Max and Kinect

Kinect is a camera system sold as an accessory[1] to the Xbox video game system. It has an imaging system that can measure distance up to 15 feet or so. This is accomplished by projecting a pattern of dots with an infrared laser[2]- an infrared camera tuned to the same wavelength can use the distortion of the pattern to calculate distances.

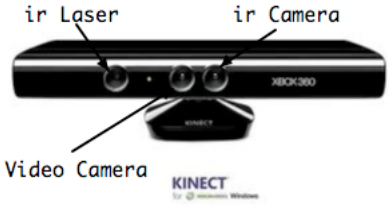


Figure 1. What the Kinect IR camera sees

There is also a standard camera in the kinnect.

Kinect can be connected to Max/jitter via the jit.freenect.grab object which was written by Jean-Marc Pelletier, Nenad Popov and Andrew Roth. It is available at jmpelletier.com. Jean-Marc is also the author of cv.jit, a very powerful suite of computer vision processing modules.

The jit.freenect.grab object works much like jit.grab. It requires an argument to determine the type of output matrix to produce. The matrix dimension will always be 640 by 480. An open command makes the connection, and bangs produce the video flow. There are also commands to operate the tilt mechanism, set the format for the center outlet (ir or video camera) and other functions.

[1] Circa 2011

[2] This thing gives me a headache if I look into the camera for very long, so don't work too close.

Figure 2. The heart of any kinnect patch.

There are four output modes: mode 0 provides the raw data, which is a distance from 0 to 2047. Anything that the kinnect cannot map is given a value of 2047. This includes objects closer than 20 inches and further than 15 feet, as well as a "shadow" that occurs on the left side of nearby objects where the ir camera cannot see the laser projection. The range does not extend all the way to 0. The nearest value (at 20") is about 375—one step below this and the value flips to 2047.

For mode 1 or mode 2 the matrix type should be float64. The values are normalized to the range 0.2 to 1.0. In mode 2 the values are reversed so closer items have higher values, up to about 0.8 and the no read value is 0.

Mode 3 is supposed to be distance in centimeters, but appears to be broken.

Detecting Hands

The image in figure 2 is just the jit.pwindow's best attempt to interpret the distance data. We generally process the information without looking at it. One process is to simply detect any pixels that fall within a specified distance range, such as further than 0.4 and closer than 0.43. I'm not really interested in the precise distance here, just something that can be grabbed with a minimum of processing. Figure 3 shows the technique. Jit.expr takes in the distance data and performs some logic. The term in[0] represents the current cell from inlet 1. You may remember that the result of a greater than (>) or less than (<) operation is either 0 or 1. The results of these are (because of the parentheses) subjected to the logic AND operation (&&) which only returns 1 if both tests were one.

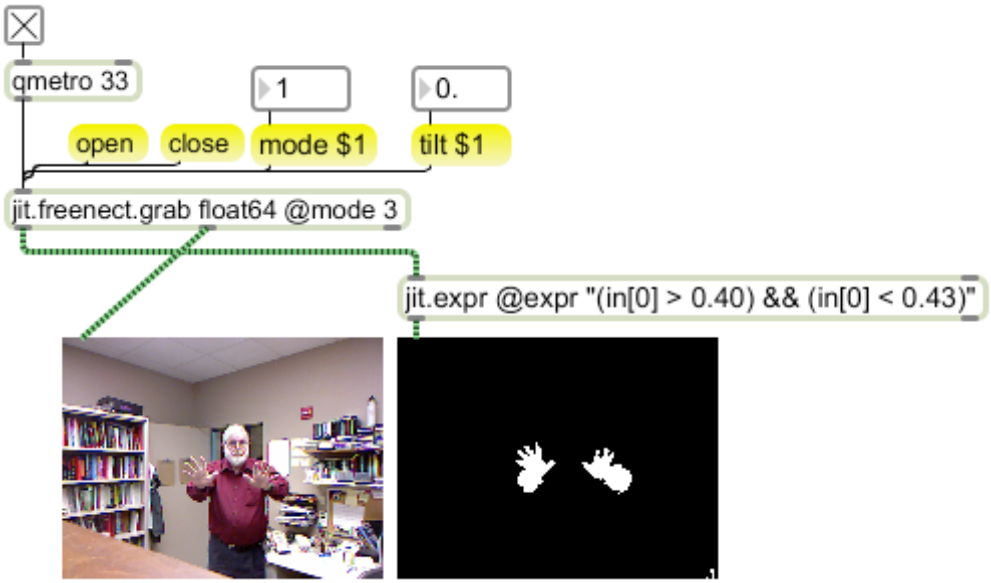


Figure 3.

Figure 3 gives a sort of cross section of the image. The easiest way to put this to work is break the image into zones and detect any white within the zone.

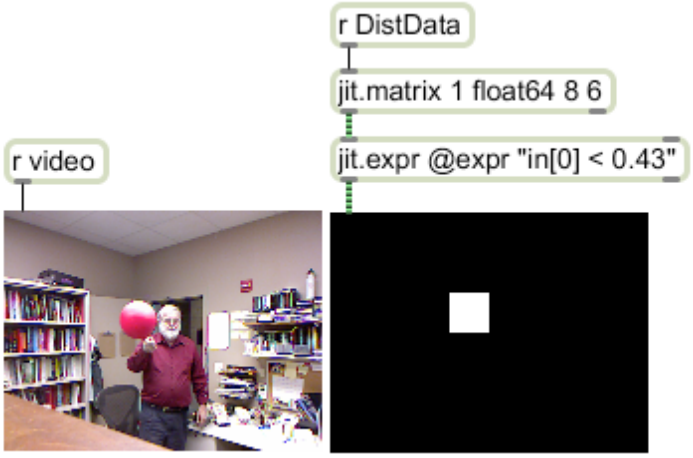
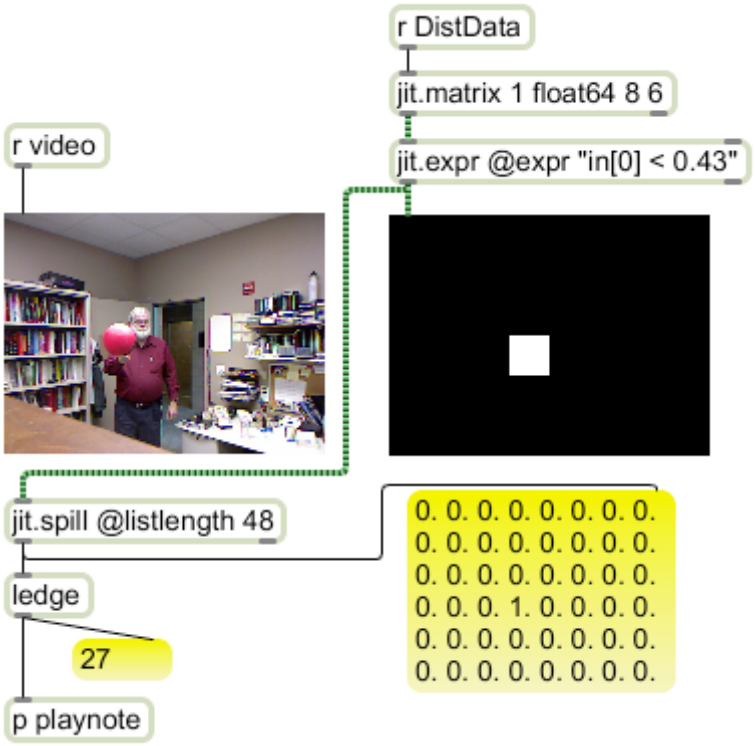


Figure 4.

Figure 4 shows a simple way to get zones—just reduce the size of the matrix. (I have simplified the drawing by sending distance data and video from the patch in figure 3.) I'm using a ball on a stick to test the action. To make music from this, use jit.spill to convert the matrix into a list as shown in figure 5.



Fingure 5.

Jit.spill is simple to use when the entire matrix is to be converted. The message box is sized to show the values in columns to match the matrix, but the data is a list. Once the data is in a list, ledge (an Lobject) will report the left edges of any block of 1s. The single 1 is at index 27 (the first index is 0). It's not hard to transform that kind of number into a note or control values.

Playing with this patch, you will quickly discover that it is difficult to trigger the corners. That's because the distances reported by connect are really the radius of a sphere, and the natural tendency is to play in a plane. One way to deal with this is to use only the center of the image—there's plenty of resolution, but if you want to reach to the corners, you can apply correction as shown in figure 6.

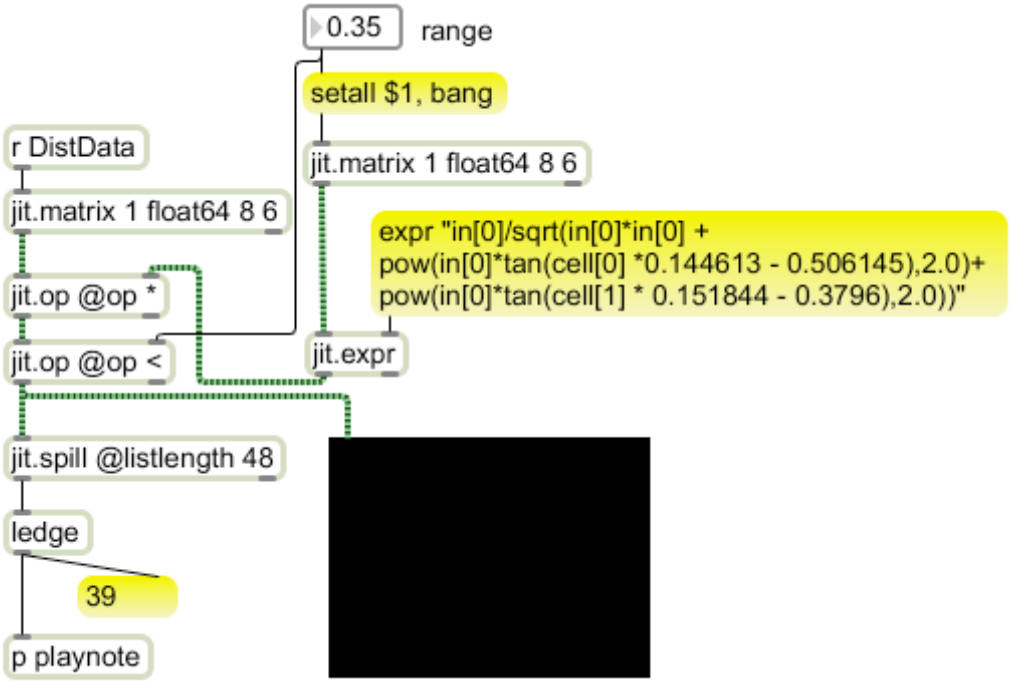
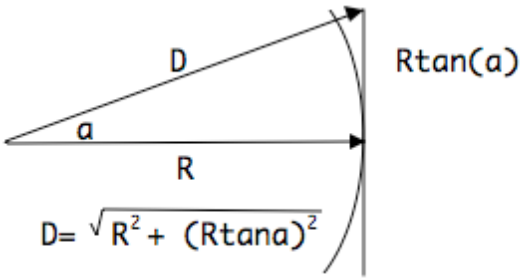


Figure 6. Corner correction

Note that the correction is applied to the reduced matrix to save computation. The expression looks intimidating, but that only happens once. The jit.expr produces a correction matrix that is multiplied by the matrix to be tested. The formula we are applying is R/D where R is the radius of a sphere that touches the plane we are playing in at the center. R is the value we use for the threshold of "crossed the plane". D is what the kinect measures for each point in the plane. This equals R at the center of the plane, but is larger at the corners. If we figure out D for each point (or square) R/D will correct the D to match R.



If you think of the plane as X by Y squares (with 0,0 at the top corner) the point at x=X/2 and y=Y/2 has D equal to R. Any other point has an angle *a* up from the kinect lens and an angle *b* to the right. The angle *a* is equal to x times the angle between any two squares, minus X/2. The angle of the entire width is 58° so 45/X is the angle between squares. The angle *b* is figured out the same way, knowing the height is 43.5°. Of course the angles have to be in radians .

Pythagoras tells us D for any square is

Sqrt(R^2 + Rtana^2 + Rtanb^2)

Translating that into expr-ese for an 8 by 6 grid
R = in[0]
a = cell[0]*0.144613-0.506145
b = cell[1]*0.151844-0.3796
so you get the expression in figure 6. For grids of other size N by M, the step for *a* is 1.01229 divided by N-1 and the step for *b* is 0.759218 divided by M-1. A scientific calculator simplifies this part.

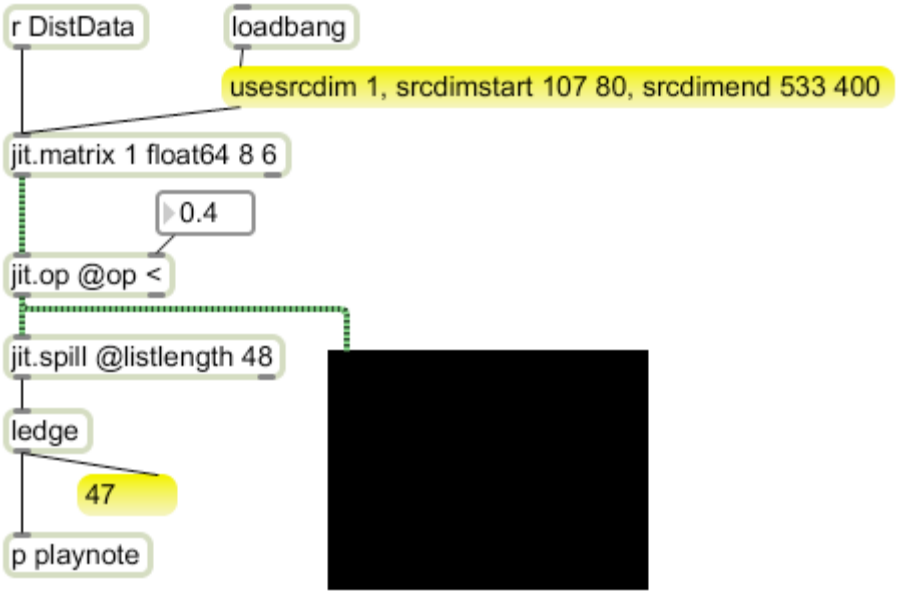


Figure 7.

Sometimes it is easier to just ignore the corners. Even with correction, the kinect does not pick up items at the edge of the field very reliably. The source dimension attribute is the way to make a matrix look at part of the source. In figure 7, I have added it to the matrix that reduces the image resolution. Source dimensions can also be used to watch different parts of the image for different functions.

Spaceframe

Here's a practical use of kinect that is getting a lot of attention. The kinect is at the heart of Tim Thomson's "Space palette", also known as multimulti-touchtouch.

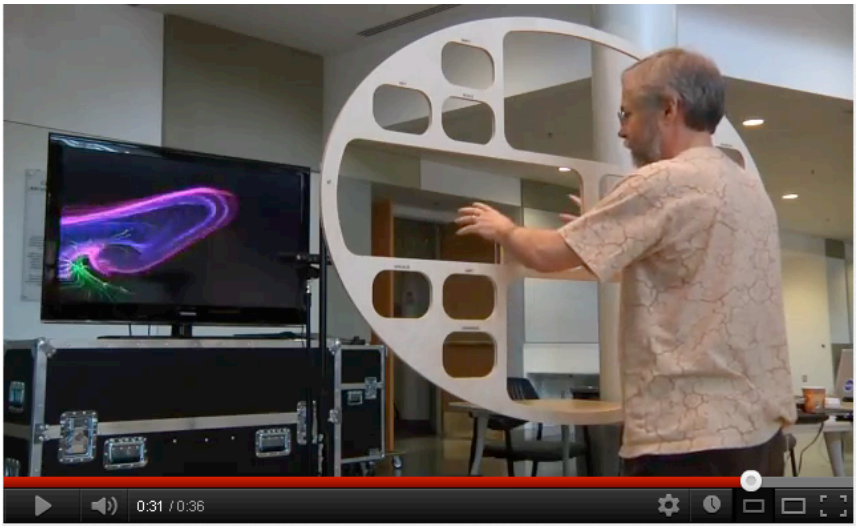


Figure 8. www.youtube.com/watch?v=oIalWiob3Z8

When the performer reaches through a hole, the kinect reacts, cueing loops and graphics. Tim is using custom software to run this, but we can make a crude approximation with Max. First, the kinect's view of a somewhat modular frame:

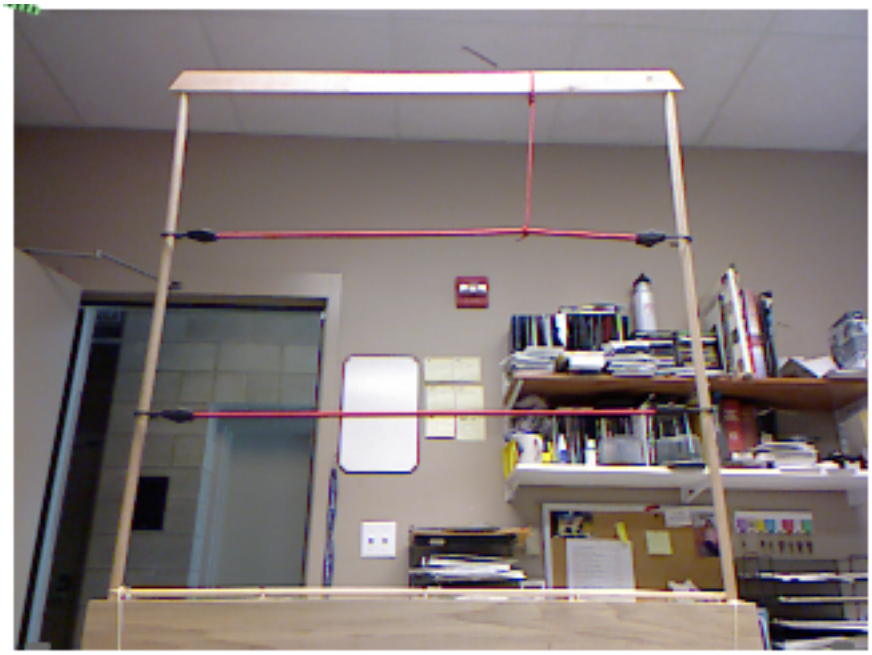


Figure 9.

The red cross cords are bungee straps, and the vertical cords are parachute cord tied around the bungee or the frame. This system allows you to set up a configuration suited to your performance. Next we need a Max patch to detect actions in the various areas. The patch must have several features:

It must correct for the geometry of the camera and frame. Tim's palette deals with the corner correction problem by being circular—this frame avoids the problem by reducing the size of the active area (although there's no rule against playing outside the lines), but adds a new one by setting the kinect even with the bottom of the frame. This produces a keystoned image.

It must provide easily adjustable zones. Tim has built his paradigm in with a jigsaw (after a year of experimentation), but I expect each student to have their own idea how to use this thing. Once set up, the zones must be saved with the patch.

It needs to detect a variety of activity within the zone. So far I've come up with:

- Presence in the zone.
- Horizontal position in the zone.
- Vertical position in the zone.
- Horizontal motion in the zone.
- Vertical motion in the zone.
- X – position in the zone.

This is necessarily a complex patch. Like all complex patches, it is an assembly of simple systems, which we can work through one at a time.

Input Kinect Data

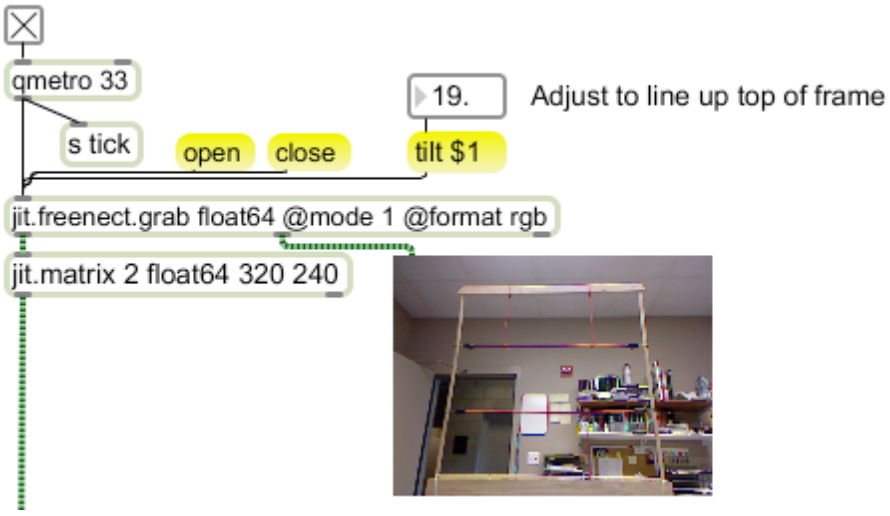


Figure 10.

Figure 10 shows the input stage. This is standard freenect.grab stuff, except I am reducing the resolution of the image to 320 by 240. This improves response time

significantly, and we won't miss the resolution. Note the use of the tilt control. This allow the kinect to be placed level with the bottom of the frame looking up.

Compensate for Geometry

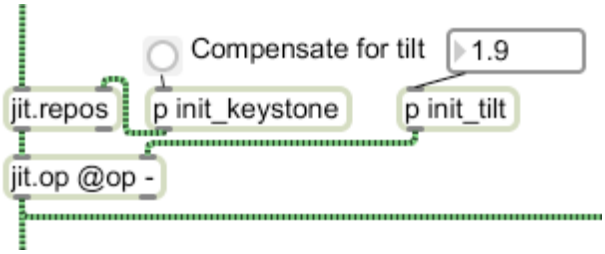


Figure 11.

Figure 11 shows what is needed to compensate for the position of the kinect. Looking at figure 10, you can see that in the uncorrected image the frame appears to narrow towards the top. We will fix this with jit.repos[3], and in the process, reverse the image right to left. (This makes the computer display match the performer's view of the frame instead of the kinect view.) We will also subtract a set of values from the kinect data to make up for the fact that the top is a foot farther away from the kinect than the bottom. Calculating the jit.repos control matrix is complex but only needs to happen once. This part of the patch has three layers of subpatch shown in Figures 12 to 14.

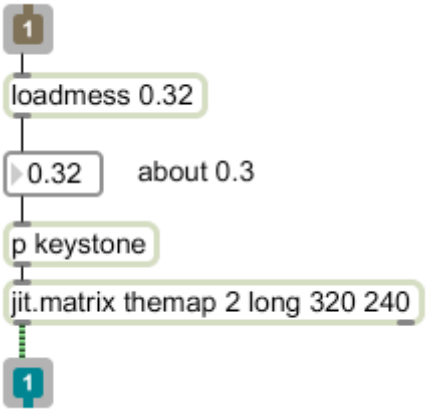


Figure 12.

This is the outer patch. We need to fill the map matrix with control values for jit.repos. Each cell will contain the address of the cell that will replace the corresponding cell of the jit.repos output.

[3] Jit.repos distorts an image by means of a control matrix. For each pixel of the input image, there is an address (X,Y) in the control matrix. The pixel at this address replaces the original pixel. I have written a tutorial about this.

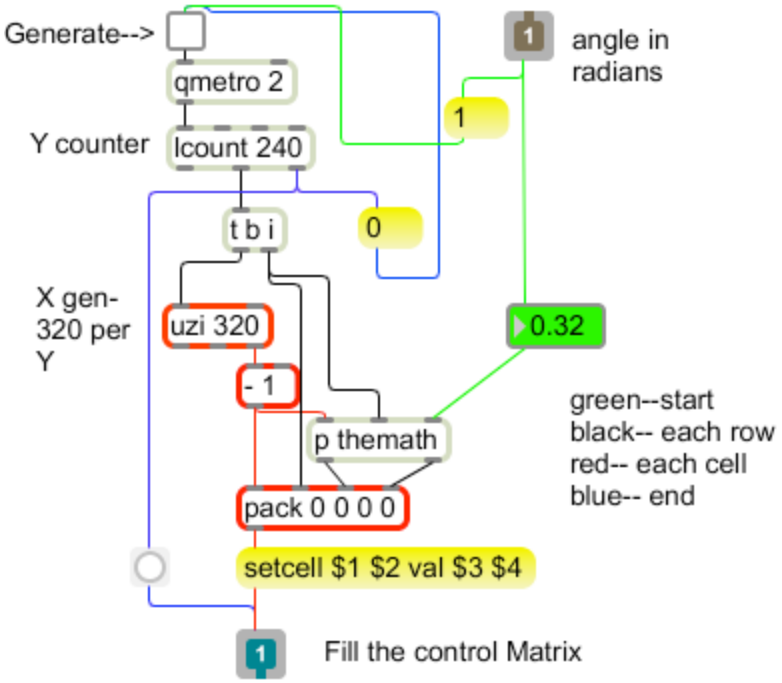


Figure 13.

The patch in figure 13 sends the address of each cell in the control matrix to the math subpatch. The color coding indicates what parts are active at any time. The patch is activated by arrival of an angle in the inlet. This starts the process via the cords shown in green. When the metro turns on, lcount steps through the rows of the matrix. The red section, powered by uzi, triggers for every cell in a row. The calculations based on the cell addresses are contained in the math subpatcher, shown in figure 14. The results are sent to the control matrix at that address. Finally, when lcount finishes the last row, the wiring shown in blue stops the metro and sends a bang to the control matrix, which will load it into jit.repos.

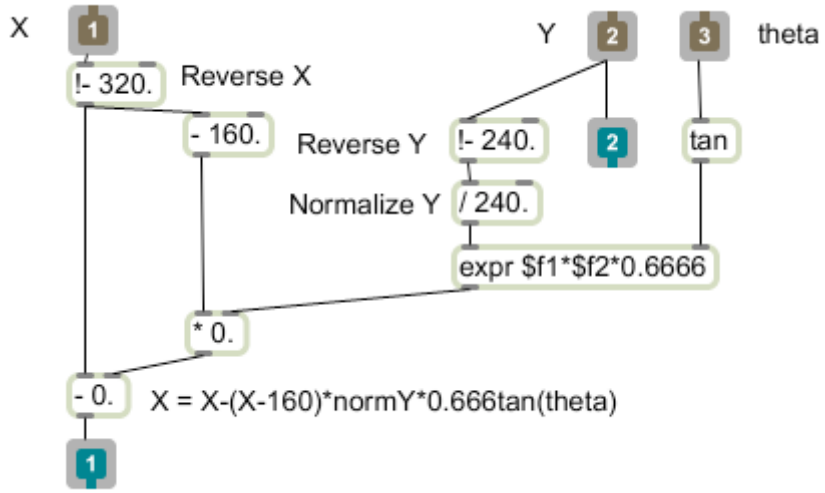


Figure 14.

The math subpatch in figure 14 calculates a source pixel for each pixel in the image. The Y input changes on each row and the X input on each cell. Note that both X and Y are complemented[4] to count backwards. Complimenting X reverses the image left to right. The Y value generates a factor that will be subtracted from X, essentially moving the pixel sideways. The factor depends on both X and Y. Y is normalized by dividing by the height of the matrix- this will make Y step from 1.0 to 0. Theta represents the angle of the edge of the frame—the distance the pixel must move is Y times the tangent of theta. The result is further multiplied by 0.666, which is the ratio of half the width to the height of the image—we subtract 160 from X to make the adjustment symmetrical around the center.

There is still the tilt correction to make. This is done by subtracting a value proportional to the height of each pixel from the distance data. The values to subtract are contained in a matrix that is calculated when the patch is loaded. The calculation of these values is shown in figure 15.

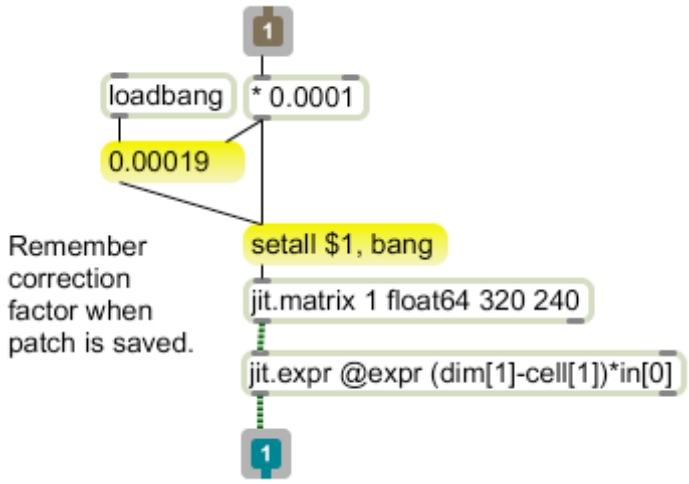


Figure 15.

All of the calculations in figure 15 are done in the jit.expr object. This applies a formula to every value of its input matrix. In this case the formula is similar to what happens in figure 14. The expression (dim[1]-cell[1]) produces the reversed Y address for each cell. (The subscript 1 refers to the second dimension of the matrix.) This is multiplied by the value in the matrix at the left inlet, indicated by in[0]. This matrix contains a tiny value in each cell, which is the difference in distance at the top and bottom of the frame[5] divided by 240. We can change this if necessary by entering a new value. I'm dividing this value by 1000 to avoid typing a lot of zeros. The value shown was produced by typing 1.9. The fact that this value is similar to

[4] Subtracted from the length of the dimension.
[5] As measured by the kinect.

the tilt value is a coincidence. The value is also affected by the distance from the frame to the kinect.

The values shown in figures 14 and 15 work with the distance I used in my tests. I have made this repeatable by attaching a measured bit of string to the frame. When this string is pulled so the knot touches the kinect at the center (between the lenses), the distance is correct.

Detect Images

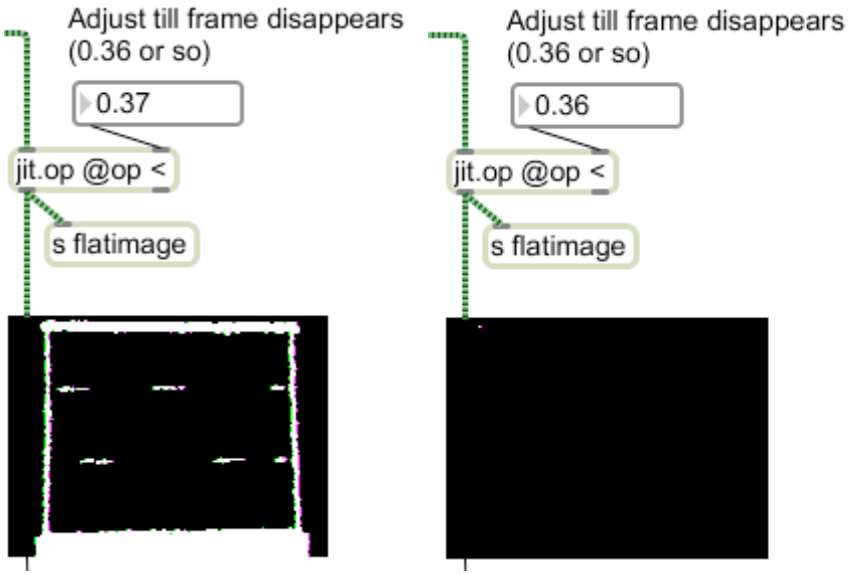


Figure 16.

The signal out of figure 11 branches. Figure 16 shows the right side of the branch, which is used to detect anything closer to the kinect than the frame. The detection is simple, anything less than the value in jit.op will produce white pixels. The sensitivity is adjusted in two stages. If it is set at 0.4, the outline of the frame will be visible. The optimum point is then found by scrubbing the value down until the frame disappears from the window. If the tilt correction is set properly, the frame will disappear all at once (more or less). If it is not, either the top or bottom will disappear first. This trick will also show if the kinect is parallel to the frame. If one side of the frame disappears before the other, adjust the angle of the kinect. This processed image is sent to various detectors via send flatimage.

The other branch from figure 11 goes to figure 17, which is used to set up detection zones. Figure 17 also has a jit.op <, but this one is set so the frame shows. The image is passed through a jit.lcd to display boxes for the various zones. Each detector is based on the source dimension trick of figure 7. Dimensions can be entered by drawing on the window in figure 17. The mechanism that makes this possible is shown in figure 18. When the mouse is clicked in a jit.pwindow, the message [mouse X Y buttons] is sent out the right outlet. Buttons includes values for modifier keys, but all we need here is the first number after the X and Y, which is 1 if the mouse is

down and 0 if it is not. When the mouse is dragged across the picture, X and Y change appropriately. When the mouse is released, one more message is sent, with the final location and the button at 0.

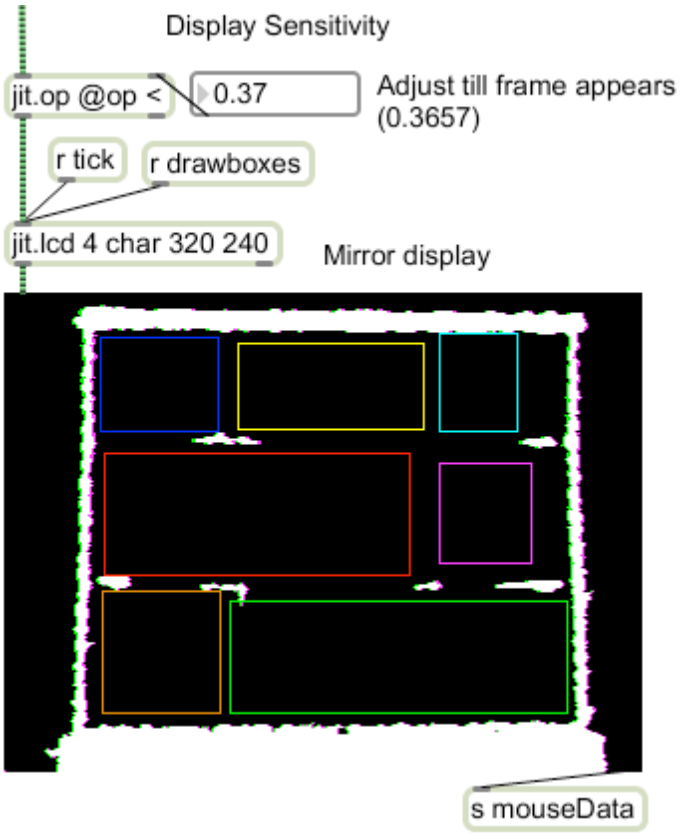


Figure 17.

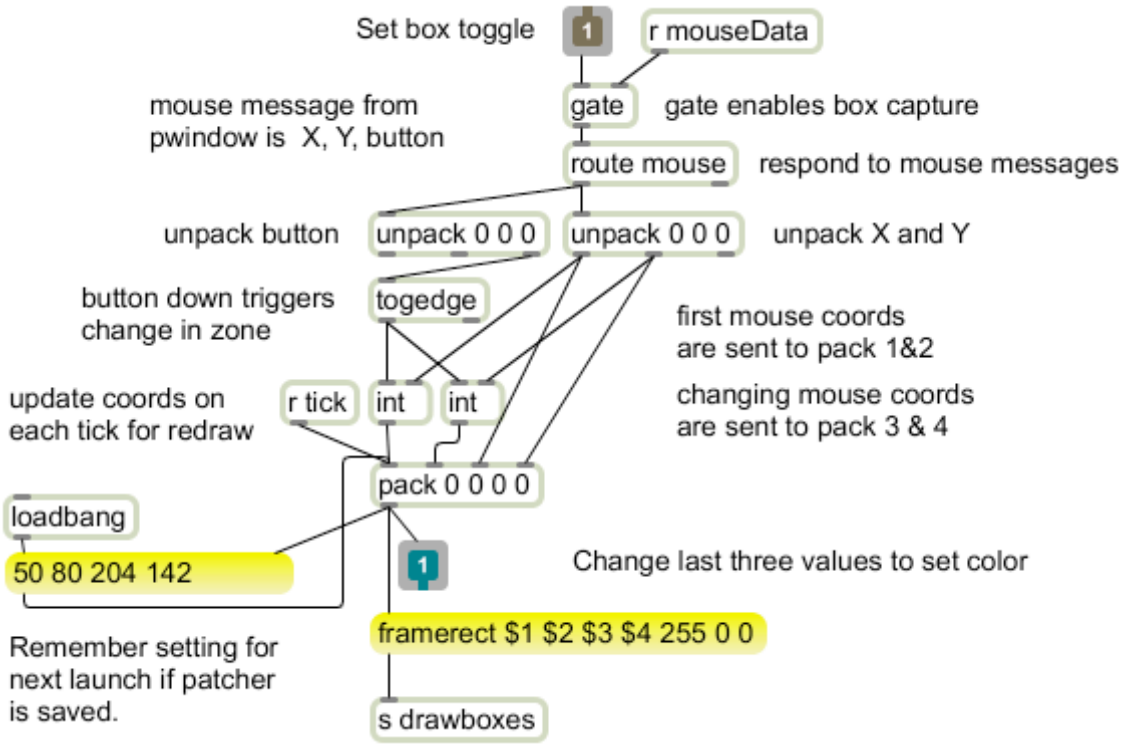


Figure 18.

The patch in figure 18 is in a subpatch labeled markzone. A toggle connected to the inlet activates the patch and allows the zone to be changed. The output of makezone is a list of the first place (X,Y) the mouse was clicked and the current location. This information is added to a framerect message and sent to the jit.lcd. The last three values in the message set the color of the box in RGB format. When you copy this patch, change the numbers so each box is unique. The box will follow the mouse as long as the mouse is held down, and then stick. Uncheck the toggle to prevent further changes. Note that the rectangle list is captured in a message box. This will be saved with the patcher so the box will reappear the next time the patch is loaded.

Figure 19 shows the complete kinectFrame patch. This can be used as the basis for your own patch and customized for each piece by adding detectors—these can be in this or another patch. I prefer to keep them separate, with a patch for each piece.

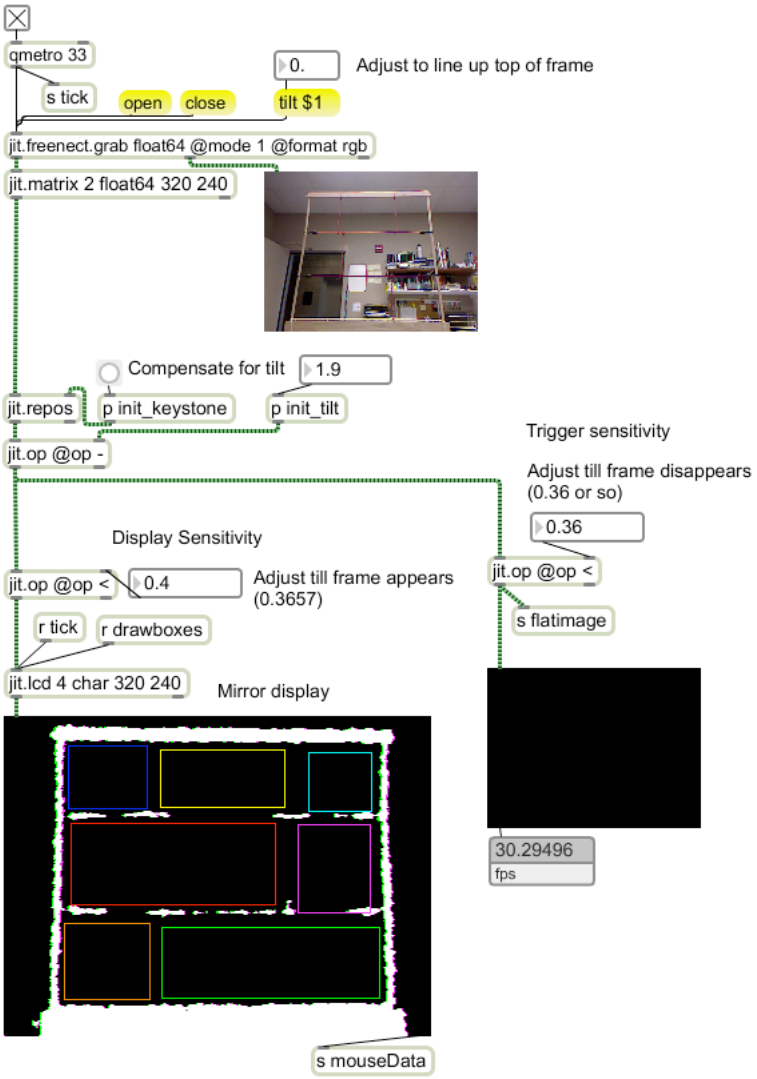


Figure 19

Action Detection

We can detect various actions by adding small patches that communicate with kinectFrame via send and receive objects. I've included a set of examples in the KinectDetect file.

Spot Detection

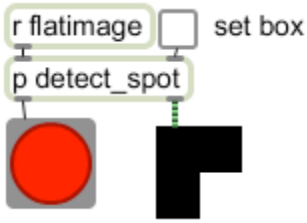


Figure 20.

The patches that detect action in the zones are quite similar. Figure 20 shows the simplest, which just reacts to the presence of anything in a zone. The patch receives the image from the KinectFrame patch via receive flatimage. There is a toggle that will enable a markzone mechanism, and something to signal the output, which in this case is a bang. A bang could be used for any number of things, such as starting a loop.

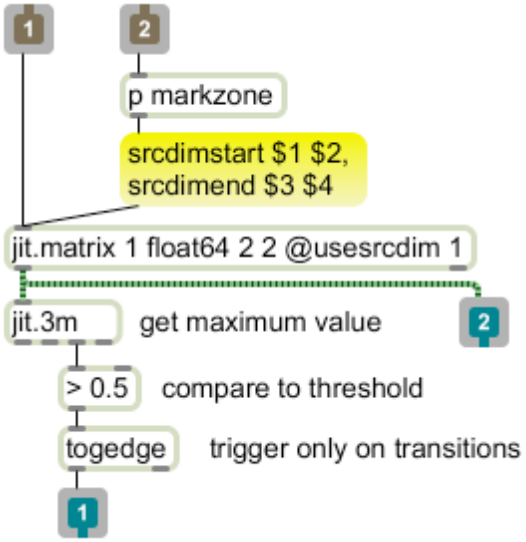


Figure 21.

In the inner patch shown in figure 21, you can see how the result of markzone sets the source dimensions of a matrix. The size of the matrix will vary according to use, but it should be at least 2 by 2. When an image that consists of a white blob in a black field is interpolated down to low resolution, the result could be just black if there is only one pixel in a dimension. On the other hand, we need a lot of these matrices in a complex setup, and the size directly affects processing speed.

The detectors will differ in the treatment of the output of the matrix. In figure 21, jit.3m reports the maximum pixel value, which will be close to 1. This is detected by a > object, which produces a 0 or 1 on every frame. The togedge object will bang only when the > transitions from 0 to 1.

Position Detection

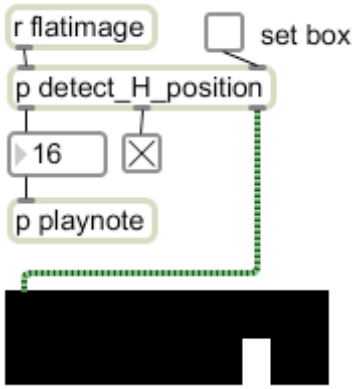


Figure 22.

The next step up is to detect position from left to right. My example uses this information to play notes, but that is really not very satisfactory. A better use would be some kind of control, perhaps of volume or tempo.

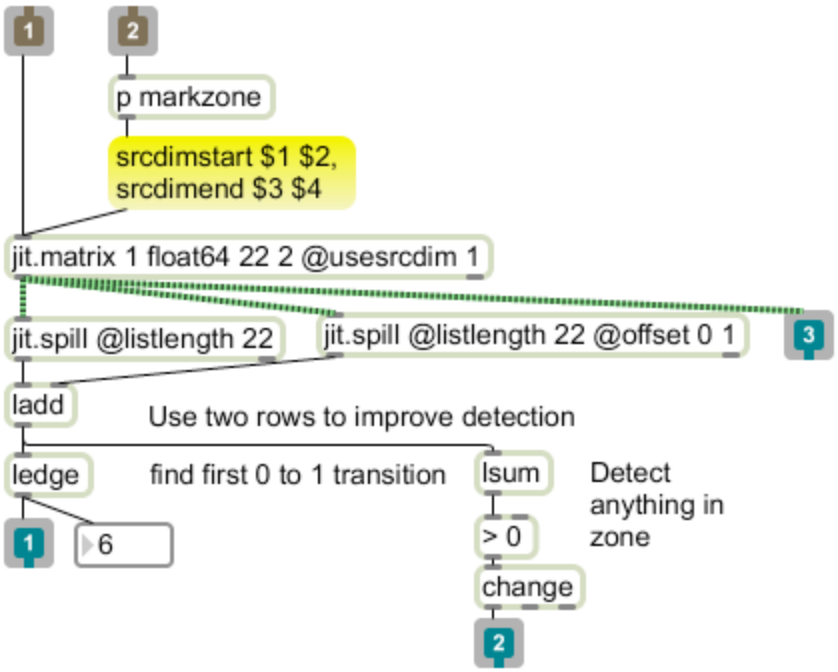


Figure 23.

The top of this detector is the same as figure 21, with a matrix that is wide but not tall. The position of the white part of the image is determined by converting the rows to a pair of lists with jit.spill. The offset attribute sets the second spill to read the second row. These lists are just added together, because I want to detect any white in either row. The ledge (say L-edge) object will report the positions of any 0 to 1 (or greater) transitions in a list, which is what we want. If you stick two hands in the zone, ledge will report both of them, which may be useful or a problem. Just

run the list through a number box to discard extra data. The playnote subpatch doesn't do anything more complicated than subtract the number from a high value (to make the notes play with the low ones on the right) and apply the result to makenote.

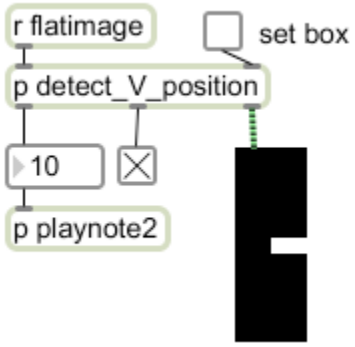


Figure 24.

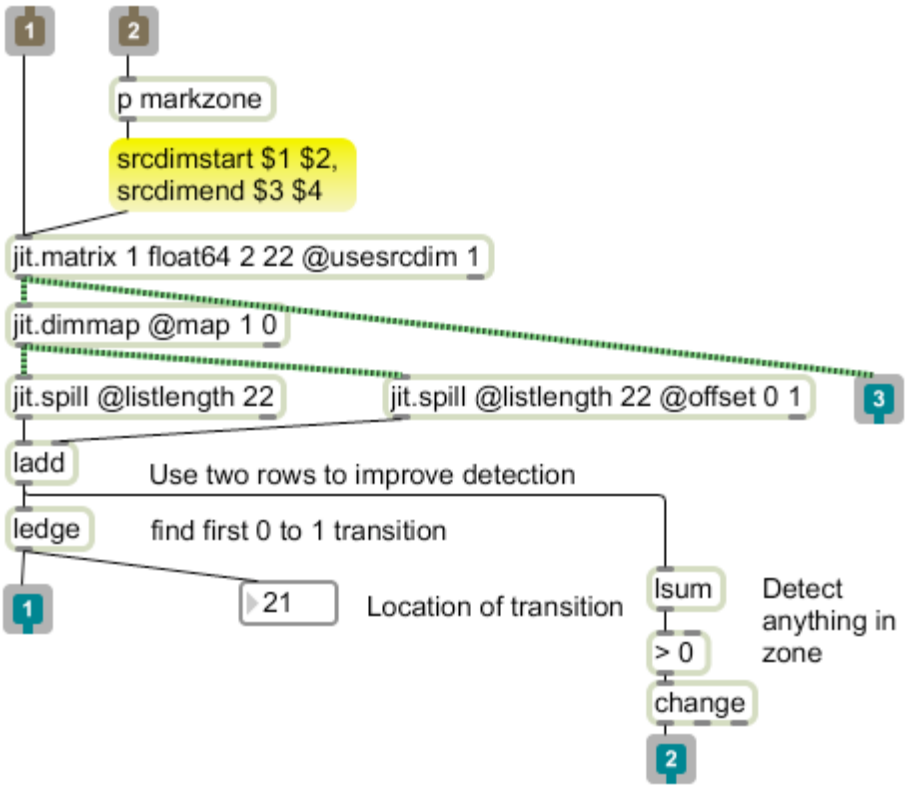


Figure 25.

Detecting vertical position is shown in figures 24 and 25. These use the same patch as horizontal detection with two changes. The matrix is shaped to fit vertical motion and then turned on its side with jit.dimmap. Otherwise, figure 25 is identical with figure 23.

Detect Motion

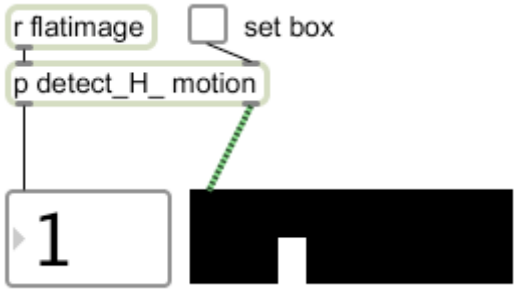


Figure 26.

In Tim's videos, you will often see him trigger something by waving his hand through a zone. The framerate and resolution of this patch is not high enough to accurately detect the velocity of motion through a zone, but we can easily capture the direction, which may be just as good.

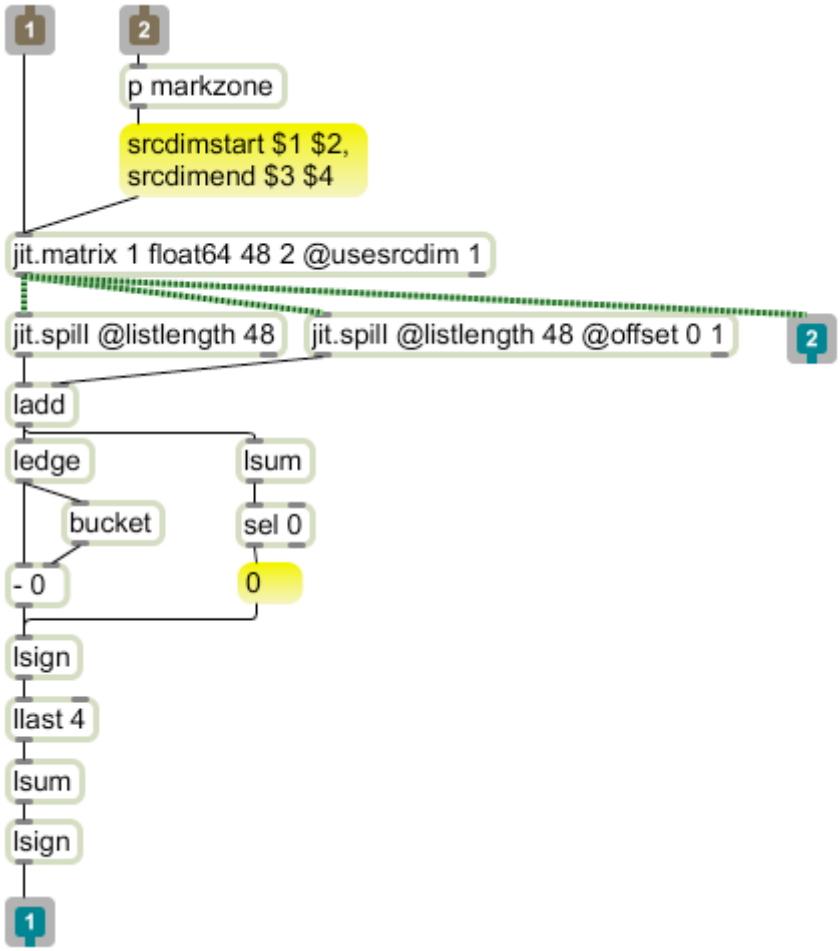


Figure 27.

Figure 27 is built on the horizontal position detector. I have added a bucket at the output of ledge—bucket holds a value and outputs is when a new value comes in. Thus the subtract object is comparing the most recent output with the last output. My lsign object reports 1 if the input is positive, -1 if the input is negative, or 0 if the input == 0. (You can easily do this with if statements). I found this to be a little unsteady with slow movement, as the ledge value will only change when the white image crosses a pixel boundary. Llast 4 collects the last four outputs. These are added up and the sign found again. This makes the patch slower to detect a reversal, but reduces the number of 0s with slow motion. Modifying this to detect vertical motion only requires a jit.dimmap as demonstrated in figure 25.

X-Y Position

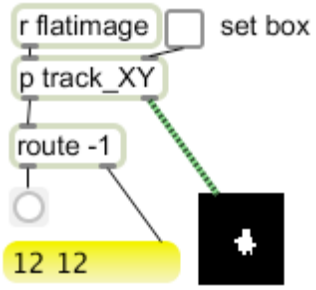


Figure 28.

You could detect both horizontal and vertical position by combining two detectors in the same area, but it may be more efficient to use jit.findbounds.

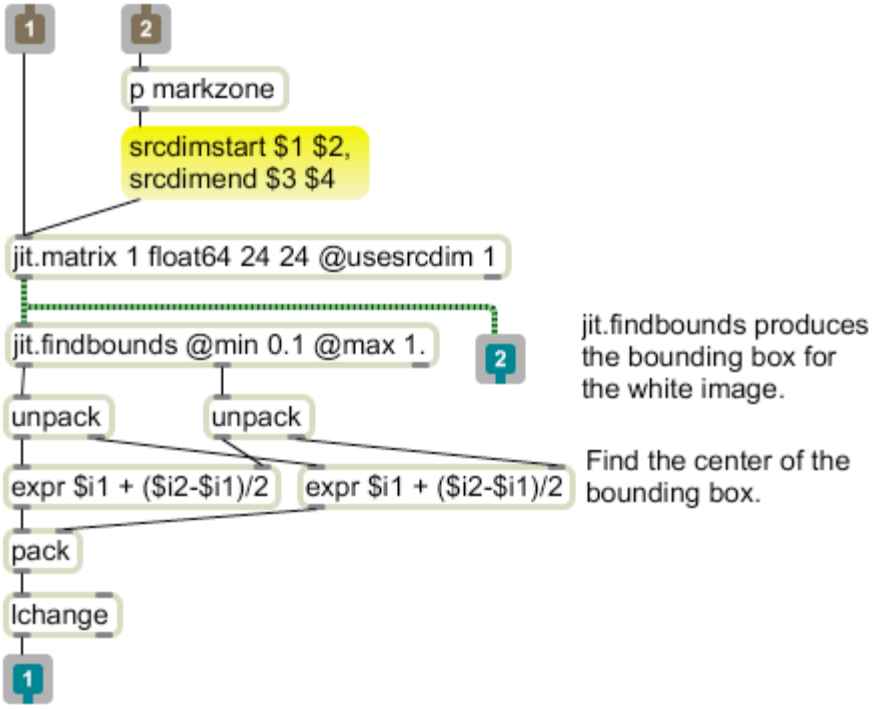


Figure 29.

Jit.findbounds reports the coordinates of the upper left corner and lower right corner of a box the encloses the image. If there is no image, it reports -1 for all values. In figure 29, expression objects find the center of the box, although you may find the corner locations more interesting. Note that the center of the white image is unlikely to be at 0,0. The -1 may be useful as an absence detector, or it may be discarded, either way, route -1 does the job.

You can build your own assortment of detectors and use them in a file in conjunction with KinectFrame. The various box settings are saved with the patch, so it should play as soon as it is loaded, assuming KinectFrame is set up properly.

Setting Up

The KinectFrame patch is tweaked to work with the kinect and frame in a particular relationship. If we use this in a concert, it is best to standardize this relationship so the audience doesn't have to watch us measure and move things.

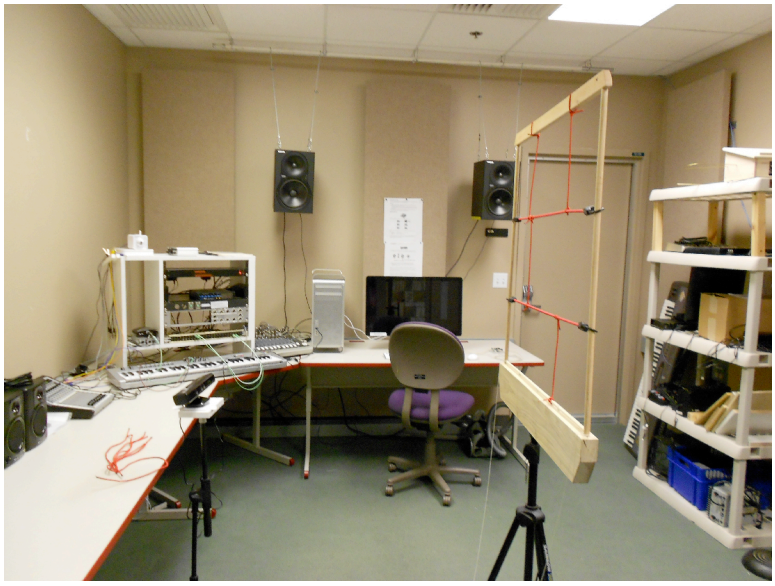


Figure 30.

The kinect should be mounted on a stand so the camera lens is the same height as the top of the bottom bar of the frame. The stand has to be placed close enough to the computer for the USB cable to reach, and there must be a pulg-in nearby for the Kinect power supply.

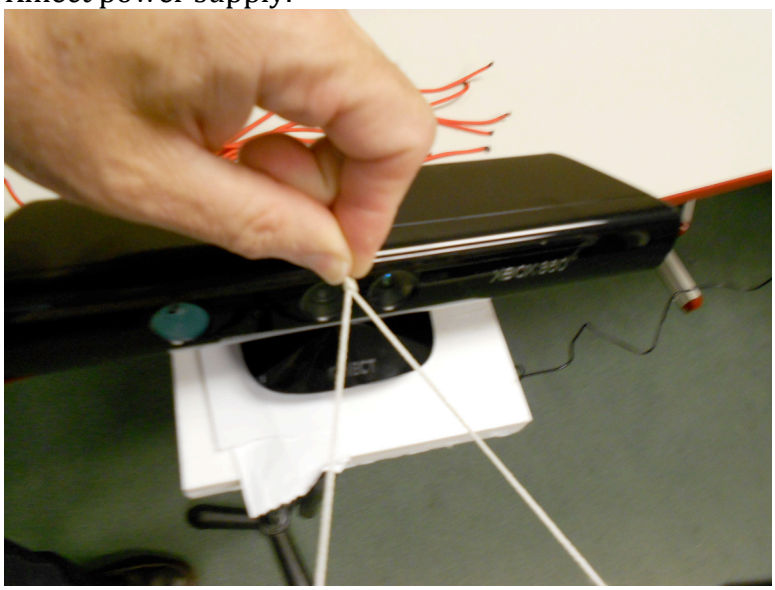


Figure 31.

There is a string attached to the bottom of the side posts. This string has a knot in it, and if the string is stretched forward in a triangle, the knot should just touch the kinect between the cameras. (The window on the left side is the ir projector.) The kinect and frame must be parallel, but this can be tweaked once an image is avaiable.

Load kinectframe. (Note: this patch requires jit.freenect.grab and my lobjects.) Start the metro, then open the jit.freenect.grab object. Sometimes you have to close and open the object again. If the video image appears, it is working. Set the tilt to 19.

The displays will both look like this:

Figure 32.

Rotate the Kinect until the frame is centered in the window. (Double check the string position after you do this.) If necessary, put a paper shim under one side of the kinect so the top bar is parallel with the window. This positioning affects everything, so make it as accurate as possible.

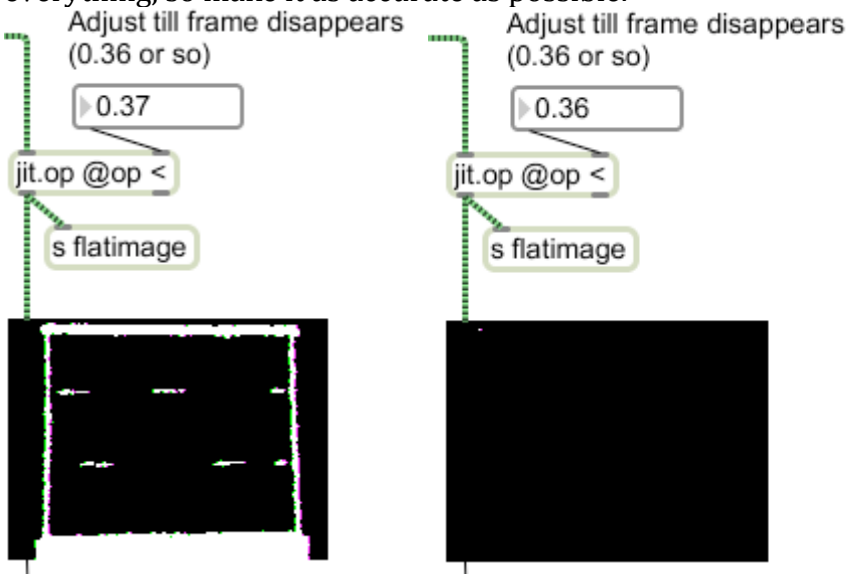


Figure 33.

Now adjust the trigger sensitivity so the image just disappears, and you are ready to go.