```
//--------------------------------------------------------------------------------
//  Filename: Nitro.h
//--------------------------------------------------------------------------------
//--------------------------------------------------------------------------------
//  Nitro Font Rendering Application Programming Interface (API)
//  Version 1.0
//  Copyright 2016, MERL, 201 Broadway, Cambridge, MA, 02139
//  All rights reserved
//  Ronald Perry
//--------------------------------------------------------------------------------


//--------------------------------------------------------------------------------
//  To handle multiple inclusion of header files
//--------------------------------------------------------------------------------
#ifndef _NITRO_
#define _NITRO_


//--------------------------------------------------------------------------------
//  Required include files for this header file. All math in Nitro is performed using
//  fixed point arithmetic. As described in FixedMath.h, an NTO_I1616 is a 32-bit
//  two's complement signed fixed point data type with 1 sign bit, 15 integer bits,
//  and 16 fractional bits. The MSB (i.e., bit 31) is the sign bit, bits 30:16 are
//  the integer bits, and bits 15:0 are the fractional bits. An NTO_I1616 value R
//  represents the mathematical value (R / 65536.0). Note that other basic data types
//  such as an NTO_I32 are also defined in FixedMath.h.
//--------------------------------------------------------------------------------
#include "FixedMath.h"


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


//--------------------------------------------------------------------------------
//  Create and return a thread-safe instance of a renderer. maxPoints defines the
//  maximum number of points allowed to represent a tessellated glyph during
//  rendering. As a guideline for applications that want to minimize the use of RAM
//  for rendering, 1500 points is sufficient to render common Latin and CJK glyphs
//  at 500 PPEM or less and requires approximately 8.5 KB whereas 3200 points will
//  permit the rendering of more atypical exotic glyphs of high complexity and
//  requires approximately 15 KB (use ntoGetRendererSize() to determine the size of
//  a renderer instance). A NULL is returned if this function fails.
//--------------------------------------------------------------------------------
void *ntoCreateRenderer (NTO_I32 maxPoints);


//--------------------------------------------------------------------------------
//  Return the size in bytes of the specified renderer
//--------------------------------------------------------------------------------
NTO_I32 ntoGetRendererSize (void *renderer);


//--------------------------------------------------------------------------------
//  Destroy the specified renderer
//--------------------------------------------------------------------------------
void ntoDestroyRenderer (void *renderer);


//--------------------------------------------------------------------------------
//  Create and return a CSM table with the specified cutoff values. A NULL is
//  returned if this function fails.
```

```
//
//  Nitro uses a new distance-based antialiasing algorithm to render glyphs. Similar
//  to Saffron, Nitro exploits the inherent properties of distance fields to provide
//  'Continuous Stroke Modulation' (CSM), i.e., continuous modulation of both stroke
//  weight and edge sharpness. CSM uses two rendering parameters, outsideCutoff and
//  insideCutoff, to control the mapping of distances to glyph density (i.e., 8-bit
//  grayscale) values.
//
//  The internal Nitro function that maps distances to density values has an outside
//  cutoff value, below which densities are set to zero, and an inside cutoff value,
//  above which densities are set to a maximum density value (i.e., 255). Between
//  these two cutoff values, the mapping is linear ranging from zero at the outside
//  cutoff value to 255 at the inside cutoff value.
//
//  Adjusting the outside and inside cutoff values affects stroke weight and edge
//  sharpness. The spacing between these two parameters is comparable to twice the
//  filter radius of classic antialiasing methods; a narrow spacing provides a
//  sharper edge while a wider spacing provides a softer, more filtered edge. When
//  the spacing is zero, the resulting density image is a bi-level bitmap. When the
//  spacing is very wide, the resulting density image has a watercolor-like edge.
//  Typically, users prefer sharp, high contrast edges at small point sizes and
//  softer edges for larger point sizes and for animated text.
//
//  Because a glyph's edge lies at the zero-valued iso-surface of its corresponding
//  distance field, the outside cutoff value typically has a negative value, the
//  inside cutoff value typically has a positive value, and their midpoint typically
//  lies near zero. Adjusting these parameters to shift the midpoint towards negative
//  infinity will increase the stroke weight; shifting the midpoint towards positive
//  infinity will decrease the stroke weight. Note that the outside cutoff value must
//  always be less than or equal to the inside cutoff value.
//
//  Implementation Notes:
//  + The outside and inside cutoff values are constrained to lie approximately in
//    the [-1.5, 1.5] interval.
//  + Using symmetric CSM values, where abs(outsideCutoff) is equal to insideCutoff,
//    will typically result in faster rendering.
//-------------------------------------------------------------------------------
void *ntoCreateCSMTable (NTO_I1616 outsideCutoff, NTO_I1616 insideCutoff);


//-------------------------------------------------------------------------------
//  Return the size in bytes of the specified CSM table
//-------------------------------------------------------------------------------
NTO_I32 ntoGetCSMTableSize (void *csmTable);


//-------------------------------------------------------------------------------
//  Destroy the specified CSM table
//-------------------------------------------------------------------------------
void ntoDestroyCSMTable (void *csmTable);


//-------------------------------------------------------------------------------
//  NTO GLYPH PATH DATA STRUCTURE
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  Glyphs are represented by an NTOPath data structure. An NTOPath is composed of a
//  series of pen commands (e.g., moveto, lineto, quadto) that define a closed path.
//  The endpoints and the control vertices of each segment of a path are specified in
//  fixed point (NTO_I1616) font units. An NTOPath consists of the following elements:
//
//  (+) minX, maxX, minY, maxY
//      (-) The exact bounding box (bBox) of the glyph
//      (-) Specified in fixed point (NTO_I1616) font units
//      (-) May have fractional values
//
//  (+) numPenCmds
```

```
//      (-) The number of pen commands contained in the path
//      (-) Specified as an NTO_I32
//
//  (+) penCmds
//      (-) Ptr to a contiguous block of fixed-sized pen commands defining the path
//      (-) All coordinates are specified in fixed point (NTO_I1616) font units
//      (-) The first command of each contour is a moveto command
//      (-) The x and y coordinates of the endpoint of the last command of each
//          contour must equal the x and y coordinates of the contour's initial
//          moveto command
//
//  Applications that prefer to scale glyphs prior to rendering via ntoRenderGlyph()
//  can simply specify all coordinates in pixel space and set the rendering scale to
//  one. This is a typical scenario when applications are scaling and hinting prior
//  to rasterization.
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//  The NTOPenCmd data structure defines a single pen command. A summary of the pen
//  commands follows:
//
//  (+) moveto
//      (-) Has the form <NTO_PEN_MOVETO_CMD x y>
//      (-) Starts a new subpath (i.e., contour) of the NTOPath
//      (-) Sets the current point to (x, y)
//
//  (+) lineto
//      (-) Has the form <NTO_PEN_LINETO_CMD x y>
//      (-) Appends a straight line segment to the current subpath (i.e., contour)
//          of the NTOPath
//      (-) The line segment extends from the current point to (x, y)
//      (-) After constructing the line segment, sets the current point to (x, y)
//
//  (+) quadto
//      (-) Has the form <NTO_PEN_QUADTO_CMD x y cx cy>
//      (-) Appends a quadratic Bezier curve segment to the current subpath (i.e.,
//          contour) of the NTOPath
//      (-) The quadratic Bezier curve segment extends from the current point to
//          (x, y) using (cx, cy) as the Bezier control point
//      (-) After constructing the curve segment, sets the current point to (x, y)
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
#define NTO_PEN_MOVETO_CMD    1     // Sets the current point of the subpath
#define NTO_PEN_LINETO_CMD    2     // Appends a line segment
#define NTO_PEN_QUADTO_CMD    3     // Appends a quadratic Bezier curve segment
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
typedef struct  {
    NTO_I32     opCode;             // NTO_PEN_MOVETO_CMD, NTO_PEN_LINETO_CMD, etc.
    NTO_I1616   x, y;               // Data for this command (see above)
    NTO_I1616   cx, cy;             // Data for this command (see above)
}   NTOPenCmd;
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
typedef struct  {
    NTO_I1616   minX;               // The minimum x-coordinate of the glyph's bBox
    NTO_I1616   maxX;               // The maximum x-coordinate of the glyph's bBox
    NTO_I1616   minY;               // The minimum y-coordinate of the glyph's bBox
    NTO_I1616   maxY;               // The maximum y-coordinate of the glyph's bBox
    NTO_I32     numPenCmds;         // Number of pen commands defining the path
    NTOPenCmd   *penCmds;           // Array of pen commands defining the path
}   NTOPath;


//-------------------------------------------------------------------------------
//  Rendering attributes for ntoRenderGlyph. If lcdMode is 0 (standard pixel-based
//  rendering is desired), buffer will contain the rendered result upon return
//  (buffer must be pre-allocated by the caller). If lcdMode is 1 (subpixel / LCD
```

```
//  rendering is desired), the three buffers bufferR, bufferG, and bufferB (one for
//  each color channel) will contain the rendered result upon return (all 3 buffers
//  must be pre-allocated by the caller). We refer to both buffer and bufferR,
//  bufferG, and bufferB as the "output rendering buffer". The width w and height h
//  of the output rendering buffer can be computed via ntoComputeBufferSize().
//  ntoComputeBufferSize() forces 4-byte alignment of scanlines in the output
//  rendering buffer and provides sufficient room for filtering the edges of the
//  glyph. The data in the output rendering buffer is stored in row major order, with
//  the first elements of the data comprising the components of the bottom-left corner
//  of the image. The BitBLT size (in pixels) of the output rendering buffer is w x h.
//  When subpixel / LCD rendering is selected, color fringing can occur. To reduce
//  these artifacts at the cost of some blurriness, set reduceColor to 1; to leave the
//  RGB values unaltered, set reduceColor to 0.
//
//  Note that the NTOPath is scaled to the requested size and translated to the
//  bottom-left corner of the image. The (x,y) translation applied, which is required
//  for BitBLTing and typesetting glyphs, can be computed via ntoComputePenOffset().
//-------------------------------------------------------------------------------
typedef struct {
    void      *renderer;    // Renderer instance
    NTOPath   *ntoPath;     // NTO path to render (allocated & set by the caller)
    NTO_I1616 scale;        // NTO_I1616 version of (PPEM / Units per EM)
    NTO_I32   w;            // BitBLT width  (computed via ntoComputeBufferSize())
    NTO_I32   h;            // BitBLT height (computed via ntoComputeBufferSize())
    NTO_I32   xOffset;      // x offset to add to pen x (see ntoComputePenOffset())
    NTO_I32   yOffset;      // y offset to add to pen y (see ntoComputePenOffset())
    NTO_I32   quality;      // 0: Normal, 1: Higher, 2: Highest
    NTO_I32   lcdMode;      // 0: Normal pixel rendering, 1: Subpixel / LCD rendering
    NTO_I32   reduceColor;  // 0: No color reduction, 1: Reduce color fringing
    void      *csmTable;    // CSM table to use for rendering
    NTO_U8    *buffer;      // Grayscale rendering buffer allocated by caller
    NTO_U8    *bufferR;     // Red channel rendering buffer allocated by caller
    NTO_U8    *bufferG;     // Green channel rendering buffer allocated by caller
    NTO_U8    *bufferB;     // Blue channel rendering buffer allocated by caller
}   NTORenderAttrs;


//-------------------------------------------------------------------------------
//  Render the glyph described by attrs with the specified renderer attrs->renderer.
//  A zero is returned upon success; a non-zero is returned upon failure. A typical
//  rendering session proceeds as follows:
//
//  void *renderer = ntoCreateRenderer(maxPoints)
//  void *csmTable = ntoCreateCSMTable(outsideCutoff, insideCutoff)
//  Allocate a target buffer to hold the result, sized to handle the largest PPEM
//  Set constant NTORenderAttrs: renderer, scale, quality, lcdMode, csmTable, buffer
//  Set initial penX, penY
//  For each glyph to render {
//      Get the NTOPath for the current glyph and set ntoPath in NTORenderAttrs
//      Compute NTORenderAttrs w and h via ntoComputeBufferSize()
//      Compute NTORenderAttrs xOffset and yOffset via ntoComputePenOffset()
//      Render the glyph into the target buffer via ntoRenderGlyph()
//      BitBLT the target buffer to the display with the bottom-left corner positioned
//      at (penX + xOffset, penY + yOffset) and a width and height in pixels of w x h
//      Advance penX and penY
//  }
//  Free target buffer
//  ntoDestroyCSMTable(csmTable)
//  ntoDestroyRenderer(renderer)
//-------------------------------------------------------------------------------
NTO_I32 ntoRenderGlyph (NTORenderAttrs *attrs);


//-------------------------------------------------------------------------------
//  Determine the width attrs->w and height attrs->h of the output rendering buffer
//  based on the specified rendering attributes
//-------------------------------------------------------------------------------
```

```
void ntoComputeBufferSize (NTORenderAttrs *attrs);


//-----------------------------------------------------------------------------
//  Determine the x and y offsets (attrs->xOffset and attrs->yOffset) used for
//  BitBLTing and typesetting glyphs based on the specified rendering attributes.
//  These offsets are added to the typesetting pen position to determine the BitBLT
//  position for the output rendering buffer.
//-----------------------------------------------------------------------------
void ntoComputePenOffset (NTORenderAttrs *attrs);


//-----------------------------------------------------------------------------
//  Return the number of points required to represent a tessellated form of the glyph
//  described by attrs. attrs->renderer, attrs->ntoPath, and attrs->scale must be set
//  prior to invocation.
//-----------------------------------------------------------------------------
NTO_I32 ntoGetGlyphNumPoints (NTORenderAttrs *attrs);


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


//-----------------------------------------------------------------------------
//  End of _NITRO_
//-----------------------------------------------------------------------------
#endif
```