VISUAL PATHS

Raphaël Sebbe [1], Michel Bagein [2], Lucie Bélanger [3], Ricardo Bose [4], Johan Dechistophoris [1], Jonathan Demeyer [1], Nathanaël Lécaudé [5], Céline Mancas-Thillou [1], Alexis Moinet [1]

[1] Laboratoire de Théorie des Circuits et Traitement du Signal (TCTS), Faculté Polytechnique de Mons (FPMs), Belgique
[2] Service d'Informatique (INFO), Faculté Polytechnique de Mons (FPMs), Belgique
[3] Département d'Informatique et de Recherche Opérationnelle (IRO), Université de Montréal, Canada
[4] Service d'Electronique et de Microélectronique(SEMI), Faculté Polytechnique de Mons (FPMs), Belgique
[5] Freelance collaborator of Daniel Danis, Arts et Sciences, Canada

ABSTRACT

The goal of this project is to build a vision-based system for creating real time curves in 3D space, to be used as input on scene, both for event generation and special effect rendering.

The system is composed of at least two cameras that are located in front of a 3D region of interest where infrared (IR) LED pens are agitated. The cameras are sensitive to IR, and a filter is used to cut off disturbing visible light. A camera calibration software based on geometric pattern recognition is implemented, whose role is to determine intrinsic (focal length and lens distortion) and extrinsic (relative transform against the pattern coordinate system) parameters, leading to a correspondence between image coordinates and scene rays in 3D.

Image processing is used to segment and locate the LED pen's center on each camera image (blob segmentation). GPU image processing is used to enforce real time performance, combined to more conventional CPU based higher level primitive handling. 3D rays are derived in scene space from 2D image coordinates and combined to retrieve the position of the pen in 3D.

Portable IR LED pens are built as ON/OFF battery powered units.

KEYWORDS

Image processing, real time, GPU, camera calibration, Wiimote

1. INTRODUCTION

This project was first discussed with Daniel Danis in early 2008, when he expressed a need for finding a position in 3D as an input to a real time digital arts system, for interaction with a 3D world and text-writing in 3D space. Some experiments at the time were already made with a single Wiimote camera, showing an interesting potential.

A first approach based on two Wiimotes being used simultaneously is investigated by Lucie and Nathanaël. After an initial calibration step that expresses the position of one remote relatively to the other one, 3D coordinates are computed and used for interactively drawing in 3D space.

Complementary to that approach, a more standard camera-based approach is investigated by the people in Mons, that makes use of Graphics Processing Unit (GPU) image processing for segmentation and a camera calibration step, based on pattern detection, to express 3D positions. GPU algorithm are becoming more and more common nowadays to solve data-parallel problems, because of the wide availability of hardware and the computing power they have. Moreover, the knowledge acquired when building this processing chain can be further leveraged in forthcoming `numediart` projects.

This document covers all aspects of building such systems, from building an IR pen, to image acquisition, processing, data visualization and programming details.

2. SYSTEM OVERVIEW

2.1. Camera-based System

Analog CCTV cameras are connected to computers that perform the segmentation of the IR blob in the image. The center of the blob is expressed as a 3D ray in space from a previously established camera calibration model, for each camera. When at least two cameras are used, a 3D position can be computed in space. Using more than two cameras can

Figure 1 shows the system diagram, where two camera-processing computers are connected to a third one that performs visualization. The system is modular, from two cameras on a single computer to N cameras on M computers ($M \leq N$).

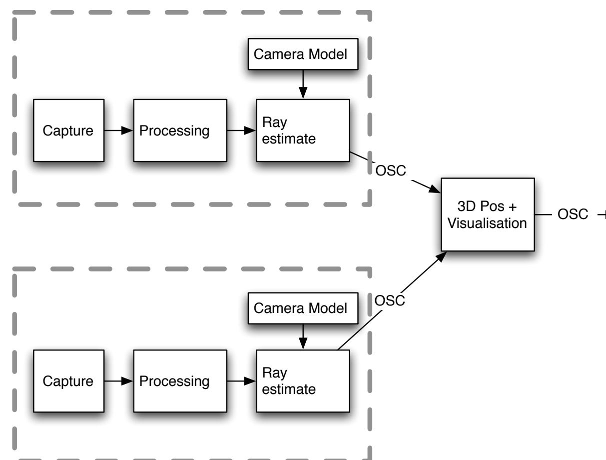

Figure 1: Entire System Overview

2.2. Wiimote-based System

The objective of the project is to track 3D handwriting in space using a virtual pen and a set of cameras. It has been decided as

a first prototype to use Nintendo Wii remotes (or Wiimotes) and infrared (IR) LEDs to achieve such a goal. This system has been chosen over cameras because of the low cost of the material and the ease of use: possibility to connect multiple remotes to a single computer, processing done on the remote, possibility to operate without wires and small form factor.

The prototype uses Nintendo Wii remotes as the acquisition system. A Wii remote can track up to four IR sources at a rate of 100 Hz, this serves perfectly our application.

In order to produce a 3D handwriting system, we use a minimum of two Wiimotes. All Wiimotes connect wirelessly to a computer using the Bluetooth protocol. All Wiimotes can be connected to the same computer which facilitates the deployment of the system in real world performance conditions. Once all Wiimotes are calibrated, we can reconstruct the hand-gesture in realtime in 3D coordinates and visualize it through Open Scene Graph (OSG, [6]).

3. IR PEN

3.1. Mons Pen

The first prototype was made with one IR LED (940 nm), a microswitch, a long supply cable and a small “bread board". A piece of “tracing paper” was used to diffuse the ir beam. The second use a double AA battery holder. Five IR LEDs (940 nm) connected in parallel, a microswitch and only one rechargeable NiMH AA 1,2V / 2700mA/h. The LEDs are placed like a cross to improve directivity. Leds 2 to 5 are 45°-oriented from the center LED. A LED consumption is ±20mA / 1.20V, ±30 mA / 1.25V or ±40mA / 1.30V.

3.2. Montreal’s Pen

The prototyped IR pen uses IR LEDs, a battery and an electronic switch. The pen uses three 1.5V LED at 940nm and 60 degrees viewing angle. It is powered by a standard rechargeable AA 1.5V battery. It has been found that polishing the LED using sand paper widens the viewing angle slightly. An 850nm LED as also been tested but the wavelength does not seem compatible with the Wii remote.

4. IMAGE ACQUISITION

In the first approach, images have to be captured from camera in real time.

4.1. Camera Setup

Different cameras are experimented, from built-in computer webcams to CCTV camera. Cameras are used with and without visible light cut filter. Visible light is needed in the calibration step where the algorithm needs to see the pattern, while it needs to be avoided when using the IR pen.

Visible light is removed with a regular pass IR filter, but we also built one ourselves by using a developed photo film negative that was exposed to artificial light only (light with no IR). The negative will then block visible light and will not affect IR.

Built-in webcams operate through USB bus while CCTV cameras are analog and must be converted to digital stream. The model of CCTV camera is Eneo VKC-1354 and the lens is a varifocal 2.4 - 6mm focal length, allowing normal to wide angle experiments. Analog to digital conversion is made with Canopus DV-55 converter, and provides DV stream on Firewire. DFG from Imaging Source and external, low cost USB webcams were also thought of for improved latency but no experiment in that direction is made at this time.

4.2. Video Capture with QuickTime on Mac OS X

Camera capture on Mac is handled with the QTKit framework, based on QuickTime, which provides abstraction to image input devices. Connected devices are enumerated (with hot plug capability), and a capture connection is set up. Images are captured at PAL resolution (720x576) with a framerate of 25 fps. Different kinds of cameras are supported (IIDC Firewire, USB webcams, Analog through DV converters).

4.3. Video Capture with V4L2 on Linux

V4L2 is a video capture API for Linux for several devices as USB webcams, TV tuners, and other, but doesn’t support Firewire cameras well. Linux1394 is used instead, it is a library in conformance of IEEE1394 standard (Firewire links). The first step was to develop a basic program based on libraw1394 and libdc1394 (both parts of Linux1394) enough for Firewire communication and frame grabber from analog video cameras over that protocol. The program pieces can be included in other software, making it easy to capture video. The second step follows the same idea, but allow the communication and video capture from digital video cameras over Firewire. Still in working progress, it uses libraw1394 and libdv and parts of Kino, an open source video editing software for Linux, to be able to work with cameras and images. Both programs will be used for any other future application about video cameras on Linux.

5. CAMERA CALIBRATION

Camera calibration is the process of expressing internal camera parameters such as its focal length, and external parameters such as its 3D location relatively to a known reference coordinate system. This step is necessary as at least two cameras are used, and their information is merged to produce a 3D position.

5.1. Simultaneous Points-based System (Wiimotes)

5.1.1. Intrinsic Parameters

The Wiimotes are only sensitive to IR light and can only track four points at a time, which makes the traditional grid calibration algorithm a difficult approach. For the sake of this project, the intrinsic parameters of the Wiimotes were estimated based on the viewing angle of the camera and the resolution of the sensor. We know the resolution of the sensor is 1024x768 and we estimate the viewing angles to be 41 degrees in the horizontal axis and 22 degrees in the vertical axis. Although this technique does not allow a precise estimation of the internal parameters, it proved to be sufficient in the context of the project.

5.1.2. Extrinsic Parameters

To calibrate the extrinsic parameters, we use a calibration algorithm based on epipolar geometry. When two cameras are looking at the same scene, a point in an image maps to an epipolar line in the other image. An epipolar line is the projection of the ray

formed by the image point and the camera center into the other camera. The fundamental and the essential matrices allow to represent this mapping between two cameras and we can then compute the pose of the camera relatively to the other.

To compute the fundamental matrix, we use the normalized 8-point algorithm as described by Hartley and Zisserman [3]. We take a set of n normalized image points, x_i' and x_i, that are viewed by both cameras. The fundamental matrix is obtained by solving a linear system such that $x_i'TFx_i = 0$. From the fundamental matrix and the intrinsic parameters matrix, we can compute the essential matrix with $E = KTFK$. From the essential matrix, we can extract four possible solutions for the pose. The pose of one of the two cameras is the identity and the origin of the world coordinates. We need to find the pose of the second camera relatively to the first camera. Since we have a priori knowledge about the camera, its orientation and translation direction, we can extract the pose corresponding to the second camera.

5.2. Grid Based Calibration

For the regular camera approach, a full calibration is performed as expressed in Zhang [8]. It consists of showing a chessboard-like pattern to the camera and perform corner detection, each detection resulting in a set of structured points for which relative positions are known (square size expressed in millimeter is a parameter). Showing multiple instances of this pattern enables the determination of intrinsic parameters, while choosing a specific pattern image determines the six degrees of freedom (DOF) of the camera in respect to that specific pattern location. If the same pattern is used for that last step for each camera, their positions and orientations are then expressed in the same coordinate system.

5.2.1. Intrinsic Parameters

Intrinsic camera parameters are

- the focal length expressed in pixels for x and y (2),
- the coordinate of the optical axis in the image (2),
- the linear skew coefficient (1),
- the non linear distortion coefficients, radial and tangential (4).

They do depend on lens configuration for varifocal lenses (that we used) but they do not depend on camera position (see next section).

The method used to evaluate those parameters is the one present by Zhang [8], and OpenCV implementation is used. Multiple pattern configurations must be captured by the camera and processed in the optimization method, leading to computed values for the nine values. This has to be done once per camera (if the lens is not changed).

5.2.2. Extrinsic Parameters

Extrinsic parameters express the camera location / orientation in a reference coordinate system. The 6 DOF are generally expressed as a 3D translation and rotation vectors, or more generally as a 4 by 4 transformation matrix (homogeneous).

Evaluation of the extrinsic parameters is made after the intrinsic evaluation, and only for a single pattern. The translation and rotation vectors express the position and orientation of the camera in respect to the coordinate system of the pattern. This coordinate system is defined as x and y as the pattern axes, and z as orthogonal to these (Figure 2).

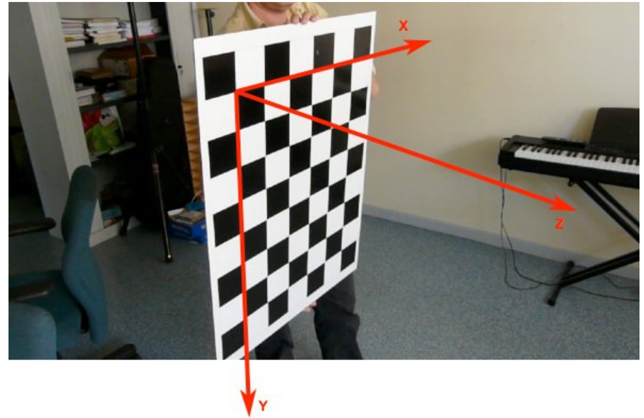

Figure 2: Pattern Coordinate System

6. IMAGE PROCESSING

For years image (and video) processing algorithms have been using only the power of the computer central processing unit (CPU). Many image processing software perform heavy calculations with the CPU which is not well suited to image processing and has to be shared with other applications.

On the contrary graphical processing unit (GPU) have a great deal of image processing computational resources which are often under-used.

These last few years, a new trend has emerged which attempts to use that GPU's power to achieve general purpose computation (GPGPU). In the domain of image and video processing, this guiding principle leads to applications performing real time frame processing where their CPU equivalents would last one second or more by image.

We decided to follow that direction in order to make real time detection of the IR pens position on the video streams acquired by the cameras.

The next sub-sections explain the algorithm we implemented and its Core Image implementation on Mac OS X as well as its GLSL cross-platform implementation.

6.1. GPU-based blob segmentation algorithm

We implement a variation of an algorithm for object detection described in [5]. It consists in three steps, all implemented as shader in the GPU programmable pipeline [7].

The first step extracts the IR pen light, removing noise and other possible IR sources (e.g outside and ambient lights...). For each video frame, every pixel's color is compared with a threshold value. When a color is above the threshold, its *RGBA* value is set to $(1.0, 1.0, 1.0, 1.0)$, otherwise it is set to $(0.0, 0.0, 0.0, 0.0)$. This results in a *mask* similar to Figure 3.

Then we find the IR pen's position by computing with (1) the center of gravity of the mask (considering each white pixel as a unitary mass and each black pixel as vacuum).

$$\begin{pmatrix} c_x \\ c_y \end{pmatrix} = \frac{\sum_{x,y} m(x,y) \begin{pmatrix} x \\ y \end{pmatrix}}{\sum_{x,y} m(x,y)}, \tag{1}$$

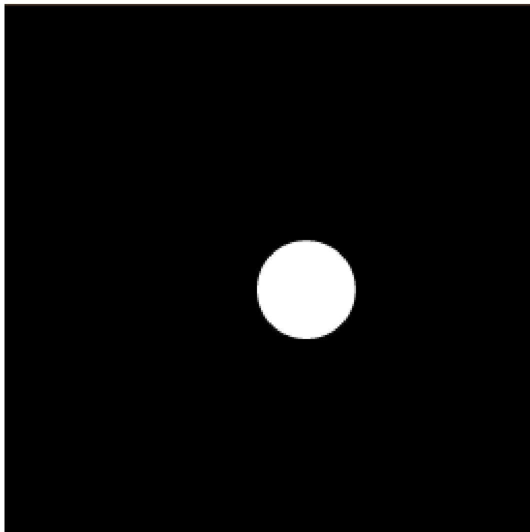

Figure 3: Resulting mask after thresholding of the original image.

where $m(x, y)$ is the value (0 or 1) of the mask at pixel coordinates (x,y).

The second step of the algorithm computes the value of every $m(x, y)\begin{pmatrix} x \\ y \end{pmatrix}$. For every pixel, the first and second RGBA parameters are multiplied by the coordinates of the pixel. This is equivalent to a pixel-wise multiplication between the mask and an image whose pixel's RGBA color is $(x, y, 1.0, 1.0)$ (see Figure 4).

The resulting image is a mask weighted by the coordinates of the pixel. If we were to sum all the RGB values of the image, we would obtain the two elements of the numerator in (1) as R and G and the denominator as B.

Instead of slowly computing that sum on the CPU, we enter in the third step of the algorithm which uses a shader that reduces the image size. This shader is "applied" in a loop until the resulting image is a one-by-one pixel. The RGB value of that pixel will contain respectively the two numerators and the denominator of (1).

Due to the linear interpolation ability of the graphic card, we can easily create a simple shader able to downsample by a factor of four. Figure 5 is an illustration of its principle. Let us denote the input image *I* and the output image O,

- For every pixel of O (black $\times$ in Figure 5) we multiply its central coordinates (x, y) by 4 to obtain the center of the corresponding 4×4-pixel subset from *I* (white circle in Figure 5). (e.g. $(0.5, 0.5) \rightarrow (2.0, 2.0)$)
- In image *I*, we consider four pixels located at $(+1, +1)$, $(+1, -1)$,$(-1, +1)$ and $(-1, -1)$ from that center. Thus we obtain the position of four points located in the center of 2×2-pixel blocks (4 black circles on Figure 5).
- Since these points are positioned between four pixels, their corresponding RGBA values are automatically obtained by linear interpolation of the surrounding pixel colors.
- Finally we compute the mean value of the 4 resulting RGBA values to obtain the mean color of the 16-pixel block which is then affected to pixel (x, y) of image O

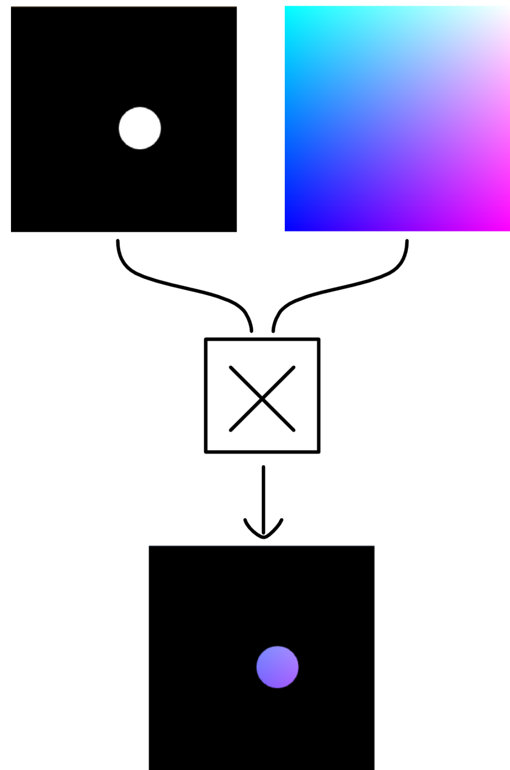

Figure 4: Resulting image after the mask has been weighted with pixel coordinates.

Note that if the video frame is not power-of-two-sized, its size can be increased to the closest power-of-two dimension by filling its *mask* with black pixels. Moreover, if the image is not a square, the result will not be a 1×1-pixel image and it will need additional downsampling in the longest-side direction (horizontal or vertical).

6.2. Core Image implementation

To segment the LED area appearing on each frame of the videos in order to track the artist or the object, we use the Core Image technology. One of the main reasons is the easiness to code GPU-based (Graphics Processing Unit) algorithms, whose aim is to be very fast because driven by the graphical card, contrarily to CPU-based (Central Processing Unit) ones.

6.2.1. Core Image Technology: a Brief Overview

Core Image technology greatly enables to build real time or near real time image processing algorithms. Available on Mac OS X platform, it gives access to built-in image filters for both videos and still images but also enables the creation of custom filters, whom we will give details in the following subsections. Core Image technology is built upon OpenGL specifications using a language subset, based on GLSL (OpenGL Shading Language), called CIKL and standing for Core Image Kernel Language. It is of primary importance to cite the main advantages of the Core Image API:

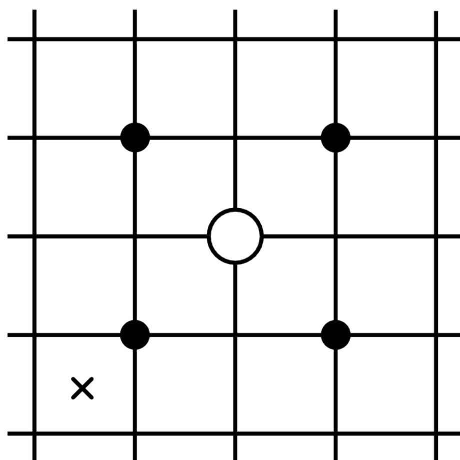

Figure 5: Downsampling from 16 pixels to one in a single pass.

- Based on OpenGL GLSL, which is compatible for all platforms, each code could be easily cross-platform with few additional modifications.
- As it is at a higher level than basic graphics processing, programming it is easier.
- Moreover, it adds a supplementary wrapper in order for your code to work even if a GPU is not available (CPU fallback)
- Although multiple filters can be chained together, the actual image processing is the concatenated compiled chain which ensures a faster algorithm. No intermediate rendering is done: this is called a "Lazy evaluation model" and makes your filters fast and efficient.
- For security reasons, it makes the distinction between "CPU-executable" and "CPU-non executable" filters. The first ones may run on the CPU whereas the second ones only on GPU, which is safer because running on a restricted environment, protected against security threats. For more details of implementation, the reader may refer to [2].

Each filter usually has one or several input and output images and in-between, the most important part is the kernel, which maps after calculation pixels of a source image to those of a destination one. In a general use, for each kernel to code, an image sampler needs to be defined, by specifying a coordinate transform, an interpolation mode between processed pixels and a wrapping mode when destination and source areas to process are different, because Core Image performs operations in a device-dependent working space. Note that if source and destination areas do not coincide, only CPU-executable filters may be created.

GPU-based programming is a bit tricky using Core Image technology. As the destination rendering is the final operation, it is based on inverse mapping of each pixel through filters. The following subsections will detail how to code that way.

6.2.2. Core Image Programming

For both types of filters (CPU-executable or not), the kernel code has to be programmed to create a custom filter. For the second category, filters need to be packaged as part of an image unit, using XCode, the development environment of MacOS. Several image units, which are built-in kernels, are also available but are out of scope of this paragraph. As a kernel operates on pixels using the GPU, the code needs to be straightforward to be efficient (meaning no loop is allowed as no accumulator on pixels is available). A kernel routine must return a 4-element vector (for red, green, blue pixels and the alpha channel) and maps the source pixels to the destination ones. Algorithm 1 shows a kernel routine to create a mask from a defined color:

Algorithm 1:

```
kernel vec4 maskFromColor(sampler image,
__color color, float threshold)
{
    float  d;
    vec4   p;

    // Compute distance between current
    // pixel color and reference color
    p = sample(image, samplerCoord(image));
    d = distance(p.rgb, color.rgb);

    // If color difference is larger than
    // threshold, return black.
    return  (d > threshold)  ?  vec4(0.0)
     :  vec4(p.a);
}
```

Input parameters are the source image itself, the defined color and a threshold to apply on the image. The same sampler is used for the output 4-element vector p, meaning the source and destination areas coincide. Then a distance is computed between the source pixels and the defined color. This function is created before and computes a 3D Euclidean distance. Finally, if the distance value for each pixel is larger than the threshold, the output pixel is black, otherwise, the source color values of the pixel is kept. For more details of kernel routine creation, the reader may refer to [4].

Thanks to the Quartz Composer (QC) software, the kernel routine may be easily prototyped by pasting your code inside a "Core Image Filter" patch. Input parameters may be set up by other patches or by the user directly with default values. For this straightforward example, the final step is to connect a "Billboard" patch to display the output image.

Finally, to create an independent custom filter, you need to declare an interface for the filter inside XCode (meaning the declaration of input parameters), to write an Init method for them, and to write an output image method to get the result after kernel computation.

6.2.3. Our Custom Kernel-based Segmentation

After these brief explanations on Core Image technology, here are the details of our custom Core Image kernel-based segmentation:

- Create a mask from color to segment the LED area appearing on each video frame by thresholding pixel values,
- Multiply mask with individual pixel coordinates,

- Compute the centroid from the mask area by using CIAreaAverage filter (built-in Core Image),
- Get some information about depth by using the area of the segmented blob.

6.3. GLSL Implementation

The program is currently implemented in C and we use GLUT as a working environment to test the different steps of the algorithm. However it can be easily ported to C++ for an eventual integration within TiCore. Moreover GLUT being quite limited for user interactions, the final version will also use a more GUI-oriented framework like GTK+.

Each frame to be analyzed is loaded in an OpenGL texture which is applied to a rectangular quad in a *viewport* whose dimensions are power-of-two values[1]. As explained in section 6.1, all the pixels around the rectangular quad are black-colored.

Note that the RGBA color space of each pixel is stored in a set of four 32-bits floating point values. This is a very important point since we store all the parameters needed to compute equation (1) as pixel color values. Floating point allows the storage of a very large range of values, contrary to 8-bits integer values usually used to store color parameters.

The first and second parts of the algorithm, namely the mask computation and its multiplication by pixel coordinates are implemented in the same fragment shader [7] (file "*threshold.frag*" in the source code). This shader is applied to the viewport and the resulting image is mapped in a frame buffer object (FBO) texture instead of being displayed.

The last part of the object detection is the downsampling which is implemented in a second fragment shader ("*down.frag*"). This shader is applied to the FBO texture which results in a smaller image stored in another FBO texture.

The shader is applied again to the new texture, and so on. Each iteration of the loop divides the size of the image by a factor of four and the loop stops when the accumulated downsampling factor is equal to the largest side of the viewport.

Note that all the FBO textures have the same size as the initial viewport. Therefore the viewport is filled with more and more black pixels after each iteration. Thus the final texture will be full of black pixels with only one non-black pixel (located at $(0,0)$) containing the numerators and denominator of (1).

Consequently, in the case of rectangular images, one-dimensional final downsampling explained in 6.1 is unnecessary because a $1\times$N image is completed with the surrounding black pixels and thus it can be seen as a N$\times$N image to be downsampled until only one non-black pixel remains.

7. 3D POSITION ESTIMATION, NETWORK COMMUNICATION

The 3D estimated ray, by motion camera or Wiimote, is based on the real characteristics of the optical sensor, calibration and spatial location. All those parameters are subject to imprecision and then, different rays, produced by several cameras, from a unique spot light do not cross on a unique location. The crossing ray problem can be solved in two steps.

[1]Not necessarily a square though, it can be rectangular.

7.1. Mathematical method

7.1.1. *Nearest Location between two Rays*

In a 3D context, the condition to identify the crossing location of two rays is that the rays are on the same plane. This condition is rarely met due to the rounding errors and camera model imprecision, leading to skew lines. The nearest point between both lines is the best estimate, and corresponds to the middle point of the line orthogonal to both rays.

7.1.2. *Generalization to N Rays*

The idea is to find the point nearest to N lines in the sense of least squares. The squared distance between a point $\bar{p} = (x, y, z)$ and a line i passing through $\bar{p}_{i1}$ and $\bar{p}_{i2}$ is given by

$$d_i^2(\bar{p}) = \frac{|(\bar{p}_{i2} - \bar{p}_{i1}) \wedge (\bar{p}_{i1} - \bar{p})|^2}{|\bar{p}_{i2} - \bar{p}_{i1}|^2}. \tag{2}$$

For N lines, the total squared distance is:

$$d_T^2(\bar{p}) = \sum_i d_i^2(\bar{p}), \tag{3}$$

where i is the line index.

The value of $\bar{p}$ for the nearest point is the one corresponding to the minimum of equation 3. By deriving this expression according to x, y, and z, we can solve this linear system to find that specific point.

7.2. OSC Implementation

The Open Sound Control (OSC) protocol over UDP is used to transmit ray coordinates from each camera to the crossing ray application. Each network frame from camera contains an identification header (name) followed by a list of 6 floats (ray coordinates). To simulate a camera live context, several camera servers send sequences of ray frames 30 times per second with a random 3ms jitter. At each received frame, the crossing ray application computes a new estimation of the target location and sends its coordinate (3 floats) on the network under UDP broadcasting.

The broadcast is justified by the possibility to connect different viewers (3D viewer, recorder, console, etc.), acting as plug and play clients, without impacting the overall network load.

7.3. Computer Discovery with Bonjour

7.3.1. *Concept*

Bonjour is an implementation of Zeroconf (Zero Configuration Networking) given by Apple. Zeroconf is a service discovery protocol which allows devices to show up and connect to other seamlessly. Bonjour works on Mac OS X and Windows (There is also an implementation for BSD and Posix but it's not as "high-level" as for Mac OS X or Windows).

There are three steps for two devices to communicate together: The publication, the discovery and the resolution.

The publication is the process where a device will register a service. Basically, the service broadcasts four informations: the service name, the service type, the host name and the port number (Figure 6).

If a name collision on the local network occurs, a Bonjour host finds a new name automatically (in the case of a device without a screen) or by asking the user (in the case of a personal computer).

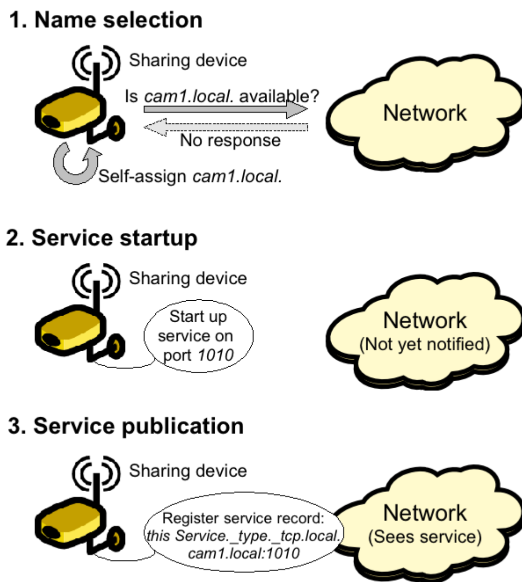

Figure 6: The publication process.

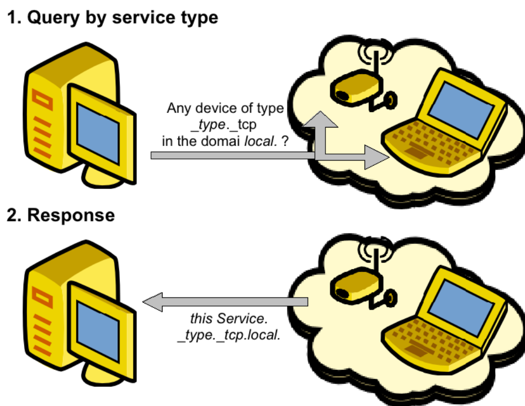

Figure 7: The discovery process

The device has to resend periodically its information in order to refresh name servers' lists.

When a service is accessed, a device needs to discover who offers the service. The device will send a request to the network to get back all the service names that corresponds to the service type (Figure 7).

When the discovery is done, the client does not already know the host name and the port number of collected services. This way the client can access the service even if its port number, IP address or host name changes, as long as the service remains the same (Fig 8).

If the resolution fails the device who sends the request tells the others that the service is no more available.

Bonjour can browse domains in order to find or publish services outside the local network but we haven't needed to investigate this functionality for our application.

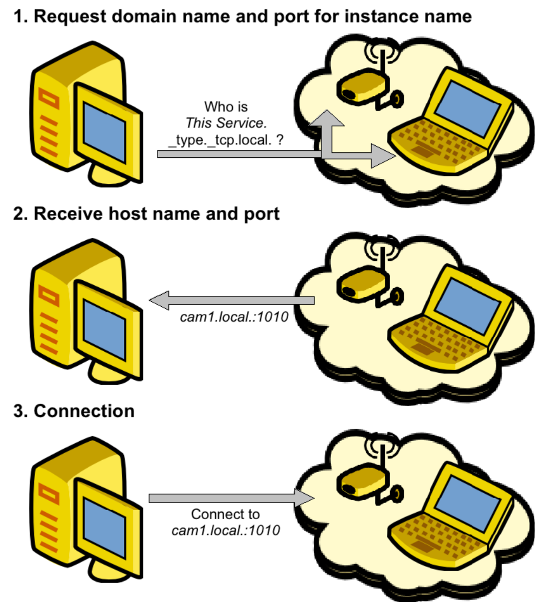

Figure 8: The resolve process

7.3.2. Implementation

For our purpose, the application handles a camera that needs to know where to send the position through OSC packets. In this configuration, the former application is the client. The application who synthesizes the information from different cameras is the server[2].

From the server side, the application will publish a service of type *vposc* specifying the port number and the transport protocol (TCP or UDP). A name for the service can also be given, if not provided, the system automatically advertises the service using the computer name as the service name. The publication done, the server will wait for connections.

Each document created in the client can send its OSC packets to an arbitrary destination. When the Bonjour search is activated, a thread browses the network to find a service whose type is also *vposc* and logs all the services names available.

When the user selects the service where he wants to deliver informations, the third phase comes up. The application resolves the service name for a hostname or an IP (and the port number) and update the configuration of the OSC packets transfer.

8. VISUALIZATION

8.1. Blender Based Interactive Environment

The main 3D viewer is build with Blender [1]. Blender comes with many advanced features. In our application we mainly use the graphical 3D editor (based on the OpenGL lib), the real time 3D game engine (including collision dynamics between different 3D objects), and the Python scripting on each object events (video frame, object collision, keyboard, mouse...). The proposed scene illustrating a 3D control is based on a hemispheric scene with two balls. The small one (grey moon ball in the centre of the scene) is controlled by the crossing ray application and the large

[2] It is thus the opposite of the diagram in the previous section

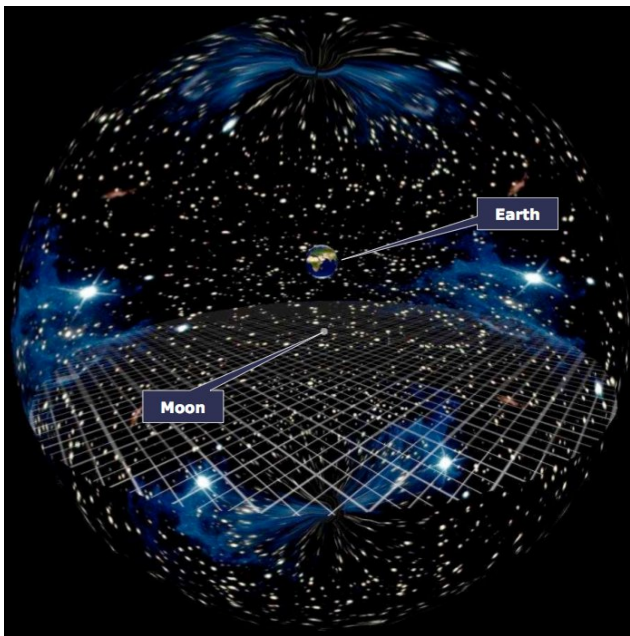

Figure 9: Blender Game Engine Running our Demo

one (earth ball) reacts as a windbag (Figure 9). Communications between crossing ray application and blender is written in two Python scripts (network initialization and moon positioning). At the Blender demo start, the Earth is unlocked and starts to fall down on the white grid (gravity effect). The controlled moon can then knocks earth, each collision, from the moon object or the scene, is detected by the earth object and its speed, direction and rotation are updated.

8.1.1. Broadcasting 3D object location

The earth location is determined by the main 3D viewer. To reflect the earth location towards other viewers, a Python script, called at each video frame, broadcasts earth coordinate on the network. Depending of the 3D scene complexity, the fastest rate of the data stream is generally limited by the video frame rate (60 Hz for a usual LCD monitor).

8.1.2. Network streaming aspect for a viewer

With a unique scene video projection, it could be hard to estimate the distance between the spot and the point of view (POV). Multiple points of view (static POV as top, front or side, dynamic POV as head or hand POV, etc.) can help the performer to locate himself in the virtual scene. Each slave 3D virtual monitor, plugged on the network, listen the data streams (moon and earth location-orientation object). It is unimaginable to synchronize data streams between producer and consumer, but each time a consumer needs data, it must be certain to read the latest network frame. In usual implementation (e.g. socket), each data received by the network adapter is stored in queue (FIFO buffer): the latest data is then always at the input gate of the receive queue. As the reader is always connected to the output gate of the receive queue, it has to empty the queue until it can extract the latest frame.

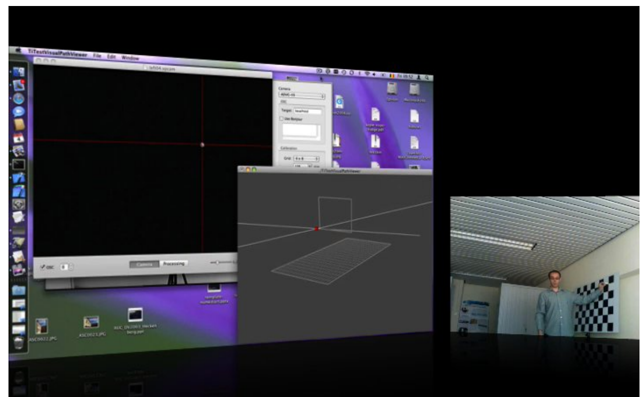

Figure 10: Ray representation in 3D with Open Scene Graph

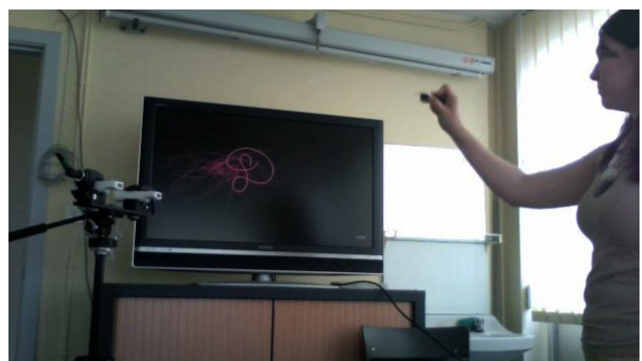

Figure 11: 3D Pen Drawing with Osg Lua from Wiimotes

8.2. Open Scene Graph Ray Visualization

Open Scene Graph (OSG) [6], is a cross platform open source library whose purpose is to render 3D objects that form a hierarchy (or graph), that is built on top of OpenGL. It is used here to represent the 3D rays that are computed by the system for each camera. A screen capture shows the system in action on Figure 10.

The rays coordinates (6, 3 for position, 3 for direction) are sent through OSC. The pattern used for extrinsic calibration is also displayed as a vertical rectangular frame, and the estimate 3D position of the pen as a red sphere. This position can be observed in real time as the pen moves and the view angle of the scene can be changed with the mouse.

8.3. Open Scene Graph Visualization using osgLua

In the context of plays and performances, directors and actors often wish to make fast visual tests or changes. In this situation, a fast and flexible development platform is required. For the visualization with the 3D handwriting using Wiimotes, we decided to use Open Scene Graph. In order to efficiently manipulate the 3D environment and to control it from multiple computers, we use a dynamic wrapper using the Lua script called osgLua. Since we can control the 3D scene with a scripting language, it allows very fast prototyping. OsgLua functions are simple and can be sent from other computers on our local network, for example, we can easily change the color of the scene based on the rhythm of the music processed on another computer. This is represented in Figure 11

9. CONCLUSIONS

The Visual Paths project was an opportunity to setup a full real time image processing pipeline for solving an actual artist request, that is, providing the 3D position of a pen. This imaging pipeline includes image capture devices (Wiimotes, cameras and capture boards), GPU-based processing techniques, computer vision calibration, and visualization techniques, as well as building the LED based IR pen.

This project demonstrates the feasibility of such a system by proposing two different possible routes. Moreover, the acquired knowledge in computer vision and GPU processing will definitely be re-used in forthcoming projects.

Finally, the software built in this project will be made available as a free download package for non commercial use on the numediart website.

10. ACKNOWLEDGMENTS

Prof. Sébastien Roy and Louis Bouchard from Université de Montréal, Canada, also contributed to the Wiimote-based system.

Thanks to Sébastien Noël, for his help and his knowledge of Blender.

numediart is a long-term research program centered on Digital Media Arts, funded by Région Wallonne, Belgium (grant N° 716631).

11. REFERENCES

11.1. Scientific references

[3] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision, Second Edition*. Cambridge University Press, ISBN: 0521540518, 2004. P. 281. P.: 45.

[5] Hubert Nguyen. *GPUGems 3*. Addison-Wesley Professional, 2007. P.: 45.

[7] Randi J. Rost. *OpenGL Shading Language, Second Edition*. Addison-Wesley Professional, 2006. Pp.: 45, 48.

[8] Zhengyou Zhang. “A Flexible New Technique for Camera Calibration”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.11 (2000). Pp. 1330–1334. URL: citeseer.ist.psu.edu/zhang98flexible.html. P.: 45.

11.2. Software and technologies

[1] “Blender, the free open source 3D content creation suite”. URL: http://www.blender.org/. P.: 49.

[2] “Core Image Programming Guide”. GPU-based Image Processing Language. URL: http://developer.apple.com/documentation/GraphicsImaging/Conceptual/CoreImaging/ci_intro/chapter_1_section_1.html. P.: 47.

[4] “Image Unit Tutorial”. URL: http://developer.apple.com/documentation/GraphicsImaging/Conceptual/ImageUnitTutorial/Introduction/chapter_2_section_1.html. P.: 47.

[6] “Open Scene Graph”. 3D Rendering Library. URL: http://www.openscenegraph.org. Pp.: 44, 50.