Introduction

As I am sure we all know, it can be difficult to keep an eye on your pet all the time. That is the idea promoting this project. I am presenting a beginning implementation of a system designed to keep an eye on your animal for you. Using the Microsoft Kinect device, it is possible to create a depth and image stream of your pet and not only be able to watch them from afar, but also have the system monitor them. The system I have designed uses this device and a modified version of Microsoft’s SkeletalViewer. I have added a CSkelNetworkServer class which is used to both handle the incoming connection and process the images.

Microsoft’s Code

In order to show what I do, I will write about how the Microsoft Kinect SDK presents data, so that my processing can be understood. When connected, Microsoft’s code performs skeletal processing in the background and I have not as of yet been able to locate where this is done. The provided framework handles all communication with the Kinect in the background also. When a new image or depth frame is captured, a callback function is then called in the code with this data. The image frames are provided in 640x480 and the depth in 320x240. When an individual steps in front of the device and is recognized as a person, the system also calls a callback function indicating that skeletal information has been processed. It also changes how the depth frames are presented.

The pixels for the image are presented as an RGBQUAD, basically an RGBX format. The depth pixels are sent as an USHORT, which is a 2 byte unsigned integer. The entire 16 bits are not used, however, only the 13 most significant bits actually contain the depth. The 3 least significant contain the player number. The system is currently designed to handle up to 7 players, with 0 being no player. Every time the SkeletalViewer processes a depth pixel, it assigns it a color based on the player number, so I have used this concept for my code.
Implementation

The current system preprocesses the depth and image information provided by the Kinect and then streams the data to the connected system. The streaming is handled by a simple TCP connection and requires no further discussion. I will now discuss the preprocessing that is done.

When a depth frame is received, and before it is converted by the existing code, I send it to the CSkelNetworkServer to preprocess. This is where the core of my code is. In order to easily be able to process new ‘players’, I decided I must first remove the background completely. I was having issues where the floor/walls were becoming part of the object. To do this, I have a calibration phrase, which is composed of the first 200 frames of depth. This value was decided arbitrarily and could still use optimization. Before this is done, a 320x240 USHORT array is initialized to the maximum supported depth. Then, to process, for the first 200 frames I store the minimum seen value for each pixel. This assumes that the Kinect will be completely stationary and the background will not change, but it can always be recalibrated. I also ignore all zeroes that are seen. I had noticed a shadow on objects and after a little research found out the zero depth basically means there is an error with how the two cameras line up and this information is not needed.

Once the background frame has been initialized, the depth information can begin streaming. I then process the new depth frames so that each pixel is only displayed if the depth is 5cm or closer than the background, therefore completely eliminating the background from the picture. As I do this, there is another frame I create that contains only 1s and 0s. I place a 1 wherever a new object is. Then, the next loop goes back through this frame and attempts to process each individual object. I currently allow up to 512 objects accounting for noise, but plan to have future work eliminate noise. The next loop simply looks at each pixel going from left to right, top to bottom. When it encounters a 1, it then stores the coordinates of the first found pixel and switches to search mode. Search mode is a breadth-first search (BFS) algorithm that assigns a player number to each pixel of the current object. It first assigns the current player number (starting at 2) to the current pixel, and then to each adjacent pixel (vertically and horizontally only) until it finds the entire object. It then resumes the linear search until the entire frame is processed. The loop will recheck previously checked pixels, but they will no longer contain a 1 and it will bypass them.

Next, I modified the SkeletalViewer code a little so that objects I have found that have not been recognized as a person by background system are processed differently. They are currently displayed in solid blue with the top few rows of pixels highlighted red. This is just to show where you would begin when trying to look for the head of an object (animal). It is of course temporary and will be modified as future work progresses.
Finally, the images are compressed with a very simple and efficient compression method. This was done in order to improve frame rate, as 640x480 24-bit RGB images can require a lot of bandwidth. To make it easy, each image is shrunk to be 160x120, exactly ¼ of the depth image and 1/16 of the image image. This is done simply by selecting squares of pixels and averaging them. This was a quick process and did not seem to distort the image too much, which is what I had expected as it looks like most pixels are similar to their neighbors except near the few borders. 160x120 was the limit, though; anything smaller gets harder to see. These compressed images are then sent over the network to the client and displayed.

Figure 1: Example Run

**Future Work (Problems and possible fixes)**

The system still requires a lot of work, so I will go through what I would like to work on in the near future. First, there is a lot of noise in the object finding process. If there is a slight murmur at a pixel, it will create a single object just for that pixel. Obviously, this can create a lot of unnecessary processing. I would like to ignore these murmurs up to a certain threshold. This can be a ‘pixel mass’ threshold and perhaps a height-to-width ratio, or maybe a ‘pixel density’ measure of how many pixels are in the object compared to an encompassing rectangle.

Then, the networking code needs revamped also. Currently, the server waits for a connection before the program even starts running. It then creates a connection and begins streaming. If the connection is terminated it cannot be restarted without closing it and restarting the program itself. I need to learn
more on how to work with Windows Multi-threading and Networking so that I can design a more robust networking scheme.

Another important item, once the network is fixed, is the client-side user interface. Currently it simply displays the frames as received. What I want to do is create a backchannel that can provide control information, such as which images to display, etc.

Next, the BFS algorithm needs serious work. It currently is a quickly-implemented recursive version which is sub-optimal. I would like to take a little time and turn this into an iterative algorithm in a way to minimize overhead and running time. Every once in a while the code produces a stack overflow error that is traced to when the BFS code is called, suggesting that the program is making far too many nested recursive calls.

Another simple thing that needs updating is properly dividing objects. As is, depth is not a factor the BFS algorithm either, so if you were to place two objects a few meters apart and overlapping it would detect them as one object. This should be a simple fix.

Finally, I need to actually place a skeleton on the dog/animal so that information can actually be gathered for processing. This work will be the most complicated of the lot and I have a few ideas on how it may be done. The most promising idea I can think of so far is to simply start at the middle of the object and perform a BFS, looking for the farthest points. This quite conceivable would be able to locate 6 or 7 key points, these being the 4 legs, tail, head, and snout.