Greenfoot kinectserver protocol

Neil Brown

April 12, 2011

1 Endian-ness

All the integers mentioned in this protocol are transmitted **big-endian**. This is done because it was a good fit with Java. However, all x86 hardware is little-endian, so you can't just memcpy (or similar) between the network buffer and an array of ints (well, you can, but you'll have the wrong byte ordering).

2 Overview

At a high-level the protocol is very simple. The server accepts TCP socket connections on port 0x2020 (8224 in decimal). After the socket connection is made, no data is sent by default. The high-level protocol is simply that a client sends a single request (see section 3). and the server sends a single corresponding response (see section 4). After this, a client may send another request and receive another response. The server will never send the data from the same frame to the same client twice, but it may miss frames if the client requests are too slow or the machine is under a heavy load.

3 Request

The client request consists of a protocol version, followed by version-specific data, which will then get a version-specific response. Currently only version 1 is fully supported (0 exists but is deprecated).

3.1 Version 1

The client request is always 12 bytes:

Type	Name	Value
4-byte unsigned int		Protocol version, constant: 0x00000001
4-byte unsigned int	*reqWidth*	Image width (see note below)
4-byte unsigned int	*reqHeight*	Image height (see note below)

The image width and height specify the resolution at which the client wishes to receive the RGB image. The Kinect supplies a 640 by 480 image, and this should be seen as the default. You may specify any smaller image size than this which exactly divides the image by a constant, or zero by zero. Some example supported image sizes are 640×480, 320×240, 80×60, 0×0.

Often you will just want to use 640×480. 0×0 is useful to suppress sending the images if all you want is the joint data. The main reason to choose a smaller image is that it does reduce processing time on both ends, which can be useful on a slower machine (e.g. a netbook).

4 Response

The server's response is always determined by the protocol version sent by the client. There will always be exactly one response for each request sent by the client, in the same order as the responses were sent.

4.1 Version 1

		Type	Name	Value
		4-byte unsigned int		The total size of this response message (**excluding** these 4 bytes). A value of zero indicates an error in the request: the requested image size was invalid, or the requested protocol version is unsupported.
		4-byte unsigned int	*numUsers*	The number of users currently seen by the sensor. See section 5.
numUsers×		4-byte unsigned int		User id number. This will never be higher than 0x0000FFFF, but it is still sent as a 4-byte number. See section 5.
		4-byte unsigned int	*userState*	User state. This will be one of three values: 0x00000000, 0x00000001 or 0x00000002. See section 5.
	If *userState*=2: 15×	4-byte float		Confidence[1]
		4-byte float		World position, X[1]
		4-byte float		World position, Y[1]
		4-byte float		World position, Z[1]
		4-byte float		Screen position, X[1]
		4-byte float		Screen position, Y[1]
		4-byte float		Screen position, Z[1]
	Variable length	*RLE pixel labels*		Compressed user pixel labels, see section 4.1.1.
	reqHeight × *reqWidth* ×	4-byte unsigned int		ARGB pixel value; highest byte is alpha, lowest byte is blue.

Note that the alpha value (the highest byte) for the pixels is guaranteed to be 0xFF. This does mean a lot of redundant data is being sent, but it makes the alignment and manipulation easier in the Java client.

Note also that if *reqHeight* and *reqWidth* are zero, the last two items (pixel labels and RGB image) will effectively not be sent (technically, they will be there, but they will both be of zero size).

[1] See section 6

4.1.1 Pixel labels

Conceptually, the pixel labels are an "image" of 2-byte user identifiers of size *reqWidth* × *reqHeight*. Each pixel is either labelled with a zero (meaning no user occupies that pixel on the image) or a non-zero user identifier (which specifies which user occupies that pixel). Therefore doing a "cut-out" of a user involves taking the RGB image and for each pixel, looking up the pixel label at the corresponding position: if the label is the desired user, keep the pixel, otherwise replace the pixel with a background colour (or transparency).

Since the pixel labels are an array with large blocks of the same value, we use simple run-length encoding to transmit them. In total, there will be *reqWidth* × *reqHeight* labels. They can be decoded by reading one 2-byte unsigned int at a time. If the value is less than or equal to 0x7FFF, it should be taken as a single pixel label, stored, and then the next value should be read. If the value is greater than or equal to 0x8000, then the lower fifteen bits (i.e. use the bitmask 0x7FFF) are a replication count for the following value (which is guaranteed to be 0x7FFF or lower). The replication counts may go across several horizontal lines.

We provide a simple example with an 8×6 image of pixel labels. The original might be:

0	0	0	0	0	0	0	0
0	0	0	0	0	7	0	0
0	0	0	0	7	7	7	0
0	0	0	7	0	7	0	7
0	0	0	0	0	7	0	0
0	0	0	0	7	7	7	0

This would be encoded as:

0x800D 0x0000 0x0007 0x8006 0x0000 0x8003 0x0007 0x8004 0x0000 0x0007 0x0000 0x0007 0x0000 0x0007 0x8005 0x0000 0x0007 0x8006 0x0000 0x8003 0x0007 0x0000

5 Users

The Kinect sensor is very quick to notice that a person is standing in front of the camera, and is immediately able to detect their outline. It's very rare that a person is there but not recognised, but sometimes it does think a chair is a person! As soon as it thinks there is a person in front of the sensor, that person (or chair) will be assigned a user id, and their outline will be detected. At this stage we give them a *userState* value of 0x00000000.

If the user moves out of the sensor's field of view, they will generally stay registered as a user for five seconds or so (in case they step back into the field of view) and then disappear from the data. After this point, their user id may be recycled and used again by a future user. You should be guaranteed at least one frame (probably several seconds) where the identifier does not feature in the data before it is used again.

If the user adopts the calibration pose (arms up), they will switch to the calibrating state, which is when *userState* is equal to 0x00000001. At this point the sensor is trying to calibrate their dimensions. If this fails or the user adopts a different pose, they may return to state 0x00000000. If this succeeds (typically with a second or two), they will switch to the calibrated state, with *userState* equal to 0x00000002. In this state (and only in this state), their joint position information will be sent. See the next section on joints.

6 Joints

When the user is in the calibrated state, information will be sent for all of their 15 joints. The joints are (starting at index 0):

0	Head
1	Neck
2	Torso
3	Left shoulder
4	Left elbow
5	Left hand
6	Right shoulder
7	Right elbow
8	Right hand
9	Left hip
10	Left knee
11	Left foot
12	Right hip
13	Right knee
14	Right foot

Note that the neck seems to always be the midpoint of the two shoulders, and the torso always seems to be the midpoint of the two shoulders and two hips, so really only 13 joints are being tracked properly (3 per limb, and 1 for the head). But we send all the data anyway.

Each joint has an associated confidence rating. This will be low if, for example, that joint is out of view of the camera (because it's outside the field of view, or something else is in front of it, such as furniture or other users). It has the range 0 to 1. Each joint also has an associated world position and screen position (with Z being depth). Generally, you will want to use screen position for things like games, and the world position for things like 3D modelling. The screen position X and Y will always be in the 0–640 and 0–480 range respectively, regardless of what image size was requested by the client.

7 Writing a Client

When writing a client, you will often have your client code in some function called each frame (e.g. an idle event in a GUI or GLUT program). Initially it may seem best to write the function like this:

```
void eachFrame()
{
  sendRequestToServer(socket);
  waitForResponseFromServer(socket, &respData);
  processData(&respData);
}
```

The problem with this is that there will be a delay between the request and response while the server builds the data packet and streams it over the socket. A better model is this:

```
void eachFrame()
{
  waitForResponseFromServer(socket, &respData);
  processData(&respData);
  sendRequestToServer(socket);
}
```

You must also add a call to `sendRequestToServer(socket);` when you initially connect to the server or else there will not be a response at first. This way there is time between calls to the function for the server to prepare and send the data ready to be read next time the function is called.

Note also that a lot of data is sent each frame; the biggest part of the data is the RGB image, which will be just over a megabyte if you use a full-sized image. Increasing the size of your socket's receive buffer to one or two megabytes may help performance.

If you are connecting over a network to the server, note that keeping up with the full 30FPS that the Kinect can manage will therefore require a server to client throughput of over 30MB/s for full-sized image data. This requires a 1Gbps network connection to the server; if you haven't got this, consider using a smaller image size. Switching to a 320×240 image will roughly quarter the amount of data sent, which means it may manage on a 100Mbps network. If you are connecting to a server on the same machine, this should not be a consideration.