

TGAPaint

P. Baillehache

October 18, 2017

Contents

1	Interface	1
2	Code	8
2.1	tgapaint.c	8
2.2	tgafont.c	34
3	Makefile	51
4	Usage	52

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 TGAPAIINT_H
#define TGAPAIINT_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

// ===== 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 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;

// Main TGA structure
typedef struct TGA {
    // Header
    TGAHeader *_header;

```

```

    // Pixels (stored by rows)
    TGA_Pixel *_pixels;
} TGA;

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

// Enumeration of TGA_Pencil's shapes
typedef enum tgaPencilShape {
    // Square shape
    tgaPenSquare,
    // Round shape
    tgaPenRound,
    // Pixel mode
    tgaPenPixel
} tgaPencilShape;

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

// One character in a TGA_Font
typedef struct TGA_Char {
    // Number of curve defining this character
    int _nbCurve;
    // Definition of the curves
    // (1st anchor(x,y), 1st ctrl point(x,y),
    // 2nd ctrl point(x,y), 2nd anchor(x,y))
    // in pixels
    B_Curve *_curves[TGA_NBMAXCURVECHAR];
} TGA_Char;

// 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
// (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);

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

// Draw one stroke at 'pos' with 'pen'
// Don't do anything 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 TGADrawCurve(TGA *tga, BCurve *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 TGAFreePixel(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* TGABlendPixel(TGAPixel *pixA, TGAPixel *pixB, float blend);

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

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

// 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'
// 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 'tgaPenSquare'
// Do nothing if arguments are invalid
void TGAPencilSetShapeSquare(TGAPencil *pen);

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

// 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 absciss
// 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);

#endif

```

2 Code

2.1 tgapaint.c

```

// ***** TGA_PAINT.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);

// Function to calculate the ratio of coverage of pixel 'q' by a square

```



```

// centered on 'p' with a size of 'r'
// Return 1.0 if arguments are invalid
float TGA RatioCoveragePixelSquare(VecFloat *p, float r, VecFloat *q);

// Function to calculate the ratio of coverage of pixel 'q' by a circle
// centered on 'p' with a radius of 'r'
// Return 1.0 if arguments are invalid
float TGA RatioCoveragePixelRound(VecFloat *p, float r, VecFloat *q);

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

// Create a TGA of width dim[0] and height dim[1] and background
// color equal to pixel
// (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 || pixel == 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->_pixels = 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;
    // Allocate memory for the pixels
    ret->_pixels =
        (TGAPixel*)malloc(h->_width * h->_height * sizeof(TGAPixel));
    // If we couldn't allocate memory
    if (ret->_pixels == NULL) {
        // Free the memory for the TGA and its header
        free(ret->_header);
        free(ret);
        // Return NULL
        return NULL;
    }
}

```

```

    }
    // Set a pointer to the pixels
    TGA_Pixel *p = ret->_pixels;
    // For each pixel
    for (int i = 0; i < h->_width * h->_height; ++i) {
        // For each value RGBA
        for (int irgb = 0; irgb < 4; ++irgb)
            // Initialize the value
            p[i]._rgba[irgb] = pixel->_rgba[irgb];
        // Initialize in read-write
        p[i]._readOnly = false;
    }
    // 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));
        // Allocate memory for the pixels
        ret->_pixels =
            (TGA_Pixel*)malloc(ret->_header->_width *
                                ret->_header->_height * sizeof(TGA_Pixel));
        // If we couldn't allocate memory
        if (ret->_pixels == NULL) {
            // Free the memory for the header
            free(ret->_header);
            // Free memory for the cloned TGA
            free(ret);
            // Return NULL
            return NULL;
        }
        // Copy the pixels
        memcpy(ret->_pixels, tga->_pixels,
               ret->_header->_width * ret->_header->_height * sizeof(TGA_Pixel));
    }
    // 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 pixels
TGAFreePixel(&((*tga)->_pixels));
// 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);
    // 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)->_pixels = 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);
// Allocate memory for the pixels
(*tga)->_pixels =
    (TGAPixel*)malloc(h->_width * h->_height * sizeof(TGAPixel));
// If we couldn't allocate memory
if ((*tga)->_pixels == NULL) {
    // Stop here
    TGAFree(tga);
    fclose(fptr);
    return 2;
}
// Set a pointer to the pixel
TGAPixel *pix = (*tga)->_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;
// 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->_pixels == NULL)
        return 2;
    // Open the file

```

```

FILE *fptr = fopen(fileName,"w");
// If we couldn'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->_pixels[i]._rgba[2], fptr);
    putc(tga->_pixels[i]._rgba[1], fptr);
    putc(tga->_pixels[i]._rgba[0], fptr);
    putc(tga->_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);
}

// Get a pointer to the pixel at coord (x,y) = (pos[0],pos[1])

```

```

// Return NULL in case of invalid arguments
TGA Pixel* TGAGetPix(TGA *tga, VecShort *pos) {
    // Check arguments
    if (tga == NULL || pos == NULL ||
        tga->_pixels == NULL || tga->_header == NULL)
        return NULL;
    if (VecGet(pos, 0) < 0 || VecGet(pos, 0) >= tga->_header->_width ||
        VecGet(pos, 1) < 0 || VecGet(pos, 1) >= tga->_header->_height)
        return NULL;
    // Set a pointer to the pixels
    TGA Pixel *p = tga->_pixels;
    // Calculate the index of the requested pixel
    int i = VecGet(pos, 1) * tga->_header->_width + 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'
// Do nothing in case of invalid arguments
void TGASetPix(TGA *tga, VecShort *pos, TGA Pixel *pix) {
    // Check arguments
    if (tga == NULL || pos == NULL || pix == NULL ||
        tga->_pixels == NULL || tga->_header == NULL)
        return;
    // Set a pointer to the pixels
    TGA Pixel *p = TGAGetPix(tga, pos);
    // If the pixel is not null and not in read only mode
    if (p != NULL && TGA PixelIsReadOnly(p) == false)
        // Set the value of the pixel
        memcpy(p, pix, sizeof(TGA Pixel));
}

// Draw one stroke at 'pos' with 'pen'
// Don't do anything in case of invalid arguments
void TGASrokePix(TGA *tga, VecFloat *pos, TGAPencil *pen) {
    // Check arguments
    if (tga == NULL || pos == NULL || pen == NULL ||
        tga->_pixels == NULL || tga->_header == NULL) return;
    // If the shape of the pencil is pixel
    if (pen->_shape == tgaPenPixel) {
        // 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 current pixel of the tga
        TGA Pixel *pixTga = TGAGetPix(tga, q);
        // If the pixel is not in read only mode
        if (TGA PixelIsReadOnly(pixTga) == false) {
            // Get the current pixel of the pencil
            TGA Pixel *pixPen = TGAPencilGetPixel(pen);
            // Get a blend of colors according to pen opacity
            TGA Pixel *pix = TGA BlendPixel(pixTga, pixPen,
                (float)(pixPen->_rgba[3]) / 255.0);
            // Correct opacity
            if (pix->_rgba[3] < 255 - pixPen->_rgba[3])
                pix->_rgba[3] += pixPen->_rgba[3];
            else
                pix->_rgba[3] = 255;
            // Set the color of the current pixel

```

```

        memcpy(pixTga, pix, sizeof(TGAPixel));
        // Free the memory used by the pixel from the pencil
        TGAFreePixel(&pixPen);
        TGAFreePixel(&pix);
        VecFree(&q);
    }
    // Else, if the shape of the pencil is square or round
} else if (pen->_shape == tgaPenRound ||
pen->_shape == tgaPenSquare) {
    // Set a pointer to pixels
    TGAPixel *pixels = tga->_pixels;
    // Get the current color of the pencil
    TGAPixel *pix = TGAPencilGetPixel(pen);
    // Declare variable for coordinates of pixel
    VecFloat *p = VecFloatCreate(2);
    if (p == NULL) {
        return;
    }
    // Calculate the radius of the area affected by the pencil
    float r = pen->_thickness * 0.5;
    // For each pixel in the area affected by the pencil
    for (VecSet(p, 0, VecGet(pos, 0) - r);
        VecGet(p, 0) < VecGet(pos, 0) + r + TGA_EPSILON;
        VecSet(p, 0, VecGet(p, 0) + 1.0)) {
        for (VecSet(p, 1, VecGet(pos, 1) - r);
            VecGet(p, 1) < VecGet(pos, 1) + r + TGA_EPSILON;
            VecSet(p, 1, VecGet(p, 1) + 1.0)) {
            // Declare a variable for the integer position of the
            // current pixel
            VecShort *q = VecShortCreate(2);
            if (q == NULL) {
                VecFree(&p);
                VecFree(&q);
                return;
            }
            VecSet(q, 0, (short)floor(VecGet(p, 0)));
            VecSet(q, 1, (short)floor(VecGet(p, 1)));
            // If the current pixel is in the TGA
            if (VecGet(q, 0) >= 0 && VecGet(q, 0) < tga->_header->_width &&
                VecGet(q, 1) >= 0 && VecGet(q, 1) < tga->_header->_height) {
                // Calculate the distance of the current pixel to
                // the center of the pencil
                float l = VecDist(p, pos);
                // If the pencil is squared, or round and current pixel is
                // in the pencil area
                if ((pen->_shape == tgaPenRound && floor(l) <= floor(r)) ||
                    pen->_shape == tgaPenSquare) {
                    // Calculate the index of the current pixel
                    int iPix = VecGet(q, 1) * tga->_header->_width +
                        VecGet(q, 0);
                    // If the pen doesn't use antialias
                    if (pen->_antialias == false) {
                        // Set the value of the pixel
                        memcpy(pixels + iPix, pix, sizeof(TGAPixel));
                    }
                    // Else, if the pencil uses antialias
                } else {
                    // Declare a variable to calculate the coverage ratio
                    float ratio = 1.0;
                    // Declare a variable to calculate the coordinates of the
                    // bottom left of the current pixel
                    VecFloat *qf = VecFloatCreate(2);
                    if (qf == NULL) {

```



```

        TGAFreePixel(&pix);
        VecFree(&p);
        return;
    }
    VecSet(qf, 0, floor(VecGet(p, 0)));
    VecSet(qf, 1, floor(VecGet(p, 1)));
    // If the pencil is square
    if (pen->_shape == tgaPenSquare) {
        // Calculate the coverage ratio
        ratio = TGA RatioCoveragePixelSquare(pos, r, qf);
    } // Else, if the pencil is round
    } else if (pen->_shape == tgaPenRound) {
        // Calculate the coverage ratio
        ratio = TGA RatioCoveragePixelRound(pos, r, qf);
    }
    // Get a pointer to the current pixel
    TGAPixel *curPix = TGAGetPix(tga, q);
    // If the pointer is not null
    if (curPix != NULL) {
        // Blend the current pixel with the pixel from
        // the pencil
        TGAPixel *blendPix = TGABlendPixel(curPix, pix, ratio);
        // If the blended pixel is not null
        if (blendPix != NULL) {
            // Set the current pixel to the blended pixel
            memcpy(pixels + iPix, blendPix, sizeof(TGAPixel));
            // Free memory used by the blended pixel
            TGAFreePixel(&blendPix);
        }
    }
    // Free memory
    VecFree(&qf);
}
}
// Free memory
VecFree(&q);
}
}
// Free the memory used by the pixel from the pencil
TGAFreePixel(&pix);
VecFree(&p);
}
}

// 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 TGADrawCurve(TGA *tga, BCurve *curve, TGAPencil *pen) {
    // Check arguments
    if (tga == NULL || curve == NULL || pen == NULL ||
        BCurveDim(curve) != 2 || 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 = 0.5 / l;
    // Declare the parameter of the curve
    float t = 0.0;
    // 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
    VecShort *prevPos = VecShortCreate(2);
    if (prevPos == NULL)
        return;
    for (int dim = 2; dim--;)
        VecSet(prevPos, dim, (short)floor(VecGet(curve->_ctrl[0], dim)));
    // Stroke the first pixel
    TGASrokePix(tga, curve->_ctrl[0], pen);
    // While we haven't reached the end of the curve
    while (t <= 1.0) {
        // Calculate the current position on the curve
        VecFree(&pos);
        pos = BCurveGet(curve, t);
        // If the current position is not on the same pixel as previously
        // stroke
        if ((short)floor(VecGet(pos, 0)) != VecGet(prevPos, 0) ||
            (short)floor(VecGet(pos, 1)) != VecGet(prevPos, 1)) {
            // Set the blend value of the pencil to calculate the pencil
            // current color
            TGAPencilSetBlend(pen, t);
            // Stroke the pixel
            TGASrokePix(tga, pos, pen);
            // Update the position of the last stroke pixel
            for (int dim = 2; dim--;)
                VecSet(prevPos, dim, (short)floor(VecGet(pos, dim)));
        }
        // Move along the curve by dt
        t += dt;
    }
    // If the last pixel hasn't been stroke
    if ((short)floor(VecGet(curve->_ctrl[curve->_order], 0)) != VecGet(prevPos, 0) ||
        (short)floor(VecGet(curve->_ctrl[curve->_order], 1)) != VecGet(prevPos, 1))
        // Stroke the last pixel
        TGASrokePix(tga, curve->_ctrl[curve->_order], pen);
    // Free memory
    VecFree(&pos);
    VecFree(&prevPos);
}

// 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 BCurves equivalent to the Shapoid
    GSet *set = ShapoidGetApproxBCurve2D(s);
    // If the set is not empty
    if (set != NULL) {
        // Get the first curve
        BCurve *curve = (BCurve*)(GSetPop(set));
        // While there is a curve to draw
        while (curve != NULL) {
            // Draw the curve
            TGA_DrawCurve(tga, curve, pen);
            // Free memory used by the curve
            BCurveFree(&curve);
            // Get the next curve
            curve = (BCurve*)(GSetPop(set));
        }
        GSetFree(&set);
    }
}

// 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;
// Get the bounding box
Shapoid *bounding = ShapoidGetBoundingBox(s);
// If we could get the bounding box
if (bounding != NULL) {
    // 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
                TGASTrokePix(tga, pos, pen);
            }
        }
    }
    // Free memory
    ShapoidFree(&bounding);
    VecFree(&to);
    VecFree(&pos);
}

// 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(boundingBox->_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(boundingBox->_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(boundingBox->_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(boundingBox->_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 cursors
// It's the upper left corner of the bounding box minus the height
// of one character
VecCopy(cursor, boundingBox->_pos);
VecOp(cursor, 1.0, boundingBox->_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 absciss 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 absciss to the next multiple
    // of the tab parameter
    l = TGAFontGetNextPosByTab(font, l);
    VecCopy(cursor, bbox->_pos);
    VecOp(cursor, 1.0, bbox->_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, bbox->_pos);
    VecOp(cursor, 1.0, bbox->_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 absciss 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(&bbox);
}

// 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;
    // Set a pointer to the requested character's definition
    TGACHar *ch = font->_char + c;
    // Declare a variable to memorize the angle between the absciss
    // and the right direction of the font
    float theta = TGAFontGetAngleWithAbciss(font);
    // For each curve in the character
    for (int iCurve = 0; iCurve < ch->_nbCurve; ++iCurve) {
        // Clone the curve to Set a pointer to the current curve
        BCurve *curve = BCurveClone(ch->_curves[iCurve]);
        if (curve == NULL)

```

```

        return;
    // Scale the curve
    VecFloat *scale = VecGetOp(font->_scale, font->_size, NULL, 0.0);
    if (scale == NULL)
        return;
    BCurveScale(curve, scale);
    // Rotate the curve
    BCurveRot2D(curve, theta);
    // Translate the curve
    BCurveTranslate(curve, pos);
    // Draw the curve
    TGADrawCurve(tga, curve, pen);
    // Free memory
    BCurveFree(&curve);
    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 TGAFreePixel(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* TGABlendPixel(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;
}

// Create a default TGAPencil with all color set to transparent
// solid mode, thickness = 1.0, square shape, 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
        TGAFreePixel(&pixel);
        // Set the default value of the pencil
        ret->_activeColor = 0;
        ret->_modeColor = tgaPenSolid;
        ret->_shape = tgaPenSquare;
        ret->_blendColor[0] = 0;
        ret->_blendColor[1] = 1;
        ret->_blend = 0.0;
        ret->_thickness = 1.0;
        ret->_antialias = false;
    }
}

```

```

    // Return the new pencil
    return ret;
}

// Free the memory used by the TGAPencil 'pen'
void TGAFreePencil(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 = TGAPencil();
    // If we could get a pencil
    if (ret != NULL) {
        // Select the first color
        TGAPencilSetColor(ret, 0);
        // Get a black pixel
        TGAPixel *pixel = TGAGetBlackPixel();
        // If we couldn't get the pixel
        if (pixel == NULL) {
            // Free memory
            TGAFreePencil(&ret);
            // Return NULL
            return NULL;
        }
        // Set the color to the black pixel
        TGAPencilSetColor(ret, pixel);
        // Free memory used by the pixel
        TGAFreePixel(&pixel);
    }
    // Return the new pencil
    return ret;
}

// Select the active color of TGAPencil 'pen' to 'iCol'
// Do nothing if arguments are invalid
void TGAPencilSetColor(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 (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'
// Do nothing if arguments are invalid
void TGAPencilSetThickness(TGAPencil *pen, float v) {
    // Check arguments
    if (pen == NULL || v < 0.0)
        return;
    // 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;
    // Set the 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 'tgaPenSquare'
// Do nothing if arguments are invalid
void TGAPencilSetShapeSquare(TGAPencil *pen) {
    // Check arguments
    if (pen == NULL)
        return;
    // Set the shape
    pen->_shape = tgaPenSquare;
}

// Set the shape of the TGAPencil 'pen' to 'tgaPenRound'
// Do nothing if arguments are invalid
void TGAPencilSetShapeRound(TGAPencil *pen) {
    // Check arguments
    if (pen == NULL)
        return;
    // Set the shape
    pen->_shape = tgaPenRound;
}

// 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;
}

```

```

}

// 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);
    }
}

// Function to calculate the ratio of coverage of pixel 'q' by a square
// centered on 'p' with a size of 'r'
// Return 1.0 if arguments are invalid
float TGAARatioCoveragePixelSquare(VecFloat *p, float r, VecFloat *q) {
    float ratio = 1.0;
    // Check arguments
    if (p == NULL || q == NULL)
        return ratio;
    // Get the intersecting box
    float box[4];

```

```

    box[0] = (VecGet(p, 0) - r < VecGet(q, 0) ?
        VecGet(q, 0) : VecGet(p, 0) - r);
    box[1] = (VecGet(p, 1) - r < VecGet(q, 1) ?
        VecGet(q, 1) : VecGet(p, 1) - r);
    box[2] = (VecGet(p, 0) + r > VecGet(q, 0) + 1.0 ?
        VecGet(q, 0) + 1.0 : VecGet(p, 0) + r);
    box[3] = (VecGet(p, 1) + r > VecGet(q, 1) + 1.0 ?
        VecGet(q, 1) + 1.0 : VecGet(p, 1) + r);
    // The ratio is equal to the area of the intersecting box because the
    // pixel area is 1
    ratio = (box[2] - box[0]) * (box[3] - box[1]);
    // Return the ratio
    return ratio;
}

// Function to calculate the ratio of coverage of pixel 'q' by a circle
// centered on 'p' with a radius of 'r'
// Return 1.0 if arguments are invalid
float TGA RatioCoveragePixelRound(VecFloat *p, float r, VecFloat *q) {
    float ratio = 1.0;
    // Check arguments
    if (p == NULL || q == NULL)
        return ratio;
    // Calculate the ratio by checking a grid of 100 points inside
    // the pixel
    // Declare variables for the calcul
    float delta = 0.1;
    float dp[2];
    float sum = 0.0;
    // For each point
    for (dp[0] = 0.0; dp[0] < 1.0; dp[0] += delta) {
        for (dp[1] = 0.0; dp[1] < 1.0; dp[1] += delta) {
            // Calculate the distance of this point to the center of
            // the circle
            float l = sqrt(pow(VecGet(p, 0) - (VecGet(q, 0) + dp[0]), 2.0) +
                pow(VecGet(p, 1) - (VecGet(q, 1) + dp[1]), 2.0));
            // If the point is in the circle
            if (l <= r) {
                // Increment the number of points inside the circle
                sum += 1.0;
            }
        }
    }
    // The ratio is the number of points divided by the total number of
    // points
    ratio = sum / pow(1.0 / delta, 2.0);
    // Return the ratio
    return ratio;
}

// 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;
}

```

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 = tgaFontAnchorTopLeft
// _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);
        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]._nbCurve = 0;
        // 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--;)
        for (int iCurve = (*font)->_char[iChar]._nbCurve; iCurve--;)
            BCurveFree((*font)->_char[iChar]._curves + iCurve);
    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 absciss
// 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
VecSet(res->_pos, 1, -height);
switch (font->_anchor) {
    case tgaFontAnchorTopLeft:
        break;
    case tgaFontAnchorTopCenter:
        VecSet(res->_pos, 0, -0.5 * VecGet(dim, 0));
        break;
    case tgaFontAnchorTopRight:
        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:
    VecSet(res->_pos, 1, VecGet(res->_pos, 1) + VecGet(dim, 1));
    break;
case tgaFontAnchorBottomCenter:
    VecSet(res->_pos, 1, VecGet(res->_pos, 1) + VecGet(dim, 1));
    VecSet(res->_pos, 0, -0.5 * VecGet(dim, 0));
    break;
case tgaFontAnchorBottomRight:
    VecSet(res->_pos, 1, VecGet(res->_pos, 1) + VecGet(dim, 1));
    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, float *c) {
    for (int iCurve = ch->_nbCurve; iCurve--;) {
        ch->_curves[iCurve] = BCurveCreate(3, 2);
        if (ch->_curves[iCurve] != NULL)
            for (int iCtrl = 4; iCtrl--;)
                for (int dim = 2; dim--;)
                    VecSet(ch->_curves[iCurve]->_ctrl[iCtrl], dim,
                        c[iCurve * 8 + iCtrl * 2 + dim]);
    }
}

// Create the curves of each characters for the default font
void TGAFontCreateDefault(TGAFont *font) {
    TGAChar *ch = NULL;
    ch = font->_char + 'A';
    ch->_nbCurve = 3;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 4;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 4;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 5;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 5;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 3;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 5;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 3;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 3;
    TGAFontInitChar(ch,

```

```

(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';
ch->_nbCurve = 3;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 3;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 3;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 5;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 5;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;

```

```

TGAFontInitChar(ch,
(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';
ch->_nbCurve = 3;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 3;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 5;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 3;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 6;
TGAFontInitChar(ch,
(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';

```

```

ch->_nbCurve = 3;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 5;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 6;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 6;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 5;
TGAFontInitChar(ch,
(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 + '!';
ch->_nbCurve = 3;
TGAFontInitChar(ch,
(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 + '"';
    ch->_nbCurve = 2;
    TGAFontInitChar(ch,
        (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 + '\';
    ch->_nbCurve = 1;
    TGAFontInitChar(ch,
        (float[]){
            0.25,1.00,0.25,1.00,0.25,0.49,0.00,0.50
        });
    ch = font->_char + '#';
    ch->_nbCurve = 4;
    TGAFontInitChar(ch,
        (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 + '$';
    ch->_nbCurve = 6;
    TGAFontInitChar(ch,
        (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 + '%';
    ch->_nbCurve = 9;
    TGAFontInitChar(ch,
        (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 + '&';
    ch->_nbCurve = 6;
    TGAFontInitChar(ch,
        (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 + '(';
    ch->_nbCurve = 1;

```

```

TGAFontInitChar(ch,
    (float[]){
        1.00,1.00,0.75,0.75,0.75,0.25,1.00,0.00
    });
ch = font->_char + ')';
ch->_nbCurve = 1;
TGAFontInitChar(ch,
    (float[]){
        0.00,1.00,0.25,0.75,0.25,0.25,0.00,0.00
    });
ch = font->_char + '=';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
    (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 + '~';
ch->_nbCurve = 1;
TGAFontInitChar(ch,
    (float[]){
        0.00,0.50,0.33,0.75,0.66,0.25,1.00,0.50
    });
ch = font->_char + '`';
ch->_nbCurve = 1;
TGAFontInitChar(ch,
    (float[]){
        0.75,1.00,0.75,1.00,0.75,0.49,1.00,0.50
    });
ch = font->_char + '{';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
    (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 + '}';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
    (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 + '*';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
    (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 + '+';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
    (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 + '<';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
    (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 + '>';
    ch->_nbCurve = 2;
    TGAFontInitChar(ch,
        (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 + '?';
    ch->_nbCurve = 5;
    TGAFontInitChar(ch,
        (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 + '.';
    ch->_nbCurve = 2;
    TGAFontInitChar(ch,
        (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 + ',';
    ch->_nbCurve = 1;
    TGAFontInitChar(ch,
        (float[]){
            0.25,0.18,0.25,0.18,0.25,-0.33,0.00,-0.32
        });
    ch = font->_char + '/';
    ch->_nbCurve = 1;
    TGAFontInitChar(ch,
        (float[]){
            1.00,1.00,1.00,1.00,0.00,0.00,0.00,0.00
        });
    ch = font->_char + '\\';
    ch->_nbCurve = 1;
    TGAFontInitChar(ch,
        (float[]){
            0.00,1.00,0.00,1.00,1.00,0.00,1.00,0.00
        });
    ch = font->_char + '[';
    ch->_nbCurve = 3;
    TGAFontInitChar(ch,
        (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 + ']';
    ch->_nbCurve = 3;
    TGAFontInitChar(ch,
        (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 + '-';
    ch->_nbCurve = 1;

```

```

TGAFontInitChar(ch,
    (float[]){
        0.00,0.50,0.00,0.50,1.00,0.50,1.00,0.50
    });
ch = font->_char + '|';
ch->_nbCurve = 1;
TGAFontInitChar(ch,
    (float[]){
        0.50,1.00,0.50,1.00,0.50,0.00,0.50,0.00
    });
ch = font->_char + '_';
ch->_nbCurve = 1;
TGAFontInitChar(ch,
    (float[]){
        0.00,0.00,0.00,0.00,1.00,0.00,1.00,0.00,
    });
ch = font->_char + ',';
ch->_nbCurve = 3;
TGAFontInitChar(ch,
    (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 + ':';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
    (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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
    (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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
    (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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
    (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';
ch->_nbCurve = 4;

```

```

TGAFontInitChar(ch,
(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';
ch->_nbCurve = 6;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 6;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 3;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 5;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 5;
TGAFontInitChar(ch,
(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';
    ch->_nbCurve = 4;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 6;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 5;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 3;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 4;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 5;
    TGAFontInitChar(ch,
        (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';

```

```

ch->_nbCurve = 5;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 5;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 3;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 2;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,
(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';
ch->_nbCurve = 4;
TGAFontInitChar(ch,

```

```

        (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';
    ch->_nbCurve = 3;
    TGAFontInitChar(ch,
        (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';
    ch->_nbCurve = 3;
    TGAFontInitChar(ch,
        (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 + '@';
    ch->_nbCurve = 8;
    TGAFontInitChar(ch,
        (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 + '^';
    ch->_nbCurve = 2;
    TGAFontInitChar(ch,
        (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

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

main.o : main.c tgapaint.h Makefile

```

```

gcc $(OPTIONS) -I$(INCPATH) -c main.c

tgapaint.o : tgapaint.c tgafont.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
    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
    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
TGAFilterGaussBlur(theTGA, 0.5, 2.0);
// Draw a rectangle
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
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
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
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
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
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);
TGAPencilSetAntialias(pen, true);
TGAPencilSetModeColorSolid(pen);
TGAPencilSetThickness(pen, 5.0);
TGADrawCurve(theTGA, curve, pen);
BCurveFree(&curve);
// Print some strings
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;
}
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 *)"ABCDEFGHJ\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 *)"abcdefghijklmnopqrst\nuvwxyz^@", from);
VecSet(from, 0, 5.0); VecSet(from, 1, 152.0);
TGAPrintString(theTGA, pen, font,
    (unsigned char *)"!\"#$%&'()=\\n~`{ }*+<>?,\\n./\\[]-|_:", from);
// Draw some Shapoid
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
TGAPencilSetModeColorBlend(pen, 0, 1);
TGAPencilSetShapePixel(pen);
pix->_rgba[3] = 255;
pix->_rgba[0] = 255; pix->_rgba[1] = 0; pix->_rgba[2] = 0;
TGAPencilSetColor(pen, 0);
TGAPencilSetColor(pen, pix);
pix->_rgba[0] = 0; pix->_rgba[1] = 0; pix->_rgba[2] = 255;
TGAPencilSetColor(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);
    TGAFreePixel(&pix);
    TGAFreePencil(&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):

