Rainbow-Po

Rafael Macito Zils, Gabriel Borges, Bruno Fonseca

15 de junho de 2015

Resumo

O objetivo do projeto é utilizar métodos de visão computacional para processar imagens obtidos a partir de uma câmera e fazer o tratamento da mesma em tempo real. O artigo detalha o processo todos os algoritmos utilizados, criados e alterados, durante o processo de desenvolvimento. Também discuti sobre os resultados obtidos ao final do projeto, como métodos de análise abordados e a eficiência dos mesmos.

1 Introdução

Esse projeto tem a finalidade de construir um algoritmo de análise e processamento de imagens consistente e preciso para criação de um jogo de jokenpo. O principal desafio está em construir um algoritmo rápido o suficiente para analisar os eventos em tempo real sem perder precisão, e que ao mesmo tempo não exija muito do jogador, tendo uma boa usabilidade.

No caso deste projeto, o jokenpo escolhido foi um criado pela série de TV The Big Bang Theory[5], que utiliza, como o original, pedra, papel e tesoura, com a adição de dois novos elementos: Lagarto e Spok, como na imagem a seguir:

 ,

2 Ferramentas

A biblioteca utilizada para tratar as imagens capturadas pela câmera é o OpenCV[2]. Porém suas funções não serão utilizadas diretamente, sendo chamada através de uma segunda biblioteca criado pelo Marcelo Hashimoto[4].

A biblioteca Allegro[3] está sendo usada para criação do jogo e suas mecânicas.

E a linguagem de programação usada no projeto é C[1].

3 Métodos

3.1 Análise

Para construção do algoritmo, levou-se em consideração três pontos focais distintos, sendo eles: Precisão, Velocidade e Jogabilidade.

Encontrar um equilíbrio entre esses três pontos foi essencial para que o algoritmo conseguisse ser preciso o suficiente para identificar a mão do jogador e suas respectivas jogadas, rápido o suficiente para não perder muitos quadros, o que poderia gerar uma baixa na precisão por analisar menos quadros por segundos, e por fim a jogabilidade, para não tirar muito conforto do jogador.

3.2 Soluções

O processo de desenvolvimento pôde ser separado em duas partes, a captura do movimento da mão para contagem do balançar da mão, mais focado em velocidade no processamento, para sempre acompanhar a mão e perder o mínimo de movimentos possível, e a captura da jogada selecionada pelo jogador, esse sim tendo que ser muito mais preciso.

Para encontrar um ponto intermediário entre os três fatores, foi construído um algoritmo que foca somente na imagem da mão do jogador, obtida na calibração, para diminuir o vetor de busca da mão e otimizar o tempo de processamento, mantendo a precisão.

Para tornar tanto a calibragem como o processamento da imagem mais viável, uma "luva colorida"azul com a ponta dos dedos magenta foi usada.

4 Algoritmos Base

4.1 Detectação de Movimento

A detectção de movimento é um algoritmo consideravelmente simples, e pode ser dado por:

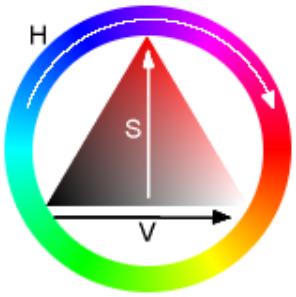
Entrada: Duas imagens
Resultado: retorna se houve ou não movimento
para cada *pixel das duas imagens* **faça**
  se *pixel1 != pixel2* **então**
    contador = contador + 1;
  fim
fim
se *contador >= n* **então**
  retorna *True;*
senão
  retorna *False;*
fim

Algoritmo 1: Detecta Movimento

Onde n é o numero que delimita quantos pixels precisam ser diferentes para detectar o movimento.

4.2 RGB para HSV

HSV é um sistema de cores composto por Hue (matiz), que é o tipo da cor, abrangendo todas as cores do espectro, desde o vermelho até o violeta, mais o magenta. Saturation (saturação) , também chamado de "pureza", é a tonalidade de cinza do pixel, quanto menor esse valor, mais com tom de cinza aparecerá a imagem e quanto maior o valor, mais "pura"é a imagem. Value (luminosidade) que define o brilho da cor.



Se um pixel estiver no formato RGB, com os valores variando de 0 a 1.0, então as seguintes formulas se aplicão a transformação para HSV:

$$H = \begin{cases} 60 \times \frac{G-B}{MAX-MIN} + 0, & \text{if } MAX = R \\ & \text{and } G \geq B \\ 60 \times \frac{G-B}{MAX-MIN} + 360, & \text{if } MAX = R \\ & \text{and } G < B \\ 60 \times \frac{B-R}{MAX-MIN} + 120, & \text{if } MAX = G \\ 60 \times \frac{R-G}{MAX-MIN} + 240, & \text{if } MAX = B \end{cases}$$

$$S = \frac{MAX - MIN}{MAX}$$

$$V = MAX$$

4.3 Disjoint-Set

Disjoint-set é uma estrutura de dados que mantem controle de um conjunto de elementos particionados em um numero de subconjuntos disjuntos. Basicamente é uma estrutura que agrupo elementos que não estão a primórdio no mesmo conjunto mas que tem algo em comum. Para essa estrutura poder ser utilizada nesse projeto, foi implementada através de um grafo, onde cada vértice representa um conjunto, e cada pixel da imagem está relacionado a um único conjunto, e os conjuntos podem também se inter-relacionar, criando o arco entre os vértices. Uma outra alteração foi feita, para agilizar o processo, cada pixel pertencente ao conjunto representa apenas uma massa do vértice, que somadas, dão a massa final daquele conjunto. As principais funções de operações são:

Entrada: variável do tipo Pixel
Resultado: retorna o conjunto (vértice) do grafo que representa aquele pixel
retorna *representante do pixel*

Algoritmo 2: Encontrar o conjunto representante do pixel

E a função que une os conjuntos alcançáveis a partir de um vértice:

Entrada: Grafo com Conjuntos
Resultado: Unir os conjuntos semelhantes
para cada *vértice do grafo* **faça**
    Encontrar todos os vértices alcançáveis;
    Somar a massa de todos os vértices alcançáveis no vértice chamador;
    Zerar a massa desse vértice já que agora ela pertence ao vértice chamador;
fim

Algoritmo 3: Unir Conjuntos

Basicamente esse algoritmo pode ser implementado com um algoritmo de busca em profundidade ou largura, já que se trata de um grafo, e ambos terão o mesmo resultado para a aplicação.

4.4 Componente Conexo

Componente Conexo (ou Connected Components) é um algoritmo que identifica quais pixels na imagem fazem parte do mesmo conjunto, perguntando se ele possui algum pixel com a mesma cor dos lados:

```
Entrada: Imagem
Resultado: Grafo com componentes da imagem
se pixel atual == cor desejada então
    para cada pixel adjacente faça
        se pixel adjacente == cor desejada então
            pertencem ao mesmo conjunto;
        fim
        se mais de um pixel adjacente == cor desejada então
            pixel atual pertence ao menor conjunto;
            criar um arco entre os dois ou mais conjuntos encontrados;
        fim
        se Nenhum pixel adjacente == cor desejada então
            cirar um novo conjunto e associar o pixel atual a esse conjunto;
        fim
    fim
fim
```

Algoritmo 4: Connected Components Labeling

Depois de ter os conjuntos criados, é só fazer a união deles, usando o algoritmo descrito na seção acima (Unir Conjuntos). No final terá um grafo formado com todos os conjuntos existentes tendo massa maior que zero.

4.5 Filtros de Erosão e Dilatação

Os filtros de Erosão e de Dilatação são operações de processamento de imagem morfológica. Os filtros foram implementados nesse projeto seguindo os seguintes pseudocódigo:

```
Entrada: Imagem
Resultado: Imagem Erodida
para cada pixel da imagem faça
    se pixel == cor desejada então
        para cada pixel de uma dada matriz de kernel faça
            se pixel != cor desejada então
                pixel central = branco;
            fim
        fim
    fim
fim
```

Algoritmo 5: Filtro de Erosão

Assim sendo, para cada pixel encontrado, se ele tiver um pixel de cor diferente em volta, ele perde seu valor.

O algoritmo de Dilatação é a operação inversa da Erosão, e pode ser dado por:

```
Entrada: Imagem
Resultado: Imagem Dilatada
para cada pixel da imagem faça
    se pixel != cor desejada então
        para cada pixel de uma dada matriz de kernel faça
            se pixel == cor desejada então
                pixel central = cor desejada;
            fim
        fim
    fim
fim
```

Algoritmo 6: Filtro de Dilatação

Seguindo a lógica, esse filtro pega todos os pixels que forem diferentes da cor desejada, e se ele tiver algum com a cor em volta, ele recebe essa mesma cor, assim, dilatando os pixels com a cor desejada na imagem.

5 Calibragem

A calibração consiste em detectar a média da luminosidade do local obtida através da soma da luminosidade de cada pixel quando convertido de RGB para HSV, e dividir pela quantidade de pixels existente na imagem. Assim pode-se adaptar os valores da matiz e saturação da mão e dos dedos de acordo com o ambiente.

Depois de aproximar os valores, será aplicado um algoritmo de connected components até encontrar apenas um componente na imagem, que será a mão. Caso não encontro apenas um depois de um dado tempo, chega-se a conclusão de que o ambiente possui um nível exacerbado de ruído, o que, infelizmente, o algoritmo construído para esse projeto não consegue lidar.

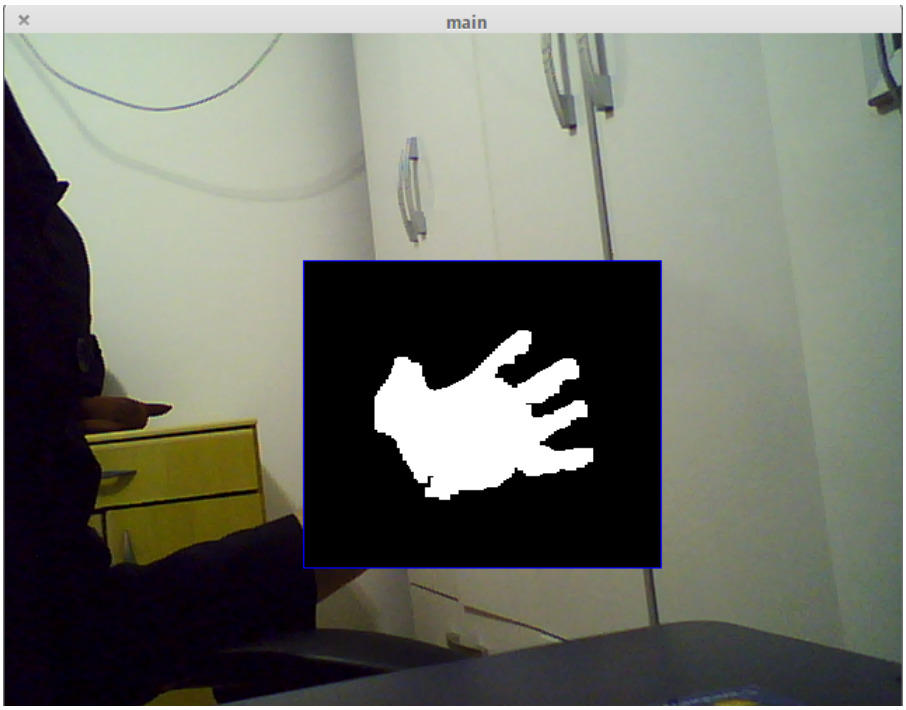
Para ajudar no processo, filtros de erosão e dilatação podem ser usados e, quando encontrar apenas a mão do jogador na imagem, deverá ser construído o rastreador em volta da mesma.

O rastreador nada mais é do que a distância entre o primeiro e o último pixel pertencentes a mão na horizontal e vertical, mais uma constante para quando o jogador mover a mão, tendo quatro variáveis, norte, sul, leste e oeste.

6 Rastreamento do Movimento da Mão

Após os parâmetros serem calibrados, é iniciado o processo de atualizar o rastreador a cada frame da imagem. O processo de obtenção e redefinição do rastreador é bem simples, para não deixar o processo lento. A imagem obtida pela câmera é primeiro transformada de RGB para HSV, depois a imagem é varrida de todos os lados até achar um pixel de cor selecionada, então esse será seu novo limite, aplicando esse método às todas variáveis do rastreador.

Para melhorar a precisão do processo de atualizar o rastreador, foi utilizado um filtro de erosão para eliminar possível ruídos da imagem, porém esse processo também acaba eliminando alguns pixels da borda da mão, então um outro filtro, agora de dilatação, recupera esses pixels perdidos.



Esse processo será repetido durante todo o programa, e sempre que o rastreador terminar de ser atualizado, será verificado: se o centro do rastreador (que deve ser próximo ao centro da mão) ultrapassou o delimitador, se sim, somar 1 a um contador até que chegue a 3, e caso a mão do jogador esteja descendo, uma flag do rastreador deve ser setada como verdadeiro, para controlar quando a mão do jogador está subindo ou descendo.

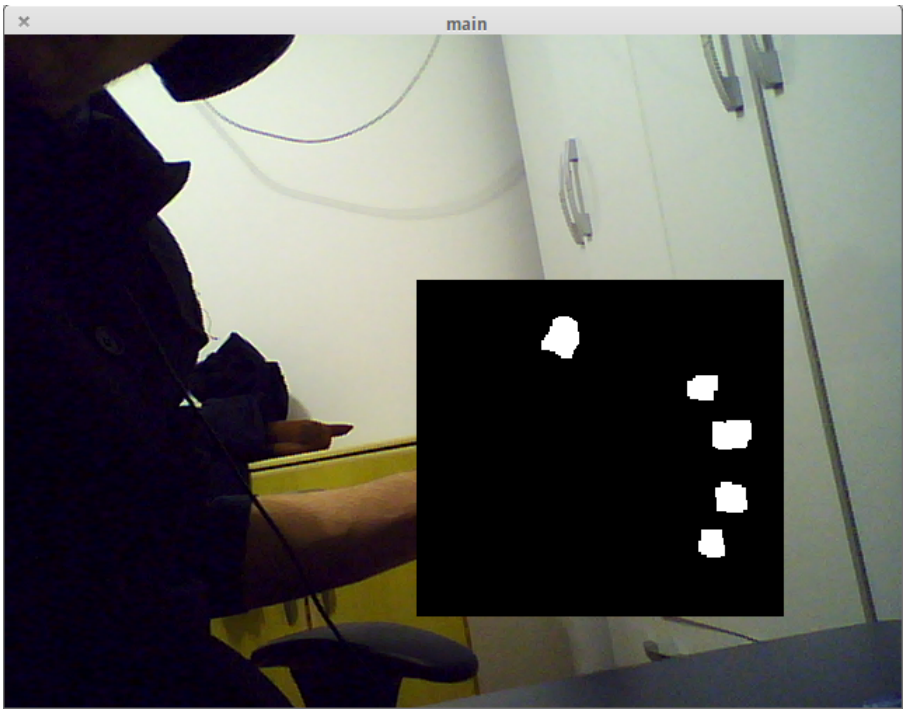
7 Analise da Jogada

Após o jogador balançar a mão três vezes, o algoritmo continuará atualizando o rastreador até que o movimento pare, para isso será usado o algoritmo de detecção de movimento.

Só quando o movimento parar, será executado o algoritmo que analisa a jogada, sendo que esse necessita ser muito mais preciso que o anterior, pois qualquer alteração na imagem leva a uma analisa errada e o computador pode interpretar como uma jogada diferente do que o jogador realmente jogou.

Agora será executado um algoritmo de connected components na imagem, porém com um adicional, para cada pixel encontrado com a cor desejada, também será verificado se algum dos pixels em volta possuí a cor da mão, caso sim, esse conjunto é um dedo, e caso não, é apenas ruído da imagem.

Após obter um grafo com todos os componentes da imagem, pode-se criar um novo vetor para salvar apenas os componentes que forem considerados dedos pelo algoritmo, para evitar ter que percorrer o grafo inteiro sempre e passar por conjuntos já dados como inválidos.



Com tudo isso feito, sobra analisar a quantidade de conjuntos que sobraram:

```
Entrada: Vetor com Componentes Encontrados na Imagem
Resultado: Jogada Feita Pelo Jogador
se número de componentes == 1 então
    retorna pedra;
fim
se número de componentes == 2 então
    aplicar um sort no vetor;
    se maior componente > 2 vezes o menor então
        retorna papel;
    senão
        retorna lagarto;
    fim
fim
se número de componentes == 3 então
    aplicar um sort no vetor;
    média = (maior componente + menor componente) / 2;
    se componente mediano > média então
        retorna spock;
    senão
        retorna tesoura;
    fim
fim
se número de componentes == 5 então
    retorna papel;
fim
se qualquer outra situação então
    retorna jogada invalida;
fim
```

Algoritmo 7: Analisa Jogada

8 Resultados

A implementação dos algoritmos não teve uma complexidade tão alta, porém é extremamente complicado trabalhar apenas com análise de cores, porque dependendo do ambiente, o nível de ruído que a câmera trará será muito alto. Não só o ambiente, mas a câmera também faz muita diferença, mudando muito a cor, a saturação e a luminosidade de câmera para câmera. Já não bastasse esses dois problemas, o método de compressão da imagem feita pelo software também pode acarretar em perda de dados até alteração da cor da imagem.

Esses problemas podem ser amenizados fazendo uma calibragem, porém nunca será 100% garantido, o que acarreta em uma análise imprecisa, mas não ruim, só que a precisão vai variar de situação para situação.

Porém vendo por outro lado, enquanto se perde muita consistência, se ganha com velocidade, pois os métodos para obtenção de respostas são muito mais rápidos, sendo que sua precisão varia. Em um ambiente de condições perfeitas, o que não é difícil de se obter, a precisão é extremamente alta, e como sempre mantém uma boa velocidade de processamento, o algoritmo se mostra extremamente eficiente.

9 Conclusão

Com esses resultados pode-se concluir que não existe um algoritmo perfeito para esse caso, pois se um método de análise mais preciso fosse utilizado, o algoritmo passaria a ficar tão lendo que não mais conseguiria acompanhar o movimento do jogador. Portanto, para se obter a velocidade necessária, teve-se que sacrificar consistência, e a precisão acaba variando um pouco com o ambiente. É como diria o orientador deste projeto: "Não existe almoço grátis".

Referências

[1] C, linguagem de programação, versão C99.

[2] OpenCV, Biblioteca de Computação Visual, versão 2.4.10, 2014.

[3] Allegro, biblioteca livre para criação de jogos, versão 5.0.10, 2013.

[4] Marcelo Hashimoto.

[5] Jokenpo do BBT, criado na série de TV The Big Bang Theory.