EasyMocap - *AGI09*

Irena Andonov, Arnaud Ramey - {ireand18/ramey}@kth.se

January 2010

Abstract

This report is about a project with EasyMocap and a computer animated figure. EasyMocap is like Motion Capture, but with only one camera used. By shooting an objects movement with a camera, a file with the movements recorded is created. Then the file is applied on the animated figure that will perform exactly the same movements. The procedure how it is made possible is described, and also the results with images are presented here.

Contents

Introduction

The first motion to be captured and displayed in a sequence was made possible 1878, when Edward Muybridge photographed a galloping horse [Muybridge]. He placed out cameras with self timer, that were triggered a millisecond after each other and photographed the horse.

The method to 'capture motion' with camera has developed a lot, and one of the latest technologies is to record motions of an object with a camera and apply it on computer animated figures with a very similar structure as the object- at least the parts that are to be moved with this method. It is a method that is very distinguished from of the method of the pioneer of motion recording and displaying.

Motion capture is commonly used in animated games and movies. It is a quite new technology and have made it possible to make computer animated figures perform motions that looks naturally.

That would be very difficult and complicated to do with programming and the result wouldn't be as good as with the use of EasyMocap.

We will explain how the Maya figure is built. We have used color markers and this will also be explained more thoroughly. Also how the code works, and at last - the results from making all this possible the cost of memory from executing it.

1 Purpose

The 3D-figure is built in Maya and transformed to OpenGL. We want to make the arms and legs move separately. We will have 10 separate body parts of our animated figure that are used in OpenGL. The arms and legs are split in upper and lower parts, and the torso will also be marked with an own color. Only the head wont be marked.

We will use 5 different colors to mark an actors body part whose movements are filmed by a camera. We use only 5 colors because of the code that only recognize a certain range of RGB-components. This range for us is divided in 5 sections - that is red (255,0,0), blue (0,0,255), pink (255,0,255), yellow (255,255,0) and green (0,255,0). If we would use two colors whose RGB-components would be very close, they would not be perceived as two different colors by the code, but as one. So we have to be careful with that.

The camera is connected to a C++ code, that is creating a file with the recorded movements.

Every body part is marked with a different color so it can be recognized as a separate part by the code, like the upper arm and the forearm has to be two separate parts. We have 9 body parts to mark and 5 different colors, so we have

to use four colors twice. We have to put the markers on the body of the actor so the same colors wont be to close and make a confusion of the code.

2 General structure of the program

The idea is to make an actor move in front of a camera. He is equipped with **markers** on his body. These markers are in fact simple pieces of fabric, with a vivid color. We detect the position and orientation of these markers. By those means, we can ”rebuild” the body and send the data of its movement to a computer generated figure.

1. **Image analysis :** finding where the markers are on the video
   The aim of this part is to extract the relevant portions of the image that correspond to the members of the person doing the motion capture. In input, we have a video recorded with a random camera. In output, we have the orientation of every zone of pixels with a color matching the colors of the members. This is an OpenCV [OpenCV] program.
2. **Human body :** find the orientation on the actor on the video, according to the markers
   We convert the information we extract from the video images to compute the orientations of the human body.
3. **Representation of the body :** send the data about the orientation to the computer generated body
   We send to the OpenGL animated body the orientation of the human body that we computed earlier. Thus, we create an animation of a computer generated character, controlled by a real human in real time. This is an OpenGL [OpenGL] program. How an OpenGL animated body is represented is explained in the part 2.1.

2.1 Structure of a body skeleton

At the beginning, we wanted to represent a 3D body (where the joint between 2 members would enable each of these two members to move in every direction). However, we chose to constrain ourselves to 2D moves. 2 principal reasons :

1. Finding the orientation of the human body in 3D can be hard.
2. Making the computations for the absolute positions along a chain of members orientated in 3D can be pretty hard.

So, only moves in 2D are possible (for example, lifting the arms in the axes of the shoulders, or waving goodbye).

A body is thus represented as a *position* and a tree of `Member`.

The position is a 3D point. It gives the position of the center of mass of the body.

A `Member` is made of :

- **a pointer to the "father"** f, that is the `Member` which it is linked to. For example, the head is linked to the torso, the left forearm is linked to the left arm, etc. The torso is the "root" of this tree structure.
- **The length of the member**.
- **The abscissa of fixation on** f. Thus, if it values 0, it means the member is fixated on the base of its father f, if it values $\frac{f.length}{2}$, it is on the middle of f and if is $f.length$ it means the joint is at the end of the f (the most common situation).
- **The angle between** f **and the current member**. It is expressed on the coordinates relative to f coordinates. For example, if this angle values 0, it means f and the member are aligned, if it values $\frac{\pi}{2}$ it means there is a right angle between both members.
- **A pointer to the RTG file used to draw the member**. In case no file is specified, the member will be drawn with a white orientated cylinder.

A representation of a human body is made on the figure 1.

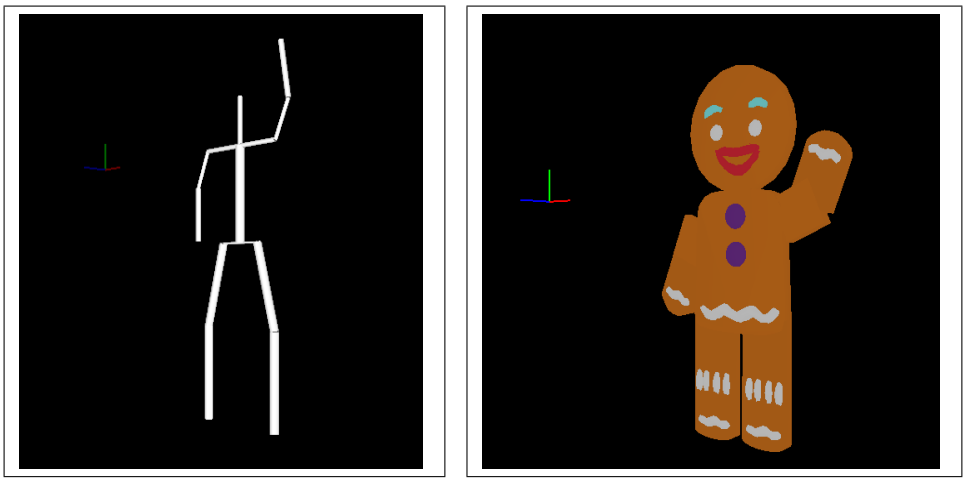

Figure 1: *On the left, an simple OpenGL computer generated body. This example is shaped as a human body. On the right, a more elaborated OpenGL body. This one uses RTG files coming from Maya for its representation.*

2.2 The Maya models

We decided to make our own Gingy (from Shrek) in Maya. It has a human body structure so it works with EasyMocap where we want to use a human actor for the movements. It is built with primitive polygons and textured with assigning

a material and a certain color on the polygons. Afterwards we placed the ten different body parts; the upper-arms, the forearms, the torso, the head, the shins and the down-legs separate on origin and made rtg-files of every part to use in OpenGL.

You can see some screenshots of Gingy rendered in Maya on fig 2.

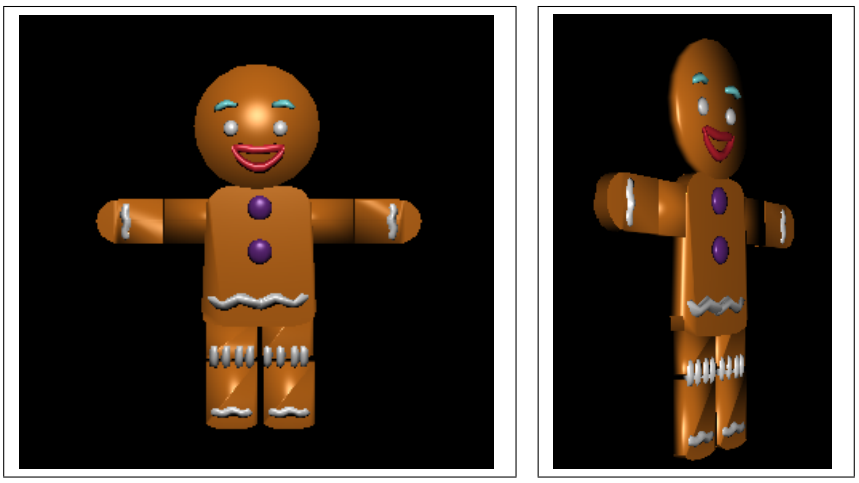

Figure 2: *Snaphots of rendered Gingy in Maya..*

3 The algorithmic

We will present here some relevant steps from the algorithm.

3.1 Color conversion : from RGB to HSV

RGB color mode The images supplied by the camera are color images. The camera supplies them in a **RGB color mode**. It means the images are made of three layers : *Red, Green, Blue*. Each layer is in fact an array of size (w, h), where w is the width of the image and h its height, containing int numbers between 0 and 255.

For instance, if a a pixel is made of the three components $(255, 0, 0)$, it is in a bright red color in the image.

It could be possible to make filters on the RGB components of the image. However, these components are highly variable to light exposure and the brightness of the room. For instance, a purple sheet of paper successively brought from a bright ambiance to a dark one will see its RGB components change a lot. Thus, it is hard to define accurate filters which are robust to light variations.

HSV color mode That is the reason why we chose to convert the image in **HSV color mode**. HSV stands for *Hue, Saturation, Value*. A further description of these three components can be found on Wikipedia [WikiHSV]. In our representation, *Hue* belongs to $[0, 360]$ while *Saturation* and *Value* are in $[0, 255]$. What is important to keep in mind is that the color itself of a pixel is encoded in the Hue component, while its intensity is represented in the two others.

Consequently, we can define accurate filters on colors by setting narrow *Hue* filters, and loose *Saturation* and *Value* filters.

3.2 Filtering the image

Now that we have converted the image of the camera in HSV color mode, we need to only keep in the image the pixels relevant to the color filters we defined.

One color filter will only allow one of the colors we used for the fabrics. For instance, the *Yellow* filter is defined by :

- for *Hue*, a minimal allowed value of 20 and a maximal value of 60
- for *Saturation*, a minimal allowed value of 150 and a maximal value of 255 (no upper limit)
- for *Value*, a minimal allowed value of 100 and a maximal value of 255 (idem)

These values have to be calibrated by hand at the beginning of the use of the program. However, as we explained before, they are more or less independent of the lightning conditions.

Here is a bit more in detail what we want to do. We consider we have a 1 channel output image out, which has the same dimensions as the image of the camera and is initially black. We consider a pixel of coordinates $(x, y) \in \mathbb{N}^2$ and of HSV components $(h, s, v) \in [0, 360] \times [0, 255]^2$. Our purpose is that, if (h, s, v) belongs to the values allowed by one of the filter, say the filter with index i, then $out(x, y) = i$.

3.2.1 First method : manual filtering

The first method is to scan the HSV image, that means to read the *Hue, Saturation, Value* components of each pixel thanks to iterators. So, for every pixel (x, y), we see if its components match one of the filters. If it is the case, we change $out(x, y)$ to the filter index.

In pseudo code, this algorithm is written :

```
function HSV(Image input, filters F)
```

```
    clear out
    init H, S, V to the values of input(0,0)
    for y : 0 -> input.h
    for x : 0 -> input.w
        for each filter f in F
        if (H, S, V) matches f
            out(x, y) = f.index
            break
    increment H, S, V
    return out
```

3.2.2 Second method : OpenCV filtering

In the first method, we were filtering ”by ourself” the input image with a *if / else* structure.

We can also use OpenCV primitives. For instance, using thresholds, it is easy to filter a one channel image : you allow the interval of possible values, and every value outside this interval will be turned to 0.

The idea is then to split our 3-channels HSV image into three separate one-channel images. Then you apply the corresponding filter on each one of the layers and get the total filter by applying a boolean AND on the three images.

In pseudo code, this algorithm is written :

```
function HSV2(Image input, filters F)
    clear out
    split input in H_layer, S_layer, V_layer
    for each filter f in F
        H_filtered = minmax_filter (H_layer, f.Hmin, f.Hmax)
        S_filtered = minmax_filter (S_layer, f.Smin, f.Smax)
        V_filtered = minmax_filter (V_layer, f.Vmin, f.Vmax)
        H_filtered = and (H_filtered, S_filtered, V_filtered)
        out = max (out, H_filtered)
    return out
```

3.2.3 Comparison

The output images given by the 2 methods are, as we could foresee, strictly identical. However, the time performances are different.

We could expect method 1 to be faster, as it only requires going through the whole image once. However, after benchmarking on a wide set of images, that method 2 was around 60% faster.

We think this is due to the fact that OpenCV primitives are highly optimized to go fast (basic operators on image), and going manually and iteratively through an image does not benefit from the same degree of optimization.

So, **method 2 is the method used in the program.**

3.3 Finding components : Union Find using the Disjoint-Sets data structure

Now that we have filtered the image, we need to find which pixels correspond to the color markers. We have to do a theoretical interlude.

We consider a set of points $P = \left\{p = (x, y) \in \mathbb{N}^2\right\} \in \mathbb{N}^{2\mathbb{N}}$. In fact, these points represent the non null points of an image.

A **connected component** of P is a subset C of P points such as

$$\forall c \in C, \exists \tilde{c} \in C, \left\|c - \tilde{c}\right\|_{L_1} = \max\left|c.x - \tilde{c}.x\right|, \left|c.y - \tilde{c}.y\right| = 1$$

That corresponds to the usual definition of connected components for non-oriented graphs, supposing that two points at an euclidean distance of 1 on P are two linked nodes in the corresponding graph. This equivalence is illustrated in figure 3.

Now, we look for a quick way to find every connected component of P. More accurately, it is to obtain a partition $\{P_i, 1 \leq i \leq n\}$ of P, where n is the number of connected components of P and $\forall i \in [1; n],\ P_i$ is a connected component. This is illustrated in figure 4. There are many ways to make it, we chose to use the **Union-Find** algorithm with the **Disjoint-Sets** data structure.

Disjoint sets The disjoint-sets data structure was first presented in [Galler] in 1964. A set is a structure of tree, in which each node has a reference to his father (while it is references to his sons in the classic tree structure). A disjoint-set forest is a list of sets.

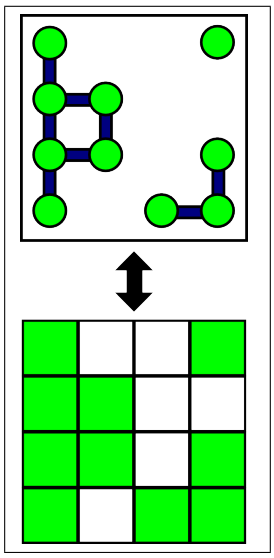

Figure 3: *Illustration of the correspondence between connected components in images and graphs*

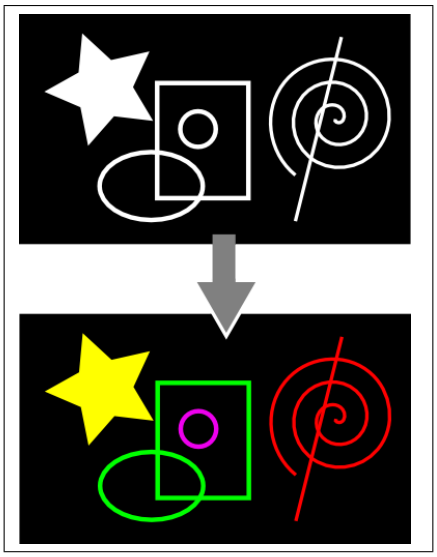

Figure 4: *The objective of this part is to extract the components. Above, the original image : each non-black point represents a point of P. Under are the connected components $\{P_i\}$. Each component has been drawn in a different color.*

Union Find A union-find algorithm supplies a disjoint-set forest two functions : `union` and `find`. The former allows the fusion of two sets, while the latter finds the root of the tree in which is located the supplied argument.

Basic implementation : A simple implementation would be :

```
function Find(Set A)
    Set F = A;
```

```
    while F.father != F
        F = F.father;
    return F;

function Union(Set A, Set B)
    Set FA = Find (A);
    Set FB = Find (B);
    FB.father = FA;
```

This is visually illustrated on image 5.

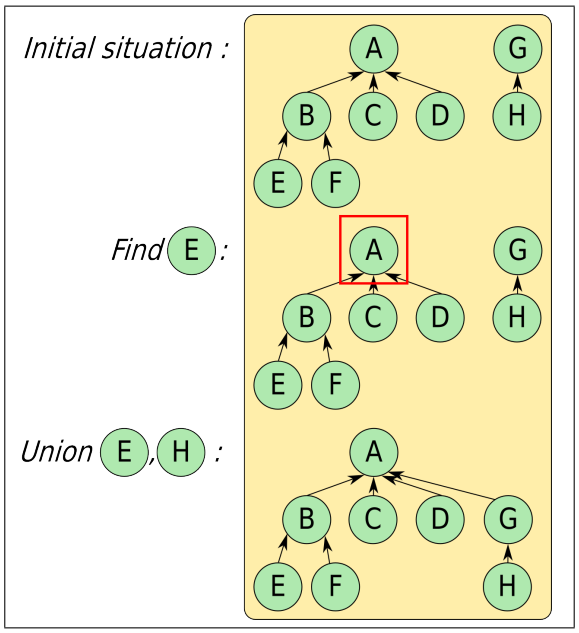

Figure 5: *Illustration of the basic implementation of the two functions $Find$ and $Union$ with an example disjoint-set forest.*

Improved implementation : Two improvements are usually made when using the Union Find algorithm. Both of them are aimed to shorten the height of the trees in the disjoint-set forest.

The first one, called *Path Compression*, is to modify the function `Find`. For instance, if a call of `Find(A)` returns `F`, then the improved function will connect directly to `F` each node in the path from `A` to `F`. In pseudo-code, this would be written :

```
function Find(Set A)
    Set F = A;
```

```
    while F.father != F
        F = F.father;
    // now update the fathers
    Set B = A, C;
    while B != F
        C = B.father;
        B.father = F;
        B = C;
```

The second one usually made is called *Union by Rank*. The idea is to always attach the smaller tree to the root of the larger tree, rather than always the first one. However, in our case (to find connected components), it is of primordial importance to always attach the current node to the node which has a smaller y-coordinate. Actually, we add a heuristic `index = y * width + x` to the components of the node : thus, the `Union` function will always link the higher-index node to the lower-index one.

```
function Union(Set A, Set B)
    Set FA = Find (A);
    Set FB = Find (B);
    if FB.index > FA.index
        FB.father = FA;
    else
        FA.father = FB;
```

The two improvements are illustrated on image 6.

The algorithm Now that we are more familiar with disjoint-sets forests and Union-Find algorithms, we can give a pseudo-code version of the algorithm.

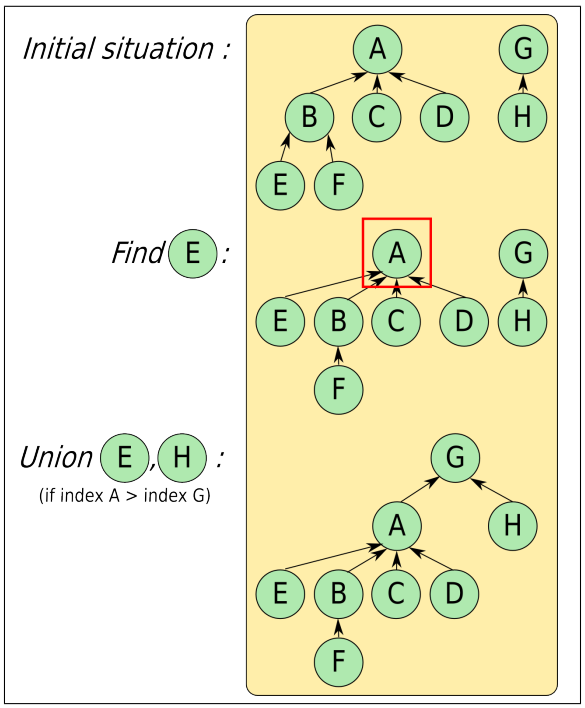

Figure 6: *Illustration of the improved implementation of the same example than in figure 5. We can notice that $Find$ has now modified the path between the searched node and its father. Moreover, $Union$ now defines as root the node with the lowest index.*

```
function FindComponents( List of point P, int width, int height )
  // computing the disjoint set
  for each p = (x, y) in P
    create Node (x,y)

  for int y = 0 to height
    for int x = 0 to width
      Node curr = Node (x, y);
      Node up = Node (x, y-1);
      Node left = Node (x-1, y);

      if x > 0 and y > 0 and exists up and exists left
        Union (up, curr);
        Union (left, curr);
      else if x > 0 and exists left
        Union (left, curr);
```

```
    else if y > 0 and exists up
      Union (up, curr);

 // now create the lists
create A, an array of size width * height of blank point lists;

for int y = 0 to height
  for int x = 0 to width
    Node curr = Node (x,y);
    add (x,y) at A[curr.father];

 // now collect the results
create R, a list of list of points;
for int y = 0 to height
  for int x = 0 to width
    if A[y * width + x] is not empty
      add A[y * width + x] to R;
return R;
```

3.4 Transforming a component into an oriented box

We consider the following problem : we have a point set standing for a connected component in our image, and we want to find the circumscribed rectangle of minimal area for this given 2D point set.

A simple solution can be to compute the bounding box for the set of points. However, first we are not sure it will be the minimal area rectangle, and then it would be constraining the problem by asking this box to have horizontal and vertical borders (that means an angle equal to 0 or 90 degrees).

We found a solution by using a function integrated in the OpenCV library : we use the function `cvMinAreaRect2()`. It returns the minimal rectangle that will bound the set, and this rectangle may be inclined relative to the vertical. This is illustrated on fig 7.

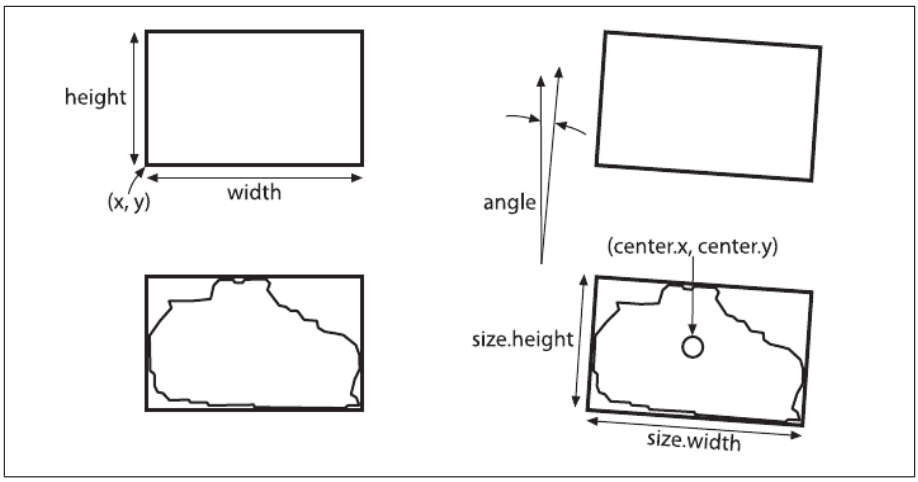

Figure 7: *On the left, the bounding box of a set of points. On the right, the orientated box of the same set of points. As we can see, it fits better the structure of the list of points.*

4 Results

The program was benchmarked on the following computer :

- **CPU:** Intel Core 2 Duo CPU P8600 @ 2.40GHz
- **GPU:** nVidia Corporation G98 [GeForce 9200M GS]
- **RAM:** 4 GiB RAM
- **Input images :** VGA images (640x480x24 bits, with a JPEG compression)

4.1 Samples

We chose to bound ourselves to the use of the upper members only for the tests. However, our code must be working for a full body.

We couldnt manage to fix the problem with UV-texturing our mayamodel. Something wasnt working right when we wanted to put a png-file on the UV-mapped components of the polygons. Thats why we decided the last day before the demonstration that we needed to test our Maya-model in OpenGL and gave it several materials with colors.

You can see some illustrations in fig 8.

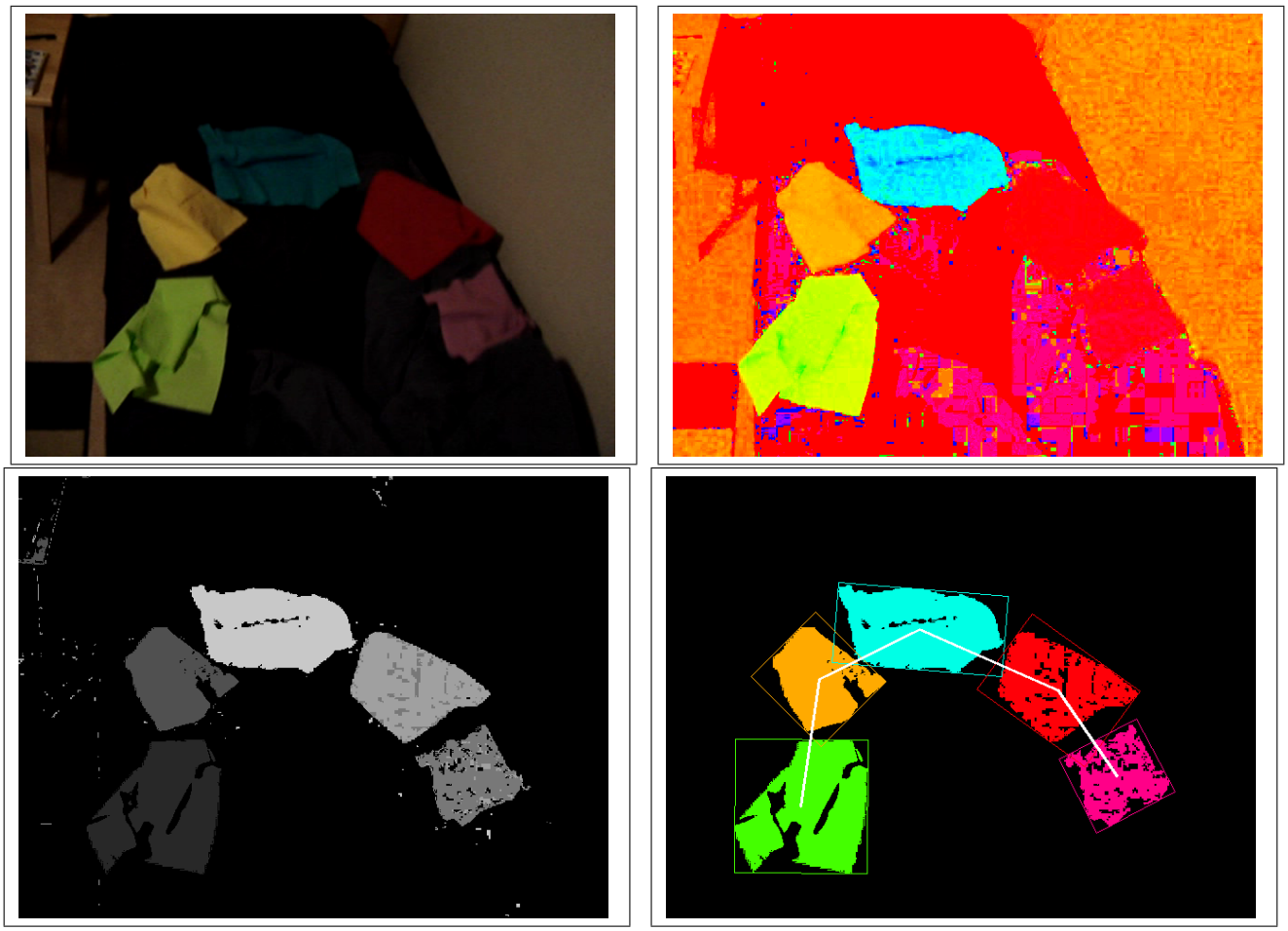

Figure 8: *Upper left, the input image. Upper right, the Hue component of the image (with S and V at their maximum). Lower left, the filtered image. Lower right, an image where we only keep the relevant components.*

4.2 Time costs

The analysis of an image and its rendering time are usually 60 milliseconds. They are split into the following way :

Type	Time (ms)
HSV conversion	5
Filtering image	4
All connected components	13
Biggest connected components	9
Computing orientated boxes	7
OpenCV image display and OpenGL rendering	33

This is illustrated on fig 9.

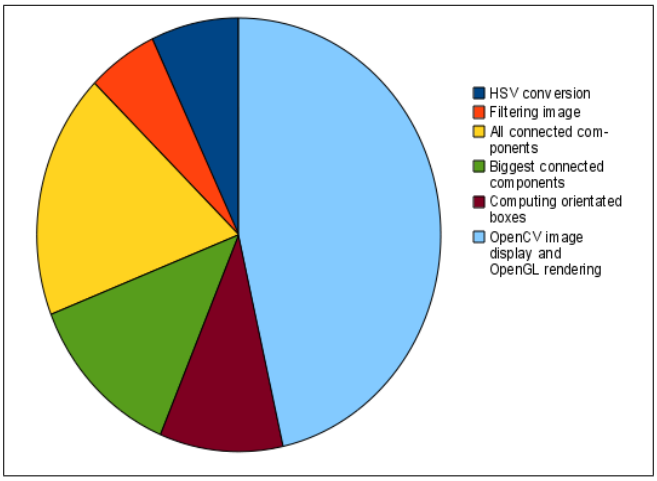

Figure 9: *Diagram of time consumption of the different steps.*

5 How to use the program

5.1 How to build the program

It is easy to build the program on a Unix computer. You need the following libraries before compiling : OpenCV [OpenCV], OpenGL [OpenGL], make.

5.1.1 How to use the program

To display the help, just launch in a terminal `./easy_mocap.exe -h`. It will display the help of the program.

Conclusion

We managed to develop a motion capture program that works in real time with one camera.

It was quite hard to obtain a version that was working fast enough to accept a real time version, but we got it. We learned quite a lot about the methods to optimize a code. On the other hand, we made some sacrifices with the complexity of the problem : we bound ourselves to 2D moves and to the upper members of the body.

This project was an interesting deepening experience in the world of OpenGL and Maya. It was also the occasion for us to bind them with OpenCV, the computer vision library.

References

[Galler] Bernard A. Galler and Michael J. Fischer, ”An improved equivalence algorithm”, *Communications of the ACM*, Volume 7, Issue 5 (May 1964), p301-303 ; 1964.

[WikiHSV] http://en.wikipedia.org/wiki/HSL_and_HSV

[Muybridge] http://www.associatedcontent.com/article/461209/the_first_movie_ever_made_a_history.html?cat=37

[OpenCV] http://opencv.willowgarage.com/wiki/

[OpenGL] http://www.opengl.org/