```
//--------------------------------------------------------------------------------
//  Filename: gfxICEEngine.h
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//  Infinite Canvas Engine (ICE) Application Programming Interface (API)
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//  TABLE OF CONTENTS
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//  + Introduction to the ICE Drawing Engine
//  + Quick Function Reference Guide
//  + The ICE Drawing Engine API
//      + ICE Drawing Engine API version number
//      + Fundamental Data Types
//      + Memory Allocation
//      + ICE Instances
//      + ICE Canvases
//      + ICE Layers
//      + Drawing
//      + Rendering Canvases and Layers
//      + Transforming Canvases and Layers
//      + Selection
//      + Undo and Redo
//      + Saving and Loading ICE Canvases
//--------------------------------------------------------------------------------


//--------------------------------------------------------------------------------
//  INTRODUCTION TO THE ICE DRAWING ENGINE
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//  The Infinite Canvas Engine (ICE) Advantage
//
//  The ICE Drawing Engine uses a new approach for digital drawing which allows the
//  creation of detailed and textured artwork typical of pixel-based systems such as
//  Adobe Photoshop but with the infinite, artifact-free scalability and small memory
//  footprint of vector-based (SVG) systems such as Adobe Illustrator. ICE provides
//  the following advantages over today's digital drawing systems and representations:
//  + The ability to create graphical elements that exhibit the richness of pixels
//    with the artifact-free scalability and small size of SVG.
//  + An infinite canvas in both spatial (x and y) extent and scale (z) thereby
//    providing seamless infinite zoom and infinite definition, features not
//    available in any other system.
//  + A new multi-scale rendering algorithm for image textures that enables a wide
//    variety of scanned paper textures to be rendered at any scale (infinite paper).
//  + Very small file sizes, providing an ideal solution for memory- or bandwidth-
//    limited environments.
//  + High-quality tunable anti-aliased rendering. Support for a wealth of primitives
//    such as variable width textured strokes with intricate sub-pixel features.
//  + Realtime fast rendering by exploiting the massive parallelism inherent in GPUs
//    enabling immediate feedback during drawing and interactive canvas
//    transformations.
//  + An enabling technology for a novel representation of existing pixel-based
//    images that provides the intricate detail and texture that pixels allow but
//    with the small size and artifact-free transformations only available with SVG.
//  + An enabling technology for a novel representation of maps (and possibly other
//    ubiquitous graphical forms) that provides not only better appearance, size,
//    quality, and performance characteristics but functionality (such as seamless
//    infinite zoom) previously not possible.
//  + An internal representation that enables stylization for different looks and
//    levels of abstraction.
//  + Support for layers, an indispensable metaphor for content creation, and map
//    design principles.
//  + A superior drawing experience for content creators:
//      + A natural, smooth, pencil-on-paper feel, where pen strokes match the
```

```
//        artist's input precisely
//      + Immediate visual feedback during drawing due to on-the-fly curve fitting
//        and fast rendering
//      + A rich assortment of pressure-sensitive, customizable, scalable, textured
//        pens
//      + An organic, free flowing drawing experience where ideas can be expanded
//        without bounds, both spatially and in z, due to the infinite canvas which
//        requires no upfront planning regarding size and resolution
//      + The ability to create seamless, hierarchical, multi-resolution, story
//        within a story experiences not possible with the current state of the art
//        systems such as Adobe Photoshop and Adobe Illustrator.
//
//  An Overview of the ICE Drawing Engine
//
//  + A ICE instance is a thread-safe instantiation of the ICE Drawing Engine.
//    Because ICE instances are completely independent, they can be assigned to
//    separate cores and run concurrently.
//  + A ICE instance has a set of canvases. A ICE canvas has
//      + An opaque paper background, which can be defined as a solid color, an image-
//        based paper texture, or a procedural paper texture. If multi-scale rendering
//        is enabled, paper textures exhibit a fractal-like quality when rendered so
//        that, when a canvas is zoomed, the individual features of a paper texture
//        are scaled while the characteristic structure of the paper texture is
//        maintained. The multi-scale rendering method eliminates the aliasing and
//        pixelization artifacts that are typical when scaling images, thereby
//        allowing applications to use a wide variety of scanned paper textures for
//        the paper background. Both RGB and grayscale paper textures are supported.
//        Paper textures can be colorized with a specified RGB value. In addition,
//        the strength of the individual features in a paper texture can be modulated
//        between 0 and 1.
//      + An ordered list of canvas layers upon which to draw, where
//          + Layers reside above the paper background
//          + Drawing occurs on the currently active layer, with the active layer set
//            by the application
//          + Layers can be added, removed, cleared, re-ordered, and merged
//          + Layers can be individually translated, rotated, and scaled
//          + Layer visibility can be turned on or off
//          + Layer opacity can be set between zero (fully transparent) and one
//            (fully opaque)
//  + A current pen is maintained for each ICE canvas and used for drawing on the
//    active layer of a canvas. Pen attributes such as type, color, width, and
//    pressure sensitivity can be set and customized by the application.
//  + A selection buffer is maintained for each ICE instance to enable cut, copy,
//    and paste operations
//  + A history list of reversible operations is maintained for each canvas to
//    facilitate infinite undo and redo. The reversible operations include 1) drawing
//    operations (i.e., adding freehand strokes, stroked geometric shapes, and images
//    to the canvas), 2) cutting and pasting selections, 3) adding, deleting,
//    clearing, moving, and merging layers, and 4) applying transformations to layers.
//    The ICE Drawing Engine does not directly support undo and redo for other
//    operations such as changing drawing or layer attributes, applying
//    transformations to canvases, or creating and destroying canvases; it is the
//    application's responsibility to provide the necessary support for undo and redo
//    for such operations if so desired.
//  + When a canvas is rendered, each layer is first rendered independently by
//    compositing its drawing operations in the order in which they were drawn. Next,
//    each rendered layer, modulated by its opacity, is composited (in order) onto the
//    canvas paper using the conventional graphics over operator (see Porter and Duff,
//    1984); this is equivalent to blending layers using the 'Normal' blend mode in
//    Adobe Photoshop.
//  + A canvas is viewed, drawn on, and manipulated through a canvas viewport. A
//    canvas viewport is specified by a width w and height h in pixels and an offset
//    from the bottom-left corner of a display window. Note that the ICE Drawing
//    Engine assumes a Cartesian coordinate system; the origin (0,0) of a canvas
//    viewport lies at its bottom-left corner, with x increasing from 0 to w-1 along
//    its bottom edge and y increasing from 0 to h-1 along its left edge. When drawing
```

```
//    on or manipulating a canvas, positions are specified by floating point pixel
//    values in canvas viewport coordinates. The use of floating point pixel values
//    enables sub-pixel accuracy during drawing, allowing applications to take full
//    advantage of the sub-pixel accuracy available from most graphics tablets.
//  + A canvas has both infinite spatial extent and infinite zoom (subject to floating
//    point representation limits for canvas transformations); the rectangular region
//    of the canvas that is viewed through the canvas viewport is determined by a
//    canvas transform.
//  + The canvas transform positions, scales, and rotates the canvas relative to the
//    canvas viewport. When the canvas transform is set to the identity, the canvas
//    origin maps to the bottom-left corner of the canvas viewport, its x- and y-axes
//    map to the bottom and left edges of the canvas viewport, respectively, and one
//    unit in the canvas corresponds to one pixel in the canvas viewport.
//  + Both on-screen and off-screen rendering of a canvas are supported. For on-
//    screen rendering, a rectangular region of the canvas is rendered to the canvas
//    viewport. The specific rectangular region is determined by the canvas transform.
//    For off-screen rendering, a rectangular region of the canvas, specified in
//    canvas viewport coordinates, is rendered to an image. The size of the off-screen
//    image (i.e., its dimensions in image pixels) is set independently by specifying
//    the width of the off-screen image in pixels (the image height is computed by the
//    ICE Drawing Engine to preserve the aspect ratio of the rectangular region);
//    this permits any rectangular region of the canvas to be rendered off-screen at
//    any resolution.
//  + Off-screen rendering of individual layers is also supported and provides
//    compatibility with pixel-based applications that support layers. For example,
//    individual layers can be rendered off-screen (by the ICE Drawing Engine) and
//    written to disk (by the application) in Adobe's PSD format. The resultant PSD
//    file can then be read into Adobe Photoshop for further processing.
//  + Methods are provided so that an entire canvas or a specified rectangular region
//    of a canvas can be saved to and reloaded from a file. Both lossless and lossy
//    compression are supported.
//
//  Technical Requirements and Assumptions
//
//  + The ICE Drawing Engine requires OpenGL 2.0 or greater and the OpenGL
//    Extension Wrangler (GLEW) library (glew.h contains the OpenGL API). Note that
//    GLSL (GL Shading Language), which is used by the ICE Drawing Engine, was
//    formally included into the OpenGL 2.0 core by the OpenGL ARB.
//  + Both on-screen and off-screen rendering make use of the graphics processing
//    unit (GPU)
//      + On-screen rendering renders a canvas into its canvas viewport in the OpenGL
//        back buffer. The application is responsible for transferring the canvas
//        viewport to the OpenGL front buffer for display. The transfer can be
//        performed using a bitblt or by swapping the OpenGL back and front buffers.
//      + Off-screen rendering renders a canvas or layer into a GFX_Image which is
//        then provided to the application. The application is responsible for
//        freeing the GFX_Image when it is no longer needed.
//      + Both on-screen and off-screen rendering make use of the OpenGL back buffer.
//        Consequently, the width and height of the canvas viewport should be no
//        bigger than the width and height of the OpenGL back buffer, respectively.
//        For off-screen rendering, the size of the image is not limited by the size
//        of the canvas viewport. However, off-screen rendering is more efficient
//        when the canvas viewport is as large as possible (i.e., the same size as
//        the OpenGL back buffer).
//  + The application must configure the OpenGL back buffer to enable RGBA rendering
//    with support for both a depth buffer (with a minimum bit depth of 24 bits) and
//    a stencil buffer (with a minimum bit depth of 1 bit). Selecting hardware
//    accelerated pixel formats that satisfy these requirements will provide optimal
//    performance.
//  + Due to the high cost of OpenGL state changes in most OpenGL implementations,
//    applications must carefully manage OpenGL state to achieve optimal performance.
//    The ICE Drawing Engine assumes that OpenGL is in its default state when
//    rendering (i.e., when gfxRenderCanvas or gfxRenderToImage is invoked);
//    applications can use various OpenGL functions, such as glPushAttrib and
//    glPopAttrib, to meet this expectation. To improve performance, the ICE
//    Drawing Engine uses as little OpenGL state as possible. Any OpenGL state used
```

```
//    by the ICE Drawing Engine will be reset to its default value after rendering.
//  + Drawing with the ICE Drawing Engine is greatly enhanced by the use of a
//    graphics tablet with pressure sensitive pen input and sub-pixel accuracy
//-------------------------------------------------------------------------------


//-------------------------------------------------------------------------------
//  QUICK FUNCTION REFERENCE GUIDE
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  ICE Instances
//
//  void *gfxCreateICEInst (GFX_ICEInstAttrs *instAttrs)
//  void gfxDestroyICEInst (void *ICEInst)
//  void gfxSetICEInstAttrs (void *ICEInst, GFX_ICEInstAttrs *instAttrs)
//  void gfxGetICEInstAttrs (void *ICEInst, GFX_ICEInstAttrs *instAttrs)
//  GFX_I32 gfxGetError (void *ICEInst)
//
//
//  ICE Canvases
//
//  void *gfxCreateCanvas (void *ICEInst, GFX_CanvasAttrs *canvasAttrs)
//  void gfxDestroyCanvas (void *canvas)
//  void gfxSetCanvasAttrs (void *canvas, GFX_CanvasAttrs *canvasAttrs)
//  void gfxGetCanvasAttrs (void *canvas, GFX_CanvasAttrs *canvasAttrs)
//
//
//  ICE Layers
//
//  void gfxAddLayer (void *canvas, GFX_LayerAttrs *layerAttrs)
//  void gfxDeleteLayer (void *canvas)
//  void gfxSetLayerAttrs (void *canvas, GFX_I32 idxLayer, GFX_LayerAttrs *layerAttrs)
//  void gfxGetLayerAttrs (void *canvas, GFX_I32 idxLayer, GFX_LayerAttrs *layerAttrs)
//  void gfxClearLayer (void *canvas)
//  void gfxClearAllLayers (void *canvas)
//  void gfxMoveLayer (void *canvas, GFX_I32 dstIdx)
//  void gfxMergeLayerDown (void *canvas)
//
//
//  Drawing
//
//  void gfxInitDrawing (void *canvas, GFX_DrawingAttrs *drawingAttrs)
//  void gfxUpdateDrawing (void *canvas, GFX_DrawingAttrs *drawingAttrs)
//  void gfxFinalizeDrawing (void *canvas, GFX_DrawingAttrs *drawingAttrs)
//
//
//  Rendering Canvases and Layers
//
//  void gfxSetCanvasViewport (void *canvas, GFX_Rect *canvasViewport)
//  void gfxGetCanvasViewport (void *canvas, GFX_Rect *canvasViewport)
//  void gfxRenderCanvas (void *canvas)
//  GFX_Image *gfxRenderToImage (void *canvas, GFX_RenderToImageAttrs *renToImgAttrs)
//
//
//  Transforming Canvases and Layers
//
//  void gfxResetCanvasXform (void *canvas)
//  void gfxInitCanvasXform (void *canvas, GFX_XformAttrs *xFormAttrs)
//  void gfxUpdateCanvasXform (void *canvas, GFX_XformAttrs *xFormAttrs)
//  void gfxFinalizeCanvasXform (void *canvas, GFX_XformAttrs *xFormAttrs)
//  void gfxResetLayerXform (void *canvas)
//  void gfxInitLayerXform (void *canvas, GFX_XformAttrs *xFormAttrs)
//  void gfxUpdateLayerXform (void *canvas, GFX_XformAttrs *xFormAttrs)
//  void gfxFinalizeLayerXform (void *canvas, GFX_XformAttrs *xFormAttrs)
//
//
```

```
//  Selection
//
//  GFX_I32 gfxCutSelection (void *canvas, GFX_Rect *selectionRect)
//  GFX_I32 gfxCopySelection (void *canvas, GFX_Rect *selectionRect)
//  void gfxPasteSelection (void *canvas, GFX_F32 x, GFX_F32 y)
//  GFX_I32 gfxInitSelectionXform (void *canvas, GFX_Rect *srcRect);
//  void gfxUpdateSelectionXform (void *canvas, GFX_RotatedRect *dstRect)
//  void gfxFinalizeSelectionXform (void *canvas, GFX_RotatedRect *dstRect)
//
//  Undo and Redo
//
//  void gfxUndo (void *canvas)
//  void gfxRedo (void *canvas)
//  void gfxUndoAll (void *canvas)
//  void gfxRedoAll (void *canvas)
//
//
//  Saving and Loading a ICE Canvas
//
//  GFX_I8 *gfxSaveCanvas (void *canvas, GFX_I32 format)
//  GFX_I8 *gfxSaveCanvasRegion (void *canvas, GFX_Rect *saveRegion, GFX_I32 format)
//  void *gfxLoadCanvas (void *ICEInst, GFX_I8 *byteStream)
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
//  THE ICE DRAWING ENGINE API
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  To avoid multiple inclusion of header files
//-----------------------------------------------------------------------------
#ifndef _GFX_ICE_API_
#define _GFX_ICE_API_


//-----------------------------------------------------------------------------
//  Required include files for this header file
//-----------------------------------------------------------------------------
#include <stddef.h>


//-----------------------------------------------------------------------------
//  To make functions accessible from C++ code
//-----------------------------------------------------------------------------
#ifdef __cplusplus
extern "C" {
#endif


//-----------------------------------------------------------------------------
//  ICE DRAWING ENGINE API VERSION NUMBER
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  The 16 most significant bits of GFX_ICE_API_VER_NUMBER identify the major
//  revision number of the ICE Drawing Engine API; the 16 least significant bits
//  of GFX_ICE_API_VER_NUMBER identify the minor revision number of the ICE
//  Drawing Engine API.
//
//  This is version 1:3 (Major:Minor) of the ICE Drawing Engine API.
//-----------------------------------------------------------------------------
#define GFX_ICE_API_VER_NUMBER 0x00010003


//-----------------------------------------------------------------------------
//  FUNDAMENTAL DATA TYPES
//-----------------------------------------------------------------------------
```

```
//--------------------------------------------------------------------------------
//  Fundamental data types for characters, integers, and floating point numbers.
//  Applications should modify these definitions based on their target platform and
//  compiler settings.
//--------------------------------------------------------------------------------
typedef char                GFX_I8;
typedef short               GFX_I16;
typedef int                 GFX_I32;
typedef long long           GFX_I64;
typedef unsigned char       GFX_U8;
typedef unsigned short      GFX_U16;
typedef unsigned int        GFX_U32;
typedef unsigned long long  GFX_U64;
typedef float               GFX_F32;
typedef double              GFX_F64;


//--------------------------------------------------------------------------------
//  Applications should set GFX_INLINE to the keyword used by their compiler to
//  identify inline functions. The default setting is __inline, the keyword used by
//  the Microsoft Visual Studio 2008 compiler to identify inline functions.
//--------------------------------------------------------------------------------
#define GFX_INLINE __inline


//--------------------------------------------------------------------------------
//  Data representation for pen points
//--------------------------------------------------------------------------------
typedef struct {
    GFX_F32 x;      // Pen x-coordinate
    GFX_F32 y;      // Pen y-coordinate
    GFX_F32 p;      // Pen pressure value with range [0,1]
}   GFX_Point;


//--------------------------------------------------------------------------------
//  Data representation for an axis-aligned rectangle
//--------------------------------------------------------------------------------
typedef struct {
    GFX_F32 x;      // x-coordinate of bottom-left corner of the rectangle
    GFX_F32 y;      // y-coordinate of bottom-left corner of the rectangle
    GFX_F32 w;      // Rectangle width
    GFX_F32 h;      // Rectangle height
}   GFX_Rect;


//--------------------------------------------------------------------------------
//  Data representation for a rotated rectangle
//--------------------------------------------------------------------------------
typedef struct {
    GFX_F32 cx;     // x-coordinate of the center point of the rectangle
    GFX_F32 cy;     // y-coordinate of the center point of the rectangle
    GFX_F32 w;      // Rectangle width
    GFX_F32 h;      // Rectangle height
    GFX_F32 angle;  // Rotation angle, in degrees, about the center point (cx,cy)
}   GFX_RotatedRect;


//--------------------------------------------------------------------------------
//  Data representation for images. Images are input to or output from the ICE
//  Drawing Engine as a byte stream via this data structure.
//
//  The ICE Drawing Engine supports a set of compressed image formats (e.g., JPEG) and
//  a set of simple raw image formats. When a compressed image format is specified,
//  the byteStream element of a GFX_Image points to a buffer containing the compressed
//  image. When a raw image format is specified, the byteStream element points to a
```

```
//  buffer that consists of a 32-bit width, followed by a 32-bit height, followed by
//  the raw image data stored in row major order, with the first elements of the data
//  comprising the components of the bottom-left corner of the image. The raw image
//  data is represented as a sequence of 8 bit components: interleaved R,G,B,R,G,B,...
//  for RGB raw images; interleaved R,G,B,A,R,G,B,A,... for RGBA raw images; and
//  G,G,G,... for grayscale raw images.
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
#define GFX_IMAGE_RGB_JPEG  0   // RGB JPEG image format
#define GFX_IMAGE_RGB_RAW   1   // A simple RGB raw image format
#define GFX_IMAGE_RGBA_RAW  2   // A simple RGBA raw image format
#define GFX_IMAGE_GRAY_RAW  3   // A simple grayscale raw image format
#define GFX_IMAGE_RGB_PNG   4   // RGB PNG image format
#define GFX_IMAGE_RGBA_PNG  5   // RGBA PNG image format
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
typedef struct {
    GFX_I32 format;         // GFX_IMAGE_RGB_JPEG, GFX_IMAGE_RGB_RAW, ...
    GFX_I32 sizeInBytes;    // Size of the image byte stream in bytes
    GFX_I8  *byteStream;    // Pointer to the image byte stream
}   GFX_Image;


//-------------------------------------------------------------------------------
//  Data representation for general purpose textures (e.g., for defining paper
//  textures for canvases). appData is a GFX_I32 value that can be set and used by
//  the application to associate application-specific data with a GFX_Texture.
//
//  Currently, only GFX_IMAGE_RGB_JPEG image-based textures are supported. Future
//  releases of the ICE Drawing Engine will incorporate additional texture types
//  (such as procedural textures that are not image-based) and their corresponding
//  attributes in the GFX_Texture data structure.
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
#define GFX_TEXTURE_TYPE_IMAGE  0   // Image-based texture
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
typedef struct {
    GFX_I32   type;         // Currently only GFX_TEXTURE_TYPE_IMAGE is supported
    GFX_I32   appData;      // Application-specific data
    GFX_Image textureImage; // Texture image data for GFX_TEXTURE_TYPE_IMAGE type
}   GFX_Texture;


//-------------------------------------------------------------------------------
//  MEMORY ALLOCATION
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  The following typedefs define function pointers for all memory allocation tasks
//  performed by this system. The last argument to each memory allocation function
//  is an opaque pointer memAppData to application specific data. The application can
//  specify its own memory allocation function pointers when a ICE instance is
//  created; if the function pointers are not specified, the C-library malloc, free,
//  and realloc functions are used and memAppData is ignored.
//
//  GFX_MallocFP returns a pointer to space for an object of size numBytes, or NULL
//  if the request cannot be satisfied. The space is uninitialized.
//
//  GFX_FreeFP deallocates the space pointed to by object; it does nothing if object
//  is NULL. object must be a pointer to space previously allocated by GFX_MallocFP
//  or GFX_ReallocFP.
//
//  GFX_ReallocFP changes the size of object to numBytes. object must be a pointer to
//  space previously allocated by GFX_MallocFP or GFX_ReallocFP. The contents of
//  object will be unchanged up to the minimum of the old and new sizes of object.
//  If the new size of object is larger, the additional space is uninitialized.
```

```
//  GFX_ReallocFP returns a pointer to the reallocated space for object, or NULL if
//  the request cannot be satisfied, in which case object is unchanged. Note that the
//  returned pointer may be different than the input (object) pointer.
//-----------------------------------------------------------------------------
typedef void* (*GFX_MallocFP)  (size_t numBytes, void *memAppData);
typedef void  (*GFX_FreeFP)    (void *object, void *memAppData);
typedef void* (*GFX_ReallocFP) (void *object, size_t numBytes, void *memAppData);


//-----------------------------------------------------------------------------
//  ICE INSTANCES
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  A ICE instance includes the following elements
//  + The canvases of the ICE instance
//  + A selection buffer for supporting cut, copy, and paste operations
//  + ICE instance attributes including
//      + Memory allocation function pointers and memAppData
//      + A flag indicating whether the selection buffer is empty or not
//  + The most recent error code for the ICE instance. Every ICE function call
//    associated with a ICE instance sets the error code for the ICE instance
//    prior to returning.
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
//  Public Data Structures
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  The data structure for ICE instance attributes and related constants. Bit
//  flags (BFs) are used for setting and getting ICE instance attributes. For
//  example, to set the ICE instance memory function pointers, the application
//  might declare a GFX_ICEInstAttrs variable instAttrs, set instAttrs.bitFlags to
//  GFX_BF_MALLOC_FP | GFX_BF_FREE_FP | GFX_BF_REALLOC_FP, and set instAttrs.mallocFP,
//  instAttrs.freeFP, and instAttrs.reallocFP to appropriate function pointers. Next,
//  the application would call gfxSetICEInstAttrs with a pointer to instAttrs and
//  the ICE Drawing Engine would set its internal state accordingly. An analogous
//  procedure is used for getting ICE instance attributes.
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
#define GFX_BF_MALLOC_FP            0x0001
#define GFX_BF_FREE_FP              0x0002
#define GFX_BF_REALLOC_FP           0x0004
#define GFX_BF_MEM_APP_DATA         0x0008
#define GFX_BF_SELECTION_BUF_STATE  0x0010
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
#define GFX_SEL_BUFFER_EMPTY    0
#define GFX_SEL_BUFFER_FULL     1
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
typedef struct {
    //----Set/get bit flags-------------------------------------------------------
    GFX_I32 bitFlags;        // Bitwise OR of bit flags for set/get of instance attrs
    //-----------------------------------------------------------------------------
    //----Memory function pointers and application specific data-----------------
    GFX_MallocFP mallocFP;   // Memory malloc fn ptr; defaults to C-library malloc
    GFX_FreeFP freeFP;       // Memory free fn ptr; defaults to C-library free
    GFX_ReallocFP reallocFP; // Memory realloc fn ptr; defaults to C-library realloc
    void *memAppData;        // Opaque ptr of app data for mem fns; defaults to NULL
    //-----------------------------------------------------------------------------
    //----Selection buffer state--------------------------------------------------
    GFX_I32 selBufferState;  // GFX_SEL_BUFFER_EMPTY or GFX_SEL_BUFFER_FULL
    //-----------------------------------------------------------------------------
}   GFX_ICEInstAttrs;
```

```
//-----------------------------------------------------------------------------
//  Public Functions
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  Create a thread-safe ICE instance. Specific instance attributes can be
//  specified when the ICE instance is created by setting the corresponding bit
//  flags in instAttrs (using the convention described in the GFX_ICEInstAttrs
//  data structure comment block). Default values (see above) are used for any
//  attributes that are not identified by instAttrs->bitFlags. Alternatively, if
//  instAttrs is NULL, default values are used for all attributes. This function
//  returns an opaque pointer to the ICE instance upon success; a NULL pointer is
//  returned if the request cannot be satisfied.
//-----------------------------------------------------------------------------
void *gfxCreateICEInst (GFX_ICEInstAttrs *instAttrs);


//-----------------------------------------------------------------------------
//  Destroy (recursively) the specified ICE instance and all of its associated
//  data structures (e.g., its canvases, the layers comprising each canvas, etc.)
//-----------------------------------------------------------------------------
void gfxDestroyICEInst (void *ICEInst);


//-----------------------------------------------------------------------------
//  Set the requested ICE instance attributes for the specified ICE instance.
//  Note that changing certain instance attributes of a ICE instance from values
//  specified when the ICE instance was created (e.g., the memory function
//  pointers) may have undesirable consequences. Note also that the application can
//  only observe the selection buffer state; it cannot set the selection buffer
//  state.
//-----------------------------------------------------------------------------
void gfxSetICEInstAttrs (void *ICEInst, GFX_ICEInstAttrs *instAttrs);


//-----------------------------------------------------------------------------
//  Get the requested ICE instance attributes for the specified ICE instance
//-----------------------------------------------------------------------------
void gfxGetICEInstAttrs (void *ICEInst, GFX_ICEInstAttrs *instAttrs);


//-----------------------------------------------------------------------------
//  Return the most recent error code for the specified ICE instance. Every ICE
//  function call associated with a ICE instance sets the error code for the
//  ICE instance prior to returning.
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
#define GFX_ERR_NO_ERROR                    0
#define GFX_ERR_INSUFFICIENT_CPU_MEMORY     1
#define GFX_ERR_INSUFFICIENT_GPU_MEMORY     2
#define GFX_ERR_INSUFFICIENT_GL_SUPPORT     3
#define GFX_ERR_INSUFFICIENT_GPU_SUPPORT    4
#define GFX_ERR_UNSUPPORTED_FEATURE         5
#define GFX_ERR_INVALID_INPUT               6
#define GFX_ERR_INTERNAL_ERROR              7
#define GFX_ERR_FINALIZE_DRAWING_REQUIRED   8
#define GFX_ERR_FINALIZE_XFORM_REQUIRED     9
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
GFX_I32 gfxGetError (void *ICEInst);


//-----------------------------------------------------------------------------
//  ICE CANVASES
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
```

```
//  A ICE canvas has the following elements
//  + Canvas attributes including
//      + Canvas layer data such as
//          + An ordered list of layers, sorted from bottom-most (i.e., closest to
//            the paper) to top-most. Layers are identified by their index in this
//            list; layer indices run from zero (for the bottom-most layer) to N-1
//            (for the top-most layer), where N is the number of layers in the list.
//          + The index of the currently active layer. Note that a valid index for
//            the active layer of a canvas with N layers lies between zero and N-1;
//            if the canvas does not have an active layer (e.g., if the layer list is
//            empty), the index of the currently active layer is negative.
//      + Attributes of the canvas paper such as
//          + Paper type, e.g., a solid color or image-based paper texture
//          + Paper color for solid color papers or colorized paper textures
//          + Paper texture strength in the range [0,1] for modulating the strength of
//            individual features of a paper texture
//          + A Boolean determining if paper textures are colorized. If the Boolean is
//            true, paper textures are colorized with the paper color.
//          + A Boolean determining if multi-scale rendering is used to render paper
//            textures. If multi-scale rendering is used, paper textures exhibit a
//            fractal-like quality when rendered so that, when a canvas is zoomed, the
//            individual features of a paper texture are scaled while the
//            characteristic structure of the paper texture is maintained. Multi-scale
//            rendering eliminates the aliasing and pixelization artifacts typical
//            when scaling image-based paper textures. If multi-scale rendering is not
//            used, conventional sampling and interpolation methods are used to render
//            transformed paper textures.
//          + A GFX_Texture for paper textures. For image-based paper textures, the
//            texture image should be both seamless and tileable, i.e., the image can
//            be repeated indefinitely in any two dimensional plane without noticeable
//            seams at the edges where two instances of the image meet.
//      + Attributes of the current pen such as
//          + Pen type, e.g., a pencil or marker
//          + Pen color
//          + A level of graininess, in the range [0,1], for pencil-type pens, where
//            more graininess emulates a softer pencil or a rougher surface
//          + The maximum pen width, i.e., the pen width in pixels at maximum pen
//            pressure, and the pen width sensitivity in the range [0,1]. The pen
//            width sensitivity defines how the pen width varies with pen pressure.
//            If the pen width sensitivity is 0, the pen width is independent of pen
//            pressure; if the pen width sensitivity is 1, the pen width varies from 0
//            pixels at no pressure to the maximum pen width at maximum pen pressure.
//          + The maximum pen opacity, i.e., the pen opacity at maximum pen pressure,
//            and the pen opacity sensitivity, where both attributes are in the range
//            [0,1]. The pen opacity sensitivity defines how the pen opacity varies
//            with pen pressure. If the pen opacity sensitivity is 0, the pen opacity
//            is independent of pen pressure; if the pen opacity sensitivity is 1, the
//            pen opacity varies from 0 at no pressure to the maximum pen opacity at
//            maximum pen pressure.
//          + The pen's draw/erase mode. Any pen type can be used in either draw or
//            erase mode. A pen in draw mode adds color and opacity to the active
//            layer thereby building up opacity in the layer (up to a maximum opacity
//            of one), while a pen in erase mode subtracts opacity from the active
//            layer (down to a minimum opacity of zero).
//          + A curve smoothing level used when fitting curves to input pen points
//      + A global pen scale that can be used to modify the stroke widths of every
//        freehand stroke and stroked geometric shape in the canvas. This scale can be
//        used, for example, to change the visual impact of a drawing by thickening or
//        thinning its strokes.
//      + Canvas metrics such as the number of bytes of GPU memory currently in use
//        by the canvas
//  + An internal reference to the ICE instance to which this canvas belongs so
//    that subsequent calls to functions that operate on the canvas have access to
//    the ICE instance attributes. Note that a canvas can belong to only one
//    ICE instance.
//  + Rendering state
```

```
//      + The canvas viewport and canvas transform
//      + An internal flag indicating the current render status of the canvas (i.e.,
//        'up-to-date', 'needs incremental update', 'needs full render'). When the
//        ICE render function gfxRenderCanvas is called, rendering is performed
//        according to the current render status. The ICE Drawing Engine sets the
//        flag to the appropriate render status when a ICE function that modifies
//        the canvas is called (e.g., 'needs incremental update' is set when drawing
//        on the canvas and 'needs full render' is set when changing the canvas
//        viewport or the canvas transform). The ICE Drawing Engine resets the
//        flag to 'up-to-date' when gfxRenderCanvas is successfully completed. If
//        gfxRenderCanvas is called when the render status is 'up-to-date', the
//        minimal amount of work possible is done to refresh the canvas viewport
//        (e.g., the OpenGL back buffer may be refreshed from a stored copy of the
//        rendered canvas viewport).
//  + An ordered history list of reversible operations performed on the canvas
//------------------------------------------------------------------------------


//------------------------------------------------------------------------------
//  Public Data Structures
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//  The data structure for canvas metrics. numStrokes and bBox do not include drawing
//  operations that have been cleared or undone. The ICE Drawing Engine allocates
//  blocks of system and GPU memory for efficient memory management. sysMemAlloc and
//  GPUMemAlloc represent the total amount of system and GPU memory, respectively,
//  that has been allocated for the canvas, while sysMemInUse and GPUMemInUse
//  represent the amount of memory that is currently being used to represent the
//  canvas. All memory sizes are reported in bytes.
//------------------------------------------------------------------------------
typedef struct {
    GFX_I32  numStrokes;    // # freehand strokes and stroked geometric shapes
    GFX_I64  sysMemInUse;   // System memory in use by the canvas in bytes
    GFX_I64  sysMemAlloc;   // Total system memory allocated for the canvas in bytes
    GFX_I64  GPUMemInUse;   // GPU memory in use by the canvas in bytes
    GFX_I64  GPUMemAlloc;   // Total GPU memory allocated for the canvas in bytes
    GFX_Rect bBox;          // Drawn-upon area of the canvas in viewport coordinates
}   GFX_CanvasMetrics;


//------------------------------------------------------------------------------
//  The data structure for canvas attributes and related constants, including bit
//  flags (BFs) for setting and getting canvas attributes (for a description of bit
//  flag use, see the GFX_ICEInstAttrs data structure comment block). Note that
//  the canvas metrics and the undo/redo types are used for observing the current
//  state of a canvas; they cannot be set by the application.
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
#define GFX_BF_CANVAS_NUM_LAYERS                0x00000001
#define GFX_BF_CANVAS_ACTIVE_LAYER              0x00000002
#define GFX_BF_CANVAS_PAPER_TYPE                0x00000004
#define GFX_BF_CANVAS_PAPER_COLOR               0x00000008
#define GFX_BF_CANVAS_PAPER_TEXTURE_STRENGTH    0x00000010
#define GFX_BF_CANVAS_PAPER_DO_COLORIZE         0x00000020
#define GFX_BF_CANVAS_PAPER_DO_MULTI_SCALE      0x00000040
#define GFX_BF_CANVAS_PAPER_TEXTURE             0x00000080
#define GFX_BF_CANVAS_PEN_TYPE                  0x00000100
#define GFX_BF_CANVAS_PEN_COLOR                 0x00000200
#define GFX_BF_CANVAS_PEN_GRAININESS            0x00000400
#define GFX_BF_CANVAS_PEN_MAX_WIDTH             0x00000800
#define GFX_BF_CANVAS_PEN_WIDTH_SENS            0x00001000
#define GFX_BF_CANVAS_PEN_MAX_OPACITY           0x00002000
#define GFX_BF_CANVAS_PEN_OPACITY_SENS          0x00004000
#define GFX_BF_CANVAS_DRAW_ERASE_MODE           0x00008000
#define GFX_BF_CANVAS_CURVE_FIT_LEVEL           0x00010000
#define GFX_BF_CANVAS_GLOBAL_PEN_SCALE          0x00020000
```

```
#define GFX_BF_CANVAS_METRICS                   0x00040000
#define GFX_BF_CANVAS_UNDO_TYPE                 0x00080000
#define GFX_BF_CANVAS_REDO_TYPE                 0x00100000
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
#define GFX_PAPER_TYPE_SOLID_COLOR          0
#define GFX_PAPER_TYPE_RGB_IMAGE_TEXTURE    1
#define GFX_PAPER_TYPE_GRAY_IMAGE_TEXTURE   2
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
#define GFX_PEN_TYPE_MARKER 0
#define GFX_PEN_TYPE_PENCIL 1
#define GFX_PEN_TYPE_NIB    2
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
#define GFX_PEN_DRAW_MODE   0
#define GFX_PEN_ERASE_MODE  1
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
#define GFX_CFIT_DEFAULT        0
#define GFX_CFIT_SMOOTH         1
#define GFX_CFIT_SMOOTHER       2
#define GFX_CFIT_SMOOTHEST      3
#define GFX_CFIT_MOUSE_INPUT    4
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
#define GFX_REVERSIBLE_OP_NONE              0
#define GFX_REVERSIBLE_OP_STROKE            1
#define GFX_REVERSIBLE_OP_LINE              2
#define GFX_REVERSIBLE_OP_POLYLINE          3
#define GFX_REVERSIBLE_OP_POLYGON           4
#define GFX_REVERSIBLE_OP_RECTANGLE         5
#define GFX_REVERSIBLE_OP_ELLIPSE           6
#define GFX_REVERSIBLE_OP_IMAGE             7
#define GFX_REVERSIBLE_OP_ADD_LAYER         8
#define GFX_REVERSIBLE_OP_DELETE_LAYER      9
#define GFX_REVERSIBLE_OP_CLEAR_LAYER       10
#define GFX_REVERSIBLE_OP_CLEAR_ALL_LAYERS  11
#define GFX_REVERSIBLE_OP_MERGE_LAYER       12
#define GFX_REVERSIBLE_OP_XFORM_LAYER       13
#define GFX_REVERSIBLE_OP_CUT               14
#define GFX_REVERSIBLE_OP_PASTE             15
#define GFX_REVERSIBLE_OP_MOVE_LAYER        16
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
typedef struct {
    //----Set/get bit flags----------------------------------------------------
    GFX_I32 bitFlags;       // Bitwise OR of bit flags for canvas attrs set/get
    //--------------------------------------------------------------------------
    //----Canvas layers--------------------------------------------------------
    GFX_I32 numLayers;      // Number of layers in the canvas; defaults to 0
    GFX_I32 idxActiveLayer; // Index of active layer in canvas's ordered layer list
    //--------------------------------------------------------------------------
    //----Canvas paper attributes----------------------------------------------
    GFX_I32 paperType;            // Paper type: default GFX_PAPER_TYPE_SOLID_COLOR
    GFX_U8  paperColor[3];        // Paper RGB color; defaults to white (255,255,255)
    GFX_F32 paperTextureStrength; // For paper texture: range [0,1]; defaults to 1
    GFX_I32 paperDoColorize;      // For paper texture: 1 if yes, 0 if no; default 0
    GFX_I32 paperDoMultiScale;    // For paper texture: 1 if yes, 0 if no; default 1
    GFX_Texture paperTexture;     // For paper texture: default 0-byte RGB JPEG image
    //--------------------------------------------------------------------------
    //----Pen attributes-------------------------------------------------------
    GFX_I32 penType;        // Pen type; defaults to GFX_PEN_TYPE_PENCIL
    GFX_U8  penColor[3];    // Pen RGB color; defaults to black (0,0,0)
    GFX_F32 penGraininess;  // For pencil type: range [0,1]; defaults to 0.5
    GFX_F32 penMaxWidth;    // Pen width at max pen pressure; defaults to 2 pixels
```

```
    GFX_F32 penWidthSens;   // Width sensitivity to pressure: range [0,1]; default 1
    GFX_F32 penMaxOpacity;  // Opacity at max pen pressure: range [0,1]; default 0.8
    GFX_F32 penOpacitySens; // Opacity sensitivity to pressure: range [0,1]; default 1
    GFX_I32 drawEraseMode;  // Pen drawing mode; defaults to GFX_PEN_DRAW_MODE
    GFX_I32 curveFitLevel;  // For freehand strokes; defaults to GFX_CFIT_DEFAULT
    GFX_F32 globalPenScale; // Global pen width scale; defaults to 1.0
    //-------------------------------------------------------------------------
    //----Number of strokes, canvas bounding box, memory use-------------------
    GFX_CanvasMetrics metrics;  // Canvas metrics
    //-------------------------------------------------------------------------
    //----Reversible operation types for next undo/redo------------------------
    GFX_I32 undoType;       // GFX_REVERSIBLE_OP_NONE, GFX_REVERSIBLE_OP_STROKE ...
    GFX_I32 redoType;       // GFX_REVERSIBLE_OP_NONE, GFX_REVERSIBLE_OP_STROKE ...
    //-------------------------------------------------------------------------
}   GFX_CanvasAttrs;


//-----------------------------------------------------------------------------
//  Public functions
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  Create a canvas in the specified ICE instance using the specified canvas
//  attributes canvasAttrs. Specific canvas attributes can be specified when the
//  canvas is created by setting the corresponding bit flags in canvasAttrs (using
//  the convention described in the GFX_ICEInstAttrs data structure comment block).
//  Default values (see above) are used for any attributes that are not identified by
//  canvasAttrs->bitFlags. Alternatively, if canvasAttrs is NULL, default values are
//  used for all attributes. This function returns an opaque pointer to the canvas
//  upon success; a NULL pointer is returned if the request cannot be satisfied.
//-----------------------------------------------------------------------------
void *gfxCreateCanvas (void *ICEInst, GFX_CanvasAttrs *canvasAttrs);


//-----------------------------------------------------------------------------
//  Destroy (recursively) the specified canvas and all of its associated data
//  structures (e.g., its layers, the drawing operations defined on each layer, etc.)
//-----------------------------------------------------------------------------
void gfxDestroyCanvas (void *canvas);


//-----------------------------------------------------------------------------
//  Set the requested canvas attributes for the specified canvas. Note the following
//  + The application can only set the number of layers indirectly by adding,
//    deleting, and merging layers via gfxAddLayer, gfxDeleteLayer, and
//    gfxMergeLayerDown, respectively
//  + The canvas metrics and the undo/redo types are used for observing the current
//    state of a canvas; they cannot be set by the application
//-----------------------------------------------------------------------------
void gfxSetCanvasAttrs (void *canvas, GFX_CanvasAttrs *canvasAttrs);


//-----------------------------------------------------------------------------
//  Get the requested canvas attributes for the specified canvas
//-----------------------------------------------------------------------------
void gfxGetCanvasAttrs (void *canvas, GFX_CanvasAttrs *canvasAttrs);


//-----------------------------------------------------------------------------
//  ICE LAYERS
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  A canvas has an opaque paper background and an ordered list of transparent layers
//  upon which to draw. Drawing directly on the paper is not supported; drawing is
//  always performed on the active layer, which is set by the application. Layers can
//  be added and deleted from a canvas, cleared, reordered, and merged down onto the
//  layer below. The visibility of a layer can be set to on or off and the opacity of
```

```
//  a layer can be varied from zero (fully transparent) to one (fully opaque).
//
//  When a canvas is rendered, each layer is first rendered independently by
//  compositing its drawing operations in the order in which they were drawn. Next,
//  each rendered layer, modulated by its opacity, is composited (in bottom-most to
//  top-most order) onto the canvas paper using the conventional graphics over
//  operator (see Porter and Duff, 1984); this is equivalent to blending layers using
//  the 'Normal' blend mode in Adobe Photoshop.
//
//  Each layer has a set of layer attributes (e.g., the name, opacity, visibility
//  Boolean, and a texture map ID of the layer). The texture map ID is an OpenGL
//  texture ID that can be used, for example, to create a thumbnail for previewing
//  the layer. The OpenGL texture map identified by the texture map ID contains the
//  most-recently rendered rendition of the layer (rendered via gfxRenderCanvas). To
//  render a thumbnail of the layer using OpenGL, an application would enable texture
//  mapping, bind the layer's texture map, and render a quadrilateral of the desired
//  size. Note that the aspect ratio of the quadrilateral should match the aspect
//  ratio of the canvas viewport that was in effect when gfxRenderCanvas was last
//  called (see Rendering Canvases and Layers).
//------------------------------------------------------------------------------


//------------------------------------------------------------------------------
//  Public Data Structures
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//  The data structure for layer attributes and related constants, including bit
//  flags (BFs) for setting and getting layer attributes (for a description of bit
//  flag use, see the GFX_ICEInstAttrs data structure comment block)
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
#define GFX_BF_LAYER_NAME           0x0001
#define GFX_BF_LAYER_OPACITY        0x0002
#define GFX_BF_LAYER_IS_VISIBLE     0x0004
#define GFX_BF_LAYER_TEXMAP_ID      0x0008
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
typedef struct {
    //----Set/get bit flags----------------------------------------------------
    GFX_I32 bitFlags;       // Bitwise OR of bit flags for set/get of layer attrs
    //------------------------------------------------------------------------
    //----Layer attributes-----------------------------------------------------
    GFX_I8  name[256];      // Defaults to "ICE Layer"
    GFX_F32 opacity;        // Layer opacity: range [0,1]; defaults to 1
    GFX_I32 isVisible;      // Visibility: 1 if visible (default); 0 if not visible
    GFX_U32 texMapID;       // OpenGL texture ID for the layer; 0 if not valid
    //------------------------------------------------------------------------
}   GFX_LayerAttrs;


//------------------------------------------------------------------------------
//  Public functions
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//  Add a new layer to the specified canvas using the specified layer attributes
//  layerAttrs. Specific layer attributes can be specified when the layer is added
//  by setting the corresponding bit flags in layerAttrs (using the convention
//  described in the GFX_ICEInstAttrs data structure comment block). Default
//  values (see above) are used for any attributes that are not identified by
//  layerAttrs->bitFlags. Alternatively, if layerAttrs is NULL, default values are
//  used for all attributes.
//
//  The new layer is added immediately above the canvas's active layer (or at the top
//  of the canvas's ordered list of layers if there is no active layer) and the
//  canvas's active layer is set to the new layer. Like all function calls in the
//  ICE Drawing Engine, this function sets the error code in the ICE instance
```

```
//  to which this canvas belongs; if a layer cannot be added, the appropriate error
//  code will be set; the application can retrieve the error via gfxGetError.
//-----------------------------------------------------------------------------
void gfxAddLayer (void *canvas, GFX_LayerAttrs *layerAttrs);


//-----------------------------------------------------------------------------
//  Delete the canvas's active layer. The canvas's active layer is reset as follows:
//  if the deleted layer was the only layer in the canvas, the index of the canvas's
//  active layer is set to -1; if the canvas had more than one layer and the deleted
//  layer was the bottom layer, the canvas's active layer is set to the new bottom
//  layer; otherwise, the canvas's active layer is set to the layer that was
//  immediately below the deleted layer.
//-----------------------------------------------------------------------------
void gfxDeleteLayer (void *canvas);


//-----------------------------------------------------------------------------
//  Set the requested layer attributes of the layer of the specified canvas indexed
//  by idxLayer
//-----------------------------------------------------------------------------
void gfxSetLayerAttrs (void *canvas, GFX_I32 idxLayer, GFX_LayerAttrs *layerAttrs);


//-----------------------------------------------------------------------------
//  Get the requested layer attributes of the layer of the specified canvas indexed
//  by idxLayer
//-----------------------------------------------------------------------------
void gfxGetLayerAttrs (void *canvas, GFX_I32 idxLayer, GFX_LayerAttrs *layerAttrs);


//-----------------------------------------------------------------------------
//  Clear the active layer of the specified canvas
//-----------------------------------------------------------------------------
void gfxClearLayer (void *canvas);


//-----------------------------------------------------------------------------
//  Clear all layers of the specified canvas
//-----------------------------------------------------------------------------
void gfxClearAllLayers (void *canvas);


//-----------------------------------------------------------------------------
//  Move the active layer of the specified canvas from its current position in the
//  canvas's ordered list of layers to the destination position identified by the
//  index dstIdx. dstIdx must be a valid index for the layer list, i.e., it must be
//  between zero and N-1, where N is the number of layers in the layer list. Other
//  layers are shifted up or down in the layer list to accommodate the moved layer.
//-----------------------------------------------------------------------------
void gfxMoveLayer (void *canvas, GFX_I32 dstIdx);


//-----------------------------------------------------------------------------
//  Merge the active layer of the specified canvas down onto the layer immediately
//  below it. This function has no effect if the active layer is immediately above
//  the paper of the specified canvas. The layer onto which the active layer is
//  merged becomes the active layer.
//-----------------------------------------------------------------------------
void gfxMergeLayerDown (void *canvas);


//-----------------------------------------------------------------------------
//  DRAWING
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
```

```
//  Drawing on a canvas modifies the currently active layer. Drawing operations,
//  which include drawing freehand strokes, lines, polylines, polygons, rectangles,
//  ellipses, and placing images, are performed with a sequence of three ICE
//  function calls to initialize, update, and finalize the drawing operation. The
//  initialize and finalize function calls are each performed once per drawing
//  operation and the update function call may be repeated as many times as desired
//  before the finalize function is invoked. This sequence mechanism allows the ICE
//  Drawing Engine to perform very fast interactive updates to the canvas. Typically,
//  an application would render the canvas after each update function call and
//  transfer the result to the display (e.g., using a bitblt), thereby providing
//  interactive drawing for artists. Note that only drawing update and render canvas
//  function calls (gfxUpdateDrawing and gfxRenderCanvas, respectively) will be
//  processed by the ICE Drawing Engine between the initialize and finalize drawing
//  function calls.
//
//  As an example, a typical application might implement freehand stroke drawing by
//  initializing a freehand stroke operation with the point corresponding to a mouse
//  down event, updating the freehand stroke operation with points collected from
//  mouse motion events, and finalizing the freehand stroke operation with the point
//  corresponding to a mouse up event. By rendering the canvas after handling each
//  motion event, the application would provide visual feedback of the stroke to the
//  user during drawing.
//
//  Freehand strokes, lines, polylines, and polygons are drawn using pen points. A
//  pen point's (x,y) coordinates are specified in canvas viewport coordinates. The
//  pressure component of a pen point is restricted to values between zero and one;
//  the effect of pen pressure on the rendered drawing operation depends on the pen
//  type.
//
//  Rectangles, ellipses, and images are defined by a GFX_RotatedRect, which is
//  specified by its center point, width, height, and angle of rotation about its
//  center point. The center point, width, and height are specified in viewport
//  coordinates and the angle is specified in degrees.
//
//  Note that update function calls are optional, especially for drawing operations
//  such as line segments which are defined solely by their initial and final end
//  points, or for rectangles, ellipses, and images which are defined solely by their
//  final destination rectangle. For example, to draw a set of prescribed lines (e.g.,
//  guide lines or grid lines for page layout), the application could draw these lines
//  directly (i.e., non-interactively) with a sequence of initialize/finalize function
//  calls to specify the first and last endpoint of each line.
//
//  Note also that the initialize, update, and finalize function calls do not need to
//  correspond directly to a conventional mouse down / mouse drag / mouse up drawing
//  sequence. For example, to provide more control over rectangle placement, an
//  application might use the following steps: 1) initialize the size and placement
//  of a rectangle with initialize and update function calls corresponding to a mouse
//  down / mouse drag / mouse up drawing sequence; 2) provide an interface (e.g., a
//  set of handles such as those used in Adobe Photoshop) to enable the modification
//  of the rectangle's position, scale, and rotation via additional update function
//  calls; and 3) invoke the finalize function call when an 'accept' button in the
//  interface is activated by the user.
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  The ICE Drawing Engine supports infinite zoom which enables applications to
//  create drawings at 'multiple levels of detail', e.g., a drawing of a map of a
//  country that can be zoomed in to see city streets which in turn can be zoomed in
//  to see the molecular structure of the pavement of a single street. Although the
//  ICE Drawing Engine supports sub-pixel accuracy during drawing, the internal
//  representation of pen point coordinates is limited to an accuracy of 1/32nd of a
//  pixel to facilitate small file sizes. Consequently, applications should make use
//  of canvas transforms (see “Transforming Canvases and Layers”) to achieve multiple
//  level of detail drawings. For the map example given above, an application might
//  draw details at the country scale when the canvas transform is set to the
//  identity transform, details at the city street scale when the transform is scaled
//  so that the city streets fill the canvas viewport, and details at the molecular
```

```
//  scale when the transform is scaled so that the molecular structure of the pavement
//  of a single street fills the canvas viewport.
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
//  Public data structures
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  The data structure and related constants for defining a drawing operation.
//  Different attributes are used by different drawing operation types as follows:
//
//  Pen points and the number of pen points are used to draw freehand strokes, lines,
//  polylines, and polygons. They are ignored by other drawing operation types.
//
//  dstRect is used to specify the rotated bounding box of drawn rectangles, ellipses,
//  and images; for an ellipse, the width and height specify the lengths of the axes
//  of the ellipse. dstRect is ignored by other drawing operation types.
//
//  image is set when the drawing sequence for a placed image is initialized. It is
//  ignored by all other drawing operation types.
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
#define GFX_DRAW_TYPE_STROKE    0
#define GFX_DRAW_TYPE_LINE      1
#define GFX_DRAW_TYPE_POLYLINE  2
#define GFX_DRAW_TYPE_POLYGON   3
#define GFX_DRAW_TYPE_RECTANGLE 4
#define GFX_DRAW_TYPE_ELLIPSE   5
#define GFX_DRAW_TYPE_IMAGE     6
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
typedef struct {
    GFX_I32         type;       // GFX_DRAW_TYPE_STROKE, GFX_DRAW_TYPE_LINE, ...
    GFX_I32         numPoints;  // For updating strokes, lines, polylines, polygons
    GFX_Point       *pts;       // Pen points for strokes, lines, polylines, polygons
    GFX_RotatedRect dstRect;    // Destination rect for rectangles, ellipses, images
    GFX_Image       image;      // For initializing an image drawing operation
}   GFX_DrawingAttrs;


//-----------------------------------------------------------------------------
//  Public functions
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  Initialize a drawing operation of the specified drawing type in the currently
//  active layer of the specified canvas with the given drawing attributes. This
//  function initializes a drawing sequence which must be terminated by a call to
//  gfxFinalizeDrawing. The use and interpretation of the drawing attributes for each
//  drawing type are defined as follows:
//
//  + GFX_DRAW_TYPE_STROKE: the first point in a new stroke is set to pts[0]
//  + GFX_DRAW_TYPE_LINE: the start and end points of a new line segment are both set
//    to pts[0]
//  + GFX_DRAW_TYPE_POLYLINE: the first vertex of a new polyline is set to pts[0]
//  + GFX_DRAW_TYPE_POLYGON: the first vertex of a new polygon is set to pts[0]
//  + GFX_DRAW_TYPE_RECTANGLE: a new rotated rectangle is defined by dstRect
//  + GFX_DRAW_TYPE_ELLIPSE: a new ellipse is defined by dstRect. Width and height
//    define the axes of the ellipse, (cx,cy) is its center point, and the ellipse
//    is rotated about (cx,cy) by angle, which is specified in degrees.
//  + GFX_DRAW_TYPE_IMAGE: a new image specified by image is copied and placed in
//    dstRect. Currently, only GFX_IMAGE_RGB_JPEG, GFX_IMAGE_RGB_PNG, and
//    GFX_IMAGE_RGBA_PNG images are supported.
//-----------------------------------------------------------------------------
void gfxInitDrawing (void *canvas, GFX_DrawingAttrs *drawingAttrs);
```

```
//-------------------------------------------------------------------------------
//  Update the currently active drawing operation in the specified canvas with the
//  given drawing attributes. The drawing operation must have been initialized via
//  gfxInitDrawing. gfxUpdateDrawing can be called repeatedly during a drawing
//  sequence until gfxFinalizeDrawing is invoked. The interpretation of the drawing
//  attributes for each drawing type is defined as follows:
//
//  + GFX_DRAW_TYPE_STROKE: numPts pen points beginning at pts are appended to the
//    end of the current stroke
//  + GFX_DRAW_TYPE_LINE: the end point of the current line segment is reset to pts[0]
//  + GFX_DRAW_TYPE_POLYLINE: numPts vertices beginning at pts are appended to the end
//    of the current polyline
//  + GFX_DRAW_TYPE_POLYGON: numPts vertices beginning at pts are appended to the end
//    of the current polygon
//  + GFX_DRAW_TYPE_RECTANGLE: the current rectangle is redefined to dstRect
//  + GFX_DRAW_TYPE_ELLIPSE: the current ellipse is redefined by dstRect
//  + GFX_DRAW_TYPE_IMAGE: the placement of the current image is redefined to dstRect
//-------------------------------------------------------------------------------
void gfxUpdateDrawing (void *canvas, GFX_DrawingAttrs *drawingAttrs);


//-------------------------------------------------------------------------------
//  Finalize the currently active drawing operation in the specified canvas with the
//  given drawing attributes. The drawing operation must have been initialized via
//  gfxInitDrawing. This call is required to terminate a drawing sequence. The
//  interpretation of the drawing attributes for each drawing type is defined as
//  follows:
//
//  + GFX_DRAW_TYPE_STROKE: if numPoints is one, pts[0] is appended to the end of the
//    current stroke and the stroke is finalized. If numPoints is zero, the current
//    stroke is finalized at the previous point.
//  + GFX_DRAW_TYPE_LINE: if numPoints is one, the end point of the current line
//    segment is finalized at pts[0]. If numPoints is zero, the current line segment
//    is finalized at the previous point.
//  + GFX_DRAW_TYPE_POLYLINE: if numPoints is one, pts[0] is appended to the end of
//    the current polyline and the current polyline is finalized. If numPoints is
//    zero, the current polyline is finalized at the previous point.
//  + GFX_DRAW_TYPE_POLYGON: if numPoints is one, pts[0] is appended to the end of the
//    current polygon and the polygon is closed. If numPoints is zero, the polygon is
//    closed from the previous point.
//  + GFX_DRAW_TYPE_RECTANGLE: the current rectangle is finalized to dstRect
//  + GFX_DRAW_TYPE_ELLIPSE: the current ellipse is finalized by dstRect
//  + GFX_DRAW_TYPE_IMAGE: the placement of the current image is finalized to dstRect
//-------------------------------------------------------------------------------
void gfxFinalizeDrawing (void *canvas, GFX_DrawingAttrs *drawingAttrs);


//-------------------------------------------------------------------------------
//  RENDERING CANVASES AND LAYERS
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  The ICE Drawing Engine supports both on-screen canvas rendering and off-screen
//  canvas, layer, and canvas paper rendering. On-screen rendering renders a canvas
//  into its canvas viewport in the OpenGL back buffer. The application is
//  responsible for transferring the canvas viewport to the OpenGL front buffer for
//  display. The transfer can be performed using a bitblt or by swapping the OpenGL
//  back and front buffers. Off-screen rendering renders a canvas, layer, or canvas
//  paper into a GFX_Image which is then provided to the application. The application
//  is responsible for freeing the GFX_Image when it is no longer needed.
//
//  A canvas has both infinite spatial extent and infinite zoom. The specific
//  rectangular region of the infinite canvas to be rendered is determined by setting
//  a canvas viewport and a canvas transform.
//
//  For on-screen rendering, the canvas viewport defines the width and height of the
```

```
//  rendered image in integer pixel coordinates and the offset of the canvas viewport
//  from the bottom-left corner of the window in which the canvas viewport is
//  displayed. The canvas transform positions, scales, and rotates the canvas
//  relative to the canvas viewport and is described below in "Transforming Canvases
//  and Layers".
//
//  The application is responsible for making ICE rendering calls. Rendering calls
//  should be made to display changes in the canvas (e.g., during a drawing sequence,
//  after undo or redo operations, when the canvas paper is changed, and during a
//  transform sequence). A canvas keeps track of what needs to be rendered using its
//  internal render status flag (see "ICE Canvases"). ICE functions that modify
//  the canvas set the render status flag to 'needs incremental update' or 'needs
//  full render' as appropriate. Rendering a canvas via gfxRenderCanvas resets the
//  render status flag to 'up-to-date'.
//
//  For off-screen rendering, a rectangular region of a canvas, layer, or canvas
//  paper, specified in canvas viewport coordinates, is rendered to an image. The
//  size of the off-screen image (i.e., its dimensions in image pixels) is set
//  independently by specifying the width of the off-screen image in pixels (the
//  image height is computed by the ICE Drawing Engine to preserve the aspect
//  ratio of the rectangular region); this permits any rectangular region of a
//  canvas, layer, or canvas paper to be rendered off-screen at any resolution.
//------------------------------------------------------------------------------


//------------------------------------------------------------------------------
//  Public Data Structures
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//  The data structure for a progress bar mechanism. The progress bar mechanism can
//  be used by the application to monitor the progress of certain non-real-time
//  ICE functions, such as off-screen rendering of large images, which may require
//  several seconds to perform. To use the progress bar mechanism, the application
//  provides the monitored ICE function with a pointer to a GFX_ProgressBar data
//  structure that contains pointers to application functions (i.e., InitProgressBarFP
//  and FinalizeProgressBarFP) to be invoked at the beginning and end of the monitored
//  ICE function as well as an update function (i.e., UpdateProgressBarFP) to be
//  invoked as the monitored ICE function progresses. The monitored ICE function
//  will periodically call the application's update function with an estimate of what
//  percent of the task has been completed. The application's update function returns
//  an integer value to the monitored ICE function; if the value returned to the
//  monitored ICE function is non-zero, the monitored ICE function will abort
//  its operation and return control to the application.
//------------------------------------------------------------------------------
typedef struct {                                        // App function pointers (FPs)
    void    (*InitProgressBarFP) (void);                // Initialize progress bar FP
    GFX_I32 (*UpdateProgressBarFP) (GFX_F32 pctDone); // Update progress bar FP
    void    (*FinalizeProgressBarFP) (void);            // Finalize progress bar FP
}   GFX_ProgressBar;


//------------------------------------------------------------------------------
//  Attributes required for rendering a source rectangle of a canvas, layer, paper
//  background or a canvas without the paper background (i.e., with a transparent
//  background) to an image
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
#define GFX_IMAGE_SRC_TYPE_CANVAS           0   // Render canvas to image
#define GFX_IMAGE_SRC_TYPE_LAYER            1   // Render layer to image
#define GFX_IMAGE_SRC_TYPE_PAPER            2   // Render paper background to image
#define GFX_IMAGE_SRC_TYPE_CANVAS_NO_PAPER  3   // Render canvas w/o paper to image
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
typedef struct {
    GFX_I32 srcType;              // GFX_IMAGE_SRC_TYPE_CANVAS, ...
    GFX_I32 idxLayer;             // Layer index for rendering a layer to an image
```

```
    GFX_Rect srcRect;             // Source rectangle in viewport coordinates
    GFX_I32 imageWidth;           // Width in pixels of image to be rendered
    GFX_ProgressBar *progressBar; // For monitoring render progress; not used if NULL
}   GFX_RenderToImageAttrs;


//-----------------------------------------------------------------------------
//  Public functions
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  Set the canvas viewport rectangle of the specified canvas to canvasViewport.
//  canvasViewport specifies both the offset (x,y) of the bottom-left corner of the
//  canvas viewport from the bottom-left corner of the window in which the canvas
//  viewport is displayed and the size (w,h) of the canvas viewport in pixels. Note
//  that (x,y) and (w,h) must be specified as integer pixel values.
//-----------------------------------------------------------------------------
void gfxSetCanvasViewport (void *canvas, GFX_Rect *canvasViewport);


//-----------------------------------------------------------------------------
//  Get the canvas viewport rectangle of the specified canvas. canvasViewport
//  specifies both the offset (x,y) of the bottom-left corner of the canvas viewport
//  from the bottom-left corner of the window in which the canvas viewport is
//  displayed and the size (w,h) of the canvas viewport in pixels.
//-----------------------------------------------------------------------------
void gfxGetCanvasViewport (void *canvas, GFX_Rect *canvasViewport);


//-----------------------------------------------------------------------------
//  Render the paper and visible layers of the specified canvas 'on-screen' to the
//  rectangular region of the OpenGL back buffer defined by the canvas viewport of
//  the specified canvas. The region of the canvas that is rendered is defined by the
//  canvas transform.
//-----------------------------------------------------------------------------
void gfxRenderCanvas (void *canvas);


//-----------------------------------------------------------------------------
//  Render the rectangular region specified by source rectangle renToImgAttrs->srcRect
//  'off-screen' into a GFX_Image. renToImgAttrs->srcRect is specified in viewport
//  coordinates. renToImgAttrs->srcType indicates whether the visible canvas, the
//  layer indexed by renToImgAttrs->idxLayer, or the canvas paper background is
//  rendered to the image. The width in pixels (e.g., 1024 pixels) of the off-screen
//  image is specified by renToImgAttrs->imageWidth; the image height is computed by
//  the ICE Drawing Engine to preserve the aspect ratio of renToImgAttrs->srcRect.
//  Because rendering a large image may require several seconds, applications can make
//  use of a progress bar mechanism to monitor the progress of gfxRenderToImage (see
//  the description of the GFX_ProgressBar data structure for details). The progress
//  bar mechanism is not used if renToImgAttrs->progressBar is NULL.
//
//  The rendered image is a GFX_IMAGE_RAW_RGB raw image when rendering a canvas or
//  a canvas paper background to an image and a GFX_IMAGE_RAW_RGBA raw image when
//  rendering a layer or the canvas without the paper to an image. The RGBA color
//  space is required for rendering without the paper to preserve opacity values,
//  e.g., for compatibility with pixel-based applications that support layers. This
//  function returns a pointer to the GFX_Image upon success; a NULL pointer is
//  returned if the request cannot be satisfied (e.g., due to insufficient memory).
//  Note that the application is responsible for destroying the GFX_Image and its
//  byteStream when the GFX_Image is no longer needed using the memory free function
//  freeFP of the canvas's ICE instance (see gfxGetICEInstAttrs for retrieving
//  freeFP). For example, if the GFX_Image pointer returned by gfxRenderToImage is
//  stored in a variable named 'image', the calls freeFP(image->byteStream) and
//  freeFP(image) will free the GFX_Image.
//-----------------------------------------------------------------------------
GFX_Image *gfxRenderToImage (void *canvas, GFX_RenderToImageAttrs *renToImgAttrs);
```

```
//-----------------------------------------------------------------------------
//  TRANSFORMING CANVASES AND LAYERS
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  A canvas has both infinite spatial extent and infinite zoom (subject to floating
//  point representation limits for canvas transformations). The specific region of
//  the infinite canvas that is rendered to the canvas viewport is determined by the
//  canvas transform, which specifies the position, scale, and rotation of the canvas
//  relative to the canvas viewport. When the canvas transform is set to the
//  identity, the canvas origin maps to the bottom-left corner of the canvas
//  viewport, its x- and y-axes map to the bottom and left edges of the canvas
//  viewport, respectively, and one unit in the canvas corresponds to one pixel in
//  the canvas viewport.
//
//  Transformations can be applied to both canvases and the active layer of a canvas.
//  Changing the canvas transform can modify the position, scale, and rotation of the
//  canvas relative to the canvas viewport. Changing a layer transform can modify the
//  position, scale, and rotation of the layer relative to its canvas.
//
//  Modifying a canvas or active layer transform requires a sequence of ICE
//  function calls to initialize, update, and finalize the transformation. The
//  initialize and finalize function calls are each performed once per transformation
//  and the update function call may be repeated as many times as desired (typically
//  to support interactive transformations) before the finalize function is invoked.
//  Alternatively, the update function call may be omitted so that the finalize
//  function call transforms the canvas or layer directly to its final placement.
//
//  After a transformation sequence is initialized, the ICE Drawing Engine will
//  only process corresponding transformation update calls (i.e., gfxUpdateCanvasXform
//  or gfxUpdateLayerXform) and render canvas function calls (i.e., gfxRenderCanvas)
//  until the transformation sequence is finalized.
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
//  Public data structures
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
//  The data structure and related constants for defining a transformation. The
//  scale, translate, and rotation values in the GFX_XformAttrs data structure are
//  all specified as total amounts since the transform sequence was initialized via
//  gfxInitCanvasXform or gfxInitLayerXform. For type GFX_XFORM_TYPE_SCALE, scale is
//  the uniform scale applied to the canvas or layer. For type GFX_XFORM_TYPE_XYSCALE,
//  scale is the x-component of the xy-scale applied to the canvas or layer and yScale
//  is the y-component of the xy-scale applied to the canvas or layer.
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
#define GFX_XFORM_TYPE_SCALE        0
#define GFX_XFORM_TYPE_TRANSLATE    1
#define GFX_XFORM_TYPE_ROTATE       2
#define GFX_XFORM_TYPE_XYSCALE      3
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
typedef struct {
    GFX_I32 type;    // GFX_XFORM_TYPE_SCALE, GFX_XFORM_TYPE_TRANSLATE, ...
    GFX_F32 originX; // x-coord of xform origin in canvas viewport coordinates
    GFX_F32 originY; // y-coord of xform origin in canvas viewport coordinates
    GFX_F32 scale;   // Total scale (or x-component of xy-scale) about xform origin
    GFX_F32 yScale;  // Total y-component of xy-scale about xform origin
    GFX_F32 dx;      // Total x-translation in viewport coords from xform origin
    GFX_F32 dy;      // Total y-translation in viewport coords from xform origin
    GFX_F32 angle;   // Total rotation in degrees about xform origin
}   GFX_XformAttrs;
```

```
//-------------------------------------------------------------------------------
//  Public functions
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  Reset the canvas transform of the specified canvas to the identity transform
//-------------------------------------------------------------------------------
void gfxResetCanvasXform (void *canvas);


//-------------------------------------------------------------------------------
//  Initialize a transform sequence for the specified canvas. xFormAttrs must specify
//  the type of transformation (i.e., GFX_XFORM_TYPE_SCALE, GFX_XFORM_TYPE_TRANSLATE,
//  GFX_XFORM_TYPE_ROTATE, or GFX_XFORM_TYPE_XYSCALE) and the transformation origin
//  (originX,originY). Other fields of xFormAttrs are ignored by gfxInitCanvasXform.
//  This function initializes a canvas transform sequence which must be terminated by
//  a call to gfxFinalizeCanvasXform.
//-------------------------------------------------------------------------------
void gfxInitCanvasXform (void *canvas, GFX_XformAttrs *xFormAttrs);


//-------------------------------------------------------------------------------
//  Update the canvas transform of the specified canvas using the given xFormAttrs.
//  The transformation type and the transformation origin are assumed to be the same
//  as those set when the transformation sequence was initialized via
//  gfxInitCanvasXform. xFormAttrs must contain valid data in the field(s)
//  corresponding to the transformation type, i.e., a valid scale for
//  GFX_XFORM_TYPE_SCALE, a valid dx and dy for GFX_XFORM_TYPE_TRANSLATE, a valid
//  angle for GFX_XFORM_TYPE_ROTATE, and a valid x-scale and y-scale for
//  GFX_XFORM_TYPE_XYSCALE. Other fields in xFormAttrs are ignored by
//  gfxUpdateCanvasXform. gfxUpdateCanvasXform can be called repeatedly during a
//  canvas transform sequence until gfxFinalizeCanvasXform is invoked.
//-------------------------------------------------------------------------------
void gfxUpdateCanvasXform (void *canvas, GFX_XformAttrs *xFormAttrs);


//-------------------------------------------------------------------------------
//  Finalize the canvas transform of the specified canvas using the given xFormAttrs.
//  The transformation type and the transformation origin are assumed to be the same
//  as those set when the transformation sequence was initialized via
//  gfxInitCanvasXform. xFormAttrs must contain valid data in the field(s)
//  corresponding to the transformation type, i.e., a valid scale for
//  GFX_XFORM_TYPE_SCALE, a valid dx and dy for GFX_XFORM_TYPE_TRANSLATE, a valid
//  angle for GFX_XFORM_TYPE_ROTATE, and a valid x-scale and y-scale for
//  GFX_XFORM_TYPE_XYSCALE. Other fields in xFormAttrs are ignored by
//  gfxFinalizeCanvasXform. This call is required to terminate a canvas transform
//  sequence.
//-------------------------------------------------------------------------------
void gfxFinalizeCanvasXform (void *canvas, GFX_XformAttrs *xFormAttrs);


//-------------------------------------------------------------------------------
//  Reset the layer transform of the active layer of the specified canvas to the
//  identity transform
//-------------------------------------------------------------------------------
void gfxResetLayerXform (void *canvas);


//-------------------------------------------------------------------------------
//  Initialize a transform sequence for the active layer of the specified canvas. See
//  gfxInitCanvasXform for a description of xFormAttrs. This function initializes a
//  layer transform sequence which must be terminated by a call to
//  gfxFinalizeLayerXform.
//-------------------------------------------------------------------------------
void gfxInitLayerXform (void *canvas, GFX_XformAttrs *xFormAttrs);
```

```
//-------------------------------------------------------------------------------
//  Update the layer transform of the active layer of the specified canvas using the
//  given xFormAttrs. See gfxUpdateCanvasXform for a description of xFormAttrs.
//  gfxUpdateLayerXform can be called repeatedly during a layer transform sequence
//  until gfxFinalizeLayerXform is invoked.
//-------------------------------------------------------------------------------
void gfxUpdateLayerXform (void *canvas, GFX_XformAttrs *xFormAttrs);


//-------------------------------------------------------------------------------
//  Finalize the layer transform of the active layer of the specified canvas using
//  the given xFormAttrs. See gfxFinalizeCanvasXform for a description of xFormAttrs.
//  This call is required to terminate a layer transform sequence.
//-------------------------------------------------------------------------------
void gfxFinalizeLayerXform (void *canvas, GFX_XformAttrs *xFormAttrs);


//-------------------------------------------------------------------------------
//  SELECTION
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  The ICE Drawing Engine supports the selection of a rectangular region of the
//  active layer of a specified canvas to enable cut, copy, and paste operations
//-------------------------------------------------------------------------------


//-------------------------------------------------------------------------------
//  Public functions
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  Copy a rectangular region from the active layer of the specified canvas into the
//  selection buffer of the ICE instance associated with the canvas. The
//  rectangular region is specified by selectionRect, which is specified in canvas
//  viewport coordinates. If the rectangular region of the active layer is empty, the
//  contents of the selection buffer are not altered and a zero value is returned.
//  Otherwise, a non-zero value is returned.
//-------------------------------------------------------------------------------
GFX_I32 gfxCopySelection (void *canvas, GFX_Rect *selectionRect);


//-------------------------------------------------------------------------------
//  Cut a rectangular region from the active layer of the specified canvas into the
//  selection buffer of the ICE instance associated with the canvas. Note that the
//  cut operation clears the rectangular region of the active layer. The rectangular
//  region is specified by selectionRect, which is specified in canvas viewport
//  coordinates. If the rectangular region of the active layer is empty, the contents
//  of the selection buffer are not altered and a zero value is returned. Otherwise,
//  a non-zero value is returned.
//-------------------------------------------------------------------------------
GFX_I32 gfxCutSelection (void *canvas, GFX_Rect *selectionRect);


//-------------------------------------------------------------------------------
//  Paste the contents of the selection buffer of the ICE instance associated with
//  the specified canvas into the active layer of the specified canvas at the given
//  point (x,y), where (x,y) specifies the bottom-left corner of the pasted selection
//  in canvas viewport coordinates
//-------------------------------------------------------------------------------
void gfxPasteSelection (void *canvas, GFX_F32 x, GFX_F32 y);


//-------------------------------------------------------------------------------
//  Initialize a selection transform sequence. This function cuts the source rectangle
//  srcRec from the active layer and pastes it back in place in the same layer. srcRec
//  is specified in canvas viewport coordinates. This function initializes a selection
//  transform sequence which must be terminated by a call to gfxFinalizeSelectionXform.
```

```
//  If the rectangular region of the active layer is empty or the initialization fails,
//  a zero value is returned. Otherwise, a non-zero value is returned.
//-------------------------------------------------------------------------------
GFX_I32 gfxInitSelectionXform (void *canvas, GFX_Rect *srcRect);


//-------------------------------------------------------------------------------
//  Update the selection transform sequence by re-positioning the pasted rectangle
//  into the destination rectangle dstRect. gfxUpdateSelectionXform can be called
//  repeatedly during a selection transform sequence until gfxFinalizeSelectionXform
//  is invoked.
//-------------------------------------------------------------------------------
void gfxUpdateSelectionXform (void *canvas, GFX_RotatedRect *dstRect);


//-------------------------------------------------------------------------------
//  Finalize the selection transform sequence by re-positioning the pasted rectangle
//  into the destination rectangle dstRect. This call is required to terminate a
//  selection transform sequence.
//-------------------------------------------------------------------------------
void gfxFinalizeSelectionXform (void *canvas, GFX_RotatedRect *dstRect);


//-------------------------------------------------------------------------------
//  UNDO AND REDO
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  The ICE Drawing Engine supports infinite undo and redo of reversible
//  operations performed on a canvas. Reversible operations include 1) drawing
//  operations (adding freehand strokes, stroked geometric shapes, and images to the
//  canvas), 2) cutting and pasting selections, 3) adding, deleting, clearing,
//  moving, and merging layers, and 4) applying transformations to layers. The ICE
//  Drawing Engine does not directly support undo and redo for other operations such
//  as changing drawing or layer attributes, transforming canvases, or creating and
//  destroying canvases; it is the application's responsibility to provide the
//  necessary support for undo and redo for such operations if so desired.
//
//  Undo and redo for a canvas are supported by maintaining a history list of the
//  reversible operations that have been performed on the canvas. An index of the
//  'last active operation' in the history list is maintained, where the 'last active
//  operation' is defined as the current last reversible operation used to render the
//  canvas. An undo decrements the index (down to zero) and a redo increments the
//  index (up to the end of the history list). When a new reversible operation is
//  performed, the new operation is placed at the position following the last active
//  operation, the index of the last active operation is incremented, and the history
//  list is truncated after the new operation (thereby making operations undone prior
//  to the new operation no longer redo-able).
//
//  The history list of a saved canvas is truncated after the last active operation.
//  Consequently, operations undone prior to the save cannot be redone when the saved
//  canvas is reloaded.
//-------------------------------------------------------------------------------


//-------------------------------------------------------------------------------
//  Public functions
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  Undo the last active reversible operation in the history list of the specified
//  canvas. This function is ignored if there are no active reversible operations in
//  the history list (i.e., the index of the last active operation is zero).
//-------------------------------------------------------------------------------
void gfxUndo (void *canvas);


//-------------------------------------------------------------------------------
```

```
//  Redo the reversible operation following the last active reversible operation in
//  the history list of the specified canvas. This function is ignored if the last
//  active reversible operation is the last reversible operation in the history list.
//-------------------------------------------------------------------------------
void gfxRedo (void *canvas);


//-------------------------------------------------------------------------------
//  Undo all of the reversible operations performed on the specified canvas
//-------------------------------------------------------------------------------
void gfxUndoAll (void *canvas);


//-------------------------------------------------------------------------------
//  Redo all of the reversible operations performed on the specified canvas
//-------------------------------------------------------------------------------
void gfxRedoAll (void *canvas);


//-------------------------------------------------------------------------------
//  SAVING AND LOADING ICE CANVASES
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  Methods are provided so that an entire canvas or a specified rectangular region
//  of a canvas can be saved to and reloaded from a file. The ICE Drawing Engine
//  supports both lossless and lossy encoding to compress a canvas when the canvas
//  or a region of the canvas is saved.
//-------------------------------------------------------------------------------


//-------------------------------------------------------------------------------
//  Public functions
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  Save the specified canvas. format determines how the canvas is compressed
//  (e.g., GFX_ENCODE_LOSSLESS_V01 or GFX_ENCODE_LOSSY_V01). This function returns
//  a pointer to a byte stream representing the specified canvas, where the byte
//  stream consists of a block of memory beginning with a 32 bit unsigned length
//  element L followed by L bytes; a NULL pointer is returned if the request cannot
//  be satisfied. The application is responsible for freeing the byte stream when it
//  is no longer needed using the memory free function of the ICE instance
//  associated with the specified canvas (available using gfxGetICEInstAttrs).
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
#define GFX_ENCODE_LOSSLESS_V01 0
#define GFX_ENCODE_LOSSY_V01    1
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
GFX_I8 *gfxSaveCanvas (void *canvas, GFX_I32 format);


//-------------------------------------------------------------------------------
//  Save a version of the specified canvas in which the content of each layer of the
//  canvas is cropped by the specified rectangular region, with the rectangular
//  region defined in canvas viewport coordinates. The content of each layer is
//  repositioned to place the bottom left corner of the rectangular region at the
//  canvas origin. A cropped canvas behaves like a regular canvas and can be loaded
//  via gfxLoadCanvas. Note, however, that operations applied to each layer of a
//  cropped canvas before saving via gfxSaveCanvasRegion can only be undone or
//  redone as a group; they cannot be individually undone or redone. format
//  determines how the cropped canvas is compressed (e.g., GFX_ENCODE_LOSSLESS_V01
//  or GFX_ENCODE_LOSSY_V01).  This function returns a pointer to a byte stream
//  representing the cropped canvas, where the byte stream consists of a block of
//  memory beginning with a 32 bit unsigned length element L followed by L bytes; a
//  NULL pointer is returned if the request cannot be satisfied. The application is
//  responsible for freeing the byte stream when it is no longer needed using the
```

```
//  memory free function of the ICE instance associated with the specified canvas
//  (available using gfxGetICEInstAttrs).
//-------------------------------------------------------------------------------
GFX_I8 *gfxSaveCanvasRegion (void *canvas, GFX_Rect *saveRegion, GFX_I32 format);


//-------------------------------------------------------------------------------
//  Load a canvas into the specified ICE instance. byteStream is a pointer to a
//  block of memory beginning with a 32 bit unsigned length element L followed by L
//  bytes representing a compressed canvas that was previously saved via
//  gfxSaveCanvas. Upon a successful load, a canvas is created and a pointer to the
//  created canvas is returned; a NULL pointer is returned if the request cannot be
//  satisfied.
//-------------------------------------------------------------------------------
void *gfxLoadCanvas (void *ICEInst, GFX_I8 *byteStream);


//-------------------------------------------------------------------------------
//  End of C++ wrapper
//-------------------------------------------------------------------------------
#ifdef __cplusplus
}
#endif


//-------------------------------------------------------------------------------
//  End of _GFX_ICE_API_
//-------------------------------------------------------------------------------
#endif
```