Pierre-Luc Bertrand
Ramzy Hissin
Justin Mereb
Eric Zaino

Comp 471 / Cart 498: Final Project Report

Work Presented to Dr. Sha Xin Wei

Concordia University
December 11, 2006

Team Members

Pierre-Luc Bertrand	5330211
Ramzy Hissin	4925548
Justin Mereb	4885368
Eric Zaino	4936957

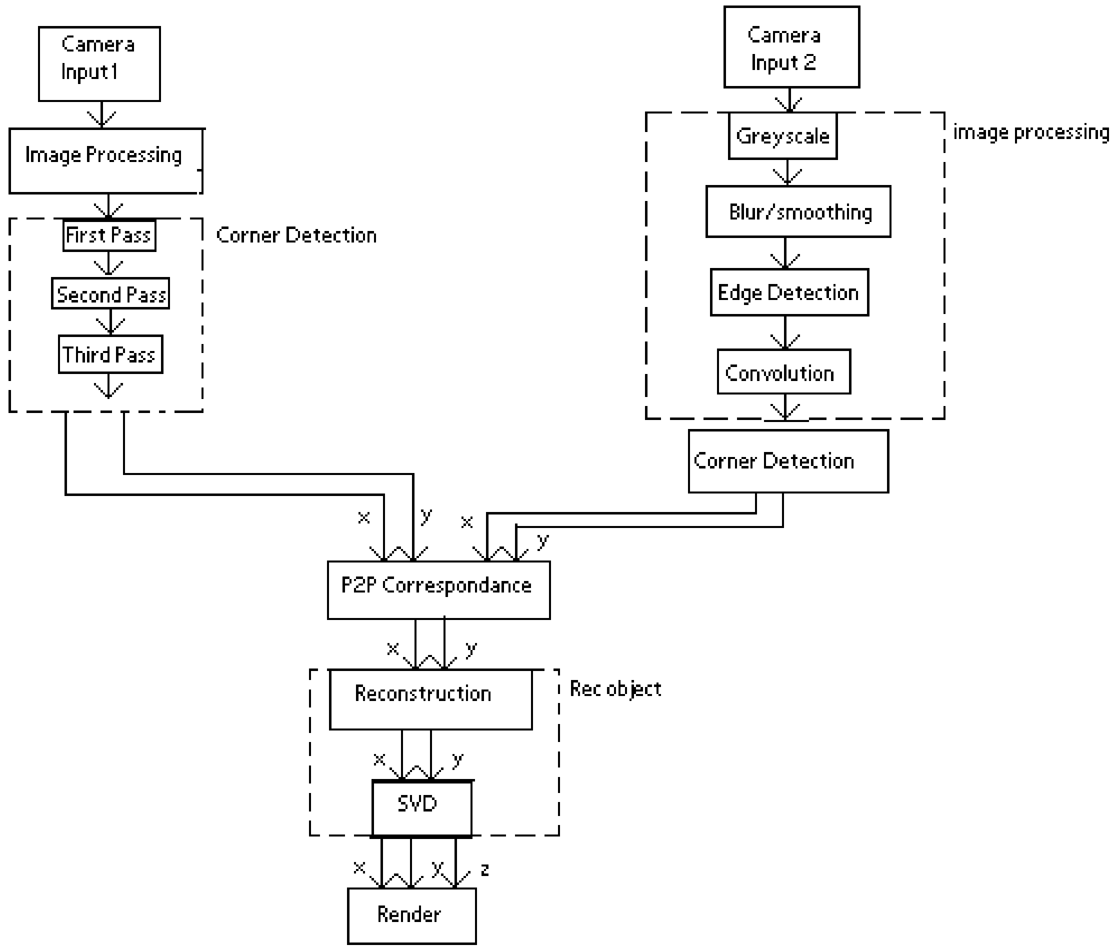
Roles

Concept Lead: *Pierre-Luc*
Programming: *Entire Group*
Maths: *Pierre-Luc*
Set Construction: *Entire Group*

Concept

Taste of Reality is a project that was born out of the interest for the field of computer vision, teaching machines to see and recognize. This is a very important topic today. We wanted to be apart of this movement and implement our own small version of these ideas. Our very basic approach allows a computer to see and recognize basic geometric shapes and redraw them in a three-dimensional model on the computer using OpenGL The simple shape we chose to implement was a cube. This project is not a game or any form of entertainment, but a direct scientific and practical application.

Below is a flowchart of the design concept of our system. Please follow this figure. Note that only a general overview is given here, a more detailed explanation is given in later sections. As can be seen, two camera inputs are taken. An image from each camera is then sent to an image-processing object, which contains several steps shown on the right. The output of this step is a processed image that is ready to be used. The image is sent to a corner detection algorithm where the corners are extracted. The x,y coordinates of each corner are then sent to a point to point correspondence where the corners are matched. The z coordinate (depth of the cube) is then calculated via the Rec object and finally all 3 coordinates are sent to the render object that draws the cube.



Camera Parameters [2]

A camera can be represented as a perspective projection matrix (henceforth simply camera matrix) **P** where:

$$\mathbf{P} = \mathbf{A}[\,\mathbf{R} \mid \mathbf{t}\,]$$

A is the camera intrinsic parameters.

R and **t** are the external parameters.

Camera Intrinsic Parameters [2]

The camera intrinsic parameters are the parameters of the camera only. These parameters can be acquired by using the calibration toolbox [1] running on MatLab.

The following matrix is the intrinsic parameters.

$$\mathbf{A} = \begin{pmatrix} \alpha_u & \gamma & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{pmatrix}$$

Where $\alpha_u = -fk_u, \alpha_v = -fk_v$ are the focal lengths in horizontal and vertical pixels, respectively (*f* is the focal length in millimeters, k_u and k_v are the effective number of pixels per millimeter along the u and v axes).

(u_0, v_0) are the coordinates of the *principal point*, given by the intersection of the optical axis with the retinal plane.

γ is the skew factor. I can be set to 0 for most cameras.

Camera External Parameters

The camera external parameters are the parameters describing the translation and rotation of the second camera with respect to the first camera.

The rotation matrix **R** is a 3x3 matrix and **t** is a 3x1 matrix.

Getting the Camera Parameters [2]

Once we have the parameters, we can have the camera parameters that include both internal and external parameters using the following formula:

$$\mathbf{P} = \begin{pmatrix} \alpha_u & \gamma & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{pmatrix} \left[\begin{pmatrix} \alpha_u & \gamma & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \right]$$

3D Reconstruction [3]

With the cameras calibrated, we can now move on to the 3D reconstruction from matching pair of points.

Using this formula: $\mathbf{x = P\,X} \Leftrightarrow \mathbf{x} \times \mathbf{(P\,X) = 0}$

One can extract parts of it to

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \text{ and } \mathbf{AX} = \begin{bmatrix} \mathbf{A_1} \\ \mathbf{A_2} \\ \mathbf{A_3} \end{bmatrix} \mathbf{X} = \begin{bmatrix} \mathbf{A_1X} \\ \mathbf{A_2X} \\ \mathbf{A_3X} \end{bmatrix}$$

Now we can make them single equations.

1. $x(\mathbf{A_3X}) - (\mathbf{A_1X}) = 0$
2. $y(\mathbf{A_3X}) - (\mathbf{A_2X}) = 0$
3. $x(\mathbf{A_2X}) - y(\mathbf{A_2X}) = 0$

One can see that the third equation is a linear combination of the first two.

Now we want to relate the pair of points coming from the two different pictures.

$$\mathbf{P} = \begin{bmatrix} x\mathbf{A_3} - \mathbf{A_1} \\ y\mathbf{A_3} - \mathbf{A_2} \\ x'\mathbf{A_3}' - \mathbf{A_1}' \\ y'\mathbf{A_3}' - \mathbf{A_2}' \end{bmatrix} \mathbf{X} = 0$$

Singular Value Decomposition

With this **P**, we can now get **X**. To do so, we use the Singular Value Decomposition (SVD). The SVD of **P**, $\mathbf{P} = \mathbf{U}\ \mathbf{D}\ \mathbf{V}^{\mathbf{T}}$

X is the last column of **V**. **D** must be in a descending order.
As for the Singular Value Decomposition, I am not completely sure of what it does but I will explain how I think it works.

By giving the camera parameters and a point, we are forming a line in the space since our camera is a pin-hole camera. A pin-hole camera has a center point and by using the point on the retinal plane, we can define a line in the space. Doing that for two cameras and two points, we are defining two lines. In theory, these two lines will intersect in the space but in practice, they will probably not. From my understanding of Eigen values and eigenvectors, the Eigen value will give us the critical point in this system. By taking the smallest critical point, we are in fact taking the smallest distance between the two lines which is in our case, close to the intersection. By taking the eigenvector, we get the coordinates of this close to an intersection point which gives us our coordinate of interest.

Image Processing

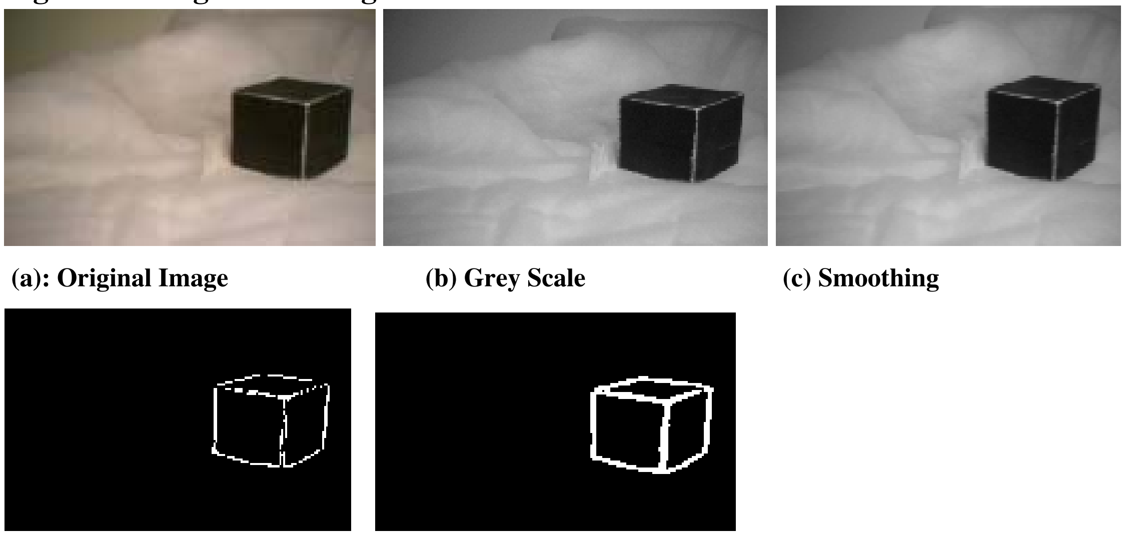
The first step in the algorithm is to process the image. The inputs from the eyesights are fed into the first patcher named "imageprocessing". This is a file we wrote to take in a camera feed and output an edge detected image that we can send to the corner detection external. The output of each step is shown in Figure 1below. Figure 1 (a) is the original image.
The first step in the patcher is to convert the image into greyscale; this is done to lower the info for each pixel by setting all the RGB values equal to each other. Also greyscaling an image is very convenient for programming in image processing as it sharpens an image. This is done using " jit.rgb2luma". This is shown in figure 1 (b).

The next step is to blur the image. This serves to smooth out the jagged edges, and hence remove unwanted noise elements. Blurring is done in jitter using "jit.fastblur". Blurring is usually done by applying the image with a Gaussian distribution, which is essentially passing the image through a low pass filter. Shown in Figure 1 (c).

Then edge detection is performed followed by an inversion of black and white to make the image brighter. All the pixels are scanned and whenever there is a difference between one pixel and its neighbour, that pixel will be filled with white colour. If there is no difference between two pixels then they will both be black, or filled with no colour. This is done with he "jit.robcross" and "jit.op" objects. Shown in figure 1(d).

The final step is to do convolution to sharpen the image. A small window, called the convolution kernel, is scanned across the image. The center of the kernel is applied to each pixel in the image and then each value in the kernel is multiplied with its corresponding value in the image area that is covering. These values are then summed up and stored in the center pixel. Convolution continues until the kernel has been applied to all values in the image. Our convolution kernel is set to 2.71. Shown in figure 1(e).

Figure 1: Image Processing



(a): Original Image **(b) Grey Scale** **(c) Smoothing**

(d) Edge detection **(e) Convolution**

Corner Detection

This algorithm takes in an image-processed picture and extracts the corners. The code was written in Java and designed by the team. An mxj external was written to import this code into jitter. It was hoped to use an already designed algorithm from a published paper, however due to time constraints this approach was abandoned and a unique one was implemented. This also made the design more interesting for the team.
The algorithm as it stands works nicely but is fragile. The correct corners are extracted every pass but from time to time there are false negatives or false positives. This means that the algorithm will find any corners in the screen, not only those of the cube, and that it might mistake a point on one of the edges of the cube as a corner.

How it works

The algorithm functions in a nearest neighbour approximation: The image is fed into the mxj external as a down sampled matrix of 1’s and 0’s shown in figure 2 below.

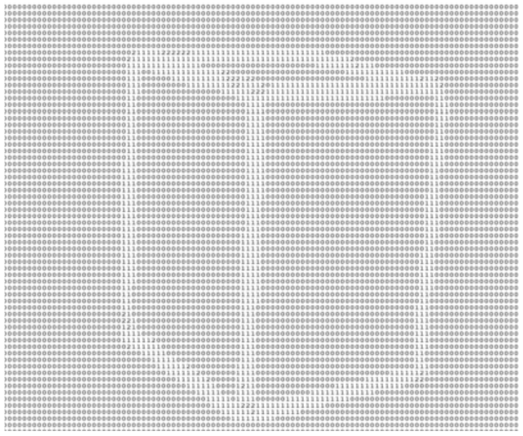


Figure 2: Corner Detection input

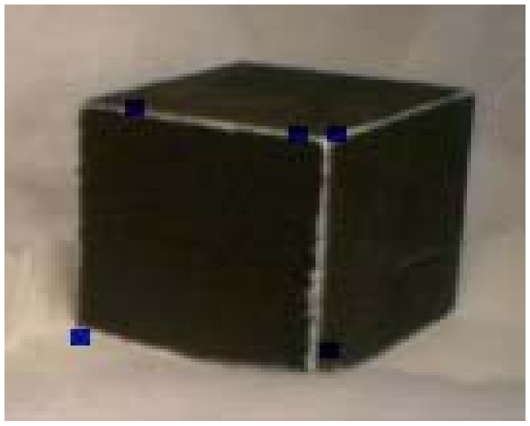
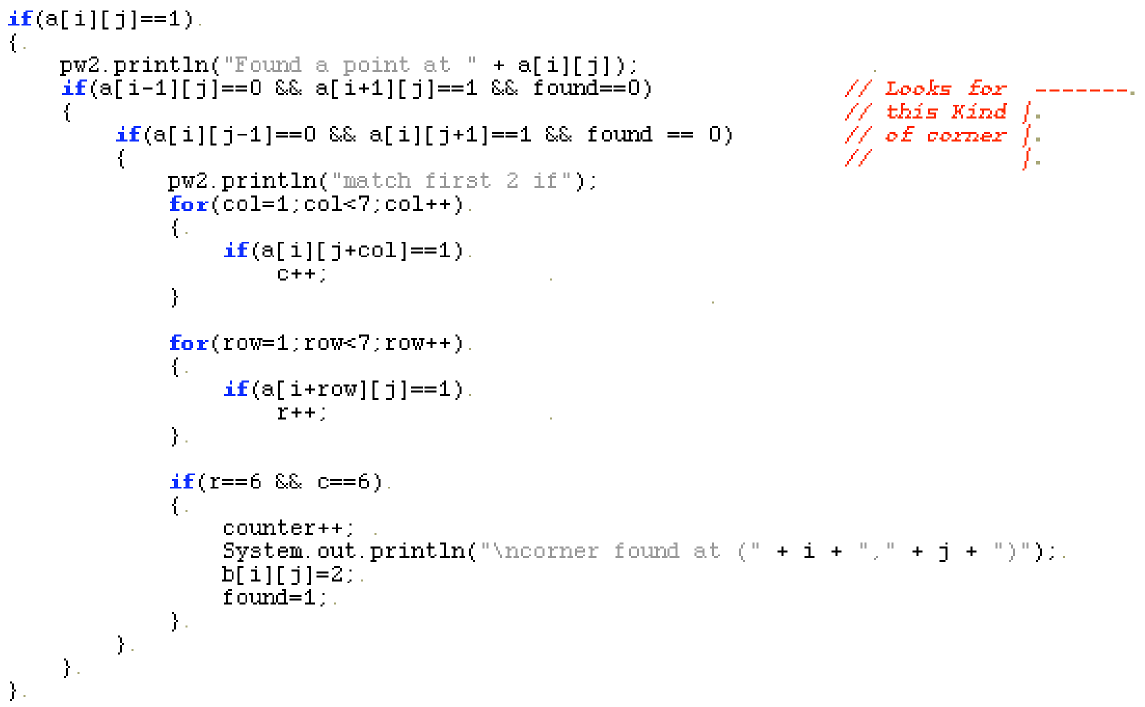


Figure 3: Detected Corners

The algorithm then analyses every pixel and stops when it finds a 1. It then proceeds to search all its neighbours for known corner configurations. It is as if the image is being run through with several different masks, matching each type of corner shown here:

```
----   ----    |          |      |            |        ----     ----            |
|         |    |          |      |            |       /          \              |
|         |    |          |       \         /       /             \        \    |
|         |    ----    ----         \      /       /               \        \   |
```

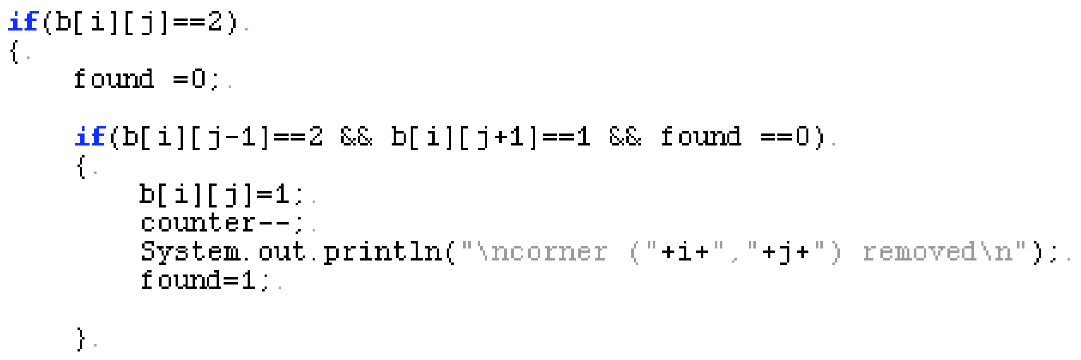
An example of one of the search algorithms in the first pass is shown in figure (4) below. Please note that an array element value of 2 defines a corner, and a 1 defines an edge. As can be seen in the comments, it identifies what kind of corner is being searched for.



```
if(a[i][j]==1)
{
    pw2.println("Found a point at " + a[i][j]);
    if(a[i-1][j]==0 && a[i+1][j]==1 && found==0)           // Looks for  -------
    {                                                       // this Kind |
        if(a[i][j-1]==0 && a[i][j+1]==1 && found == 0)     // of corner |
        {                                                   //           |
            pw2.println("match first 2 if");
            for(col=1;col<7;col++)
            {
                if(a[i][j+col]==1)
                    c++;
            }

            for(row=1;row<7;row++)
            {
                if(a[i+row][j]==1)
                    r++;
            }

            if(r==6 && c==6)
            {
                counter++;
                System.out.println("\ncorner found at (" + i + "," + j + ")");
                b[i][j]=2;
                found=1;
            }
        }
    }
}
```

Figure 4: Corner detection code

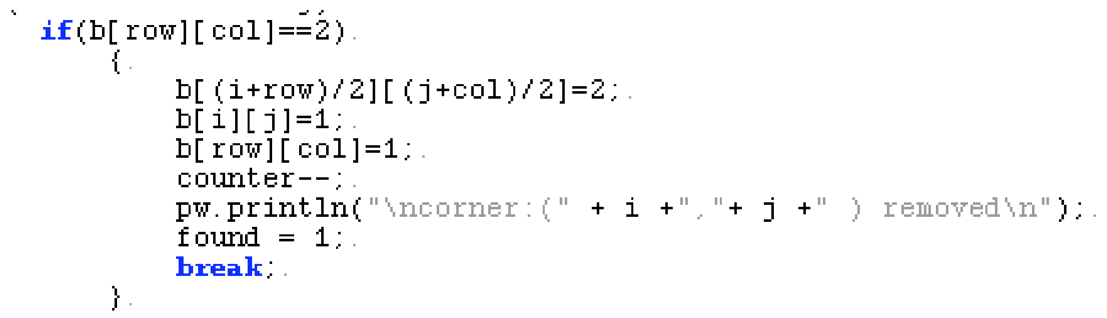
After the first pass, suspected corner points are marked and the algorithm begins its second pass.
The point of the second pass is remove non-corner points. If there are two pixels beside each other that have both been marked as corners, then one will be removed, usually the one that is not an absolute edge. Other non-corner points are removed using similar criteria. An example of one of the checking code is shown below.



```
if(b[i][j]==2)
{
    found =0;

    if(b[i][j-1]==2 && b[i][j+1]==1 && found ==0)
    {
        b[i][j]=1;
        counter--;
        System.out.println("\ncorner ("+i+","+j+") removed\n");
        found=1;

    }
```

Figure 5: second pass

Then the image is run through a third and final pass. The point of this pass is to combine suspected corner points that are a certain distance from each other: if a corner point exists in a radius, maybe 10 pixels, from another suspected corner, then the midpoint between these two is marked as a corner and the original two points are unmarked. This is to ensure that not more than one point is marked as a corner in a given area. The accompanying ASCII text is shown with the results of each pass.



```
 if(b[row][col]==2)
     {
         b[(i+row)/2][(j+col)/2]=2;
         b[i][j]=1;
         b[row][col]=1;
         counter--;
         pw.println("\ncorner:(" + i +","+ j +" ) removed\n");
         found = 1;
         break;
     }
```

Figure 6: Third pass to group together close corners

After the third pass the corner points are exported into a paint object defined below, and the output of the corners is shown on the original image, shown in figure 3 above. Please note that there is a phase issue, the corners do not all flash at the same time. All the corners are found, but flash at different times.

The Paint Object

The paint object is a patcher we designed to take the coordinates found by the corner detection and paint flashing squares on the original image. This was done to visually enhance the corner detection algorithm and make it nicer to look at. The list of x and y coordinates of the corners are inputted into the patch. A copy of each list goes through a vexpr object so that 10 is added to each element. This addition defines the length and width of the square we will draw. As shown in the figure below, the outputs of each unpack object are colour coded to make the code easier to follow.

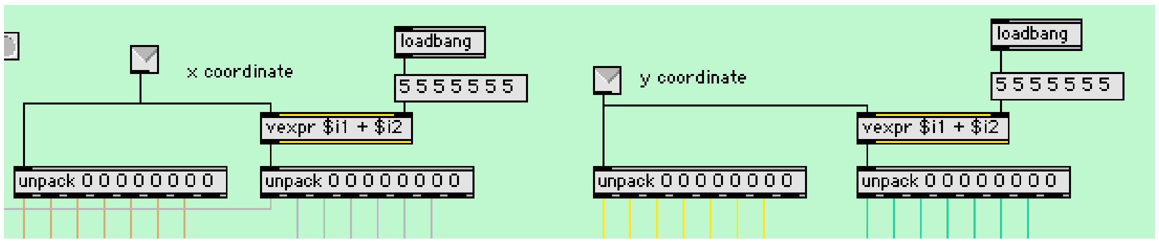


Figure 7: Paint input

The x and y coordinates, along with the length and width of the square are then packed into a list and sent to a paintrect object shown in figure 8 below which draws the initial square. The first four numbers with $ signs are so that we can dynamically enter in the coordinates, as well as the size and width of the square, meaning during run time. The last three numbers are the RGB colours that we chose to make the cube. In our case, the colour of the square are Red=0, Green =0, Blue =255. The squares are painted blue as shown in figure 3.

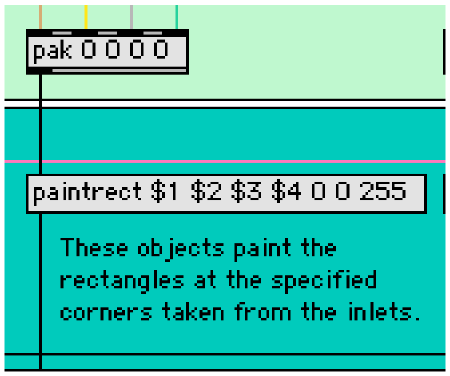

pak 0 0 0 0
paintrect $1 $2 $3 $4 0 0 255
These objects paint the rectangles at the specified corners taken from the inlets.

Figure 8: Paintrect object

This only draws the initial cube. We would like the cube to flash, so that the corners scream out, “HERE I AM” and are immediately discernible on the original image. The cubes are made to flash with the following objects:

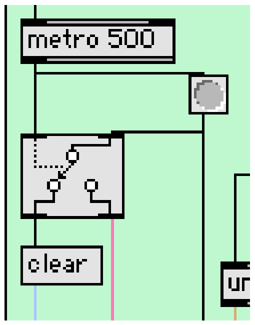


Figure 9: Switching code to make squares flash

The metro sends out a bang every half a second that causes the switch to alternate between the clear object and the paintrect object not seen in this image, The pink wire coming out of the right side of the switch enters the paintrect object shown in figure 8 above. So the square is first painted blue, then cleared from the screen, and then painted blue again. This process repeats itself for the duration of the runtime of the program, and the squares will flash where the corners are found.

The following code serves to combine the flashing squares in the same output window. It is here that the phasing problem occurs, in which all the found corners do not flash at the same time. The jit.op object only allows the flashing squares through.

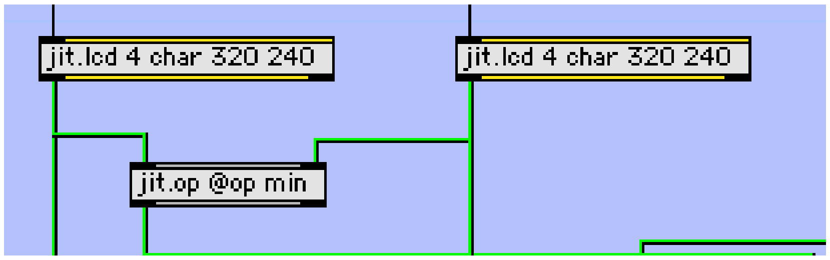


Figure 10: Combine flashing squares

Finally each flashing corner is shown in a separate window at the bottom of the patch, as well as a final window with all of them combined. As can be seen below, the final combined window is still flashing out of phase. Only the bottom-most pwindow is outputted.

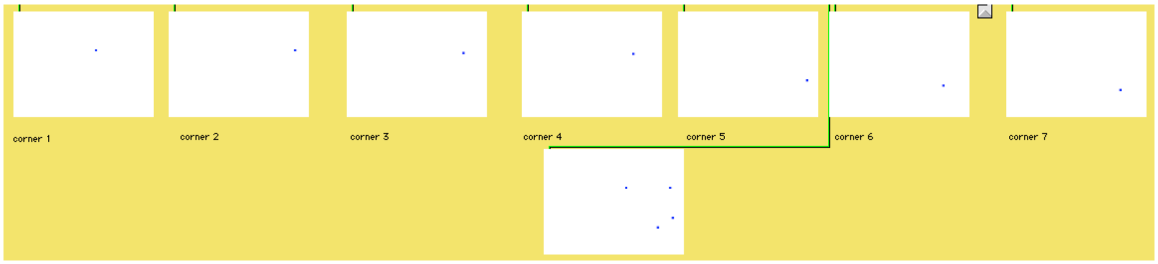


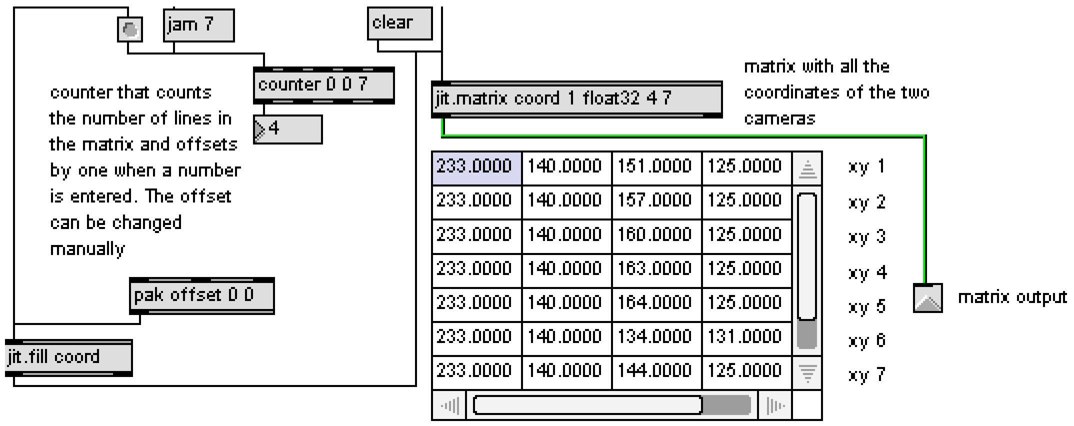
Figure 11: Paint output

This output is then superimposed on the original image to produce the corner-detected image shown in figure 3 above.

Corner Correspondences

The CornerCorrespondences patch was created to make our lives much easier. Since our corner detection algorithm was not finding all the corners reliably, we needed to create this patch so that we could choose the right corners of the left and right cameras and enter them into a matrix which is then sent to be reconstructed. Due to time constraints this was a necessary patcher to design.

Its design is quite simple; in essence, it takes the initial camera inputs, and then applies the paint objects onto each corner to actually see which corners where detected. After that, user input is required to find all seven visible corners of the cube on both cameras. This is done by clicking on a corner on the left camera view and then clicking on the corresponding corner on the other camera view. This information is then entered into a 4x7 jit.matrix object using a counter object connected to jit.fill to enter the x,y,x’,y’ coordinates one by one with an offset of y =1 each time, so that the coordinates already entered in the matrix aren’t being overwritten. The output of the CornerCorrespondences patch is the matrix, which is then sent to the Rec patcher. The figure below is what this part of the patch looks like.



counter that counts the number of lines in the matrix and offsets by one when a number is entered. The offset can be changed manually

matrix with all the coordinates of the two cameras

233.0000	140.0000	151.0000	125.0000	xy 1
233.0000	140.0000	157.0000	125.0000	xy 2
233.0000	140.0000	160.0000	125.0000	xy 3
233.0000	140.0000	163.0000	125.0000	xy 4
233.0000	140.0000	164.0000	125.0000	xy 5
233.0000	140.0000	134.0000	131.0000	xy 6
233.0000	140.0000	144.0000	125.0000	xy 7

Figure 12: Corner Correspondence Matrix Output

Basically, the counter goes up to seven, and then when it reaches that number, it goes back down to zero. This counter is directly connected to the pak offset 0 0 object connected to jit.fill, which offsets the matrix by a set amount.

SortCoord

This object is essentially what sorts the matrix of coordinates coming out of cornercorrespondences the same way the corners are organized on the cube shown below.

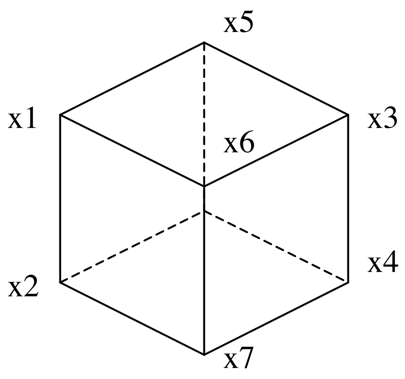


Figure 13: Order of the Corners

The reason this is done is that the cube render patch does not do a sorting and the order is hard coded. If there were no sorting, then the cube wouldn't really look like a cube.

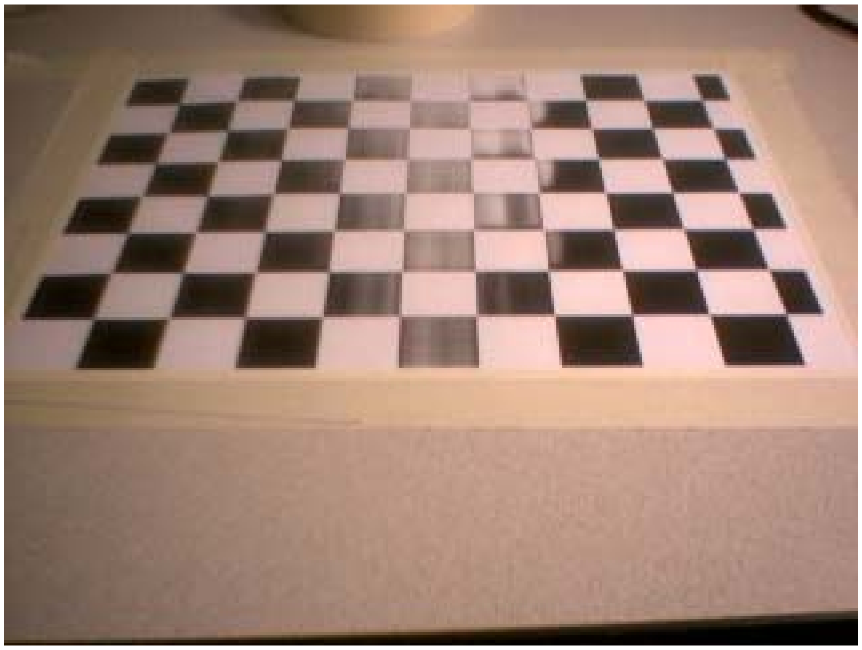
The algorithm that was used for the sorting was a simple one; in other words, not very efficient, but easy to write. Its efficiency is around O(n^2). Basically, it scans through the matrix trying to find coordinates bigger or smaller than the initial one, corresponding to the corner that is being found. In other words, if the initial corner is x6 and we are trying to find x5, then will look for a smaller x and a smaller z. The output of this sorting algorithm is a list containing the x,y,x',y' of each corner starting at x1 and ending at x7. Here's a part of the code written in JAVA.

```
public Coord[] sort(double[][] a)
{
        Coord[] all = new Coord[7];

        for(int i=0;i<a.length;i++)
        {
                all[i] = new Coord(a[i][1], a[i][0]);
        }

        SortRows(all);

        Coord[] left = { all[0], all[1] };
        Coord[] center = { all[2], all[3], all[4] };
        Coord[] right = { all[5], all[6] };

        SortCol(left);
        SortCol(center);
        SortCol(right);

        all[0] = left[0];
        all[1] = left[1];
        all[2] = right[0];
        all[3] = right[1];
        all[4] = center[0];
        all[5] = center[1];
        all[6] = center[2];

        return all;
}

private void SortCol(Coord[] c)
{
        double smallest = 0.0;

        for(int i=0;i<c.length;i++)
        {
                smallest = c[i].getCol();

                for(int j=i;j<c.length;j++)
                {
                        if(c[j].getCol() < smallest)
                        {
                                smallest = c[j].getCol();
                                swap(c, i, j);
                        }
                }
        }
}

private void SortRows(Coord[] c)
{
        double smallest = 0.0;

        for(int i=0;i<c.length;i++)
        {
                smallest = c[i].getRow();

                for(int j=i;j<c.length;j++)
                {
                        if(c[j].getRow() < smallest)
                        {
                                smallest = c[j].getRow();
```

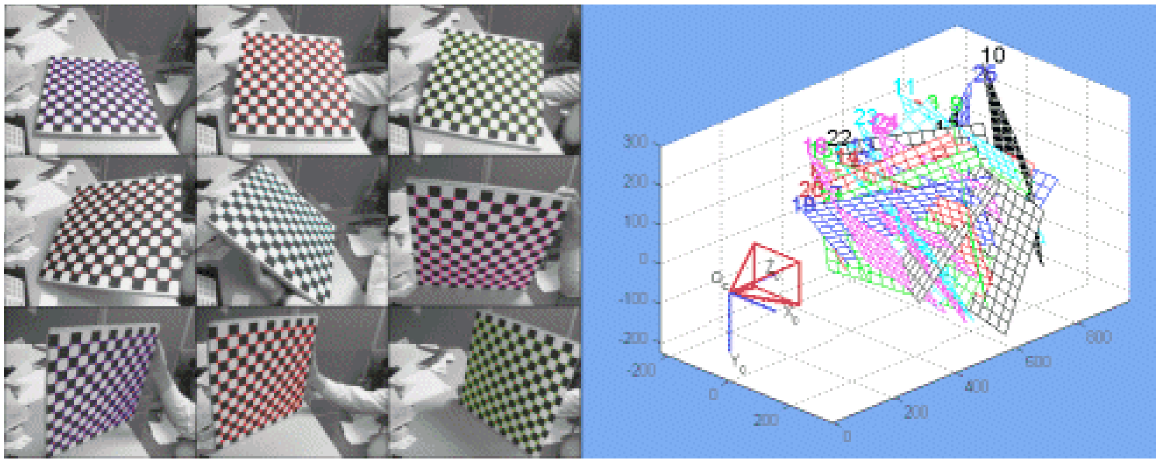
```
                                        swap(c, i, j);
                              }
                    }
          }
}

private void swap(Coord[] c, int index1, int index2)
{
          Coord dummy = c[index1];
          c[index1] = c[index2];
          c[index2] = dummy;
}
```

Matlab Calibration

Even though everyone was using iSights for filming, each camera has little differences that can make quite a big difference when measuring focal lengths and such. This is why calibrating the camera is very important. To calibrate the camera, we used a built in calibration tool found in Matlab. To use this tool it is necessary to first print a checkerboard.



This checkerboard is used to find all the corners of each square. To first step to perform the calibration is to take around twenty different pictures of the checkerboard using one camera and then do the same thing with the other camera, Once this is completed, the pictures are then input into Matlab, which will ask that the user inputs the origin of the first square on the checkerboard and then make a big square using each corners of the checkerboard. This needs to be done on each picture taken for both cameras.



Once this is done, the software will give a whole page of results, including the focal points and principal points, which are needed in the project. These results are specific for each camera and the differences can be quite large; so it is important to first perform this calibration before starting the project so that the cube that will be measured can be accurately represented. The text below shows the results of a calibration test we did.

```
Focal Length:          fc = [ 1024.79922   1026.49224 ] ± [ 9.33714   7.86167 ]
Principal point:       cc = [ 407.55057   367.49343 ] ± [ 8.53734   12.97527 ]
Skew:             alpha_c = [ 0.00000 ] ± [ 0.00000  ]   => angle of pixel axes = 90.00000
± 0.00000 degrees
Distortion:            kc = [ -0.03070   -0.53234   -0.00402   0.00180  0.00000 ] ± [
0.02940   0.20436   0.00211   0.00211  0.00000 ]
Pixel error:          err = [ 0.49771   0.61276 ]


Focal Length:          fc = [ 1019.96701   1027.31694 ] ± [ 25.19010   16.74503 ]
Principal point:       cc = [ 378.16745   341.12452 ] ± [ 18.79007 45.51434 ]
Skew:             alpha_c = [ 0.00000 ] ± [ 0.00000  ]   => angle of pixel axes = 90.00000
± 0.00000 degrees
Distortion:            kc = [ -0.01660   -0.24866   0.00996   -0.01020   0.00000 ] ± [
0.03942   0.28394   0.00337   0.00535  0.00000 ]
Pixel error:          err = [ 0.32283   0.51939 ]

KK = [ 1019.96701 0 378.16745 ; 0 1027.31694 341.12452 ; 0 0 1 ]

% KK =
%   1.0e+003 *
%    1.0200         0    0.3782
%         0    1.0273    0.3411
%         0         0    0.0010


P1 = KK * [ 1 0 0 0 ; 0 1 0 0 ; 0 0 1 0 ]

% P1 =
%   1.0e+003 *
%    1.0200         0    0.3782         0
%         0    1.0273    0.3411         0
%         0         0    0.0010         0

P2 = KK * [ 1 0 0 247 ; 0 1 0 0 ; 0 0 1 0 ]

% P2 =
%   1.0e+005 *
%    0.0102         0    0.0038    2.5193
%         0    0.0103    0.0034         0
%         0         0    0.0000
```

The actual result is quite a bit longer than what is shown above, but what is needed is actually the focal lengths and principal points.

Internal and External Parameters

For our project, we use MatLab to get the internal parameters and then we wrote them in the internal parameters patcher.

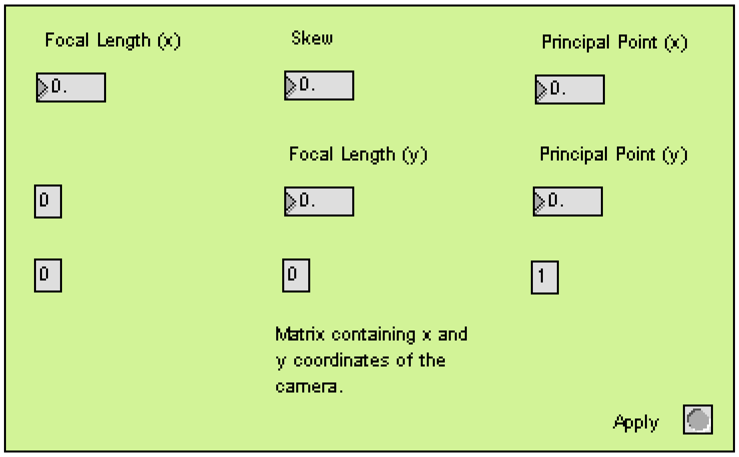


Figure 14: Internal Parameters Patcher

As for the external parameters, we kept it simple by not having rotations of cameras because at first we though we would be using only the corner detection. Using the corner detection only, it makes thing very complicated to have matching corners because we would have to use the epipolar geometry. Using only a translation on the x axis would keep the corners on the same height therefore would keep the algorithm easy.

We have done a patcher to put in the external parameters. In our project, we had only the translation in x but we let the users the freedom to do translation in other axis.

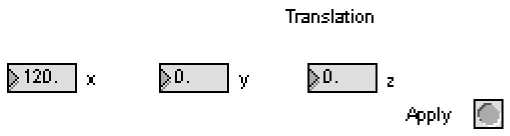


Figure 15: External Parameters Patcher

Rec

This is a necessary patcher which essentially unpacks the list coming out of the sorting algorithm and sends the (x, y) coordinates of each camera to seven reconstruction patchers, which will send back a list of (x, y, z) coordinates. These coordinates are then grouped together and then are sent back to the main project patcher.

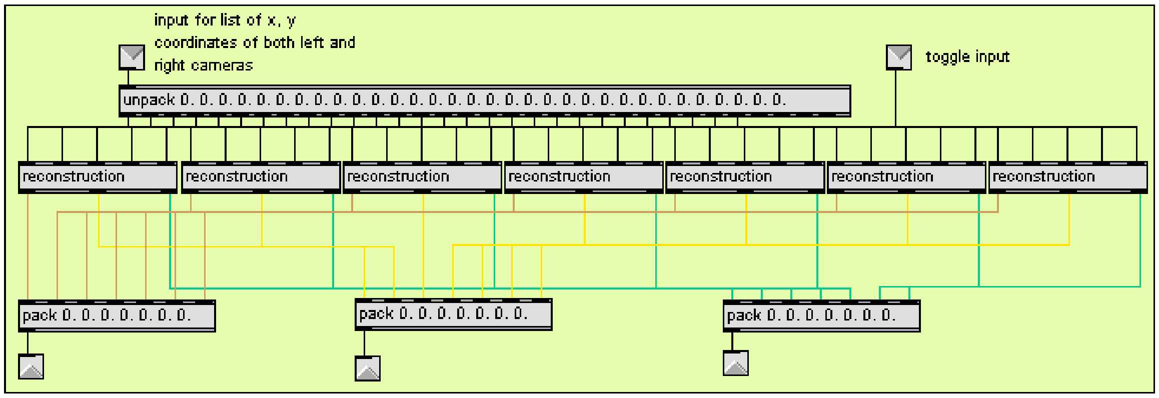

input for list of x, y
coordinates of both left and
right cameras
toggle input
unpack 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
reconstruction
reconstruction
reconstruction
reconstruction
reconstruction
reconstruction
reconstruction
pack 0. 0. 0. 0. 0. 0. 0. 0.
pack 0. 0. 0. 0. 0. 0. 0. 0.
pack 0. 0. 0. 0. 0. 0. 0. 0.

Figure 16: Rec Patcher

This patch is pretty straightforward as it simply redirects the coordinates to their respective location so that the real world coordinates can be calculated and then sent to be rendered.

Reconstruction, CameraTest and LineOperation

The reconstruction patch consists of a few steps. First, the matrix containing the values of the camera calibration test, which were obtained using matlab, is obtained using the cameratest22 patcher which essentially multiplies the external camera values with the internal camera values for both cameras. This multiplication is obtained using the jit.la.mult which conducts a cross product on the two incoming matrices. The cross product is essentially the multiplication of each row of the first matrix with each column of the second matrix. The final matrices, called P1 and P2, are then split into three distinct matrices each which are called P11, P12, P13, P21, P22 and P23 respectively.

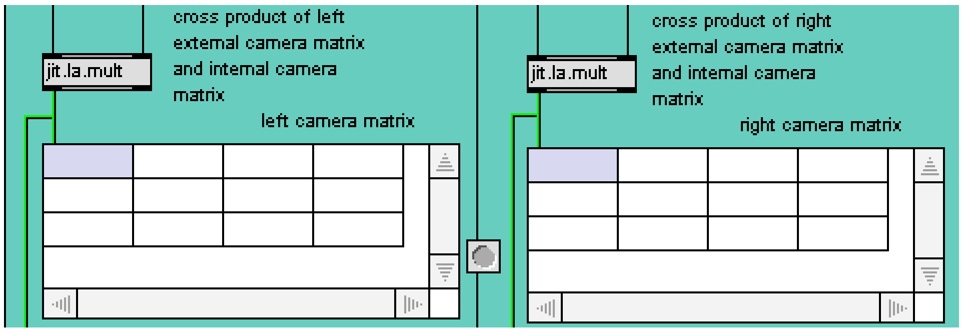

jit.la.mult
cross product of left external camera matrix and internal camera matrix
left camera matrix
jit.la.mult
cross product of right external camera matrix and internal camera matrix
right camera matrix

Figure 17: Cross product

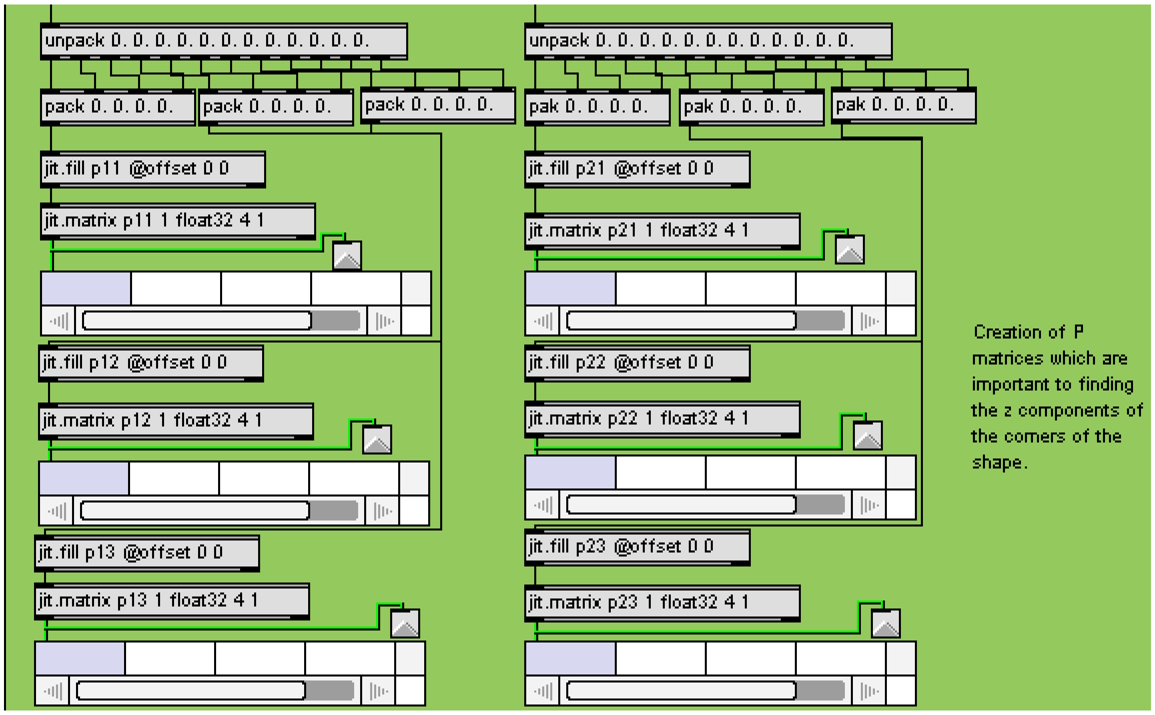

unpack 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
pack 0. 0. 0. 0.
pack 0. 0. 0. 0.
pack 0. 0. 0. 0.
jit.fill p11 @offset 0 0
jit.matrix p11 1 float32 4 1
jit.fill p12 @offset 0 0
jit.matrix p12 1 float32 4 1
jit.fill p13 @offset 0 0
jit.matrix p13 1 float32 4 1
unpack 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
pak 0. 0. 0. 0.
pak 0. 0. 0. 0.
pak 0. 0. 0. 0.
jit.fill p21 @offset 0 0
jit.matrix p21 1 float32 4 1
jit.fill p22 @offset 0 0
jit.matrix p22 1 float32 4 1
jit.fill p23 @offset 0 0
jit.matrix p23 1 float32 4 1
Creation of P matrices which are important to finding the z components of the corners of the shape.

Figure 18: Outputs of cameratest22

These matrices are then sent to the LineOperation patcher which does a scalar multiplication between the x or y of each camera and P13 or P23, depending on whether it's from the left camera or the right camera, and then subtracted by P11, P12, P21 or P22, as shown on the figure below.

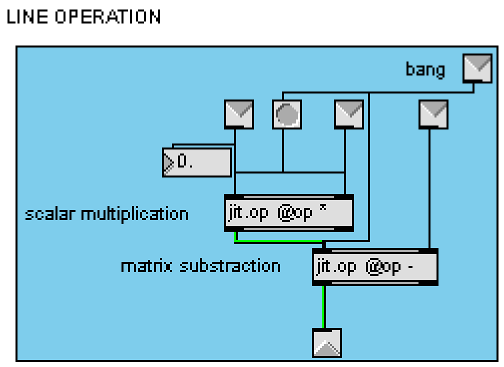
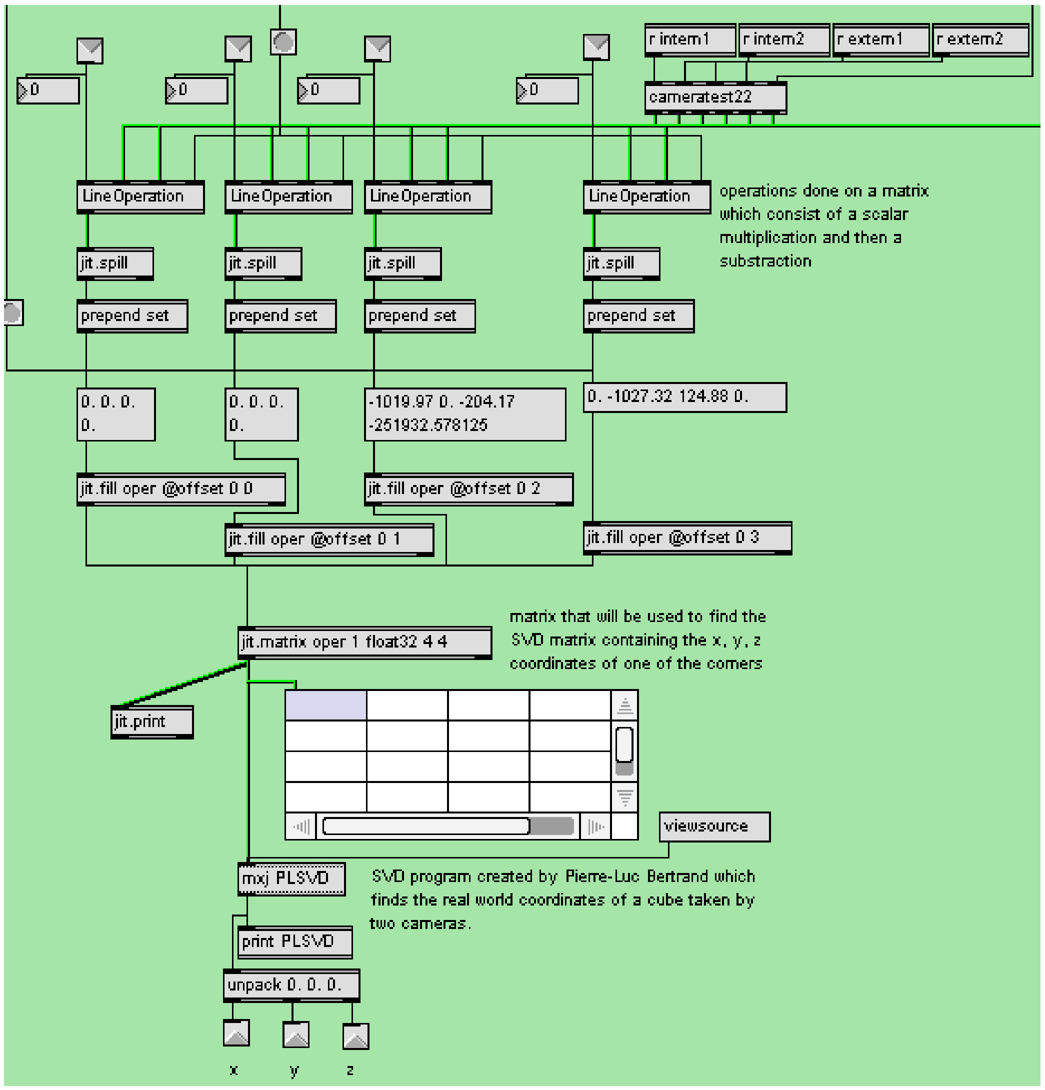


Figure 19: Line Operation

When this is done, all of these 4x1 matrices are put into one 4x4 matrix which is passed on to the PLSVD algorithm written in JAVA.



operations done on a matrix which consist of a scalar multiplication and then a substraction

matrix that will be used to find the SVD matrix containing the x, y, z coordinates of one of the corners

SVD program created by Pierre-Luc Bertrand which finds the real world coordinates of a cube taken by two cameras.

Figure 20: Reconstruction Patcher

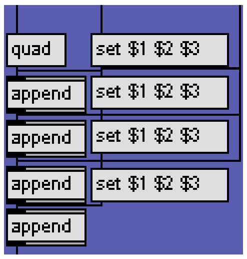
Getting the Z coordinate

PLSVD is using the code from http://www.idiom.com/~zilla/Computer/Javanumeric/ for the Singular Value Decomposition. The problem from this source was that sometimes the diagonal matrix was not in a descending order therefore we had to sort them. The PLSVD java code is sorting these values. The diagonal matrix is the Eigen values and the **V** matrix is their corresponding eigenvector so when the Eigen values had to be swapped, their respective eigenvector had to be swapped as well.

The last column of the **V** matrix is the interesting column. We are using this column to get our real world coordinates in millimeters. The first three rows are the x, y and z component multiplied by some factor. To get the real x, y and z component, we have to divide them by the last row of the column. PLSVD is doing that as well then it outputs it as a list.

Cube Render

The cuberender patch was created in a few steps. First the coordinates are sent through three inlets, which correspond to the x, y, and z coordinates. These coordinates are sent through an unpack object which separates them and then rearranges them in packs of three, equivalent to a real (x, y, z) coordinate. Then, these coordinates are sent to six message boxes corresponding to the six planes of a cube. Since we couldn't just use a message box with $1 - $12, for each quad, the append object was used to add each corner coordinate. Basically, a message box containing the sentence set $1 $2 $3 was connected to an append object. This is done a total of three times. All of these append objects are connected to a message box containing the message quad. The picture shown below shows this part of the patch.



```
quad      set $1 $2 $3
append    set $1 $2 $3
append    set $1 $2 $3
append    set $1 $2 $3
append
```

Figure 21: Message Box containing the plane of the cube

When taking a live input feed of a cube using two cameras that are only translated and not rotated, there is always going to be a point which will not be visible. This is a problem we faced which meant that we needed to come up with an algorithm to calculate this eight coordinate. The first thing we thought of was to take the x, y, and z coordinates of the corners around it. While this may have worked, it required unnecessary work since we had found a simple and cleverer algorithm. This algorithm consists of calculating the vector of the front of the cube and then subtracting the rear top corner with the measured vector, which would give us the missing coordinate. The picture shown below explains in more detail how this all works out; x6 – x7 is the vector we are measuring and x8 is the point we are looking for. So, x5 – (x6-x7) will give x8.

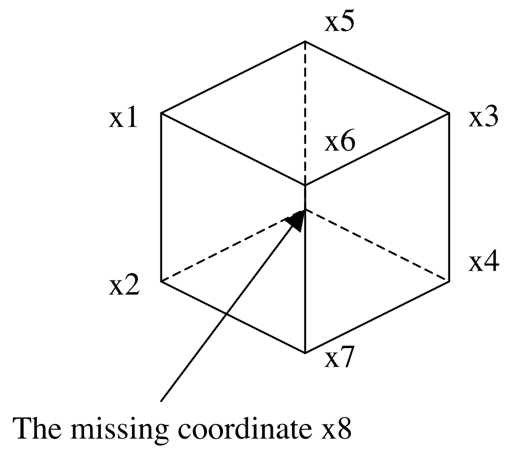


Figure 22: Missing Coordinate

Once all the corners are entered into the plane, it is now ready to be sketched using the jit.gl.sketch object, which receives all the plane information as well as the color for each of these planes. The jit.gl.handle object is also used to take care of the handling of the cube once it's rendered. The final step of the cuberender patch is the rendering using the jit.gl.render object, which takes the information given by jit.gl.sketch and jit.gl.handle as well as a camera message box and a position message box, which gives it the capability to change the camera position relative to the object and the position of the object on the 3D plane.

There was one major issue which we were unable to resolve in the allotted timeframe and that is the position of the object in 3D space. This is a big problem because 3D space is enormous and finding a relatively small object can be quite arduous. Luckily, we were able to find the object using a specific camera angle as well as a camera position and object position. This information was entered directly to the jit.gl.render object with the use of the pak position 0. 0. 0., pak camera 0. 0. 0. and finally the lookat attributes. Unfortunately, by doing so, we lost the ability to rotate the cube using the mouse and the cube itself seemed half hidden behind a sort of mask. Due to time constraints, we were unable to solve this problem, although the axis position problem might have been solved with some object we didn't know of. The final cube rendering which shows the problem stated above is shown below.

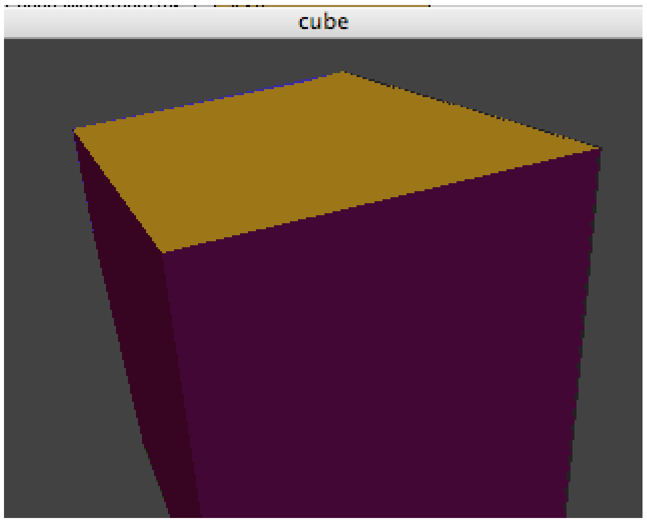


Figure 23: Output of Cube Render (=O))

References

[1] Camera Calibration Toolbox for MatLab
http://www.vision.caltech.edu/bouguetj/calib_doc/

[2] Uncalibrated Euclidean Reconstruction by Andrea Fusiello, Dipartimento Scientifico e Tecnologico, Universit'a di Verona
http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/FUSIELLO2/rectif_cvol.html

[3] Reconstruction from Multiple Views by Daniel DeMenthon
http://www.umiacs.umd.edu/~ramani/cmsc828d/lecture28_6pp.pdf