kinectTCP

Windows7 TCP/IP server

Documentation V1.1

Rolf Lakaemper, Temple University, September 2011
email: lakamper@temple.edu

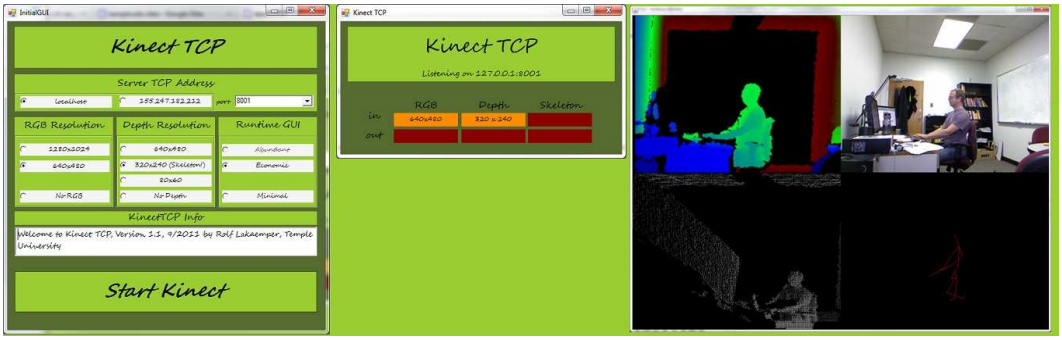
Overview

kinectTCP is a TCP/IP server that offers all video, depth and skeleton services of Microsoft's Win7 Kinect SDK. kinectTCP allows simple access to Kinect data via TCP/IP, independent from specific programming languages. It therefore allows programmers, who do not want to deal with windows specifics of libraries and languages, to utilize the Kinect using their language of choice, e.g. JAVA, BASIC, MATLAB, PYTHON, FLASH — any language that is able to build a small TCP/IP client.

kinectTCP frees the programmer from the necessity of using a Windows.Net environment.

The KinectServer offers raw RGB video data in all available resolutions, depth and player ID data in all resolutions, and the entire set of skeleton data (joints, floor plane etc.). For convenience, the server can additionally send depth data as XYZ point cloud, i.e. depth data as point cloud in a 3D volume, as well as depth data (both, depth and depth-XYZ) with corresponding RGB color information.

Resolution and modes of depth and video data can be initialized either remotely from the client side, or manually via a GUI on the server side. The KinectServer supports **multiple clients**, allowing to share data from a single device with multiple programs. The performance of the Kinect Server is depending on the TCP connection speed. If used on a single machine via localhost, the performance does not significantly decrease compared to direct Kinect SDK programming. A potentially slow connection speed of one client does not influence the speed of faster connected clients, all data is buffered per client before transmission.



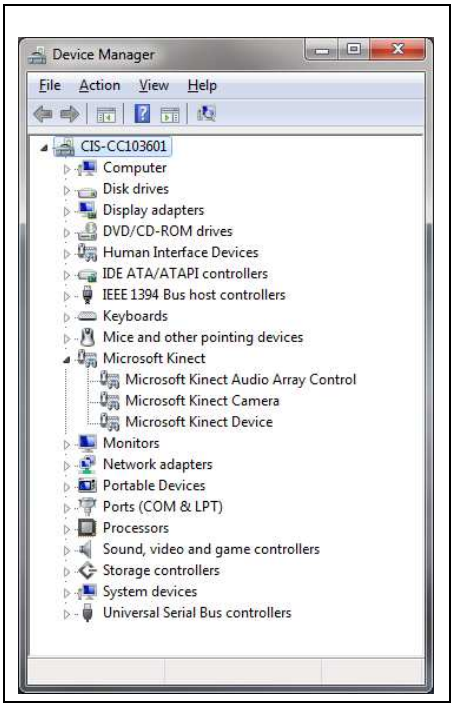
Left: Initial GUI, center: runtime GUI, right: JAVA client using depth, rgb, depth point-cloud and skeleton

Installing the Kinect Drivers

kinectTCP is based on the Win7 Kinect SDK. This has to be installed first. The installment is simple, just download the package from

http://research.microsoft.com/en-us/um/redmond/projects/kinectsdk/

Run the installer, ready.
The Kinect device should now be visible in the device manager, the green LED on the device should also blink. To open the device manager, click the W7 button on the desktop, type "device manager" in the "search programs and files" field. This opens the device manager. The Kinect should be listed.
In case kinectTCP does not work, you might also have to install the Visual Studio 2010 Enterprise environment, which is free.



Starting kinectTCP

TCP address and Server Connection (V1.1)

The Server starts immediately when the program is started, allowing for remote Kinect initialization. The default IP address is 127.0.0.1 (localhost). This setting makes the server visible only to clients that run on the same machine. To access the server from a different machine, select the button with the local physical (internal) IP address. Please note that in this case, connection via "localhost" does NOT work, even if the client runs on the same machine; clients must connect via the specific TC P address.
If no connection to the server can be established, client and server are most likely running on different sub-nets or behind a firewall. In this case, the client cannot see the internal TCP address, connection is not possible.

About TCP addresses:

- choose localhost if you want to run your program on the same computer as the kinect server. The port-selection does not really matter in this setting. Localhost is the fastest connection you can get. It is also the safest way of *not* involuntarily sharing data: the kinect data is not visible to the outside world.
- choose your (internal) IP address as the server address, if the kinect should be visible to anyone who can ping that address. If you are not in a local net, your data is visible world-wide then. The safest port to assure not to be blocked by

firewalls is port 80. However, this port is the official http port, so it should not be used if not necessary. Choose one of the other ports offered by kinectTCP first.

Manual Kinect Initialization via GUI

kinectTCP opens with a GUI offering a selection of different modes and resolutions, see figure 1. The Kinect devices are initialized after the "Start Kinect" button is pressed.

Remote Kinect Initialization via Client

The TCP server runs immediately when kinectTCP is started, enabling an alternative remote initialization of the Kinect device (see commands: initialization commands). As soon as kinectTCP receives a valid initialization command, the client takes over, i.e. the selection GUI is closed and replaced by the runtime GUI.

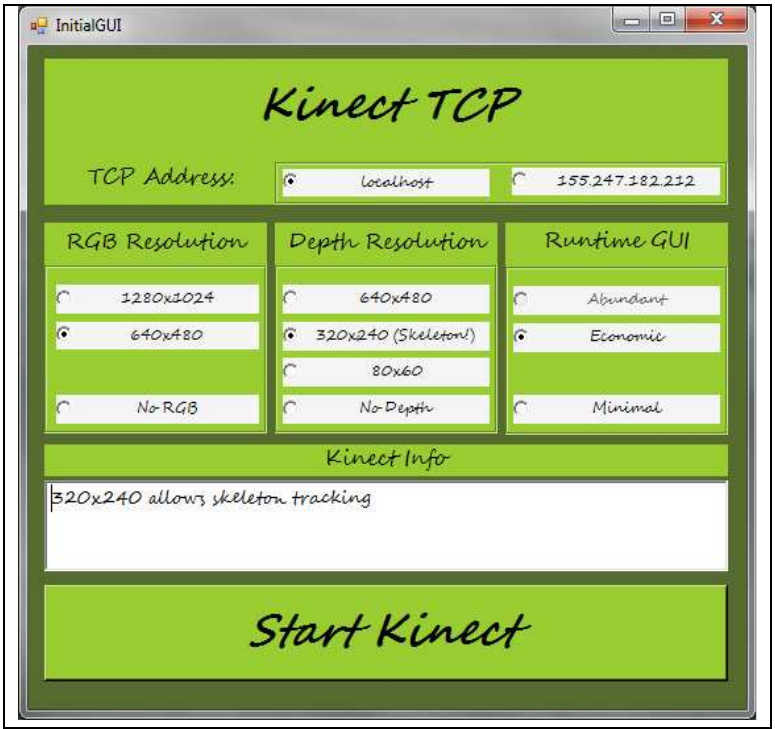


Figure 1: kinectTCP, selection GUI

General

After initialization, the Kinect devices will start when the server receives one of the 'startXXX' commands (or the 'Start' button is pressed). The server can be run in 'economy' or 'abundant' mode. The mode determines the GUI after the Kinect devices were started. While 'no GUI' will minimize the server (no output), 'economic' reports basic informations; 'abundant' visualizes all initialized streams (which is advantageous for debugging on the client side).

The first client sending a start command determines the server modes, no further change is possible.

Runtime Display 'economic'

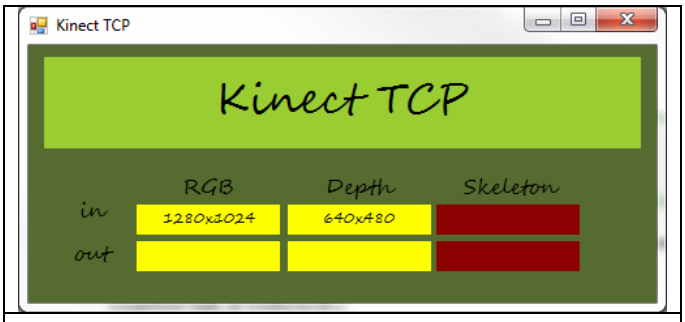


Figure 2:runtime GUI, economic

This mode shows connectivity to Kinect and client(s), see figure 2. Elements of the colored label array stay red when no connection to the respective stream is established, otherwise they change colors each 5th time data is received/sent. The 'in' row reacts to data from the Kinect, it shows incoming data independent of connected clients. For example, if RGB 640x480 mode is selected, the Kinect should send data with a frequency of 30fps, i.e. the RGB in label should blink with a frequency of 3Hz (change of color each 5 images = 6 yellow/orange switches = 3 Hz).

The 'out' labels react to data sent to the client, they switch color every 5th received command.

Multiple Clients

The server allows for multiple clients to connect. It often makes sense for a single program to open different connections for RGB and DEPTH, since the outgoing data frequency naturally varies with the amount of data (e.g. higher resolution slows down the respective stream from the server). If a single client waits for data from the server, and it is programmed as a blocking (waiting) client, all data will be received in the slowest frequency. Opening three clients in parallel (one for RGB, one for depth, one for skeleton) is sometimes useful.

In certain cases, it can be observed that the out labels blink faster than the in labels. This is the case when the TCP connection is faster than then Kinect itself (e.g. RGB 1280x1024 vs. localhost TCP). In this case, the TCP server does not wait for new data from the Kinect, but might transmit the same data repeatedly.

Communication

Commands to the server are single byte commands:

# bytes	Content
1	Command

Table 1: Command format

The server answers either with a single byte or a byte stream of the following format ('long format'):

# bytes	Content
4	Message length n+4 (length of all bytes to follow)
4	Timestamp
N	Data

Table 2: Response, long format

All bytes are unsigned (all transmitted data is positive), multi-byte values are transmitted least significant byte first. Byte format was preferred over the more convenient ASCII format due to the high amount of data (e.g. RGB 1280*1024: 1310720 pixels, currently encoded in 24bit color (~8MB)). All data is uncompressed.

Note for JAVA programmers: JAVA does not support unsigned byte. Unfortunately, the only way to deal with byte data in JAVA is to copy the byte array to an integer array, and to transform negative values v (if (v<0) v=256+v).

Currently, only data polling is supported, i.e. data must be requested from the server.

Note: internally, the server runs event based for precise skeleton tracking. The internal buffers are updated each time the Kinect sends a frameReady event. They are copied to the client buffers on request by the corresponding command for transmission.

Important: before the server receives a startKinectXXX command (commands 2,3), it will not respond to any readXXX (30-35, 50-55,...) commands. This means, blocking commands on the client side will cause the client to hang.

The following table enumerates the server commands (sent from the client to the server) Response formats are

- S: single byte response
- L: long format response, see table 2

ID	Command	Explanation	Response Format
General Server Commands			
0	Hello	Convenience test command for TCP connection. Server answers with value '214' (=> if the client interprets this as '-42', the signed/unsigned translation failed on the client side)	S
1	numDevices	Number of available Kinect devices.	S
2	startKinect	Start all Kinect Devices. Non-initialized modes are not available	S
3	startKinectDefault	Start all Kinect devices with RGB640x320 and Depth 640x320.	S
Kinect Device 1 Commands (starting at 20 = Device number * 20)			
20	initRGB640x480	Initialize RGB mode 640x480	S
21	initRGB1280x1024	Initialize RGB mode 1280x1024	S
22	getRGBMode	Current Video Mode 0 = noRGB, 3 = 640x480, 4 = 1280x1024	S
23	initDepth80x60	Initialize Depth mode 80x60	S
24	initDepth320x240	Initialize Depth mode 320x240	S
25	initDepth640x480	Initialize Depth mode 640x480	S
26	getDepthMode	Current Depth Mode 0 = noRGB, 1 = 80x60, 2= 320x240, 3=640x480	S
27	noRGB	Kinect will not deliver RGB	S
28	noDepth	Kinect will not deliver Depth / Skeleton	S
30	readRGB	Read video frame	L
31	readDepth	Read depth frame (depth information only)	L
32	readDepthXYZ	Read depth frame (as 3D point cloud)	L
33	readDepthRGB	Read depth frame and corresponding color information	L
34	readDepthXYZRGB	Read depth frame as point cloud with corresponding color information	L
35	readSkeleton	Read skeleton information	L
Kinect Device n repeats the commands above, starting at n*20			
40	initRGB640x480	Initialize RGB mode 640x480 <default>	S
…	…	…	…

Table 3: Server Commands

Details:

- After starting the devices, no mode change is possible
- For commands 2,20,21,23,24,25,(40,41,43,…) the single byte answer is '1' on success, '0' otherwise.

Long Format Responses

Command 30: **readRGB**

Stream of color pixels, rows top to bottom, left to right. Color is RGB encoded, 1 byte per color, order R,G,B.

1	2	3
R	G	B

Data size n: 640x480: 921600 bytes
1280x1024: 3932160 bytes

Command 31: **readDepth**

Stream of distances, rows top to bottom, left to right. Distance is the (positive) distance to the sensor plane in mm. Additionally, the depth information contains the player ID encoded: if a pixel is recognized as representing a player, the ID is set accordingly (background pixels have ID '0'). The ID is encoded in the 3 most significant bits of the high byte. The distance is encoded in the first 13 bits of low and high byte. Distance is computed as:

```
int distance = lowbyte + (highbyte & 0x1f) * 256;
```

Data size n: 80x60: 9600 bytes
320x240: 153600 bytes
640x480: 614400 bytes

Command 32: **readDepthXYZ**

Stream of 3D volume pixel coordinates, rows top to bottom, left to right.
The coordinate system's XY origin lies in the center of the view direction (i.e. in the 'middle of the picture'), Z=0 denotes the sensor plane. Units are in mm. Since X and Y can be negative, but the transmit data format supports only positive values, an offset of 32768 is added to all coordinates (X, Y and Z). All values are transmitted as 2 bytes, least significant byte first.

1	2	3	4	5	6
X low	X high	Y low	Y high	Z low	Z high

Example (depthXYZ is the received byte stream (note, again, for JAVA programmers: first transform to positive integers!): to read X,Y,Z for pixel column c, row r:

```
index = r*sizeX*6 + c*6;  // sizeX: either 80, 320 or 640
x = depthXYZ[index]+256*depthXYZ[index+1] – 32768;
y = depthXYZ[index+2]+256*depthXYZ [index+3] – 32768;
z = depthXYZ[index+4]+256*depthXYZ [index+5] – 32768;
```

Data size n: 80x60: 28800 bytes
320x240: 460800 bytes
640x480: 1843200 bytes

Command 33: **readDepthRGB**

Like Command 31 (readDepth), but with additional RGB information for each pixel. For each pixel, the 2 byte depth information is followed by 3 bytes RGB, hence each pixel is transmitted using 2+3 = 5 bytes. As in command 31, depth contains player ID

1	2	3	4	5
Depth low Bits 0-2: playerID, Bits 3-5: depth low	Depth high in bits 0-5	R	G	B

Data size n: 80x60: 24000 bytes
320x240: 384000 bytes
640x480: 1536000 bytes

Command 34: **readDepthXYZRGB**

Like Command 32 (readDepthXYZ), but with additional RGB information for each pixel. For each pixel, the 6 byte depthXYZ information is followed by 3 bytes RGB, hence each pixel is transmitted using 6+3 = 9 bytes.

1	2	3	4	5	6	7	8	9
X low	X high	Y low	Y high	Z low	Z high	R	G	B

Data size n: 80x60: 43200 bytes
320x240: 691200 bytes
640x480: 2764800 bytes

Command 35: **readSkeleton**

Stream of Skeleton related data, includes tracking state, floor plane and joint data.

#bytes	Data
1	Number of skeletons. Max: 6
8	Floor clip plane. 2 bytes each for X,Y,Z,W, least significant byte first.
8	Normal to Gravity. 2 bytes each for X,Y,Z,W, least significant byte first.

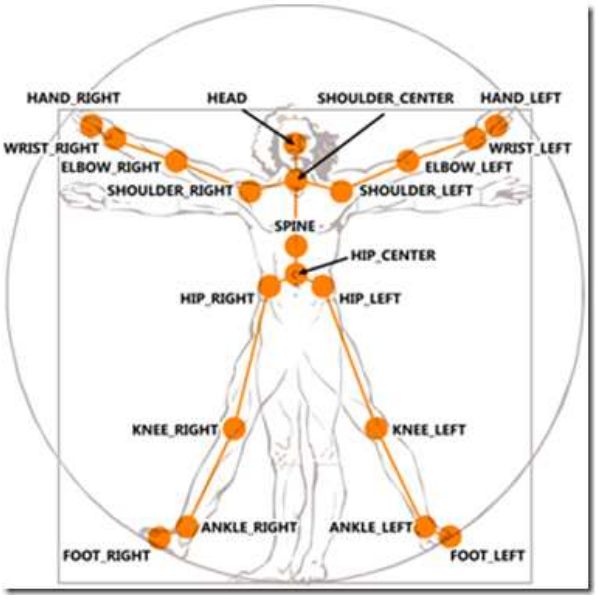
For each Skeleton (Skeletons 1..numberOfSkeletons)		
1	ID	
1	State. 0 = passive, 1=active.	
1	Quality Flags	
6	Center of Mass. 2 bytes each for X,Y,Z	
	For each joint (ALWAYS 20 joints per skeleton)	
	1	State. 0=not tracked, 1=inferred, 2=tracked
	2	X
	2	Y
	2	Z
(9 + 20*7 = 149 bytes per skeleton)		

Joint positions are in mm, an offset of 32768 is added. The coordinate space has its origin in the center of the depth sensor.

Data size n: depends on numbers of tracked skeletons: `17 + #skeletons*149`

Joint Index System:

0. HipCenter
1. Spine
2. ShoulderCenter
3. Head
4. ShoulderLeft
5. ElbowLeft
6. WristLeft
7. HandLeft
8. ShoulderRight
9. ElbowRight
10. WristRight
11. HandRight
12. HipLeft
13. KneeLeft
14. AnkleLeft
15. FootLeft
16. HipRight
17. KneeRight
18. AnkleRight
19. FootRight



Communication Examples

Example 1: kinectTCP manually set to RGB 640x480, depth 320x240, and started

Client	Direction	Server	Explanation
0	→		"hello" (not required)
	←	214	
30	→		readRGB
	←	Long Format Response (921608 bytes)	4 byte message length, 4 byte timestamp 921600 bytes RGB data
31	→		readDepth
	←	Long Format Response (614408 bytes)	4 byte message length, 4 byte timestamp 614400 bytes RGB data

Known Issues (TODO List)

- depth 80x60 seems to have issues. I'm not sure if it's a client or server side problem (tested with java client)
- Abundant runtime mode is not implemented error free, hence it is disabled.
- readDepthXYZRGB is not implemented
- server exits only if client terminates, too (server window does only hide)
- Supports single kinect only

Version Changes

V1.0

- new initial GUI
- client init/start commands are implemented (except for GUI init)
- new commands noRGB, noDepth are added
- getMode modes values changed in protocol
- getMode implemented

V1.1

- TCP/IP selectable localhost/physical local IP address for external visibility

Contact Info:

Rolf Lakaemper,
Dept. of Computer Science
Temple University, Philadelphia, PA
lakamper@temple.edu