TGAPaint

P. Baillehache

November 8, 2017

Contents

Introduction

TGAPaint library is a C library to create and manipulate pictures in TGA format.

It offers functions to create, open and save TGA files, restricted to types 2 (uncompressed true-color image) and 10 (run-length encoded true-color image), pixel depths of 16, 24, and 32, and color map 0 (no color map) and 1 (standard TGA color map).The user can access the header and pixels values, paint simple geometric shapes (point, line, curve, rectangle, filled rectangle, ellipse and filled ellipse) or Shapoid and print text (ascii characters) with a virtual pencil (round/square shape, solid/blend color, antialias), and apply gaussian blur to the picture.

1 Interface

```
// *************** TGAPAINT.H ***************
```

```
#ifndef TGAPAINT_H
#define TGAPAINT_H

// ================= Include =================

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "bcurve.h"

// ================= Define ==================

// Maximum number of colors in a TGAPencil
#define TGA_NBCOLORPENCIL 10
// Maximum number of curves in the definition of a font's character
#define TGA_NBMAXCURVECHAR 10

// ================= Generic functions ==================

void TGATypeUnsupported(void*t, ...);
#define TGADrawCurve(T,C,P) _Generic((C), \
  BCurve*: TGADrawBCurve, \
  SCurve*: TGADrawSCurve, \
  default: TGATypeUnsupported)(T,C,P)

// ================= Data structure ===================

// Header of a TGA file
typedef struct TGAHeader {
  // Origin of the color map
  short int _colorMapOrigin;
  // Length of the color map
  short int _colorMapLength;
  // X coordinate of the origin
  short int _xOrigin;
  // Y coordinate of the origin
  short int _yOrigin;
  // Width of the TGA
  short _width;
  // Height of the TGA
  short _height;
  // Length of a string located located after the header
  char _idLength;
  // Type of the color map
  char _colorMapType;
  // Type of the image
  char _dataTypeCode;
  // Depth of the color map
  char _colorMapDepth;
  // Number of bit per pixel
  char _bitsPerPixel;
  // Image descriptor
  char _imageDescriptor;
} TGAHeader;

// One pixel of the TGA
typedef struct TGAPixel {
  // RGB and transparency values
  unsigned char _rgba[4];
  // Flag to memorize if this pixel is in read only mode
  bool _readOnly;
} TGAPixel;

// One layer of pixels in the TGA
typedef struct TGALayer {
  // Dimension of the layer
  VecShort *_dim;
  // Pixels (stored by rows)
  TGAPixel *_pixels;
} TGALayer;

// Main TGA structure
typedef struct TGA {
  // Header
  TGAHeader *_header;
  // Set of layers (first one is the deepest)
  GSet *_layers;
  // Current layer
  TGALayer *_curLayer;
  // Current layer index
  int _curLayerIndex;
  // Temporary working layer
  TGALayer *_tmpLayer;
} TGA;

// Enumeration of TGAPencil's color modes
typedef enum tgaPencilModeColor {
  // Constant color
  tgaPenSolid,
  // Blend between two colors
  tgaPenBlend
} tgaPencilModeColor;

// Enumeration of TGAPencil's shapes
typedef enum tgaPencilShape {
  // Shapoid
  tgaPenShapoid,
  // Pixel mode
  tgaPenPixel
} tgaPencilShape;

// Pencil to draw on a TGA
typedef struct TGAPencil {
  // List of available colors in this pencil
  TGAPixel _colors[TGA_NBCOLORPENCIL];
  // Currently active color (index in _colors)
  int _activeColor;
  // Current color mode
  tgaPencilModeColor _modeColor;
  // Current shape
  tgaPencilShape _shape;
  // Shapoid of the tip of the pen
  Shapoid *_tip;
  // The 2 colors used when color mode is tgaPenBlend (index in _colors)
  int _blendColor[2];
  // Parameter cotnroling the blend when color mode is tgaPenBlend
  // (0.0 -> _blendColor[0], 1.0 -> _blendColor[1])
  float _blend;
  // Thickness of the TGAPencil, in pixel
  float _thickness;
  // Apply antialiasing if true
  bool _antialias;
} TGAPencil;

// One character in a TGAFont
typedef struct TGAChar {
  // SCurve defining this character
  SCurve *_curve;
} TGAChar;

// Enumeration of available fonts
typedef enum tgaFont {
  // Default font
  tgaFontDefault
} tgaFont;

// Enumeration of available anchor position for fonts
typedef enum tgaFontAnchor {
  tgaFontAnchorTopLeft, tgaFontAnchorTopCenter, tgaFontAnchorTopRight,
  tgaFontAnchorCenterLeft, tgaFontAnchorCenterCenter,
  tgaFontAnchorCenterRight, tgaFontAnchorBottomLeft,
  tgaFontAnchorBottomCenter, tgaFontAnchorBottomRight
} tgaFontAnchor;

// Font to write on the TGA
typedef struct TGAFont {
  // Size in pixel of one character
  float _size;
  // Definition of the characters
  TGAChar _char[256];
  // Space between character, (x,y), in pixel
  // _space[0] is added to x after each character in a string
  // _space[1] is added to y when '\n' is printed
  VecFloat *_space;
  // Scale of the characters, (x,y), multiplied to _size
  VecFloat *_scale;
  // Tabulation size, in pixel, when '\t' is printed move x to
  // (floor(p/_tabSize)+1)*_tabSize, where p is current x position
  float _tabSize;
  // Anchor (position in the printed text corresponding to 'pos'
  // in TGAPrintString)
  tgaFontAnchor _anchor;
  // Direction to the right of the font
  VecFloat *_right;
} TGAFont;

// ================ Functions declaration ====================

// Create a TGA of width dim[0] and height dim[1] and background
// color equal to pixel
// If 'pixel' is NULL rgba(0,0,0,0) is used
// (0,0) is the bottom left corner, x toward right, y toward top
// Return NULL in case of invalid arguments or memory allocation
// failure
TGA* TGACreate(VecShort *dim, TGAPixel *pixel);

// Clone a TGA
// Return NULL in case of failure
TGA* TGAClone(TGA *tga);

// Free the memory used by the TGA
void TGAFree(TGA **tga);
```

```
// Load a TGA from the file pointed to by 'fileName'
// If 'tga' already contains a TGA, it is overwritten
// return 0 upon success, else
// 1 : couldn't open the file
// 2 : malloc failed
// 3 : can only handle image type 2 and 10
// 4 : can only handle pixel depths of 16, 24, and 32
// 5 : can only handle colour map types of 0 and 1
// 6 : unexpected end of file
// 7 : invalid arguments
int TGALoad(TGA **tga, char *fileName);

// Save the TGA 'tga' to the file pointed to by 'fileName'
// return 0 upon success, else
// 1 : couldn't open the file
// 2 : invalid arguments
int TGASave(TGA *tga, char *fileName);

// Print the header of 'tga' on 'stream'
// If arguments are invalid, do nothing
void TGAPrintHeader(TGA *tga, FILE *stream);

// Return true if 'pos' is inside 'tga'
// Return false else, or if arguments are invalid
bool TGAIsPosInside(TGA *tga, VecShort *pos);

// Get a pointer to the pixel at coord (x,y) = (pos[0],pos[1])
// in the current layer
// Return NULL in case of invalid arguments
TGAPixel* TGAGetPix(TGA *tga, VecShort *pos);

// Set the color of one pixel at coord (x,y) = (pos[0],pos[1]) to 'pix'
// in the current layer
// Do nothing in case of invalid arguments
void TGASetPix(TGA *tga, VecShort *pos, TGAPixel *pix);

// Draw one stroke at 'pos' with 'pen'
// in current layer
// Do nothing in case of invalid arguments
void TGAStrokePix(TGA *tga, VecFloat *pos, TGAPencil *pen);

// Draw a line between 'from' and 'to' with pencil 'pen'
// pixels outside the TGA are ignored
// do nothing if arguments are invalid
void TGADrawLine(TGA *tga, VecFloat *from, VecFloat *to, TGAPencil *pen);

// Draw the BCurve 'curve' (must be of dimension 2 and order > 0)
// do nothing if arguments are invalid
void TGADrawBCurve(TGA *tga, BCurve *curve, TGAPencil *pen);

// Draw the SCurve 'curve' (must be of dimension 2)
// do nothing if arguments are invalid
void TGADrawSCurve(TGA *tga, SCurve *curve, TGAPencil *pen);

// Draw a rectangle between 'from' and 'to' with pencil 'pen'
// pixels outside the TGA are ignored
// do nothing if arguments are invalid
void TGADrawRect(TGA *tga, VecFloat *from, VecFloat *to, TGAPencil *pen);

// Fill a rectangle between 'from' and 'to' with pencil 'pen'
// pixels outside the TGA are ignored
// do nothing if arguments are invalid
```

```
void TGAFillRect(TGA *tga, VecFloat *from, VecFloat *to, TGAPencil *pen);

// Draw a ellipse at 'center' of radius 'r' (Rx,Ry)
// with pencil 'pen'
// pixels outside the TGA are ignored
// do nothing if arguments are invalid
void TGADrawEllipse(TGA *tga, VecFloat *center, VecFloat *r, TGAPencil *pen);

// Fill an ellipse at 'center' of radius 'r' (Rx, Ry) with pencil 'pen'
// pixels outside the TGA are ignored
// do nothing if arguments are invalid
void TGAFillEllipse(TGA *tga, VecFloat *center, VecFloat *r, TGAPencil *pen);

// Draw the shapoid 's' with pencil 'pen'
// The shapoid must be of dimension 2
// Pixels outside the TGA are ignored
// Do nothing if arguments are invalid
void TGADrawShapoid(TGA *tga, Shapoid *s, TGAPencil *pen);

// Fill the shapoid 's' with pencil 'pen'
// The shapoid must be of dimension 2
// Pixels outside the TGA are ignored
// Do nothing if arguments are invalid
void TGAFillShapoid(TGA *tga, Shapoid *s, TGAPencil *pen);

// Apply a gaussian blur of 'strength' and 'range' perimeter on the TGA
// Do nothing if arguments are invalid
void TGAFilterGaussBlur(TGA *tga, float strength, float range);

// Print the string 's' with its anchor position at 'pos', TGAPencil
// 'pen' and font 'font'
void TGAPrintString(TGA *tga, TGAPencil *pen, TGAFont *font,
  unsigned char *s, VecFloat *pos);

// Print the char 'c' with its (bottom, left) position at 'pos'
// and (width, height) dimension 'dim' with font 'font'
void TGAPrintChar(TGA *tga, TGAPencil *pen, TGAFont *font,
  unsigned char c, VecFloat *pos);

// Get a white TGAPixel
TGAPixel* TGAGetWhitePixel(void);

// Get a black TGAPixel
TGAPixel* TGAGetBlackPixel(void);

// Get a transparent TGAPixel
TGAPixel* TGAGetTransparentPixel(void);

// Free the memory used by tgapixel
void TGAPixelFree(TGAPixel **pixel);

// Return a new TGAPixel which is a blend of 'pixA' and 'pixB'
// newPix = (1 - blend) * pixA + blend * pixB
// Return NULL if arguments are invalid
TGAPixel* TGAPixelBlend(TGAPixel *pixA, TGAPixel *pixB, float blend);

// Return a new TGAPixel which is the addition of 'ratio'
// (in [0.0,1.0]) * 'pixB' to 'pixA'
// Return NULL if arguments are invalid
TGAPixel* TGAPixelMix(TGAPixel *pixA, TGAPixel *pixB, float ratio);

// Create a default TGAPencil with all color set to transparent
// solid mode, thickness = 1.0, tip as facoid, no antialias
// Return NULL if it couldn't allocate memory
TGAPencil* TGAGetPencil(void);

// Free the memory used by the TGAPencil 'pen'
void TGAPencilFree(TGAPencil **pen);

// Clone the TGAPencil 'pen'
// Return NULL if it couldn't clone
TGAPencil* TGAPencilClone(TGAPencil *pen);

// Create a TGAPencil with 1st color active and set to black
// Return NULL if it couldn't create
TGAPencil* TGAGetBlackPencil(void);

// Select the active color of TGAPencil 'pen' to 'iCol'
// Do nothing if arguments are invalid
void TGAPencilSelectColor(TGAPencil *pen, int iCol);

// Get the index of active color of TGAPencil 'pen'
// Return -1 if arguments are invalid
int TGAPencilGetColor(TGAPencil *pen);

// Get a TGAPixel equal to the active color of the TGAPencil 'pen'
// Return NULL if arguments are invalid
TGAPixel* TGAPencilGetPixel(TGAPencil *pen);

// Get the

// Set the active color of TGAPencil 'pen' to TGAPixel 'col'
// Do nothing if arguments are invalid
void TGAPencilSetColor(TGAPencil *pen, TGAPixel *col);

// Set the active color of TGAPencil 'pen' to 'rgba'
// Do nothing if arguments are invalid
void TGAPencilSetColRGBA(TGAPencil *pen, unsigned char *rgba);

// Set the thickness of TGAPencil 'pen' to 'v'
// Equivalent to a scale of the shapoid of the tip
// Do nothing if arguments are invalid
void TGAPencilSetThickness(TGAPencil *pen, float v);

// Set the antialias of the TGAPencil 'pen' to 'v'
// Do nothing if arguments are invalid
void TGAPencilSetAntialias(TGAPencil *pen, bool v);

// Set the blend value 'v' of the TGAPencil 'pen'
// Do nothing if arguments are invalid
void TGAPencilSetBlend(TGAPencil *pen, float v);

// Set the shape of the TGAPencil 'pen' to 'tgaPenShapoid' and
// set the tip of the pen to a new facoid centered on the origin
// and scaled to the pen thickness
// Do nothing if arguments are invalid
void TGAPencilSetShapeSquare(TGAPencil *pen);

// Set the shape of the TGAPencil 'pen' to 'tgaPenShapoid' and
// set the tip of the pen to a new ellipsoid scaled to the pen thickness
// Do nothing if arguments are invalid
void TGAPencilSetShapeRound(TGAPencil *pen);

// Set the shape of the TGAPencil 'pen' to 'tgaPenShapoid' and
// set the tip of the pen to a clone of the Shapoid 'shape'
// 'shape' is considered to be centered and given at a thickness
// of 1.0 before rescaling to 'pen' thickness
// Do nothing if arguments are invalid
void TGAPencilSetShapeShapoid(TGAPencil *pen, Shapoid *shape);

// Set the shape of the TGAPencil 'pen' to 'tgaPenPixel'
// Do nothing if arguments are invalid
void TGAPencilSetShapePixel(TGAPencil *pen);

// Set the mode of the TGAPencil 'pen' to 'tgaPenSolid'
// Do nothing if arguments are invalid
void TGAPencilSetModeColorSolid(TGAPencil *pen);

// Set the mode of the TGAPencil 'pen' to 'tgaPenBlend'
// Blend is done from 'fromCol' to 'toCol'
// Do nothing if arguments are invalid
void TGAPencilSetModeColorBlend(TGAPencil *pen, int fromCol, int toCol);

// Create a TGAFont with set of character 'font',
// _fontSize = 18.0, _space[0] = _space[1] = 3.0,
// _scale[0] = 0.5, _scale[1] = 1.0, _anchor = tgaFrontAnchorTopLeft
// _dir = <1.0, 0.0>, _tabSize = _fontSize
// Return NULL if it couldn't create
TGAFont* TGAFontCreate(tgaFont font);

// Free memory used by TGAFont
// Do nothing if arguments are invalid
void TGAFreeFont(TGAFont **font);

// Set the font size of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetSize(TGAFont *font, float v);

// Set the font tab size of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetTabSize(TGAFont *font, float v);

// Set the font scale of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetScale(TGAFont *font, VecFloat *v);

// Set the font spacing of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetSpace(TGAFont *font, VecFloat *v);

// Set the anchor of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetAnchor(TGAFont *font, tgaFontAnchor v);

// Set the right direction of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetRight(TGAFont *font, VecFloat *v);

// Get the bounding box as a facoid of order 2 and dim 2 in pixels
// of the block of text representing string 's' printed with 'font'
// Return NULL if arguments are invalid
Shapoid* TGAFontGetStringBound(TGAFont *font, unsigned char *s);

// Get the angle of the right vector of the font with the abciss
// Return 0.0 if the arguments are invalid or memory allocation failed
float TGAFontGetAngleWithAbciss(TGAFont *font);
```

```
// Get the average color of the whole image
// Return a TGAPixel set to the avergae color, or NULL if the arguments
// are invalid
TGAPixel *TGAGetAverageColor(TGA *tga);

// Set the read only flag of a TGAPixel
// Do nothing if arguments are invalid
void TGAPixelSetReadOnly(TGAPixel *pix, bool v);

// Set the read only flag of all the TGAPixel of a TGA
// Do nothing if arguments are invalid
void TGAPixelSetAllReadOnly(TGA *tga, bool v);

// Get the read only flag of a TGAPixel
// Return true if arguments are invalid
bool TGAPixelIsReadOnly(TGAPixel *pix);

// Create a TGALayer of width dim[0] and height dim[1] and background
// color equal to 'pixel'
// If 'pixel' is NULL rgba(0,0,0,0) is used
// Return NULL in case of invalid arguments or memory allocation
// failure
TGALayer* TGALayerCreate(VecShort *dim, TGAPixel *pixel);

// Clone a TGALayer
// Return NULL in case of failure
TGALayer* TGALayerClone(TGALayer *that);

// Free the memory used by the TGALayer
void TGALayerFree(TGALayer **that);

// Set the current layer to the 'iLayer'-th layer
// Do nothing if arguments are invalid
void TGASetCurLayer(TGA *that, int iLayer);

// Add a layer above the current one
// Do nothing if the arguments are invalid
void TGAAddLayer(TGA *that);

// Blend layers 'that' and 'tho', the result is stored into 'that'
// 'tho' is considered to above 'that'
// If VecShort 'bound' is not null only pixels inside the box
// (bound[0],bound[1])-(bound[2],bound[3]) (included) are blended
// 'that' and 'tho' must have same dimension
// Do nothing if arguments are invalid
void TGALayerBlend(TGALayer *that, TGALayer *tho, VecShort *bound);

// Get a pointer to the pixel at coord (x,y) = (pos[0],pos[1])
// in the layer 'that'
// Return NULL in case of invalid arguments
TGAPixel* TGALayerGetPix(TGALayer *that, VecShort *pos);

// Set the color of one pixel at coord (x,y) = (pos[0],pos[1]) to 'pix'
// in the layer 'that'
// Do nothing in case of invalid arguments
void TGALayerSetPix(TGALayer *that, VecShort *pos, TGAPixel *pix);

// Draw one stroke at 'pos' with 'pen'
// in layer 'that'
// Do nothing in case of invalid arguments
void TGALayerStrokePix(TGALayer *that, VecFloat *pos, TGAPencil *pen);
```

```
// Return true if 'pos' is inside 'that'
// Return false else, or if arguments are invalid
bool TGALayerIsPosInside(TGALayer *that, VecShort *pos);

// Erase the content of the layer 'that'
// (set all pixel to rgba(0,0,0,0) and readonly to false)
// Do nothing in case of invalid argument
void TGALayerClean(TGALayer *that);

#endif
```

2 Code

2.1 tgapaint.c

```
// *************** TGAPAINT.C ***************

// ================= Include =================

#include "tgapaint.h"
#include "tgafont.c"

// ================= Define ==================

#define TGA_PI 3.14159
#define TGA_EPSILON 0.001

// ================ Functions declaration ====================

// Function to decode rgba values when loading a TGA file
// Do nothing if arguments are invalid
void MergeBytes(TGAPixel *pixel, unsigned char *p, int bytes);

// Draw one stroke at 'pos' with 'pen' of type tgaPenShapoid
// in current layer
// Don't do anything in case of invalid arguments
void TGAStrokePixShapoid(TGA *tga, VecFloat *pos, TGAPencil *pen);

// Draw one stroke at 'pos' with 'pen' of type tgaPenPixel
// in current layer
// Don't do anything in case of invalid arguments
void TGAStrokePixOnePixel(TGA *tga, VecFloat *pos, TGAPencil *pen);

// Draw one stroke at 'pos' with 'pen' of type tgaPenShapoid
// in layer 'that'
// Don't do anything in case of invalid arguments
void TGALayerStrokePixShapoid(TGALayer *that,
  VecFloat *pos, TGAPencil *pen);

// Draw one stroke at 'pos' with 'pen' of type tgaPenPixel
// in layer 'that'
// Don't do anything in case of invalid arguments
void TGALayerStrokePixOnePixel(TGALayer *that,
  VecFloat *pos, TGAPencil *pen);

// Add the BCurve 'curve' (must be of dimension 2 and order > 0)
// in 'layer'
// do nothing if arguments are invalid
void TGALayerAddCurve(TGALayer *layer, BCurve *curve, TGAPencil *pen);

// ================ Functions implementation ==================

// Create a TGA of width dim[0] and height dim[1] and background
// color equal to pixel
// If 'pixel' is NULL rgba(0,0,0,0) is used
// (0,0) is the bottom left corner, x toward right, y toward top
// Return NULL in case of invalid arguments or memory allocation
// failure
TGA* TGACreate(VecShort *dim, TGAPixel *pixel) {
  // Check arguments
  if (dim == NULL) return NULL;
  // Allocate memory
  TGA *ret = (TGA*)malloc(sizeof(TGA));
  // If we couldn't allocate memory
  if (ret == NULL)
    // Return NULL
    return NULL;
  // Set the pointers to NULL
  ret->_header = NULL;
  ret->_layers = NULL;
  ret->_curLayer = NULL;
  // Allocate memory for the header
  ret->_header = (TGAHeader*)malloc(sizeof(TGAHeader));
  // If we couldn't allocate memory
  if (ret->_header == NULL) {
    // Free memory for the TGA
    free(ret);
    // Return NULL
    return NULL;
  }
  // Set a pointer to the header
  TGAHeader *h = ret->_header;
  // Initialize the header values
  h->_idLength = 0;
  h->_colorMapType = 0;
  h->_dataTypeCode = 2;
  h->_colorMapOrigin = 0;
  h->_colorMapLength = 0;
  h->_colorMapDepth = 0;
  h->_xOrigin = 0;
  h->_yOrigin = 0;
  h->_width = VecGet(dim, 0);
  h->_height = VecGet(dim, 1);
  h->_bitsPerPixel = 32;
  h->_imageDescriptor = 0;
  // Create the set of layers
  ret->_layers = GSetCreate();
  if (ret->_layers == NULL) {
    // Free the memory for the TGA
    free(ret->_header);
    free(ret);
    // Return NULL
    return NULL;
  }
  // Create one layer
  ret->_curLayer = TGALayerCreate(dim, pixel);
  // Create the temporary working layer
  ret->_tmpLayer = TGALayerCreate(dim, pixel);
  // If we couldn't allocate memory
  if (ret->_curLayer == NULL || ret->_tmpLayer == NULL) {
    // Free the memory for the TGA
    TGALayerFree(&(ret->_tmpLayer));
    TGALayerFree(&(ret->_curLayer));
    GSetFree(&(ret->_layers));
    free(ret->_header);
    free(ret);
    // Return NULL
    return NULL;
  }
  // Add the layer to the set
  GSetPush(ret->_layers, ret->_curLayer);
  // Initialize the current layer index
  ret->_curLayerIndex = 0;
  // Return the created TGA
  return ret;
}

// Clone a TGA
// Return NULL in case of failure
TGA* TGAClone(TGA *tga) {
  // Check arguments
  if (tga == NULL)
    return NULL;
  // Allocate memory for the cloned TGA
  TGA *ret = (TGA*)malloc(sizeof(TGA));
  // If we could allocate memory
  if (ret != NULL) {
    // Allocate memory for the header
    ret->_header = (TGAHeader*)malloc(sizeof(TGAHeader));
    // If we couldn't allocate memory
    if (ret->_header == NULL) {
      // Free the memory for the cloned TGA
      free(ret);
      // Return NULL
      return NULL;
    }
    // Copy the header
    memcpy(ret->_header, tga->_header, sizeof(TGAHeader));
    // Clone the layers
    GSetElem *elem = tga->_layers->_head;
    while (elem != NULL) {
      TGALayer *layer = TGALayerClone((TGALayer*)(elem->_data));
      if (layer == NULL) {
        TGAFree(&ret);
        return NULL;
      }
      GSetAppend(ret->_layers, layer);
      elem = elem->_next;
    }
  }
  // Return the cloned TGA
  return ret;
}

// Free the memory used by the TGA
void TGAFree(TGA **tga) {
  // Check arguments
  if (tga == NULL || *tga == NULL)
    return;
  // If the header has been allocated
  if ((*tga)->_header != NULL) {
    // Free the memory for the header
    free((*tga)->_header);
    (*tga)->_header = NULL;
  }
  // Free the layers
  TGALayerFree(&((*tga)->_tmpLayer));
  (*tga)->_curLayer = NULL;
  TGALayer *layer = (TGALayer*)GSetPop((*tga)->_layers);
  while (layer != NULL) {
    TGALayerFree(&layer);
    layer = (TGALayer*)GSetPop((*tga)->_layers);
  }
  GSetFree(&((*tga)->_layers));
  // Free the TGA
  free(*tga);
  *tga = NULL;
}

// Load a TGA from the file pointed to by 'fileName'
// If 'tga' already contains a TGA, it is overwritten
// return 0 upon success, else
// 1 : couldn't open the file
// 2 : malloc failed
// 3 : can only handle image type 2 and 10
// 4 : can only handle pixel depths of 16, 24, and 32
// 5 : can only handle colour map types of 0 and 1
// 6 : unexpected end of file
// 7 : invalid arguments
int TGALoad(TGA **tga, char *fileName) {
  // Check arguments
  if (fileName == NULL) return 7;
  // If the TGA in argument is already used
  if (*tga != NULL)
    // Free memory
    TGAFree(tga);
  // Create a VecShort to memorize the dimensions
  VecShort *dim = VecShortCreate(2);
  // If we couldn't allocate memory
  if (dim == NULL) {
    // Stop here
    return 2;
  }
  // Allocate memory for the TGA
  *tga = (TGA*)malloc(sizeof(TGA));
  // If we couldn't allocate memory
  if (*tga == NULL) {
    // Stop here
    TGAFree(tga);
    return 2;
  }
  // Set pointers to NULL
  (*tga)->_header = NULL;
  (*tga)->_layers = NULL;
  (*tga)->_curLayer = NULL;
  (*tga)->_tmpLayer = NULL;
  // Declare variables used during decoding
  int n = 0, i = 0, j = 0;
  unsigned int bytes2read = 0, skipover = 0;
  unsigned char p[5] = {0};
  size_t ret = 0;
  // Open the file
  FILE *fptr = fopen(fileName,"r");
```

```
// If we couldn't open the file
if (fptr == NULL) {
  // Stop here
  TGAFree(tga);
  return 1;
}
// Allocate memory for the header
(*tga)->_header = (TGAHeader*)malloc(sizeof(TGAHeader));
// If we couldn't allocate memory
if ((*tga)->_header == NULL) {
  // Stop here
  TGAFree(tga);
  fclose(fptr);
  return 2;
}
// Set a pointer to the header
TGAHeader *h = (*tga)->_header;
// Read the header's values
h->_idLength = fgetc(fptr);
h->_colorMapType = fgetc(fptr);
h->_dataTypeCode = fgetc(fptr);
ret = fread(&(h->_colorMapOrigin), 2, 1, fptr);
ret = fread(&(h->_colorMapLength), 2, 1, fptr);
h->_colorMapDepth = fgetc(fptr);
ret = fread(&(h->_xOrigin), 2, 1, fptr);
ret = fread(&(h->_yOrigin), 2, 1, fptr);
ret = fread(&(h->_width), 2, 1, fptr);
ret = fread(&(h->_height), 2, 1, fptr);
h->_bitsPerPixel = fgetc(fptr);
h->_imageDescriptor = fgetc(fptr);
// Create the set of layers
(*tga)->_layers = GSetCreate();
if ((*tga)->_layers == NULL) {
  // Free the memory for the TGA
  free((*tga)->_header);
  free((*tga));
  return 2;
}
// Create one layer
VecSet(dim, 0, h->_width);
VecSet(dim, 1, h->_height);
(*tga)->_curLayer = TGALayerCreate(dim, NULL);
// If we couldn't allocate memory
if ((*tga)->_curLayer == NULL) {
  // Free the memory for the TGA
  free((*tga)->_layers);
  free((*tga)->_header);
  free((*tga));
  return 2;
}
// Add the layer to the set
GSetPush((*tga)->_layers, (*tga)->_curLayer);
// Set a pointer to the pixel
TGAPixel *pix = (*tga)->_curLayer->_pixels;
// For each pixel
for (i = 0; i < h->_width * h->_height; ++i) {
  // For each value RGBA
  for (int irgb = 0; irgb < 4; ++irgb)
    // Initialize the value to 0
    pix[i]._rgba[irgb] = 0;
  pix[i]._readOnly = false;
}
```

```
// If the data type is not supported
if (h->_dataTypeCode != 2 && h->_dataTypeCode != 10) {
  // Stop here
  TGAFree(tga);
  fclose(fptr);
  return 3;
}
// If the number of byte per pixel is not supported
if (h->_bitsPerPixel != 16 &&
  h->_bitsPerPixel != 24 &&
  h->_bitsPerPixel != 32) {
  // Stop here
  TGAFree(tga);
  fclose(fptr);
  return 4;
}
// If the color map type is not supported
if (h->_colorMapType != 0 &&
  h->_colorMapType != 1) {
  // Stop here
  TGAFree(tga);
  fclose(fptr);
  return 5;
}
// Skip the unused information
skipover += h->_idLength;
skipover += h->_colorMapType * h->_colorMapLength;
fseek(fptr,skipover,SEEK_CUR);
// Calculate the number of byte per pixel
bytes2read = h->_bitsPerPixel / 8;
// For each pixel
while (n < h->_width * h->_height) {
  // Read the pixel according to the data type, merge and
  // move to the next pixel
  if (h->_dataTypeCode == 2) {
    if (fread(p, 1, bytes2read, fptr) != bytes2read) {
      TGAFree(tga);
      fclose(fptr);
      return 6;
    }
    MergeBytes(&(pix[n]), p, bytes2read);
    ++n;
  } else if (h->_dataTypeCode == 10) {
    if (fread(p, 1, bytes2read + 1, fptr) != bytes2read + 1) {
      TGAFree(tga);
      fclose(fptr);
      return 6;
    }
    j = p[0] & 0x7f;
    MergeBytes(&(pix[n]), &(p[1]), bytes2read);
    ++n;
    if (p[0] & 0x80) {
      for (i = 0; i < j; ++i) {
         MergeBytes(&(pix[n]), &(p[1]), bytes2read);
         ++n;
      }
    } else {
      for (i = 0; i < j; ++i) {
        if (fread(p, 1, bytes2read, fptr) != bytes2read) {
          TGAFree(tga);
          fclose(fptr);
          return 6;
          }
          MergeBytes(&(pix[n]), p, bytes2read);
          ++n;
        }
      }
    }
  }
  // Close the file
  fclose(fptr);
  // To avoid warning
  ret = ret;
  // Free memory
  VecFree(&dim);
  // Return success code
  return 0;
}

// Save the TGA 'tga' to the file pointed to by 'fileName'
// return 0 upon success, else
// 1 : couldn't open the file
// 2 : invalid arguments
int TGASave(TGA *tga, char *fileName) {
  // Check arguments
  if (tga == NULL || fileName == NULL ||
    tga->_header == NULL || tga->_layers == NULL)
    return 2;
  // Open the file
  FILE *fptr = fopen(fileName,"w");
  // If we couln't open the file
  if (fptr == NULL)
    // Stop here
    return 1;
  // Write the header
  // Set a pointer to the header
  TGAHeader *h = tga->_header;
  putc(h->_idLength, fptr);
  putc(h->_colorMapType, fptr);
  putc(2, fptr); // _dataTypeCode
  fwrite(&(h->_colorMapOrigin), 2, 1, fptr);
  fwrite(&(h->_colorMapLength), 2, 1, fptr);
  putc(h->_colorMapDepth, fptr);
  fwrite(&(h->_xOrigin), 2, 1, fptr);
  fwrite(&(h->_yOrigin), 2, 1, fptr);
  fwrite(&(h->_width), 2, 1, fptr);
  fwrite(&(h->_height), 2, 1, fptr);
  putc(32, fptr); // _bitsPerPixel
  putc(h->_imageDescriptor, fptr);
  // For each pixel
  for (int i = 0;
    i < tga->_header->_height * tga->_header->_width; ++i) {
    // Write the pixel values
    putc(tga->_curLayer->_pixels[i]._rgba[2], fptr);
    putc(tga->_curLayer->_pixels[i]._rgba[1], fptr);
    putc(tga->_curLayer->_pixels[i]._rgba[0], fptr);
    putc(tga->_curLayer->_pixels[i]._rgba[3], fptr);
  }
  // Close the file
  fclose(fptr);
  // Return the success code
  return 0;
}
```

```
// Print the header of 'tga' on 'stream'
// If arguments are invalid, do nothing
void TGAPrintHeader(TGA *tga, FILE *stream) {
  // Check arguments
  if (tga == NULL || stream == NULL) return;
  // Set a pointer to the header
  TGAHeader *h = tga->_header;
  // If the header is not defined
  if (h == NULL)
    // Stop here
    return;
  // Print the header info
  fprintf(stream, "ID length:         %d\n", h->_idLength);
  fprintf(stream, "Colourmap type:    %d\n", h->_colorMapType);
  fprintf(stream, "Image type:        %d\n", h->_dataTypeCode);
  fprintf(stream, "Colour map offset: %d\n", h->_colorMapOrigin);
  fprintf(stream, "Colour map length: %d\n", h->_colorMapLength);
  fprintf(stream, "Colour map depth:  %d\n", h->_colorMapDepth);
  fprintf(stream, "X origin:          %d\n", h->_xOrigin);
  fprintf(stream, "Y origin:          %d\n", h->_yOrigin);
  fprintf(stream, "Width:             %d\n", h->_width);
  fprintf(stream, "Height:            %d\n", h->_height);
  fprintf(stream, "Bits per pixel:    %d\n", h->_bitsPerPixel);
  fprintf(stream, "Descriptor:        %d\n", h->_imageDescriptor);
}

// Return true if 'pos' is inside 'tga'
// Return false else, or if arguments are invalid
bool TGAIsPosInside(TGA *tga, VecShort *pos) {
  // Check arguments
  if (tga == NULL || pos == NULL || VecDim(pos) < 2)
    return false;
  // If the position is in the tga
  if (VecGet(pos, 0) >= 0 && VecGet(pos, 0) < tga->_header->_width &&
    VecGet(pos, 1) >= 0 && VecGet(pos, 1) < tga->_header->_height)
    return true;
  // Else, the position is not in the tga
  else
    return false;
}

// Get a pointer to the pixel at coord (x,y) = (pos[0],pos[1])
// in the current layer
// Return NULL in case of invalid arguments
TGAPixel* TGAGetPix(TGA *tga, VecShort *pos) {
  // Check arguments
  if (tga == NULL || pos == NULL ||
    tga->_layers == NULL || tga->_header == NULL)
    return NULL;
  // Return a pointer toward the requested pixel in the current layer
  return TGALayerGetPix(tga->_curLayer, pos);
}

// Set the color of one pixel at coord (x,y) = (pos[0],pos[1]) to 'pix'
// Do nothing in case of invalid arguments
void TGASetPix(TGA *tga, VecShort *pos, TGAPixel *pix) {
  // Check arguments
  if (tga == NULL)
    return;
  // Set the pixel in the current layer
  TGALayerSetPix(tga->_curLayer, pos, pix);
}
```

```
// Draw one stroke at 'pos' with 'pen' of type tgaPenShapoid
// in current layer
// Don't do anything in case of invalid arguments
void TGAStrokePixShapoid(TGA *tga, VecFloat *pos, TGAPencil *pen) {
  // Check arguments
  if (tga == NULL)
    return;
  // Stroke in the current layer
  TGALayerStrokePixShapoid(tga->_curLayer, pos, pen);
}

// Draw one stroke at 'pos' with 'pen' of type tgaPenPixel
// in current layer
// Don't do anything in case of invalid arguments
void TGAStrokePixOnePixel(TGA *tga, VecFloat *pos, TGAPencil *pen) {
  // Check arguments
  if (tga == NULL)
    return;
  // Stroke in the current layer
  TGALayerStrokePixOnePixel(tga->_curLayer, pos, pen);
}

// Draw one stroke at 'pos' with 'pen' of type tgaPenPixel
// in layer 'that'
// Don't do anything in case of invalid arguments
void TGALayerStrokePixOnePixel(TGALayer *that,
  VecFloat *pos, TGAPencil *pen) {
  // Check arguments
  if (that == NULL || pos == NULL || pen == NULL) return;
  // Declare a variable for the integer position of the
  // current pixel
  VecShort *q = VecShortCreate(2);
  if (q == NULL)
    return;
  VecSet(q, 0, (short)floor(VecGet(pos, 0)));
  VecSet(q, 1, (short)floor(VecGet(pos, 1)));
  // Get the curent pixel of the tga
  TGAPixel *pixTga = TGALayerGetPix(that, q);
  // If the pixel is not in read only mode
  if (TGAPixelIsReadOnly(pixTga) == false) {
    // Get the curent pixel of the pencil
    TGAPixel *pixPen = TGAPencilGetPixel(pen);
    // Get a mix of colors
    TGAPixel *pix = TGAPixelMix(pixTga, pixPen, 1.0);
    // Set the color of the current pixel
    memcpy(pixTga, pix, sizeof(TGAPixel));
    // Free the memory used by the pixel from the pencil
    TGAPixelFree(&pixPen);
    TGAPixelFree(&pix);
    VecFree(&q);
  }
}

// Draw one stroke at 'pos' with 'pen' of type tgaPenShapoid
// in layer 'that'
// Don't do anything in case of invalid arguments
void TGALayerStrokePixShapoid(TGALayer *that,
  VecFloat *pos, TGAPencil *pen) {
  // Check arguments
  if (that == NULL || pos == NULL || pen == NULL) return;
  // Get the curent color of the pencil
```

```
TGAPixel *pix = TGAPencilGetPixel(pen);
// Declare variable for coordinates of pixel
VecFloat *p = VecFloatCreate(2);
// Declare a clone of the pen tip
Shapoid *penTip = ShapoidClone(pen->_tip);
// Declare a variable for the integer position of the
// current pixel
VecShort *q = VecShortCreate(2);
// Declare a Facoid to represent the pixel
Shapoid *pixel = FacoidCreate(2);
// If we couldn't allocate memory or get the necessary information
if (q == NULL || p == NULL || pixel == NULL || penTip == NULL) {
  // Free memory and stop here
  VecFree(&p);
  VecFree(&q);
  ShapoidFree(&pixel);
  ShapoidFree(&penTip);
  return;
}
// Translate the clone of the pen tip to the pos
ShapoidTranslate(penTip, pos);
// Get the bounding box of the pen tip
Shapoid *tipBox = ShapoidGetBoundingBox(penTip);
// If we couldn't allocate memory
if (tipBox == NULL) {
  // Free memory and stop here
  VecFree(&p);
  VecFree(&q);
  ShapoidFree(&pixel);
  ShapoidFree(&penTip);
  return;
}
// Get the end pos of the tip box to avoid recalculate them
float end[2];
for (int i = 2; i--;)
  end[i] = VecGet(tipBox->_pos, i) + VecGet(tipBox->_axis[i], i);
// Declare a variable to memorize the step in position
float delta = 0.5 * pen->_thickness;
if (delta > 1.0) delta = 1.0;
// For each pixel in the area affected by the pencil
for (VecSet(p, 0, VecGet(tipBox->_pos, 0));
  VecGet(p, 0) < end[0] + TGA_EPSILON;
  VecSet(p, 0, VecGet(p, 0) + delta)) {
  for (VecSet(p, 1, VecGet(tipBox->_pos, 1));
    VecGet(p, 1) < end[1] + TGA_EPSILON;
    VecSet(p, 1, VecGet(p, 1) + delta)) {
    if (ShapoidIsPosInside(penTip, p) == true) {
      // Get the integer position of the current pixel
      for (int i = 2; i--;)
        VecSet(q, i, (short)floor(VecGet(p, i)));
      // Get a pointer to the current pixel
      TGAPixel *curPix = TGALayerGetPix(that, q);
      // If the pixel is in the tga
      if (curPix != NULL && TGAPixelIsReadOnly(curPix) == false) {
        // If the pen doesn't use antialias
        if (pen->_antialias == false) {
          // Set the value of the pixel
          memcpy(curPix->_rgba, pix->_rgba,
            sizeof(unsigned char) * 4);
        // Else, if the pencil uses antialias
        } else {
          // Position the pixel Facoid
```

```
            for(int i = 2; i--;)
              VecSet(pixel->_pos, i, floor(VecGet(p, i)));
            // Get the ratio coverage of this pixel by the pen tip
            float ratio = ShapoidGetCoverage(penTip, pixel);
            // Blend the current pixel with the pixel from
            // the pencil
            TGAPixel *blendPix = TGAPixelMix(curPix, pix, ratio);
            //TGAPixel *blendPix = TGAPixelBlend(curPix, mixPix, ratio);
            //TGAPixel *blendPix = TGAPixelBlend(curPix, pix, ratio);
            // If the blended pixel is not null
            if (blendPix != NULL) {
              // Set the current pixel to the blended pixel
              memcpy(curPix->_rgba, blendPix->_rgba,
                sizeof(unsigned char) * 4);
              // Free memory used by the blended pixel
              TGAPixelFree(&blendPix);
            }
            //if (ratio >= 1.0 - PBMATH_EPSILON)
              //curPix->_readOnly = true;
          }
        }
      }
    }
  }
  // Free memory
  TGAPixelFree(&pix);
  VecFree(&p);
  VecFree(&q);
  ShapoidFree(&tipBox);
  ShapoidFree(&pixel);
  ShapoidFree(&penTip);
}

// Draw one stroke at 'pos' with 'pen'
// in current layer
// Do nothing in case of invalid arguments
void TGAStrokePix(TGA *tga, VecFloat *pos, TGAPencil *pen) {
  // Check arguments
  if (tga == NULL)
    return;
  // Stroke in current layer
  TGALayerStrokePix(tga->_curLayer, pos, pen);
}

// Draw a line between 'from' and 'to' with pencil 'pen'
// pixels outside the TGA are ignored
// do nothing if arguments are invalid
void TGADrawLine(TGA *tga, VecFloat *from, VecFloat *to,
  TGAPencil *pen) {
  // Create a BCurve equivalent to the line
  BCurve *curve = BCurveCreate(1, 2);
  BCurveSet(curve, 0, from);
  BCurveSet(curve, 1, to);
  // Draw a curve with control points located at anchor points
  TGADrawCurve(tga, curve, pen);
  // Free memory
  BCurveFree(&curve);
}

// Draw the BCurve 'curve' (must be of dimension 2 and order > 0)
// do nothing if arguments are invalid
void TGADrawBCurve(TGA *tga, BCurve *curve, TGAPencil *pen) {
  // Check arguments
  if (tga == NULL || curve == NULL || pen == NULL ||
    BCurveOrder(curve) < 1)
    return;
  // Clean the working layer
  TGALayerClean(tga->_tmpLayer);
  // Draw the curve in the working layer
  TGALayerAddCurve(tga->_tmpLayer, curve, pen);
  // Get the bounding box of the curve
  VecShort *bound = NULL;
  Shapoid *shape = BCurveGetBoundingBox(curve);
  if (shape != NULL) {
    bound = VecShortCreate(4);
    if (bound != NULL) {
      for (int i = 2; i--;) {
        VecSet(bound, i, VecGet(shape->_pos, i) - pen->_thickness);
        VecSet(bound, 2 + i, VecGet(shape->_pos, i) + pen->_thickness);
      }
      for (int i = 2; i--;)
        VecSet(bound, 2 + i,
          VecGet(bound, 2 + i) + VecGet(shape->_axis[i], i));
    }
  }
  // Blend the working layer in the current layer
  TGALayerBlend(tga->_curLayer, tga->_tmpLayer, bound);
  // Free memory
  VecFree(&bound);
  ShapoidFree(&shape);
}

// Draw the SCurve 'curve' (must be of dimension 2)
// do nothing if arguments are invalid
void TGADrawSCurve(TGA *tga, SCurve *curve, TGAPencil *pen) {
  // Check arguments
  if (tga == NULL || curve == NULL || pen == NULL)
    return;
  // Clean the working layer
  TGALayerClean(tga->_tmpLayer);
  // Declare a pointer to loop on BCurves of the SCurve
  GSetElem *ptr = curve->_curves->_head;
  while (ptr != NULL) {
    // Draw the curve in the working layer
    TGALayerAddCurve(tga->_tmpLayer, (BCurve*)(ptr->_data), pen);
    // Move to the next curve
    ptr = ptr->_next;
  }
  // Get the bounding box of the curve
  VecShort *bound = NULL;
  Shapoid *shape = SCurveGetBoundingBox(curve);
  if (shape != NULL) {
    bound = VecShortCreate(4);
    if (bound != NULL) {
      for (int i = 2; i--;) {
        VecSet(bound, i, VecGet(shape->_pos, i) - pen->_thickness);
        VecSet(bound, 2 + i, VecGet(shape->_pos, i) + pen->_thickness);
      }
      for (int i = 2; i--;)
        VecSet(bound, 2 + i,
          VecGet(bound, 2 + i) + VecGet(shape->_axis[i], i));
    }
  }
  // Blend the working layer in the current layer
  TGALayerBlend(tga->_curLayer, tga->_tmpLayer, bound);
  // Free memory
  VecFree(&bound);
  ShapoidFree(&shape);
}

// Draw a rectangle between ’from’ and ’to’ with pencil ’pen’
// pixels outside the TGA are ignored
// do nothing if arguments are invalid
void TGADrawRect(TGA *tga, VecFloat *from, VecFloat *to,
  TGAPencil *pen) {
  // Check arguments
  if (tga == NULL || from == NULL || to == NULL || pen == NULL)
    return;
  // Create the Facoid equivalent to the rectangle
  Shapoid *facoid = FacoidCreate(2);
  if (facoid != NULL) {
    ShapoidSetPos(facoid, from);
    VecFloat *s = VecGetOp(to, 1.0, from, -1.0);
    ShapoidScale(facoid, s);
    VecFree(&s);
    // Draw the Facoid
    TGADrawShapoid(tga, facoid, pen);
    // Free memory
    ShapoidFree(&facoid);
  }
}

// Fill a rectangle between ’from’ and ’to’ with pencil ’pen’
// pixels outside the TGA are ignored
// do nothing if arguments are invalid
void TGAFillRect(TGA *tga, VecFloat *from, VecFloat *to,
  TGAPencil *pen) {
  // Check arguments
  if (tga == NULL || from == NULL || to == NULL || pen == NULL)
    return;
  // Create the Facoid equivalent to the rectangle
  Shapoid *facoid = FacoidCreate(2);
  if (facoid != NULL) {
    ShapoidSetPos(facoid, from);
    VecFloat *s = VecGetOp(to, 1.0, from, -1.0);
    ShapoidScale(facoid, s);
    VecFree(&s);
    // Draw the Facoid
    TGAFillShapoid(tga, facoid, pen);
    // Free memory
    ShapoidFree(&facoid);
  }
}

// Draw a ellipse at ’center’ of radius ’r’ (Rx,Ry)
// with pencil ’pen’
// pixels outside the TGA are ignored
// do nothing if arguments are invalid
void TGADrawEllipse(TGA *tga, VecFloat *center, VecFloat *r,
  TGAPencil *pen) {
  // Check arguments
  if (tga == NULL || center == NULL || r == NULL || pen == NULL ||
    VecGet(r, 0) <= 0.0 || VecGet(r, 1) <= 0.0)
    return;
  // Create the Spheroid equivalent to the ellipse
  Shapoid *spheroid = SpheroidCreate(2);
  if (spheroid != NULL) {
    ShapoidSetPos(spheroid, center);
    // Declare a variable to memorize the diameter of the ellipse
    VecFloat *diameter = VecGetOp(r, 2.0, NULL, 0.0);
    if (diameter != NULL) {
      // Scale the Spheroid
      ShapoidScale(spheroid, diameter);
      VecFree(&diameter);
      // Draw the Spheroid
      TGADrawShapoid(tga, spheroid, pen);
    }
    // Free memory
    ShapoidFree(&spheroid);
  }
}

// Fill an ellipse at 'center' of radius 'r' (Rx, Ry) with pencil 'pen'
// pixels outside the TGA are ignored
// do nothing if arguments are invalid
void TGAFillEllipse(TGA *tga, VecFloat *center, VecFloat *r,
  TGAPencil *pen) {
  // Check arguments
  if (tga == NULL || center == NULL || r == NULL || pen == NULL ||
    VecGet(r, 0) <= 0.0 || VecGet(r, 1) <= 0.0)
    return;
  // Create the Spheroid
  Shapoid *spheroid = SpheroidCreate(2);
  if (spheroid != NULL) {
    ShapoidSetPos(spheroid, center);
    // Declare a variable to memorize the diameter of the ellipse
    VecFloat *diameter = VecGetOp(r, 2.0, NULL, 0.0);
    if (diameter != NULL) {
      // Scale the Spheroid
      ShapoidScale(spheroid, diameter);
      VecFree(&diameter);
      // Draw the Spheroid
      TGAFillShapoid(tga, spheroid, pen);
    }
    // Free memory
    ShapoidFree(&spheroid);
  }
}

// Draw the shapoid 's' with pencil 'pen'
// The shapoid must be of dimension 2
// Pixels outside the TGA are ignored
// Do nothing if arguments are invalid
void TGADrawShapoid(TGA *tga, Shapoid *s, TGAPencil *pen) {
  // Check arguments
  if (tga == NULL || s == NULL || pen == NULL || ShapoidGetDim(s) != 2)
    return;
  // Get the SCurve equivalent to the Shapoid
  SCurve *curve = Shapoid2SCurve(s);
  // If we could get the SCurve
  if (curve != NULL) {
    // Draw the SCurve
    TGADrawSCurve(tga, curve, pen);
    // Free memory
    SCurveFree(&curve);
  }
}
```

```
// Fill the shapoid 's' with pencil 'pen'
// The shapoid must be of dimension 2
// Pixels outside the TGA are ignored
// Do nothing if arguments are invalid
void TGAFillShapoid(TGA *tga, Shapoid *s, TGAPencil *pen) {
  // Check arguments
  if (tga == NULL || s == NULL || pen == NULL ||
    ShapoidGetDim(s) != 2)
    return;
  // Clean the working layer
  TGALayerClean(tga->_tmpLayer);

  // Get the bounding box of the curve
  VecShort *bound = NULL;
  Shapoid *bounding = ShapoidGetBoundingBox(s);
  // If we could get the bounding box
  if (bounding != NULL) {
    bound = VecShortCreate(4);
    if (bound != NULL) {
      for (int i = 2; i--;) {
        VecSet(bound, i, VecGet(bounding->_pos, i) - pen->_thickness);
        VecSet(bound, 2 + i, VecGet(bounding->_pos, i) +
          pen->_thickness);
      }
      for (int i = 2; i--;)
        VecSet(bound, 2 + i,
          VecGet(bound, 2 + i) + VecGet(bounding->_axis[i], i));
    }
    // Declare a variable to memorize the upper right limit of
    // the bounding box
    VecFloat *to =
      VecGetOp(bounding->_pos, 1.0, bounding->_axis[0], 1.0);
    VecOp(to, 1.0, bounding->_axis[1], 1.0);
    // If we couldn't get the upper right limit
    if (to == NULL) {
      // Free memory and stop here
      ShapoidFree(&bounding);
      return;
    }
    // Declare a variable to memorize the pixel position
    VecFloat *pos = VecFloatCreate(2);
    // If we couldn't allocate memory
    if (pos == NULL) {
      // Free memory and stop here
      ShapoidFree(&bounding);
      VecFree(&to);
      return;
    }
    // For each pixel in the bounding box
    for (VecSet(pos, 0, VecGet(bounding->_pos, 0));
      VecGet(pos, 0) < VecGet(to, 0) + PBMATH_EPSILON;
      VecSet(pos, 0, VecGet(pos, 0) + 1.0)) {
      for (VecSet(pos, 1, VecGet(bounding->_pos, 1));
        VecGet(pos, 1) < VecGet(to, 1) + PBMATH_EPSILON;
        VecSet(pos, 1, VecGet(pos, 1) + 1.0)) {
        // If the pixel is in the Shapoid
        if (ShapoidIsPosInside(s, pos) == true) {
          // Set the blend of the pencil with the depth of the pos
          // in the shapoid for the case the pencil is in
          // tgaPenBlend mode
          TGAPencilSetBlend(pen, 1.0 - ShapoidGetPosDepth(s, pos));
          // Draw the pixel
          TGALayerStrokePix(tga->_tmpLayer, pos, pen);
        }
      }
    }
    // Blend the working layer in the current layer
    TGALayerBlend(tga->_curLayer, tga->_tmpLayer, bound);
    // Free memory
    ShapoidFree(&bounding);
    VecFree(&to);
    VecFree(&pos);
    VecFree(&bound);
  }
}

// Apply a gaussian blur of 'strength' and 'range' perimeter on the TGA
// Do nothing if arguments are invalid
void TGAFilterGaussBlur(TGA *tga, float strength, float range) {
  // Check arguments
  if (tga == NULL || tga->_header == NULL || strength <= 0.0)
    return;
  // Create a Gauss
  Gauss *gauss = GaussCreate(0.0, strength);
  // If we couldn't create the gauss
  if (gauss == NULL) {
    // Stop here
    return;
  }
  // Allocate memory for a temporary buffer
  float *drgb = (float*)malloc(tga->_header->_width *
    tga->_header->_height * 4 * sizeof(float));
  // If we couldn't allocate memory
  if (drgb == NULL) {
    // Stop here
    GaussFree(&gauss);
    return;
  }
  // Declare a variable for passing argument
  VecShort *v = VecShortCreate(2);
  if (v == NULL) {
    // Stop here
    GaussFree(&gauss);
    free(drgb);
    return;
  }
  // Declare variable to memorize current pixel
  short px[2];
  // Declare variable to memorize index of rgba
  int irgb = 0;
  // For each pixel
  for (px[0] = tga->_header->_width; px[0]--;) {
    for (px[1] = tga->_header->_height; px[1]--;) {
      // Get index of the current pixel
      long int index = 4 * (px[1] * tga->_header->_width + px[0]);
      // For each rgba value
      for (irgb = 4; irgb--;)
        // Initialize the value in the temporary buffer to 0
        drgb[index + irgb] = 0.0;
    }
  }
  // For each pixel
  for (px[0] = tga->_header->_width; px[0]--;) {
    for (px[1] = tga->_header->_height; px[1]--;) {
```

```
    // Get index of the current pixel
    long int indexp = 4 * (px[1] * tga->_header->_width + px[0]);
    // For each rgba value
    for (irgb = 4; irgb--;) {
      // Declare a variable to memorize position of pixel in range
      short qx[2];
      // Declare variables to calculate new value of rgba
      double sum = 0.0;
      double p = 0.0;
      // Calculate the corners positions of the area in range
      short from[2];
      short to[2];
      from[0] = (px[0] > range ? px[0] - range : 0);
      from[1] = (px[1] > range ? px[1] - range : 0);
      to[0] = (px[0] < tga->_header->_width - range ?
        px[0] + range : tga->_header->_width);
      to[1] = (px[1] < tga->_header->_height - range ?
        px[1] + range : tga->_header->_height);
      // For each pixel in range
      for (qx[0] = from[0]; qx[0] < to[0]; ++(qx[0])) {
        for (qx[1] = from[1]; qx[1] < to[1]; ++(qx[1])) {
          // Calculate the distance of this pixel to the current pixel
          double dist = sqrt(pow(qx[0] - px[0], 2.0) +
            pow(qx[1] - px[1], 2.0));
          // If this pixel is in range
          if (dist < range) {
            // Calculate the Gauss coefficient
            double g = GaussGet(gauss, dist);
            // Update the values to calculate the new rgba
            sum += g;
            VecSet(v, 0, qx[0]);
            VecSet(v, 1, qx[1]);
            TGAPixel *pixelQ = TGAGetPix(tga, v);
            p += g * (double)(pixelQ->_rgba[irgb]);
          }
        }
      }
      // Update the new value of the current pixel in the
      // temporary buffer
      drgb[indexp + irgb] = p / sum;
    }
  }
}
// For each pixel
for (px[0] = tga->_header->_width; px[0]--;) {
  for (px[1] = tga->_header->_height; px[1]--;) {
    // Get the index of the pixel
    long int index = 4 * (px[1] * tga->_header->_width + px[0]);
    // Get a pointer to the pixel
    VecSet(v, 0, px[0]);
    VecSet(v, 1, px[1]);
    TGAPixel *pixel = TGAGetPix(tga, v);
    // For each rgba value
    for (irgb = 4; irgb--;) {
      // Copy the new value from the temporary buffer to the tga
      pixel->_rgba[irgb] =
        (unsigned char)round(drgb[index + irgb]);
    }
  }
}
// Free memory
VecFree(&v);
```

```
  GaussFree(&gauss);
  free(drgb);
  drgb = NULL;
}

// Print the string 's' with its anchor position at 'pos', TGAPencil
// 'pen' and font 'font'
void TGAPrintString(TGA *tga, TGAPencil *pen, TGAFont *font,
  unsigned char *s, VecFloat *pos) {
  // Check arguments
  if (tga == NULL || pen == NULL || font == NULL || s == NULL ||
    pos == NULL)
    return;
  // Get the bounding box in pixel
  Shapoid* boundbox = TGAFontGetStringBound(font, s);
  // If we couldn't allocate memory
  if (boundbox == NULL)
    return;
  ShapoidTranslate(boundbox, pos);
  // Declare a variable to memorize the 'down by one line' vector
  VecFloat *down = VecClone(boundbox->_axis[1]);
  // If we couldn't allocate memory
  if (down == NULL)
    return;
  // Set the 'down by one line' vector
  VecNormalise(down);
  VecOp(down, -1.0 * font->_size * VecGet(font->_scale, 1), NULL, 0.0);
  // Declare a variable to memorize the 'down by one interspace' vector
  VecFloat *downspace = VecClone(boundbox->_axis[1]);
  // If we couldn't allocate memory
  if (downspace == NULL)
    return;
  // Set the 'down by one interspace' vector
  VecNormalise(downspace);
  VecOp(downspace, -1.0 * VecGet(font->_space, 1), NULL, 0.0);
  // Declare a variable to memorize the 'right by one char' vector
  VecFloat *right = VecClone(boundbox->_axis[0]);
  // If we couldn't allocate memory
  if (right == NULL)
    return;
  // Set the 'right by one char' vector
  VecNormalise(right);
  VecOp(right, font->_size * VecGet(font->_scale, 0), NULL, 0.0);
  // Declare a variable to memorize the normalized right vector
  VecFloat *rightnorm = VecClone(boundbox->_axis[0]);
  // If we couldn't allocate memory
  if (rightnorm == NULL)
    return;
  // Set the normalized right vector
  VecNormalise(rightnorm);
  // Declare a variable to memorize the 'right by one interspace' vector
  VecFloat *rightspace = VecClone(boundbox->_axis[0]);
  // If we couldn't allocate memory
  if (rightspace == NULL)
    return;
  // Set the 'right by one interspace' vector
  VecNormalise(rightspace);
  VecOp(rightspace, VecGet(font->_space, 0), NULL, 0.0);
  // Declare a variable to memorize the position of the current
  // character
  VecFloat *cursor = VecFloatCreate(2);
  // If we couldn't allocate memory
```

```
if (cursor == NULL)
  return;
// Set the start position of the cursor in the bounding box
// It's the upper left corner of the bounding box minus the height
// of one character
VecCopy(cursor, boundbox->_pos);
VecOp(cursor, 1.0, boundbox->_axis[1], 1.0);
VecOp(cursor, 1.0, down, 1.0);
// Get the number of character in the string
int nbChar = strlen((char*)s);
// Declare a variable to memorize the index of current line
int iLine = 1;
// Declare a variable to memorize length of the current line
float l = 0.0;
// for each character in the string
for (int iChar = 0; iChar < nbChar; ++iChar) {
  // If the character is a space
  if (s[iChar] == ' ') {
    // Increment the position in abciss by one character
    // plus interspace
    VecOp(cursor, 1.0, right, 1.0);
    VecOp(cursor, 1.0, rightspace, 1.0);
    // Increment length of current line
    l += VecNorm(right);
    l += VecNorm(rightspace);
  // Else, if the character is a tab
  } else if (s[iChar] == '\t') {
    // Set the position in abciss to the next multiple
    // of the tab parameter
    l = TGAFontGetNextPosByTab(font, l);
    VecCopy(cursor, boundbox->_pos);
    VecOp(cursor, 1.0, boundbox->_axis[1], 1.0);
    VecOp(cursor, 1.0, rightnorm, l);
    VecOp(cursor, 1.0, down, (float)iLine);
    VecOp(cursor, 1.0, downspace, (float)(iLine - 1));
  // Else, if the char is a line return
  } else if (s[iChar] == '\n') {
    // Increment index of line
    ++iLine;
    // Put the position to the start position of next line
    VecCopy(cursor, boundbox->_pos);
    VecOp(cursor, 1.0, boundbox->_axis[1], 1.0);
    VecOp(cursor, 1.0, down, (float)iLine);
    VecOp(cursor, 1.0, downspace, (float)(iLine - 1));
    // Reset length of current line
    l = 0.0;
  // Else, the character should be a printable character
  } else {
    // Print the character
    TGAPrintChar(tga, pen, font, s[iChar], cursor);
    // Increment the position in abciss by one character plus
    // interspace
    VecOp(cursor, 1.0, right, 1.0);
    VecOp(cursor, 1.0, rightspace, 1.0);
    // Increment length of current line
    l += VecNorm(right);
    l += VecNorm(rightspace);
  }
}
// Free memory
VecFree(&cursor);
VecFree(&right);
```

```
  VecFree(&down);
  VecFree(&rightspace);
  VecFree(&rightnorm);
  VecFree(&downspace);
  ShapoidFree(&boundbox);
}

// Print the char 'c' with its (bottom, left) position at 'pos'
// and (width, height) dimension 'dim' with font 'font'
void TGAPrintChar(TGA *tga, TGAPencil *pen, TGAFont *font,
  unsigned char c, VecFloat *pos) {
  // Check arguments
  if (tga == NULL || pen == NULL || font == NULL || pos == NULL)
    return;
  // Declare a vecfloat to scale the curve
  VecFloat *scale = VecGetOp(font->_scale, font->_size, NULL, 0.0);
  if (scale == NULL)
    return;
  // Set a pointer to the requested character's definition
  TGAChar *ch = font->_char + c;
  // Declare a variable to memorize the angle between the abciss
  // and the right direction of the font
  float theta = TGAFontGetAngleWithAbciss(font);
  // Clone the curve
  SCurve *clone = SCurveClone(ch->_curve);
  // If we could clone the curve
  if (clone != NULL) {
    // Scale the curve
    SCurveScale(clone, scale);
    // Rotate the curve
    SCurveRot2D(clone, theta);
    // Translate the curve
    SCurveTranslate(clone, pos);
    // Draw the curve
    TGADrawSCurve(tga, clone, pen);
    // Free memory
    SCurveFree(&clone);
  }
  VecFree(&scale);
}

// Get a white TGAPixel
TGAPixel* TGAGetWhitePixel(void) {
  // Allocate memory for the pixel
  TGAPixel *ret = (TGAPixel*)malloc(sizeof(TGAPixel));
  // If we could allocate memory
  if (ret != NULL) {
    // Set the pixel rgba values
    ret->_rgba[0] = ret->_rgba[1] = ret->_rgba[2] = ret->_rgba[3] = 255;
    // Set the read only property
    ret->_readOnly = false;
  }
  // Return the pixel
  return ret;
}

// Get a black TGAPixel
TGAPixel* TGAGetBlackPixel(void) {
  // Allocate memory for the pixel
  TGAPixel *ret = TGAGetWhitePixel();
  // If we could allocate memory
  if (ret != NULL) {
    // Set the pixel rgba values
    ret->_rgba[0] = ret->_rgba[1] = ret->_rgba[2] = 0;
    ret->_rgba[3] = 255;
  }
  // Return the pixel
  return ret;
}

// Get a transparent TGAPixel
TGAPixel* TGAGetTransparentPixel(void) {
  // Allocate memory for the pixel
  TGAPixel *ret = TGAGetWhitePixel();
  // If we could allocate memory
  if (ret != NULL) {
    // Set the pixel rgba values
    ret->_rgba[0] = ret->_rgba[1] = ret->_rgba[2] = 255;
    ret->_rgba[3] = 0;
  }
  // Return the pixel
  return ret;
}

// Free the memory used by tgapixel
void TGAPixelFree(TGAPixel **pixel) {
  // Check arguments
  if (pixel == NULL || *pixel == NULL)
    return;
  // Free the memory
  free(*pixel);
  *pixel = NULL;
}

// Return a new TGAPixel which is a blend of 'pixA' and 'pixB'
// newPix = (1 - blend) * pixA + blend * pixB
// Return NULL if arguments are invalid
TGAPixel* TGAPixelBlend(TGAPixel *pixA, TGAPixel *pixB, float blend) {
  // Check arguments
  if (pixA == NULL || pixB == NULL || blend < 0.0 || blend > 1.0)
    return NULL;
  // Get a transparent pixel
  TGAPixel *ret = TGAGetTransparentPixel();
  // If we could get a transparent pixel
  if (ret != NULL) {
    // For each rgba value
    for (int i = 4; i--;)
      // Calculate the blended value
      ret->_rgba[i] = (1.0 - blend) * pixA->_rgba[i] +
        blend * pixB->_rgba[i];
  }
  // Return the blend pixel
  return ret;
}

// Return a new TGAPixel which is the addition of 'ratio'
// (in [0.0,1.0]) * 'pixB' to 'pixA'
// Return NULL if arguments are invalid
TGAPixel* TGAPixelMix(TGAPixel *pixA, TGAPixel *pixB, float ratio) {
  // Check arguments
  if (pixA == NULL || pixB == NULL)
    return NULL;
  // Get a transparent pixel
  TGAPixel *ret = TGAGetTransparentPixel();
  // If we could get a transparent pixel
  if (ret != NULL) {
    // Declare a variable to memorize the opacity in [0,1]
    float opA = (float)(pixA->_rgba[3]) / 255.0;
    float opB = ratio * (float)(pixB->_rgba[3]) / 255.0;
    // If both pixel are not transparent
    if (opA + opB > 1.0 / 255.0) {
      // For each rgb value
      for (int i = 3; i--;) {
        // Calculate the mixed value
        float v = (opA * (float)(pixA->_rgba[i]) +
          opB * (float)(pixB->_rgba[i])) / (opA + opB);
        ret->_rgba[i] = (unsigned char)floor(v);
      }
      // Calculate mixed opacity (max of pixels opacity)
      if (opA < opB)
        ret->_rgba[3] = (unsigned char)floor(opB * 255.0);
      else
        ret->_rgba[3] = pixA->_rgba[3];
    }
  }
  // Return the mixed pixel
  return ret;
}

// Create a default TGAPencil with all color set to transparent
// solid mode, thickness = 1.0, tip as facoid, no antialias
// Return NULL if it couldn't allocate memory
TGAPencil* TGAGetPencil(void) {
  // Allocate memory for the new pencil
  TGAPencil *ret = (TGAPencil*)malloc(sizeof(TGAPencil));
  // If we could allocate memory
  if (ret != NULL) {
    // Get a transparent pixel
    TGAPixel *pixel = TGAGetTransparentPixel();
    // If we couldn't get the pixel
    if (pixel == NULL) {
      // Free memory
      free(ret);
      // Return NULL
      return NULL;
    }
    // Initialise all the color of the pencil to the transparent pixel
    for (int iCol = TGA_NBCOLORPENCIL; iCol--;)
      memcpy(ret->_colors + iCol, pixel, sizeof(TGAPixel));
    // Free memory used for the pixel
    TGAPixelFree(&pixel);
    // Set the default value of the pencil
    ret->_activeColor = 0;
    ret->_modeColor = tgaPenSolid;
    ret->_blendColor[0] = 0;
    ret->_blendColor[1] = 1;
    ret->_blend = 0.0;
    ret->_thickness = 1.0;
    ret->_antialias = false;
    ret->_tip = NULL;
    TGAPencilSetShapeSquare(ret);
  }
  // Return the new pencil
  return ret;
}
```

```
// Free the memory used by the TGAPencil 'pen'
void TGAPencilFree(TGAPencil **pencil) {
  // Check arguments
  if (pencil == NULL || *pencil == NULL)
    return;
  // Free memory used by the pencil
  free(*pencil);
  *pencil = NULL;
}

// Clone the TGAPencil 'pen'
// Return NULL if it couldn't clone
TGAPencil* TGAPencilClone(TGAPencil *pen) {
  // Check arguments
  if (pen == NULL)
    return NULL;
  // Allocate memory for the cloned pencil
  TGAPencil *ret = (TGAPencil*)malloc(sizeof(TGAPencil));
  // If we could allocate memory
  if (ret != NULL) {
    // Copy the pencil in the clone
    memcpy(ret, pen, sizeof(TGAPencil));
  }
  // Return the cloned pencil
  return ret;
}

// Create a TGAPencil with 1st color active and set to black
// Return NULL if it couldn't create
TGAPencil* TGAGetBlackPencil(void) {
  // Get a default pencil
  TGAPencil *ret = TGAGetPencil();
  // If we could get a pencil
  if (ret != NULL) {
    // Select the first color
    TGAPencilSelectColor(ret, 0);
    // Get a black pixel
    TGAPixel *pixel = TGAGetBlackPixel();
    // If we couldn't get the pixel
    if (pixel == NULL) {
      // Free memory
      TGAPencilFree(&ret);
      // Return NULL
      return NULL;
    }
    // Set the color to the black pixel
    TGAPencilSetColor(ret, pixel);
    // Free memory used by the pixel
    TGAPixelFree(&pixel);
  }
  // Return the new pencil
  return ret;
}

// Select the active color of TGAPencil 'pen' to 'iCol'
// Do nothing if arguments are invalid
void TGAPencilSelectColor(TGAPencil *pen, int iCol) {
  // Check arguments
  if (pen == NULL || iCol < 0 || iCol >= TGA_NBCOLORPENCIL)
    return;
  // Set the active color
  pen->_activeColor = iCol;
}

// Get the index of active color of TGAPencil 'pen'
// Return -1 if arguments are invalid
int TGAPencilGetColor(TGAPencil *pen) {
  // Check arguments
  if (pen == NULL)
    return -1;
  // Return the active color
  return pen->_activeColor;
}

// Get a TGAPixel equal to the active color of the TGAPencil 'pen'
// Return NULL if arguments are invalid
TGAPixel* TGAPencilGetPixel(TGAPencil *pen) {
  // Check arguments
  if (pen == NULL)
    return NULL;
  // Get a white pixel
  TGAPixel *ret = TGAGetWhitePixel();
  // If we couldn't get the pixel
  if (ret == NULL) {
    // Return nuLL
    return NULL;
  }
  // If the pen's color mode is tgaPenSolid
  if (pen->_modeColor == tgaPenSolid) {
    // Set the active color to the pixel
    memcpy(ret, pen->_colors + pen->_activeColor, sizeof(TGAPixel));
  // Else, if the pen's color mode is tgaPenBlend
  } else if (pen->_modeColor == tgaPenBlend) {
    // Calculate the current color
    for (int irgb = 0; irgb < 4; ++irgb)
      ret->_rgba[irgb] = (unsigned char)round((1.0 - pen->_blend) *
        (float)(pen->_colors[pen->_blendColor[0]]._rgba[irgb]) +
        pen->_blend *
        (float)(pen->_colors[pen->_blendColor[1]]._rgba[irgb]));
  }
  // Return the pixel
  return ret;
}

// Set the active color of TGAPencil 'pen' to TGAPixel 'col'
// Do nothing if arguments are invalid
void TGAPencilSetColor(TGAPencil *pen, TGAPixel *col) {
  // Check arguments
  if (pen == NULL || col == NULL)
    return;
  // Set the color values
  memcpy(pen->_colors + pen->_activeColor, col, sizeof(TGAPixel));
}

// Set the active color of TGAPencil 'pen' to 'rgba'
// Do nothing if arguments are invalid
void TGAPencilSetColRGBA(TGAPencil *pen, unsigned char *rgba) {
  // Check arguments
  if (pen == NULL || rgba == NULL)
    return;
  // Set the color values
  memcpy(&(pen->_colors[pen->_activeColor]._rgba), rgba,
    sizeof(unsigned char) * 4);
}
```

```
// Set the thickness of TGAPencil 'pen' to 'v'
// Equivalent to a scale of the shapoid of the tip
// Do nothing if arguments are invalid
void TGAPencilSetThickness(TGAPencil *pen, float v) {
  // Check arguments
  if (pen == NULL || v < 0.0)
    return;
  // If the pen tip is a shapoid
  if (pen->_tip != NULL) {
    // Declare a variable to memorize the scaling in each dimension
    VecFloat *s = VecFloatCreate(ShapoidGetDim(pen->_tip));
    // If we could allocate memory
    if (s != NULL) {
      // Set the scale values
      for (int i = VecDim(s); i--;)
        VecSet(s, i, v / pen->_thickness);
      // Grow the shapoid
      ShapoidGrow(pen->_tip, s);
      // Free memory
      VecFree(&s);
    }
  }
  // Set the thickness
  pen->_thickness = v;
}

// Set the antialias of the TGAPencil 'pen' to 'v'
// Do nothing if arguments are invalid
void TGAPencilSetAntialias(TGAPencil *pen, bool v) {
  // Check arguments
  if (pen == NULL || (v != true && v != false))
    return;
  // Setthe antialias
  pen->_antialias = v;
}

// Set the blend value 'v' of the TGAPencil 'pen'
// Do nothing if arguments are invalid
void TGAPencilSetBlend(TGAPencil *pen, float v) {
  // Check arguments
  if (pen == NULL || v < 0.0 || v > 1.0)
    return;
  pen->_blend = v;
}

// Set the shape of the TGAPencil 'pen' to 'tgaPenShapoid' and
// set the tip of the pen to a new facoid centered on the origin
// and scaled to the pen thickness
// Do nothing if arguments are invalid
void TGAPencilSetShapeSquare(TGAPencil *pen) {
  // Check arguments
  if (pen == NULL)
    return;
  // Declare a VecFloat used for Shapoid creation
  VecFloat *v = VecFloatCreate(2);
  // If we couldn't allocate memory
  if (v == NULL) {
    // Stop here
    return;
  }
  // Set the shape
  pen->_shape = tgaPenShapoid;
  // Free the eventual actual shapoid
  ShapoidFree(&(pen->_tip));
  // If there was a shapoid allocated for the pen tip
  if (pen->_tip != NULL)
    // Free this shapoid
    ShapoidFree(&(pen->_tip));
  // Create a new Facoid
  pen->_tip = FacoidCreate(2);
  // If we could allocate memory
  if (pen->_tip != NULL) {
    // Scale the Shapoid
    for (int i = 2; i--;)
      VecSet(v, i, pen->_thickness);
    ShapoidScale(pen->_tip, v);
    // Center the Shapoid on origin
    for (int i = 2; i--;)
      VecSet(v, i, -0.5 * pen->_thickness);
    ShapoidTranslate(pen->_tip, v);
  // Else, if we couldn't allocate memory
  } else {
    // Reset the pen shape to pixel for safety
    pen->_shape = tgaPenPixel;
  }
  // Free memory
  VecFree(&v);
}

// Set the shape of the TGAPencil 'pen' to 'tgaPenShapoid' and
// set the tip of the pen to a new ellipsoid scaled to the pen thickness
// Do nothing if arguments are invalid
void TGAPencilSetShapeRound(TGAPencil *pen) {
  // Check arguments
  if (pen == NULL)
    return;
  // Declare a VecFloat used for Shapoid creation
  VecFloat *v = VecFloatCreate(2);
  // If we couldn't allocate memory
  if (v == NULL) {
    // Stop here
    return;
  }
  // Set the shape
  pen->_shape = tgaPenShapoid;
  // If there was a shapoid allocated for the pen tip
  if (pen->_tip != NULL)
    // Free this shapoid
    ShapoidFree(&(pen->_tip));
  // Free the eventual actual shapoid
  ShapoidFree(&(pen->_tip));
  // Create a new Facoid
  pen->_tip = SpheroidCreate(2);
  // If we could allocate memory
  if (pen->_tip != NULL) {
    // Scale the Shapoid
    for (int i = 2; i--;)
      VecSet(v, i, pen->_thickness);
    ShapoidScale(pen->_tip, v);
  // Else, if we couldn't allocate memory
  } else {
    // Reset the pen shape to pixel for safety
    pen->_shape = tgaPenPixel;
  }
  // Free memory
  VecFree(&v);
}

// Set the shape of the TGAPencil 'pen' to 'tgaPenShapoid' and
// set the tip of the pen to a clone of the Shapoid 'shape'
// 'shape' is considered to be centered and given at a thickness
// of 1.0 before rescaling to 'pen' thickness
// Do nothing if arguments are invalid
void TGAPencilSetShapeShapoid(TGAPencil *pen, Shapoid *shape) {
  // Check arguments
  if (pen == NULL || shape == NULL)
    return;
  // Declare a VecFloat used for Shapoid creation
  VecFloat *v = VecFloatCreate(2);
  // If we couldn't allocate memory
  if (v == NULL) {
    // Stop here
    return;
  }
  // Set the shape
  pen->_shape = tgaPenShapoid;
  // If there was a shapoid allocated for the pen tip
  if (pen->_tip != NULL)
    // Free this shapoid
    ShapoidFree(&(pen->_tip));
  // Create the new pen tip
  pen->_tip = ShapoidClone(shape);
  // If we could allocate memory
  if (pen->_tip != NULL) {
    // Grow the Shapoid
    for (int i = 2; i--;)
      VecSet(v, i, pen->_thickness);
    ShapoidGrow(pen->_tip, v);
  // Else, if we couldn't allocate memory
  } else {
    // Reset the pen shape to pixel for safety
    pen->_shape = tgaPenPixel;
  }
  // Free memory
  VecFree(&v);
}

// Set the shape of the TGAPencil 'pen' to 'tgaPenPixel'
// Do nothing if arguments are invalid
void TGAPencilSetShapePixel(TGAPencil *pen) {
  // Check arguments
  if (pen == NULL)
    return;
  // Set the shape
  pen->_shape = tgaPenPixel;
  // If there was a shapoid allocated for the pen tip
  if (pen->_tip != NULL)
    // Free this shapoid
    ShapoidFree(&(pen->_tip));
}


// Set the mode of the TGAPencil 'pen' to 'tgaPenSolid'
// Do nothing if arguments are invalid
void TGAPencilSetModeColorSolid(TGAPencil *pen) {
  // Check arguments
  if (pen == NULL)
    return;
  // Set the color mode
  pen->_modeColor = tgaPenSolid;
}

// Set the mode of the TGAPencil 'pen' to 'tgaPenBlend'
// Blend is done from 'fromCol' to 'toCol'
// Do nothing if arguments are invalid
void TGAPencilSetModeColorBlend(TGAPencil *pen, int fromCol, int toCol) {
  // Check arguments
  if (pen == NULL || fromCol < 0 || fromCol >= TGA_NBCOLORPENCIL ||
    toCol < 0 || toCol >= TGA_NBCOLORPENCIL)
    return;
  // Set the color mode
  pen->_modeColor = tgaPenBlend;
  pen->_blendColor[0] = fromCol;
  pen->_blendColor[1] = toCol;
}

// Function to decode rgba values when loading a TGA file
// Do nothing if arguments are invalid
void MergeBytes(TGAPixel *pixel, unsigned char *p, int bytes) {
  // Check arguments
  if (pixel == NULL || p == NULL)
    return;
  // Merge bytes
  if (bytes == 4) {
    pixel->_rgba[0] = p[2];
    pixel->_rgba[1] = p[1];
    pixel->_rgba[2] = p[0];
    pixel->_rgba[3] = p[3];
  } else if (bytes == 3) {
    pixel->_rgba[0] = p[2];
    pixel->_rgba[1] = p[1];
    pixel->_rgba[2] = p[0];
    pixel->_rgba[3] = 255;
  } else if (bytes == 2) {
    pixel->_rgba[0] = (p[1] & 0x7c) << 1;
    pixel->_rgba[1] = ((p[1] & 0x03) << 6) | ((p[0] & 0xe0) >> 2);
    pixel->_rgba[2] = (p[0] & 0x1f) << 3;
    pixel->_rgba[3] = (p[1] & 0x80);
  }
}

// Get the average color of the whole image
// Return a TGAPixel set to the avergae color, or NULL if the arguments
// are invalid
TGAPixel *TGAGetAverageColor(TGA *tga) {
  // Check arguments
  if (tga == NULL)
    return NULL;
  // Declare the returned TGAPixel
  TGAPixel *pixel = TGAGetWhitePixel();
  // Declare a variable to calculate the average value
  float rgba[4] = {0.0};
  // Calculate the average color
  VecShort *pos = VecShortCreate(2);
  for (VecSet(pos, 0, 0); VecGet(pos, 0) < tga->_header->_width;
    VecSet(pos, 0, VecGet(pos, 0) + 1)) {
    for (VecSet(pos, 1, 0); VecGet(pos, 1) < tga->_header->_width;
      VecSet(pos, 1, VecGet(pos, 1) + 1)) {
      TGAPixel *pix = TGAGetPix(tga, pos);
      if (pix != NULL) {
        for (int iRGB = 0; iRGB < 4; ++iRGB)
          rgba[iRGB] += (float)(pix->_rgba[iRGB]);
      }
    }
  }
  VecFree(&pos);
  for (int iRGB = 0; iRGB < 4; ++iRGB)
    rgba[iRGB] /=
      (float)(tga->_header->_width) * (float)(tga->_header->_height);
  // Set the result pixel value
  for (int iRGB = 0; iRGB < 4; ++iRGB)
    pixel->_rgba[iRGB] = (char)floor(rgba[iRGB]);
  // Return the result pixel
  return pixel;
}

// Set the read only flag of a TGAPixel
// Do nothing if arguments are invalid
void TGAPixelSetReadOnly(TGAPixel *pix, bool v) {
  // Check arguments
  if (pix == NULL)
    return;
  pix->_readOnly = v;
}

// Set the read only flag of all the TGAPixel of a TGA
// Do nothing if arguments are invalid
void TGAPixelSetAllReadOnly(TGA *tga, bool v) {
  // Check arguments
  if (tga == NULL)
    return;
  VecShort *pos = VecShortCreate(2);
  for (VecSet(pos, 0, 0); VecGet(pos, 0) < tga->_header->_width;
    VecSet(pos, 0, VecGet(pos, 0) + 1)) {
    for (VecSet(pos, 1, 0); VecGet(pos, 1) < tga->_header->_width;
      VecSet(pos, 1, VecGet(pos, 1) + 1)) {
      TGAPixelSetReadOnly(TGAGetPix(tga, pos), v);
    }
  }
  VecFree(&pos);
}

// Get the read only flag of a TGAPixel
// Return true if arguments are invalid
bool TGAPixelIsReadOnly(TGAPixel *pix) {
  // Check arguments
  if (pix == NULL)
    return true;
  return pix->_readOnly;
}

// Create a TGALayer of width dim[0] and height dim[1] and background
// color equal to 'pixel'
// If 'pixel' is NULL rgba(0,0,0,0) is used
// Return NULL in case of invalid arguments or memory allocation
// failure
TGALayer* TGALayerCreate(VecShort *dim, TGAPixel *pixel) {
  // Check arguments
  if (dim == NULL)
```

```
    return NULL;
  // Allocate memory
  TGALayer *ret = (TGALayer*)malloc(sizeof(TGALayer));
  // If we couldn't allocate memory
  if (ret == NULL)
    // Return NULL
    return NULL;
  // Set the pointers to NULL
  ret->_dim = NULL;
  ret->_pixels = NULL;
  // Copy the dimensions
  ret->_dim = VecClone(dim);
  // If we couldn't allocate memory
  if (ret->_dim == NULL) {
    // Free the memory
    free(ret);
    // Return NULL
    return NULL;
  }
  // Allocate memory for the pixels
  ret->_pixels = (TGAPixel*)malloc(VecGet(dim, 0) * VecGet(dim, 1) *
    sizeof(TGAPixel));
  // If we couldn't allocate memory
  if (ret->_pixels == NULL) {
    // Free the memory
    VecFree(&(ret->_dim));
    free(ret);
    // Return NULL
    return NULL;
  }
  // Set a pointer to the pixels
  TGAPixel *p = ret->_pixels;
  // For each pixel
  for (int i = 0; i < VecGet(dim, 0) * VecGet(dim, 1); ++i) {
    // For each value RGBA
    for (int irgb = 0; irgb < 4; ++irgb)
      // Initialize the value
      if (pixel != NULL)
        p[i]._rgba[irgb] = pixel->_rgba[irgb];
      else
        p[i]._rgba[irgb] = 0;
    // Initialize in read-write
    p[i]._readOnly = false;
  }
  // Return the created TGALayer
  return ret;
}

// Clone a TGALayer
// Return NULL in case of failure
TGALayer* TGALayerClone(TGALayer *that) {
  // Check arguments
  if (that == NULL)
    return NULL;
  // Allocate memory for the cloned TGALayer
  TGALayer *ret = (TGALayer*)malloc(sizeof(TGALayer));
  // If we could allocate memory
  if (ret != NULL) {
    // Clone the dimension
    ret->_dim = VecClone(that->_dim);
    // If we couldn't allocate memory
    if (ret->_dim == NULL) {
      // Free memory
      free(ret);
      // Return NULL
      return NULL;
    }
    // Allocate memory for the pixels
    ret->_pixels = (TGAPixel*)malloc(VecGet(that->_dim, 0) *
      VecGet(that->_dim, 1) * sizeof(TGAPixel));
    // If we couldn't allocate memory
    if (ret->_pixels == NULL) {
      // Free memory
      VecFree(&(ret->_dim));
      free(ret);
      // Return NULL
      return NULL;
    }
    // Copy the pixels
    memcpy(ret->_pixels, that->_pixels, VecGet(that->_dim, 0) *
      VecGet(that->_dim, 1) * sizeof(TGAPixel));
  }
  // Return the cloned TGA
  return ret;
}

// Free the memory used by the TGALayer
void TGALayerFree(TGALayer **that) {
  // Check arguments
  if (that == NULL || *that == NULL)
    return;
  // Free the memory
  VecFree(&((*that)->_dim));
  TGAPixelFree(&((*that)->_pixels));
  free(*that);
  *that = NULL;
}

// Set the current layer to the 'iLayer'-th layer
// Do nothing if arguments are invalid
void TGASetCurLayer(TGA *that, int iLayer) {
  // Check arguments
  if (that == NULL || iLayer < 0 || iLayer >= that->_layers->_nbElem)
    return;
  // Set the current layer
  that->_curLayerIndex = iLayer;
  that->_curLayer = GSetGet(that->_layers, iLayer);
}

// Add a layer above the current one
// Do nothing if the arguments are invalid
void TGAAddLayer(TGA *that) {
  // Check arguments
  if (that == NULL)
    return;
  // Create the new layer
  TGALayer *layer = TGALayerCreate(that->_curLayer->_dim, NULL);
  // If we could create the layer
  if (layer != NULL) {
    // Add it above the current layer
    GSetInsert(that->_layers, layer, that->_curLayerIndex + 1);
  }
}
```

```
// Blend layers 'that' and 'tho', the result is stored into 'that'
// 'tho' is considered to above 'that'
// If VecShort 'bound' is not null only pixels inside the box
// (bound[0],bound[1])-(bound[2],bound[3]) (included) are blended
// 'that' and 'tho' must have same dimension
// Do nothing if arguments are invalid
void TGALayerBlend(TGALayer *that, TGALayer *tho, VecShort *bound) {
  // Check arguments
  if (that == NULL || tho == NULL || VecIsEqual(that->_dim, tho->_dim) == false)
    return;
  // Declare a flag to memorize if we have created the bounds locally
  bool flagBound = false;
  // If there is no bound given
  if (bound == NULL) {
    // Set the flag
    flagBound = true;
    // Create a local bound equal to the dimension of the layer
    bound = VecShortCreate(4);
    VecSet(bound, 0, 0);
    VecSet(bound, 1, 0);
    VecSet(bound, 2, VecGet(that->_dim, 0) - 1);
    VecSet(bound, 3, VecGet(that->_dim, 1) - 1);
  }
  // Create a vector for looping on the pixels
  VecShort *pos = VecShortCreate(2);
  // If we couldn't allocate memory or the bounding box is invalid
  if (bound == NULL || pos == NULL || VecGet(bound, 0) > VecGet(bound, 2) || VecGet(bound, 1) > VecGet(bound, 3)) {
    VecFree(&pos);
    if (flagBound == true)
      VecFree(&bound);
  }
  // Loop on the pixels
  for (VecSet(pos, 0, VecGet(bound, 0));
    VecGet(pos, 0) <= VecGet(bound, 2);
    VecSet(pos, 0, VecGet(pos, 0) + 1)) {
    for (VecSet(pos, 1, VecGet(bound, 1));
      VecGet(pos, 1) <= VecGet(bound, 3);
      VecSet(pos, 1, VecGet(pos, 1) + 1)) {
      // Get the pixel in each layer
      TGAPixel *pixThat = TGALayerGetPix(that, pos);
      TGAPixel *pixTho = TGALayerGetPix(tho, pos);
      // If both pixel exists and the one in 'that' is not readonly
      if (pixThat != NULL && pixTho != NULL &&
        TGAPixelIsReadOnly(pixThat) == false) {
        TGAPixel *pixBlend = TGAPixelBlend(pixThat, pixTho,
          (float)(pixTho->_rgba[3]) / 255.0);
        // If we could blend the pixel
        if (pixBlend != NULL) {
          // Correct the opacity of the blended pixel
          if (255.0 - (float)(pixThat->_rgba[3]) >
            (float)(pixTho->_rgba[3]))
            pixBlend->_rgba[3] = pixThat->_rgba[3] + pixTho->_rgba[3];
          else
            pixBlend->_rgba[3] = 255.0;
          // Copy the resulting pixel in 'that'
          TGALayerSetPix(that, pos, pixBlend);
        }
        // Free memory
        TGAPixelFree(&pixBlend);
      }
    }
  }
  // Free memory
  VecFree(&pos);
  if (flagBound == true)
    VecFree(&bound);
}

// Get a pointer to the pixel at coord (x,y) = (pos[0],pos[1])
// in the layer 'that'
// Return NULL in case of invalid arguments
TGAPixel* TGALayerGetPix(TGALayer *that, VecShort *pos) {
  // Check arguments
  if (that == NULL || pos == NULL)
    return NULL;
  if (VecGet(pos, 0) < 0 ||
    VecGet(pos, 0) >= VecGet(that->_dim, 0) ||
    VecGet(pos, 1) < 0 ||
    VecGet(pos, 1) >= VecGet(that->_dim, 1))
    return NULL;
  // Set a pointer to the pixels
  TGAPixel *p = that->_pixels;
  // Calculate the index of the requested pixel
  int i = VecGet(pos, 1) * VecGet(that->_dim, 0) + VecGet(pos, 0);
  // Return a pointer toward the requested pixel
  return &(p[i]);
}

// Set the color of one pixel at coord (x,y) = (pos[0],pos[1]) to 'pix'
// in the layer 'that'
// Do nothing in case of invalid arguments
void TGALayerSetPix(TGALayer *that, VecShort *pos, TGAPixel *pix) {
  // Check arguments
  if (that == NULL || pos == NULL || pix == NULL)
    return;
  // Set a pointer to the pixels
  TGAPixel *p = TGALayerGetPix(that, pos);
  // If the pixel is not null and not in read only mode
  if (p != NULL && TGAPixelIsReadOnly(p) == false)
    // Set the value of the pixel
    memcpy(p, pix, sizeof(TGAPixel));
}

// Add the BCurve 'curve' (must be of dimension 2 and order > 0)
// in 'layer'
// do nothing if arguments are invalid
void TGALayerAddCurve(TGALayer *layer, BCurve *curve, TGAPencil *pen) {
  // Check arguments
  if (layer == NULL || curve == NULL || pen == NULL ||
    BCurveOrder(curve) < 1)
    return;
  // GetThe approximate length of the curve
  float l = BCurveApproxLen(curve);
  // Declare a variable to memorize the step of the parameter of
  // the BCurve
  float dt = 1.0 / l;
  // Declare the parameter of the curve
  float t = 0.0;
  // Declare the parameter value of last drawn pixel
  float lastT = t;
  // Declare a variable to memorize the position on the curve
  VecFloat *pos = VecClone(curve->_ctrl[0]);
  // Declare a variable to memorize the last pixel stroke to avoid
  // stroking several time the same pixel as dt is underestimated
```

```
VecFloat *prevPos = VecClone(pos);
if (prevPos == NULL)
  return;
// Set the blend value of the pencil to calculate the pencil
// current color
TGAPencilSetBlend(pen, 0.0);
// Stroke the first pixel
TGALayerStrokePix(layer, curve->_ctrl[0], pen);
// While we haven't reached the end of the curve
while (t < 1.0 + dt) {
  // Calculate the current position on the curve
  VecFree(&pos);
  pos = BCurveGet(curve, t);
  // Declare a variable to memorize the pixel distance to the previous
  // drawn pixel
  float pixelDist = VecPixelDist(prevPos, pos);
  // Declare a flag to memorize if we need to draw the current pixel
  bool flagDraw = false;
  // If we are still on the previous pixel
  if (pixelDist < 0.5) {
    // Update the position of the last stroke pixel
    VecCopy(prevPos, pos);
    // Update the parameter value of last drawn pixel
    lastT = t;
    // Move along the curve by dt
    t += dt;
  // Else, we have moved to a different pixel
  } else {
    // If we are on a side pixel
    if (pixelDist < 1.5) {
      // We are good to draw
      flagDraw = true;
    // Else, we are at a pixel distance of more than 2.0
    // It means we jumped over, or moved in diagonal
    } else {
      // If we are on a diagonal pixel
      if (abs((int)floor(VecGet(pos, 0)) -
        (int)floor(VecGet(prevPos, 0))) <= 1 &&
        abs((int)floor(VecGet(pos, 1)) -
        (int)floor(VecGet(prevPos, 1))) <= 1) {
        // We are good to draw
        flagDraw = true;
      // Else, we have jump over a pixel
      } else {
        // Move back to cancel the jump over pixel
        t -= (t - lastT) * 0.9;
      }
    }
  }
  // If we are good to draw
  if (flagDraw == true) {
    // Set the blend value of the pencil to calculate the pencil
    // current color
    TGAPencilSetBlend(pen, t);
    // Stroke the pixel
    TGALayerStrokePix(layer, pos, pen);
    // Update the position of the last stroke pixel
    VecCopy(prevPos, pos);
    // Update the parameter value of last drawn pixel
    lastT = t;
    // Move along the curve by dt
    t += dt;
    }
  }
  // If the last pixel hasn’t been stroke
  if (VecPixelDist(prevPos, curve->_ctrl[curve->_order]) > 0.5)
    // Stroke the last pixel
    TGALayerStrokePix(layer, curve->_ctrl[curve->_order], pen);
  // Free memory
  VecFree(&pos);
  VecFree(&prevPos);
}

// Draw one stroke at ’pos’ with ’pen’
// in layer ’that’
// Do nothing in case of invalid arguments
void TGALayerStrokePix(TGALayer *that, VecFloat *pos, TGAPencil *pen) {
  // Check arguments
  if (that == NULL || pos == NULL || pen == NULL) return;
  // If the shape of the pencil is pixel
  if (pen->_shape == tgaPenPixel) {
    TGALayerStrokePixOnePixel(that, pos, pen);
  // Else, if the shape of the pencil is shapoid
  } else if (pen->_shape == tgaPenShapoid) {
    TGALayerStrokePixShapoid(that, pos, pen);
  }
}

// Return true if ’pos’ is inside ’that’
// Return false else, or if arguments are invalid
bool TGALayerIsPosInside(TGALayer *that, VecShort *pos) {
  // Check arguments
  if (that == NULL || pos == NULL || VecDim(pos) < 2)
    return false;
  // If the position is in the tga
  if (VecGet(pos, 0) >= 0 && VecGet(pos, 0) < VecGet(that->_dim, 0) &&
    VecGet(pos, 1) >= 0 && VecGet(pos, 1) < VecGet(that->_dim, 1))
    return true;
  // Else, the position is not in the tga
  else
    return false;
}

// Erase the content of the layer ’that’
// (set all pixel to rgba(0,0,0,0) and readonly to false)
// Do nothing in case of invalid argument
void TGALayerClean(TGALayer *that) {
  // Check arguments
  if (that == NULL)
    return;
  // Set a pointer to the pixels
  TGAPixel *p = that->_pixels;
  // For each pixel
  for (int i = 0;
    i < VecGet(that->_dim, 0) * VecGet(that->_dim, 1); ++i) {
    // For each value RGBA
    for (int irgb = 0; irgb < 4; ++irgb)
      // Set the value
      p[i]._rgba[irgb] = 0;
    // Set read-write
    p[i]._readOnly = false;
  }
}
```

2.2 tgafont.c

```
// *************** TGAFONT.C ***************

// ================ Functions declaration ====================

// Create the curves of each characters for the default font
void TGAFontCreateDefault(TGAFont *font);

// Get the next position form 'p' incremented by one tabulation
// of 'font'
float TGAFontGetNextPosByTab(TGAFont *font, float p);

// ================ Functions implementation ==================

// Create a TGAFont with set of character 'font',
// _fontSize = 18.0, _space[0] = _space[1] = 3.0,
// _scale[0] = 0.5, _scale[1] = 1.0, _anchor = tgaFrontAnchorTopLeft
// _dir = <1.0, 0.0>, _tabSize = _fontSize
// Return NULL if it couldn't create
TGAFont* TGAFontCreate(tgaFont font) {
  // Allocate memory
  TGAFont *ret = (TGAFont*)malloc(sizeof(TGAFont));
  // If we could allocate memory
  if (ret != NULL) {
    // Set the default size
    ret->_size = 18.0;
    // Set the default tab size
    ret->_tabSize = ret->_size;
    // Set the default space
    ret->_space = VecFloatCreate(2);
    if (ret->_space == NULL) {
      free(ret);
      return NULL;
    }
    VecSet(ret->_space, 0, 3.0);
    VecSet(ret->_space, 1, 3.0);
    // Set the default scale
    ret->_scale = VecFloatCreate(2);
    if (ret->_scale == NULL) {
      VecFree(&(ret->_space));
      free(ret);
      return NULL;
    }
    VecSet(ret->_scale, 0, 1.0);
    VecSet(ret->_scale, 1, 1.0);
    // Set the default anchor
    ret->_anchor = tgaFontAnchorTopLeft;
    // Set the default orientation
    ret->_right = VecFloatCreate(2);
    if (ret->_right == NULL) {
      VecFree(&(ret->_space));
      VecFree(&(ret->_scale));
      free(ret);
      return NULL;
    }
    VecSet(ret->_right, 0, 1.0);
    VecSet(ret->_right, 1, 0.0);
    // For each character
    for (int iChar = 256; iChar--;) {
      // By default set this character definition as empty (no curves)
      ret->_char[iChar]._curve = SCurveCreate(2);
      if (ret->_char[iChar]._curve == NULL) {
        VecFree(&(ret->_space));
        VecFree(&(ret->_scale));
        VecFree(&(ret->_right));
        free(ret);
        return NULL;
      }
    }
    // If the requested font is the default one
    if (font == tgaFontDefault)
      // Create the default font characters' curves
      TGAFontCreateDefault(ret);
  }
  // Return the created font
  return ret;
}

// Free memory used by TGAFont
// Do nothing if arguments are invalid
void TGAFreeFont(TGAFont **font) {
  // If the argument are invalid, stop here
  if (font == NULL || *font == NULL)
    return;
  // Free the memory
  for (int iChar = 256; iChar--;)
    SCurveFree(&((*font)->_char[iChar]._curve));
  VecFree(&((*font)->_scale));
  VecFree(&((*font)->_space));
  VecFree(&((*font)->_right));
  free(*font);
  *font = NULL;
}

// Set the font size of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetSize(TGAFont *font, float v) {
  if (font == NULL || v <= 0.0)
    return;
  font->_size = v;
}

// Set the font tab size of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetTabSize(TGAFont *font, float v) {
  if (font == NULL || v <= 0.0)
    return;
  font->_tabSize = v;
}

// Set the font scale of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetScale(TGAFont *font, VecFloat *v) {
  // If the argument are invalid, stop here
  if (font == NULL || v == NULL)
    return;
  // Set the scale
  VecCopy(font->_scale, v);
}

// Set the font spacing of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetSpace(TGAFont *font, VecFloat *v) {
  // If the argument are invalid, stop here
  if (font == NULL || v == NULL)
    return;
  // Set the space
  VecCopy(font->_space, v);
}

// Set the anchor of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetAnchor(TGAFont *font, tgaFontAnchor v) {
  // If the argument are invalid, stop here
  if (font == NULL)
    return;
  // Set the anchor
  font->_anchor = v;
}

// Set the right direction of TGAFont 'font' to 'v'
// Do nothing if arguments are invalid
void TGAFontSetRight(TGAFont *font, VecFloat *v) {
  // If the argument are invalid, stop here
  if (font == NULL || v == NULL)
    return;
  // Set the right direction
  VecCopy(font->_right, v);
  // Ensure its normalized
  VecNormalise(font->_right);
}

// Get the next position form 'p' incremented by one tabulation
// of 'font'
float TGAFontGetNextPosByTab(TGAFont *font, float p) {
  return (floor(p / font->_tabSize) + 1.0) * font->_tabSize;
}

// Get the angle of the right vector of the font with the abciss
// Return 0.0 if the arguments are invalid or memory allocation failed
float TGAFontGetAngleWithAbciss(TGAFont *font) {
  if (font == NULL)
    return 0.0;
  VecFloat *abciss = VecFloatCreate(2);
  if (abciss == NULL)
    return 0.0;
  VecSet(abciss, 0, 1.0); VecSet(abciss, 1, 0.0);
  float theta = VecAngleTo2D(abciss, font->_right);
  VecFree(&abciss);
  return theta;
}

// Get the bounding box as a facoid of order 2 and dim 2 in pixels
// of the block of text representing string 's' printed with 'font'
// Return NULL if arguments are invalid
Shapoid* TGAFontGetStringBound(TGAFont *font, unsigned char *s) {
  // Check arguments
  if (font == NULL)
    return NULL;
  // Declare a variable to memorize the height of lines and the max
  // width of a line in pixels
  VecFloat *dim = VecFloatCreate(2);
  // If we couldn't allocate memory
  if (dim == NULL)
    return NULL;
```

```
// Declare a variable for the result
Shapoid *res = FacoidCreate(2);
// If we couldn't allocate memory
if (res == NULL)
  return NULL;
// Declare a variable to memorize the total heights of the lines
float height = 0.0;
// If the string is not empty
if (s != NULL) {
  // Initialise the dimensions
  VecSet(dim, 0, 0.0);
  VecSet(dim, 1, font->_size * VecGet(font->_scale, 1));
  // Declare a variable to memorize the length of the current line
  float l = 0.0;
  // Declare a variable to memorize if we are at the beginning
  // of the line
  bool flagStart = true;
  // For each character
  int nb = strlen((char*)s);
  for (int iChar = 0; iChar < nb; ++iChar) {
    // If this character is a line return
    if (s[iChar] == '\n') {
      // Increment height
      float h = font->_size * VecGet(font->_scale, 1) +
        VecGet(font->_space, 1);
      height += h;
      VecSet(dim, 1, VecGet(dim, 1) + h);
      // Reset the length of line
      l = 0.0;
      // Reset the flag
      flagStart = true;
    // Else, if this character is a tabulation
    } else if (s[iChar] == '\t') {
      // Increment length to the next tab
      l = TGAFontGetNextPosByTab(font, l);
      // If the current line is longer than the longest one
      if (VecGet(dim, 0) < l)
        // Update the length of the
        VecSet(dim, 0, l);
    // Else, for others character
    } else {
      // If it's not the first char
      if (flagStart == false)
        // Add the space between character
        l += VecGet(font->_space, 0);
      // Update the flag of beginning of line
      flagStart = false;
      // Increment the length of the current line
      l += font->_size * VecGet(font->_scale, 0);
      // If the current line is longer than the longest one
      if (VecGet(dim, 0) < l)
        // Update the length
        VecSet(dim, 0, l);
    }
  }
}
// Scale the Facoid
ShapoidScale(res, dim);
// Reposition the Facoid according to the anchor
switch (font->_anchor) {
  case tgaFontAnchorTopLeft:
    VecSet(res->_pos, 1, VecGet(res->_pos, 1) - VecGet(dim, 1));
```

```
      break;
    case tgaFontAnchorTopCenter:
      VecSet(res->_pos, 1, VecGet(res->_pos, 1) - VecGet(dim, 1));
      VecSet(res->_pos, 0, -0.5 * VecGet(dim, 0));
      break;
    case tgaFontAnchorTopRight:
      VecSet(res->_pos, 1, VecGet(res->_pos, 1) - VecGet(dim, 1));
      VecSet(res->_pos, 0, -1.0 * VecGet(dim, 0));
      break;
    case tgaFontAnchorCenterLeft:
      VecSet(res->_pos, 1,
        VecGet(res->_pos, 1) - 0.5 * VecGet(dim, 1));
      break;
    case tgaFontAnchorCenterCenter:
      VecSet(res->_pos, 1,
        VecGet(res->_pos, 1) - 0.5 * VecGet(dim, 1));
      VecSet(res->_pos, 0, -0.5 * VecGet(dim, 0));
      break;
    case tgaFontAnchorCenterRight:
      VecSet(res->_pos, 1,
        VecGet(res->_pos, 1) - 0.5 * VecGet(dim, 1));
      VecSet(res->_pos, 0, -1.0 * VecGet(dim, 0));
      break;
    case tgaFontAnchorBottomLeft:
      break;
    case tgaFontAnchorBottomCenter:
      VecSet(res->_pos, 0, -0.5 * VecGet(dim, 0));
      break;
    case tgaFontAnchorBottomRight:
      VecSet(res->_pos, 0, -1.0 * VecGet(dim, 0));
      break;
    default:
      break;
  }
  // Rotate the Facoid
  float theta = TGAFontGetAngleWithAbciss(font);
  ShapoidRotate2D(res, theta);
  // The rotation must also be applied to the position which may be
  // not at the origin
  VecRot2D(res->_pos, theta);
  // Free memory
  VecFloatFree(&dim);
  // Return the result
  return res;
}

// Function to initialize the curves of one char
void TGAFontInitChar(TGAChar *ch, int nbCurve, float *c) {
  BCurve *curve = BCurveCreate(3, 2);
  if (curve != NULL) {
    for (int iCurve = 0; iCurve < nbCurve; ++iCurve) {
      for (int iCtrl = 4; iCtrl--;)
        for (int dim = 2; dim--;)
          VecSet(curve->_ctrl[iCtrl], dim,
            c[iCurve * 8 + iCtrl * 2 + dim]);
      SCurveAdd(ch->_curve, curve);
    }
  }
  BCurveFree(&curve);
}

// Create the curves of each characters for the default font
```

```
void TGAFontCreateDefault(TGAFont *font) {
  TGAChar *ch = NULL;
  ch = font->_char + 'A';
  TGAFontInitChar(ch, 3,
    (float[]){
        0.0,0.0,0.0,0.18,0.32,1.0,0.5,1.0,
        0.5,1.0,0.68,1.0,1.0,0.18,1.0,0.0,
        0.15,0.5,0.15,0.5,0.85,0.5,0.85,0.5
    });
  ch = font->_char + 'B';
  TGAFontInitChar(ch, 4,
    (float[]){
        0.00,0.00,0.00,0.00,0.00,1.00,0.00,1.00,
        0.00,1.00,0.77,1.00,0.77,0.58,0.00,0.59,
        0.00,0.59,0.50,0.60,1.01,0.50,1.00,0.26,
        1.00,0.26,1.00,0.00,0.50,0.00,0.00,0.00
    });
  ch = font->_char + 'C';
  TGAFontInitChar(ch, 4,
    (float[]){
        1.00,0.67,1.00,0.82,1.00,1.00,0.50,1.00,
        0.50,1.00,0.00,1.00,0.00,0.81,0.00,0.50,
        0.00,0.50,0.00,0.18,0.00,0.00,0.50,0.00,
        0.50,0.00,1.00,0.00,1.00,0.17,1.00,0.33
    });
  ch = font->_char + 'D';
  TGAFontInitChar(ch, 5,
    (float[]){
        0.00,1.00,0.00,1.00,0.00,0.00,0.00,0.00,
        0.00,0.00,1.00,0.00,1.00,0.00,1.00,0.50,
        1.00,0.50,1.00,1.00,0.50,1.00,0.00,1.00,
        0.00,1.00,-0.11,1.00,0.00,0.00,0.00,0.00,
        0.00,0.00,0.00,0.00,0.00,0.00,0.00,0.00
    });
  ch = font->_char + 'E';
  TGAFontInitChar(ch, 5,
    (float[]){
        1.00,1.00,1.00,1.00,0.12,1.01,0.06,0.95,
        0.06,0.95,-0.01,0.90,0.00,0.10,0.05,0.05,
        0.05,0.05,0.11,-0.01,1.00,0.00,1.00,0.00,
        1.00,0.00,1.00,0.00,0.00,0.00,0.00,0.00,
        0.00,0.50,0.00,0.50,0.50,0.50,0.50,0.50
    });
  ch = font->_char + 'F';
  TGAFontInitChar(ch, 3,
    (float[]){
        0.00,0.50,0.00,0.50,0.50,0.50,0.50,0.50,
        1.00,1.00,1.00,1.00,0.12,1.01,0.06,0.95,
        0.06,0.95,-0.01,0.90,0.00,0.00,0.00,0.00
    });
  ch = font->_char + 'G';
  TGAFontInitChar(ch, 5,
    (float[]){
        1.00,0.84,1.00,1.00,0.74,1.00,0.50,1.00,
        0.50,1.00,0.00,1.00,0.00,0.81,0.00,0.50,
        0.00,0.50,0.00,0.18,0.00,0.00,0.50,0.00,
        0.50,0.00,1.00,0.00,1.00,0.50,1.00,0.50,
        1.00,0.50,1.00,0.50,0.50,0.50,0.50,0.50
    });
  ch = font->_char + 'H';
  TGAFontInitChar(ch, 3,
    (float[]){
        1.00,1.00,1.00,1.00,1.00,0.00,1.00,0.00,
        0.00,0.50,0.00,0.50,1.00,0.50,1.00,0.50,
        0.00,1.00,0.00,1.00,0.00,0.00,0.00,0.00
    });
  ch = font->_char + 'I';
  TGAFontInitChar(ch, 3,
    (float[]){
        0.00,0.00,0.00,0.00,1.00,0.00,1.00,0.00,
        0.50,1.00,0.50,1.00,0.50,0.00,0.50,0.00,
        0.10,1.00,0.10,1.00,0.90,1.00,0.90,1.00
    });
  ch = font->_char + 'J';
  TGAFontInitChar(ch, 3,
    (float[]){
        0.66,1.00,0.66,1.00,1.00,0.00,0.50,0.00,
        0.50,0.00,0.00,0.00,0.00,0.33,0.00,0.50,
        0.00,1.00,0.00,1.00,1.00,1.00,1.00,1.00
    });
  ch = font->_char + 'K';
  TGAFontInitChar(ch, 4,
    (float[]){
        0.50,0.54,0.50,0.00,1.00,0.00,1.00,0.00,
        0.00,0.50,0.00,0.50,0.00,0.50,0.33,0.50,
        0.33,0.50,0.67,0.51,1.00,1.00,1.00,1.00,
        0.00,1.00,0.00,1.00,0.00,0.00,0.00,0.00
    });
  ch = font->_char + 'L';
  TGAFontInitChar(ch, 2,
    (float[]){
        0.00,1.00,0.00,1.00,0.00,0.12,0.05,0.05,
        0.05,0.05,0.08,0.00,1.00,0.00,1.00,0.00
    });
  ch = font->_char + 'M';
  TGAFontInitChar(ch, 4,
    (float[]){
        0.00,0.00,0.00,0.00,0.00,1.00,0.00,1.00,
        0.00,1.00,0.00,1.00,0.34,0.67,0.50,0.67,
        0.50,0.67,0.66,0.67,1.00,1.00,1.00,1.00,
        1.00,1.00,1.00,1.00,1.00,0.00,1.00,0.00
    });
  ch = font->_char + 'N';
  TGAFontInitChar(ch, 3,
    (float[]){
        0.00,0.00,0.00,0.00,0.00,1.00,0.00,1.00,
        0.00,1.00,0.33,1.00,0.66,0.00,1.00,0.00,
        1.00,0.00,1.00,0.00,1.00,1.00,1.00,1.00
    });
  ch = font->_char + 'O';
  TGAFontInitChar(ch, 4,
    (float[]){
        0.50,1.00,1.00,1.00,1.00,1.00,1.00,0.50,
        1.00,0.50,1.00,0.00,1.00,0.00,0.50,0.00,
        0.50,0.00,0.00,0.00,0.00,0.00,0.00,0.50,
        0.00,0.50,0.00,1.00,0.00,1.00,0.50,1.00
    });
  ch = font->_char + 'P';
  TGAFontInitChar(ch, 3,
    (float[]){
        0.00,0.00,0.00,0.00,0.00,1.00,0.00,1.00,
        0.00,1.00,0.50,1.00,1.00,1.00,1.00,0.67,
        1.00,0.67,1.00,0.33,0.50,0.33,0.00,0.33
    });
```

```
ch = font->_char + 'Q';
TGAFontInitChar(ch, 5,
  (float[]){
      0.66,0.33,0.66,0.33,1.00,0.00,1.00,0.00,
      0.50,1.00,1.00,1.00,1.00,1.00,1.00,0.50,
      1.00,0.50,1.00,0.00,1.00,0.00,0.50,0.00,
      0.50,0.00,0.00,0.00,0.00,0.00,0.00,0.50,
      0.00,0.50,0.00,1.00,0.00,1.00,0.50,1.00
  });
ch = font->_char + 'R';
TGAFontInitChar(ch, 4,
  (float[]){
      0.00,0.33,0.33,0.00,1.00,0.00,1.00,0.00,
      0.00,0.00,0.00,0.00,0.00,1.00,0.00,1.00,
      0.00,1.00,0.50,1.00,1.00,1.00,1.00,0.67,
      1.00,0.67,1.00,0.33,0.50,0.33,0.00,0.33
  });
ch = font->_char + 'S';
TGAFontInitChar(ch, 5,
  (float[]){
      1.00,0.83,1.00,0.99,1.00,1.00,0.50,1.00,
      0.50,1.00,0.00,1.00,0.00,0.83,0.00,0.67,
      0.00,0.67,0.00,0.50,1.00,0.67,1.00,0.50,
      1.00,0.50,1.00,0.33,1.00,0.00,0.50,0.00,
      0.50,0.00,0.00,0.00,0.00,0.16,0.00,0.33
  });
ch = font->_char + 'T';
TGAFontInitChar(ch, 2,
  (float[]){
      0.50,1.00,0.50,1.00,0.50,0.00,0.50,0.00,
      0.00,1.00,0.00,1.00,1.00,1.00,1.00,1.00
  });
ch = font->_char + 'U';
TGAFontInitChar(ch, 2,
  (float[]){
      0.00,1.00,0.00,0.50,0.01,0.00,0.50,0.00,
      0.50,0.00,1.00,0.00,1.00,0.51,1.00,1.00
  });
ch = font->_char + 'V';
TGAFontInitChar(ch, 2,
  (float[]){
      0.00,1.00,0.00,1.00,0.34,0.00,0.50,0.00,
      0.50,0.00,0.67,0.00,1.00,1.00,1.00,1.00
  });
ch = font->_char + 'W';
TGAFontInitChar(ch, 4,
  (float[]){
      0.00,1.00,0.00,1.00,0.16,0.00,0.33,0.00,
      0.33,0.00,0.50,0.00,0.50,0.50,0.50,0.50,
      0.50,0.50,0.50,0.50,0.50,0.00,0.66,0.00,
      0.66,0.00,0.82,0.00,1.00,1.00,1.00,1.00
  });
ch = font->_char + 'X';
TGAFontInitChar(ch, 4,
  (float[]){
      1.00,1.00,1.00,1.00,0.50,0.67,0.50,0.51,
      0.50,0.51,0.50,0.33,0.00,0.00,0.00,0.00,
      0.00,1.00,0.00,1.00,0.50,0.67,0.50,0.50,
      0.50,0.50,0.50,0.33,1.00,0.00,1.00,0.00
  });
ch = font->_char + 'Y';
TGAFontInitChar(ch, 3,
  (float[]){
      1.00,1.00,1.00,1.00,0.50,0.67,0.50,0.50,
      0.00,1.00,0.00,1.00,0.50,0.67,0.50,0.50,
      0.50,0.50,0.50,0.33,0.50,0.00,0.50,0.00
  });
ch = font->_char + 'Z';
TGAFontInitChar(ch, 3,
  (float[]){
      0.00,1.00,0.00,1.00,1.00,1.00,1.00,1.00,
      1.00,1.00,1.00,0.67,0.00,0.33,0.00,0.00,
      0.00,0.00,0.00,0.00,1.00,0.00,1.00,0.00
  });
ch = font->_char + '0';
TGAFontInitChar(ch, 5,
  (float[]){
      0.00,0.00,0.00,0.00,1.00,1.00,1.00,1.00,
      0.50,1.00,1.00,1.00,1.00,1.00,1.00,0.50,
      1.00,0.50,1.00,0.00,1.00,0.00,0.50,0.00,
      0.50,0.00,0.00,0.00,0.00,0.00,0.00,0.50,
      0.00,0.50,0.00,1.00,0.00,1.00,0.50,1.00
  });
ch = font->_char + '1';
TGAFontInitChar(ch, 3,
  (float[]){
      0.00,0.00,0.00,0.00,1.00,0.00,1.00,0.00,
      0.00,0.67,0.33,0.67,0.50,1.00,0.50,1.00,
      0.50,1.00,0.50,1.00,0.50,0.00,0.50,0.00
  });
ch = font->_char + '2';
TGAFontInitChar(ch, 4,
  (float[]){
      0.00,0.67,0.00,1.00,0.34,1.00,0.50,1.00,
      0.50,1.00,0.66,1.00,1.00,1.00,1.00,0.67,
      1.00,0.67,1.00,0.50,0.00,0.33,0.00,0.00,
      0.00,0.00,0.00,0.00,1.00,0.00,1.00,0.00
  });
ch = font->_char + '3';
TGAFontInitChar(ch, 6,
  (float[]){
      0.00,0.67,0.00,0.83,0.00,1.00,0.50,1.00,
      0.50,1.00,1.00,1.00,1.00,0.83,1.00,0.67,
      1.00,0.67,1.00,0.50,0.50,0.50,0.50,0.50,
      0.50,0.50,0.50,0.50,1.00,0.50,1.00,0.33,
      1.00,0.33,1.00,0.00,1.00,0.00,0.50,0.00,
      0.50,0.00,0.00,0.00,0.00,0.16,0.00,0.33
  });
ch = font->_char + '4';
TGAFontInitChar(ch, 3,
  (float[]){
      1.00,0.33,1.00,0.33,0.00,0.33,0.00,0.33,
      0.00,0.33,0.50,0.50,0.66,1.00,0.66,1.00,
      0.66,1.00,0.66,1.00,0.66,0.00,0.66,0.00
  });
ch = font->_char + '5';
TGAFontInitChar(ch, 5,
  (float[]){
      1.00,1.00,1.00,1.00,0.33,1.00,0.33,1.00,
      0.33,1.00,0.33,1.00,0.00,0.67,0.00,0.67,
      0.00,0.67,0.00,0.67,1.00,1.01,1.00,0.33,
      1.00,0.33,1.00,0.00,0.67,0.00,0.50,0.00,
      0.50,0.00,0.33,0.00,0.00,0.16,0.00,0.33
  });
```

```
ch = font->_char + '6';
TGAFontInitChar(ch, 6,
  (float[]){
      0.00,0.33,0.00,0.50,0.33,0.50,0.50,0.50,
      0.50,0.50,0.67,0.50,1.00,0.50,1.00,0.33,
      1.00,0.33,1.00,0.16,1.00,0.00,0.50,0.00,
      0.50,0.00,0.00,0.00,0.00,0.33,0.00,0.50,
      0.00,0.50,0.00,1.00,0.50,1.00,0.50,1.00,
      0.50,1.00,0.50,1.00,1.00,1.00,1.00,0.67
  });
ch = font->_char + '7';
TGAFontInitChar(ch, 2,
  (float[]){
      0.00,1.00,0.00,1.00,1.00,1.00,1.00,1.00,
      1.00,1.00,1.00,1.00,0.33,0.67,0.33,0.00
  });
ch = font->_char + '8';
TGAFontInitChar(ch, 6,
  (float[]){
      0.50,1.00,1.00,1.00,1.00,0.67,0.50,0.67,
      0.50,0.67,0.33,0.67,0.00,0.50,0.00,0.33,
      0.00,0.33,0.00,0.00,0.33,0.00,0.50,0.00,
      0.50,0.00,0.66,0.00,1.00,0.00,1.00,0.33,
      1.00,0.33,1.00,0.50,0.66,0.67,0.50,0.67,
      0.50,0.67,0.00,0.67,0.00,1.00,0.50,1.00
  });
ch = font->_char + '9';
TGAFontInitChar(ch, 5,
  (float[]){
      0.33,0.00,0.50,0.00,1.00,0.00,1.00,0.50,
      1.00,0.50,1.00,1.00,0.66,1.00,0.50,1.00,
      0.50,1.00,0.33,1.00,0.00,1.00,0.00,0.67,
      0.00,0.67,0.00,0.50,0.33,0.50,0.50,0.50,
      0.50,0.50,0.67,0.50,1.00,0.50,1.00,0.67
  });
ch = font->_char + '!';
TGAFontInitChar(ch, 3,
  (float[]){
      0.50,0.18,0.44,0.18,0.44,0.07,0.50,0.07,
      0.50,0.07,0.56,0.07,0.56,0.18,0.50,0.18,
      0.50,1.00,0.50,1.00,0.50,0.33,0.50,0.33
  });
ch = font->_char + '"';
TGAFontInitChar(ch, 2,
  (float[]){
      0.66,1.00,0.66,1.00,0.66,0.75,0.66,0.75,
      0.33,1.00,0.33,1.00,0.33,0.75,0.33,0.75
  });
ch = font->_char + '\'';
TGAFontInitChar(ch, 1,
  (float[]){
      0.25,1.00,0.25,1.00,0.25,0.49,0.00,0.50
  });
ch = font->_char + '#';
TGAFontInitChar(ch, 4,
  (float[]){
      0.75,1.00,0.75,1.00,0.66,0.00,0.66,0.00,
      0.33,1.00,0.33,1.00,0.25,0.00,0.25,0.00,
      0.00,0.25,0.00,0.25,1.00,0.25,1.00,0.25,
      0.00,0.67,0.00,0.67,1.00,0.67,1.00,0.67
  });
ch = font->_char + '$';
```

```
TGAFontInitChar(ch, 6,
  (float[]){
      0.50,1.00,0.50,1.00,0.50,0.00,0.50,0.00,
      1.00,0.83,1.00,0.99,1.00,1.00,0.50,1.00,
      0.50,1.00,0.00,1.00,0.00,0.83,0.00,0.67,
      0.00,0.67,0.00,0.50,1.00,0.67,1.00,0.50,
      1.00,0.50,1.00,0.33,1.00,0.00,0.50,0.00,
      0.50,0.00,0.00,0.00,0.00,0.16,0.00,0.33
  });
ch = font->_char + '%';
TGAFontInitChar(ch, 9,
  (float[]){
      0.75,0.50,1.00,0.50,1.00,0.50,1.00,0.25,
      1.00,0.25,1.00,0.00,1.00,0.00,0.75,0.00,
      0.75,0.00,0.50,0.00,0.50,0.00,0.50,0.25,
      0.50,0.25,0.50,0.50,0.50,0.50,0.75,0.50,
      0.25,1.00,0.50,1.00,0.50,1.00,0.50,0.75,
      0.50,0.75,0.50,0.50,0.50,0.50,0.25,0.50,
      0.25,0.50,0.00,0.50,0.00,0.50,0.00,0.75,
      0.00,0.75,0.00,1.00,0.00,1.00,0.25,1.00,
      0.00,0.00,0.00,0.00,1.00,1.00,1.00,1.00
  });
ch = font->_char + '&';
TGAFontInitChar(ch, 6,
  (float[]){
      1.00,0.00,1.00,0.33,0.76,0.67,0.50,0.67,
      0.50,0.67,0.00,0.66,0.00,1.00,0.50,1.00,
      0.50,1.00,1.00,1.00,1.00,0.67,0.50,0.67,
      0.50,0.67,0.33,0.67,0.00,0.50,0.00,0.33,
      0.00,0.33,0.00,0.00,0.33,0.00,0.50,0.00,
      0.50,0.00,0.66,0.00,1.00,0.17,1.00,0.50
  });
ch = font->_char + '(';
TGAFontInitChar(ch, 1,
  (float[]){
      1.00,1.00,0.75,0.75,0.75,0.25,1.00,0.00
  });
ch = font->_char + ')';
TGAFontInitChar(ch, 1,
  (float[]){
      0.00,1.00,0.25,0.75,0.25,0.25,0.00,0.00
  });
ch = font->_char + '=';
TGAFontInitChar(ch, 2,
  (float[]){
      0.00,0.33,0.00,0.33,1.00,0.33,1.00,0.33,
      0.00,0.67,0.00,0.67,1.00,0.67,1.00,0.67
  });
ch = font->_char + '~';
TGAFontInitChar(ch, 1,
  (float[]){
      0.00,0.50,0.33,0.75,0.66,0.25,1.00,0.50
  });
ch = font->_char + '`';
TGAFontInitChar(ch, 1,
  (float[]){
      0.75,1.00,0.75,1.00,0.75,0.49,1.00,0.50
  });
ch = font->_char + '{';
TGAFontInitChar(ch, 2,
  (float[]){
      1.00,1.00,0.75,1.00,1.00,0.50,0.75,0.50,
      0.75,0.50,1.00,0.50,0.76,0.00,1.00,0.00
  });
ch = font->_char + '}';
TGAFontInitChar(ch, 2,
  (float[]){
      0.00,1.00,0.25,1.00,0.00,0.50,0.25,0.50,
      0.25,0.50,-0.02,0.50,0.25,0.00,0.00,0.00
  });
ch = font->_char + '*';
TGAFontInitChar(ch, 2,
  (float[]){
      0.00,0.00,0.00,0.00,1.00,1.00,1.00,1.00,
      0.00,1.00,0.00,1.00,1.00,0.00,1.00,0.00
  });
ch = font->_char + '+';
TGAFontInitChar(ch, 2,
  (float[]){
      0.00,0.50,0.00,0.50,1.00,0.50,1.00,0.50,
      0.50,1.00,0.50,1.00,0.50,0.00,0.50,0.00
  });
ch = font->_char + '<';
TGAFontInitChar(ch, 2,
  (float[]){
      1.00,1.00,1.00,1.00,0.00,0.50,0.00,0.50,
      0.00,0.50,0.00,0.50,1.00,0.00,1.00,0.00
  });
ch = font->_char + '>';
TGAFontInitChar(ch, 2,
  (float[]){
      0.00,1.00,0.00,1.00,1.00,0.50,1.00,0.50,
      1.00,0.50,1.00,0.50,0.00,0.00,0.00,0.00
  });
ch = font->_char + '?';
TGAFontInitChar(ch, 5,
  (float[]){
      0.00,0.67,0.00,1.00,0.34,1.00,0.50,1.00,
      0.50,1.00,0.66,1.00,1.00,1.00,1.00,0.67,
      1.00,0.67,1.00,0.33,0.50,0.66,0.50,0.33,
      0.50,0.18,0.44,0.18,0.44,0.07,0.50,0.07,
      0.50,0.07,0.56,0.07,0.56,0.18,0.50,0.18
  });
ch = font->_char + '.';
TGAFontInitChar(ch, 2,
  (float[]){
      0.13,0.25,0.00,0.25,0.00,0.00,0.13,0.00,
      0.13,0.00,0.25,0.00,0.25,0.25,0.13,0.25
  });
ch = font->_char + ',';
TGAFontInitChar(ch, 1,
  (float[]){
      0.25,0.18,0.25,0.18,0.25,-0.33,0.00,-0.32
  });
ch = font->_char + '/';
TGAFontInitChar(ch, 1,
  (float[]){
      1.00,1.00,1.00,1.00,0.00,0.00,0.00,0.00
  });
ch = font->_char + '\\';
TGAFontInitChar(ch, 1,
  (float[]){
      0.00,1.00,0.00,1.00,1.00,0.00,1.00,0.00
  });
```

```
ch = font->_char + '[';
TGAFontInitChar(ch, 3,
  (float[]){
      1.00,1.00,1.00,1.00,0.75,1.00,0.75,1.00,
      0.75,1.00,0.75,1.00,0.75,0.00,0.75,0.00,
      0.75,0.00,0.75,0.00,1.00,0.00,1.00,0.00
  });
ch = font->_char + ']';
TGAFontInitChar(ch, 3,
  (float[]){
      0.00,1.00,0.00,1.00,0.25,1.00,0.25,1.00,
      0.25,1.00,0.25,1.00,0.25,0.0,0.25,0.0,
      0.25,0.0,0.25,0.0,0.00,0.0,0.00,0.0
  });
ch = font->_char + '-';
TGAFontInitChar(ch, 1,
  (float[]){
      0.00,0.50,0.00,0.50,1.00,0.50,1.00,0.50
  });
ch = font->_char + '|';
TGAFontInitChar(ch, 1,
  (float[]){
      0.50,1.00,0.50,1.00,0.50,0.00,0.50,0.00
  });
ch = font->_char + '_';
TGAFontInitChar(ch, 1,
  (float[]){
      0.00,0.00,0.00,0.00,1.00,0.00,1.00,0.00,
  });
ch = font->_char + ';';
TGAFontInitChar(ch, 3,
  (float[]){
      0.25,0.47,0.18,0.47,0.18,0.36,0.25,0.36,
      0.25,0.36,0.30,0.36,0.30,0.47,0.25,0.47,
      0.25,0.18,0.25,0.18,0.25,-0.33,0.00,-0.32,
  });
ch = font->_char + ':';
TGAFontInitChar(ch, 4,
  (float[]){
      0.50,0.72,0.44,0.72,0.44,0.61,0.50,0.61,
      0.50,0.61,0.56,0.61,0.56,0.72,0.50,0.72,
      0.50,0.39,0.44,0.39,0.44,0.28,0.50,0.28,
      0.50,0.28,0.56,0.28,0.56,0.39,0.50,0.39
  });
ch = font->_char + 'a';
TGAFontInitChar(ch, 4,
  (float[]){
      0.66,0.67,0.25,0.67,0.00,0.66,0.00,0.33,
      0.00,0.33,0.00,0.00,0.26,0.01,0.49,0.01,
      0.49,0.01,0.74,0.01,0.75,0.33,0.75,0.67,
      0.75,0.67,0.75,0.25,0.75,0.01,1.00,0.00
  });
ch = font->_char + 'b';
TGAFontInitChar(ch, 4,
  (float[]){
      0.00,1.00,0.00,0.50,0.00,0.00,0.50,0.00,
      0.50,0.00,1.00,0.00,1.00,0.33,1.00,0.50,
      1.00,0.50,1.00,0.67,0.59,0.67,0.42,0.67,
      0.42,0.67,0.25,0.67,0.06,0.58,0.06,0.33
  });
ch = font->_char + 'c';
TGAFontInitChar(ch, 4,
  (float[]){
      1.00,0.50,1.00,0.67,0.67,0.67,0.50,0.67,
      0.50,0.67,0.33,0.67,0.00,0.66,0.00,0.33,
      0.00,0.33,0.00,0.00,0.34,0.00,0.50,0.00,
      0.50,0.00,0.66,0.00,1.00,0.00,1.00,0.25
  });
ch = font->_char + 'd';
TGAFontInitChar(ch, 4,
  (float[]){
      1.00,1.00,1.01,0.50,1.00,0.00,0.50,0.00,
      0.50,0.00,0.00,0.00,0.00,0.33,0.00,0.50,
      0.00,0.50,0.00,0.67,0.44,0.66,0.59,0.66,
      0.59,0.66,0.75,0.66,0.95,0.59,0.95,0.34
  });
ch = font->_char + 'e';
TGAFontInitChar(ch, 6,
  (float[]){
      1.00,0.25,1.00,0.00,0.66,0.00,0.50,0.00,
      0.50,0.00,0.34,0.00,0.00,0.00,0.00,0.33,
      0.00,0.33,0.00,0.66,0.33,0.67,0.50,0.67,
      0.50,0.67,0.67,0.67,1.00,0.67,1.00,0.50,
      1.00,0.50,1.00,0.33,0.67,0.33,0.50,0.33,
      0.50,0.33,0.33,0.33,0.00,0.33,0.00,0.33
  });
ch = font->_char + 'f';
TGAFontInitChar(ch, 4,
  (float[]){
      0.00,0.50,0.00,0.50,0.66,0.50,0.66,0.50,
      1.00,0.75,1.00,1.00,0.75,1.00,0.50,1.00,
      0.50,1.00,0.25,1.00,0.25,0.83,0.25,0.67,
      0.25,0.67,0.25,0.50,0.25,0.00,0.25,0.00
  });
ch = font->_char + 'g';
TGAFontInitChar(ch, 6,
  (float[]){
      1.00,0.33,1.00,0.00,0.67,0.00,0.50,0.00,
      0.50,0.00,0.33,0.00,0.00,-0.01,0.00,0.33,
      0.00,0.33,0.00,0.67,0.25,0.67,0.50,0.67,
      0.50,0.67,0.75,0.67,1.00,0.66,1.00,0.33,
      1.00,0.33,1.00,0.00,1.00,-0.33,0.50,-0.33,
      0.50,-0.33,0.41,-0.33,0.33,-0.33,0.33,-0.33
  });
ch = font->_char + 'h';
TGAFontInitChar(ch, 3,
  (float[]){
      0.00,0.33,0.25,0.67,1.00,1.00,1.00,0.50,
      1.00,0.50,1.00,0.25,1.00,0.00,1.00,0.00,
      0.00,1.00,0.00,1.00,0.00,0.00,0.00,0.00
  });
ch = font->_char + 'i';
TGAFontInitChar(ch, 5,
  (float[]){
      0.25,0.87,0.19,0.87,0.19,0.76,0.25,0.76,
      0.25,0.76,0.31,0.76,0.31,0.87,0.25,0.87,
      0.00,0.00,0.25,0.00,0.25,0.42,0.25,0.50,
      0.25,0.50,0.25,0.25,0.26,0.00,0.50,0.00,
      0.50,0.00,0.72,0.00,1.00,0.00,1.00,0.00
  });
ch = font->_char + 'j';
TGAFontInitChar(ch, 5,
  (float[]){
      0.75,0.87,0.69,0.87,0.69,0.76,0.75,0.76,
      0.75,0.76,0.81,0.76,0.81,0.87,0.76,0.87,
      0.00,0.00,0.00,-0.33,0.33,-0.33,0.50,-0.33,
      0.50,-0.33,0.75,-0.33,0.75,0.33,0.75,0.50,
      0.75,0.50,0.75,0.33,0.76,0.00,1.00,0.00
  });
ch = font->_char + 'k';
TGAFontInitChar(ch, 4,
  (float[]){
      0.00,0.50,0.25,0.67,1.00,0.75,1.00,0.50,
      1.00,0.50,1.00,0.25,0.50,0.33,0.00,0.33,
      0.00,0.33,0.32,0.33,0.75,0.25,1.00,0.00,
      0.00,1.00,0.00,1.00,0.00,0.00,0.00,0.00
  });
ch = font->_char + 'l';
TGAFontInitChar(ch, 6,
  (float[]){
      0.00,0.00,0.25,0.00,0.25,0.34,0.25,0.50,
      0.25,0.50,0.25,0.66,0.25,1.00,0.50,1.00,
      0.50,1.00,0.66,1.00,0.75,1.00,0.75,0.76,
      0.75,0.76,0.75,0.51,0.50,0.33,0.25,0.33,
      0.25,0.33,0.26,0.00,0.33,0.00,0.66,0.00,
      0.66,0.00,0.76,0.00,1.00,0.00,1.00,0.00
  });
ch = font->_char + 'm';
TGAFontInitChar(ch, 5,
  (float[]){
      0.00,0.67,0.00,0.67,0.00,0.00,0.00,0.00,
      0.00,0.25,0.00,0.59,0.25,0.67,0.33,0.67,
      0.33,0.67,0.50,0.66,0.50,0.00,0.50,0.00,
      0.50,0.00,0.50,0.00,0.50,0.67,0.74,0.67,
      0.74,0.67,1.00,0.67,1.00,0.00,1.00,0.00
  });
ch = font->_char + 'n';
TGAFontInitChar(ch, 3,
  (float[]){
      0.00,0.67,0.00,0.67,0.00,0.00,0.00,0.00,
      0.00,0.25,0.00,0.50,0.25,0.67,0.66,0.67,
      0.66,0.67,1.00,0.67,1.00,0.24,1.00,0.00
  });
ch = font->_char + 'o';
TGAFontInitChar(ch, 4,
  (float[]){
      0.50,0.67,1.00,0.67,1.00,0.66,1.00,0.33,
      1.00,0.33,1.00,0.00,1.00,0.00,0.50,0.00,
      0.50,0.00,0.00,0.00,0.00,-0.01,0.00,0.33,
      0.00,0.33,0.00,0.67,0.00,0.67,0.50,0.67
  });
ch = font->_char + 'p';
TGAFontInitChar(ch, 5,
  (float[]){
      0.00,-0.33,0.00,-0.33,0.00,0.16,0.00,0.33,
      0.00,0.33,0.00,0.50,0.00,0.67,0.50,0.67,
      0.50,0.67,1.00,0.67,1.00,0.50,1.00,0.33,
      1.00,0.33,1.00,0.16,1.00,0.00,0.50,0.00,
      0.50,0.00,0.00,0.00,0.00,0.00,0.00,0.00
  });
ch = font->_char + 'q';
TGAFontInitChar(ch, 5,
  (float[]){
      1.00,0.00,1.00,0.00,0.75,0.00,0.50,0.00,
      0.50,0.00,0.25,0.00,0.00,-0.01,0.00,0.33,
      0.00,0.33,0.00,0.67,0.25,0.67,0.50,0.67,
      0.50,0.67,0.75,0.67,1.00,0.66,1.00,0.33,
      1.00,0.33,1.00,0.00,1.00,-0.33,1.00,-0.33
  });
ch = font->_char + 'r';
TGAFontInitChar(ch, 2,
  (float[]){
      0.00,0.67,0.00,0.67,0.00,0.00,0.00,0.00,
      0.00,0.33,0.25,0.67,1.00,1.00,1.00,0.50
  });
ch = font->_char + 's';
TGAFontInitChar(ch, 5,
  (float[]){
      1.00,0.50,1.00,0.66,1.00,0.67,0.50,0.67,
      0.50,0.67,0.00,0.67,0.00,0.66,0.00,0.50,
      0.00,0.50,0.00,0.33,1.00,0.50,1.00,0.33,
      1.00,0.33,1.00,0.16,1.00,0.00,0.50,0.00,
      0.50,0.00,0.00,0.00,0.00,0.08,0.00,0.25
  });
ch = font->_char + 't';
TGAFontInitChar(ch, 4,
  (float[]){
      0.00,0.00,0.25,0.00,0.25,0.17,0.25,0.25,
      0.00,0.67,0.00,0.67,0.50,0.67,0.50,0.67,
      0.25,1.00,0.25,1.00,0.25,0.33,0.25,0.25,
      0.25,0.25,0.25,0.01,0.50,0.00,1.00,0.00
  });
ch = font->_char + 'u';
TGAFontInitChar(ch, 3,
  (float[]){
      0.00,0.67,0.00,0.33,0.00,0.00,0.50,0.00,
      0.50,0.00,1.00,0.00,1.00,0.33,1.00,0.67,
      1.00,0.67,1.00,0.33,1.00,0.00,1.00,0.00
  });
ch = font->_char + 'v';
TGAFontInitChar(ch, 2,
  (float[]){
      0.00,0.67,0.00,0.67,0.34,0.00,0.50,0.00,
      0.50,0.00,0.66,0.00,1.00,0.67,1.00,0.67
  });
ch = font->_char + 'w';
TGAFontInitChar(ch, 4,
  (float[]){
      0.00,0.67,0.00,0.67,0.16,0.00,0.33,0.00,
      0.33,0.00,0.50,0.00,0.50,0.50,0.50,0.50,
      0.50,0.50,0.50,0.50,0.50,0.00,0.66,0.00,
      0.66,0.00,0.82,0.00,1.00,0.67,1.00,0.67
  });
ch = font->_char + 'x';
TGAFontInitChar(ch, 4,
  (float[]){
      0.00,0.00,0.25,0.00,0.51,0.24,0.50,0.33,
      0.50,0.33,0.50,0.41,0.76,0.67,1.00,0.67,
      0.00,0.67,0.25,0.67,0.50,0.41,0.50,0.33,
      0.50,0.33,0.50,0.25,0.75,0.00,1.00,0.00
  });
ch = font->_char + 'y';
TGAFontInitChar(ch, 3,
  (float[]){
      0.00,0.67,0.00,0.67,0.00,0.00,0.66,0.00,
      1.00,0.67,1.00,0.67,0.82,0.33,0.66,0.00,
      0.66,0.00,0.50,-0.33,0.50,-0.33,0.25,-0.33
  });
  ch = font->_char + 'z';
  TGAFontInitChar(ch, 3,
    (float[]){
        0.00,0.67,0.00,0.67,1.00,0.67,1.00,0.67,
        1.00,0.67,1.00,0.50,0.00,0.25,0.00,0.00,
        0.00,0.00,0.00,0.00,1.00,0.00,1.00,0.00
    });
  ch = font->_char + '@';
  TGAFontInitChar(ch, 8,
    (float[]){
        0.61,0.66,0.36,0.66,0.21,0.65,0.21,0.45,
        0.21,0.45,0.21,0.25,0.36,0.25,0.51,0.25,
        0.51,0.25,0.66,0.25,0.67,0.45,0.67,0.66,
        0.67,0.66,0.66,0.40,0.66,0.25,0.82,0.25,
        0.82,0.25,0.97,0.24,0.94,0.72,0.75,0.79,
        0.75,0.79,0.56,0.85,0.36,0.84,0.25,0.78,
        0.25,0.78,0.03,0.66,0.05,0.21,0.25,0.11,
        0.25,0.11,0.45,0.01,0.67,0.07,0.75,0.13
    });
  ch = font->_char + '^';
  TGAFontInitChar(ch, 2,
    (float[]){
        0.00,0.75,0.00,0.75,0.50,1.00,0.50,1.00,
        0.50,1.00,0.50,1.00,1.00,0.75,1.00,0.75
    });
}
```

3 Makefile

```
OPTIONS_DEBUG=-ggdb -g3 -Wall
OPTIONS_RELEASE=-O3
OPTIONS=$(OPTIONS_RELEASE)
INCPATH=/home/bayashi/Coding/Include
LIBPATH=/home/bayashi/Coding/Include

all : main testCurve

main: main.o tgapaint.o Makefile $(LIBPATH)/bcurve.o $(LIBPATH)/pbmath.o $(LIBPATH)/gset.o
gcc $(OPTIONS) main.o tgapaint.o  $(LIBPATH)/pbmath.o $(LIBPATH)/bcurve.o $(LIBPATH)/gset.o -o main -lm

testCurve: testCurve.o tgapaint.o Makefile $(LIBPATH)/bcurve.o $(LIBPATH)/pbmath.o $(LIBPATH)/gset.o
gcc $(OPTIONS) testCurve.o tgapaint.o  $(LIBPATH)/pbmath.o $(LIBPATH)/bcurve.o $(LIBPATH)/gset.o -o testCurve -lm

main.o : main.c tgapaint.h Makefile
gcc $(OPTIONS) -I$(INCPATH) -c main.c

testCurve.o : testCurve.c tgapaint.h Makefile
gcc $(OPTIONS) -I$(INCPATH) -c testCurve.c

tgapaint.o : tgapaint.c tgafont.c tgabrush.c tgapaint.h $(INCPATH)/bcurve.h $(INCPATH)/gset.h Makefile
gcc $(OPTIONS) -I$(INCPATH) -c tgapaint.c

clean :
rm -rf *.o main

valgrind :
valgrind -v --track-origins=yes --leak-check=full --gen-suppressions=yes --show-leak-kinds=all ./main
```

```
install :
cp tgapaint.h ../Include; cp tgapaint.o ../Include
```

4 Usage

```
#include <stdio.h>
#include <stdlib.h>
#include "tgapaint.h"

int main(void) {
  int ret;
  TGA *theTGA;
  // Create the TGA
  VecShort *dim = VecShortCreate(2);
  VecSet(dim, 0, 120); VecSet(dim, 1, 270);
  TGAPixel *pix = TGAGetWhitePixel();
  theTGA = TGACreate(dim, pix);
  if (theTGA == NULL) {
    fprintf(stderr, "Error while creating the tga\n");
    return 1;
  }
  // Set the color of some pixels
  printf("Set the color of some pixels\n");
  VecShort *pos = VecShortCreate(2);
  if (pos == NULL) {
    fprintf(stderr, "VecShortCreate failed\n");
    return 2;
  }
  VecSet(pos, 0, 60); VecSet(pos, 1, 50);
  TGASetPix(theTGA, pos, pix);
  pix->_rgba[0] = 255; pix->_rgba[1] = 0; pix->_rgba[2] = 0;
  VecSet(pos, 0, 90); VecSet(pos, 1, 50);
  TGASetPix(theTGA, pos, pix);
  pix->_rgba[0] = 0; pix->_rgba[1] = 0; pix->_rgba[2] = 255;
  VecSet(pos, 0, 60); VecSet(pos, 1, 25);
  TGASetPix(theTGA, pos, pix);
  pix->_rgba[0] = 0; pix->_rgba[1] = 255; pix->_rgba[2] = 0;
  VecSet(pos, 0, 30); VecSet(pos, 1, 75);
  TGASetPix(theTGA, pos, pix);
  // Draw some lines
  printf("Draw some lines\n");
  TGAPencil *pen = TGAGetBlackPencil();
  pix->_rgba[0] = 0; pix->_rgba[1] = 0; pix->_rgba[2] = 0;
  TGAPencilSetColor(pen, pix);
  VecFloat *from = VecFloatCreate(2);
  if (from == NULL) {
    fprintf(stderr, "VecFloatCreate failed\n");
    return 3;
  }
  VecFloat *to = VecFloatCreate(2);
  if (to == NULL) {
    fprintf(stderr, "VecFloatCreate failed\n");
    return 4;
  }
  VecSet(from, 0, 50.5); VecSet(from, 1, 40.5);
  VecSet(to, 0, 50.5); VecSet(to, 1, 60.5);
  TGADrawLine(theTGA, from, to, pen);
  VecSet(from, 0, 50.5); VecSet(from, 1, 60.5);
  VecSet(to, 0, 70.5); VecSet(to, 1, 60.5);
  TGADrawLine(theTGA, from, to, pen);
  pix->_rgba[0] = 255; pix->_rgba[1] = 0; pix->_rgba[2] = 255;
  VecSet(from, 0, -10.5); VecSet(from, 1, 50.5);
  VecSet(to, 0, 60.5); VecSet(to, 1, -10.5);
  TGADrawLine(theTGA, from, to, pen);
  VecSet(from, 0, 60.5); VecSet(from, 1, -10.5);
  VecSet(to, 0, 130.5); VecSet(to, 1, 50.5);
  TGADrawLine(theTGA, from, to, pen);
  VecSet(from, 0, 130.5); VecSet(from, 1, 50.5);
  VecSet(to, 0, 60.5); VecSet(to, 1, 110.5);
  TGADrawLine(theTGA, from, to, pen);
  VecSet(from, 0, 60.5); VecSet(from, 1, 110.5);
  VecSet(to, 0, -10.5); VecSet(to, 1, 50.5);
  TGADrawLine(theTGA, from, to, pen);
  // Apply gaussian blur
  printf("Apply Gaussian blur\n");
  TGAFilterGaussBlur(theTGA, 0.5, 2.0);
  // Draw a rectangle
  printf("Draw a rectangle\n");
  pix->_rgba[0] = 0; pix->_rgba[1] = 255; pix->_rgba[2] = 255;
  TGAPencilSetColor(pen, pix);
  VecSet(from, 0, 70.5); VecSet(from, 1, 40.5);
  VecSet(to, 0, 100.5); VecSet(to, 1, 10.5);
  TGADrawRect(theTGA, from, to, pen);
  // Draw a filled rectangle
  printf("Draw a filled rectangle\n");
  pix->_rgba[0] = 255; pix->_rgba[1] = 255; pix->_rgba[2] = 0;
  TGAPencilSetColor(pen, pix);
  VecSet(from, 0, 75.5); VecSet(from, 1, 35.5);
  VecSet(to, 0, 95.5); VecSet(to, 1, 15.5);
  TGAFillRect(theTGA, from, to, pen);
  // Draw an ellipse
  printf("Draw an ellipse\n");
  pix->_rgba[0] = 128; pix->_rgba[1] = 128; pix->_rgba[2] = 128;
  TGAPencilSetColor(pen, pix);
  VecFloat *center = VecFloatCreate(2);
  VecSet(center, 0, 30.5); VecSet(center, 1, 50.5);
  VecFloat *radius = VecFloatCreate(2);
  VecSet(radius, 0, 15.5); VecSet(radius, 1, 20.5);
  TGADrawEllipse(theTGA, center, radius, pen);
  // Draw a filled ellipse
  printf("Draw a filled ellipse\n");
  pix->_rgba[0] = 200; pix->_rgba[1] = 200; pix->_rgba[2] = 200;
  TGAPencilSetColor(pen, pix);
  VecSet(center, 0, 60.5); VecSet(center, 1, 75.5);
  VecSet(radius, 0, 25.5); VecSet(radius, 1, 10.5);
  TGAFillEllipse(theTGA, center, radius, pen);
  // Draw a line using blend colors
  printf("Draw a line using blend color\n");
  VecSet(from, 0, 30.5); VecSet(from, 1, 25.5);
  VecSet(to, 0, 90.5); VecSet(to, 1, 75.5);
  pix->_rgba[0] = pix->_rgba[3] = 255;
  pix->_rgba[1] = pix->_rgba[2] = 0;
  TGAPencilSetColor(pen, pix);
  pix->_rgba[2] = pix->_rgba[3] = 255;
  pix->_rgba[1] = pix->_rgba[0] = 0;
  TGAPencilSelectColor(pen, 1);
  TGAPencilSetColor(pen, pix);
  TGAPencilSetModeColorBlend(pen, 0, 1);
  TGADrawLine(theTGA, from, to, pen);
  // Draw a curve
```

```
printf("Draw a curve\n");
VecFloat *ctrlFrom = VecFloatCreate(2);
VecSet(ctrlFrom, 0, 40.5); VecSet(ctrlFrom, 1, 0.5);
VecFloat *ctrlTo = VecFloatCreate(2);
VecSet(ctrlTo, 0, 80.5); VecSet(ctrlTo, 1, 50.5);
BCurve *curve = BCurveCreate(3, 2);
if (curve == NULL) {
  fprintf(stderr, "Can’t create the curve\n");
  return 5;
}
BCurveSet(curve, 0, from);
BCurveSet(curve, 1, ctrlFrom);
BCurveSet(curve, 2, ctrlTo);
BCurveSet(curve, 3, to);
TGAPencilSetShapeRound(pen);
//TGAPencilSetShapePixel(pen);
TGAPencilSetAntialias(pen, true);
TGAPencilSetModeColorSolid(pen);
TGAPencilSetThickness(pen, 5.0);
TGADrawCurve(theTGA, curve, pen);
BCurveFree(&curve);
// Print some strings
printf("Print some strings\n");
TGAPencilSetThickness(pen, 1.0);
pix->_rgba[0] = pix->_rgba[1] = pix->_rgba[2] = 0;
TGAPencilSetColor(pen, pix);
TGAFont *font = TGAFontCreate(tgaFontDefault);
if (font == NULL) {
  fprintf(stderr, "Can’t create the font\n");
  return 6;
}
TGAFontSetAnchor(font, tgaFontAnchorTopLeft);
VecSet(from, 0, 5.0); VecSet(from, 1, 212.0);
TGAFontSetSize(font, 12.0);
VecFloat *v = VecFloatCreate(2);
VecSet(v, 0, 0.5); VecSet(v, 1, 1.0);
TGAFontSetScale(font, v);
VecSet(v, 0, 5.0); VecSet(v, 1, 3.0);
TGAFontSetSpace(font, v);
TGAPrintString(theTGA, pen, font,
  (unsigned char *)"ABCDEFGHIJ\nKLMNOPQRST\nUVWXYZ", from);
VecSet(from, 0, 5.0); VecSet(from, 1, 167.0);
TGAPrintString(theTGA, pen, font,
  (unsigned char *)"0123456789", from);
VecSet(from, 0, 5.0); VecSet(from, 1, 257.0);
TGAPrintString(theTGA, pen, font,
  (unsigned char *)"abcdefghij\nklmnopqrst\nuvwxyz^@", from);
VecSet(from, 0, 5.0); VecSet(from, 1, 152.0);
TGAPrintString(theTGA, pen, font,
  (unsigned char *)"!\"#$%&’()=\n~‘{}*+<>?,\n./\\[]-|_;:", from);
// Draw some Shapoids
printf("Draw some Shapoids\n");
Shapoid *shapoid = FacoidCreate(2);
if (shapoid == NULL) {
  fprintf(stderr, "Can’t create the shapoid\n");
  return 7;
}
VecSet(v, 0, 20.0); VecSet(v, 1, 0.0);
ShapoidSetAxis(shapoid, 0, v);
VecSet(v, 0, 10.0); VecSet(v, 1, 20.0);
ShapoidSetAxis(shapoid, 1, v);
VecSet(v, 0, 10.0); VecSet(v, 1, 40.0);
```

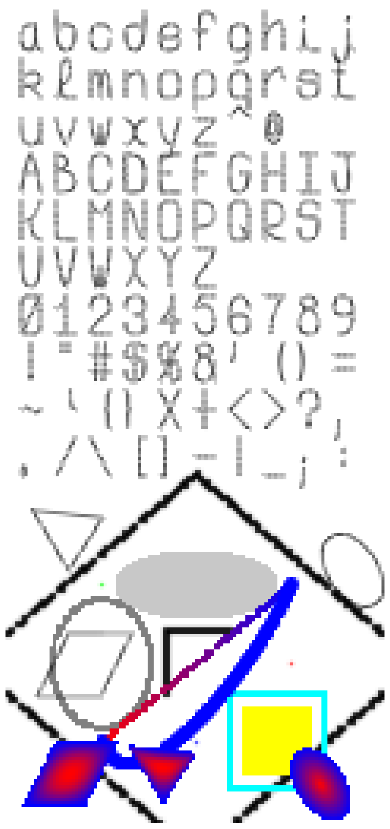
```
ShapoidSetPos(shapoid, v);
TGADrawShapoid(theTGA, shapoid, pen);
shapoid->_type = ShapoidTypePyramidoid;
VecSet(v, 0, 20.0); VecSet(v, 1, 80.0);
ShapoidSetPos(shapoid, v);
ShapoidRotate2D(shapoid, 1.0);
TGADrawShapoid(theTGA, shapoid, pen);
shapoid->_type = ShapoidTypeSpheroid;
VecSet(v, 0, 110.0); VecSet(v, 1, 80.0);
ShapoidSetPos(shapoid, v);
ShapoidRotate2D(shapoid, 0.5);
TGADrawShapoid(theTGA, shapoid, pen);
// Draw some filled shapoid with depth gradation
printf("Draw some shapoid with depth gradation\n");
TGAPencilSetModeColorBlend(pen, 0, 1);
TGAPencilSetShapePixel(pen);
pix->_rgba[3] = 255;
pix->_rgba[0] = 255; pix->_rgba[1] = 0; pix->_rgba[2] = 0;
TGAPencilSelectColor(pen, 0);
TGAPencilSetColor(pen, pix);
pix->_rgba[0] = 0; pix->_rgba[1] = 0; pix->_rgba[2] = 255;
TGAPencilSelectColor(pen, 1);
TGAPencilSetColor(pen, pix);
shapoid->_type = ShapoidTypeFacoid;
VecSet(v, 0, 20.0); VecSet(v, 1, 0.0);
ShapoidSetAxis(shapoid, 0, v);
VecSet(v, 0, 10.0); VecSet(v, 1, 20.0);
ShapoidSetAxis(shapoid, 1, v);
VecSet(v, 0, 5.0); VecSet(v, 1, 5.0);
ShapoidSetPos(shapoid, v);
TGAFillShapoid(theTGA, shapoid, pen);
shapoid->_type = ShapoidTypePyramidoid;
VecSet(v, 0, 50.0); VecSet(v, 1, 5.0);
ShapoidSetPos(shapoid, v);
ShapoidRotate2D(shapoid, 1.0);
TGAFillShapoid(theTGA, shapoid, pen);
shapoid->_type = ShapoidTypeSpheroid;
VecSet(v, 0, 100.0); VecSet(v, 1, 12.0);
ShapoidSetPos(shapoid, v);
ShapoidRotate2D(shapoid, 0.5);
TGAFillShapoid(theTGA, shapoid, pen);
// Save the TGA
TGASave(theTGA, "./out.tga");
//Free the tga
TGAFree(&theTGA);
// Load the TGA
ret = TGALoad(&theTGA, "./out.tga");
if (ret != 0) {
  fprintf(stderr, "Error while opening the file : %d\n", ret);
  return 8;
}
// Print its header on standard output stream
TGAPrintHeader(theTGA, stdout);
// Free the memory
ShapoidFree(&shapoid);
VecFree(&pos);
VecFree(&dim);
VecFree(&v);
VecFree(&ctrlFrom);
VecFree(&ctrlTo);
VecFree(&center);
VecFree(&radius);
```

```
  VecFree(&from);
  VecFree(&to);
  TGAFreeFont(&font);
  TGAFree(&theTGA);
  TGAPixelFree(&pix);
  TGAPencilFree(&pen);
  return 0;
}
```

Output:

```
ID length:         0
Colourmap type:    0
Image type:        2
Colour map offset: 0
Colour map length: 0
Colour map depth:  0
X origin:          0
Y origin:          0
Width:             120
Height:            270
Bits per pixel:    32
Descriptor:        0
```

Resulting image (enlarge):


abcdefghij
klmnopqrst
uvwxyz ^ @
ABCDEFGHIJ
KLMNOPQRST
UVWXYZ
0123456789
! " # $ % & ' ( ) =
~ ` { } X + < > ? ,
. / \ [ ] - | _ ; :