Multipipe SDK 3.2 Reference

Name

`MPKArena` - MPKArena functional interface.

Header File

#include <mpk/arena.h>

Synopsis

void* **mpkMalloc**(size_t *size*);
void **mpkFree**(void* *ptr*);
void* **mpkCalloc**(size_t *nelem*, size_t *elsize*);
void* **mpkRealloc**(void* *ptr*, size_t *size*);
char* **mpkStrDup**(const char* *s1*);

Description

The MPKArena functional interface provides a simple memory allocation package that enables OpenGL Multipipe SDK applications to allocate data regardless of their current execution mode (see **`MPKGlobal`**).

Function descriptions

mpkMalloc returns a pointer to a block of at least *`size`* bytes suitably aligned for any use.

The argument to **mpkFree** is a pointer to a block previously allocated by **mpkMalloc**, **mpkCalloc** or **mpkStrDup**; after **mpkFree** is performed this space is made available for further allocation, but its contents are left undisturbed.

mpkCalloc allocates space for an array of *`nelem`* elements of size *`elsize`*. The space is initialized to zeros.

mpkRealloc changes the size of the block pointed to by *`ptr`* to *`size`* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

mpkStrDup returns a pointer to a new string which is a duplicate of the string pointed to by s1. The space for the new string is obtained using **mpkMalloc**. If the new string can not be created, it returns NULL.

See also

`MPKGlobal`

Multipipe SDK 3.2 Reference

Name

MPKChannel - MPKChannel functional interface.

Header File

#include <mpk/channel.h>

Synopsis

Creating and Destroying

MPKChannel*	**mpkChannelNew**(void);
void	**mpkChannelDelete**(MPKChannel* *channel*);

Fields Access

void	**mpkChannelSetName**(MPKChannel* *c*, char* *name*);
const char*	**mpkChannelGetName**(MPKChannel* *c*);
void	**mpkChannelSetUserData**(MPKChannel* *channel*, void* *userData*);
void*	**mpkChannelGetUserData**(MPKChannel* *c*);
void	**mpkChannelSetViewport**(MPKChannel* *c*, float *vp[4]*);
void	**mpkChannelGetViewport**(MPKChannel* *c*, float *vp[4]*);
void	**mpkChannelSetNearFar**(MPKChannel* *c*, float *n*, float *f*);
void	**mpkChannelGetNearFar**(MPKChannel* *c*, float* *n*, float* *f*);
MPKConfig*	**mpkChannelGetConfig**(MPKChannel* *c*);
MPKPipe*	**mpkChannelGetPipe**(MPKChannel* *c*);
MPKWindow*	**mpkChannelGetWindow**(MPKChannel* *c*);
MPKChannel*	**mpkChannelGetProxy**(MPKChannel* *c*);

void	**mpkChannelSetProjection**(MPKChannel* *c*, const float* *origin*, float *distance*, const float* *fov*, const float* *hpr*);
int	**mpkChannelGetProjection**(MPKChannel* *c*, float* *origin*, float* *d*, float* *fov*, float* *hpr*);
void	**mpkChannelSetWall**(MPKChannel* *c*, const float *bl[3]*, const float *br[3]*, const float *tl[3]*);
int	**mpkChannelGetWall**(MPKChannel* *c*, float* *bl*, float* *br*, float* *tl*);
void	**mpkChannelSetOrthoWall**(MPKChannel* *c*, const float *bl[3]*, const float *br[3]*, const float *tl[3]*);
int	**mpkChannelGetOrthoWall**(MPKChannel* *c*, float* *bl*, float* *br*, float* *tl*);

Attributes

void	**mpkChannelSetAttribute**(MPKChannel* *channel*, int *attr*, int *value*);
void	**mpkChannelUnsetAttribute**(MPKChannel* *channel*, int *attr*);
void	**mpkChannelResetAttribute**(MPKChannel* *channel*, int *attr*);
int	**mpkChannelTestAttribute**(MPKChannel* *channel*, int *attr*);

int **mpkChannelGetAttribute**(MPKChannel* *channel*, int *attr*, int* *value*);

Callbacks

void **mpkChannelSetDrawCB**(MPKChannel* *c*, int *which*, MPKChannelDrawCB *cb*);

MPKChannelDrawCB **mpkChannelGetDrawCB**(MPKChannel* *c*, int *which*);

void **mpkChannelSetCullCB**(MPKChannel* *c*, int *which*, MPKChannelCullCB *cb*);

MPKChannelCullCB **mpkChannelGetCullCB**(MPKChannel* *c*, int *which*);

Operations

void **mpkChannelApplyBuffer**(MPKChannel* *c*);

void **mpkChannelApplyViewport**(MPKChannel* *c*);

void **mpkChannelApplyNearFar**(MPKChannel* *c*, float *n*, float *f*);

void **mpkChannelApplyFrustum**(MPKChannel* *c*);

void **mpkChannelApplyHeadTransform**(MPKChannel* *c*);

void **mpkChannelApplyViewTransform**(MPKChannel* *c*);

void **mpkChannelApplyOrtho**(MPKChannel* *c*, int *orthomode*, const float *zoom[2]*);

void **mpkChannelGetOrtho**(MPKChannel* *c*, int *orthomode*, const float *zoom[2]*, float *ortho[6]*, float *xform[16]*);

void **mpkChannelGetFrustum**(MPKChannel* *c*, int *eye*, float *frust[6]*, float *xform[16]*);

int **mpkChannelGetEye**(MPKChannel* *c*);

void **mpkChannelGetPixelViewport**(MPKChannel* *c*, int* *pvp*);

void **mpkChannelUpdatePixelViewport**(MPKChannel* *c*);

int **mpkChannelGetRange**(MPKChannel* *c*, float* *range*);

Custom Assembly

void **mpkChannelPushGLState**(MPKChannel* *c*, int *state*);

void **mpkChannelPopGLState**(MPKChannel* *c*);

void **mpkChannelAssembleFrame**(MPKChannel* *c*, MPKFrame* *frame*);

void **mpkChannelDrawImage**(MPKChannel* *channel*, MPKImage* *image*, float *region[4]*);

Adaptive Readback

void **mpkChannelDeclareROI**(MPKChannel* *c*, float *region[4]*);

void **mpkChannelReadFrame**(MPKChannel* *c*, MPKFrame* *frame*, float *region[4]*);

Performer Integration

pfChannel* **mpkChannelGetPfChannel**(MPKChannel* *c*);

Culling

void* **mpkChannelNextData**(MPKChannel* *channel*);
int **mpkChannelCheckData**(MPKChannel* *channel*);
void **mpkChannelPassData**(MPKChannel* *channel*, void* *data*);
void **mpkChannelFlushData**(MPKChannel* *channel*);
void **mpkChannelPutData**(MPKChannel* *channel*, void* *data*);

Description

The MPKChannel data structure essentially describes a viewport in an **`MPKWindow`**. The corresponding projection rectangle can be specified in real-world coordinates via **mpkChannelSetWall** or **mpkChannelSetProjection**. The application can then be written independently from the current stereo pass, from the viewer's head-position and from the decomposition specified by the current **`MPKCompound`**.

This genericity is illustrated by the typical code below :

```
main( int argc, char *argv[] )
{
    ...
    mpkConfigSetChannelInitCB( config, initChannel );
    ...
    mpkConfigInit( config );
    while ( !exit ) {
        ...
        mpkConfigSetHeadPosition( config, head.position );
        mpkConfigFrame( config, framedata );
    }
}

void initChannel( MPKChannel *c )
{
    mpkChannelSetDrawCB(c, MPK_CHANNEL_DRAWCB_INIT, loadChannel);
    mpkChannelSetDrawCB(c, MPK_CHANNEL_DRAWCB_CLEAR, clearChannel);
    mpkChannelSetDrawCB(c, MPK_CHANNEL_DRAWCB_UPDATE, updateChannel);
    mpkChannelSetNearFar( c, 0.001, 100.0 );
}

void loadChannel( MPKChannel *c, void *framedata )
{
    myLoadTextures( framedata );
}

void clearChannel( MPKChannel *c, void *framedata )
{
    mpkChannelApplyBuffer(c);
    mpkChannelApplyViewport(c);

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
}

void updateChannel( MPKChannel *c, void *framedata )
{
    glMatrixMode( GL_PROJECTION );
```

```
        glLoadIdentity();
        mpkChannelApplyFrustum(c);

        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity();
        mpkChannelApplyHeadTransform(c);

        myDrawData( framedata );
    }
```

Function descriptions

Creating and Destroying

mpkChannelNew creates and returns a handle to an MPKChannel.

mpkChannelDelete deletes the passed MPKChannel.

Fields Access

mpkChannelSetName sets the name of the passed MPKChannel to *`name`*. This is done by copy and not by reference.

mpkChannelGetName returns the name of the passed MPKChannel.

mpkChannelSetUserData enables the application to specify passthrough data to be transported within the *`channel`* structure. Transport is done by reference and not by copy.

mpkChannelGetUserData enables the application to retrieve the passthrough data specified by mpkChannelSetUserData().

mpkChannelSetViewport sets the fractional viewport values of the passed MPKChannel to the values pointed to by *`vp`*.

mpkChannelGetViewport reads the fractional viewport values of the passed MPKChannel in *`vp`*.

mpkChannelSetNearFar sets the near and far distances of the passed MPKChannel's frustum to the arguments *`n`* and *`f`*.

mpkChannelGetNearFar reads the near and far distances of the passed MPKChannel's frustum.

mpkChannelGetConfig returns the MPKChannel's parent configuration.

mpkChannelGetPipe returns the MPKChannel's parent pipe.

mpkChannelGetWindow returns the MPKChannel's parent window.

mpkChannelGetProxy returns the MPKChannel's Xinerama meta channel, or NULL if this channel in not a channel of a Xinerama base window.

mpkChannelSetProjection sets the MPKChannel's physical layout.

Here the channel is assimilated to the rectangle which would be produced by a hypothetical projection system located at *`origin`*, in the attitude characterized by the *`hpr`* angles, and projecting orthogonally onto a wall situated at *`distance`*. The horizontal and vertical fields of view of this projector are specified by the argument *`fov`*.

```
    projection {
        origin      [ 0., 0., 0. ]
        distance    3.
        fov         [ 54., 47. ]
        hpr         [ 0., 0., 0. ]   # CENTRE WALL
        # hpr       [  50., 0., 0. ] # LEFT WALL
        # hpr       [ -50., 0., 0. ] # RIGHT WALL
    }
```

mpkChannelApplyFrustum() will use the last specified wall or projection description to compute its OpenGL frustum and modeling transformation.

mpkChannelGetProjection enables the application to retrieve the data specified by mpkChannelSetProjection().

mpkChannelSetWall describes the physical layout of the passed MPKChannel providing the real-world coordinates of the bottom-left, bottom-right and top-left corners of its projection rectangle.

```
     # 80cm x 60cm screen located 1m in front of the viewer

     wall {
         bottom_left     [ -.4, -.3, -1. ]
         bottom_right    [  .4, -.3, -1. ]
         top_left        [ -.4,  .3, -1. ]
     }
```

mpkChannelApplyFrustum() will use the last specified wall or projection description to compute its OpenGL frustum and modeling transformation.

mpkChannelGetWall enables the application to retrieve the data specified by mpkChannelSetWall().

mpkChannelSetOrthoWall specifies an alternate wall description, to be used by mpkChannelApplyOrtho() to compute its OpenGL ortho and modeling transformation.

mpkChannelGetOrthoWall enables the application to retrieve the data specified by mpkChannelSetOrthoWall().

Attributes

See the **`MPKGlobal`** man page for a description of all MPKChannel attributes and their default or possible values.

mpkChannelSetAttribute sets the value of the MPKChannel attribute specified by *`attr`* to *`value`*.

mpkChannelUnsetAttribute unsets the attribute specified by *`attr`* or, if *`attr`* is `MPK_CATTR_ALL`, unsets all attributes for the passed MPKChannel.

mpkChannelResetAttribute resets the attribute specified by *`attr`* to its corresponding default value or, if *`attr`* is `MPK_CATTR_ALL`, it resets all attributes for the passed MPKChannel to their default value.

mpkChannelTestAttribute returns 1 if the attribute specified by *attr* is set for the passed MPKChannel, 0 otherwise.

mpkChannelGetAttribute reads the current value of the attribute specified by *attr* and returns 1 if the attribute is set for the passed MPKChannel, 0 otherwise.

Callbacks

mpkChannelSetDrawCB sets the MPKChannel draw callback specified by *which* to the passed function, of type:

void (***MPKChannelDrawCB**)(MPKChannel*, void*);

Accepted values for *which* are

MPK_CHANNEL_DRAWCB_INIT, **MPK_CHANNEL_DRAWCB_CLEAR** or
MPK_CHANNEL_DRAWCB_UPDATE

mpkChannelGetDrawCB returns the MPKWindow draw callback function specified by *which*.

mpkChannelSetCullCB sets the MPKChannel cull callback specified by *which* to the passed function, of type :

void (***MPKChannelCullCB**)(MPKChannel*, void*);

Accepted values for *which* are

MPK_CHANNEL_CULLCB_INIT, **MPK_CHANNEL_CULLCB_UPDATE**

mpkChannelGetCullCB returns the MPKWindow cull callback function specified by *which*.

Operations

mpkChannelApplyBuffer applies the current `GL_DRAW_BUFFER` and `GL_READ_BUFFER` for the passed MPKChannel with respect to the current stereo mode and eye pass.

mpkChannelApplyViewport applies the current OpenGL viewport and scissor area for the passed MPKChannel. The channel's pixel viewport is computed from the parent window's pixel viewport (ie. width and height) and channel's fractional viewport using the formula:

```
#define IRND(a) ((int)((a)+.5))

// compute first pixel position of the channel
channel.pvp[0] = IRND(channel.vp[0] * window.pvp[2]);
channel.pvp[1] = IRND(channel.vp[1] * window.pvp[3]);

// compute last pixel position of the channel
channel.pvp[2] = IRND((channel.vp[0]+channel.vp[2]) * window.pvp[2]);
channel.pvp[3] = IRND((channel.vp[1]+channel.vp[3]) * window.pvp[3]);

// compute channel's dimension
channel.pvp[2] -= channel.pvp[0];
channel.pvp[3] -= channel.pvp[1];
```

This method honors positions over dimensions in order to ensure adjacency whenever possible, e.g. in a 1280x1024 window :

```
vp(1): [0.     0. 0.3333 1. ]    pvp(1): [0   0 427 1024]
vp(2): [0.3333 0. 0.3333 1. ]    pvp(2): [427 0 426 1024]
```

Note that in full-screen stereo mode (type "rect") during the left eye pass the value of the `MPKGlobal` variable MPK_DATTR_FULLSTEREO_OFFSET will be added to channel.pvp[1]

mpkChannelApplyNearFar applies the *`n`* and *`f`* distances used for the following frustum or ortho operations. This function can be used to dynamically adjust the near and far distance from the channel update callbacks.

mpkChannelApplyFrustum applies an OpenGL frustum matrix for the passed MPKChannel with respect to the current eye pass, eye position and latest layout from mpkChannelSetWall() or mpkChannelSetProjection().

mpkChannelApplyHeadTransform applies the modeling transformation needed to position and orient the viewing pyramid specified by the latest call to mpkChannelApplyFrustum() or mpkChannelApplyOrtho(). mpkChannelApplyHeadTransform replaces the deprecated function mpkChannelApplyTransformation().

mpkChannelApplyViewTransform applies the view transformation needed to position and orient the viewer in the current scene.

mpkChannelApplyOrtho provides an alternative to mpkChannelApplyFrustum() as it applies an OpenGL orthographic matrix for the passed MPKChannel.

mpkChannelApplyOrtho uses the layout given by mpkChannelSetOrthoWall() if one has been, otherwise it will use the latest layout specified via mpkChannelSetWall() or mpkChannelSetProjection().

If *`orthomode`* is MPK_ORTHO_STILL, then mpkChannelApplyOrtho simply uses the half-width and half-height dimensions of the channel layout to produce the distances used in glOrtho(3G).

Otherwise, if *`orthomode`* is MPK_ORTHO_TRACKED, then mpkChannelApplyOrtho uses the current view direction (e.g. from mpkConfigSetHeadOrientation) in order to produce consistent viewing across all the config's channels.

The argument *`zoom`*, if not NULL, specifies two-dimensional scaling on the X and Y Screen coordinates.

mpkChannelGetOrtho retrieves the MPKChannel's complete orthographic transformation, ie. so that the following code sequence :

```
mpkChannelGetOrtho(c, orthomode, zoom, ortho, xform);
glOrtho( ortho[0], ortho[1], ortho[2], ortho[3], ortho[4], ortho[5] );
glMultMatrixf( xform );
```

is completely equivalent to :

```
mpkChannelApplyOrtho(c, orthomode, zoom);
mpkChannelApplyHeadTransform(c);
```

mpkChannelGetFrustum retrieves the MPKChannel's complete frustum transformation, ie. so that the following code sequence :

```
mpkChannelGetFrustum(c, mpkChannelGetEye(), frust, xform);
glFrustum(frust[0], frust[1], frust[2], frust[3], frust[4], frust[5]);
glMultMatrixf( xform );
```

is completely equivalent to :

```
mpkChannelApplyFrustum(c);
mpkChannelApplyHeadTransform(c);
```

mpkChannelGetEye returns the current eye pass of the channel update, ie. returns `MPK_EYE_CYCLOP` in mono mode, `MPK_EYE_LEFT` or `MPK_EYE_RIGHT` if the channel is rendered in stereo mode.

mpkChannelGetPixelViewport reads the latest updated pixel viewport for the passed MPKChannel in *`pvp`*.

mpkChannelUpdatePixelViewport forces update of the MPKChannel's pixel viewport from its parent window's pixel viewport (ie. width and height) and channel's fractional viewport. See above mpkChannelApplyViewport() for how these computations are done.

Note that in full-screen stereo mode (type "rect") during the left eye pass the value of the `MPKGlobal` variable MPK_DATTR_FULLSTEREO_OFFSET will be added to the vertical pixel viewport coordinate.

mpkChannelGetRange reads the *`range`* values of the passed MPKChannel if this is relevant, returning 0 otherwise. The range of a MPKChannel is inherited at rendering time from the currently overriding `MPKCompound`. This information should then be used to render only a portion of the database, e.g. :

```
mpkChannelGetRange(c,range) )
DrawDatabase( range[0], range[1] );
```

Note that the range of an MPKCompound is specified relatively to its hierarchy, whereas the MPKChannel inherits absolute values.

Custom Assembly

mpkChannelPushGLState pushes the current OpenGL matrices and attributes to the stack and set's up the state to perform one of the following operations:

```
MPK_PIXEL_OPERATION     sets up the projection matrix to
                        perform drawing of input frames
```

mpkChannelPopGLState pops the last OpenGL matrices and attributes from the stack.

mpkChannelAssembleFrame assembles a frame into the specified MPKChannel, as described in the pseudo code below.

```
if frame format has depth bit set
    enable depth test
    draw all depth images using mpkChannelDrawImage()
    disable depth test

if frame format has color bit set
    draw all color images using mpkChannelDrawImage()

if frame format has stencil bit set
    draw all stencil images using mpkChannelDrawImage()
```

mpkChannelDrawImage draw the *image* into the *region* of *channel*.

Adaptive Readback

mpkChannelDeclareROI declares the region which was updated during the current channel update draw callback. Among other optimisations, the default compound read output callback mpkCompoundReadOutputFrame() will only read the specified region. The *region* is interpreted as (x, y, width, height), and the area it describes has to be clipped to [0, 0] - [1, 1].

mpkChannelDeclareROI should only be called from the channel update draw callback.

mpkChannelReadFrame read the channel's framebuffer content into *frame*.

frame can be obtained by using the function mpkCompoundGetOutputFrame(). *region* specifies the 2D fractional viewport to be read into *frame* with respect to the channel *c* viewport.

The function reallocates new images buffers in the passed *frame* if they are not big enough for reading the region.

Performer Integration

mpkChannelGetPfChannel returns the Performer channel used internally by the MPKChannel whenever the execution mode MPK_EXECUTION_PERFORMER is used.

Culling

mpkChannelNextData returns the next item from the input queue. The items received from mpkChannelNextData are either produced by mpkConfigFrameData(), mpkChannelPassData() or mpkChannelPutData(), depending on how *`channel`* is involved in the configuration. Please see the `MPKCompound` documentation for further explanations on culling.

This function should only be called from the update cull or update draw callback.

mpkChannelCheckData returns a positive value if items are available on the input queue, 0 otherwise.

This function should only be called from the update cull or update draw callback.

mpkChannelPassData adds an item to the output queue.

Latency-correct memory management for *`data`* can be done via frame data referenciation and dereferenciation callbacks, to be specified prior to mpkConfigInit() via MPKConfigSetFrameDataRefCB() and MPKConfigSetFrameDataUnrefCB().

This function should only be called from the update cull callback.

mpkChannelFlushData forces a flush of the input buffer to the output queue, filled using the function mpkChannelPassData(). The size of the buffer can be set using mpkGlobalSetAttributei() for attribute MPK_CHANNEL_PASS_CACHE_SIZE prior mpkChannelNew().

This function should only be called from the update cull callback.

mpkChannelPutData adds an item to the input queue. This function can be used to put back items to the input queue, in order to enable parallelization between multiple cull processes. The following pseudo-code illustrates one usage:

```
    while( data = mpkChannelNextData( channel ))
    {
        switch( visiblity( frustum, data ))
        {
            case FULL_VISIBLE:    // draw whole tree
                mpkChannelPassData( channel, data );
                break;

            case PARTIAL_VISIBLE: // re-test each subtree
                foreach child of data
                    mpkChannelPutData( channel, child );
                break;

            case NOT_VISIBLE:     // discard
                break;
        }
    }
```

Latency-correct memory management for *`data`* can be done via frame data referenciation and dereferenciation callbacks, to be specified prior to mpkConfigInit() via MPKConfigSetFrameDataRefCB() and MPKConfigSetFrameDataUnrefCB().

This function should only be called from the update cull callback.

File Format/Defaults

1. MPKChannel File Format specification :

```
channel {
                                   # channel FIELDS description

                         name       "channel-name"
                         viewport   [ xf, yf, wf, hf ]

                                   # channel WALL or PROJECTION description

                         wall       { channel-wall description }
                         projection { channel-projection description }

                                   # channel ORTHO-WALL description

                         ortho-wall { channel-ortho-wall description }

}
```

2. MPKChannel-wall File Format specification :

```
wall {
                                        bottom_left   [ x, y, z ]
                                        bottom_right  [ x, y, z ]
                                        top_left      [ x, y, z ]

}
```

3. MPKChannel-projection File Format specification :

```
projection {
                                     origin    [ x, y, z ]
                                     distance  value
                                     fov       [ horizontal, vertical ]
                                     hpr       [ head, pitch, roll ]

}
```

4. MPKChannel-ortho-wall File Format specification :

```
ortho-wall {
                                        bottom_left   [ x, y, z ]
                                        bottom_right  [ x, y, z ]
```

top_left **[** x, y, z **]**

}

Notes

`viewport` parameters are relative to the parent window size, and therefore their values should be in the range 0.0 to 1.0

See also

`MPKCompound`, `MPKGlobal`, `MPKWindow`

Multipipe SDK 3.2 Reference

Name

`MPKCompound` - MPKCompound functional interface.

Header File

#include <mpk/compound.h>

Synopsis

Creating and Destroying

MPKCompound*	**mpkCompoundNew**(void);
void	**mpkCompoundDelete**(MPKCompound* *compound*);

Traversal

void **mpkCompoundTraverseAll**(MPKCompound* *compound*, MPKCompoundCB *preCB*, MPKCompoundCB *leafCB*, MPKCompoundCB *postCB*, void* *userdata*);

void **mpkCompoundTraverseActive**(MPKCompound* *compound*, MPKCompoundCB *preCB*, MPKCompoundCB *leafCB*, MPKCompoundCB *postCB*, void* *data*);

void **mpkCompoundTraverseCurrent**(MPKCompound* *compound*, MPKCompoundCB *preCB*, MPKCompoundCB *leafCB*, MPKCompoundCB *postCB*, void* *data*);

Fields Access

void	**mpkCompoundSetMode**(MPKCompound* *compound*, int *mode*, int *flags*);
void	**mpkCompoundGetMode**(MPKCompound* *compound*, int* *mode*, int* *flags*);
void	**mpkCompoundSetOperation**(MPKCompound* *compound*, int *operation*);
int	**mpkCompoundGetOperation**(MPKCompound* *compound*);
void	**mpkCompoundSetName**(MPKCompound* *compound*, char* *name*);
const char*	**mpkCompoundGetName**(MPKCompound* *compound*);
void	**mpkCompoundSetSplit**(MPKCompound* *compound*, char* *split*);
const char*	**mpkCompoundGetSplit**(MPKCompound* *compound*);
void	**mpkCompoundSetDisplayName**(MPKCompound* *compound*, char* *name*);
const char*	**mpkCompoundGetDisplayName**(MPKCompound* *compound*);
void	**mpkCompoundSetChannel**(MPKCompound* *compound*, MPKChannel* *c*);
MPKChannel*	**mpkCompoundGetChannel**(MPKCompound* *compound*);
void	**mpkCompoundSetViewport**(MPKCompound* *compound*, float* *vp*);
void	**mpkCompoundGetViewport**(MPKCompound* *compound*, float* *vp*);
void	**mpkCompoundSetEye**(MPKCompound* *compound*, int *eye*);
int	**mpkCompoundGetEye**(MPKCompound* *compound*);
void	**mpkCompoundSetFormat**(MPKCompound* *compound*, int *format*);

```
int            mpkCompoundGetFormat(MPKCompound* compound);
void           mpkCompoundSetRange(MPKCompound* compound, float range[2]);
void           mpkCompoundGetRange(MPKCompound* compound, float range[2]);
int            mpkCompoundNChildren(MPKCompound* c);
MPKCompound* mpkCompoundGetChild(MPKCompound* c, int i);
void           mpkCompoundAddChild(MPKCompound* compound, MPKCompound* child);
int            mpkCompoundRemoveChild(MPKCompound* c, MPKCompound* child);
MPKCompound* mpkCompoundGetNext(MPKCompound* c);
MPKCompound* mpkCompoundGetParent(MPKCompound* c);
MPKConfig*     mpkCompoundGetConfig(MPKCompound* c);
MPKCompound* mpkCompoundFindChild(MPKCompound* c, const char* name);
void           mpkCompoundSetUserData(MPKCompound* compound, void* userData);
void*          mpkCompoundGetUserData(MPKCompound* c);
```

Custom Compound Interface

```
MPKFrame* mpkCompoundGetAssemblyFrame(MPKCompound* compound, int i);
void        mpkCompoundPreAssemble(MPKCompound* compound, void* data);
void        mpkCompoundPostAssemble(MPKCompound* compound, void* data);
```

Adaptive Readback Interface

```
MPKFrame* mpkCompoundGetOutputFrame(MPKCompound* compound);
void        mpkCompoundReadOutputFrame(MPKCompound* compound);
```

Custom Compound Clear Interface

```
void mpkCompoundClear(MPKCompound* compound, void* data);
```

Description

The MPKCompound data structure is essentially a container for children of MPKCompound, each associated with an existing **MPKChannel**. The rendering of the top-most MPKChannel in the hierarchy will be parallelized among the child channels, by either:

- portions of the destination viewport (mode **2D**)
- multipass rendering (mode **FSAA**)
- portions of the frame data (mode **3D** or **DB**)
- stereo eye pass (mode **EYE** or **HMD**)
- pipelined rendering cycles (mode **DPLEX**)
- separating the cull and draw operation (mode **CULL**)

Recomposition of the destination channel's image is done automatically, and can be customised using the Custom Assembly interface.

Drawing and Culling

The operation field of the MPKCompound data structure defines what tasks are executed. Given the following compound specification:

```
compound {
    mode     [ CULL ]
    channel  "dest"

    region { cull channel "cull" }
    region { draw channel "dest" }
}
```

MPK would invoke the associated update cull callback on the channel "cull", and the update draw callback on channel "dest". These two operations are executed in parallel, if the two channels are associated to two different rendering threads.

The default operation is cull-draw, ie. MPK invokes first the update cull callback, and then the update draw callback on each leaf node.

The operation mode can be set using mpkCompoundSetOperation() The functions mpkConfigFrameData(), mpkChannelNextData(), mpkChannelCheckData(), mpkChannelPassData() and mpkChannelPutData() provide a generic mechanism to pass data through the application.

In the specification above, data passed from the update cull callback of the channel "cull" will be available in the update draw callback of channel "draw". Likewise, the data available to the update cull callback of channel "cull" originates from the application.

Custom Assembly

The purpose of the MPK custom compound interface is to allow customization of the MPK compound pre- and post-assembly pass.

Given the following compound specification:

```
compound {

    mode [2D]
    channel "destination"

    region {
        viewport [ 0. 0. .5 1. ]
        channel  "source::0"
    }

    region {
        viewport [ .5 0. .5 1. ]
        channel  "source::1"
    }
 }
```

This compound specification corresponds to the following tree:

```
                "destination"
                      o
                      |
           +----------+----------+
           |                     |
           |                     |
           o                     o
       "source::0"           "source::1"
```

Upon mpkConfigFrameBegin() each MPK window thread traverses the config's compound tree[s] and updates each compound whose channel belongs to the window: if the compound is a leaf node then MPK invokes the user-specified clear and update callbacks on the associated [source] channel. Otherwise, ie. if the compound is not a leaf node, then MPK assembles the frames output from the compound children into its associated [destination] channel.

This traversal actually occurs in several passes:

1. clear all source channels
2. invoke pre-assemble callback on all destination channels using top-bottom, left-right traversal order. Example of pre-assemble callback is the above 2D-decomposition in mode ASYNC: then the images output from the source channels during last frame are assembled in the destination channel prior to any rendering.
3. update source channels (user-specified clear and update CB)
4. invoke post-assemble callbacks on all destination channels using bottom-up, left-right traversal order. Example of post-assemble callback is the above 2D-decomposition in mode not ASYNC: then the images from the source channels are assembled in the destination channel during the same frame, as soon as they have been produced.

Neither the pre- nor post- assemble callback are invoked if any of the following conditions are true:

- compound channel is NULL.
- compound channel's window is frozen.
- compound latency is greater than current frame number (countdown).

The default pre- and post-assemble callbacks do not perform any assembly if the following conditions are true:

- compound mode is NOCOPY.
- compound number of assembly frame is 0.

If the compound channel is identical to its parent channel then no output frame is generated for this compound.

The functions mpkConfigSetCompoundPreAssembleCB() and mpkConfigSetCompoundPostAssembleCB() are used to specify the pre- or post- assemble compound callbacks.

Adaptive readback

This interface enables the programmer to customize the frame buffer readback of a compound input channel. The default callback, mpkCompoundReadOutputFrame(), uses the information provided by mpkChannelDeclareROI() to optimize the data to be read back, transported and assembled during compound operations, using mpkChannelReadFrame() on the handle returned by mpkCompoundGetOutputFrame().

Custom Compound Clear

The compound clear callback is invoked on all destination compounds. The default callback, mpkCompoundClear(), only clears the framebuffer in special cases for example, when the adaptive readback callback is used. This callback should only be customized if really necessary since mpkCompoundClear() optimizes the clear as extensively as possible. The compound clear callback can be specified using the function mpkConfigSetCompoundClearCB().

Function descriptions

Creating and Destroying

mpkCompoundNew creates and returns a handle to an MPKCompound.

mpkCompoundDelete deletes the pass MPKCompound.

Traversal

The following functions provide a general mechanism for traversing a compound hierarchy in a top-to-bottom, left-to-right order.

```
typedef int (*MPKCompoundCB)( MPcompound *, void *userdata );

void mpkCompoundTraverseAll(MPKCompound *, MPKCompoundCB preCB,
        MPKCompoundCB leafCB, MPKCompoundCB postCB, void *userdata );
void mpkCompoundTraverseActive(MPKCompound *, MPKCompoundCB preCB,
        MPKCompoundCB leafCB, MPKCompoundCB postCB, void *userdata );
void mpkCompoundTraverseCurrent(MPKCompound *, MPKCompoundCB preCB,
        MPKCompoundCB leafCB, MPKCompoundCB postCB, void *userdata );
```

The *leafCB* function will be applied only on the leaf nodes of the compound tree, ie. on compounds without any children. The *preCB* and *postCB* functions will be applied when traversing parent compounds.

The return values must be either `MPK_TRAV_CONT`, `MPK_TRAV_PRUNE` or `MPK_TRAV_TERM` to indicate that the traversal should continue, skip this node or terminate, respectively. `MPK_TRAV_PRUNE` is equivalent to `MPK_TRAV_CONT` for the *postCB* function.

mpkCompoundTraverseAll will traverse all children of the specified MPKCompound, regardless of any state information.

mpkCompoundTraverseActive will only traverse the active children of the passed MPKCompound with respect to the current stereo mode.

mpkCompoundTraverseCurrent will only traverse the active and current children of the passed MPKCompound with respect to the current stereo mode and current `DPLEX` cycle.

Fields Access

mpkCompoundSetMode sets the passed MPKCompound's mode attributes.

id characterizes the decomposition mode, and accepts the following values: MPK_COMPOUND_2D, MPK_COMPOUND_3D, MPK_COMPOUND_DB, MPK_COMPOUND_FSAA, MPK_COMPOUND_EYE, MPK_COMPOUND_HMD, MPK_COMPOUND_DPLEX and MPK_COMPOUND_CULL.

flags specifies additional flags for the compound mode. Currently, the following flags can be specified:

```
MPK_COMPOUND_MONO       compound is active in mono mode.
MPK_COMPOUND_STEREO     compound is active in stereo mode.
MPK_COMPOUND_ASYNC      compound execution is asynchronous (1).
MPK_COMPOUND_ADAPTIVE   compound is automatically load balanced (2).
MPK_COMPOUND_NOCOPY     compound and subtree pixel transfer is disabled.
MPK_COMPOUND_HW         use hardware for composition (3).
```

(1)

The MPK_COMPOUND_ASYNC flag specifies that the passed MPKCompound execution mode must be asynchronous, ie. that recomposition and all destination channel's rendering should be postponed by one frame, eg.:

```
  mode=2D          Dest. Channel        Source Channel[s]
 ____________________________________________________________

  Frame N          DrawPixels[N-1]      Render[N]
                   Render[N-1]          .
                   .                    .
                   .                    .
                   .                    ReadPixels[N]
 ____________________________________________________________

  Frame N+1        DrawPixels[N]        Render[N+1]
                   Render[N]            .
                   .                    .
                   .                    .
                   .                    ReadPixels[N+1]
```

Asynchronous execution provides in general better performances by providing better load balancing and by serializing the transfers on the bus.

(2)

For the compound modes 2D, DB and 3D, all children of this compound will be automatically load balanced by MPK. The load balancing algorithm is using the times needed by all children to render the destination channel's last frame to compute the distribution for the next frame. Note that this approach improves rendering performance for most cases, in some cases static decomposition may provide better results. It is recommended the all source channels have at least the size of the destination channel. The function mpkCompoundSetSplit() can be used to define the tiling scheme or z-axis split of an adaptive compound.

(3)

The current support modes for hardware composition are MPK_COMPOUND_DPLEX, MPK_COMPOUND_FSAA, MPK_COMPOUND_EYE and MPK_COMPOUND_2D. For DPlex hardware compositing support you need the Onyx2 DPLEX Option Hardware properly installed. For 2D, EYE and FSAA hardware compositing support you need the Scalable Graphics Compositor.

mpkCompoundGetMode reads the passed MPKCompound's mode attributes in the passed arguments which are not set to NULL.

mpkCompoundSetOperation sets the *compound* operation to MPK_COMPOUND_OP_DRAW, MPK_COMPOUND_OP_CULL, MPK_COMPOUND_OP_CULLDRAW or MPK_UNDEFINED.

The operation specifies which callbacks are invoked when this compound is updated. If the *operation* is MPK_UNDEFINED, the operation is inherited from the parent compound, or set to MPK_COMPOUND_OP_CULLDRAW if *compound* has no parent.

mpkCompoundGetOperation returns the operation of *compound*.

mpkCompoundSetName sets the name of the passed MPKCompound to *name*. This is done by copy and not by reference.

mpkCompoundGetName returns the name of the passed MPKCompound.

mpkCompoundSetSplit sets the split string of the passed MPKCompound to *split*. This is done by copy and not by reference. The split string defines the tiling scheme or z-axis split for adaptive compounds. See the File Format section for the split string syntax.

mpkCompoundGetSplit returns the split string of the passed MPKCompound.

mpkCompoundSetDisplayName sets the display name of the passed MPKCompound to *name*. This is done by copy and not by reference. The display name is used for setting up hardware compounds to be used with Xinerama in full overlap mode. For more information about scalable graphics hardware read the MPK User's Guide.

mpkCompoundGetDisplayName returns the display name of the passed MPKCompound.

mpkCompoundSetChannel specifies the MPKChannel to be used by the passed MPKCompound for rendering. The first channel specified in the hierarchy constitutes the destination channel of the subtree below, ie. the channel conditioning the final frame.

mpkCompoundGetChannel returns the MPKCompound's channel.

mpkCompoundSetViewport sets a 2D-compound's fractional viewport with respect to its destination MPKChannel and relatively to its parent MPKCompound, if any.

mpkCompoundGetViewport reads the compound's fractional viewport in *vp*.

mpkCompoundSetEye specifies the *eye* selection of an EYE or HMD compound, one of MPK_EYE_LEFT, MPK_EYE_RIGHT or MPK_EYE_CYCLOP.

mpkCompoundGetEye returns the *eye* selection of the passed MPKCompound.

mpkCompoundSetFormat specifies the *format* of the pixels to be transferred within the MPKCompound's hierarchy. *format* should be a bitwise combination of MPK_COLOR_BIT, MPK_DEPTH_BIT, MPK_STENCIL_BIT.

mpkCompoundGetFormat returns the passed MPKCompound's format, as a bitwise combination of MPK_COLOR_BIT, MPK_DEPTH_BIT, MPK_STENCIL_BIT.

mpkCompoundSetRange sets the two-dimensional *range* of the passed MPKCompound, as a fraction of its parent's range.

mpkCompoundGetRange reads the MPKCompound's two-dimensional *range*, relatively to its parent's range. The absolute "range" information can be retrieved by the application at rendering time via mpkChannelGetRange(), for the relevant portion of the database to be rendered accordingly.

mpkCompoundNChildren returns the number of MPKCompound children of the passed MPKCompound.

mpkCompoundGetChild returns the *i*th child of the passed MPKCompound.

mpkCompoundAddChild appends *child* to the list of children for the passed MPKCompound.

mpkCompoundRemoveChild searches for *child* in the list of children for the passed MPKCompound and removes it from the list if it is found.

mpkCompoundGetNext returns the sibling compound of an MPKCompound.

mpkCompoundGetParent returns the parent compound of an MPKCompound, or NULL if it is a top-level MPKCompound.

mpkCompoundGetConfig returns the parent config of an MPKCompound.

mpkCompoundFindChild searches for MPKCompound with specified *name* in the passed MPKCompound and returns the match if found or `NULL` otherwise.

mpkCompoundSetUserData enables the application to specify passthrough data to be transported within the *compound* structure. Transport is done by reference and not by copy.

mpkCompoundGetUserData enables the application to retrieve the passthrough data specified by mpkCompoundSetUserUserData().

Custom Compound Interface

mpkCompoundGetAssemblyFrame returns the *i*th frame from the *compound* assembly list for the current stereo eye pass. This function should only be called from the config's compound pre- or post- assemble callback. If *compound* mode is 2D, DB or CULL then *i* matches the child index responsible for the frame (e.g. the returned frame may then be NULL if both child and parent channels are identical). For other modes *i* should be 0.

mpkCompoundPreAssemble takes the images output from the source channels during last frame and assembles them in the destination channel prior to any rendering. MPK sets the compound pre-assemble callback by default to this function. It does something only if the ASYNC compound mode flag is set.

mpkCompoundPostAssemble takes the images from the source channels as soon as they have been produced and assembles them in the destination channel during the same frame. MPK sets the compound post-assemble callback by default to this function. It does something only if the ASYNC compound mode flag is not set.

Adaptive Readback Interface

mpkCompoundGetOutputFrame returns the *compound*'s current output frame. This function should only be called from the config's compound adaptive readback callback.

mpkCompoundReadOutputFrame executes the default adaptive readback callback.

Custom Compound Clear Interface

mpkCompoundClear executes the default clear callback. This callback is invoked on all destination compounds. This function only clears the framebuffer in special cases, for example when the adaptive readback callback is used.

File Format/Defaults

1. MPKCompound File Format specification:

```
compound {
                                        # compound FIELDS description

                                name    "compound-name"
                                channel "compound-channel's name"

                                mode    [ compound-mode description ]
                                format  [ compound-format description ]
                                split   " compound-split description "

                                        # compound REGIONS description

                                        region { region-1 description }
                                        region { region-2 description }
                                ...

}
```

2. MPKCompound-region File Format specification:

```
region {
                                  # region FIELDS description

                          viewport [ xf, yf, wf, hf ]
                          range    [ minf, maxf ]
                          eye      which

                                  # region CHANNEL or COMPOUND description

                          channel  "region-channel's name"
                                   compound { region-compound description }

}
```

viewport parameters are relative to the parent compound *viewport*, and therefore their values should be in the range 0.0 to 1.0

range parameters are relative to the parent compound *range*, and therefore their values should be in the range 0.0 to 1.0

eye field specification accepts only the following File Format identifiers: **cyclop** [default], **left** and **right**.

mode field specification accepts the following File Format identifiers for the mode-id: **2D**, **DB**, **3D**, **FSAA**, **EYE**, **HMD**, **DPLEX** and **CULL**. If none is specified, then MPK will simply synchronize the graphics update of all compound children, taking into account their respective latency. The mode's flags can be specified using the File Format identifiers **ASYNC**, **ADAPTIVE**, **NOCOPY**, **HW**, **STEREO** or **MONO**.

`split` field specifies the tiling scheme or z-axis split used when the compound is in ADAPTIVE mode. The split value is a string, as shown in the following example:

```
         split  "[[1 | 2] - [3 | 4]]"
```

The numbers 1, 2, 3, and 4 represent the regions in the compound (source channels). These numbers map the regions declared in the compound data structure in the order of declaration. All the regions declared in the compound data structure must be included into the split string. The axis that is split is represented by the following operators:

```
     |    axis x
     -    axis y
     /    axis z
```

The split operators '|' and '-' can be used only with 2D compounds and the operator '/', only with 3D or DB compounds.

The formal syntax of the split field is following:

```
  split          "splitString"
  splitString  : [ group axis group]
  group        : region | splitString
  axis         : '|'  | '-' |  '/'
  region       : [ integer ]
```

Notes

See also

[MPKChannel]

Multipipe SDK 3.2 Reference

Name

`MPKConfig` - MPKConfig functional interface.

Header File

#include <mpk/config.h>

Synopsis

Creating and Destroying

MPKConfig*	**mpkConfigNew**(void);
void	**mpkConfigDelete**(MPKConfig* *config*);
MPKConfig*	**mpkConfigLoad**(const char* *fileName*);
void	**mpkConfigOutput**(MPKConfig* *config*, int *tab*);

Fields Access

void	**mpkConfigSetName**(MPKConfig* *config*, const char* *name*);
const char*	**mpkConfigGetName**(MPKConfig* *config*);
void	**mpkConfigSetRunon**(MPKConfig* *config*, int *cpu*);
int	**mpkConfigGetRunon**(MPKConfig* *config*);
void	**mpkConfigSetMode**(MPKConfig* *config*, int *mode*);
int	**mpkConfigGetMode**(MPKConfig* *config*);
void	**mpkConfigSetMonitor**(MPKConfig* *config*, int *mode*, const char* *cmd*);
const char*	**mpkConfigGetMonitor**(MPKConfig* *config*, int *mode*);
void	**mpkConfigSetUserData**(MPKConfig* *config*, void* *data*);
void*	**mpkConfigGetUserData**(MPKConfig* *config*);
int	**mpkConfigNPipes**(MPKConfig* *config*);
MPKPipe*	**mpkConfigGetPipe**(MPKConfig* *config*, int *i*);
void	**mpkConfigAddPipe**(MPKConfig* *config*, MPKPipe* *p*);
int	**mpkConfigRemovePipe**(MPKConfig* *config*, MPKPipe* *p*);
int	**mpkConfigNCompounds**(MPKConfig* *config*);
MPKCompound*	**mpkConfigGetCompound**(MPKConfig* *config*, int *i*);
void	**mpkConfigAddCompound**(MPKConfig* *config*, MPKCompound* *c*);
int	**mpkConfigRemoveCompound**(MPKConfig* *config*, MPKCompound* *c*);
MPKCompound*	**mpkConfigFindCompound**(MPKConfig* *config*, const char* *name*);
MPKPipe*	**mpkConfigFindPipe**(MPKConfig* *config*, const char* *name*);
MPKWindow*	**mpkConfigFindWindow**(MPKConfig* *config*, const char* *name*);
MPKChannel*	**mpkConfigFindChannel**(MPKConfig* *config*, const char* *name*);
MPKWindow*	**mpkConfigMatchWindow**(MPKConfig* *config*, XID *drawable*);

Callbacks

void **mpkConfigSetPipeInitCB**(MPKConfig* *config*, MPKConfigPipeCB *cb*);
void **mpkConfigSetPipeExitCB**(MPKConfig* *config*, MPKConfigPipeCB *cb*);
void **mpkConfigSetWindowInitCB**(MPKConfig* *config*, MPKConfigWindowCB *cb*);
void **mpkConfigSetWindowExitCB**(MPKConfig* *config*, MPKConfigWindowCB *cb*);
void **mpkConfigSetChannelInitCB**(MPKConfig* *config*, MPKConfigChannelCB *cb*);
void **mpkConfigSetChannelExitCB**(MPKConfig* *config*, MPKConfigChannelCB *cb*);
void **mpkConfigSetDataFreeCB**(MPKConfig* *config*, MPKConfigDataCB *cb*);
void **mpkConfigSetFrameDataRefCB**(MPKConfig* *config*, MPKConfigFrameDataCB *cb*);
void **mpkConfigSetFrameDataUnrefCB**(MPKConfig* *config*, MPKConfigFrameDataCB *cb*);
void **mpkConfigSetIdleCB**(MPKConfig* *config*, MPKConfigIdleCB *cb*);
void **mpkConfigSetEventCB**(MPKConfig* *config*, MPKConfigEventCB *cb*);
void **mpkConfigSetCompoundPreAssembleCB**(MPKConfig* *config*, MPKCompoundAssembleCB *cb*);
void **mpkConfigSetCompoundPostAssembleCB**(MPKConfig* *config*, MPKCompoundAssembleCB *cb*);
void **mpkConfigSetCompoundReadOutputCB**(MPKConfig* *config*, MPKCompoundReadOutputCB *cb*);
void **mpkConfigSetCompoundClearCB**(MPKConfig* *config*, MPKCompoundClearCB *cb*);

MPKConfigPipeCB	**mpkConfigGetPipeInitCB**(MPKConfig* *config*);
MPKConfigPipeCB	**mpkConfigGetPipeExitCB**(MPKConfig* *config*);
MPKConfigWindowCB	**mpkConfigGetWindowInitCB**(MPKConfig* *config*);
MPKConfigWindowCB	**mpkConfigGetWindowExitCB**(MPKConfig* *config*);
MPKConfigChannelCB	**mpkConfigGetChannelInitCB**(MPKConfig* *config*);
MPKConfigChannelCB	**mpkConfigGetChannelExitCB**(MPKConfig* *config*);
MPKConfigDataCB	**mpkConfigGetDataFreeCB**(MPKConfig* *config*);
MPKConfigFrameDataCB	**mpkConfigGetFrameDataRefCB**(MPKConfig* *config*);
MPKConfigFrameDataCB	**mpkConfigGetFrameDataUnrefCB**(MPKConfig* *config*);
MPKConfigIdleCB	**mpkConfigGetIdleCB**(MPKConfig* *config*);
MPKConfigEventCB	**mpkConfigGetEventCB**(MPKConfig* *config*);
MPKCompoundAssembleCB	**mpkConfigGetCompoundPreAssembleCB**(MPKConfig* *config*);
MPKCompoundAssembleCB	**mpkConfigGetCompoundPostAssembleCB**(MPKConfig* *config*);
MPKCompoundReadOutputCB	**mpkConfigGetCompoundReadOutputCB**(MPKConfig* *config*);
MPKCompoundClearCB	**mpkConfigGetCompoundClearCB**(MPKConfig* *config*);

Operations

int **mpkConfigInit**(MPKConfig* *config*, int *setmon*);
void **mpkConfigExit**(MPKConfig* *config*);
void **mpkConfigFreeze**(MPKConfig* *config*, int *freeze*);
void **mpkConfigFrame**(MPKConfig* *config*, void* *framedata*);
void **mpkConfigFrameBegin**(MPKConfig* *config*, void* *framedata*);
void **mpkConfigFrameEnd**(MPKConfig* *config*);
int **mpkConfigChangeMode**(MPKConfig* *config*, int *mode*);
int **mpkConfigGetLatency**(MPKConfig* *config*);
int **mpkConfigIsIdle**(MPKConfig* *config*);

View Matrix Control

```
void mpkConfigSetViewPosition(MPKConfig* config, const float* pos);
void mpkConfigSetViewOrientation(MPKConfig* config, const float* hpr);
void mpkConfigSetViewMatrix(MPKConfig* config, const float* matrix);
```

Stereo & Head-Tracking

```
void mpkConfigSetHeadPosition(MPKConfig* config, const float* pos);
void mpkConfigSetHeadOrientation(MPKConfig* config, const float* hpr);
void mpkConfigSetHeadMatrix(MPKConfig* config, const float* matrix);
void mpkConfigSetEyeOffset(MPKConfig* config, float offset);
float mpkConfigGetEyeOffset(MPKConfig* config);
void mpkConfigEyeUpdate(MPKConfig* config);
```

Timing

```
void   mpkConfigTimerEnable(MPKConfig* config, int mode);
void   mpkConfigTimerDisable(MPKConfig* config, int mode);
void   mpkConfigTimerSetTime(MPKConfig* config, int mode, double t);
double mpkConfigTimerGetTime(MPKConfig* config, int mode);
```

Events

```
void       mpkConfigSelectInput(MPKConfig* config, long event_mask);
MPKEvent*  mpkConfigNextEvent(MPKConfig* config, double time);
int        mpkConfigCheckEvent(MPKConfig* config);
void       mpkConfigHandleEvents(MPKConfig* config);
```

Culling

```
void mpkConfigFrameData(MPKConfig* config, void* data);
void mpkConfigFrameFlush(MPKConfig* config);
```

Description

The MPKConfig data structure primarily describes the rendering resources of an OpenGL Multipipe SDK application, as a hierarchy of:

- hardware rendering pipelines (`MPKPipe`)
- GLX software rendering threads (`MPKWindow`)
- OpenGL framebuffer rendering areas (`MPKChannel`)

It may also describe various parallelization schemes (`MPKCompound`) of the rendering across channels, in order to scale performances.

The MPKConfig can be read from an ASCII file via **mpkConfigLoad** and launched via **mpkConfigInit**. Rendering threads are then spawned and the MPKConfig initialization callbacks invoked. These should in turn specify the rendering callbacks that will be triggered by **mpkConfigFrame**.

The role of the application is then simply to update the database and package the data pertaining to each frame, as illustrated below :

```
main( int argc, char *argv[] )
{
    mpkInit();
    MPKConfig *config = mpkConfigLoad( "1-window" );

    mpkConfigSetPipeInitCB( config, ... );
    mpkConfigSetWindowInitCB( config, ... );
    mpkConfigSetChannelInitCB( config, ... );
    mpkConfigSetDataFreeCB( config, ... );

    mpkConfigInit( config );
    while ( !exit ) {
        ...
        // update database
        ...
        framedata = newFrameData( db );
        mpkConfigFrame( config, framedata );
    }

    mpkConfigSetPipeExitCB( config, ... );
    mpkConfigSetWindowExitCB( config, ... );
    mpkConfigSetChannelExitCB( config, ... );

    mpkConfigExit( config );
}

static FrameData *frameDataBuffer = NULL;
FrameData *newFrameData( Database *db )
{
    FrameData *framedata;
    if ( frameDataBuffer == NULL ) {
        framedata = (FrameData *) mpkMalloc( sizeof(FrameData) );
    }
    else {
        framedata = frameDataBuffer;
        frameDataBuffer = framedata->next;
    }
    framedata->next = NULL;

    // copy relevant information from database into framedata
    ...
    return framedata;
}

void freeFrameData( MPKConfig *config, void *data )
{
    FrameData *framedata = (FrameData *)data;
    framedata->next = frameDataBuffer;
    frameDataBuffer = framedata;
```

```
        }
```

Function descriptions

Creating and Destroying

mpkConfigNew creates and returns a handle to an MPKConfig.

mpkConfigDelete deletes the passed MPKConfig.

mpkConfigLoad reads in and returns a handle to the MPKConfig described in *`file`*, or NULL upon any parsing error. The environment variable **MPK_PARSER_CMD** can be used in order to specify a pre-processing command to be applied to the file, typically /usr/lib/cpp. The environment variable **MPK_CONFIG_PATH** can be used to describe a search path for *`file`*.

mpkConfigOutput outputs the passed MPKConfig on stdout, with a left margin of *`tab`* tabulations.

Fields Access

mpkConfigSetName sets the name of the passed MPKConfig to *`name`*. This is done by copy and not by reference.

mpkConfigGetName returns the name of the passed MPKConfig.

mpkConfigSetRunon assigns all threads of the passed MPKConfig to be executed on the specified *`cpu`*, unless specified otherwise by mpkWindowSetRunon(). In addition, the following symbolic values can be used:

MPK_RUNON_AUTO	The window threads will be automatically placed on a CPU close to their respective graphics pipe, if possible.
MPK_RUNON_FREE	All threads are free to execute on whatever processor the system deems suitable.
MPK_UNDEFINED	The thread placement is defined by the MPK_DEFAULT_RUNON_POLICY, set using mpkGlobalSetAttributei()

mpkConfigGetRunon returns the cpu assignment specified with mpkConfigSetRunon() or `-1` if no assignment was specified.

mpkConfigSetMode sets the stereo mode of the passed MPKConfig to *`mode`*. Valid values for *`mode`* are `MPK_STEREO` and `MPK_MONO`.

mpkConfigGetMode returns the current stereo mode of the passed MPKConfig as either `MPK_STEREO` or `MPK_MONO`.

mpkConfigSetMonitor specifies the shell *`cmd`* to be executed when switching to the specified stereo *`mode`*. Valid values for *`mode`* are **MPK_STEREO** and **MPK_MONO**.

mpkConfigGetMonitor returns the monitor command for mode *`cmd`* specified with mpkConfigGetMonitor() or NULL. Accepted modes are **MPK_STEREO** and **MPK_MONO**.

mpkConfigSetUserData enables the application to specify passthrough *`data`* to be transported within the *`config`* structure. Transport is done by reference and not by copy.

mpkConfigGetUserData enables the application to retrieve the passthrough data specified by mpkConfigSetUserData().

mpkConfigNPipes returns the number of `MPKPipe` in passed *`config`*.

mpkConfigGetPipe returns the *`i`*th `MPKPipe` in passed *`config`*.

mpkConfigAddPipe appends `MPKPipe` *`p`* to list of pipes for passed *`config`*.

mpkConfigRemovePipe searches for *`p`* in list of `MPKPipe` for passed *`config`* and removes it from the list if it is found.

mpkConfigNCompounds returns the number of top-level `MPKCompound` in *`config`*.

mpkConfigGetCompound returns the *`i`*th top-level `MPKCompound` in *`config`*.

mpkConfigAddCompound appends the top-level `MPKCompound` *`c`* to *`config`*.

mpkConfigRemoveCompound searches for *`c`* in the list of top-level `MPKCompound` for the passed *`config`* and removes it from the list if found.

mpkConfigFindCompound searches for `MPKCompound` with specified *`name`* in the passed MPKConfig and returns the match if found or `NULL` otherwise.

mpkConfigFindPipe searches for `MPKPipe` with specified *`name`* in the passed MPKConfig and returns the match if found or `NULL` otherwise.

mpkConfigFindWindow searches for `MPKWindow` with specified *`name`* in the passed MPKConfig and returns the match if found or `NULL` otherwise.

mpkConfigFindChannel searches for `MPKChannel` with specified *`name`* in the passed MPKConfig and returns the match if found or `NULL` otherwise.

mpkConfigMatchWindow searches for `MPKWindow` with the same *`drawable`* in the passed MPKConfig and returns the match if found or `NULL` otherwise.

Callbacks

mpkConfigSetPipeInitCB sets the MPKConfig pipes initialization callback to the passed function, of type :

void (***MPKConfigPipeCB**)(MPKPipe*);

mpkConfigSetPipeExitCB sets the MPKConfig pipes exit callback to the passed function, of type :

void (***MPKConfigPipeCB**)(MPKPipe*);

mpkConfigSetWindowInitCB sets the MPKConfig windows initialization callback to the passed function, of type :

void (***MPKConfigWindowCB**)(MPKWindow*);
Default setting is to invoke mpkWindowCreate().

mpkConfigSetWindowExitCB sets the MPKConfig windows exit callback to the passed function, of type :

void (***MPKConfigWindowCB**)(MPKWindow*);
Default setting is to invoke mpkWindowDestroy().

mpkConfigSetChannelInitCB sets the MPKConfig channels initialization callback to the passed function, of type :

void (***MPKConfigChannelCB**)(MPKChannel*);

mpkConfigSetChannelExitCB sets the MPKConfig channels exit callback to the passed function, of type :

void (***MPKConfigChannelCB**)(MPKChannel*);

mpkConfigSetDataFreeCB sets the MPKConfig de-allocation callback to the passed function, of type :

void (***MPKConfigDataCB**)(MPKConfig*, void*);
This function gets invoked when the frame data which was passed to mpkConfigFrameBegin() is not used any more.

mpkConfigSetFrameDataRefCB sets the MPKConfig frame data referenciation callback to the passed function, of type :

void (***MPKConfigFrameDataCB**)(MPKConfig*, void*);

This function can be used in conjunction with the frame data dereferenciation callback to implement memory handling for data passed to mpkConfigFrameData(), mpkChannelPassData() or mpkChannelPutData().

mpkConfigSetFrameDataUnrefCB sets the MPKConfig frame data dereferenciation callback to the passed function, of type :

void (***MPKConfigFrameDataCB**)(MPKConfig*, void*);

This function can be used in conjunction with the frame data referenciation callback to implement memory handling for data passed to mpkConfigFrameData(), mpkChannelPassData() or mpkChannelPutData().

mpkConfigSetIdleCB sets the MPKConfig idle callback to the passed function, of type :

void (***MPKConfigIdleCB**)(MPKConfig*);
This function gets invoked by the application during the idle time after all non-threaded windows have been updated, as shown by the time diagram below:

```
Application                  MPKWindow 1            MPKWindow 2
_________________________________________________________________

mpkConfigFrameBegin          .
.                            Update channels        Update channels
mpkConfigFrameEnd            Update channels        Update channels
update non-threaded windows  Update channels        Update channels
idle callback                Update channels        Update channels
idle callback                .                      Update channels
idle callback                .                      .
.                            mpkWindowSwapBuffers   mpkWindowSwapBuffers
```

The function mpkConfigIsIdle() can be used to determine if windows are still beeing updated.

mpkConfigSetEventCB sets the MPKConfig event callback to the passed function, of type :

void (***MPKConfigEventCB**)(MPKConfig*);
This function gets invoked by the application after all windows have drawn and swapbuffered. The default event callback is mpkConfigHandleEvents().

mpkConfigSetCompoundPreAssembleCB sets the `config`'s compounds pre assemble callback to the passed function, of type :

void (***MPKCompoundAssembleCB**)(MPKCompound*, void*);

The default callback is mpkCompoundPreAssemble().

mpkConfigSetCompoundPostAssembleCB sets the *config*'s compounds post assemble callback to the passed function, of type :

void (***MPKCompoundAssembleCB**)(MPKCompound*, void*);

The default callback is mpkCompoundPostAssemble().

mpkConfigSetCompoundReadOutputCB sets the *config*'s compounds read output callback to the passed function, of type :

void (***MPKCompoundReadOutputCB**)(MPKCompound*);

The default callback is mpkCompoundReadOutputFrame().

mpkConfigSetCompoundClearCB sets the *config*'s compounds clear callback to the passed function, of type :

void (***MPKCompoundClearCB**)(MPKCompound*, void*);

The default callback is mpkCompoundClear().

mpkConfigGetPipeInitCB returns the *config*'s pipes init callback.

mpkConfigGetPipeExitCB returns the *config*'s pipes exit callback.

mpkConfigGetWindowInitCB returns the *config*'s windows init callback. Default setting is to invoke mpkWindowCreate().

mpkConfigGetWindowExitCB returns the *config*'s windows exit callback. Default setting is to invoke mpkWindowDestroy().

mpkConfigGetChannelInitCB returns the *config*'s channels init callback.

mpkConfigGetChannelExitCB returns the *config*'s channels exit callback.

mpkConfigGetDataFreeCB returns the *config*'s de-allocation callback.

mpkConfigGetFrameDataRefCB returns the *config*'s frame data referenciation callback.

mpkConfigGetFrameDataUnrefCB returns the *config*'s frame data dereferenciation callback.

mpkConfigGetIdleCB returns the *config*'s idle callback.

mpkConfigGetEventCB returns the *config*'s event callback.

mpkConfigGetCompoundPreAssembleCB returns the *config*'s compounds pre-assemble callback. Default setting is to invoke mpkCompoundPreAssemble().

mpkConfigGetCompoundPostAssembleCB returns the *config*'s compounds post-assemble callback. Default setting is to invoke mpkCompoundPostAssemble().

mpkConfigGetCompoundReadOutputCB returns the *config*'s compounds read output callback. Default setting is to invoke mpkCompoundReadOutputFrame().

mpkConfigGetCompoundClearCB returns the *config*'s compounds clear callback. Default setting is to invoke mpkCompoundClear().

Operations

mpkConfigInit launches the passed MPKConfig and spawns the MPKWindow loop threads. The config's initialization callbacks are invoked in the order described by the pseudo-code below :

```
for each MPKPipe in the config
    invoke config's pipes initialization callback
    for each MPKWindow in the pipe
       launch the window thread, which :

         invoke config's windows initialization callback
         for each MPKChannel in the window
             invoke config's channels initialization callback
         end for
         enter lifelong loop

    end for
end for
```

mpkConfigInit() will block until all MPKWindow threads have entered their lifelong loop. It then returns the number of threads launched.

If the argument flag *setmon* is set, then the shell commands will be invoked that have been specified via mpkConfigSetMonitor(), if any. Note that the config's windows initialization callback is set by default to mpkWindowCreate().

mpkConfigExit exit's the given *config*, according to the pseudo-code below:

```
for each MPKPipe in the config
    for each MPKWindow in the pipe
       exit the window thread, which :

         for each MPKChannel in the window
             invoke config's channels exit callback
         end for
         invoke config's windows exit callback
         exit window thread

    end for
    invoke config's pipes exit callback
end for
```

Note that the config's windows exit callback is set by default to mpkWindowDestroy().

mpkConfigFreeze with a non-zero *freeze* argument causes subsequent mpkConfigFrame() to perform without invoking any rendering callback, ie. the windows will be "frozen". Otherwise, frames will be rendered as usual.

mpkConfigFrame drives the passed MPKConfig to execute one frame of rendering, which causes all window threads to invoke their rendering callbacks. It is provided as a convenience function and executes the following code:

```
mpkConfigFrameBegin( config, framedata );
mpkConfigFrameEnd( config );
```

The *framedata* de-allocation should be done via a de-allocation callback function, to be specified prior to mpkConfigInit() via mpkConfigSetDataFreeCB().

mpkConfigFrameBegin triggers the rendering of a new frame. The passed *framedata* will be propagated to the config's channels rendering callbacks appropriately to the config's compounds latency.

The function mpkConfigFrameData() can be used after mpkConfigFrameBegin() to send data describing the frame to be rendered.

The function mpkConfigFrameEnd() is used to synchronize the end of the frame triggered by mpkConfigFrameBegin().

The *framedata* de-allocation should be done via a de-allocation callback function, to be specified prior to mpkConfigInit() via MPKConfigFreeDataCB().

mpkConfigFrameEnd synchronizes the frame started with mpkConfigFrameBegin(). Among other things, this function will synchronize the swapbuffer of all windows involved in the *config*.

mpkConfigChangeMode changes the MPKConfig's stereo mode to mode, which may involve exiting and restarting the configuration. If the new mode is the same as the old, then the change is ignored and 0 is returned. Otherwise the return value of mpkConfigInit() is returned.

mpkConfigGetLatency returns the maximum latency of the passed *config*. This value represents the maximum frame-delay between the config's compound source- and destination- channels updates. It also characterizes the maximum frame-delay between a user-input and the corresponding final composited frame. This function should be called on a running configuration, that is, after mpkConfigInit() has been called.

mpkConfigIsIdle is supposed to be called from the idle callback. It returns 1 if the application thread is still idle, ie. at least one window thread is still rendering. Otherwise it returns 0. When not called from the idle callback, the behaviour of this function is undefined. This function can be used to optimize the usage of the idle callback, as illustrated below:

```
void configIdle( MPKConfig *config )
{
    while( mpkConfigIsIdle( config ))
    {
        // do some processing
    }
}
```

Note that no data currently used in the rendering callbacks should be modified in the idle callback.

View Matrix Control

mpkConfigSetViewPosition specifies the position of the viewer.

mpkConfigSetViewOrientation specifies the orientation of the viewer as specified by the *`hpr`* angles, hence "hpr" specify the Euler angles of the head.

mpkConfigSetViewMatrix specifies the position and orientation of the viewer.

Stereo & Head-Tracking

mpkConfigSetHeadPosition specifies the position of the viewer in the arbitrary World Coordinates System used to describe the *`config`*.

mpkConfigSetHeadOrientation specifies the *`hpr`* angles of the line-of-sight, in degrees. "hpr" stands for head-pitch-roll, and describes the Euler angles of the head in the World Coordinates System used to describe the *`config`* with respect to the OpenGL convention, ie. the counter-clockwise rotation around the Y axis [head], X axis [pitch] and Z axis [roll] viewed from the positive side of the axis.

mpkConfigSetHeadMatrix specifies the 4x4 head transformation *`matrix`* in the arbitrary World Coordinates System used to describe the *`config`* :

```
head.matrix = TRANSLATE( head.position ) x
              ROTATE( head.hpr[0], 'y' ) x
              ROTATE( head.hpr[1], 'x' ) x
              ROTATE( head.hpr[2], 'z' )

where "hpr" stands for head, pitch, roll.
```

mpkConfigSetEyeOffset sets the *`offset`* from each eye to the "head" position, ie. half the interoccular distance, to be used by *`config`*.

mpkConfigGetEyeOffset returns the offset from each eye to the head position, ie. half the interoccular distance, used by *`config`*.

mpkConfigEyeUpdate forces the current head transformation and eye positions to be recomputed from changes made to the head position, head orientation, head matrix or eye-offset via the functions above. mpkConfigFrame() invokes mpkConfigEyeUpdate().

Timing

mpkConfigTimerEnable with *mode* set to MPK_TIMER_AUTO activates automatic load-balancing. This mode is enabled by default for DPLEX MPKCompound. If *mode* is set to MPK_TIMER_FRAME then MPK will measure the duration of subsequent frames, which can be retrieved via mpkConfigTimerGetTime(). This mode is disabled by default.

mpkConfigTimerDisable disables the timer *mode* for *config*. Valid values for *mode* are MPK_TIMER_AUTO and MPK_TIMER_FRAME.

mpkConfigTimerSetTime with argument *mode* set to MPK_TIMER_FRAME specifies the desired minimal duration in milliseconds for the subsequent config frames. Default value is 0., ie. no time constraint.

mpkConfigTimerGetTime returns either the actual duration in milliseconds of the last config frame, ie. if *mode* is MPK_TIMER_FRAME and this mode is enabled, or, if *mode* is MPK_TIMER_AUTO, it returns MPK recommended minimal duration for next frame, which will be applied if *config* involves DPLEX MPKCompound.

Events

mpkConfigSelectInput loops over all windows of *config*, and sets the window's event mask if this window has an input display and an X window drawable. Note that the window's input display is set via mpkWindowOpenDisplay() or mpkWindowSetInputDisplay().

mpkConfigNextEvent returns the next MPKEvent on the event queue. If there is no MPKEvent queued, this function blocks until a MPKEvent is received.

If *time* is non-zero, it specifies a maximum interval in milliseconds to wait. If *time* is zero, mpkConfigNextEvent blocks indefinitely.

The returned MPKEvent is valid until the next call to mpkConfigNextEvent.

mpkConfigCheckEvent returns 0 if there are no events pending, 1 otherwise.

mpkConfigHandleEvents processes pending events on all windows. Note that MPKConfig's event callback is set by default to mpkConfigHandleEvents.

Culling

mpkConfigFrameData is used to describe the current frame between mpkConfigFrameBegin() and mpkConfigFrameEnd(). MPK passes *data* to the cull and draw callbacks, as defined in the given *config*.

Latency-correct memory management for *data* can be done via frame data referenciation and dereferenciation callbacks, to be specified prior to mpkConfigInit() via MPKConfigSetFrameDataRefCB() and MPKConfigSetFrameDataUnrefCB().

An application using the culling infrastructure of MPK would typically be programmed as described in the pseudo-code below:

```
int main(...)
{
    ...
    while ( !exit ) {
         ...
         // update database
         ...
         framedata = newFrameData( db );
         mpkConfigFrameBegin( config, framedata );
         mpkConfigFrameData( config, data1 );
         ...
         mpkConfigFrameData( config, dataN );
         mpkConfigFrameEnd( config );
     }
     ...
}

void cullChannel( MPKChannel *c, void *data )
{
     ...
     while( (data = mpkChannelNextData( c )) != NULL )
     {
         if( isVisible( data ))
         {
              mpkChannelPassData( c, data )
         }
     }
}

void updateChannel( MPKChannel *c, void *data )
{
     ...
     while( (data = mpkChannelNextData( c )) != NULL )
     {
         render( data );
     }
}
```

mpkConfigFrameFlush forces a flush of the input buffer to the config's frame data queue, filled using the function mpkConfigFrameData(). The size of the buffer can be set using mpkGlobalSetAttributei() for attribute MPK_CONFIG_FRAME_CACHE_SIZE prior mpkConfigNew().

File Format/Defaults

config {

# config FIELDS description

```
    name   "config-name"
    runon  processor-id

    mode   stereo-mode
    mono   "shell-command"
    stereo "shell-command"

            # config PIPES description

           pipe { pipe-1 description }
           pipe { pipe-2 description }
    ...

            # config COMPOUNDS description

           compound { compound-1 description }
           compound { compound-2 description }
    ...

}
```

Notes

`stereo-mode` description accepts only the following File Format identifiers : **mono** [default] and **stereo**.

On machines with small hardware counters the timer counter wraps. MPK detects this overflow, but assumes that only one overflow happened. If the time needed for one frame is very long (around one minute), the timer interface may behave incorrect on this machines. See the clock_gettime(2) man page for further details.

See also

`MPKChannel`, `MPKCompound`, `MPKEvent`, `MPKPipe`, `MPKWindow`

Multipipe SDK 3.2 Reference

Name

`MPKEvent` - MPKEvent functional interface.

Header File

#include <mpk/event.h>

Synopsis

Fields Access

void* **mpkEventGetData**(MPKEvent* *event*);

Description

The MPKEvent data structure encapsulates an X11 event. It provides convenience functions decoding the data in the corresponding XEvent. Note that the MPKEvent is freed automatically by MPK, so the pointer to one MPKEvent should not be stored within the application.

Function descriptions

Fields Access

mpkEventGetData returns the event specific data. Currently, this function returns a pointer to an MPKEventXData structure, containing:

```
typedef struct
{
    struct  { int key, ctrl, shft, state; } keyboard;
    struct  { int left, middle, right; } button;
    struct  { int x, y, dx, dy, xref, yref; } mouse;

    XEvent  *x;
}
MPKEventXData;
```

keyboard.key contains the KeySym value of the last modified key, as returned by XLookupString(3X11). keyboard.ctrl and keyboard.shft are either TRUE or FALSE depending on the state information of the last modified key. keyboard.state contains either MPK_RELEASE or MPK_PRESS, depending on the key state.

button.left, button.middle and middle.right fields contain either MPK_RELEASE or MPK_PRESS, depending on the mouse-button state.

mouse.x and mouse.y fields contain the current mouse position, while mouse.xref and mouse.yref contain the last registered mouse position, and mouse.dx and mouse.dy contain the incremental variation of the mouse position.

Multipipe SDK 3.2 Reference

Name

`MPKFrame` - MPKFrame functional interface.

Header File

#include <mpk/frame.h>

Synopsis

Creating and Destroying

```
MPKFrame* mpkFrameNew(void );
void      mpkFrameDelete(MPKFrame* frame);
```

Fields Access

```
int       mpkFrameNImages(MPKFrame* frame, int type);
MPKImage* mpkFrameGetImage(MPKFrame* frame, int type, int i);
void      mpkFrameAddImage(MPKFrame* frame, int type, MPKImage* image);
int       mpkFrameRemoveImage(MPKFrame* frame, int type, MPKImage* image);
void      mpkFrameSetFormat(MPKFrame* frame, int format);
int       mpkFrameGetFormat(MPKFrame* frame);
void      mpkFrameSetRegion(MPKFrame* frame, float region[4]);
void      mpkFrameGetRegion(MPKFrame* frame, float region[4]);
```

Operations

```
void  mpkFrameSetUserData(MPKFrame* frame, void* data);
void* mpkFrameGetUserData(MPKFrame* frame);
```

Description

The MPKFrame data structure primarily describes a frame in an MPK application. It is a container for MPKImage images. See mpkChannelDrawImage() for a detailed explanation on how to use MPKFrame and MPKImage structures.

Function descriptions

Creating and Destroying

mpkFrameNew creates and returns a handle to an MPKFrame.

mpkFrameDelete deletes the passed MPKFrame.

Fields Access

mpkFrameNImages returns the number of MPKImages for the specified *type*. Accepted values for *type* are MPK_COLOR_BIT, MPK_DEPTH_BIT and MPK_STENCIL_BIT.

mpkFrameGetImage returns the *i*th MPKImage of *type*. Accepted values for *type* are MPK_COLOR_BIT, MPK_DEPTH_BIT and MPK_STENCIL_BIT.

mpkFrameAddImage add *image* of *type* to this MPKFrame. Accepted values for *type* are MPK_COLOR_BIT, MPK_DEPTH_BIT and MPK_STENCIL_BIT.

mpkFrameRemoveImage removes *image* of the specified *type* from this MPKFrame. Accepted values for *type* are MPK_COLOR_BIT, MPK_DEPTH_BIT and MPK_STENCIL_BIT.

mpkFrameSetFormat set the frame format. *format* is a bitwise combination of MPK_COLOR_BIT, MPK_DEPTH_BIT and MPK_STENCIL_BIT.

mpkFrameGetFormat returns the frame format. The returned value is a bitwise combination of MPK_COLOR_BIT, MPK_DEPTH_BIT and MPK_STENCIL_BIT.

mpkFrameSetRegion set the *region* of the frame. The *region* describes the 2D fractional viewport with respect to the channel viewport.

mpkFrameGetRegion returns the *region* of the frame. The *region* describes the 2D fractional viewport with respect to the channel viewport.

Operations

mpkFrameSetUserData enables the application to specify passthrough data to be transported within the *frame* structure. Transport is done by reference and not by copy.

mpkFrameGetUserData enables the application to retrieve the passthrough data specified by mpkFrameSetUserData().

Multipipe SDK 3.2 Reference

Name

MPKGlobal - MPKGlobal functional interface.

Header File

#include <mpk/global.h>

Synopsis

void **mpkInit**(void);
void **mpkExit**(void);
void **mpkGlobalOutput**(int *tab*);
const char* **mpkGetString**(int *name*);

Execution Mode

void **mpkGlobalSetExecutionMode**(int *mode*);
int **mpkGlobalGetExecutionMode**(void);

Arena Attributes

void **mpkGlobalSetArenaAttributei**(int *aattr*, int *val*);
int **mpkGlobalGetArenaAttributei**(int *aattr*);
void **mpkGlobalSetArenaPath**(const char* *path*);
const char* **mpkGlobalGetArenaPath**(void);

Global Attributes

void **mpkGlobalSetAttributei**(int *attr*, int *val*);
int **mpkGlobalGetAttributei**(int *attr*);
void **mpkGlobalSetAttributef**(int *attr*, float *val*);
float **mpkGlobalGetAttributef**(int *attr*);

Pipe Attributes

void **mpkGlobalSetPipeAttributei**(int *pattr*, int *val*);
int **mpkGlobalGetPipeAttributei**(int *pattr*);

Window Attributes

void **mpkGlobalSetWindowAttributei**(int *wattr*, int *val*);

int **mpkGlobalGetWindowAttributei**(int *wattr*);

Channel Attributes

void **mpkGlobalSetChannelAttributei**(int *cattr*, int *val*);
int **mpkGlobalGetChannelAttributei**(int *cattr*);
void **mpkGlobalSetChannelAttributef**(int *cattr*, float *val*);
float **mpkGlobalGetChannelAttributef**(int *cattr*);

Description

The MPKGlobal data structure specifies OpenGL Multipipe SDK default attribute values. Execution Mode and MPKArena attributes are not accessible via the File Format interface.

Function descriptions

mpkInit initializes internal OpenGL Multipipe SDK data structures and must be the first OpenGL Multipipe SDK call in an application except for the following:

- mpkGetString
- mpkGlobalSetExecutionMode
- mpkGlobalSetArenaAttributei
- mpkGlobalSetArenaPath

mpkExit exits OpenGL Multipipe SDK.

mpkGlobalOutput outputs the MPKGlobal attributes which have been set, either by default or by the application. *tab* specifies the number of tabulations to be applied for the left margin.

mpkGetString returns a pointer to a static string describing some aspect of the current OpenGL Multipipe SDK library. *name* can be one of the following:

- MPK_VERSION
- MPK_VENDOR

Execution Mode

mpkGlobalSetExecutionMode sets the execution mode of an OpenGL Multipipe SDK application to either MPK_EXECUTION_PTHREAD, MPK_EXECUTION_SPROC, MPK_EXECUTION_FORK or MPK_EXECUTION_PERFORMER.

mpkGlobalGetExecutionMode returns the application's execution mode.

Arena Attributes

The following functions will not have any effect unless the execution mode is set to **MPK_EXECUTION_SPROC**, **MPK_EXECUTION_FORK** or **MPK_EXECUTION_PERFORMER** in which case mpkInit() creates an internal shared arena matching the specified attributes.

mpkGlobalSetArenaAttributei sets the value of the passed MPKArena attribute. Default values are set for MPK_AATTR_SIZE (2^28) and MPK_AATTR_USERS (100).

mpkGlobalGetArenaAttributei returns the value of the specified MPKArena attribute, or MPK_UNDEFINED if this value has not been set.

mpkGlobalSetArenaPath specifies the path to a directory where the application has read-write permission for mpkInit() to create the MPKArena. Default path is "/usr/tmp".

mpkGlobalGetArenaPath returns the path to the arena.

Global Attributes

mpkGlobalSetAttributei sets the specified attribute. The following symbols are accepted for attr:

MPK_TIMER_SIGNAL
[**int**] Specifies the signal which is used as the notification mechanism by a timer when firing. The default value is SIGALRM. Alter this value if you already make use of the particular signal in your application.

MPK_XINERAMA
[**bool**] Setting this variable to 0 enables MPK to perform faster window creation, in case no "Xinerama-aware" windows are used. Note that setting this variable to 0 while using "Xinerama_aware" windows might cause unpredictible results. The default value is 1.

MPK_CHANNEL_AUTO_ACTIVATE
[**bool**] Enables or disables the channel auto activation feature. If enabled, MPK automatically creates a compound for each channel not referenced by an existing compound during mpkConfigInit(). Therefore, unused channels are automatically actived. The default value is 1 (enabled).

MPK_DEFAULT_RUNON_POLICY
[**bool**] Sets the default thread runon policy which applies to windows which have an unspecified runon value. Accepted values are MPK_RUNON_FREE and MPK_RUNON_AUTO. The default value is MPK_RUNON_FREE.

MPK_CONFIG_FRAME_CACHE_SIZE
[**int**] Specifies the cache size for the config frame data queue used for culling. This attribute influences the granularity and performance of the data processing for data passed using mpkConfigFrameData(). The default value is 100.

MPK_CHANNEL_PASS_CACHE_SIZE
[**int**] Specifies the cache size for the frame data queues used for culling. This attribute influences the granularity and performance of the data processing for data passed using mpkChannelPassData(). The default value is 50.

MPK_CHANNEL_PUT_CACHE_SIZE
[**int**] Specifies the cache size for the culling data queues. This attribute influences the granularity and performance of the data processing for data passed using mpkChannelPutData(). The default value is 10.

mpkGlobalGetAttributei returns the value of the specified attribute, or MPK_UNDEFINED if this value has not been set.

mpkGlobalSetAttributef sets the specified attribute. The following symbols are accepted for attr:

MPK_DEFAULT_EYE_OFFSET
Specifies the default value of the eye offset used by MPKChannel frustum computations. The default value is 0.035

mpkGlobalGetAttributef returns the value of the specified attribute.

Pipe Attributes

mpkGlobalSetPipeAttributei sets the default values for the specified MPKPipe attribute. The following symbols are accepted for pattr:

MPK_PATTR_MONO_WIDTH [**int**] Specifies the MPKPipe width in mono display mode.

MPK_PATTR_MONO_HEIGHT [**int**] Specifies the MPKPipe height in mono display mode.

MPK_PATTR_STEREO_TYPE [**int**] Specifies the MPKPipe stereo type. Accepted values are **MPK_STEREO_USER**, **MPK_STEREO_QUAD**, **MPK_STEREO_RECT**, **MPK_STEREO_TOP**, **MPK_STEREO_BOT**.

MPK_PATTR_STEREO_WIDTH [**int**] Specifies the MPKPipe width in stereo display mode.

MPK_PATTR_STEREO_HEIGHT [**int**] Specifies the MPKPipe height in stereo display mode.

MPK_PATTR_STEREO_OFFSET [**int**] Specifies the MPKPipe stereo offset for **MPK_STEREO_TOP** and **MPK_STEREO_BOT** stereo modes.

mpkGlobalGetPipeAttributei returns the value of the specified Pipe attribute, or MPK_UNDEFINED if this value has not been set.

Window Attributes

mpkGlobalSetWindowAttributei sets the default values for the specified MPKWindow attribute. The following symbols are accepted for wattr:

MPK_WATTR_HINTS_VISUAL [**enum**] Specifies the MPKWindow visual type. Accepted values are **MPK_GLX_TRUE_COLOR**, **MPK_GLX_PSEUDO_COLOR**, **MPK_GLX_DIRECT_COLOR**, **MPK_GLX_STATIC_COLOR**, **MPK_GLX_GRAYSCALE**, **MPK_GLX_STATIC_GRAY**

MPK_WATTR_HINTS_DRAWABLE [**enum**] Specifies the MPKWindow drawable type. Accepted values are **MPK_GLX_WINDOW**, **MPK_GLX_PBUFFER**, **MPK_GLX_PIXMAP**

MPK_WATTR_HINTS_CAVEAT [**enum**] Specifies the caveats associated with the MPKWindow framebuffer configuration. Accepted values are **MPK_GLX_SLOW**, **MPK_GLX_NOCAVEAT**, **MPK_GLX_NON_CONFORMANT**

MPK_WATTR_HINTS_X_RENDERABLE [**bool**] Specifies whether only frame-buffer configuration that have associated X visuals (and can be used to render to Windows and/or GLX pixmaps) should be considered.

MPK_WATTR_HINTS_DIRECT [**bool**] Specifies whether the MPKWindow GLXContext should be direct.

MPK_WATTR_HINTS_DECORATION [**bool**] Specifies whether the Window should have MOTIF decorations.

MPK_WATTR_HINTS_MOVE [**bool**] Specifies whether the window can be moved via mwm/4Dwm, or not.

MPK_WATTR_HINTS_RESIZE

[**bool**] Specifies whether the window can be resized or not.

MPK_WATTR_HINTS_ASPECT

[**bool**] Specifies whether a fixed window aspect ratio is enforced.

MPK_WATTR_HINTS_MINMAX

[**bool**] Specifies whether the window can be minimized/maximized via mwm/4Dwm.

MPK_WATTR_HINTS_OVERRIDEREDIRECT

[**bool**] Specifies whether the window has the override_redirect attribute set at creation time.

MPK_WATTR_HINTS_RGBA

[**bool**] Specifies whether MPKWindow visuals should support RGBA rendering mode. The default value is 1. (true).

MPK_WATTR_HINTS_DOUBLEBUFFER

[**bool**] Specifies whether the MPKWindow frame buffer configuration should be double-buffered. Note that setting this attribute on a created MPKWindow will affect the behaviour of **mpkWindowSwapBuffers**. The default value is 1 (true).

MPK_WATTR_HINTS_STEREO

[**bool**] Specifies whether the MPKWindow frame buffer configuration should support quad-buffer stereo.

MPK_WATTR_HINTS_TRANSPARENT

[**bool**] Specifies whether the MPKWindow frame buffer configuration should be transparent.

MPK_WATTR_HINTS_LARGEST

[**bool**] Specifies whether the largest available pbuffer should be obtained, if the allocation requested by the window size would have failed. The width and height of the allocated pixel buffer will never exceed the specified window width or height, respectively. This attribute will be ignored by MPKWindows for which the DRAWABLE hint is not set to MPK_GLX_PBUFFER.

MPK_WATTR_HINTS_PRESERVED

[**bool**] Specifies whether the contents of the pixel buffer should be preserved when a resource conflict occurs. This attribute will be ignored by MPKWindows for which the DRAWABLE hint is not set to MPK_GLX_PBUFFER.

MPK_WATTR_HINTS_THREAD

[**bool**] Specifies whether the MPKWindow should be made a separate thread from the application.

MPK_WATTR_HINTS_XINERAMA

[**bool**] Specifies whether the MPKWindow should use Xinerama(1) or be "Xinerama-aware"(0).

MPK_WATTR_PLANES_LEVEL

[**int**] Specifies the MPKWindow buffer level. The default value is 0.

MPK_WATTR_PLANES_AUX

[**int**] Specifies the number of auxiliary buffers.

MPK_WATTR_PLANES_DEPTH

[**int**] Specifies the minimum size of the depth buffer. The default value is 1.

MPK_WATTR_PLANES_STENCIL

[**int**] Specifies the minimum size of the stencil buffer.

MPK_WATTR_PLANES_SAMPLES

[**int**] Specifies the minimum number of samples required in the multisample buffer.

MPK_WATTR_PLANES_COLOR

[**int**] Specifies the minimum color index buffer size. This attribute is ignored if the RGBA hint of the MPKWindow is set to 1.

MPK_WATTR_PLANES_RED

[**int**] Specifies the minimum number of red bitplanes. This attribute is ignored if the RGBA hint of the MPKWindow is not set. The default value is 1.

MPK_WATTR_PLANES_GREEN

[**int**] Specifies the minimum number of green bitplanes. This attribute is ignored if the RGBA hint of the MPKWindow is not set. The default value is 1.

MPK_WATTR_PLANES_BLUE

[**int**] Specifies the minimum number of blue bitplanes. This attribute is ignored if the RGBA hint of the MPKWindow is not set. The default value is 1.

MPK_WATTR_PLANES_ALPHA

[**int**] Specifies the minimum number of alpha bitplanes. This attribute is ignored if the RGBA hint of the MPKWindow is not set. The default value is 0.

MPK_WATTR_PLANES_ACCUM_RED

[**int**] Specifies the minimum number of accumulation red bitplanes. This attribute is ignored if the RGBA hint of the MPKWindow is not set.

MPK_WATTR_PLANES_ACCUM_GREEN

[**int**] Specifies the minimum number of accumulation green bitplanes. This attribute is ignored if the RGBA hint of the MPKWindow is not set.

MPK_WATTR_PLANES_ACCUM_BLUE

[**int**] Specifies the minimum number of accumulation blue bitplanes. This attribute is ignored if the RGBA hint of the MPKWindow is not set.

MPK_WATTR_PLANES_ACCUM_ALPHA

[**int**] Specifies the minimum number of accumulation alpha bitplanes. This attribute is ignored if the RGBA hint of the MPKWindow is not set.

MPK_WATTR_TRANSPARENT_RED

[**int**] Specifies the red component of the MPKWindow transparent color. This attribute is ignored if the RGBA hint of the MPKWindow is not set, or if the TRANSPARENT hint of the MPKWindow is not set.

MPK_WATTR_TRANSPARENT_GREEN

[**int**] Specifies the green component of the MPKWindow transparent color. This attribute is ignored if the RGBA hint of the MPKWindow is not set, or if the TRANSPARENT hint of the MPKWindow is not set.

MPK_WATTR_TRANSPARENT_BLUE

[**int**] Specifies the blue component of the MPKWindow transparent color. This attribute is ignored if the RGBA hint of the MPKWindow is not set, or if the TRANSPARENT hint of the MPKWindow is not set.

MPK_WATTR_TRANSPARENT_ALPHA

[**int**] Specifies the alpha component of the MPKWindow transparent color. This attribute is ignored if the RGBA hint of the MPKWindow is not set, or if the TRANSPARENT hint of the MPKWindow is not set.

MPK_WATTR_TRANSPARENT_INDEX

[**int**] Specifies the MPKWindow transparent index. This attribute is ignored if the RGBA hint of the MPKWindow is set, or if the TRANSPARENT hint of the MPKWindow is not set.

More information about GLX visual attributes specifications can be found in the **glXChooseFBConfig**(3G) and **glXChooseVisual**(3G) man pages.

mpkGlobalGetWindowAttributei returns the value of the specified MPKWindow attribute, or MPK_UNDEFINED if this value has not been set.

Channel Attributes

mpkGlobalSetChannelAttributei sets the default values for the specified MPKChannel attribute. Note that not all format, type combinations are valid. Please refer to the glReadPixels man page for detailed information about accepted combinations. The following symbols are accepted for cattr:

MPK_CATTR_READ_COLOR_FORMAT

[**enum**] Specifies the default MPKFrame format used by the MPKChannel to read color pixel data. Accepted values are **GL_RGBA**, **GL_RGB**, **GL_BGRA**, **GL_BGR**, **GL_ABGR_EXT**, **GL_COLOR_INDEX**, **GL_RED**, **GL_GREEN**, **GL_BLUE**, **GL_ALPHA**, **GL_LUMINANCE**, **GL_LUMINANCE_ALPHA**

MPK_CATTR_READ_COLOR_TYPE

[**enum**] Specifies the default MPKFrame type used by the MPKChannel to read color pixel data. Accepted values are **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_BITMAP**, **GL_UNSIGNED_BYTE_3_3_2**, **GL_UNSIGNED_BYTE_2_3_3_REV**, **GL_UNSIGNED_SHORT_5_6_5**, **GL_UNSIGNED_SHORT_5_6_5_REV**, **GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**, **GL_UNSIGNED_INT_8_8_8_8**, **GL_UNSIGNED_INT_8_8_8_8_REV**, **GL_UNSIGNED_INT_10_10_10_2**, **GL_UNSIGNED_INT_2_10_10_10_REV**

MPK_CATTR_READ_DEPTH_FORMAT

[**enum**] Specifies the default MPKFrame format used by the MPKChannel to read depth pixel data. Accepted values are **GL_DEPTH_COMPONENT**, **GL_DEPTH_COMPONENT24_SGIX**

MPK_CATTR_READ_DEPTH_TYPE [**enum**] Specifies the default MPKFrame type used by the MPKChannel to read depth pixel data. Accepted values are **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**

MPK_CATTR_READ_STENCIL_FORMAT [**enum**] Specifies the default MPKFrame format used by the MPKChannel to read stencil pixel data. Accepted values are **GL_STENCIL_INDEX**

MPK_CATTR_READ_STENCIL_TYPE [**enum**] Specifies the default MPKFrame type used by the MPKChannel to read stencil pixel data. Accepted values are **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_BITMAP**

mpkGlobalGetChannelAttributei returns the value of the specified MPKChannel attribute, or MPK_UNDEFINED if this value has not been set.

mpkGlobalSetChannelAttributef currently only sets the default near and far distances of the MPKChannel. These values are preempted by mpkChannelSetNearFar().

mpkGlobalGetChannelAttributef returns the value of the specified MPKChannel attribute.

File Format/Defaults

```
global {
                        MPK_DEFAULT_EYE_OFFSET          0.035
                        MPK_XINERAMA                    1
                        MPK_CATTR_NEAR                  0.01
                        MPK_CATTR_FAR                   100.

                        MPK_PATTR_STEREO_HEIGHT         492
                        MPK_PATTR_STEREO_OFFSET         532

                        MPK_WATTR_HINTS_THREAD          1

                        MPK_WATTR_HINTS_RGBA            1
                        MPK_WATTR_HINTS_DOUBLEBUFFER    1
                        MPK_WATTR_PLANES_LEVEL          0
                        MPK_WATTR_PLANES_DEPTH          1
                        MPK_WATTR_PLANES_RED            1
                        MPK_WATTR_PLANES_GREEN          1
                        MPK_WATTR_PLANES_BLUE           1
                        MPK_WATTR_PLANES_ALPHA          0

}
```

Notes

MPK_WATTR_HINTS_DRAWABLE description accepts only the following File Format identifiers : **none**, **window** [default], **pbuffer** and **pixmap**.

MPK_WATTR_HINTS_CAVEAT description accepts only the following File Format identifiers : **none** [default], **slow** and **non-conformant**.

MPK_WATTR_HINTS_VISUAL description accepts only the following File Format identifiers : **true-color** [default], **pseudo-color**, **direct-color**, **static-color**, **static-gray** or **grayscale**.

Multipipe SDK 3.2 Reference

Name

MPKImage - MPKImage functional interface.

Header File

#include <mpk/image.h>

Synopsis

Creating and Destroying

MPKImage* **mpkImageNew**(void);
void **mpkImageDelete**(MPKImage* *image*);

Fields Access

int **mpkImageSetPixel**(MPKImage* *image*, int *format*, int *type*);
void **mpkImageGetPixel**(MPKImage* *image*, int* *format*, int* *type*);
int **mpkImageGetPixelSize**(MPKImage* *image*);
void **mpkImageSetSize**(MPKImage* *image*, int *size[2]*);
void **mpkImageGetSize**(MPKImage* *image*, int *size[2]*);
void **mpkImageSetOffset**(MPKImage* *image*, int *offset[2]*);
void **mpkImageGetOffset**(MPKImage* *image*, int *offset[2]*);
void **mpkImageSetBuffer**(MPKImage* *image*, void* *buffer*, size_t *size*);
void* **mpkImageGetBuffer**(MPKImage* *image*, size_t* *size*);
void **mpkImageSetUserData**(MPKImage* *image*, void* *data*);
void* **mpkImageGetUserData**(MPKImage* *image*);

Description

The MPKImage data structure primarily describes raw pixel data. See mpkChannelDrawImage() for a detailed explanation on how to use MPKFrame and MPKImage structures.

Function descriptions

Creating and Destroying

mpkImageNew creates and returns a handle to an MPKImage.

mpkImageDelete deletes the passed MPKImage.

Fields Access

mpkImageSetPixel set the `format` and `type` of the `image`.

format specifies the format of the pixel data. The following symbolic values are accepted: GL_RGBA, GL_ABGR_EXT, GL_RGB, GL_LUMINANCE_ALPHA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE, GL_COLOR_INDEX, GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT24_SGIX.

type specifies the data type of the pixel data. Must be one of GL_UNSIGNED_BYTE_3_3_2_EXT, GL_UNSIGNED_SHORT_4_4_4_4_EXT, GL_UNSIGNED_SHORT_5_5_5_1_EXT, GL_UNSIGNED_INT_8_8_8_8_EXT, GL_UNSIGNED_INT_10_10_10_2_EXT, GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT.

All other symbolic format or type value not listed above are not allowed by MPK.

mpkImageGetPixel returns the *format* and *type* of the *image*. See mpkImageSetPixel() for a list of symbolic values returned by this function.

mpkImageGetPixelSize returns the size of one pixel data in byte. If the current format and/or type of this image is not supported by MPK, this function returns 0.

mpkImageSetSize set the *size* of the *image*. If the new *size* is bigger than the initial one, a new array is not reallocated for the pixels.

mpkImageGetSize returns the *size* of the image.

mpkImageSetOffset set the *offset* in pixels of the *image*, with respect to the position defined by the frame region, specified using mpkFrameSetRegion().

mpkImageGetOffset returns the *offset* in pixels of the *image* with respect to the position defined by the frame region, specified using mpkFrameSetRegion().

mpkImageSetBuffer set the *buffer* of the *image*. The *buffer* must be allocated before and has a size of *size* bytes.

mpkImageGetBuffer returns a pointer to the pixel data of the *image*. The pixels format and type can be retrieve with mpkImageGetPixel(). The function also returns the current allocated *size* of the buffer.

mpkImageSetUserData enables the application to specify passthrough data to be transported within the *image* structure. Transport is done by reference and not by copy.

mpkImageGetUserData enables the application to retrieve the passthrough data specified by mpkImageSetUserData().

Multipipe SDK 3.2 Reference

Name

`MPKIntro` - Overview of OpenGL Multipipe SDK

Header File

#include <mpk/mpk.h>

Description

Welcome to the OpenGL Multipipe Software Development Kit !

More and more creative, technical and business professionals are using multipipe environments like the SGI Reality Center to gain insight into their data. These environments typically use multiple graphics pipes drawing into multiple displays, creating a challenge for the application developer as well as for the users to efficiently manage their available graphics resources.

To help application developers and center operators solve these issues, SGI has developped **OpenGL Multipipe SDK** (MPK), a programming interface that leaves view-specific information to be specified outside of the application, at run-time (via a simple ASCII configuration file). MPK thus enables the application to take advantage of the scalability provided by additional pipes and other scalable graphics hardware, as well as to support immersive environments.

MPK provides your application with the following specific features :

- Run-time Configurability
- Run-time Scalability
- Integrated support for Scalable Graphics Hardware
- Integrated support for Stereo and Immersive environments

The ease of configuring your application to accomodate multiple hadware pipes, head-tracking devices and different display areas makes MPK ideal for use with immersive environments, where portability is a premium issue. MPK product components are :

Application Programming Interface

MPK programming model enables programmers to adapt their OpenGL graphics application to support multipipe environments.

MPK essentially provides a C functional interface to the `MPKConfig` data structure, which describes the rendering resources of the application as a hierarchy of hardware graphics pipes (`MPKPipe`), GLX software rendering threads (`MPKWindow`) and OpenGL framebuffer rendering areas (`MPKChannel`), together with the parallel decomposition schemes to be applied on these resources (`MPKCompound`).

Examples are installed under the / *`usr`*/ *`share`*/ *`Multipipe`*/ *`src`* directory.

Configuration File Interface

MPK simple ASCII File Format interface is designed for Reality Center operators to configure MPK applications to run in their environment, by specifying how the framebuffer resources are mapped onto the physical projection areas, together with the parallel decomposition schemes to be applied by the application.

See also

`MPKChannel`, `MPKCompound`, `MPKConfig`, `MPKPipe`, `MPKWindow`

Multipipe SDK 3.2 Reference

Name

MPKPipe - MPKPipe functional interface.

Header File

```
#include <mpk/pipe.h>
```

Synopsis

Creating and Destroying

```
MPKPipe* mpkPipeNew(void );
void     mpkPipeDelete(MPKPipe* pipe);
```

Fields Access

```
void        mpkPipeSetName(MPKPipe* pipe, const char* name);
const char* mpkPipeGetName(MPKPipe* pipe);
void        mpkPipeSetDisplayName(MPKPipe* pipe, const char* name);
const char* mpkPipeGetDisplayName(MPKPipe* pipe);
void        mpkPipeSetUserData(MPKPipe* pipe, void* userData);
void*       mpkPipeGetUserData(MPKPipe* pipe);
int         mpkPipeNWindows(MPKPipe* pipe);
MPKWindow*  mpkPipeGetWindow(MPKPipe* pipe, int i);
void        mpkPipeAddWindow(MPKPipe* pipe, MPKWindow* w);
int         mpkPipeRemoveWindow(MPKPipe* pipe, MPKWindow* w);
MPKConfig*  mpkPipeGetConfig(MPKPipe* pipe);
MPKWindow*  mpkPipeFindWindow(MPKPipe* pipe, const char* name);
MPKChannel* mpkPipeFindChannel(MPKPipe* pipe, const char* name);
MPKPipe*    mpkPipeGetProxy(MPKPipe* p);
```

Operations

```
int  mpkPipeInit(MPKPipe* pipe, int setmon);
void mpkPipeExit(MPKPipe* pipe);
void mpkPipeFreeze(MPKPipe* pipe, int freeze);
void mpkPipeApplyMonitor(MPKPipe* pipe);
void mpkPipeSelectInput(MPKPipe* pipe, long event_mask);
```

Attributes

void **mpkPipeSetAttribute**(MPKPipe* *pipe*, int *attr*, int *value*);
void **mpkPipeUnsetAttribute**(MPKPipe* *pipe*, int *attr*);
void **mpkPipeResetAttribute**(MPKPipe* *pipe*, int *attr*);
int **mpkPipeTestAttribute**(MPKPipe* *pipe*, int *attr*);
int **mpkPipeGetAttribute**(MPKPipe* *pipe*, int *attr*, int* *value*);

Description

The MPKPipe data structure primarily describes the rendering resources within an **MPKConfig** that are assigned to a given hardware rendering pipe. The pipe itself is characterized by the name of its corresponding X11 display, as well as the expected **mono** and **stereo** characteristics (full-screen vs quad-buffer, etc.) to be applied by its rendering threads (**MPKWindow**).

Note that the display sizes corresponding to the various stereo modes can be specified via the **MPKGlobal** attributes, otherwise the values returned by **DisplayWidth**(3X11) and **DisplayHeight**(3X11) will be used.

Function descriptions

Creating and Destroying

mpkPipeNew creates and returns a handle to an MPKPipe.

mpkPipeDelete deletes the passed MPKPipe.

Fields Access

mpkPipeSetName sets the name of the passed MPKPipe to *name*. This is done by copy and not by reference.

mpkPipeGetName returns the name of the passed MPKPipe.

mpkPipeSetDisplayName sets the display name of the passed MPKPipe to *name*. This is done by copy and not by reference.

mpkPipeGetDisplayName returns the display name of the passed MPKPipe.

mpkPipeSetUserData enables the application to specify passthrough data to be transported within the *pipe* structure. Transport is done by reference and not by copy.

mpkPipeGetUserData enables the application to retrieve the passthrough data specified by mpkPipeSetUserData().

mpkPipeNWindows returns the number of MPKWindow in passed *pipe*.

mpkPipeGetWindow returns the *i*th MPKWindow in passed *pipe*.

mpkPipeAddWindow appends MPKWindow *w* to list of windows for passed *pipe*.

mpkPipeRemoveWindow searches for *w* in list of MPKWindow for passed *pipe* and removes it from the list if it is found.

mpkPipeGetConfig returns the parent MPKConfig of passed *pipe*.

mpkPipeFindWindow searches for MPKWindow with specified *name* in the passed MPKPipe and returns the match if found or NULL otherwise.

mpkPipeFindChannel searches for MPKChannel with specified *name* in the passed MPKPipe and returns the match if found or `NULL` otherwise.

mpkPipeGetProxy returns the MPKPipe's Xinerama meta pipe, or NULL if this pipe in not a Xinerama base pipe.

Operations

mpkPipeInit launches the passed MPKPipe and spawns the `MPKWindow` threads. The MPKConfig initialization callbacks are invoked in the order described by the pseudo-code below :

```
invoke the config's pipes initialization callback

for each MPKWindow in the pipe
    launch the window thread, which :

    | invoke the config's windows initialization callbacks
    | for each MPKChannel in the window
    |     invoke the config's channels initialization callback
    | end for
    | enter lifelong loop

end for
```

mpkPipeInit does wait for all `MPKWindow` threads having entered their lifelong loop. It returns the number of threads launched.

If the argument flag *`setmon`* is set, then the shell commands will be invoked that have been specified via mpkConfigSetMonitor(), if any. Note that the config's MPKWindow initialization callback is set by default to mpkWindowCreate().

mpkPipeExit exits the passed MPKPipe and all the `MPKWindow` threads. The MPKConfig exit callbacks are invoked in the order described by the pseudo-code below :

```
for each MPKWindow in the pipe
    exit the window thread, which :

    | for each MPKChannel in the window
    |     invoke the config's channels exit callback
    | end for
    | invoke the config's windows exit callback
    | exit thread

end for

invoke the config's pipes exit callback
```

Note that the config's windows exit callback is set by default to mpkWindowDestroy().

mpkPipeFreeze with a non-zero *freeze* argument causes subsequent mpkConfigFrame() to perform without invoking any rendering callback for the window threads pertaining to the pipe, ie. the windows will be "frozen". Otherwise, frames will be rendered as usual.

mpkPipeApplyMonitor executes the *pipe*'s config monitor shell command, after having set the current X Display to *pipe*'s display name. The previous display is restored before the function returns.

mpkPipeSelectInput loops over all windows of *pipe*, and sets the window's event mask if this window has an input display and an X window drawable. Note that the window's input display is set via mpkWindowOpenDisplay() or mpkWindowSetInputDisplay().

Attributes

See the **MPKGlobal** man page for a description of all MPKPipe attributes and their default or possible values.

mpkPipeSetAttribute sets the value of the MPKPipe attribute specified by *attr* to *value*.

mpkPipeUnsetAttribute unsets the attribute specified by *attr* or, if *attr* is MPK_PATTR_ALL, unsets all attributes for the passed MPKPipe.

mpkPipeResetAttribute resets the attribute specified by *attr* to its corresponding default value or, if *attr* is MPK_PATTR_ALL, it resets all attributes for the passed MPKPipe to their default value.

mpkPipeTestAttribute returns 1 if the attribute specified by *attr* is set for the passed MPKPipe, 0 otherwise.

mpkPipeGetAttribute reads the current value of the attribute specified by *attr* and returns 1 if the attribute is set for the passed MPKPipe, 0 otherwise.

File Format/Defaults

pipe {

# pipe FIELDS description

name "pipe-name"
display "display-name"

attributes { attributes description **}**

# pipe WINDOWS description

window { window-1 description **}**
window { window-2 description **}**

...

}

1. MPKPipe-attributes File Format specification :

attributes {

mono **{** pipe-attribute mono description **}**

stereo { **pipe-attribute stereo description }**

}

2. MPKPipe-attribute-mono File Format specification :

mono {

width	w
height	h

}

3. MPKPipe-attribute-stereo File Format specification :

stereo {

type	stereo-type
width	w
height	h
offset	o

}

Notes

`w`, `h` and `o` must be integer.

`stereo-type` description accepts only the following File Format identifiers : **none** [default] **quad**, **rect**, **top**, **bottom** and **user**. If no stereo-type is specified, **quad** is used.

See also

`MPKChannel`, `MPKConfig`, `MPKGlobal`, `MPKWindow`

Multipipe SDK 3.2 Reference

Name

MPKWindow - MPKWindow functional interface.

Header File

```
#include <mpk/window.h>
```

Synopsis

Creating and Destroying

```
MPKWindow* mpkWindowNew(void );
void       mpkWindowDelete(MPKWindow* window);
```

Fields Access

```
void        mpkWindowSetName(MPKWindow* window, const char* name);
const char* mpkWindowGetName(MPKWindow* window);
void        mpkWindowSetRunon(MPKWindow* window, int cpu);
int         mpkWindowGetRunon(MPKWindow* window);
void        mpkWindowSetViewport(MPKWindow* window, float vp[4]);
void        mpkWindowGetViewport(MPKWindow* window, float vp[4]);
int         mpkWindowNChannels(MPKWindow* window);
MPKChannel* mpkWindowGetChannel(MPKWindow* window, int i);
void        mpkWindowAddChannel(MPKWindow* window, MPKChannel* c);
int         mpkWindowRemoveChannel(MPKWindow* window, MPKChannel* c);
MPKPipe*    mpkWindowGetPipe(MPKWindow* window);
MPKConfig*  mpkWindowGetConfig(MPKWindow* window);
MPKChannel* mpkWindowFindChannel(MPKWindow* window, const char* name);
MPKWindow*  mpkWindowGetProxy(MPKWindow* w);
```

Attributes

```
void mpkWindowSetAttribute(MPKWindow* window, int attr, int value);
void mpkWindowUnsetAttribute(MPKWindow* window, int attr);
void mpkWindowResetAttribute(MPKWindow* window, int attr);
int  mpkWindowTestAttribute(MPKWindow* window, int attr);
int  mpkWindowGetAttribute(MPKWindow* window, int attr, int* value);
```

Callbacks

```
void mpkWindowSetEventCB(MPKWindow* window, int which, MPKWindowEventCB cb);
void mpkWindowSetDrawCB(MPKWindow* window, int which, MPKWindowDrawCB cb);
void mpkWindowSetCullCB(MPKWindow* window, int which, MPKWindowCullCB cb);
```

```
MPKWindowEventCB mpkWindowGetEventCB(const MPKWindow* window, int which);
MPKWindowDrawCB  mpkWindowGetDrawCB(const MPKWindow* window, int which);
MPKWindowCullCB  mpkWindowGetCullCB(const MPKWindow* window, int which);
```

Operations

```
int   mpkWindowInit(MPKWindow* window);
void  mpkWindowExit(MPKWindow* window);
void  mpkWindowFreeze(MPKWindow* window, int freeze);
void  mpkWindowResize(MPKWindow* window);
void  mpkWindowApplyViewport(MPKWindow* window);
void  mpkWindowUpdatePixelViewport(MPKWindow* window);
void  mpkWindowSetPixelViewport(MPKWindow* window, int pvp[4]);
void  mpkWindowGetPixelViewport(MPKWindow* window, int pvp[4]);
int   mpkWindowGetMode(MPKWindow* window);
void  mpkWindowSetUserData(MPKWindow* window, void* userData);
void* mpkWindowGetUserData(MPKWindow* window);
```

Events

```
void mpkWindowSelectInput(MPKWindow* window, long event_mask);
void mpkWindowProcessEvent(MPKWindow* window, MPKEvent* event);
```

X11/GLX Interface

```
void          mpkWindowCreate(MPKWindow* window);
void          mpkWindowDestroy(MPKWindow* window);
void          mpkWindowOpenDisplay(MPKWindow* window);
void          mpkWindowCloseDisplay(MPKWindow* window);
void          mpkWindowCreateDrawable(MPKWindow* window);
void          mpkWindowDestroyDrawable(MPKWindow* window);
void          mpkWindowMapDrawable(MPKWindow* window);
void          mpkWindowCreateContext(MPKWindow* window);
void          mpkWindowDestroyContext(MPKWindow* window);
int           mpkWindowMakeCurrent(MPKWindow* window);
int           mpkWindowMakeCurrentNone(MPKWindow* window);
void          mpkWindowSwapBuffers(MPKWindow* window);
void          mpkWindowSetDisplay(MPKWindow* window, Display* display);
Display*      mpkWindowGetDisplay(MPKWindow* window);
void          mpkWindowSetInputDisplay(MPKWindow* window, Display* display);
Display*      mpkWindowGetInputDisplay(MPKWindow* window);
void          mpkWindowSetParent(MPKWindow* window, XID parent);
```

XID	**mpkWindowGetParent**(MPKWindow* *window*);
void	**mpkWindowSetScreen**(MPKWindow* *window*, int *screen*);
int	**mpkWindowGetScreen**(MPKWindow* *window*);
GLXFBConfig*	**mpkWindowChooseFBConfig**(MPKWindow* *window*, int* *nitems*);
void	**mpkWindowSetFBConfig**(MPKWindow* *window*, GLXFBConfig *fbConfig*);
GLXFBConfig	**mpkWindowGetFBConfig**(MPKWindow* *window*);
XVisualInfo*	**mpkWindowChooseVisual**(MPKWindow* *window*);
void	**mpkWindowSetVisual**(MPKWindow* *window*, XVisualInfo* *visInfo*);
XVisualInfo*	**mpkWindowGetVisual**(MPKWindow* *window*);
void	**mpkWindowSetPixmap**(MPKWindow* *window*, Pixmap *pixmap*);
Pixmap	**mpkWindowGetPixmap**(MPKWindow* *window*);
void	**mpkWindowSetDrawable**(MPKWindow* *window*, XID *drawable*);
XID	**mpkWindowGetDrawable**(MPKWindow* *window*);
void	**mpkWindowSetContext**(MPKWindow* *window*, GLXContext *context*);
GLXContext	**mpkWindowGetContext**(MPKWindow* *window*);

Description

The MPKWindow data structure primarily describes a thread within an **`MPKPipe`**, potentially associated with an X11 **Drawable** for rendering. After its creation by **mpkWindowInit** the thread loops through the following sequence:

```
wait for next mpkConfigFrameBegin()

if (first time)
    for each MPKChannel [of the window]
         invoke the channel's cull-init callback function
         invoke the channel's draw-init callback function
    end for
end if

invoke the window's draw-update callback function

for each MPKChannel [of the window]
    if ( in mono )
        for eye MPK_EYE_CYCLOP
            invoke the channel's cull-update callback function
            invoke the channel's draw-clear callback function
            invoke the channel's draw-update callback function
        end for
    else ( in stereo )
        for eye MPK_EYE_LEFT and eye MPK_EYE_RIGHT
            invoke the channel's cull-update callback function
            invoke the channel's draw-clear callback function
            invoke the channel's draw-update callback function
        end for
    end if
end for

synchronize mpkWindowSwapBuffers (through mpkConfigFrameEnd())
```

Function descriptions

Creating and Destroying

mpkWindowNew creates and returns a handle to an MPKWindow.

mpkWindowDelete deletes the passed MPKWindow.

Fields Access

mpkWindowSetName sets the name of the passed MPKWindow to *name*. This is done by copy and not by reference.

mpkWindowGetName returns the name of the passed MPKWindow.

mpkWindowSetRunon specifies the *cpu* on which to assign the passed MPKWindow thread. In addition, the following symbolic values can be used:

MPK_RUNON_AUTO The window threads will be automatically placed on a CPU close to their respective graphics pipe, if possible.

MPK_RUNON_FREE All threads are free to execute on whatever processor the system deems suitable.

MPK_UNDEFINED The thread placement is defined by the config's runon value.

mpkWindowGetRunon return value indicates on which cpu the passed MPKWindow thread is assigned. A negative value means that the thread is free to execute on whatever processor the system deems suitable.

mpkWindowSetViewport sets the fractional viewport of the passed MPKWindow to the values pointed to by *vp*.

mpkWindowGetViewport reads the fractional viewport of the passed MPKWindow in *vp*.

mpkWindowNChannels returns the number of MPKChannel in passed *window*.

mpkWindowGetChannel returns the *i*th MPKChannel in passed *window*.

mpkWindowAddChannel appends *c* to list of channels for passed *window*.

mpkWindowRemoveChannel searches for *c* in list of MPKChannel for passed *window* and removes it from the list if it is found.

mpkWindowGetPipe returns the parent MPKPipe of the passed MPKWindow.

mpkWindowGetConfig returns the parent MPKConfig of the passed MPKWindow.

mpkWindowFindChannel searches for MPKChannel with specified *name* in the passed MPKWindow and returns the match if found or NULL otherwise.

mpkWindowGetProxy returns the MPKWindow's Xinerama meta window, or NULL if this window is not a Xinerama base window.

Attributes

See the **MPKGlobal** man page for a description of all MPKWindow attributes and their default or possible values.

mpkWindowSetAttribute sets the value of the MPKWindow attribute specified by *`attr`* to *`value`*.

mpkWindowUnsetAttribute unsets the attribute specified by *`attr`* or, if *`attr`* is `MPK_WATTR_ALL`, unsets all attributes for the passed MPKWindow.

mpkWindowResetAttribute resets the attribute specified by *`attr`* to its corresponding default value or, if *`attr`* is `MPK_WATTR_ALL`, it resets all attributes for the passed MPKWindow to their default value.

mpkWindowTestAttribute returns 1 if the attribute specified by *`attr`* is set for the passed MPKWindow, 0 otherwise.

mpkWindowGetAttribute reads the current value of the attribute specified by *`attr`* and returns 1 if the attribute is set for the passed MPKWindow, 0 otherwise.

Callbacks

mpkWindowSetEventCB sets the MPKWindow event callback specified by *`which`* to the passed function, of type:

void (***MPKWindowEventCB**)(MPKWindow*, MPKEvent*);
Accepted values for *`which`* are

MPK_WINDOW_EVENTCB_ANY, **MPK_WINDOW_EVENTCB_CONFIGURE**, **MPK_WINDOW_EVENTCB_EXPOSE**, **MPK_WINDOW_EVENTCB_KEYBOARD**, **MPK_WINDOW_EVENTCB_MOUSE**, **MPK_WINDOW_EVENTCB_BUTTON**, **MPK_WINDOW_EVENTCB_EXIT**

The default config event callback, mpkConfigHandleEvents() is called at the end of each frame and will invoke the `MPK_WINDOW_EVENTCB_ANY` callback for each event received on that window.

By default the `MPK_WINDOW_EVENTCB_ANY` callback is set to mpkWindowProcessEvent().

mpkWindowSetDrawCB sets the MPKWindow draw callback specified by *`which`* to the passed function, of type:

void (***MPKWindowDrawCB**)(MPKWindow*);
Accepted values for *`which`* are

MPK_WINDOW_DRAWCB_INIT_X, **MPK_WINDOW_DRAWCB_INIT_GL**, **MPK_WINDOW_DRAWCB_EXIT_X**, **MPK_WINDOW_DRAWCB_EXIT_GL**, **MPK_WINDOW_DRAWCB_UPDATE**, **MPK_WINDOW_DRAWCB_RESIZE**

The init and exit callbacks are specifying the functions which are invoked to initialise and exit X11 and GLX/OpenGL.

The update callback specifies a function to be invoked once a frame by the window thread, prior to any rendering.

The resize callback is invoked everytime a window resize should be performed. Currently, this can happen only in mpkConfigChangeMode(). It is called when the lightweight stereo switch is performed and the pipe resolution is different in mono and stereo mode. The default function to be invoked is mpkWindowResize().

mpkWindowSetCullCB sets the MPKWindow cull callback specified by *`which`* to the passed function, of type:

void (***MPKWindowCullCB**)(MPKWindow*);
Accepted values for *`which`* are

MPK_WINDOW_CULLCB_INIT, **MPK_WINDOW_CULLCB_EXIT**, **MPK_WINDOW_CULLCB_UPDATE**

The init and exit callbacks are specifying the functions which are invoked to initialise and exit windows used for culling.

The update callback specifies a function to be invoked once a frame by the window thread, prior to any culling.

mpkWindowGetEventCB returns the MPKWindow event callback function specified by *which*. Accepted values for *which* are

MPK_WINDOW_EVENTCB_ANY, **MPK_WINDOW_EVENTCB_CONFIGURE**, **MPK_WINDOW_EVENTCB_EXPOSE**, **MPK_WINDOW_EVENTCB_KEYBOARD**, **MPK_WINDOW_EVENTCB_MOUSE**, **MPK_WINDOW_EVENTCB_BUTTON**, **MPK_WINDOW_EVENTCB_EXIT**

mpkWindowGetDrawCB returns the MPKWindow draw callback function specified by *which*. Accepted values for *which* are

MPK_WINDOW_DRAWCB_INIT_X, **MPK_WINDOW_DRAWCB_INIT_GL**, **MPK_WINDOW_DRAWCB_EXIT_X**, **MPK_WINDOW_DRAWCB_EXIT_GL**, **MPK_WINDOW_DRAWCB_UPDATE**, **MPK_WINDOW_DRAWCB_RESIZE**

mpkWindowGetCullCB returns the MPKWindow cull callback function specified by *which*. Accepted values for *which* are

MPK_WINDOW_CULLCB_INIT, **MPK_WINDOW_CULLCB_EXIT**, **MPK_WINDOW_CULLCB_UPDATE**

Operations

mpkWindowInit launches the passed MPKWindow and spawns the MPKWindow loop thread, which executes according to the pseudo-code below:

```
invoke the config's windows initialization callback
for each MPKChannel in the window
    invoke the config's channels initialization callback
end for
enter lifelong loop
```

mpkWindowInit() does block until the window entered it's lifelong loop. It returns the number of threads created (see `MPK_WATTR_HINTS_THREAD` attribute).

mpkWindowExit exits *window*, which causes the window thread to execute the pseudo-code below prior to exiting:

```
for each MPKChannel in the window
    invoke the config's channels exit callback
end for
invoke the config's windows exit callback
```

Note that the config's windows exit callback is set by default to mpkWindowDestroy().

mpkWindowFreeze with a non-zero *freeze* argument causes subsequent mpkConfigFrame() to perform without invoking any rendering callback for the passed MPKWindow ie. the window will be "frozen". Otherwise, frames will be rendered as usual.

mpkWindowResize resizes the given *window* according to the current display size and fractional viewport.

mpkWindowApplyViewport applies the latest pixel viewport specified for the passed MPKWindow as an OpenGL viewport and scissor area.

mpkWindowUpdatePixelViewport forces recomputation of the window pixel viewport. If a parent window has been specified for the window, then the computation will use the parent window's dimensions, otherwise it will use the [stereo-dependent] parent pipe's display dimensions to compute the window's pixel viewport from its fractional viewport:

```
#define IRND(a) ((int)((a)+.5))

// compute first pixel position of the window
window.pvp[0] = IRND(window.vp[0] * pipe.width);
window.pvp[1] = IRND(window.vp[1] * pipe.height);

// compute last pixel position of the window
window.pvp[2] = IRND((window.vp[0]+window.vp[2]) * pipe.width);
window.pvp[3] = IRND((window.vp[1]+window.vp[3]) * pipe.height);

// compute window's dimension
window.pvp[2] -= window.pvp[0];
window.pvp[3] -= window.pvp[1];
```

This method honors positions over dimensions in order to ensure adjacency whenever possible, e.g. on a 1280x1024 display:

```
vp(1): [0.     0. 0.3333 1. ]    pvp(1): [0   0 427 1024]
vp(2): [0.3333 0. 0.3333 1. ]    pvp(2): [427 0 426 1024]
```

Note that in stereo mode type MPK_STEREO_RECT the value of the MPKGlobal variable **MPK_DATTR_FULLSTEREO_OFFSET** is added to the window's height, whereas in mode MPK_STEREO_BOTH this offset is added to its vertical position.

The update is propagated immediately to each of the window's MPKChannels.

mpkWindowSetPixelViewport sets values for the passed MPKWindow pixel viewport. Change is immediately propagated to each of the window's MPKChannel.

mpkWindowGetPixelViewport reads the latest updated pixel viewport for the passed MPKWindow in *pvp*.

mpkWindowGetMode returns the current current stereo mode of the passed MPKWindow as either `MPK_STEREO_NONE`, `MPK_STEREO_RECT`, `MPK_STEREO_QUAD`, `MPK_STEREO_TOP`, `MPK_STEREO_BOT` or `MPK_STEREO_USER`.

mpkWindowSetUserData enables the application to specify passthrough data to be transported within the *window* structure. Transport is done by reference and not by copy.

mpkWindowGetUserData enables the application to retrieve the passthrough data specified by mpkWindowSetUserData().

Events

mpkWindowSelectInput sets *window*'s event mask if this window has an input display and an X window drawable. Note that the window's input display is set via mpkWindowOpenDisplay() or mpkWindowSetInputDisplay().

mpkWindowProcessEvent returns if the passed *event*'s X window does not match the passed MPKWindow drawable. Otherwise mpkWindowProcessEvent invokes the user-specified related `MPKEvent callback`, if any.

Combined with the **MPK_WINDOW_EVENTCB_ANY** event callback function, mpkWindowProcessEvent() enables customization of the event-handling for an MPK-application, as shown by the example below:

```
void initWindow( MPKWindow *w )
{
    mpkWindowSetEventCB( w, MPK_WINDOW_EVENTCB_ANY, windowEvent );
    mpkWindowSetEventCB( w, MPK_WINDOW_EVENTCB_KEYBOARD, windowKB );
}

void windowEvent( MPKWindow *w, MPKEvent *event )
{
    MPKEventXData *data = (MPKEventXData *)mpkEventGetData(event);
    switch ( data->x->type )
    {
        // Use my own event processing for these events
        case ClientMessage:
            myProcessEvent( event );
            break;

        // Use MPK event processing for other events, eg.
        // keyboard events.
        default:
            mpkWindowProcessEvent( w, event );
    }
}

void windowKB( MPKWindow *w, MPKEvent *event )
{
    // Will be invoked by MPK upon keyboard events
    MPKEventXData *data = (MPKEventXData *)mpkEventGetData(event);

    if ( data->keyboard.state == MPK_PRESS )
        printf( "key %d pressed\n", data->keyboard.key );
}
```

X11/GLX Interface

mpkWindowCreate performs the following operations:

```
mpkWindowOpenDisplay( window );

GLXFBConfig *fbconfig = mpkWindowChooseFBConfig( window, &n );
mpkWindowSetFBConfig( window, fbconfig[0] );

mpkWindowCreateDrawable( window );
mpkWindowMapDrawable( window );
```

If the GLXFBConfig interface is not supported, then the corresponding XVisualInfo interface will be used. Note that MPKConfig's windows init callback is set by default to mpkWindowCreate.

mpkWindowDestroy performs the following operations:

```
mpkWindowDestroyDrawable( window );
mpkWindowCloseDisplay( window );

mpkWindowSetParent( window, NULL );
mpkWindowSetScreen( window, 0 );
mpkWindowSetFBConfig( window, NULL );
```

If the GLXFBConfig interface is not supported, then the corresponding XVisualInfo interface will be used. Note that MPKConfig's windows exit callback are set by default to mpkWindowDestroy.

mpkWindowOpenDisplay performs the following operations:

```
MPKPipe *p = mpkWindowGetPipe(window);
Display *display = XOpenDisplay( mpkPipeGetDisplayName(p) );

mpkWindowSetDisplay( window, display );
mpkWindowSetScreen( DefaultScreen(display) );

mpkWindowUpdatePixelViewport( window );
```

Note that mpkWindowOpenDisplay does not invoke mpkWindowSetParent(). mpkWindowOpenDisplay may use a lock to open the display connection for Xinerama-aware and Xinerama-unaware windows (see mpkGlobalSetXinerama()).

mpkWindowCloseDisplay disconnects an MPKWindow from the X server.

mpkWindowCreateDrawable creates an X Drawable that matches the passed MPKWindow attributes and parent XID, if one has been specified via mpkWindowSetParent().

If the `MPK_WATTR_HINTS_DRAWABLE` attribute is set to `MPK_GLX_PIXMAP` and a Pixmap has been specified via mpkWindowSetPixmap() then mpkWindowCreateDrawable will use glXCreateGLXPixmap(3G) on the specified Pixmap.

Note that mpkWindowCreateDrawable does not map the resulting drawable (see mpkWindowMapDrawable()).

mpkWindowDestroyDrawable destroys the *`window`*'s Drawable and Pixmap.

mpkWindowMapDrawable maps the *`window`*'s drawable, if the `MPK_WATTR_HINTS_DRAWABLE` attribute is set to `MPK_GLX_WINDOW`. Otherwise it simply returns.

mpkWindowCreateContext creates an GLXContext that matches the passed MPKWindow attributes, in particular the `MPK_WATTR_HINTS_DIRECT` attribute.

mpkWindowDestroyContext destroys the *`window`*'s GLXContext.

mpkWindowMakeCurrent attaches the *`window`*'s GLXContext to its Drawable.

mpkWindowMakeCurrentNone releases the *`window`*'s current GLXContext.

mpkWindowSwapBuffers exchanges the front and back buffers of a double-buffered MPKWindow provided its `MPK_WATTR_HINTS_DOUBLEBUFFER` attribute is not set to 0.

mpkWindowSetDisplay sets the X11 Display of the passed MPKWindow to *`display`*.

mpkWindowGetDisplay returns the current X11 Display of the passed MPKWindow.

mpkWindowSetInputDisplay sets the X11 Display used for receiving events of the passed MPKWindow to *`display`*. Note that this display is used by MPK from the application thread. This means that in fork execution mode the Display has to be opened from the application process, not from the window rendering process.

mpkWindowGetInputDisplay returns the current X11 Input Display of the passed MPKWindow.

mpkWindowSetParent sets the parent X11 Window of the passed MPKWindow to *`parent`*. This information is used by mpkWindowCreate() when creating the X11 Window and mpkWindowUpdatePixelViewport() when computing the pixel viewport of the MPKWindow from its fractional viewport.

mpkWindowGetParent returns the parent X11 Window of the passed MPKWindow, NULL if the parent is the root window of the Display.

mpkWindowSetScreen sets the X11 Screen of passed MPKWindow to *`screen`*.

mpkWindowGetScreen returns the passed MPKWindow's X11 Screen.

mpkWindowChooseFBConfig returns a list of GLX frame buffer configurations that match the passed MPKWindow attributes. *`nitems`* returns the number of elements in the list. Use XFree to free the memory returned by this function.

mpkWindowSetFBConfig sets the frame buffer configuration of the passed MPKWindow to the passed GLXFBConfig, and its visual to the corresponding XVisualInfo if a match can be found. This information is used by mpkWindowCreateContext().

mpkWindowGetFBConfig returns the current frame buffer configuration of the passed MPKWindow.

mpkWindowChooseVisual returns a visual that matches the passed MPKWindow attributes. Use XFree to free the memory returned by this function.

mpkWindowSetVisual sets the visual of the passed MPKWindow to the passed XVisualInfo, to be used in mpkWindowCreateContext(). Note that this information gets preempted by any Frame Buffer configuration specified via mpkWindowSetFBConfig().

mpkWindowGetVisual returns the current visual of the passed MPKWindow.

mpkWindowSetPixmap specifies an optional X11 Pixmap to be used by mpkWindowCreateDrawable() when the `MPK_WATTR_HINTS_DRAWABLE` attribute is set to `MPK_GLX_PIXMAP`.

mpkWindowGetPixmap returns the current X11 Pixmap used by the passed MPKWindow.

mpkWindowSetDrawable sets the current Drawable of the passed MPKWindow, to be used by mpkWindowMapDrawable(), mpkWindowProcessEvent(), mpkWindowMakeCurrent() and mpkWindowSwapBuffers().

mpkWindowGetDrawable returns the current X11 Drawable of the passed MPKWindow.

mpkWindowSetContext sets the current GLXContext of the passed MPKWindow, to be used by mpkWindowMakeCurrent() and mpkWindowSwapBuffers().

mpkWindowGetContext returns the current GLXContext of the passed MPKWindow.

File Format/Defaults

1. MPKWindow File Format specification:

```
window {
                                # window FIELDS description

                        name     "window-name"
                        runon    processor-id

                        viewport [ xf, yf, wf, hf ]

                                # window ATTRIBUTES description

                             attributes { attributes description }

                                # window CHANNELS description

                             channel { channel-1 description }
                             channel { channel-2 description }
                        ...
}
```

2. MPKWindow-attributes File Format specification:

```
attributes {
                      hints       { window-attribute hints description }
                      planes      { window-attribute planes description }
                      transparent { window-attribute transparent description }
}
```

3. MPKWindow-attribute-hints File Format specification:

```
hints {
        visual          visual-type
        drawable        drawable-type
        caveat          visual-caveat
        direct          y/n

        thread          y/n
        xinerama        y/n
        event           none/input/inputOutput
        decoration      y/n
        transparent     y/n
        X-renderable    y/n
        rgba            y/n
        doublebuffer    y/n
        stereo          y/n # quad-buffer only

        largest         y/n # pbuffer only
        preserved       y/n # pbuffer only

}
```

4. MPKWindow-attribute-planes File Format specification:

```
planes {
        level       0
        depth       1
        stencil     0
        samples     0
        auxiliary   0
        color       0
        rgba        [ 1, 1, 1, 0 ]
        accum       [ 0, 0, 0, 0 ]

}
```

5. MPKWindow-attribute-transparent File Format specification:

```
transparent {
        index  0
        rgba   [ 0, 0, 0, 0 ]

}
```

Notes

viewport parameters are relative to the parent pipe display size, and therefore their values should be in the range 0.0 to 1.0

visual attribute hint specification accepts only the following File Format identifiers: **true-color** [default], **pseudo-color**, **direct-color**, **static-color**, **static-gray** or **grayscale**.

drawable attribute hint specification accepts only the following File Format identifiers: **none**, **window** [default], **pbuffer** and **pixmap**.

caveat attribute hint specification accepts only the following File Format identifiers: **none** [default], **slow** and **non-conformant**.

See also

`MPKChannel`, `MPKConfig`, `MPKGlobal`, `MPKPipe`