Georges Duverger (6129609)
Lilian Pedrali (6130615)
Loris Blanchon (6127401)

COMP 471

Final Project Report

Screen crime

Introduction

The « Screen Crime » project is an academic project for course « Computer Graphics » (COMP471). It is a team-based project which allows students to approach some picture and video processing techniques that had been reviewed during lectures and labs.

Our team is made up of 3 students:

- Georges Duverger, who comes from an engineering school in Grenoble (France), and who studies « Computer Networks and Multimedia Communications ». Before this school, he got qualified in « Multimedia Communication ». So for him, multimedia was not a discovery, and this project was a good opportunity to put into practice some techniques he studied before, and to go further into this subject.
- Lilian Pedrali, who studies « Computer Science » in Lyon (France), in a engineering school. He has always been interested in graphics processing, and especially in mathematics used for it. So for him , this project was the opportunity to do some research about algorithms, their efficiency, their complexity, and so on...
- Loris Blanchon, who studies « Computer Science » in Toulouse (France), in a engineering school too. His favorite domain is video, so he already did some video editing, with artistic research or purely technical one. For him too, this project has been an opportunity to learn new techniques and to use already known ones.

Concept presentation

Our final team project is a video responsive system named *Screen Crime*. The obvious reference is these well known crime scenes from crime movies. In such scenes, we see a white line drawn on the floor all over a dead body, as illustrated in Figure 1. It is mainly what we tried to do but interactively.

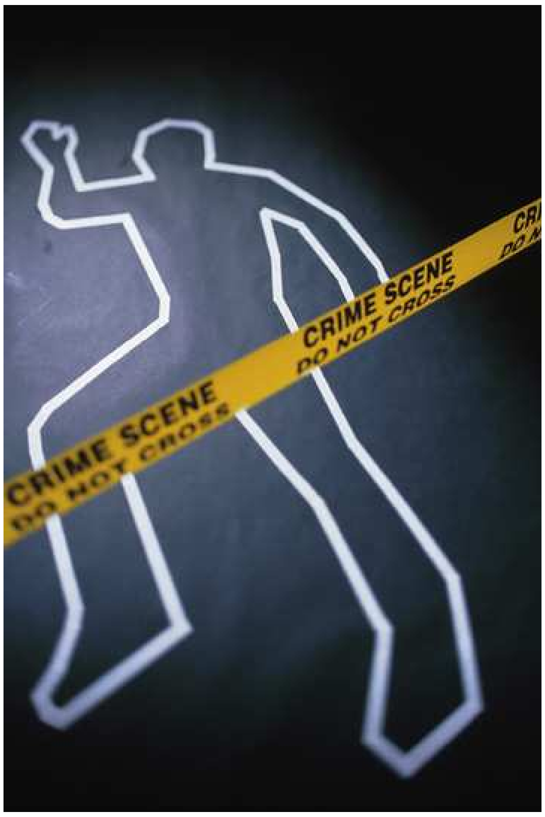

Figure 1: Crime scene from crime movie

The main scenario of our project is the following: a passer-by staying in front of the camera (i.e., looking at the previous victims) for too long will be *killed*, in his turn, by the system. The interaction is therefore quite simple. By doing so, we thought it would be easier to comprehend and more understandable by passers-by.

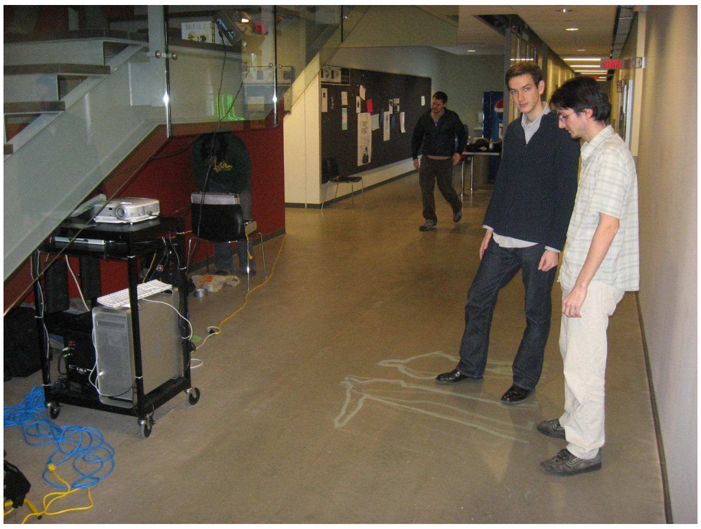

Figure 2: Main scenario

Our ambition was to recreate the *film noir* mood. We thought it would be attractive to install this scene in a public area with a lot of unaware passers-by. To reinforce this atmosphere, we also play a gun shot sound. It add surprise and, with hindsight, comprehension of the process. We also planned to scotch-tape a *do not cross* ribbon to make explicit the space of the crime scene (and therefore of our system).

We understand that our project could sound a little weird but that is one of the reasons why it is interesting. In a more symbolic point of view, we liked the idea that by looking at the previous victims and while trying to understand, the passer-by will become one of them. Thus, he will fully understand the installation when he will be shoot (i.e., too late to be saved).

Development

Our project is divided in 3 big steps. It was the easiest way to perform our project and share the work.

1. Motion capture and detection of inactivity

The objective of the beginning of the patch is to obtain position of the passer-by. To do it, we use a basic technique of edge detection.

Detecting pixel changes

For each color channel of each pixel, we do the subtraction between the value of the live input, and the one from a picture of the background (for more explanation on how is obtained this background picture, see below). Next, for each pixel, we take only the maximal value between the 3 color channels and we threshold the result with an arbitrarily defined value.
For example, consider these two situations:

- Pixel from live input and same pixel from background picture are highly different : the algorithm will detect there is something (or somebody) in front of the camera.

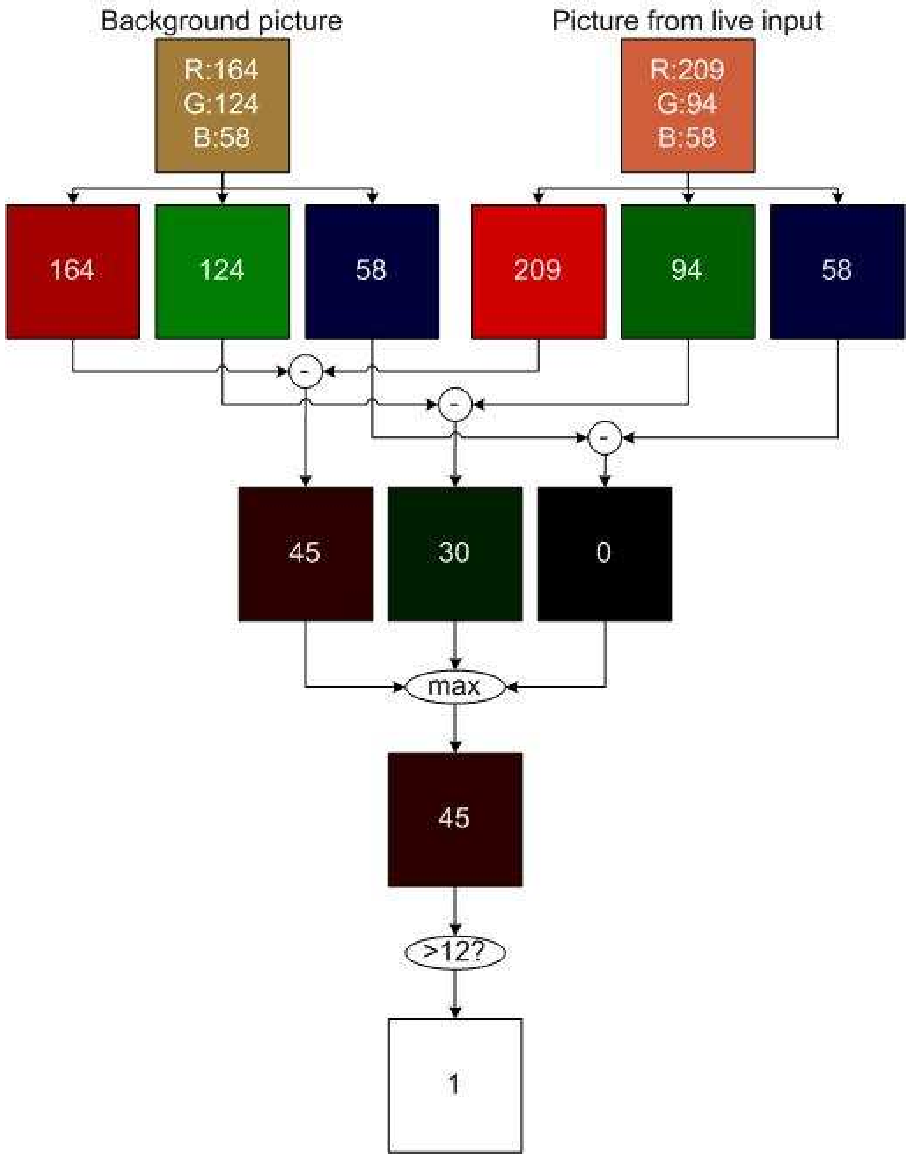
Background picture
R:164
G:124
B:58
Picture from live input
R:209
G:94
B:58
164
124
58
209
94
58
-
-
-
45
30
0
max
45
>12?
1

Figure 3

Pixel from live input and same pixel from background picture are just a bit different: the algorithm will consider that it's just noise, and ignore it.

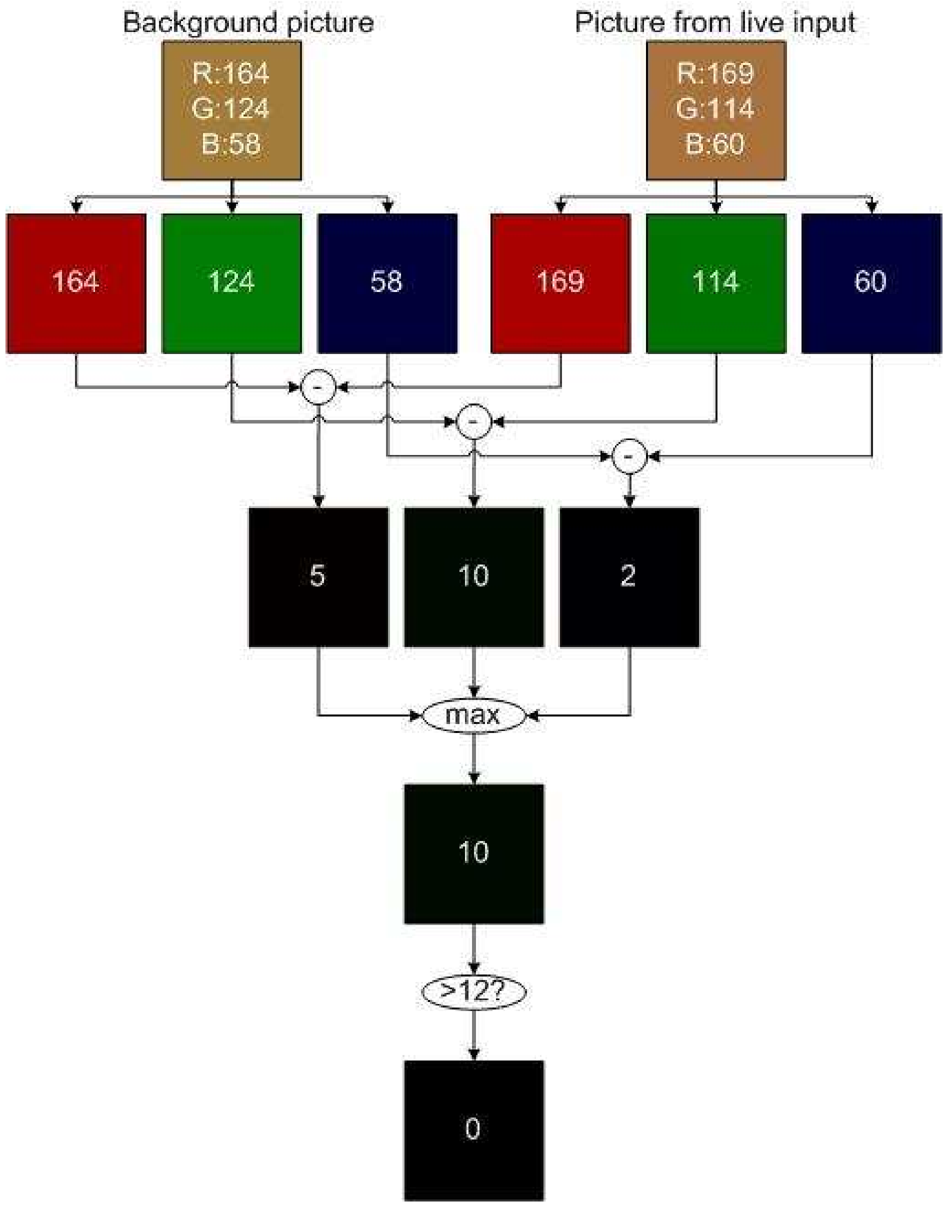

Figure 4

But a problem can occur if passer-by color and background one are nearby. In this case, program won't be able to distinguish him from the background.
To reduce this problem, we decided to make the threshold value as small as possible. But a new problem occurs, in this case. Working with a camera, pictures are not totally clean: there is some noise (even if there is no change on the scene, 2 successive frames aren't exactly the same).
With a small threshold value, obtained matrix is really noisy, so we decided to combine this technique with another one, convolution.

Reducing noise

Principle is to calculate, for each pixel, a value that is function of the values of its neighbors. The « kernel » is the given name of the matrix which indicates what this function is. In our case, we use a n-by-n matrix filled with value 1/n. So the value of each pixel is the mean of its n^2 neighbors.
Thresholding the obtained value by a high value, for example (n^2-1) if input matrix is a binary one, we can remove most noise pixels.
Look at these examples (assuming a binary matrix):

- if the considered pixel and most of its neighbors are white

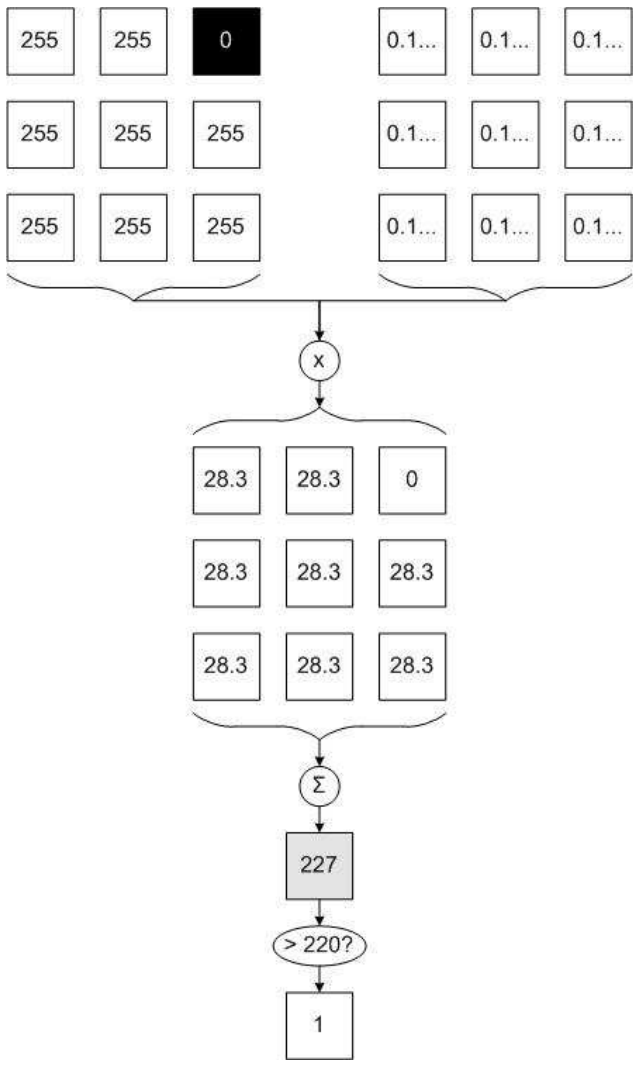

Figure 5

– in the opposite case :

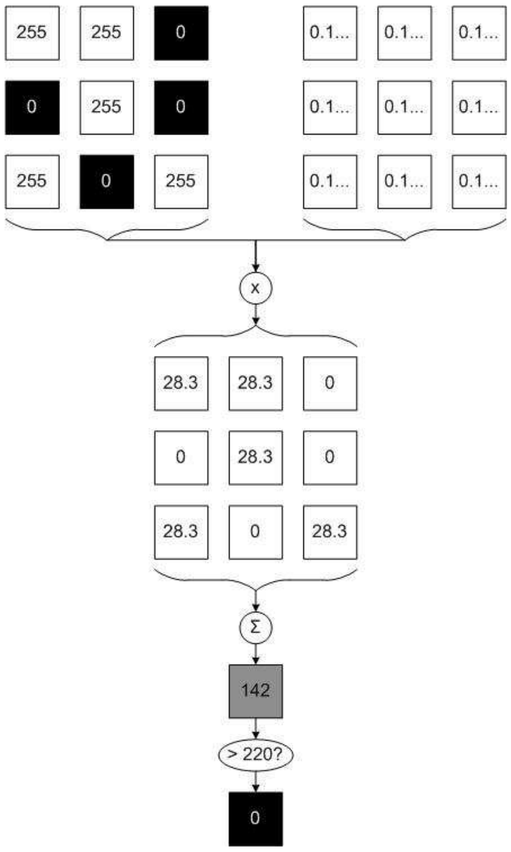

Figure 6

Updating background

We tried to imagine our project running for a long time, and thought that ambient light may change slowly with day. So we try to get some mechanism that updates continuously the background picture. The patch « background » takes the video in input.
The first frame is considered as background (so when you turn on the patch, make sure there is nobody in front of the camera).
For the next ones, it makes these operations:

– It computes the difference between the last background picture and the received frame.
– It thresholds the result with a value not too small, but not too big (12 for example).
– Pixels which have small difference with background picture are taken from live input.
– Other ones are replaced by those from the registered picture.
– The obtained picture is given to the cv.jit.mean, which computes the mean value on each channel of each pixel, over the time.
– It sends out the obtained picture.

We think this mechanism is able to obtain a background up-to-date, but we couldn't test it, because it needs to run patch for a long time.
We can note that enabling this patch reduce drastically the frame rate, so if the patch must be ran for only few time, we advise to turn it on for getting a background picture, and then to turn it off.

Detect immobility

Once the passer-by is detected, we need to detect his immobility.
To do it, we just verify that 2 conditions are verified:

- Somebody is behind the camera: We count the number of pixels that are considered as different from the background picture, and we verify that this number is greater than a threshold value.
- This person doesn't move: we subtract the number of different pixels between 2 consecutive frames, and verify that this number is smaller than another threshold value.

After several frames verifying these conditions, we can take a snapshot.

Get the border line

The next step is to get only the border line of the detected shape. We use the object « cv.jit.canny » that performs the canny algorithm.
It operates in several steps (sources: http://en.wikipedia.org/wiki/Canny http://fr.wikipedia.org/wiki/Algorithme_de_Canny):

- First, it convolves the matrix with a gaussian mask, to eliminate noise (obtained picture is a slightly blurred version of the input matrix). This is a classical gaussian filter.
- Next, it create a map of intensity gradients at each point in the image, by convolving it with Sobel operators (3-by-3 matrices : [[-1,0,1][-2,0,2][-1,0,1]] and [[1,2,1][0,0,0][-1,-2,-1]]). Highest values indicate a probable edge, and sign indicates its direction.
- It thresholds the result with hysteresis : with a first high threshold value, it defines some points with high probability to be edges. And with a second lower value, it try to complete the obtained edges, for having continuous lines.
- The result of this algorithm is a matrix, where only edges of big shapes from the input matrix are present.

2. Silhouette recognition and re-drawing

At the end of the first step, we get a sinuous and rough thin line, as illustrated in Figure 3.

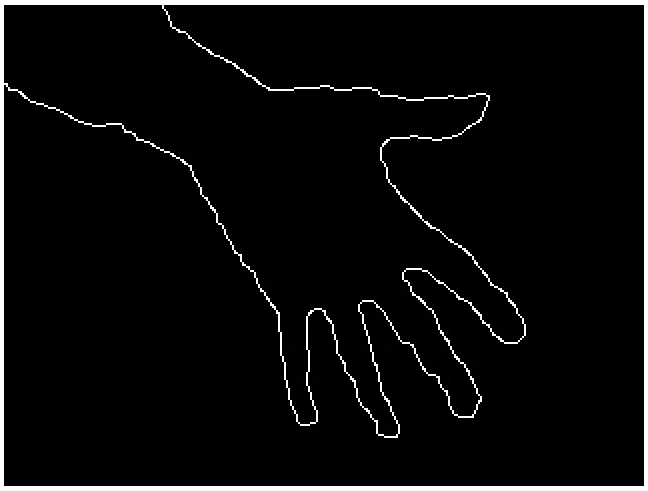
Figure 7: Original sinuous and rough thin line

Our artistic goal was to stylize this line. That means, choose some points on this original line and draw segments between them. The objective was to get closer to the crime scenes look, as illustrated in Figure 1.

Because we did not find a simple way to do so with Max/Jitter and because we had a keen interest in using Java into Jitter, we have implemented this second step in Java. This decision includes, of course, some studies and readings on *how to use* Java in Max/Jitter, for example:

- The *jitter-java api documentation* in MaxMSP 4.6/Cycling '74/java/jitter-javadoc/
- *Writing Max Externals in Java* v 0.3 by Topher La Fata
- The mxj javadoc of the com.cycling74.* packages
- The *Tutorial 51: Java Jitter* of the *Jitter Tutorial* Version 1.6
- Etc.

All Classes

Packages
com.cycling74.max
com.cycling74.msp
com.cycling74.net

com.cycling74.max
Interfaces
Executable
Classes
Atom
Callback
DataTypes
MaxBox
MaxClock
MaxContext
MaxObject
MaxPatcher
MaxQelem
MaxSystem
MaxWindow
MXJClassLoader
MXJDecompiler
MXJPreferences

Overview **Package** Class Use Tree Deprecated Index Help
PREV PACKAGE NEXT PACKAGE FR

Package com.cycling74.max

Interface Summary	
Executable	Executable provides a common interface for classes that need a method to ex

Class Summary	
Atom	An element of a list or argument of a message.
Callback	`Callback` is an implementation of the Executable interface.
DataTypes	This class contains constants describing the various data types that MaxObjec readable.
MaxBox	`MaxBox` represents the graphical element of an external in a patcher.
MaxClock	`MaxClock` provides a way for your mxj classes to set up the execution of eve
MaxContext	The context and company in which a Max object finds itself.
MaxObject	Base class for a Java Max object.
MaxPatcher	`MaxPatcher` can be used in conjunction with `MaxBox` to dynamically modify
MaxQelem	The MaxQelem object can be used to defer execution of a method from the t
MaxSystem	Collection of functions relating primarily to interaction with the Max environme
MaxWindow	`MaxWindow` is a reference to the window containing a patcher.

All Classes
JitterEvent
JitterListener
JitterMatrix
JitterNotifiable
JitterObject

Package **Class** Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAME
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAI

com.cycling74.jitter
Class JitterMatrix

```
java.lang.Object
  |
  +--com.cycling74.jitter.JitterObject
        |
        +--com.cycling74.jitter.JitterMatrix
```

public class **JitterMatrix**
extends JitterObject

`JitterMatrix` is a java wrapper for the Jitter Matrix object

Author:
Joshua Kit Clayton

Fields inherited from class com.cycling74.jitter.JitterObject
`_p_obj`

Constructor Summary
`JitterMatrix``()` creates a `JitterMatrix` with default planecount, type, dimensions and auto-generated name

Figure 8: Javadocs

Once we got used to the Max/Jitter Java API (Application Programming Interface), the process we have followed to achieve our goal (introduced above) was divided in two parts.

- Path finding including backtracking algorithm: because the input was just a flatten image without any meaning of silhouette (or even shape), we had to analyze the image in order to recognize and store each point.

- Re-draw the stylized silhouette using the Bresenham algorithm: once we get a real polygon (i.e., Java Polygon Object), we can easily choose which points to keep and draw segments between them.

Path finding including backtracking algorithm

The original image was of course just a matrix of black and white spots. In order to manipulate the silhouette as we wished, we had to make the computer understand that these black and white spots represent actually a continuous path (which defines the whole silhouette). To do so, we had to start from one white spot, look for another one in the surrounding area and so on until we reach the last one.

The Figure 5 below illustrates how we proceed. As it is said in the title of the paragraph, it includes a backtracking algorithm in the way that when a path reaches a dead end it comes back to the previous states and try to find another path.

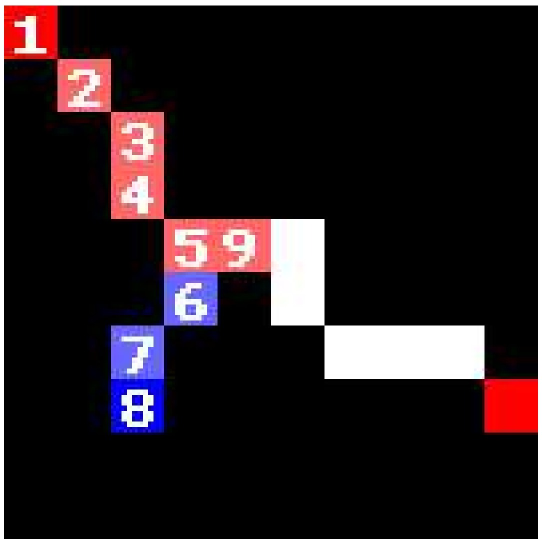

Figure 9: Backtracking path finder

Below are the signatures of the main methods. The first one returns a collection of polygons which have been found in the original matrix.

```
/**
 * Find a polygonal shape
 * @param jm The <t>JitterMatrix</t>
 * @return <t>polygon</t>
 * @throws Exception
 */
private Vector<Polygon> findPolygons(JitterMatrix jm) {...}
```

```
/**
 * Finds all the start points on the edge of the given matrix
 * @param jm The <t>JitterMatrix</t>
 * @return a <t>Vector</t> of <t>Point</t>
 */
private Vector<Point> findStartPointsOnEdge(JitterMatrix jm) {
```

In the signature below, we see that there is an attribute of type integer named *gap*. This is because the algorithm allows the path to contain some holes in it. We have implemented this extra-functionality because we realize that the *canny* object provides us with a discontinuous line sometimes. Moreover it offers more robustness to the system.

```
/**
 * Finds all the next points from the given start point in the given
matrix
 * @param jm The <t>JitterMatrix</t>
 * @return <t>true</t> if a path is found
 */
private boolean findNextPoints(JitterMatrix jm, Vector<Point>
currentPoints, int gap) {...}
```

Re-draw the stylized silhouette using the Bresenham algorithm

Once we finished the analysis part, we perform the synthesis step described below. At this moment, we have several polygons (i.e., Java Polygon Object). We could easily transform these polygons into something else, make some operations (translation, rotation, etc.) on them because it is no longer just a flatten image but a meaningful object recognized by the computer.

In our system, we only decided to skip some points and draw a line between the others. Once we get this processing done, we had to render the result into a matrix (i.e., black and white spots again). To determine which cells in the matrix will be white while tracing a segment, we use the Bresenham's line algorithm introduced below.

> *Bresenham's line algorithm is an algorithm that determines which points in an n-dimensional raster should be plotted in order to form a close approximation to a straight line between two given points. It is commonly used to draw lines on a computer screen, as it uses only integer addition, subtraction and bit shifting all of which are very cheap operations in standard computer architectures. It is one of the earliest algorithms developed in the field of computer graphics.*

Source: Wikipedia, http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

In other words, we studied this algorithm and translate it into Java language.

```
/**
 * Draw a line between two points
 * The Bresenham's line algorithm (...) is available here :
 *
http://fr.wikipedia.../Algorithme_de_trac%C3%A9_de_segment_de_Bresenham
 * @param jm The <t>JitterMatrix</t>
 * @param p1 The first <t>Point</t>
 * @param p2 The second <t>Point</t>
 */
private void drawSegment(JitterMatrix jm, Point p1, Point p2)
{...}
```

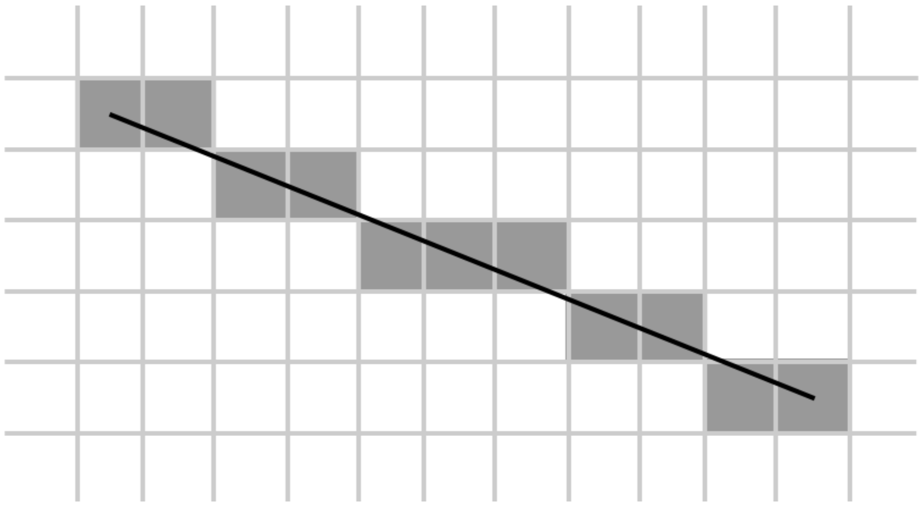

Figure 10: Bresenham's line algorithm

3. Final display with additional information

With the Java algorithm, we have a good silhouette stored in a matrix. The purpose of the step 3 is enlarging the silhouette, adding different informations on the silhouette and displaying the last persons “killed” by the responsive system. The number of silhouettes displayed can be specified by a number box.

Enlarge the edge

To enlarge the edge, we use the convolution. We implement the same object used during the step 1 with different parameters. We can specify the width of the edge.

Write the time

Just after retrieving the matrix from the step 2, we display the current time when somebody is “killed” by the responsive system. To perform that, the easiest way is using OpenGL Jitter object: jit.gl.text2d. We specify to this object the color, the position and the font of the text. We have to display the time in different position for each silhouette. To do that, we use a counter. When a new silhouette has to be displayed, a bang is sent to the counter and the output is incremented. The maximum of the counter is the number of silhouettes. The output of the counter is used for the vertical position. Like that, the time is not written at the same position for each silhouette.

Using jit.gl.text2d is not enough. Indeed, this object provides to write bitmap text with OpenGL but it does not render OpenGL. To perform that, we have to use jit.gl.render.

Display the last persons “killed”

All the last silhouettes are stored in a matrix. When somebody is captured by the system, the new silhouette is adding in this matrix. But it is better if we can see more easily the last silhouettes. Before adding the new silhouette, we multiply the matrix by a number between 0 and 1. This number is computed with this formula: $\sqrt[n]{0,1}$, n is the number of silhouettes displayed. Like that, after n silhouettes, the oldest silhouette is multiplied by 0,1 and with 0,1, the silhouette cannot be viewed.

There is not Jitter object to compute $\sqrt[n]{0,1}$. So, we have to implement this operator. To do that, we use a dichotomy algorithm.

Demonstration

The demonstration of our system was divided on two days: Thursday, December 7th and Friday, December 8th. During the first one, we show to a jury that our project woks correctly and during the second one, we took few pictures to include it in the report.

Problems encountered

For the first demonstration, we decided to project our video output on the wall because it was the easiest way to perform our system and to prove to the jury that our installation works correctly. Moreover, the Macintosh where our Max / Jitter patch ran was a G5. Indeed, our system needs a good processor, especially for the Java algorithm. With this computer, the number of frames per second reached 17. We never had a so high number of frames per second before.

The first problem that we encountered was the white balance and the auto-focus of the iSight camera. Indeed, we have to disable the white balance and the auto-focus of the camera because of the step 1 of our project: our patch performs the subtraction between the live input and the background and the settings of the camera have to be the same everytime if we want to have just the silhouette of passers-by without noise. After few tests, we pass to disable white balance and auto-focus and the demonstration was performed without problems: all the inactive passers-by in front of our camera were killed.

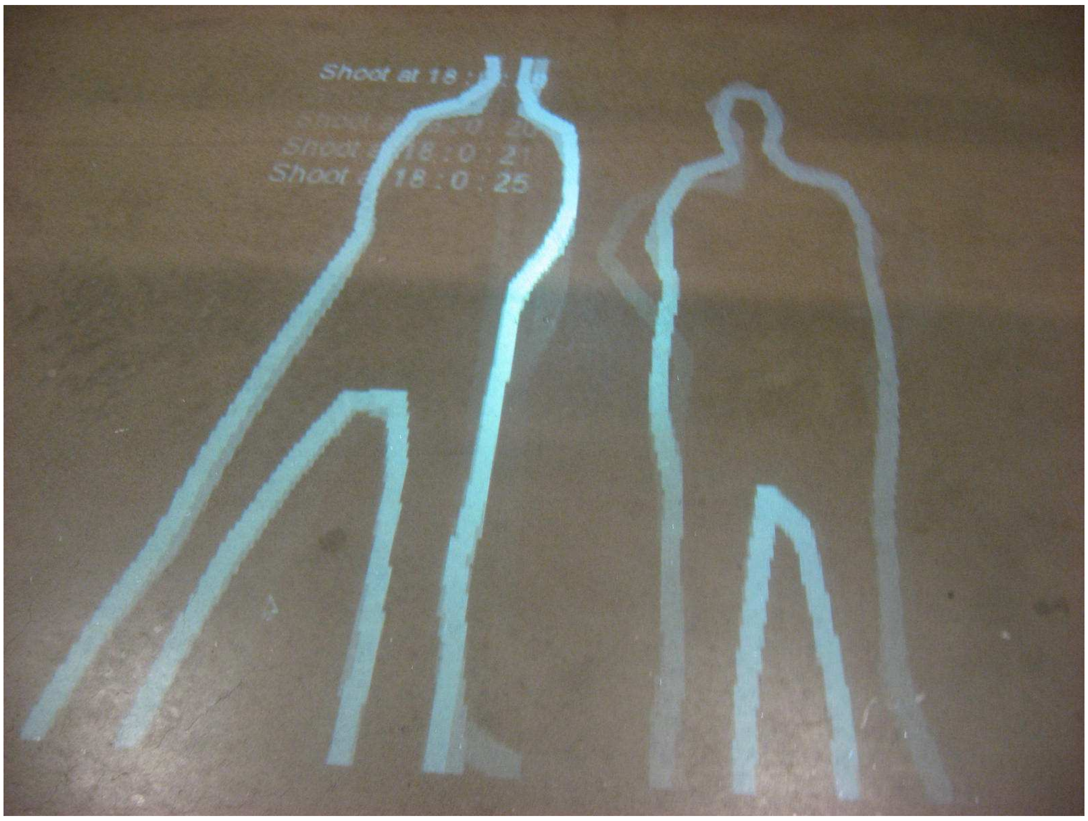

Figure 11: two innocent victims

For the second demonstration, we decided to project our video output on the floor because we wanted to reproduce a scene of crime movie. We did not do that on the first day because it is very hard to install the projector. Indeed, the projector has to be vertical. Finally, we found a solution with the stairs. We put the projector on the stairs, just at the top of our installation.

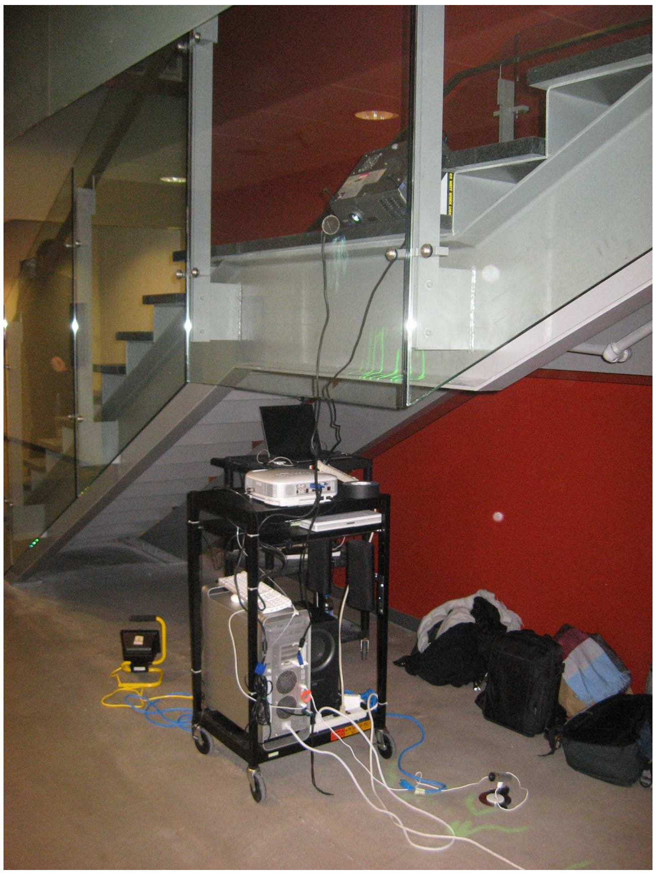
Figure 12: Our installation

With this installation, we encountered a new problem: the output was inverted. We tried to change the settings of the G5. But do not have good MacOS X skills and we did not pass to apply the good settings. So, we used a basic solution and we turned over the projector.

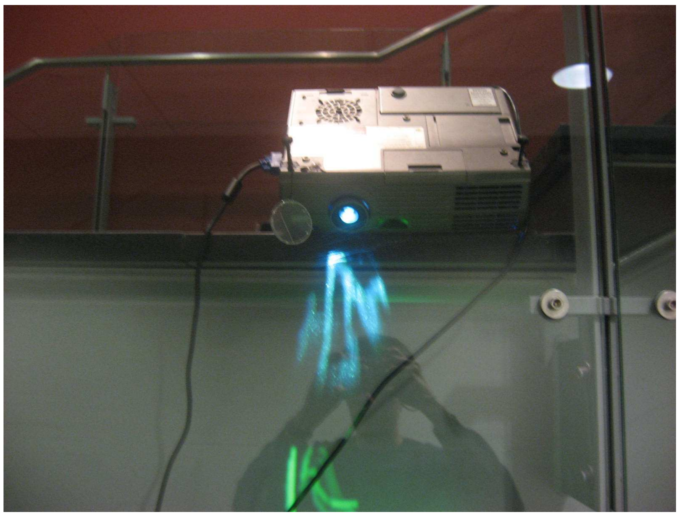
Figure 13: the inverted projector

The last problem that we encountered was the biggest problem: if there are shadows, our system can become useless. At the beginning of the step 1, the background is captured without shadow. So, when shadows appear, the subtraction between the live input and the background does not work very well and the system cannot recognize the silhouette. The only solution is being careful when we are in the front of the camera. Shadows do not have to appear. If we do not want to have shadow, the best way is looking at where the lights are. During the second day of our demonstration, we had this problem when the cameraman filmed us: there was too many people and these people did too big shadows and our system became inactive.

What did people think about our project?

Most of people like our system and play with it. For them, it was funny to be "killed" like crime movies:

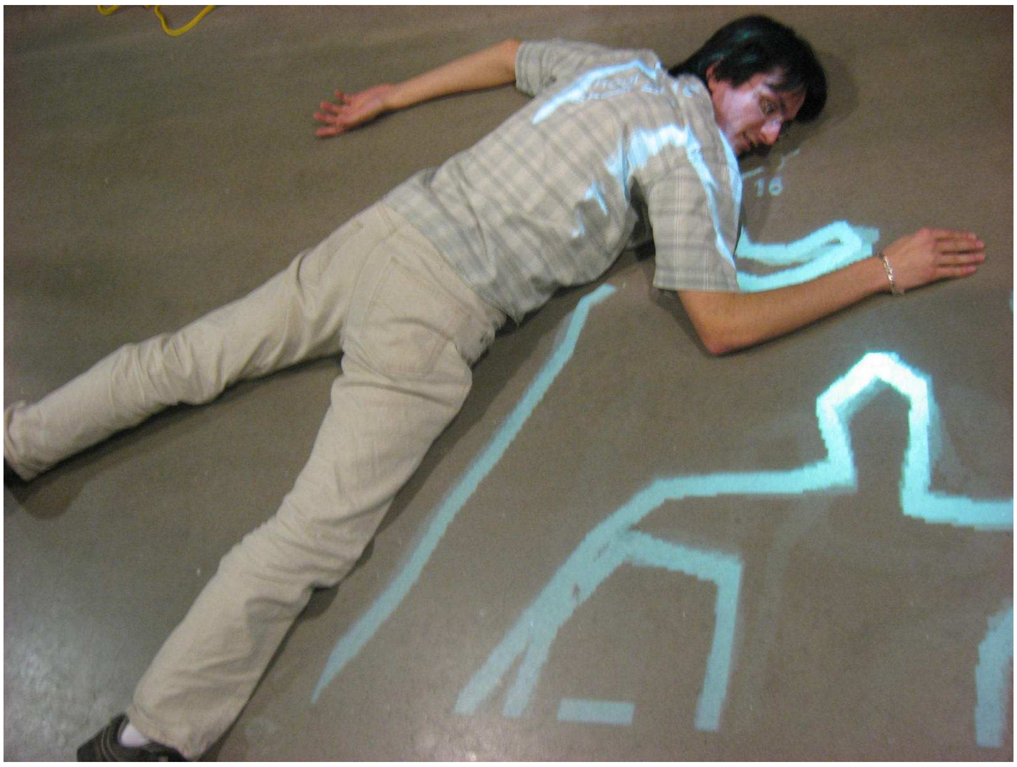

Figure 14: Lilian killed

Documentation

This project is documented in several ways :

- All patches, and java class, are all self-documented. They contain comments that explain how they work, how they can be configured, and everything you need to know for using them. The java class is also given with its own java-style documentation called JavaDoc. Every function is defined precisely with its parameters, its return values, and so on.
- A website (http://hybrid.concordia.ca/~g_duverg/) groups together all documents relative to the project, and offers many information about the concept, the development, some photos and videos of the patch, etc.

Work repartition

LILIAN

- Working on step 1: Motion capture and detection of inactivity
- Report parts: Introduction, Motion capture and detection of inactivity and Documentation
- Edge detection comparative tests

GEORGES

- Original concept
- Working on step 2: Silhouette recognition and re-drawing
- Report part : Concept presentation, Silhouette recognition and re-drawing and Conclusion
- Web site conception

LORIS

- Working on step 3 : Final display with additional information
- Report parts: Introduction of Development, Final display with additional information and Demonstration
- Report page setting

A step further

Other technical improvements

With hindsight, we think that it could be interesting to detect inactivity distinctly for each passer-by. In our current system, the entire scene in front of the camera (i.e., every passer-by) has to stay immobile for a while before we shoot. By using *blob*, we could detect the inactivity of one passer-by and just shoot him. This idea came to us during the demonstration days while more than one passer-by was using the system.

We also though that it could be interesting to improve the path finding algorithm in order to detect start points inside the matrix (instead of just on its edges). The reason why we have not done something like that was performance. By looking for a start point in the entire matrix, we reduce the performance of the system a lot. But there could be some tricks as check the diagonals of the image for example.

The two days demonstration as a reward

We think that the two days of presentation were really a good opportunities to push forward our project. Because we had something working quite well on the first day, we wanted to go a step further in our initial concept. That is why we decided to project the resulting image onto the floor (i.e., onto the crime scene). We had skipped this idea at the beginning because we thought it was logistically too complex to set up. Thus, we are glad that we have been able to do so.

Moreover, these two days of presentation, because of the passers-by feedback, have been a real reward of our work. It was really fun and enjoyable to play with our installation and, above all, to see our peers and other students do the same.

An achieved objective

To conclude, it has been a good experience for each of us. We are relatively proud of what we have done because it is quite exactly what we have planned to do since day one. Of course it is not perfect and, as we have seen, there could be some improvements but the concept of the initial idea and the mood was there (during the demonstration). Furthermore, the fact that we have been able to do everything we planned to, knowing that we did not know anything about Max/Jitter at the beginning of the year, implies that this course has been full of knowledge for us.

Figure 15: Loris, Georges and Lilian from Screen Crime project