

Detects the smallest resolution the camera supports.

Returns:

Resolution 'width' and 'height'. UC_ERR_NONE or error code.

Example:

```
int w, h;


uc_cam_min_resolution(&cam, &w, &h);
```

int uc_cam_setup_capture(uc_cam *cam, int width, int height, float fps, int prefmode);

Sets up video capture. The resolution is described in 'width' and 'height'. The framerate is described in 'fps'. 'prefmode' is a preferencial capture buffer mode the user wants. To check if the actual mode acquired is what you want check 'cam->capmode' afterwards.

Returns:

UC_ERR_NONE or error code.

Example:

```
uc_cam_setup_capture(&cam, 640, 480, 15.0, UC_NOMODE);
```

int uc_cam_capture(uc_cam *cam, uChar *buf);

Read below.

int uc_cam_capture_poll(uc_cam *cam, uChar *buf);

Captures a frame and converts it to the nearest planar YUV mode (can be obtained in 'cam->yuvmode'). Copies the frame to 'buf'. The difference between the two functions is that one blocks program execution and the other returns immediatly (poll).

Returns:

UC_ERR_NONE, UC_ERR_NO*FRAME* (poll only) or error code.

Example:

```
uc_cam_capture(&cam, vidbuf);
```

int uc_cam_has_feature(uc_cam *cam, int feat);

Checks if a camera has a given feature described by 'feat'.

Returns:

UC_TRUE, UC_FALSE or error code.

Example:

```
if(uc_cam_has_feature(&cam, UC_GAIN) == UC_TRUE)
    uc_cam_set_feature(&cam, UC_GAIN, 128);

/* or */

if(uc_cam_has_feature(&cam, UC_GAIN) > 0)
    uc_cam_set_feature(&cam, UC_GAIN, 128);
```

int uc_cam_set_feature(uc_cam *cam, int feat, int value);

Sets a camera feature described by 'feat' to a given 'value'. Don't use this function to set *white balance*. Auto mode is disabled.

Returns:

UC_ERR_NONE or error code.

Example:

```
uc_cam_set_feature(&cam, UC_EXPOSURE, 128);
```

int uc_cam_get_feature(uc_cam *cam, int feat, int *value);

Gets the current 'value' of a camera feature described by 'feat'. This function uses internal *look*up tables when possible to avoid flooding the communications bus.

Returns:

Feature value in 'value'. UC_ERR_NONE or error code.

Example:

```
int val;
uc_cam_get_feature(&cam, UC_GAMMA, &val);
```

int uc_cam_set_wb(uc_cam *cam, int blue, int red);

Sets *white balance* 'blue' and 'red' parameters. Auto mode is disabled.

Returns:

UC_ERR_NONE or error code.

Example:

```
uc_cam_set_wb(&cam, 128, 128);
```

int uc_cam_get_wb(uc_cam *cam, int *blue, int *red);

Gets *white balance* 'blue' and 'red' parameters.

Returns:

White balance in 'blue' and 'red'. UC_ERR_NONE or error code.

Example:

```
int b, r;

uc_cam_get_wb(&cam, &b, &r);
```

int uc_cam_has_auto(uc_cam *cam, int feat);

Checks if a camera feature in 'feat' has an automatic control mode available.

Returns:

UC_TRUE, UC_FALSE or error code.

Example:

```
if(uc_cam_has_auto(&cam, FEATURE_SHARPNESS) == UC_TRUE)
    uc_cam_set_auto(&cam, FEATURE_SHARPNESS, UC_ON);

/* or */

if(uc_cam_has_auto(&cam, FEATURE_SHARPNESS) > 0)
    uc_cam_set_auto(&cam, FEATURE_SHARPNESS, UC_ON);
```

int uc_cam_set_auto(uc_cam *cam, int feat, int state);

Turns the automatic mode of a feature on/off.

Returns:

UC_ERR_NONE or error code.

Example:

```
uc_cam_set_auto(&cam, FEATURE_SHUTTER, UC_OFF);
```

int uc_cam_set_auto(uc_cam *cam, int feat, int *state);

Gets the state of the automatic mode of a feature.

Returns:

UC_ON or UC_OFF in 'state'. UC_ERR_NONE or error code.

Example:

```
int st;
uc_cam_get_auto(&cam, FEATURE_BRIGHTNESS, &st);
```

int uc_cam_feature_min(uc_cam *cam, int feat);

Gets the minimum possible value of a feature.

Returns:

Minimum feature value or error code.

Example:

int mingain;

mingain = uc_cam_feature_min(&cam, FEATURE_GAIN);

int uc_cam_feature_max(uc_cam *cam, int feat);

Gets the maximum possible value of a feature.

Returns:

Maximum feature value or error code.

Example:

int max*gain*;

max*gain* = uc_cam_feature_max(&cam, FEATURE_*GAIN*);

int uc_cam_has_switch(uc_cam *cam, int sw);

Checks if a camera has a given switch described by 'sw'.

Returns:

UC_TRUE, UC_FALSE or error code.

Example:

```
if(uc_cam_has_switch(&cam, UC_LED) == UC_TRUE)
    uc_cam_set_switch(&cam, UC_LED, UC_ON);

/* or */

if(uc_cam_has_switch(&cam, UC_LED) > 0)
    uc_cam_set_switch(&cam, UC_LED, UC_ON);
```

int uc_cam_set_switch(uc_cam *cam, int sw, int state);

Turns a camera switch on/off.

Returns:

UC_ERR_NONE or error code.

Example:

uc_cam_set_switch(&cam, UC_BACKLIGHT, UC_OFF);

int uc_cam_get_switch(uc_cam *cam, int sw, int *state);

Gets the state of a camera switch on/off.

Returns:

Switch 'state'. UC_ERR_NONE or error code.

Example:

int powerstate;

powerstate = uc_cam_get_switch(&cam, UC_POWER);

int uc_cam_has_function(uc_cam *cam, int func);

Checks if a camera has a given function described by 'func'.

Returns:

UC_TRUE, UC_FALSE or error code.

Example:

```
if(uc_cam_has_function(&cam, UC_LOADMEM) == UC_TRUE)
    uc_cam_call_function(&cam, UC_LOADMEM, UC_ON);

/* or */

if(uc_cam_has_function(&cam, UC_LOADMEM) > 0)
    uc_cam_call_function(&cam, UC_LOADMEM, UC_ON);
```

int uc_cam_call_function(uc_cam *cam, int func);

Invokes a camera function.

Returns:

UC_ERR_NONE or error code.

Example:

```
uc_cam_call_function(&cam, UC_FACTORYDEFAULTS);
```

char *uc_feature_name(int feat);

See below.

char *uc_driver_name(int driver);

See below.

char *uc_mode_name(int mode);

Retrives a textual representation of a given feature, driver or buffer mode.

Returns:

Pointer to a string.

Example:

```
printf("Capture buffer mode is %s\n", uc_mode_name(cam.capmode));
```

Drivers

Driver code	Driver description
UC_DRIVER_PWC	Philips Webcam driver
UC_DRIVER_Firewire	libdc1394 API
UC_DRIVER_V4L	Generic Video4Linux driver

Errors

All error codes are negative numbers. TODO: Explain each one...

Structures

uc_cam

```
/* Camera structure, used for camera operations */
typedef struct {
	int driver;
	int width, height;
	float fps;
	int capmode;
	YuvModes yuvmode;
	/* Private variables - DON'T ACCESS DIRECTLY */
	int hasfeatknown[UC_FEATURE_COUNT];
	int hasfeat[UC_FEATURE_COUNT];
	int featvalknown[UC_FEATURE_COUNT];	/* Speedup value reading */
	int featval[UC_FEATURE_COUNT];  /* " " " */
	int featautoknown[UC_FEATURE_COUNT];	/* " " " */
	int featauto[UC_FEATURE_COUNT]; /* " " " */
	int featissw[UC_FEATURE_COUNT]; /* Is feature software emulated? */
	struct {
		int red, blue;
	} wb;
	int capturing;
	/* V4L */
	uChar *vidbuf;
	/* USB */
	int fd;
	/* Firewire */
	raw1394handle_t fw_handle;
	dc1394_cameracapture fw_camcap;
	int fw_node, fw_power;
} uc_cam;
```

uc_cam_info

```
/* Camera info structure, for use with auto-detection */
typedef struct {
```

```
        int driver;
        char modelname[UC_MAX_MODELNAME_SIZE];
        /* USB */
        char device[UC_MAX_DEV_SIZE];
        /* Firewire */
        int fw_port, fw_node;
} uc_cam_info;

uc_cam_mode
/* Camera capture resolution mode structure */
typedef struct {
        int width;
        int height;
        int capmode;
} uc_cam_mode;
```

Buffer modes

Mode code	**Mode description**
UC_NOMODE	No mode
UC_YUV411	YUV 4:1:1
UC_YUV420	YUV 4:2:0
UC_YUV422	YUV 4:2:2
UC_YUV444	YUV 4:4:4
UC_YUV420P	YUV 4:2:0 Planar
UC_RGB24	RGB 24 bits
UC_BGR24	BGR 24 bits

Camera *features*

Feature code	**Feature description**
UC_GAIN	Gain
UC_SHUTTER	Shutter mode or speed
UC_EXPOSURE	Colour exposure
UC_BRIGHTNESS	Brightness
UC_CONTRAST	Contrast
UC_GAMMA	Gamma correction

UC_SATURATION	Colour saturation
UC_HUE	Colour hue
UC_SHARPNESS	Image sharpness
UC_NOISEREDUCTION	Noise reduction
UC_WHITEBALANCE	White balance
UC_FRAMERATE	Framerate

Camera switches

Switch code	**Switch description**
UC_LED	Camera LED
UC_POWER	Camera power
UC_BACKLIGHT	Backlight compensation
UC_ANTIFLICKER	Anti-flicker mode

Camera functions

Function code	**Function description**
UC_FACTORYDEFAULTS	Restore factory default Settings
UC_LOADMEM	Load user Settings from camera memory

Software emulated features

These are features not supported directly by the hardware and are instead emulated by software. Currently only software white balance is implemented (badly) for Video4Linux cameras.

8 Bibliografia

[1] Bourgin, David - http://www.neuro.sfc.keio.ac.jp/~aly/polygon/info/*color*-space-faq.html - "*Color* Space FAQ" – (último acesso: 2006)

[2] Vários - http://en.wikipedia.org/wiki/*RGB_color*_model - "*RGB Colour* Model" – (último acesso: 10/2008)

[3] Vários - http://en.wikipedia.org/wiki/*HSL_color*_space - "*HSL Colour* Space" – (último acesso: 10/2008)

[4] Vários - http://en.wikipedia.org/wiki/Chroma_subsampling - "Chroma subsampling" – (último acesso: 2006)

[5] Vários - http://en.wikipedia.org/wiki/*HSV_color*_space - "*HSV Colour* Space" – (último acesso: 10/2008)

[6] Vários - http://en.wikipedia.org/wiki/*YIQ* - "*YIQ*" – (último acesso: 2006)

[7] Vários - http://en.wikipedia.org/wiki/Bayer_filter - "Bayer filter" - (último acesso: 31/10/2008)

[8] Vários - http://www.*gtk*.org/*API*/ - "*GTK*+ - *API* Documentation" – (último acesso: 2006)

[9] Hanbury, Allan - http://www.prip.tuwien.ac.at/~hanbury/*Colour*_histogram/ - "*Colour* Statistics Histograms" – (último acesso: 2006)

[10] Vários - http://documentation.openoffice.org/HOW_TO/formula/Formula_HowTo_1_0.pdf - "OpenOffice.org How-To" – (último acesso: 05/2008)

[11] Vários - http://en.wikipedia.org/wiki/CIE_1931_*color*_space - "CIE 1931 *Colour* Space" – (último acesso: 2006)

[12] Vários - http://www.lib*sdl*.org/cgi/docwiki.cgi/ - "*SDL* Documentation Wiki" – (último acesso: 2008)

[13] Vários - http://microrato.ua.pt – "Concurso Micro-Rato" – (último acesso: 2006)

[14] Vários - http://www.ieeta.pt/carl - "CARL" – (último acesso: 2006)

[15] Vários - http://www.ieeta.pt/atri/cambada - "CAMBADA" – (último acesso: 2006)

[16] Vários - http://www.fourcc.org/ - "Video Codec and *Pixel* Format Definitions" – (último acesso: 2006)

[17] Nave, R. - http://hyperphysics.phy-astr.gsu.edu/hbase/vision/cie.html - "CIE *Color* System" – (último acesso: 2006)

[18] Grimson, W. E. L.; Mundy, J. L. - "Computer Vision Applications" - Association for Computing Machinery. Communications of the ACM – Março 1994

[19] Mayer, Gerd; Utz, Hans; Kraetzschmar, Gerhard - "Towards Autonomous Vision Self-Calibration for Soccer Robôs" - Universade de Ulm - 2002

[20] Mayer, Gerd; Utz, Hans; Kraetzschmar, Gerhard - "Playing Robô Soccer under Natural Light: A Case Study" - The Eighth International RoboCup S*ympo*sium - 2004

[21] Ford, Adrian; Roberts, Alan - "*Colour* Space Conversions" - Agosto 1998

[22] Browning, Brett; Veloso, Manuela - "Real-Time, Adaptive *Color*-based Robô Vision" - School of Computer Science , Carnegie Mellon University – Ano desconhecido

[23] Schulz, Dirk; Fox, Dieter - "Bayesian *Color* Estimation for Adaptive Vision-based Robô Localization" - Intelligent Robôs and Systems, 2004. (IROS 2004). Proceedings. - Outubro 2004

[24] Zrimec, Tatiana; Wyatt, Andy - "Learning to Recognize Objects - Toward Automatic Calibration of *Color* Vision for Sony Robôs" - Workshop of the nineteenth international conference on machine learning (ICML-2002) - 2002

[25] Jüngel, Matthias; Hoffmann, Jan; Lötzsch, Martin - "A Real-Time Auto-Adjusting Vision System for Robôic Soccer" - 7th International Workshop on RoboCup 2003 - 2003

[26] Blauensteiner, Philipp; Wildenauer, Horst; Hanbury, Allan; Kampel, Martin - "On *Colour* Spaces for Change Detection and Shadow Suppression" - *Proceedings of the 11th CVWWS6: Computer Winter Vision Workshop – 2006*

[27] Kampel, M.; Wildenauer H.; Blauensteiner P.; Hanbury A. - "Improved motion segmentation based on shadow detection" - *Electronic Letters on Computer Vision and Image Analysis 6(3) – 1/12/2007*

[28] Hanbury, Allan; Serra, Jean – "A 3D-polar coordinate *Colour* representation well adapted to image analysis" - Proceedings of the Scandinavian Conference on Image Analysis (SCIA) – 2003

[29] François, Alexandre; Medioni, Gérard – "Adaptive Color Background Modeling for Real-Time Segmentation of Video Streams" - Proceedings of the International on Imaging Science, Systems, and Technology – 1999

[30] Fleyeh, Hasan – "Traffic and Road Sign Recognition" – Doctorate Thesis – July 2008

[31] Hanbury, Allan – "The Taming of the Hue, Saturation and Brightness *Colour* Space" - Proceedings of the 7th CVWW – 2002

[32] Hanbury, Allan; Marcotegui, Beatriz – "Waterfall Segmentation of Complex Scenes" - 7th Asian Conference on Computer Vision, Hyderabad, India – 13-16/1/2006

[33] Hanbury, Allan; Serra, Jean – "*Colour* Image Analysis in 3D-Polar Coordinates" - DAGM symposium N°25, Magdeburg , Germany - 10/9/2003

[34] Cardani, Darrin – "Adventures in HSV Space" - The Advanced Developers Hands On Conference - 31/7/2005

[35] Micusík, Branislav; Hanbury, Allan – "Supervised Texture Detection in Images" – Center for Advanced Information Processing - 2005

[36] Micusík, Branislav; Hanbury, Allan – "Automatic Image Segmentation by Positioning a Seed" - European Conference on Computer Vision - 2006

[37] Angulo, Jésus; Serra, Jean – "Color Segmentation by Ordered Mergings" - IEEE Int'l Conf. Image

Processing - 2003

[38] Sural, Shamik; Qian, Gang; Pramanik, Sakti – "Segmentation and Histogram Generation Using the HSV Color Space for Image Retrieval" - International Conference on Image Processing (ICIP) - 2002

[39] Yuchun, Fang; Tan, Tieniu – "A Novel Adaptive *Colour* Segmentation Algorithm and Its Application to Skin Detection" - Chinese Academy of Sciences - 2000