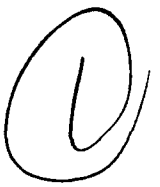

The DataCube Server

Roger E. Kahn, Michael J. Swain, and R. James Firby

University of Chicago
Animate Agent Project Working Note 2
Version 1.0, November 1993

For additional copies, write to:

Department of Computer Science
University of Chicago
1100 E. 58th Street
Chicago, Illinois 60637-1504
U.S.A.

The DataCube Server

Roger E. Kahn
University of Chicago
Department of Computer Science
kahn@cs.uchicago.edu

Michael J. Swain
University of Chicago
Department of Computer Science
swain@cs.uchicago.edu

R. James Firby
University of Chicago
Department of Computer Science
firby@cs.uchicago.edu

November 1993

Contents

Chapter 1

Using the DataCube Client Class

Intro

The DataCube server and client class facilitate the development of real-time vision systems by allowing you to write distributed programs that can access the DataCube MV200/MV20 and DigiColor boards in parallel. The DataCube server is a network interface to a library of Pipelined routines that run of the MV200/MV20 and DigiColor. The functions provided by the interface include frame grabbing, Gaussian pyramids, color histogram backprojection, color histogram intersection, motion detection from a moving platform, foveated frame display, and optionally, control of a Directed Perception pan-tilt head. Furthermore, the system is designed to be extendable so that new functions can be added to the server.

The DataCube client class is a C++ class that provides a function based application programmatic interface to the DataCube server. To use the DataCube client class one does not need to be familiar with how the DataCube is programmed. The programmers interface is simply a C++ class with a set of member functions. All that one does is call the member functions; the networking and pipelined processing happen behind the scenes.

This system has been designed as a programming interface to a robots visual system. Consequentially, it must run quickly to provide the real-time functionality required by a robot. All of the DataCube routines have been compiled into a special format called a PAT (PipeOp Altering Thread) that minimizes the overhead required to run them. Because of this all of the pipelined routines run as quickly as possible. To further improve efficiency the systems routines are based on a Gaussian pyramid. The programmer can specify an area of interest for each level in the pyramid (called an active region) and restrict processing to

only be done on that area. When a call to the DataCube returns a processed image, only the active region at the requested levels of resolution are returned, thus reducing the load on the network.

Why not just program the DataCube directly?

We have found the three most limiting aspects of the DataCube machines to be:

- Any process using the DataCube must be running on a single designated DataCube host machine.
- Only a single process can use the DataCube at a time.
- Pipelined programs are hard to program.

The DataCube server alleviates all three of these problems. Processes connect to the DataCube server via TCP-IP. Only the DataCube server must be running on the DataCube host. Clients can run on any machine that has a network connection with the host. This can be especially useful when either the fastest available machine is not the DataCube host, or when a machine other than the datacube host has special hardware that must be used.

The DataCube server allows multiple concurrent connections so that many routines can use the DataCube simultaneously. It is often the case that off-DataCube processing is required in addition to the processing done on the DataCube (especially in the development stages). By providing multiple processes with access to the DataCube, non-pipelined processing can be done in parallel. Furthermore, since clients can connect from different machines non-pipelined processing can be distributed to many microprocessors.

The DataCube becomes easier to program in three ways. Firstly, most routines will not need to have any DataCube code written at all; the server will provide all the functions needed. Secondly, when DataCube code must be written a C++ application programmatic interface to imageflow is provided that: sets up commonly used paths (like from the camera to a DataCube memory), automates memory allocation, and keeps track of volitile elements on the DataCube that a pipe must reset only when some other pipe has corrupted them (such as look up tables). And thirdly, a standard structure is provided that simplifies the integration of multiple pipelined routines into a single program; thus allowing multiple routines to use each others results and allowing a single application (such as the datacube server) to use many routines.

Hardware

Host

The host to the DataCube is the only machine that can directly talk to it. This will be the machine on which the DataCube server runs and that the pan-tilt head, if present, is connected to. For faster communications a client can also be run on the host and the network time lag is reduced, this will be particularly useful if the host is a sparc-10 with multiple CPU's once DataCube supports Solaris.

DataCube

The server expects MV200/20 and DigiColor boards to be present with image-flow installed on the system. The connection between these boards should be compatible with the following configuration file:

```
SPCVRate: T60HZ
IRQLevel: 5
MaxEvents: 64
MaxPats: 48
PatMemSize: 786432

DEV: ab00
Base: 0x10000000
AddrSpace: ADDR_A32
IRQVector: 0xE4
SPCRole: SPC_SLAVE
SPCTerminate: TRUE

DEV: dc00
Base: 0x80000000
AddrSpace: ADDR_A32
IRQVector: 0xD8
SPCRole: SPC_HVMASTER
SPCTerminate: FALSE

# DIGICOLOR's output MV200/MV20
CABLE: (dc00.P4, ab00.P7)
CABLE: (dc00.P9, ab00.P9)
CABLE: (dc00.P10, ab00.P10)

# MV200/MV20's output to DIGICOLOR
CABLE: (ab00.P5, dc00.P5)
```

CABLE: (ab00.P6, dc00.P6)
CABLE: (ab00.P8, dc00.P8)

Cameras should be connected to the DigiColor composite ports vid0 and/or vid1.

Directed Perception Head

The Directed Perception pan-tilt unit (model PTU) is supported by the server. All commands that can be sent to this head are available from the client. Angles can be specified in degrees, radians, grads, pixels, and stepper motor counts. See the documentation on the class interface to the pan-tilt unit for further details. This head does not have to be present to use the server.

Khoros

The DataCube server provides a class for operating on images that is based on the Khoros Visualization and Image File Format (VIFF) (version 1.5). Since this image class is bassed on the VIFF many of the Khoros library functions are available. Khoros provides functions in the following categories: arithmetic, classification, color conversion, data conversion, file format conversion, feature extraction, frequency filtering, spatial filtering, morphology filtering, geometric manipulation, histogram manipulation, statistics, signal generation, linear operations, segmentation, spectral estimation, subregion, and transforms.

Khoros is available via anonymous ftp from pprg.eece.unm.edu, 129.24.24.10, in the /pub/khoros directory. Khoros has a newsgroup, comp.soft-sys.khoros that is quite active and helpful.

create a directory for Khoros
setenv $KHOROS_HOME to point to the new directory
cd $KHOROS_HOME
tar xvf Khoros.basic.tar

Installation

The following procedure should be followed for installation. It is assumed that the environmental variable $KHOROS_HOME points to the directory where Khoros is installed (if Khoros is not installed see the above section on Khoros) and that $DQHOME points to the directory where Imageflow is installed. Directories are created in $KHOROS_HOME and $DQHOME to store the C++ headers for these libraries, hence write permission should be available in both those directories. A new library will be written to $KHOROS_HOME/lib so

write permission will be needed for this directory as well. A directory should be chosen to install the server in and the variable $DQ_SERVER_HOME set to point to it. Write permission must be available for $DQ_SERVER_HOME. The environmental variable CC must be set to your C++ compiler and any flags required by the compiler should be set in the CC variable (for example if using lucid C++ setenv CC "lcc -XF"). The installation steps are:

```
cp DQ_Server.tar.Z $DQ_SERVER_HOME
cp DQ_Server_doc.tar.Z $DQ_SERVER_HOME
cp khoros_include_diff.tar.Z $DQ_SERVER_HOME
cp DQ_SERVER_INSTALL $DQ_SERVER_HOME
cd $DQ_SERVER_HOME
chmod u+x DQ_SERVER_INSTALL
DQ_SERVER_INSTALL
```

To run the server, log into the DataCube host machine and go into the $DQ_SERVER_HOME/bin directory. Run dq_server with no command line arguments or run dq_server.bin with the arguments: <log-file> <init-port> <DQ-config-file> <PTH-port>.

- <log-file> is a file to log errors and commands into.
- <init-port> is the socket port that clients will connect to the server at.
- <DQ-config-file> is the system configuration file used by imageflow, usually $DQHOME/text/dq/dqsys.cfg.
- <PTH-port> is the port for the pan tilt head, usually /dev/ttya if the head exists and /dev/null if it doesn't.

These four parameters are optional, with default values of "log", "6123", "$DQCONFIG", and "/dev/null" respectively. Note <log-file> can be used to log all client requests, if it is used to do this it can grow rather large and should be deleted occasionally.

A quick test

To test your setup start the DataCube server as described in the previous section. If the server does not start check to make sure that your MV200/MV20 and DigiColor boards are connected as shown in the above example imageflow config file. While the server is running log into any other machine and go to the $DQ_SERVER_HOME/bin directory. Run the client test program dq_client_test <DataCube-host-name> <init-port>. If the test program does not connect to the server it may be because the server is not finished initializing, let the server run until it says that it is ready to accept a client and try connecting again. Once

a connection is established type "StartDisplay" to begin the display pipe. After the display pipe is running type "GrabMultiResFrameAndDisply" to grab a frame and display it. Next change the number of levels in the pyramid with "SetNumLevels 4" and foveate on the center of the screen by entering "Foveate 250 250". Grab and display another frame with "GrabMultiResFrameAndDisply". A foveated image should appear.

The Client Class

Using the client class

The client class offers many functions that can be used to do image processing and control the pan-tilt head (if present). These functions are described in the "Members of the client class" section. The client is declared in the file "client.H" as a global variable DQclient of type client (this is similar to cout begin declared as a global in iostream.h). Communication with the server should be done through this variable, for example:

```
DQclient.StartDisplay ();
DQclient.GrabHighResFrameAndDisplay (); //Grab a frame from the camera
```

Compiling clients

The following directory structure and variables are assumed:

$DQ_SERVER_HOME/include:
contains all include files for the client and server classes.
$DQ_SERVER_HOME/lib:
contains all libraries for the client and server classes.
$DQHOME
root directory for Imageflow.
$KHOROS_HOME
root directory for Khoros.

To compile a client your application should include client.H. The directories $DQ_SERVER_HOME/include and $KHOROS_HOME/include_C++ should be in the include path. It should link with libclient.a, libcomm.a libxv_class.a from $DQ_SERVER_HOME/lib and libvipl.a, libvutils.a, libvmath.a, libvgparm.a, libverror.a, and libUofC_Khoros.a from $KHOROS_HOME/lib.

Members of the Client Class

Gaussian pyramid commands

All of the functions in this class are based on the gaussian pyramid. In particular, active regions in each level of the gaussian pyramid can be specified. When an active region is set most functions will operate in that regions only.

- **void** SetCamera (**const int** cam)
- **void** DQS_SetCamera (**const int** cam)

  Select a camera for input. Cameras are numbered 0-5 where the camera number corresponds to the composite connection to the DigiColor on the signal cable box (VID0-VID5). Limitations: Currently only camera 0 or camera 1 may be requested and motion detection only is supported on camera 0.

- **void** SetColorMode (**const ColorMode** _cmColorMode)
- **void** DQS_SetColorMode (**const ColorMode** _cmColorMode)

  Specify the current color mode. The color modes are cmGREY (greyscale), and cmRGB. This will influence the type of frame grabbed from a camera and how frames are displayed.

- **ColorMode** cmGetColorMode ()
- **int** DQS_GetColorMode ()

  Return the current color mode used for acquistion, and display. If an error occurs or the color mode is illegal return cmError.

- **void** SetNumLevels (**const int** _NumLevels)
- **void** DQS_SetNumLevels (**const int** _NumLevels)

  This specifies the number of levels in the multiresolution pyramid that are computed. There can be at most 6 levels in the pyramid, ther must be at least one level. If _NumLevels specified is more than 6 or less than 1 then this function returns false and the number of levels is not changed, otherwise it returns true.

- **int** GetNumLevels ()
- **int** DQS_GetNumLevels ()

  Return the number of levels being created by the multiresolution pyramid.

- **void** SetActiveRegion (**const int** level, **const int** LowX, **const int** LowY, **const int** HighX, **const int** HighY)*

- **void** DQS_SetActiveRegion (**const int** level, **const int** LowX,
  const int LowY, **const int** HighX,
  const int HighY)

  Specify the active region of a level in the pyramid. Level reads and writes occur only in the active region. Also color histograms are only computed in the active region. When the multiresolution image is displayed only the active regions appear on the screen. The active region is specified by the upper left and lower right corners of a rectangle.

- **void** GetActiveRegion (**const int** level, **int&** LowX, **int&** LowY,
  int& HighX, **int&** HighY)

- **void** DQS_GetActiveRegion (**const int** level, **int*** LowX,
  int* LowY, **int*** HighX,
  int* HighY)

  Return the active region of a level in the pyramid. Level reads and writes occur only in the active region. Also color histograms are only computed in the active region. When the multiresolution image is displayed only the active regions appear on the screen. The structure **DqRect** contains 4 **long** fields, lXMin, lYMin, lXMax, and lYMax. These fields provide the upper left and lower right corners of a rectangle.

- **void** GrabHighResFrame ()

- **void** DQS_GrabHighResFrame ()

  Read a frame from the camera. The color mode of the frame is determined by SetColorMode.

- **void** DisplayHighResFrame ()

- **void** DQS_DisplayHighResFrame ()

  Display the last frame grabbed in high resolution. The color mode of the frame is determined by SetColorMode.

- **int** ReadImage (**xv<unsigned char**, 1>& xvc1Frame)

- **int** DQS_ReadImage1 (**struct xvimage*** pxvc1Frame)

  Read the last grey scale high resolution frame grabbed into an xv structure. The current color mode should be cmGREY when this is called.

- **int** ReadImage (**xv<unsigned char**, 3>& xvc3Frame)

- **int** DQS_ReadImage3 (**struct xvimage*** pxvc3Frame)

  Read the last 3 band high resolution frame grabbed to an xv structure. The current color mode should be cmRGB when this is used.

- **int** WriteImage (**const xv<unsigned char**, 1>& xvc1Frame)
- **int** DQS_WriteImage1 (**struct xvimage*** pxvc1Frame)

  Write a one band xv image into frame acquisition memory. Use this command to simulate input from a grey scale camera.
- **int** WriteImage (**const xv<unsigned char**, 3>& xvc3Frame)
- **int** DQS_WriteImage3 (**struct xvimage*** pxvc3Frame)

  Write a three band xv image into frame acquisition memory. Use this command to simulate input from a color camera.
- **int** ReadLevel (**xv<unsigned char**, 1>& xvc1Fovea, **const int** level)
- **int** ReadLevel1 (**struct xvimage*** pxvc1Fovea, **const int** level)

  Read a one band image from the current frames ActiveRegion. The current color mode should be cmGREY. If the color mode if not one band then return false and the information written into xvc1Frame in unspecified. Note that this function does not necessarily return the whole frame, it returns the active region.
- **int** ReadLevel (**xv<unsigned char**, 3>& xvc3Fovea, **const int** level)
- **int** ReadLevel3 (**struct xvimage*** pxvc3Fovea, **const int** level)

  Read a three band image from the current frame's ActiveRegion in the specified levels acquisition memory. The color mode of the acquisition memory should be cmRGB. If the current color mode is not three band then return false and the information written into xvc3Frame in unspecified. Note that this function does not necessarily return the whole frame, it returns the active region.
- **void** CreateMultiRes ()
- **void** DQS_CreateMultiRes ()

  Create a multiresolution pyramid from the current highres image. This is usually called after GrabHighResFrame or WriteImage. The color mode of the resulting multiresolution frame is determined by the current color mode.
- **void** GrabMultiResFrame ()
- **void** DQS_GrabMultiResFrame ()

  Grab a highresolution frame and create a multiresolution frame This is equivalent to calling GrabHighResFrame followed by a call to CreateMultiRes.

- **void** GrabMultiResFrameAndDisplay ()
- **void** DQS_GrabMultiResFrameAndDisplay ()

  Grab a multiresolution frame and project it onto a foveated display. this is equivalent to calling GrabMultiResFrame followed by a call to Project-MultiRes.
- **void** ProjectMultiRes ()
- **void** DQS_ProjectMultiRes ()

  Project the multiresolution image into display memory by successively expanding and copying each levels active region to display memory starting with the lowest level of resolution and ending at the high-res level, level 0. This results in a foveated image in the display memory. The projection is displayed until StopDisplay is called.
- **void** Foveate (**const int** x, **const int** y)
- **void** DQS_Foveate (**const int** x, **const int** y)

  Set the active regions of each level to form a foveated system around the coordinate (x, y). The number of levels in the pyramid is determined by the SetNumLevels command. Each levels active region is a rectangle where level l has a width of $1/2^l$ that of the entire screen and a height of $1/2^l$ of the entire screen. Levels active region is truncated at the edge of the screen. For example:

Color histogram commands

The standard procedure for using color histogram backprojection is: 1) Choose a model, histogram it and pass the histogram to the server with WriteModelHist. 2) Use GrabMultiResFrameAndDisplay to grab and display a multi-resolution frame. 3) Call MakeOneBandImage to convert the 24 bit color image into an 8 bit histogrammable image. 4) Call HistogramLevel to histogram any levels whose backprojection will be used. 5) Use BackProject to backproject the model onto the multires image. 6) Use BlurBackProject to smooth the backprojection. 7) Use ReadBlurredBackProjectLevel to read the blurred backprojection and find peaks in this image.

- **void** WriteModelHist (**const xvHist&** xvhModel)
- **void** DQS_WriteModelHist (int aModel)

  Write a color histogram to the DataCube to be used as a model in histogram backprojection. Each level of resolution is scaled so that its points are histogrammed appropriately for the area they cover. Therefore this histogram should be of an image in the same scale as the highest level of

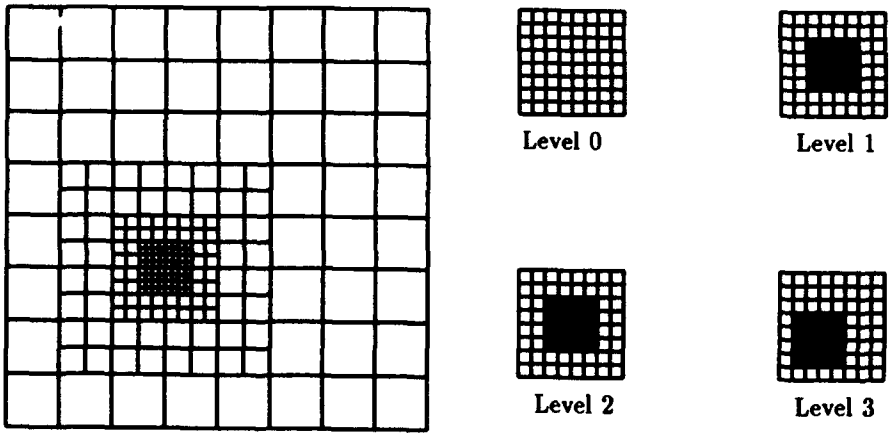

Figure 1.1: Foveated pyramid with 4 levels of resolution. The black regions show the areas covered by a higher level of resolution.

resolution no matter what level of resolution is being used for backprojection.

- **int** HistogramLevel (**xvHist&** xvh, **const int** level)
- **int** DQS_GetHistogramLevel (int ah, **const int** level)

  A single band image must be created before this is called by making a call to MakeOneBandImage. Create a histogram of the active region in a given level of resolution and return the histogram.
- **int** HistogramLevel (**const int** level)
- **int** DQS_HistogramLevel (**const int** level)

  A single band image must be created before this is called by making a call to MakeOneBandImage. Create a histogram of the active region in a given level of resolution. No histogram is returned.
- **void** MakeOneBandImage (**const xvHist&** xvh)
- **void** MakeOneBandImage (**Color2Grey** C2G)
- **void** MakeOneBandImage ()
- **void** DQS_MakeOneBandImageAndSetConversion (**Color2Grey** C2G)

- **void** DQS_MakeOneBandImage ()

  Create a single band image from a color image. Only the 5 high order bits are used in the conversion from each band, the 3 low order bits in each band are masked to zero. If an xvHist is passed then the xvHist's conversion function is used to convert pixels. If a Color2Grey is passes then it is used to convert pixels. If no argument is passed then the last function given is used to convert pixels. The function used to histogram should be constructed to histogram UNSIGNED images, i.e. 8 bit, not 7.

- **void** BackProject ()

- **void** DQS_BackProjectAll ()

  Backproject the current model onto each levels active region, before this can be called a histogram of all the needed levels must be made and their one band images must be made. Note if a levels backprojection will not be used then the level does not need to be histogrammed. Any points outside the active region in a levels backprojection will be undefined.

- **void** BackProject (**const int** Level)

- **void** DQS_BackProjectLevel (**const int** Level)

  Backproject the current model onto the levels active region, before this can be called a histogram of all the needed levels must be made and their one band images must be made. Note if a levels backprojection will not be used then the level does not need to be histogrammed. Any points outside the active region in a levels backprojection will be undefined.

- **void** BackProject (**const xvHist&** xvhRatioHist, **const int** Level)

- **void** DQS_BackProjectHist (**int*** int aRatioHist, **const int** Level)

  Backproject the given ratio histogram on to the given level. Any points outside the active region in a levels backprojection will be undefined.

- **void** DisplayBackProj (**const int** level)

- **void** DQS_DisplayBackProj (**const int** level)

  Display a levels active region of the backprojection. If the backprojection has been blurred then its blurred version is displayed. This active region is copied on top of any other image being displayed.

- **void** BlurBackProject ()

- **void** DQS_BlurBackProject ()

  Blur each levels backprojection with an 8×8 flat kernel. BackProject should be called prior to making this call.

- **double** Intersect (**const xvHist&** xvhModel, **const xvHist&** xvhImage)
- **double** DQS_Intersect (**int*** aModel, **int*** aImage)

  Compute the color histogram intersection between two histograms.
- **int** ReadOneBandLevel (**xv<unsigned char**, 1>& xvc1Level, **const int** level)
- **int** DQS_ReadOneBandLevel (**struct xvimage*** xvc1Level, **const int** level)

  Read the one band image of a given levels active region. MakeOneBandLevel should be called before making this call.
- **int** ReadBackProjectLevel (**xv<unsigned char**, 1>& xvc1BP, **const int** level)
- **int** DQS_ReadBackProjectLevel (**struct xvimage*** xvc1BP, **const int** level)

  Read the unblurred backprojection of a given levels active region. BackProject should be called before making this call.
- **int** ReadBlurredBackProjectLevel (**xv<unsigned char**, 1>& xvc1CBP, **const int** level)
- **int** DQS_ReadBlurredBackProjectLevel (**struct xvimage*** xvc1CBP, **const int** level)

  Read the blurred backprojection of a giSetven levels active region. BlurBackProject should be called before making this call.

Motion detection from a moving platform

Motion can be detected by a call to DetectMotion. This will returns the centroid of motion. DisplayMotion will display all the moved pixels. If the individual moved pixels are needed ReadDivisionThreshold can be called. The algorithm can be tweaked with a call to SetThresholds.

- **void** SetThresholds (**const int** _grad_threshold, **const float** _motion_threshold)
- **void** DQS_SetThresholds (**const int** _grad_threshold, **const float** _motion_threshold)

  This function allows the user to specify how sensitive the algorithm is to intensity variations with _grad_threshold (default 12), and how much ego motion is expected with _motion_threshold (default 2.0).
- **int** DetectMotion (**int&** pixCount, **int&** CenterX, **int&** CenterY)

- **int** DQS_DetectMotion (**int**&*pixCount, **int*** CenterX, **int*** CenterY)

  Motion detection from a moving platform. The number of moving pixels and the centroid of motion is passed back. Both the number of moving pixels and the centroid are computed over the entire image, not the active region for level 1. This call automatically sets the color mode to cmGREY.

- **void** DisplayMotion ()

- **void** DQS_DisplayMotion ()

  Run a pipe that displays the frame that motion detection was run on with all moving pixels marked in white.

- **int** ReadIntensityDifferential (**xv<unsigned char**, 1>& xvc1I_t)

- **int** DQS_ReadIntensityDifferential (**struct xvimage*** xvc1I_t)

  Pass back the derivative of intensity with respect to time of the frame used for motion detection. This function should always return true. The active region of level 1 is returned.

- **int** ReadGradient (**struct xvimage*** xvc1Grad)

  Pass back the Gradient of the frame used for motion detection. This function should always return true. The active region of level 1 is returned.

- **int** ReadDivisionThreshold (**struct xvimage*** xvc1Division)

  Pass back the thresholded division image used for motion detection. This function should always return true. The thresholded division image has all pixel values between 0 except the ones that moved which have a value of 254. The active region of level 1 is returned.

Stereo

When two cameras are connected to the DigiColor through vid0 and vid1 stereo frames can be grabbed. The frames are not grabbed at precisely the same time. The two frames are incorporated into the gaussian pyramid structure. The cameras are distinguished by the enumerated type camera which has the enumerations Left (vid0) and Right (vid1). Below are the functions provided:

- **void** GrabStereoPair ()

  Read a stereo pair with the left camera connected to vid0 and the right camera connected to vid1.

- **void** CreateStereoMultiRes ()

  Create a Gaussian pyamid from the stereo pair.

- **void** ReadStereoLevel (**xv<unsigned char**, 1>& xvc1,
  const int level, **const camera** C);

  Read a level's active region of a stereo image.

Convolution

Convolutions can be computed over any level of the grey-scale pyramid, any level of color histogram backprojection, the motion image, or a user specified image. The kernel has byte coefficients and can be any rectangle with area at most 64 bytes. Convolution produces a 40 bit result, but only 8 bits of this result can be stored. After the convolution the 40 bit result can be shifted by any value from -38 to $+10$ where negative values are right shifts and positive ones shift left. After the shift the rightmost 8 bits are saved.

To support separable filters single pass and double pass convolutions are possible. Two convolution filters may be specified, one for pass one and one for pass two. Specifying both kernels and requesting a double pass convolution is similar to calling two consecutive single pass convolutions with the appropriate kernel for each pass and with the same shift amount on each pass.

To Specify the surface for convolution the following enumerated type is provided. Custom specifies that a custom image is to be provided with the Write-Convolution command. Grey specifies that a level of the greyscale Gaussian pyramid is to be used. BackProjection specifies that a level of the histogram backprojection pyramid is to be used. Motion specifies that the thresholded division surface from motion detection is to be used.

```
enum DQImage {
  Custom,
  Grey,
  BackProjection,
  Motion
}
```

Note: Convolutions destroy the original surface.

- **void** ReadConvolution (**const ConvSurf** co,
  xv<unsigned char, 1>& xvc1)

- **void** DQS_ReadConvolution (**const ConvSurf** co,
  struct xvimage* xvc1)

  Read a convolution from the server to the client. This is only for surface that do not require a level of resolution to be specified and do not require an image number to be specified. Currently this only includes the Motion surface.

- **void** ReadConvolution (**const ConvSurf** co, **const int** Specifier,
  xv<unsigned char, 1>& xvc1)

- **void** DQS_ReadConvolutionWithLevel(**const ConvSurf** co,
  const int Specifier,
  struct xvimage* xvc1)

  Read a convolutions from the server to the client. This is for surfaces that require a level of resolution or an image number to be specified. For Grey and BackProjection images Specifier should signify the level of resolution to be read. For Custom images it should specify the custom surface number. There may be up to NumCustomSurfs custom surfaces.

- **void** WriteConvolution (**const int** Specifier,
  const const xv<unsigned char, 1>& xvc1)

- **void** DQS_WriteConvolution (**const int** Specifier,
  const struct xvimage* pxvi)

  Write a custom surface to the server. Specifier gives the custom surface number. There may be up to NumCustomSurfs custom surfaces.

- **void** SpecSingleKernel (**const int** pass,
  const xv<unsigned char, 1>& xvc1Kernel)

- **void** DQS_SpecSingleKernel (**const int** pass,
  struct xvimage* xvc1Kernel)

  Specify a kernel to be used in convolutions. In support of double pass convolutions which pass this kernel is to be used for should be specified. Pass numbers are zero based, thus legal values are 0 and 1. The kernel can be any rectangle with dimensions smaller that 8×8 or it can be a horizontal 1-D kernel of size up to 1×64.

- **void** SpecSingleShift (**const int** shift)

- **void** DQS_SpecSingleShift (**const int** shift)

  Specify the number of bits to be shifted after convolution. The 8 least significant bits of the shifted result will be stored. Legal shift values range from -38 to $+10$.

- **void** ConvolveSingleKernel (**const int** NumTimes, **const ConvSurf** co)

- **void** DQS_ConvolveSingleKernel1 (**const int** NumTimes,
  const ConvSurf co)

  Perform the convolution. NumTimes specifies the number of convolutions to do, legal values are 1 and 2. If NumTimes is 1 then the kernel written as pass 0 with SpecSingleKernel is used. If NumTimes is 2

then first the kernel written as pass 0 with SpecSingleKernel is used, then the kernel for pass 1 is used. The surface to convolve is specified in co. This is only for surface that do not require a level of resolution to be specified and do not require an image number to be specified. Currently this only includes the Motion surface.

- **void** ConvolveSingleKernel (**const int** NumTimes, **const ConvSurf** co, **const int** Level)

- **void** DQS_ConvolveSingleKernel2 (**const int** NumTimes, **const ConvSurf** co, **const int** Level)

  Perform the convolution. NumTimes specifies the number of convolutions to do, legal values are 1 and 2. If NumTimes is 1 then the kernel written as pass 0 with SpecSingleKernel is used. If NumTimes is 2 then first the kernel written as pass 0 with SpecSingleKernel is used, then the kernel for pass 1 is used. The surface to convolve is specified in co. This is only for surface that require a level of resolution to be specified or require an image number to be specified. Currently this only includes the Motion surface.

- **void** Sobel (**const ConvSurf** co)

  Preform a sobel operation on a surface. The surface to convolve is specified in co. This is only for surface that do not require a level of resolution to be specified and do not require an image number to be specified. Currently this only includes the Motion surface.

- **void** Sobel (**const ConvSurf** co,**const int** Level)

  Preform a sobel operation on a surface. The surface to convolve is specified in co. This is only for surface that require a level of resolution to be specified or require an image number to be specified. Currently this only includes the Motion surface.

DAP commands

The following commands can be used to transfer images from the DataCube to the DAP. Images are specified as they are in convolution with the DQImage enumeration and a level. The enumerations are: Custom, Grey, BackProjection, and Motion. There are up to 4 levels of Custom images and the number of levels of Grey and Backprojection images is controlled with the SetNumLevels command. No levels are specified with the Motion image. Inorder for them to work correctly the cable between the DataCube and the DAP should be disconnected when the DataCube is powered up. After the DataCube host is finished booting the cable should be attached. A single image can be transfered to the

Image	Levels	Memory
Grey	even	3
Grey	odd	0
BackProjection	even	3
BackProjection	odd	0
Custom	0	4
Custom	1	2
Custom	2	1
Custom	3	0
Motion	-	1

DAP or two can be transfered simultaneously. Not all images can be transfered together. In particular, two images cannot be transfered simultaneously if they reside in the same memories. The following shows which memories each surface is stored in.

- **void** SpecDAPOutput1 (**const DQImage** c)
- **void** DQS_SpecDAPOutput1a (**const DQImage** c)

  Transmit an image to the DAP that requires no second parameter for specification, currently this includes only the motion image. If the image is transmitted into the upper left corned of a 512 × 484 rectangle. If the image is smaller than 512 × 484 then the region to the right and below of the image in the 512 × 484 rectangle is undefined. If the image is larger than 512 × 484 then only the upper left corner of it is transmitted. The image is received by the dap as the first image in the transmitted array, the second image in this array is undefined.

- **void** SpecDAPOutput1 (**const DQImage** c, **const int** Level)
- **void** DQS_SpecDAPOutput1b (**const DQImage** c, **const int** Level)

  Transmit an image to the DAP that requires a second parameter for specification, currently this includes the custom, greyscale, and backprojection images. If the image is transmitted into the upper left corned of a 512 × 484 rectangle. If the image is smaller than 512 × 484 then the region to the right and below of the image in the 512 × 484 rectangle is undefined. If the image is larger than 512 × 484 then only the upper left corner of it is transmitted. The image is received by the dap as the first image in the transmitted array, the second image in this array is undefined.

- **int** SpecDAPOutput2 (**const DQImage** c1, **const int** Level1, **const DQImage** c2)

- **int** DQS_SpecDAPOutput2a (**const DQImage** c1, **const int** Level1, **const DQImage** c2)

  Transmit two images to the DAP where the first requires a second parameter and the second does not. The first image can be the custom, greyscale, and backprojection images. The second image should be the motion image. If the image is transmitted into the upper left corned of a 512 × 484 rectangle. If the image is smaller than 512 × 484 then the region to the right and below of the image in the 512 × 484 rectangle is undefined. If the image is larger than 512 × 484 then only the upper left corner of it is transmitted. The first image is received by the dap as the first image in the transmitted array, the second image iis the second image in that array. True is returned if the transfer is successful, false if returned if the transfer fails because the images are incompatible for dual transfer.

- **int** SpecDAPOutput2 (**const DQImage** c1, **const DQImage** c2, **const int** Level2)

- **int** DQS_SpecDAPOutput2b (**const DQImage** c1, **const DQImage** c2, **const int** Level2)

  Transmit two images to the DAP where the first requires a second parameter and the second does not. The first image should be the motion image. The second image can be the custom, greyscale, and backprojection images. If the image is transmitted into the upper left corned of a 512 × 484 rectangle. If the image is smaller than 512 × 484 then the region to the right and below of the image in the 512 × 484 rectangle is undefined. If the image is larger than 512 × 484 then only the upper left corner of it is transmitted. The first image is received by the dap as the first image in the transmitted array, the second image iis the second image in that array. True is returned if the transfer is successful, false if returned if the transfer fails because the images are incompatible for dual transfer.

- **int** SpecDAPOutput2 (**const DQImage** c1, **const int** Level1, **const DQImage** c2, **const int** Level2)

- **int** DQS_SpecDAPOutput2c (**const DQImage** c1, **const int** Level1, **const DQImage** c2, **const int** Level2)

  Transmit two images to the DAP where both images require a second parameter. These can be the custom, greyscale, and backprojection images. If the image is transmitted into the upper left corned of a 512 × 484 rectangle. If the image is smaller than 512 × 484 then the region to the right and below of the image in the 512 × 484 rectangle is undefined. If the image is larger than 512 × 484 then only the upper left

corner of it is transmitted. The first image is received by the dap as the first image in the transmitted array, the second image iis the second image in that array. True is returned if the transfer is successful, false if returned if the transfer fails because the images are incompatible for dual transfer.

- **void** CompleteDAPTransfer ()

  Halt the transfer of an image to the DAP. After this function is called all data received by the DAP is undefined.

Imageflow gs commands

The following set of functions allow you to draw on the display overlay. They consist of all the Imageflow graphics commands plus two commands for drawing sets of data.

All C versions of these routines have the DQS_ prefix and take the same arguments.

The following Imageflow gs function provide graphics on the overlay to the display. For information on these functions see the Imageflow Reference Manual. Note that only the gsInq and gsAsk functions return information, all other functions return data is lost. It should be assumed that DQ_OK would be returned for all non-Askinq funtions.

- **void** gsSaveAttr (**GsAttr** *ptGsAttr)
- **void** gsRestoreAttr (**GsAttr** *ptGsAttr)
- **void** gsSpecRasterOp (**DqEnum** eVisibility, **DqEnum** eSrcType, **DqEnum** eDstType, **DqEnum** eDrawOp)
- **int** gsAskRasterOp (**DqEnum** *peVisibility, **DqEnum** *peSrcType, **DqEnum** *peDstType, **DqEnum** *peDrawOp)
- **void** gsSetDrawingOp (**DqEnum** eDrawOp)
- **DqEnum** gsInqDrawingOp ()
- **void** gsSetLinePattern (**int** iPattern)
- **int** gsInqLinePattern ()
- **void** gsSetLineColor (**DqQByte** qColor)
- **DqQByte** gsInqLineColor ()
- **void** gsSetMarkerType (**DqEnum** eMarkType)

- **DqEnum** gsInqMarkerType ()
- **void** gsSpecMarkerSize (**int** iXRad, **int** iYRad)
- **int** gsAskMarkerSize (**int** *piXRad, **int** *piYRad)
- **void** gsSetMarkerColor (**DqQByte** qColor)
- **int** gsInqMarkerColor ()
- **void** gsSetPerimVis (**DqQByte** qPerimVis)
- **DqBool** gsInqPerimVis ()
- **void** gsSetFillStyle (**DqEnum** eFillStyle)
- **DqEnum** gsInqFillStyle ()
- **void** gsSetFillColor (**DqQByte** qColor)
- **DqQByte** gsInqFillColor ()
- **void** gsSetPerimColor (**DqQByte** qColor)
- **int** gsInqPerimColor ()
- **void** gsSetFontIndex (**int** iIndex)
- **int** gsInqFontIndex ()
- **void** gsSetTextStep (**int** iTextStep)
- **int** gsInqTextStep ()
- **void** gsSetTextColor (**DqQByte** qColor)
- **DqQByte** gsInqTextColor ()
- **void** gsSetCharOrientation (**int** iOrientation)
- **int** gsInqCharOrientation ()
- **void** gsSpecTextAlign (**DqEnum** eXAlign, **DqEnum** eYAlign)
- **int** gsAskTextAlign (**DqEnum** *peXAlign, **DqEnum** *peYAlign)
- **void** gsPoint (**long** lX, **long** lY)
- **void** gsDot (**long** lX, **long** lY)
- **void** gsLine (**long** lX1, **long** lY1, **long** lX2, **long** lY2)
- **void** gsPqolyLine (**int** iNumPoints, **long** *plPointList)

- **void gsDisjointPolyLine (int iNumPoints, long *plPointList)**
- **void gsRectangle (long lX1, long lY1, long lX2, long lY2)**
- **void gsCircle (long lX, long lY,**
- **int iRad)**
- **void gsEllipse (long lX, long lY, int iXRad, int iYRad)**
- **void gsMarker (long lX, long lY)**
- **void gsSpecViewport (long lXMin, long lYMin,**
  long lXMax, long lYMax)
- **int gsAskViewport (long *plXMin, long *plYMin,**
  long *plXMax, long *plYMax)
- **void gsSpecClipRect (long lXMin, long lYMin,**
  long lXMax, long lYMax)
- **int gsAskClipRect (long *plXMin, long *plYMin,**
  long *plXMax, long *plYMax)
- **void gsSetClipSrc (int eClipSrc)**
- **DqEnum gsInqClipSrc ()**
- **void gsResetToDefaults ()**
- **void gsHardReset ()**
- **void gsSetBackgroundColor (DqQByte qColor)**
- **DqQByte gsInqBackgroundColor ()**
- **void gsClearView (DqQByte qColor)**
- **void gsValidFont (int iIndex)**
- **void gsText (long lX, long lY, char *pcText)**
- **void gsFontDir (char *pcDir)**
- **void gsOpenFont (char *pcFile)**
- **void gsCloseFont (int iIndex)**
- **void gsChar (long lX, long lY, int iChar)**
- **void gsGetCharExtent (int iChar, long lX, long lY,**
  GsExtent *ptGsExtent)

- **void** gsGetTextExtent (**char** *pcText, **long** lX, **long** lY,
  GsExtent *ptGsExtent)

In addition to the gs functions there are two functions provided for quickly displaying large sets of points. These are DrawPointSet and DrawDotSet.

- **void** DrawPointSet (**const int** NumPoints, **int*** Points)

  Given an array of x, y coordinates draw a point at each coordinate. The n^{th} coordinate occupies the elements $2n$ and $2n + 1$.

- **void** DrawDotSet (**const int** NumDots, **int*** Dots)

  Given an array of x, y coordinates draw a dot at each coordinate. The n^{th} coordinate occupies the elements $2n$ and $2n + 1$.

Directed perception pan-tilt unit

These functions directly call their counterparts on the Directed Perception pan-tilt head. In addition to the standard Directed Perception commands, pan and tilt positions can be specified in degrees, radians, grads, pixels, as well as the default pan_tilt_ticks. The width and height of the camera in pixels can be specified for calibration with the other angular measures. BUGS: if the pan-tilt head is not connected the server will hang waiting for a reply from the head.

No C support is provided for these functions.

Resolution settings

Set the constants used to compute angular distance in terms of pixels.

- **double** PanRes (**double** d)

  Specify the expected pan resolution in *seconds/arc*. This should always be equal to the return value from the pan_res function.

- **double** TiltRes (**double** d)

  Specify the expected Tilt resolution in *seconds/arc*. This should always be equal to the return value from the tilt_res function.

- **double** CameraWidth (**double** d)

  Specify the expected camera width in *pixels/degree*. Actual pan movements in terms of pixels are computed from this and the PanRes.

- **double** CameraHeight (**double** d)

  Specify the expected camera height in *pixels/degree*. Actual tilt movements in terms of pixels are computed from this and the TiltRes.

- **double** PanRes ()

  Query expected pan resolution in *seconds/arc*. This should be the same as the return value from the pan_res function.

- **double** TiltRes ()

  Query expected tilt resolution in *seconds/arc*. This should be the same as the return value from the tilt_res function.

- **double** CameraWidth ()

  Query expected camera width in *pixels/degree*.

- **double** CameraHeight ()

  Query expected camera width in *pixels/degree*.

Positioning

These functions make position queries and requests as described in sections 4.2.1 and 4.2.2 of the pan-tilt head manual. Legal MovementType's are:

- **pan_tilt**::Relative
- **pan_tilt**::Absolute

Legal AngleTypes are:

- **pan_tilt**::pan_tilt_ticks
- **pan_tilt**::degrees
- **pan_tilt**::radians
- **pan_tilt**::grads
- **pan_tilt**::pixels
- **double** pan (**pan_tilt**::AngleType at = **pan_tilt**::pan_tilt_ticks)
- **double** tilt (**pan_tilt**::AngleType at = **pan_tilt**::pan_tilt_ticks)
- **void** pan (**double** ppos,
  pan_tilt::MovementType mt = **pan_tilt**::Relative,
  pan_tilt::AngleType at = **pan_tilt**::pan_tilt_ticks)
- **void** tilt (**double** tpos,
  pan_tilt::MovementType mt = **pan_tilt**::Relative,
  pan_tilt::AngleType at = **pan_tilt**::pan_tilt_ticks)
- **void** pan_and_tilt (**double** ppos, **double** tpos,
  pan_tilt::MovementType mt = **pan_tilt**::Relative,
  pan_tilt::AngleType at = **pan_tilt**::pan_tilt_ticks)

pan/tilt resolution

As described in section 4.2.3 or the pan-tilt manual these commands query the pan/tilt resolution in *seconds/arc*.

- **double** pan_res ()
- **double** tilt_res ()

position limits

Query the pan/tilt limits as described in section 4.2.4 of the pan-tilt manual.

- **double** pan_min (**pan_tilt**::AngleType at = **pan_tilt**::pan_tilt_ticks)
- **double** pan_max (**pan_tilt**::AngleType at = **pan_tilt**::pan_tilt_ticks)
- **double** tilt_min (**pan_tilt**::AngleType at = **pan_tilt**::pan_tilt_ticks)
- **double** tilt_max (**pan_tilt**::AngleType at = **pan_tilt**::pan_tilt_ticks)

Position limits enforcement

Query and specify whether position limits are enforced or not as described in 4.2.5 of the pan-tilt manual. PositionLimits may be set to **pan_tilt**::Disable, and **pan_tilt**::Enable.

- **int** pos_limits_enforced ()
- **int** plimits (**pan_tilt**::PositionLimits p)

Immediate/slave position execution mode

Specify execution mode as slave or immediate and execute slaves as described in 4.2.6, and 4.2.7 of the pan-tilt manual. ExecutionMode may be set to **pan_tilt**::Immediate, or **pan_tilt**::Slave.

- **int** exec_mode (**pan_tilt**::ExecutionMode em)
- **int** execute_slaves ()

Await position command completion

Pause until a positioning command is completed as described in section 4.2.8 of the pan-tilt head manual.

- **int** wait_pan_tilt_command ()

Speed

Query and specify pan/tilt speed as described in section 4.3.2 of the pan-tilt manual.

- **int** pan_speed ()
- **int** tilt_speed ()
- **int** pan_speed (**int** pspeed)
- **int** tilt_speed (**int** tspeed)

Acceleration

Query and specify pan/tilt acceleration as described in section 4.3.3 of the pan-tilt manual.

- **int** pan_accel ()
- **int** tilt_accel ()
- **int** pan_accel (**int** paccel)
- **int** tilt_accel (**int** taccel)

Base startup speed

Query and specify pan/tilt base speed as described in section 4.3.4 of the pan-tilt manual.

- **int** pan_base_speed ()
- **int** tilt_base_speed ()
- **int** pan_base_speed (**int** pbspeed)
- **int** tilt_base_speed (**int** tbspeed)

Speed bounds

Query and specify pan/tilt base speed limits as described in section 4.3.5 of the pan-tilt manual.

- **int** pan_low_speed ()
- **int** pan_high_speed ()
- **int** tilt_low_speed ()

- **int** tilt_high_speed ()
- **int** pan_low_speed (**int** plspeed)
- **int** pan_high_speed (**int** phspeed)
- **int** tilt_low_speed (**int** tlspeed)
- **int** tilt_high_speed (**int** thspeed)

Unit commands

Pan-tilt unit state commands as described in sections 4.4.1, 4.4.2, and 4.4.3 of the pan-tilt head manual.

- **int** reset ()
- **int** save ()
- **int** restore ()
- **int** restore_default ()
- **int** version ()

Power consumption

Query/specify power consumption as described in sections 4.5.1, and 4.5.2 of the pan-tilt head manual. The PowerMode enumerated type has the enumerations Hi, Regular, Low, and Off.

- **int** pan_hold_power_mode ()
- **int** pan_hold_power_mode (**pan_tilt**::PowerMode pm)
- **int** tilt_hold_power_mode ()
- **int** tilt_hold_power_mode (**pan_tilt**::PowerMode pm)
- **int** pan_move_power_mode ()
- **int** pan_move_power_mode (**pan_tilt**::PowerMode pm)
- **int** tilt_move_power_mode ()
- **int** tilt_move_power_mode (**pan_tilt**::PowerMode pm)

Scheduling commands

These commands provide the ability to insure that certain commands are executed in series without another clients commands being interleaved between them.

Sequences

Sequences are collections of commands that are treated as a single atomic request by the server. By treating the sequence as an atomic request no other clients commands can interrupt the order of execution in the sequence. Sequences are initiated with a call to BeginSequence. After BeginSequence is called all server requests are cached on the client until EndSequence is called. When EndSequence is called all cached commands are transmitted to the server in a single package. Since the cached commands are transmitted all at once the amount of network overhead in transmitting the commands is reduced. After the server receives the commands in the sequence it schedules them as an atom. While they are executed their results may be sent back to the client or discarded depending on which version of EndSequence is called.

- **void** BeginSequence ()
- **void**DQS_ BeginSequence ()

  Begin a sequence. No commands will be sent to the server until the sequence is finished with a call to EndSequence.

- **void** EndSequence ()
- **void** DQS_EndSequence ()

  End a sequence. Information from all deferred commands is discarded by the server.

- **void** EndSequence (**ExecutionCode&** ec)
- **void** DQS_EndSequence (**ExecutionCode&** ec)

  End a sequence. The ExecutionCode returned is made by or'ing each command in the sequence's execution code together and or'ing Success-NoReturnData with this sum.

- **void** EndSequence (**DataPackage*&** pdp,
  ExecutionCode*& aec,
  int& Size)

- **void** DQS_EndSequence (**DataPackage*&** pdp,
  ExecutionCode*& aec,
  int& Size)

  End a sequence. Information from all deferred commands is read from the socket into the DataPackage and ExecutionCode arrays, these array are returned in adp and aec and their size is returned in NumOfpdp. Use of this command requires knowledge of the command packaging schema and is discouraged until a less primitive interface is developed.

Mutual Exclusion

A second method for insuring that series of commands are not interrupted by other clients is to use mutual exclusion. A client may enter mutual exclusion with a call to BeginMutex. Only one client can be in mutual exclusion at a time and if a client is in mutual exclusion only its requests are scheduled by the server. A client exits mutual exclusion by calling EndMutEx or losing its connection.

Mutual exclusion differs from sequences in that DataCube server in that requests are not deferred until EndMutEx is called. Instead they are executed and return responses in the same way that they do when a client is not in mutual exclusion.

It should be noted that if client A is in mutual exclusion mode all commands sent to the server by client B will be executed, but not until client A leaves mutual exclusion. If client B executes a command that returns a result from the server when client A is in mutual exclusion then client B will block until that result can be returned (i.e. it will block until client A leaves mutual exclusion). If client B executes a command that does not return a result from the server it will *not* block, but the command will not have been executed. Thus, client B could accumulate many commands on the server if none of them returned a response.

- **int** BeginMutEx ()
- **int** DQS_BeginMutEx ()

  Begin mutual exclusion. This command stops all other users commands from being scheduled if it succeeds. This command returns true if your are able to enter mutual exclusion mode, it returns false if you are not able (i.e. another client is in mutual exclusion).

- **int** EndMutEx ()
- **int** DQS_EndMutEx ()

  End mutual exclusion. This command ends a mutual exclusion session, allowing other clients commands to be executed. This command returns true on success and false on error (i.e. if you were not in mutex when it was called).

Server administrative commands

The functions in this class provide administrative capabilities such as killing users, clearing the log file, and listing the connected clients. These functions are primarily for debugging purposes. In order to execute any command in the administration class except for AdminMode the user must be in administration mode. To enter administration mode call AdminMode.

- **int** AdminMode ()
- **int** DQS_AdminMode ()

  Enter administration mode to provide client with the privilage to execute other administrative functions.
- **int** ClientMode ()
- **int** DQS_ClientMode ()

  To exit administration mode execute this command.
- **int** KillClient (**const int** client)
- **int** DQS_KillClient (**const int** client)

  This command will close the socket connection to a given client. The client number must be known, see ListClients. This command returns true if client is a legal client number, otherwise it returns false.
- **int** ClearLog ()
- **int** DQS_ClearLog ()

  Delete all the data in the log file and reinitialize its header. This command should always return true.
- **int** Log (**const int** fLog)
- **int** DQS_Log (**const int** fLog)

  If fLog is true then turn command logging on, otherwise disable command logging. This command should always return true. The default is *not* to log commands.
- **int** SetInitPort (**const int** port)
- **int** DQS_SetInitPort (**const int** port)

  Change the initialization port. This will NOT disconnect any existing clients, but future clients will have to connect to the new port number. This command should always return true.
- **int** ListClients (**int***& aClientList)
- **int** DQS_ListClients (**int***& aClientList)

  List the currently connected clients.
- **int** NumClients ()
- **int** DQS_NumClients ()

  Return the number of clients connected.

- **int** MaxNumClients ()
- **int** DQS_MaxNumClients ()

  Return the maximum number of clients that can be connected at any time.
- **void** ShutDown ()
- **void** DQS_ShutDown ()

  Shutdown the server, closes all socket connections and disconnects the DataCube.

Networking commands

This class provides the ability to connect and disconnect to any server. It is used primarily in the constructor and destructor of the client class and to initiate DataCube sessions.

- **void** WaitForConnection (**const char** * **const** shost, **const int** port);
- **void** DQS_WaitForConnection (**const char** * **const** shost,
  const int port);

  Try to create a socket connection with a DataCube server running on the machine shost at the specified port. If the connection is unsuccessful ask the user if another attempt should be made.
- **int** Connect (**const char** * **const** shost, **const int** port)
- **int** DQS_Connect (**const char** * **const** shost, **const int** port)

  Try to create a socket connection with a DataCube server running on the machine shost at the specified port. If the connection is successful return true, otherwise return false.
- **int** Disconnect ()
- **int** DQS_Disconnect ()

  Close a connection with the server. If there was no connection to close in the first place return false otherwise return true.
- **int** SetRecvBuf (**const int** Size)
- **int** DQS_SetRecvBuf (**const int** Size)

  Specify the number of bytes in the servers receive buffer for the clients socket connection. If images are being transferred this should be large, if only integers are being transferred then this should be small.
- **int** ID ()

- **int DQS_ID ()**

  Ask the server for your ID and return it. On an error return -1.

- **void Wait ()**
- **void DQS_Wait ()**

  Block until all commands requested by the client have been executed. Commands that return no result are scheduled and executed in the background. Also sequences that do not return results are executed in the background. Wait blocks until these are completed. This is useful for keeping in sync with other devices.

- bf void SetID (const char* const sID)
- bf void DQS_SetID (const char* const sID)

  Specify a string to be associated with the client. Other clients can look at this string with GetID to determine who a client is.

- bf int GetID (const int Client, char sID[])
- bf int DQS_GetID (const int Client, char sID[])

  Ask for a clients ID string. The string sID should point to initialized memory. The ID string written into sID will always be shorter than Distribute::ClientIDLength defined in Distribute.H. Currently this value is 256 and thus sID should be of length at least 256. Return true if Client is a legal client number , else return false.

Programming client from C

To use the server from C you must link the libC.a library from C++ and the libC_client.a library from the server. The libC_client.a library defines a C++ main function that is used to initialize all static members of the classes. This main calls the function c_main which is defined to be extern "C". You should supply the c_main function instead of main.

From C there is no way to access the DQclient class directly. Instead all its member functions have C function counterparts with a "DQS_" prefix. Besides for the "DQS_" prefix, a suffix is added for members with overloaded names. The below list shows the correspondence:

Programming Tips

The client_test Interpreter

There is an interpreter in $DQ_SERVER_HOME called dq_server/client. This program allows users to type in any request to the server and the results of the

request are returned. This is a useful experimentation and debugging tool.

Trouble Shooting

- The Log file One particularity useful feature for detecting bugs is to enable command logging by the server with the Log command (note: this command requires that a client enter administration mode with a call to AdminMode). When this is enabled all commands transmitted to the server are written to the log file. The log file can be inspected to determine where a program crashed or where it made the wrong sequence of calls.

- Server wont start - Unable to Open Event Manager.

  - A server or another DataCube program is currently running.
  - You are not on the DataCube host.

- Client cannot connect to server

  - Server is being run at a different port than the client expects: tell client to use correct port or run client interpreter, execute AdminMode, and use SetInitPort to change the servers port.
  - Server is not running.
  - Server has not finished initialization.
  - Client is not on the network.

- Server crashes

  - If your program exits while it is waiting for information from the server the server will crash because it is writing to a closed socket. This may happen when the server is executing a long sequence that returns data and the client is exited with control-C.

- Server hangs

  - The server will hang if a pan-tilt command is requested while the pan-tilt head is not connected to the server.

- Template symbols are undefined at link time

  Some C++ linkers are not able to figure out which template functions have been used in the libraries they are linking and thus do not know which functions to instantiate and link. To resolve this problem one can create an object file (not a library) that declares all templates used by the libraries and link this object into the binary. Since the file containing main is often linked as an object rather than a library the definitions may

be declared in its object. For clients the following function can be added to mains object file:

```
void tmp () {
//Force template instantiation for the library routines.
  xv<unsigned char, 1> xvc1;
  xv<unsigned char, 3> xvc3;
  xvHist xvh;
}
```

When writing servers declare this function:

```
void tmp () {
//Force template instantiation for the library routines.
  xvdq<unsigned char, 1> xvc1;
  xvdq<unsigned char, 3> xvc3;
  xvHist xvh;
}
```

Upgrading

When a new version of the server is installed all clients much be recompiled.

Chapter 2

Extending the DataCube Server

Introduction

This document describes the process of adding pipelined functions to the DataCube server. It involves writting imageflow code in C++ under the dq_global class. An understanding of C++ and imageflow is necessary. A document on the dq_global class is provided as reference when writting code.

A quick overview

To add a set of routines to the DataCube server one must perform 4 steps. First they need to design the DataCube routines. This is done by creating a class for the routines to live in and adding that class to the DataCube routine class hierarchy. Second an entry in the dq_server_command enumerated type must be created for each member of the class that is to be distributed by the server. Third an exec class must be made to interpret the entries in the dq_server_command enumerated type and execute the appropriate command from the DataCube class. The executor class must be made to inherit this exec class class. Fourth a client base class must be made as an application programmatic interface to client writers and inherited by the client class.

The DataCube routine class hierarchy

The first step is to write your imageflow routines within the servers class hierarchy. The DataCube routines are currently divided into 4 class structures: the dq_global DataCube base class, the multi-resolution classes, the color histogram

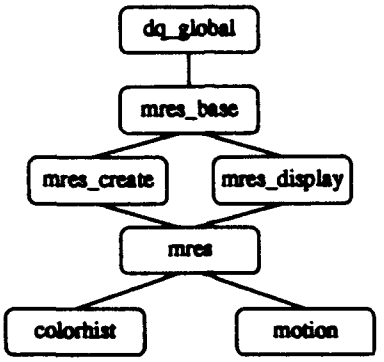

Figure 2.1: Initial class hierarchy

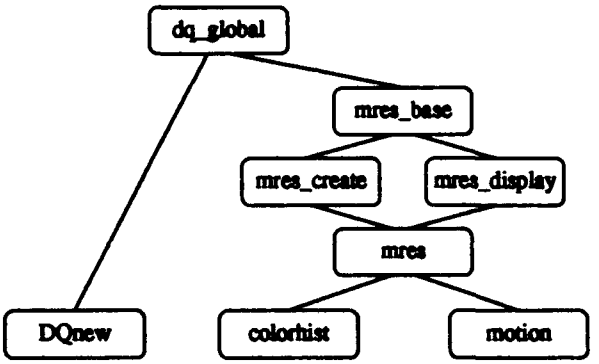

Figure 2.2: Class hierarchy after inheriting dq_global

class, and the motion class. The inheritance in this class hierarchy is shown in figure 2.1.

Inheritance hierarchies

If you were to add the class DQnew to this hierarchy there would be 5 ways to connect it. At the very least DQnew has to virtually inherit dq_global. Without dq_global DQnew would not be able to access any of the DataCube IP devices. In addition, dq_global is needed to allocate memory properly and to inform other classes when DQnew modifies certain elements (see the section on dq_global). This class hierarchy is shown in figure 2.2.

Often a class will need a frame acquisition pipe and a display pipe. In these cases you may want to virtually inherit the mres class as in figure 2.3. By virtually inheriting this class you have multi-resolution frame grabbing and display, and pipes can be made to operate on active regions rather than whole

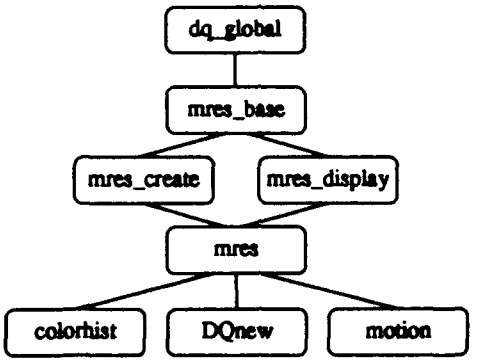

Figure 2.3: Class hierarchy after inheriting mres

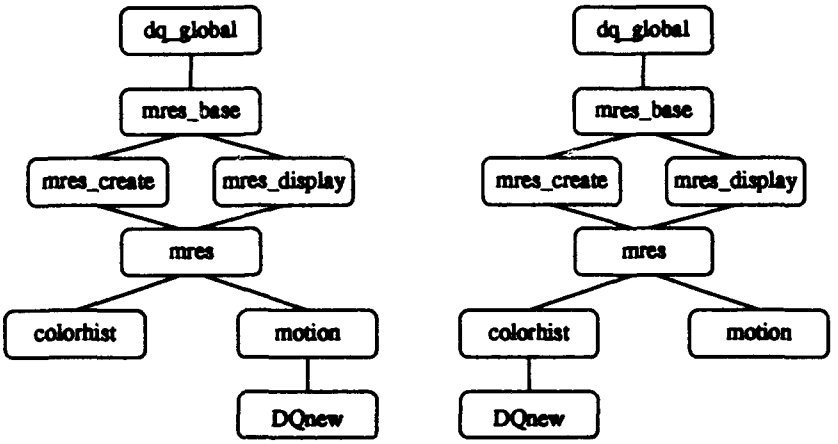

Figure 2.4: Class hierarchy after inheriting motion or colorhist

levels of a particular resolution. It is the highly recommended to operate on active regions because if all classes do this then there will be a single mechanism for restricting processing in all functions.

It may be the case that you are writting routines that will be built onto an existing set of routine. You may use the results from a color histogram backprojection and therefore will want to virtually inherit the colorhist class. Alternately you may want to use the output of the motion detecting routine. In these two cases you would result in one of the two hierarchies in figure 2.4. It may even be the case that you need both the results of motion detection and of color histograms, resulting in multiple virtual inheritance as in figure 2.5.

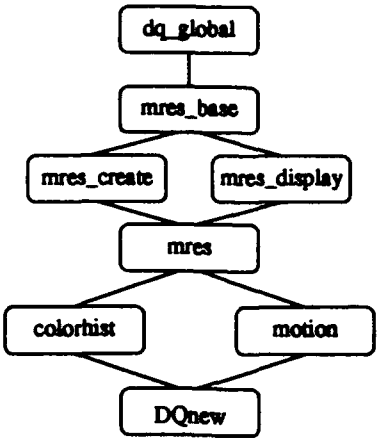

Figure 2.5: Class hierarchy after inheriting colorhist and motion

Resource management: the dq_global base class

Memory allocation

Inorder for many functions to independently use the DataCube, but to exist in a single application, ther must be a mechanism for functions to allocate memory and be sure that no other function will write to that memory. This is accomplished in the dq_global base class through a function called allocate. The function allocate returns rectangles of a requested size in a specified AM memory. Once allocate returns a rectangle it will not allocate a rectangle that intersects it to any other function. Thus, if all DataCube functions only use processing rectangles returned from allocate then they will be guaranteed not to have memory conflicts. See the section on the dq_global class for more information.

IP element state monitoring

When integrating multiple pipelined applications it is often necessary that the functions or PATs that setup and fire pipes be context independent (see the DataCube Image Processing Manual for more on context independent PATs). Since some elements can take a long time to set it is desirable that they only get set when they absolutely have to. These elements include all LUT's, the DigiColors DAC and ADC, and the NMAC. The dq_global class provides the ability to keep track of which class last modified an element so that the element can be reset only when it has been changed.

Each class can request an identification number though a call to the GetNewUserID member of dq_global. This is typically done in the classes construc-

tor and the returned ID is used throughout the life of the class. When an element that is being monitored is modified the class should call the UseElem member of dq_globals with the elements identification information and the classes ID. If a pipe is later fired that requires that the elements state has not changed GetElemUser or UnmodifiedElem can be called to see if the element has been changed; if it has been changed it should be reset to its original state and a call to UseElem should be made. It is important that UseElem be called whenever a monitored element is modified; currently the following elements are monitored: all LUT's, the DAC and ADC on the DigiColor, and the surfaces in all NMACS. See the section on dq_global for identification conventions for these elements.

Compiling into the class hierarchy

Include files

Inorder for a class to inherit into this hierarchy it must include the appropriate header files. Each class (dq_global, mres, colorhist, and motion) has an associted header in $DQ_SERVER_HOME/include of the same name with a *.H* extension (dq_global.H, mres.H, colorhist.H, motion.H). Each class that is inherited as a base should have its header included. For example, if mres is the base then mres.H should be included and if colorhist and motion are both inherited using multiple inheritance then both colorhist.H and motion.H should be included (in any order).

Libraries

The classes in the server are archived into a set of libraries. To make a new server you must link with the following libraries from $DQ_SERVER_HOME/lib:

- libserver.a
- libmotion.a
- libcolorhist.a
- libmres.a a
- libdq_base.a a
- libcomm.a a
- libdlist.a a
- libhead.a a
- libxv_class.a
- libmisc.a

From $KHOROS_HOME/lib you must link with:

- libvipl.a
- libvutils.a
- libvmath.a
- libvgparm.a
- libverror.a
- libUofC_Khoros.a

And from $DQHOME/lib you must link libdq.a. The order in which these libraries are linked is important, if the are not linked in the order listed then undefined symbol errors will occur at link time.

An example

Here is an example class definition that needs acquisition and display from the mres class.

```
class DQnew : virtual public mres {
private:
  DqSurf oSurf;
public:
  DQnew () : mres () {oSurf = allocate (0, 0, 100, 100);}
  void Func1 (int i);
  int Func2 (const xvdq<unsigned char, 1>& const xvc1, xvHist& xvh);
}
```

If the DQnew were compiled and archived into the library libDQnew.a the followinglink command would be used to link the new server. Note that since these routines use templates the include path is needed at link time. Also note that libDQnew.a is linked after (listed before) the mres libraries since it will use some of the mres functions. New DataCube libraries should always link before (be listed after) libserver.a because libserver.a needs to reference their functions.

```
CC -o dq_server dq_server.o
   -I${KHOROS_HOME}/include_C++
   -I${DQHOME}/include_C++
   -I${DQ_SERVER_HOME}/include
   -L${DQ_SERVER_HOME}/lib -lserver -lDQnew -lmotion -lcolorhist -lmres
     -ldq_base -lcomm -ldlist -lhead -lxv_class -lmisc
   -L${KHOROS_HOME}/lib -lvipl -lvutils -lvmath -lvgparm -lverror
     -lUofC_Khoros
   -L${DQHOME}/lib -ldq
```

Debugging

Before linking with the server one may wish to simply link the datacube class hierarchy together for debugging purposes. To do this libserver.a does not need to be linked, but all the other libraries listed above are needed. Then you could instantiate the DQnew class and call its member functions directly from a test program.

The dq_server_command enumerated type

Once a class is added to the DataCube hierarchy it needs to be incorporated into the server. The first step is to decide which functions of the class will be distributed by the server across the network. A code will have to be assigned to each function inorder for the client to communicate to the server that it needs that function executed. Assigning codes to functions is done in the dq_server_command enumerated type that is defined in the files commands.H and commands.C in $DQ_SERVER_HOME/lib/src. This enumeration contains a code for every command that can be executed by the server.

Before the TOTAL_NUMBER_OF_COMMANDS enumeration at the end of the dq_server_command declaration in commands.H, but after all the other enumerations in the list, you should add an enumeration for each function to be distributed plus a symbol to signify the beginning and end of your enumeration list. It is suggested that the name of the code you use be the name of your class with the name of the function appended to its end (append a number after the function name if the function is overloaded), and the name of the enumeration to signify the beginning and end of your list be the name of your class with First and Last appended to them.

There is a string associated with each of the dq_server_command enumerated type. These strings are defined in the asdq_server_command_names array in commands.C. When logging is enabled each time a command is sent to the server the string associated with the commands enumeration is written to the command log. The next step is to write these strings into the asdq_server_command_names array. It is suggested that the string associated with each enumeration simply be the name of the enumeration it corresponds to.

Example

If you were adding the class DQnew with functions Func1 and Func2 to the server the following changed would be made:

- Original Commands.H:

```
...
GRAPHGetTextExtent,
```

```
    GRAPHBitBltView,
    GRAPHLast, //197
    /* do not add new commands past this line */
    TOTAL_NUMBER_OF_COMMANDS
  };
```

- New Commands.H

```
    ...
    GRAPHGetTextExtent,
    GRAPHBitBltView,
    GRAPHLast, //197
    DQnewFirst,
    DQnewFunc1,
    DQnewFunc2,
    DQnewLast,
    /* do not add new commands past this line */
    TOTAL_NUMBER_OF_COMMANDS
  };
```

The file Commands.C would be similarily modified.

- Original Commands.C:

```
    ...
    "GRAPHGetTextExtent",
    "GRAPHBitBltView",
    "GRAPHLast", //197
    /* do not add new commands past this line */
    "TOTAL_NUMBER_OF_COMMANDS"
  };
```

- New Commands.C:

```
    ...
    "GRAPHGetTextExtent",
    "GRAPHBitBltView",
    "GRAPHLast", //197
    "DQnewFirst",
    "DQnewFunc1",
    "DQnewFunc2",
    "DQnewLast",
    /* do not add new commands past this line */
    "TOTAL_NUMBER_OF_COMMANDS"
  };
```

The executor

After the enumerations are made for your functions, a class is needed that will parse commands sent by the client, execute the appropriate function, and return the results of the function call to the client. The class is usually named exec followed by the name of your class (execDQnew for the above class). This class must inherit the new DataCube class (DQnew) and have the following member functions:

- **int** Scheduable (**const** dq_server_command command) **const**

  This function returns true if the command can be scheduled and false otherwise. Almost all commands can be scheduled; in all but very rare circumstance this should return true. Disconnect is an example of a function that is not schedulable.

- **int** fDQnewCommand (**const** SchedItem& S) **const**

  Note, the name of this function should be f<dq-class-name>Command where class name is DQnew in this example. This function returns true if the command field of S is one of the enumerated commands for this class in the dq_server_command enumerated type. The standard implementation of this function is to return:

  ((S.command > DQnewFirst) && (S.command < DQnewLast)).

- dqDataPackage& exec (**const** SchedItem& S, ExecutionCode& ec)

  This function parses the message received from the client, executes the appropriate command, and returns a response to the client. It is described in detail below:

The exec function

This function parses the message received from the client, executes the appropriate command, and returns a response to the client. There are two classes that are used by this function to receive and transmit data: dqDataPackage and SchedItem.

dqDataPackage

The dqDataPackage is a container class for an array of integeres, an array of xvdq<**unsigned char**, 3>'s, an array of xvdq<**unsigned char**, 3>'s, and an array of xvHists. These are the only data types that can currently be transmitted across the network. When the client sends a command it packages the commands arguments into a dqDataPackage before transmitting them across the network. The exec function must unpack each dqDataPackage inorder to pass its contents as parameters to the appropriate DataCube function, and it

must pack the functions results into a dqDataPackage to return to the client. See the section on DataPackages for specific information on the member functions for the dqDataPackage class.

SchedItem

SchedItem is a class containing two data fields; a dq_server_commands called command, and a pointer to a dqDataPackage called pdp. The command field contains the command to be executed and pdp contains the arguements to the commands function call. The order of the functions arguments in the arrays in pdp should be from left to right in the command function argument list.

The function exec looks at the command field in S and determines which DataCube function to call. It then unpacks the data in pdp and calls the function with pdp's data as arguements. The results from the function call are packaged into a new dqDataPackage and this dqDataPackage is returned by exec. The ExecutionCode ec (the second argument to exec) must be set before exec returns. This parameter describes the result of the DataCube function call.

Return data from the exec function

ExecutionCode is an enumerated type with enumerations of SuccessNoReturnData, FailureNoReturnData, NoReply, SuccessWithReturnData, and FailureWithReturnData. If the function call does not return any data setting ec to SuccessNoReturnData or FailureNoReturnData will return just the ExecutionCode to the client. This way the client can know that the command has completed and what the status of its completion was. Alternately ec can be set to NoReply. When ec is set to NoReply the client receives no return information from the server and thus does not know when the command is finished executing. In all three cases (SuccessNoReturnData, FailureNoReturnData, and NoReply) no dqDataPackage is returned to the clients; at most an execution code is returned. Since no dqDataPackage is sent to the client one does not need to be returned by exec. Since exec returns a reference to a dqDataPackage you can simple returned a dereferenced bogus pointer to a dqDataPackage (A dereferenced NULL pointer is recommended). Note, setting ec to NoReply is the most efficient case since no transmission to the client occurs.

When a function needs to return information to the client it sets the ec to SuccessWithReturnData or FailureWithReturnData. In this case a dqDataPackage needs to be constructed containing the return data and returned by exec. Note that exec returns a reference to a dqDataPackage and therefore the returned dqDataPackage should not be a local variable to exec. Furthermore, delete will be called on the returned dqDataPackage so it should be created by new (it should not be a static or a member of an array).

Example of execDQnew

Here is what the execution class execDQnew could look like and how it can be implemented.

```
class execDQnew : DQnew {
private:
  dqDataPackage& CallFunc1 (dqDataPackage& dp, ExecutionCode& ec);
  dqDataPackage& CallFunc2 (dqDataPackage& dp, ExecutionCode& ec);

public:
  execDQnew ();
  int Scheduable (const dq_server_command command) {return true;}
  fDQnewCommand ((const SchedItem& S) const {
    return ((command > DQnewFirst) && (command < DQnewLast));
  }
  dqDataPackage& exec (const SchedItem& S, ExecutionCode& ec);
};

dqDataPackage& execDQnew :: exec (const SchedItem& S,
                                  ExecutionCode& ec) const {
  switch (S.command) {
  case DQnewFunc1: return CallFunc1 (*(S.pdp), ec);
  case DQnewFunc2: return CallFunc2 (*(S.pdp), ec);
  }
}
dqDataPackage& execDQnew :: CallFunc1 (dqDataPackage& dp,
                                       ExecutionCode& ec) {
  assert (dp.size (1, 0, 0, 0));
  ec = NoReply;
  Func1 (dp.Data (0));
  dqDataPackage* pdp = (dqDataPackage*) NULL;
  return *pdp;      //Returned data isn't used so just dereference NULL
}
dqDataPackage& execDQnew :: CallFunc2 (dqDataPackage& dp,
                                       ExecutionCode& ec) {
  assert (dp.size (0, 1, 0, 1));
  ec = SuccessWithReturnData;
  dqDataPackage* pdp = new dpDataPackage;
  int* aData = new int [1];          //Array is expected, not "new int"
  pdp->set_data (1, aData);
  aData [0] = Func1 (dp.xvc1Data (0), dp.xvh (0));
  return *pdp;
}
```

Connecting execDQnew to the server

To connect execDQnew to the server the definition of the executor class will need to be modified. This class is defined in $DQ_SERVER_HOME/lib/src/server in the files executor.H and executor.C. The executor class needs to be modified so that it inherits execDQnew as a protected base. Modify the Scheduable member function of executor so that it returns true when its current definition would return true and execDQnew's Schedulable function returns true. Finally modify executors exec function to call your exec function if execDQnew's fDQnewCommand function returns true.

Example of executor

If the old definition of executor base classes were defined by:

```
class executor :
protected execS,
protected execGRAPH {
```

The new definition would be:

```
class executor :
protected execS,
protected execGRAPH,
protected execDQnew {
```

The exec function would change from:

```
dqDataPackage& executor::exec (const SchedItem& S, ExecutionCode& ec) {
  if (fSCommand (S)) {
    return execS::exec (S, ec);
  } else if (fGRAPHCommand (S)) {
    return execGRAPH::exec (S, ec);
  } else {
    fsLogFile << "Bad Command:   " << endl;
    ...
```

To:

```
dqDataPackage& executor::exec (const SchedItem& S, ExecutionCode& ec) {
  if (fSCommand (S)) {
    return execS::exec (S, ec);
  } else if (fGRAPHCommand (S)) {
    return execGRAPH::exec (S, ec);
  } else if (fDQnewCommand (S)) {
```

```
    return execDQnew::exec (S, ec);
  } else {
    fsLogFile << "Bad Command:   " << endl;
    ...
```

The Schedulable function changes from:

```
int executor :: Scheduable (const dq_server_command command) const {
  return (execS::Scheduable (command) &&
          execGRAPH::Scheduable (command) &&
}
```

To:

```
int executor :: Scheduable (const dq_server_command command) const {
  return (execS::Scheduable (command) &&
          execGRAPH::Scheduable (command) &&
          execDQnew::Scheduable (command));
}
```

Adding to the client class

After the server side changes have been made the client class needs to be modified so that programmers can use it to call the new functions on the server. To do this create a class whose name is client with the name of your new DataCube class appended to it, clientDQnew in this case. This class should have all the members of the DataCube class that are to be distributed (i.e. Func1 and Func2). Your class clientDQnew should virtually inherit client_base as a protected base and should be inherited by the class client as a public base. To declare clientDQnew should include $DQ_SERVER_HOME/include/client_base.H. To change the declaration of the class client you need to modify the file client.H in $DQ_SERVER_HOME/include.

Each of the functions in clientDQnew (i.e. Func1 and Func2) should package their arguments into a DataPackage (not a dqDataPackage) and call sendcomm (a member function of client_base) to send the appropriate dq_server_command and the DataPackage to the server. If no arguments are being sent to the server no DataPackage needs to be constructed and sendcomm can be called with only the dq_server_command.

After sendcomm is called there are three possibilities.

- No reply will be returned by the server because the exec function returned an ExecutionCode of NoReply.
- An execution code will be returned by the server but no data is returned.

This is the result of the exec function returning SuccessNoReturnData or FailureNoReturnData.

- An execution code will be returned along with a DataPackage. This is from the exec function returning SuccessWithReturnData or FailureWithReturnData.

No reply is made by the server

In this case the client needs to do nothing after it calls sendcomm. It can simply return.

ExecutionCode returned by server, but no DataPackage

If the command needs to wait for an ExecutionCode to be returned by the server then it should call the recvcomm member function of client_base with no arguements. This function will wait for the server to send an ExecutionCode and will return the ExecutionCode to the caller. If the function was called while a sequence is being defined this recvcomm will return NoReply, otherwise it should return SuccessNoReturnData or FailureNoReturnData.

ExecutionCode and DataPackage returned by server

If a command needs to wait for both an ExecutionCode and a DataPackage to be returned by the server it should call the recvcomm member function of client_base with an DataPackage as an argument. If the DataPackage passed to recvcomm is empty (i.e. the size of each of its arrays is 0) then memory will automatically be allocated for each of its internal arrays and the return data from the server will be written into it. If the DataPackage is not empty then it is assumed that it is the same size as the return data from the server. If this is not the case an assertion will cause the client to exit. Any data in the DataPackage will be written over by the data received from the server. If the function is called while a sequence is being defined recvcomm will return NoReply and nothing will be written into the DataPackage.

Example of the modified client class

```
class clientDQnew : protected client_base {
public:
  clientDQnew ();
  void Func1 (int i);
  int Func2 (const xvdq<unsigned char, 1>& xvc1, const xvHist& xvh);
};
```

Our old client class definition was:

```
class client : public clientcolorhist, ...
```

The new definition becomes:

```
class client : public clientDQnew, public clientcolorhist, ...
```

Here is the implementation of the functions Func1 and Func2.

```
void clientDQnew :: Func1 (int i) {
  DataPackage dp;
  int* aData = new int [1];          //Array is expected, not "new int"
  aData [0] = i;
  dp.set_data (1, aData);
  sendcomm (DQnewFunc1, dp);
}

int clientDQnew :: Func2 (const xvdq<unsigned char, 1>& xvc1,
                          const xvHist& xvh) {
  DataPackage dp;
  dp.set_data (1, &xvc1);
  dp.set_data (1, &xvh);
  sendcomm (DQnewFunc1, dp);
  dp.Release_xvc1 ();              //Dont let dp's destructor delete this
  dp.Release_xvh ();               //Dont let dp's destructor delete this
  int* aData = new int [1];          //Array is expected, not "new int"
  dp.set_data (1, aData);
  assert (recvcomm (dp) == SuccessWithReturnData);   //Always succeed
  return aData [0];
}
```

Compiling tips

When compiling it is useful to compile with the -g (debugging) option, and to make sure that NDEBUG is *not* defined. Defining NDEBUG (usually with -DNDEBUG option on the compilation command line) speeds up the server by removing assertions made with the assert macro. However, these assertions flag exceptional conditions and are useful when debugging. No assertion in the server will ever be flagged in correct code, even it the code is given incorrect input, therefore they are not needed after a program is debugged.

Chapter 3

The DataPackage Container Class

Overview

This is a class for packaging data for transmission across the network. It stores arrays of **int**'s, xv<**unsigned char**, 3>'s, xv<**unsigned char**, 3>'s, and xvHists. With the CSsocket class DataPackage's can be sent and received across sockets. Another class, called dqDataPackage has the exact same functions except that it contains arrays of dqxv<**unsigned char**, 3>'s, dqxv<**unsigned char** and 3>'s rather than xv<**unsigned char**, 3>'s, xv<**unsigned char** and 3>'s. It is legal to receive a dqDataPackage when a DataPackage was sent and vice-versa.

The data stored in this class is:

```
int Sizepxvc1;                          //Size of pxvc1 array
xv<unsigned char, 1> *pxvc1;            //Array of 1 band images
int Sizepxvc3;                          //Size of pxvc3 array
xv<unsigned char, 3> *pxvc3;            //Array of 3 band images
int Sizepxvh;                           //Size of pxvh array
xvHist *pxvh;                           //Array of histograms
int Sizep;                              //Size of p array
int* p;                                 //Array of integers
```

Member functions

- **void** clear ();

  Delete all arrays and set their sizes to zero.

- **int** NumOfint () **const**

  Return the number of elements in the integere array.

- **int** NumOfxvHist () **const**

  Return the number of elements in the histogram array.

- **int** NumOfxvc1 () **const**

  Return the number of elements in the one band image array.

- **int** NumOfxvc3 () **const**

  Return the number of elements in the three band image array.

- **int** empty () **const**

  Return if all arrays in the DataPackage are empty.

- **int** size (**const int** _Sizep, **const int** _Sizepxvc1, **const int** _Sizepxvc3, **const int** _Sizepxvh) **const**

  Return true if the size of each array is the same as that specified in the arguments. If an argument is -1 then ignore that field.

- **int*** pData () **const**

  Return the integer array, but do not remove the array from the DataPackage like release_int does.

- xvHist* pxvhData () **const**

  Return the histogram array, but do not remove the array from the DataPackage like release_xvh does.

- xv<**unsigned char**, 1>* pxvc1Data () **const**

  Return the one band image array, but do not remove the array from the DataPackage like release_xvc1 does.

- xv<**unsigned char**, 3>* pxvc3Data () **const**

  Return the three band image array, but do not remove the array from the DataPackage like release_xvc3 does.

- **int&** Data (**const int** i) **const**

  Return the i^{th} element in the integer array.

- **xvHist&** xvhData (**const int** i) **const**

  Return the i^{th} element in the histogram array.

- xv<**unsigned char**, 3>& xvc3Data (**const int** i) **const**

  Return the i^{th} element in the one band image array.

- **xv<unsigned char, 1>& xvc1Data (const int i) const**

  Return the i^{th} element in the three band image array.

- **void release_int ()**

  Return the integer array and remove any internal pointers to it do that when the DataPackage's destructor is called the array is not deleted.

- **void release_xvh ()**

  Return the histogram array and remove any internal pointers to it do that when the DataPackage's destructor is called the array is not deleted.

- **void release_xvc1 ()**

  Return the one band image array and remove any internal pointers to it do that when the DataPackage's destructor is called the array is not deleted.

- **void release_xvc3 ()**

  Return the three band image array and remove any internal pointers to it do that when the DataPackage's destructor is called the array is not deleted.

- **void set_data (const int _Sizep, int* _p)**

  Delete any currently stored integer array and replace it with the given integer array of the given size.

- **void set_data (const int _Sizepxvh, xvHist* _pxvh)**

  Delete any currently stored histogram array and replace it with the given histogram array of the given size.

- **void set_data (const int _Sizepxvc1, xv<unsigned char, 1>* _pxvc1)**

  Delete any currently stored one band image array and replace it with the given one band image array of the given size.

- **void set_data (const int _Sizepxvc3, xv<unsigned char, 3>* _pxvc3)**

  Delete any currently stored three band image array and replace it with the given three band image array of the given size.

- **int TransferSize () const**

  The number of bytes required to transfer a datapackage over the network. Useful for specifying the buffer size of a socket.

- **const CSsocket& operator<< (const CSsocket& CS, const DataPackage& dp)**

  Transmit a DataPackage to over a socket. When xv classes are transmitted only their height width and the imagedata is sent, map data, comments, and other fields are not transmitted. The transmission format is:

Sizep, Sizexvc1, Sizexvc3, Sizepxvh,

pxvc1 [0..Sizexvc1].x_max, pxvc1 [0..Sizexvc1].y_max,
pxvc3 [0..Sizexvc3].x_max, pxvc3 [0..Sizexvc1].y_max,

p [0..Sizep]
pxvc1 [0..Sizexvc1].imagedata,
pxvc3 [0..Sizexvc3].imagedata,
pxvh [0..Sizexvh].imagedata

- **const CSsocket& operator>** (**const** CSsocket& CS,
  DataPackage& dp)

  Read a DataPackage from a socket, but do not extract the data from the socket.

- **const CSsocket& operator≫** (**const** CSsocket& CS,
  DataPackage& dp)

  Read a DataPackage from a socket.

Chapter 4

Using the dq_global Base Class For DQ Applications

Intro

All classes that use the datacube should virtually inherit the class dq_global. This class provides functions that make paths between common elements (for example a path from the digicolors DC to memory, or from memory to AB's cross point switch), allocate memory from the AM devices (thus insuring that functions will not overwrite eachothers memory), and keep track of the last function to use LUT's, DAC's, and other elements with expensive state settings (to avoid redundent loadings of LUT's and the such).

Compiling with dq_global

The application must include dq_global.H from $DQ_SERVER_HOME/include and link with libdq_base.a from $DQ_SERVER_HOME/lib.

Keeping track of the last user of an element

- **int** GetNewUserID ()

  Ask for an ID to identify yourself as an element user. Typically each class will have its own ID.

- **void** UseElem (**const int** UserID_in, **const DqIPDev** oIPDev_in,
  const DqEnum eElem_in,
  const int ElemOtherAttrib_in = 0)

  Announce that you are going to use an element, to identify the element give the **DqIPDev** the element is on, the elements **DqEnum**, and use

another integer to specify any other attributes of the element. It is wise to carefully document what these three parameters are for any elements that will be tracked. Suggested conventions are:

- Elements that do not need a ElemOtherAttrib_in field for specification should not provide one (i.e. use the default of 0 for this discriptor).
- Cross point switch selections should just reserve the whole cross point switch.
- Multiplexer selections should reserve the multiplexer.
- Reserve AP_NDLY_SRCN for the AP NMAC surfaces AP_NMAC8, AP_NMAC4A, and AP_NMAC4B.
- LUT's should specify their bank in ElemOtherAttrib_in.
- calls to dcInitADCFormat should reserve DC_ADC.
- calls to dcInitDACFormat should reserve DC_DAC.

- **int** GetElemUser (**const DqIPDev** oIPDev_in, **const DqEnum** eElem_in,

  const int ElemOtherAttrib_in = 0)

  Return the ID of the last function to use this element.

- **int** UnmodifiedElem (**const int** UserID_in, **const DqIPDev** oIPDev_in,

  const DqEnum eElem_in,
  const int ElemOtherAttrib_in = 0)

  Return true if UserID_in was the last function to use this element.

- **int** GetNewUserID ()

  Return a new user ID. Typically classes call this in their constructor to get an ID that they will use throughout their existance.

Memory allocation

The suggested way to perform memory allocation is: Any surfaces that will not be moved or deleted during the life of a class should be allocated with allocate in the classes constructor. After all constructors are called in a class hierarch allocate should never be called again. After initialization any function that needs a temporary surface should use DupMemSurf with FreeSpaceLowX and FreeSpaceLowY to make a surface, and that surface should be considered unstable after the function returns.

Memory allocation is done by requesting rectangles from an AM memory. Rectangles are allocated starting in the upper left corner of a 2048 × 2048 surface and proceeding from left to right and then from top to bottom.

- **DqSurf** allocate (**const int** mem, **const int** SizeX, **const int** SizeY)

  Return a rectangle from memory number mem (0-6) of the specified size with the surfaces allignment point set such that the processing rectangles upper left corner is at (0, 0).

- **DqSurf** allocate (**const int** mem, **const DqRect** tRect)

  Return a rectangle from memory number mem (0-6) of the specified size with the surfaces allignment point set such that the processing rectangles upper left corner is at (0, 0).

- **int** ClearView (**const int** mem, **const DqQByte** qColor) **const**

  Clear an entire AM memory to a given color.

- **int** RecommendBlockSizeX (**const int** mem) **const**

  Return the largest width a rectangle can be without having to move to the next row in the memory map.

- **int** RecommendBlockSizeY (**const int** mem) **const**

  Return the largest height a rectangle can be without having to increase the height of the current row in the memory map.

- **int** FreeSpaceLowX (**const int** mem) **const**

  Return the upper left corner of the memory that is unused by the memory map. This can be used to store temporary results.

- **int** FreeSpaceLowY (**const int** mem) **const**

  Return the upper left corner of the memory that is unused by the memory map. This can be used to store temporary results.

- **DqSurf** DupMemSurf (**const int** mem) **const**

  Return a duplicate of the surface that covers this memory. Use this with FreeSpaceLowX, FreeSpaceLowY to make temporary surfaces.

Global surfaces for acquisition and display

- **DqSurf** oDC_AcqSrcSurf

  This is the acquisition source surface on the digicolor. It is initialized to be DC_ADC and of size 512 × 480.

- **DqSurf** oDC_DspSurf_Ovly

  This is the acquisition source surface on the digicolor. It is initialized to be DC_OVLY_MEM and of size 512 × 480.

- **DqSurf** oDC_DspSurf_DAC

  This is the acquisition source surface on the digicolor. It is initialized to be *DC_DAC* and *of size* 512 × 480.

- **DqSurf** aoDC_DspSurf [2]

  This array contains the above display surfaces oDC_DspSurf_Ovly and oDC_DspSurf_DAC. It can be indexed with the enumerated types Ovly and DAC.

Initialization

The constructor creates the standard system, and finds all IP devices. Then it initializes the DigiColor surfaces oDC_DspSurf_Ovly oDC_DspSurf_DAC aoDC_DspSurf and oDC_AcqSrcSurf. Finally it clears the memory map and element usage tables.

Debugging functions

- **void** PrintRect (**const DqSurf** Surf) **const**

  This function prints out various information about a surface includint what device it is on and what its processing rect is.

- **void** PrintRect (**const DqRect** tRect) **const**

  This function prints out the corners of a **DqRect**.

- **int** RectContained (**const DqRect** tBig, **const DqRect** tSmall) **const**

  Is the **DqRect** tSmall contained completely inside tBig.

Access to IP devices and the system

- **DqSystem** oSystem () **const**
- **DqIPDev** oDc () **const**
- **DqIPDev** oAb () **const**
- **DqIPDev** oAu () **const**
- **DqIPDev** oAs () **const**
- **DqIPDev** oAg () **const**
- **DqIPDev** oAp () **const**
- **DqIPDev** oAm (**const int** Mem) **const**

Referencing memory

Memory may be referenced in three ways, by its **DqIPDev**, by its index (The i'th AM device has index i) or by a surface residing in it.

- **DqIPDev** Index2Mem (**const int** Index) **const**

  Convert from a memory index to a **DqIPDev**.

- **int** Mem2Index (**const DqIPDev** oMem) **const**

  Convert from a **DqIPDev** to an index.

- **int** Surf2Index (**const DqSurf** oSurf) **const**

  Convert from a surface to an index.

- **DqIPDev** Surf2Mem (**const DqSurf** oSurf) **const**

  Convert from a surface to a **DqIPDev**.

Entry and exit points on AM devices

These functions return the entry point [AB_OP0 - AB_OP5] and exit points [DQ_IMX0 - DQ_IMX5] of the AM devices.

- **DqEnum** *MEMOUT (const int Index) const*
- **DqEnum** MEMOUT (**const DqIPDev** oMem) **const**
- **DqEnum** SURFOUT (**const DqSurf** oSurf) **const**
- **DqEnum** MEMIN (**const int** Index) **const**
- **DqEnum** MEMIN (**const DqIPDev** oMem) **const**
- **DqEnum** SURFIN (**const DqSurf** oSurf) **const**

Connections to and from memories

These functions provide connections to and from memories. Only if surface is specified it is attached to the appropriate XMT or RCV port. Only if shrinkage or expansion is specified are the shrinkage or expansion on the RCV or XMT ports set.

- **void** CPS2MEM (**const DqIPDev** oMem,
  const int shrinkX, **const int** shrinkY) **const**
- **void** CPS2MEM (**const int** Mem, **const int** shrinkX,
  const int shrinkY) **const**
- **void** CPS2MEM (**const DqIPDev** oMem) **const**

- **void CPS2MEM (const int Mem) const**
- **void CPS2SURF (const DqSurf oSurf,**
  const int shrinkX, const int shrinkY) const
- **void CPS2SURF (const DqSurf oSurf) const**
- **void MEM2CPS (const DqIPDev oMem, const int expX,**
  const int expY) const
- **void MEM2CPS (const int Mem, const int expX,**
  const int expY) const
- **void MEM2CPS (const DqIPDev oMem) const**
- **void MEM2CPS (const int Mem) const**
- **void SURF2CPS (const DqSurf oSurf, const int expX,**
  const int expY) const
- **void SURF2CPS (const DqSurf oSurf) const**
- **void SURF2SURF (const DqSurf oSurfFrom, const int expX,**
  const int expY, const DqSurf oSurfTo,
  const int ShrinkX, const int ShrinkY) const
- **void SURF2SURF (const DqSurf oSurfFrom, const int expX,**
  const int expY, const DqSurf oSurfTo) const
- **void SURF2SURF (const DqSurf oSurfFrom, const DqSurf oSurfTo,**
  const int ShrinkX, const int ShrinkY) const
- **void SURF2SURF (const DqSurf oSurfFrom,**
  const DqSurf oSurfTo) const

Frame Acquisition from the digicolor

Build a path from the ADC on the digicolor to AB's crosspoint switch, a AM device's memory, or a surface. These functions call the above memory connection functions and the same rules about attaching surfaces and setting shrinkage/expansion apply except where the acquisition source surface is concerned. If an acquisition source surface is not supplied on will be created.

Before using any of these paths the digicolors ADC format should be initialized with dcInitADCFormat. These routines return the acquisition source surface.

- **DqSurf DC2CPS (const int num_bands) const**

- **DqSurf** DC2CPS (**const int** num_bands,
  const DqSurf oAcqSrcSurf) **const**
- **DqSurf** DC2MEM (**const DqIPDev** oBand0, **const DqIPDev** oBand1,
  const DqIPDev oBand2) **const**
- **DqSurf** DC2MEM (**const DqIPDev** oBand0, **const DqIPDev** oBand1,
  const DqIPDev oBand2,
  const DqSurf oAcqSrcSurf) **const**
- **DqSurf** DC2MEM (**const int** band0, **const int** band1,
  const int band2) **const**
- **DqSurf** DC2MEM (**const int** band0, **const int** band1,
  const int band2, **const DqSurf** oAcqSrcSurf) **const**
- **DqSurf** DC2SURF (**const DqSurf** oSurf0, **const DqSurf** oSurf1,
  const DqSurf oSurf2) **const**
- **DqSurf** DC2SURF (**const DqSurf** oSurf0, **const DqSurf** oSurf1,
  const DqSurf oSurf2,
  const DqSurf oAcqSrcSurf) **const**
- **DqSurf** DC2MEM (**const DqIPDev** oBand0) **const**
- **DqSurf** DC2MEM (**const DqIPDev** oBand0,
  const DqSurf oAcqSrcSurf) **const**
- **DqSurf** DC2MEM (**const int** band0) **const**
- **DqSurf** DC2MEM (**const int** band0, **const DqSurf** oAcqSrcSurf) **const**
- **DqSurf** DC2SURF (**const DqSurf** oSurf0) **const**
- **DqSurf** DC2SURF (**const DqSurf** oSurf0,
  const DqSurf oAcqSrcSurf) **const**

Displaying memories with the digicolor

Build a path from the AB's crosspoint switch, a AM device's memory, or a surface to the digicolors DAC. These functions call the above memory connection functions and the same rules about attaching surfaces and setting shrinkage/expansion apply except where the display sink and overlay surfaces are concerned. If they are not specified they are created.

Return two surfs, the first is in DC_OVLY_MEM and the second is in DC_DAC The destination of a pipe using this path is DC_DAC Often you want

to clear DC_OVLY_MEM with gsClearView before using the path Init digicolors DAC format before using the path with dcInitDACFormat The array returned is not needed for the pipe to operate, create a pipe ending in DC_DAC, and after you arm it and clear the DC_OVLY_MEM you can delete the array returned by this, then go ahead and fire the pipe. When aoDspSurf is supplied to these functions it is an array containing the DC_DAC and DC_OVLY_MEM surfs

- **DqSurf*** CPS2DC (**const int** NumBands,
  const DqSurf aoDspSurf[2]) **const**
- **DqSurf*** CPS2DC (**const int** NumBands) **const**
- **DqSurf*** MEM2DC (**const DqIPDev** oBand0, **const DqIPDev** oBand1,
  const DqIPDev oBand2) **const**
- **DqSurf*** MEM2DC (**const DqIPDev** oBand0, **const DqIPDev** oBand1,
  const DqIPDev oBand2,
  const DqSurf aoDspSurf[2]) **const**
- **DqSurf*** MEM2DC (**const int** band0, **const int** band1,
  const int band2) **const**
- **DqSurf*** MEM2DC (**const int** band0, **const int** band1,
  const int band2,
  const DqSurf aoDspSurf[2]) **const**
- **DqSurf*** SURF2DC (**const DqSurf** oSurf0, **const DqSurf** oSurf1,
  const DqSurf oSurf2) **const**
- **DqSurf*** SURF2DC (**const DqSurf** oSurf0, **const DqSurf** oSurf1,
  const DqSurf oSurf2,
  const DqSurf aoDspSurf[2]) **const**
- **DqSurf*** MEM2DC (**const DqIPDev** oBand0) **const**
- **DqSurf*** MEM2DC (**const DqIPDev** oBand0,
  const DqSurf aoDspSurf[2]) **const**
- **DqSurf*** MEM2DC (**const int** band0) **const**
- **DqSurf*** MEM2DC (**const int** band0,
  const DqSurf aoDspSurf[2]) **const**
- **DqSurf*** SURF2DC (**const DqSurf** oSurf0) **const**
- **DDqSurf*** SURF2DC (**const DqSurf** oSurf0,
  const DqSurf aoDspSurf[2]) **const**

Simple frame acquisition

Grab color or greyscale frames by using the above path functions and dqUpdateSurf.

- **void** GrabFrame (**const DqSurf** oRed, **const DqSurf** oGreen, **const DqSurf** oBlue, **const DqSurf** oAcqSrcSurf, **const int** Shrinkage = 1, **const DqEnum** eType = DQ_DT_UNSIGNED, **const int** Camera = -1) **const**

- **void** GrabFrame (**const DqSurf** oGrey, **const DqSurf** oAcqSrcSurf, **const int** Shrinkage = 1, **const DqEnum** eType = DQ_DT_UNSIGNED, **const int** Camera = -1) **const**

Simple frame display

Begin a continuous pipe that uses the above path functions to display a color or grey-scale fram.

- **DqPipe** ShowFrame (**const DqSurf** oRed, **const DqSurf** oGreen, **const DqSurf** oBlue,**const DqSurf** aoDspSurf [2], **const int** Expansion = 1) **const**

- **DqPipe** ShowFrame (**const DqSurf** oGrey, **const DqSurf** aoDspSurf [2], **const int** Expansion = 1) **const**

Chapter 5

mres: The Multi-Resolution Image Class

Overview

The mres class uses the DigiColor and MV200 to create a 6 level Gaussian pyramid from a color or greyscale camera. The high resolution level of the Gaussian pyramid is grabbed directly from the camera. Further levels in the pyramid are created by successive blurring and subsampling. The blurring filter is a flat 2×2 averaging filter, but could easily be replaced with any filter of size up to 4×8. The subsampling makes level i half as many points wide and half as many points high as level $i + 1$.

Each level in the Gaussian pyramid covers the entire frame allowing any area in the frame to be accessed at any level of resolution. Since the entire frame is often not needed, at each level of resolution a subregion of the frame, called the active region, can be specified. It is recommended that all routines using the Gaussian pyramid be written to operate only on a levels active region rather than on the entire level. This way the active region provides a uniform mechanism for spatial attention.

Member functions

Display functions

- **void Project () const**

  Project the multires pyramid onto display memory by, starting with the lowest level of resolution, and succesively upsampling the active region and writing it to display memory. If a continuous display pipe from display

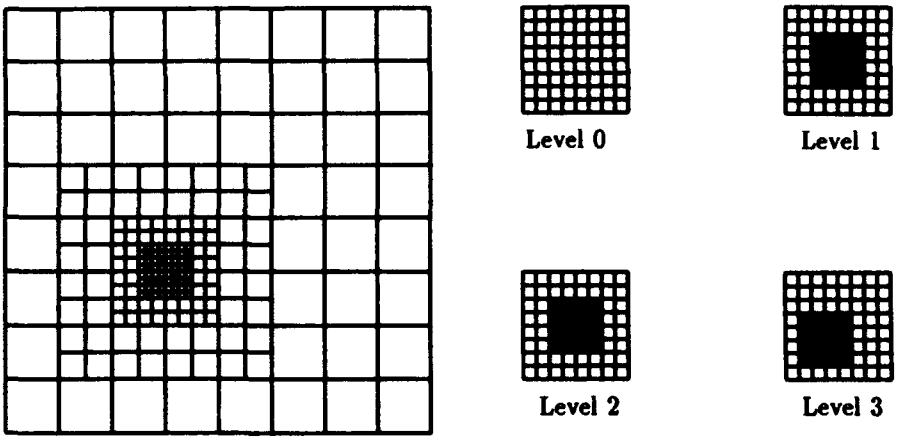

Figure 5.1: Foveated pyramid with 4 levels of resolution. The black regions show the areas covered by a higher level of resolution. Level 0 has the highest resolution.

memory to the DigiColor is not running call StopDisplay to stop any other processes display pipe and begin the display pipe.

- **void HighRes () const**

  Copy the high resolution level of the pyramid to display memory. If a continuous display pipe from display memory to the DigiColor is not running call StopDisplay to stop any other processes display pipe and begin the display pipe.

- **virtual void StopDisplay ()**

  Free the timing bus from the display pipe and mark the display pipe as inactive.

- **void StartDisplay ()**

  If the display pipe is not running fire it.

- **void Foveate (const int x, const int y)**

  Set the active regions of each level to form a foveated system around the coordinate (x, y). The number of levels in the pyramid is determined by SetNumLevels. Each levels active region is a rectangle where level l has a width of $1/2^l$ that of the entire screen and a height of $1/2^l$ of the entire screen. Levels active region is truncated at the edge of the screen. See figure 5.1.

Acquisition of highres and multires images

- **void** RunAcqPat () **const**

  Run the pat that grabs a high resolution frame. This does not create a multiresolution image.

- **void** AcqFrame ()

  Set the ADC format with dcInitADCFormat to that of the current color mode and call RunAcqPat. This is the function that should be used by other classes, it only sets the ADC format if another function has altered it.

- **void** CreateMultiRes ()

  Create a multires image from the image currently in high resolution memory. Call this after calling AcqFrame or after writing a frame into level 0's memory with dqWtRect to create a multiresolution image.

Surface access

The following operators provide access to the active regions of each level.

- **void operator>>** (**xvdq<unsigned char**, 1> axvc1Out []) **const**
- **void operator>>** (**xvdq<unsigned char**, 3> axvc3Out []) **const**
- **void operator<<** (**xvdq<unsigned char**, 1> axvc1In []) **const**
- **void operator<<** (**xvdq<unsigned char**, 3> axvc3In []) **const**
- **void** ReadLevel (**xvdq<unsigned char**, 1>& xvc1Out, **const int** level) **const**
- **void** ReadLevel (**xvdq<unsigned char**, 3>& xvc3Out, **const int** level) **const**
- **void** WriteLevel (**xvdq<unsigned char**, 1>& xvc1In, **const int** level) **const**
- **void** WriteLevel (**xvdq<unsigned char**, 3>& xvc3In, **const int** level) **const**

The following routines provide access to the high resolution image.

- **void operator<<** (**const xvdq<unsigned char**, 1>& xvc1In) **const**
- **void operator<<** (**const xvdq<unsigned char**, 3>& xvc3In) **const**
- **void operator>>** (**xvdq<unsigned char**, 1>& xvc1Out) **const**
- **void operator>>** (**xvdq<unsigned char**, 3>& xvc3Out) **const**

Internals

To access the various bands in different color modes use the enumerated type **ColorIndex**:

- **enum ColorIndex** {
  iError = -1,
  iRED = 0, iGREEN = 1, iBLUE = 2,
  iHUE = 3, iSAT = 4, iVALUE = 5,
  iY = 6, iC = 7,
  iGREY = 8
  }

Key constants are declared in the following enumeration:

- **enum** {
  NumColorModes = 4,
  NumIndices = 9,
  NumMemories = 6,
  MaxNumLevels = 6,
  NumUsedMems = 4
  }

- **virtual void** SetActiveRegion (**const int** Level, **const DqRect** tRect)
  Specify a levels active region by providing a rectangle. This effects which regions of the level will be displayed.
- **virtual void** GetActiveRegion (**const int** level, **DqRect&** tRect) **const**
  Return a levels active region size in a **DqRect**.
- **virtual void** SetColorMode (**const ColorMode** _cmColorMode)
  Specify a color mode to be cmGREY (1 band) or cmRGB (3 bands).
- **ColorMode** cmGetColorMode () **const**
  Return the current color mode.
- **virtual void** SetNumLevels (**const int** _NumLevels)
  Specify the number of levels in the pyramid (maximum of 6).
- **int** GetNumLevels () **const**
  Return the number of levels in the pyramid.

- **DqSurf** oDspSurf (**const int** band) **const**

  Return the surface used for displaying a given band.

- **DqSurf** oProjSurf (**const int** band, **const int** level) **const**

  Return a surface that is a duplicate of the display surface returned in oDspSurf, but with a different processing rectangle. The rectangle in the returned surface will cover the area that the given level's active region will be displayed in.

- **DqSurf** oLevel (**const int** band, **const int** level) **const**

  Return the surface for a given level and a given band in the current color mode.

- **DqSurf&** oLevel (**const ColorIndex** i, **const int** level)

  Return the surface for a given level and a given band in the current color mode.

- **DqSurf** oActiveRegion (**const int** band, **const int** level) **const**

  Return the surface for a given levels active region in a given band for the current color mode.

- **DqSurf&** oActiveRegion (**const ColorIndex** i, **const int** level)

  Return the surface for a given levels active region in a given band for the current color mode.

Compiling

It is recommended that routines that use the mres class be written as a class that virtually publically inherits mres.

Memory

The Gaussian pyramid uses 4 memories on the MV200. Three for a 24 bit RGB pyramid, and one for an 8 bit greyscale pyramid. The Greyscale and RGB levels in the pyramid do not overlap so color acquisition will not destroy an old greyscale frame and vice versa. See the SetColorMode function for information on switching between RGB and greyscale acquisition.

The memory for the surfaces in this class is distributed as follows: For each level the red, green, blue, and grey bands all occupy memories on different AM devices. Each bands even levels are stored on one AM device while the bands odd levels are stored on another AM device, see figure 5.2. On the 4 AM devices the levels are contained within a rectangle of size 768 × 480 where level i has

Band	Even levels	Odd levels
Red	AM0	AM1
Green	AM1	AM2
Blue	AM2	AM3
Grey	AM3	AM0

Figure 5.2: Storage devices for levels in pyramid

width $512/2^i$ and height $480/2^i$. The levels distribution within this 768 × 480 rectangle is shown in figure 5.3.

The 768 × 480 rectangle for storage of the levels has been allocated with dq_global::allocate. Further allocations made with dq_global::allocate will not overlap this rectangle (see the section on dq_global).

The display memory is a 512 × 480 rectangle in AM4. When a foveated image is displayed each level's active region is copied to display memory and a continuous pipe out of display memory to the DigiColor's DAC displays the memory's contents. This is only a greyscale display and when the color mode is set to RGB only the green band is displayed.

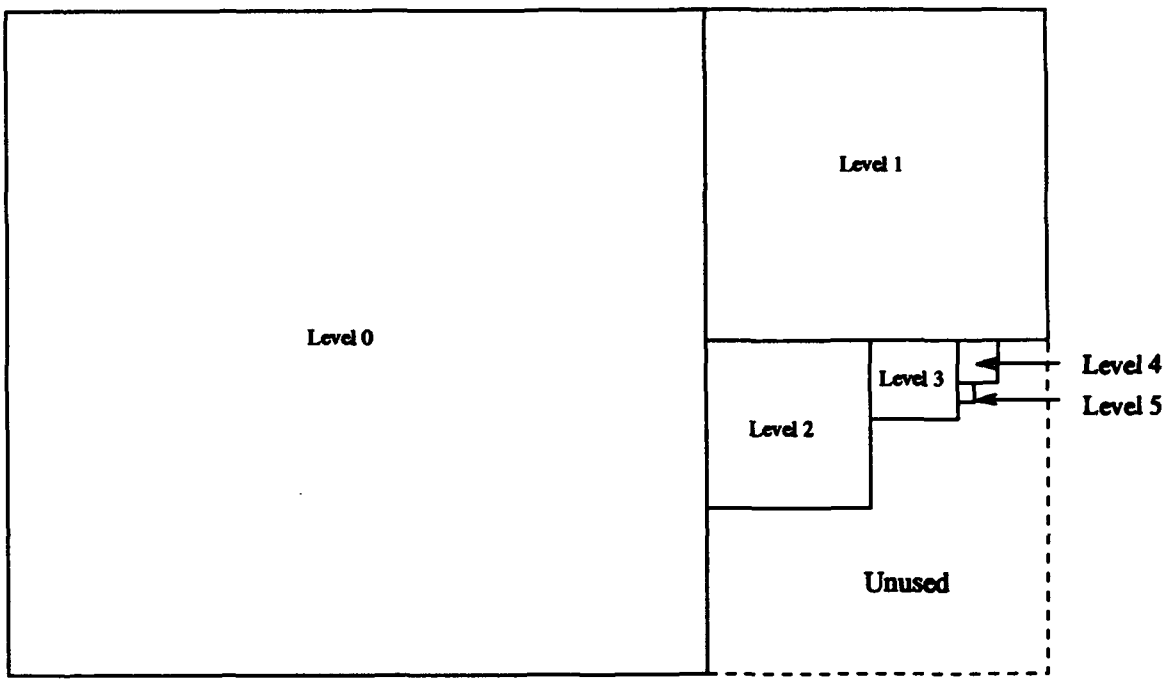

Figure 5.3: Location of levels in an AM device