TGA

P. Baillehache

June 17, 2017

Contents

Introduction

TGA library is a C library to 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).

It offers functions to access header and pixels values, paint simple geometric shapes (point, line, curve, rectangle, filled rectangle, ellipse and filled ellipse), and apply gaussian blur to the picture.

1 Interface

```
// ------------- TGA.h ----------
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
```

```
#include "string.h"

typedef struct {
  short int _colorMapOrigin;
  short int _colorMapLength;
  short int _xOrigin;
  short int _yOrigin;
  short _width;
  short _height;
  char _idLength;
  char _colorMapType;
  char _dataTypeCode;
  char _colorMapDepth;
  char _bitsPerPixel;
  char _imageDescriptor;
} TGAHeader;

typedef struct {
  unsigned char _rgba[4];
} TGAPixel;

typedef struct {
  TGAHeader *_header;
  // pixels are stored line by line
  TGAPixel *_pixels;
} TGA;

// Print the header of tga on stream
// If tga or stream is null, dont do anything
void TGAPrintHeader(TGA *tga, FILE *stream);

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

// Load tga from the file pointed to by fileName
// If tga already contains a TGA, it is overwritten
// and the memory for header and pixels are not freed
// 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 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);

// Create a tga of width dim[0] and height dim[1] and background
// color equal to pix color
// Return NULL in case of invalid arguments or memory allocation
// failure
// If the file already exists it is overwritten
TGA* TGACreate(short *dim, TGAPixel *pix);

// Set the color of the pixel at coord (x,y) = (pos[0],pos[1]) to the
// color of pix
```

```
// (0,0) is the bottom left corner, x toward right, y toward top
// Don't do anything in case of invalid arguments
void TGASetPix(TGA *tga, short *pos, TGAPixel *pix);

// Get a reference to the pixel at coord (x,y) = (pos[0],pos[1])
// (0,0) is the bottom left corner, x toward right, y toward top
// Return NULL in case of invalid arguments
TGAPixel* TGAGetPix(TGA *tga, short *pos);

// Draw a one pixel line between from and to with color pix
// pixels outside the tga are ignored
// do nothing if arguments are invalid
void TGADrawLine(TGA *tga, short *from, short *to, TGAPixel *pix);

// Draw a one pixel line between from and to with color pixFrom at from
// and color pixTo at to shading progressively in between
// pixels outside the tga are ignored
// do nothing if arguments are invalid
void TGADrawLink(TGA *tga, short *from, short *to,
  TGAPixel *pixFrom, TGAPixel *pixTo);

// Draw a one pixel curve between from and to with color pixFrom
// at from and color pixTo at to shading progressively in between
// and control points ctrlFrom and ctrlTo
// pixels outside the tga are ignored
// do nothing if arguments are invalid
void TGADrawCurve(TGA *tga, short *from, short *to,
  short *ctrlFrom, short *ctrlTo, TGAPixel *pixFrom, TGAPixel *pixTo);

// Draw a one pixel thick rectangle between from and to with color pix
// pixels outside the tga are ignored
// do nothing if arguments are invalid
void TGADrawRect(TGA *tga, short *from, short *to, TGAPixel *pix);

// Fill a rectangle between from and to with color pix
// pixels outside the tga are ignored
// do nothing if arguments are invalid
void TGAFillRect(TGA *tga, short *from, short *to, TGAPixel *pix);

// Draw a one pixel thick ellipse at center of radius r[0],r[1]
// with color pix pixels outside the tga are ignored
// do nothing if arguments are invalid
void TGADrawEllipse(TGA *tga, short *center, short *r, TGAPixel *pix);

// Fill an ellipse at center of radius r[0],r[1] with color pix
// pixels outside the tga are ignored
// do nothing if arguments are invalid
void TGAFillEllipse(TGA *tga, short *center, short *r, TGAPixel *pix);

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

2 Code

```
// ------------- TGA.c ----------
#include "tga.h"

void MergeBytes(TGAPixel *pixel, unsigned char *p, int 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);
  }
}

void TGAPrintHeader(TGA *tga, FILE *stream) {
  if (tga == NULL || stream == NULL) return;
  TGAHeader *h = tga->_header;
  if (h == NULL) return;
  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);
}

void TGAFree(TGA *tga) {
  if (tga->_header != NULL)
    free(tga->_header);
  tga->_header = NULL;
  if (tga->_pixels != NULL)
    free(tga->_pixels);
  tga->_pixels = NULL;
}

int TGALoad(TGA *tga, char *fileName) {
  if (tga == NULL || fileName == NULL) return 7;
  int n = 0, i, j;
  unsigned int bytes2read, skipover = 0;
  unsigned char p[5];
  FILE *fptr = fopen(fileName,"r");
  size_t ret;
  if (fptr == NULL)
    return 1;
  tga->_header = (TGAHeader*)malloc(sizeof(TGAHeader));
  if (tga->_header == NULL) {
    fclose(fptr);
    return 2;
  }
  TGAHeader *h = tga->_header;
  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);
tga->_pixels =
  (TGAPixel*)malloc(h->_width * h->_height * sizeof(TGAPixel));
if (tga->_pixels == NULL) {
  fclose(fptr);
  return 2;
}
TGAPixel *pix = tga->_pixels;
for (i = 0; i < h->_width * h->_height; ++i) {
  for (int irgb = 0; irgb < 4; ++irgb)
    pix[i]._rgba[irgb] = 0;
}
if (h->_dataTypeCode != 2 && h->_dataTypeCode != 10) {
  fclose(fptr);
  return 3;
}
if (h->_bitsPerPixel != 16 &&
  h->_bitsPerPixel != 24 &&
  h->_bitsPerPixel != 32) {
  fclose(fptr);
  return 4;
}
if (h->_colorMapType != 0 &&
  h->_colorMapType != 1) {
  fclose(fptr);
  return 5;
}
skipover += h->_idLength;
skipover += h->_colorMapType * h->_colorMapLength;
fseek(fptr,skipover,SEEK_CUR);
bytes2read = h->_bitsPerPixel / 8;
while (n < h->_width * h->_height) {
  if (h->_dataTypeCode == 2) {
    if (fread(p, 1, bytes2read, fptr) != bytes2read) {
      fclose(fptr);
      return 6;
    }
    MergeBytes(&(pix[n]), p, bytes2read);
    ++n;
  } else if (h->_dataTypeCode == 10) {
    if (fread(p, 1, bytes2read + 1, fptr) != bytes2read + 1) {
      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)
            return 6;
          MergeBytes(&(pix[n]), p, bytes2read);
          ++n;
        }
      }
    }
  }
  fclose(fptr);
  ret = ret;
  return 0;
}

int TGASave(TGA *tga, char *fileName) {
   if (tga == NULL || fileName == NULL ||
    tga->_header == NULL || tga->_pixels == NULL)
    return 2;
   int i;
   FILE *fptr = fopen(fileName,"w");
   if (fptr == NULL)
      return 1;
   putc(0,fptr);
   putc(0,fptr);
   putc(2,fptr);
   putc(0,fptr);
   putc(0,fptr);
   putc(0,fptr);
   putc(0,fptr);
   putc(0,fptr);
   putc(0,fptr);
   putc(0,fptr);
   putc(0,fptr);
   putc(0,fptr);
   putc((tga->_header->_width & 0x00FF), fptr);
   putc((tga->_header->_width & 0xFF00) / 256, fptr);
   putc((tga->_header->_height & 0x00FF), fptr);
   putc((tga->_header->_height & 0xFF00) / 256, fptr);
   putc(32, fptr);
   putc(0, fptr);
   for (i = 0; i < tga->_header->_height * tga->_header->_width; ++i) {
      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);
   }
   fclose(fptr);
   return 0;
}

TGA* TGACreate(short *dim, TGAPixel *pix) {
  if (dim == NULL) return NULL;
  TGA *ret = (TGA*)malloc(sizeof(TGA));
  if (ret == NULL)
    return NULL;
  ret->_header = NULL;
  ret->_pixels = NULL;
  ret->_header = (TGAHeader*)malloc(sizeof(TGAHeader));
  if (ret->_header == NULL) {
    free(ret);
    return NULL;
  }
  TGAHeader *h = ret->_header;
```

```
  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 = dim[0];
  h->_height = dim[1];
  h->_bitsPerPixel = 32;
  h->_imageDescriptor = 0;
  ret->_pixels =
    (TGAPixel*)malloc(h->_width * h->_height * sizeof(TGAPixel));
  if (ret->_pixels == NULL) {
    free(ret->_header);
    free(ret);
    return NULL;
  }
  TGAPixel *p = ret->_pixels;
  for (int i = 0; i < h->_width * h->_height; ++i) {
    for (int irgb = 0; irgb < 4; ++irgb)
      p[i]._rgba[irgb] = pix->_rgba[irgb];
  }
  return ret;
}

void TGASetPix(TGA *tga, short *pos, TGAPixel *pix) {
  if (tga == NULL || pos == NULL || pix == NULL ||
    tga->_pixels == NULL || tga->_header == NULL) return;
  if (pos[0] < 0 || pos[0] >= tga->_header->_width ||
    pos[1] < 0 || pos[1] >= tga->_header->_height) return;
  TGAPixel *p = tga->_pixels;
  int i = pos[1] * tga->_header->_width + pos[0];
  memcpy(&(p[i]), pix, sizeof(TGAPixel));
}

TGAPixel* TGAGetPix(TGA *tga, short *pos) {
  if (tga == NULL || pos == NULL ||
    tga->_pixels == NULL || tga->_header == NULL) return NULL;
  if (pos[0] < 0 || pos[0] >= tga->_header->_width ||
    pos[1] < 0 || pos[1] >= tga->_header->_height) return NULL;
  TGAPixel *p = tga->_pixels;
  int i = pos[1] * tga->_header->_width + pos[0];
  return &(p[i]);
}

void TGADrawLine(TGA *tga, short *from, short *to, TGAPixel *pix) {
  if (tga == NULL || from == NULL || to == NULL || pix == NULL ||
    tga->_header == NULL || tga->_pixels == NULL)
    return;
  short range[2];
  range[0] = to[0] - from[0];
  range[1] = to[1] - from[1];
  short pos[2];
  pos[0] = from[0];
  pos[1] = from[1];
  int step = 0;
  if (abs(range[0]) > abs(range[1])) {
    if (range[0] > 0)
      step = 1;
    else
```

```
      step = -1;
    while (pos[0] != to[0]) {
      pos[1] = from[1] +
        (short)round((float)(range[1]) *
        (float)(pos[0] - from[0]) / (float)(range[0]));
      TGASetPix(tga, pos, pix);
      pos[0] += step;
    }
  } else {
    if (range[1] > 0)
      step = 1;
    else
      step = -1;
    while (pos[1] != to[1]) {
      pos[0] = from[0] +
        (short)round((float)(range[0]) * (float)(pos[1] - from[1]) / (float)(range[1]));
      TGASetPix(tga, pos, pix);
      pos[1] += step;
    }
  }
  TGASetPix(tga, to, pix);
}

void TGADrawLink(TGA *tga, short *from, short *to,
  TGAPixel *pixFrom, TGAPixel *pixTo) {
  if (tga == NULL || from == NULL || to == NULL || pixFrom == NULL ||
    pixTo == NULL || tga->_header == NULL || tga->_pixels == NULL)
    return;
  TGAPixel pix;
  short range[2];
  range[0] = to[0] - from[0];
  range[1] = to[1] - from[1];
  short pos[2];
  pos[0] = from[0];
  pos[1] = from[1];
  int step = 0;
  if (abs(range[0]) > abs(range[1])) {
    if (range[0] > 0)
      step = 1;
    else
      step = -1;
    while (pos[0] != to[0]) {
      float t = (float)(pos[0] - from[0]) / (float)(to[0] - from[0]);
      for (int irgb = 0; irgb < 4; ++irgb)
        pix._rgba[irgb] = (unsigned char)round((1.0 - t) *
          (float)(pixFrom->_rgba[irgb]) +
          t * (float)(pixTo->_rgba[irgb]));
      pos[1] = from[1] +
        (short)round((float)(range[1]) *
        (float)(pos[0] - from[0]) / (float)(range[0]));
      TGASetPix(tga, pos, &pix);
      pos[0] += step;
    }
  } else {
    if (range[1] > 0)
      step = 1;
    else
      step = -1;
    while (pos[1] != to[1]) {
      float t = (float)(pos[1] - from[1]) / (float)(to[1] - from[1]);
      for (int irgb = 0; irgb < 4; ++irgb)
      pix._rgba[irgb] = (unsigned char)round((1.0 - t) *
```

```
        (float)(pixFrom->_rgba[irgb]) +
        t * (float)(pixTo->_rgba[irgb]));
      pos[0] = from[0] +
        (short)round((float)(range[0]) *
        (float)(pos[1] - from[1]) / (float)(range[1]));
      TGASetPix(tga, pos, &pix);
      pos[1] += step;
    }
  }
  TGASetPix(tga, to, pixTo);
}

void TGACurvePos(short *from, short *to, short *ctrlFrom,
  short *ctrlTo, float t, short *pos) {
  float A[2];
  A[0] = (1.0 - t) * (float)(from[0]) + t * (float)(ctrlFrom[0]);
  A[1] = (1.0 - t) * (float)(from[1]) + t * (float)(ctrlFrom[1]);
  float B[2];
  B[0] = (1.0 - t) * (float)(ctrlTo[0]) + t * (float)(to[0]);
  B[1] = (1.0 - t) * (float)(ctrlTo[1]) + t * (float)(to[1]);
  float C[2];
  C[0] = (1.0 - t) * (float)(ctrlFrom[0]) + t * (float)(ctrlTo[0]);
  C[1] = (1.0 - t) * (float)(ctrlFrom[1]) + t * (float)(ctrlTo[1]);
  float D[2];
  D[0] = (1.0 - t) * A[0] + t * C[0];
  D[1] = (1.0 - t) * A[1] + t * C[1];
  float E[2];
  E[0] = (1.0 - t) * C[0] + t * B[0];
  E[1] = (1.0 - t) * C[1] + t * B[1];
  float F[2];
  F[0] = (1.0 - t) * D[0] + t * E[0];
  F[1] = (1.0 - t) * D[1] + t * E[1];
  pos[0] = (short)round(F[0]);
  pos[1] = (short)round(F[1]);
}

void TGADrawCurve(TGA *tga, short *from, short *to,
  short *ctrlFrom, short *ctrlTo, TGAPixel *pixFrom, TGAPixel *pixTo) {
  if (tga == NULL || from == NULL || to == NULL || pixFrom == NULL ||
    ctrlFrom == NULL || ctrlTo == NULL ||
    pixTo == NULL || tga->_header == NULL || tga->_pixels == NULL)
    return;
  TGAPixel pix;
  short range[4];
  range[0] = from[0]; range[1] = from[1];
  range[2] = from[0]; range[3] = from[1];
  if (range[0] > to[0]) range[0] = to[0];
  if (range[1] > to[1]) range[1] = to[1];
  if (range[2] < to[0]) range[2] = to[0];
  if (range[3] < to[1]) range[3] = to[1];
  if (range[0] > ctrlTo[0]) range[0] = ctrlTo[0];
  if (range[1] > ctrlTo[1]) range[1] = ctrlTo[1];
  if (range[2] < ctrlTo[0]) range[2] = ctrlTo[0];
  if (range[3] < ctrlTo[1]) range[3] = ctrlTo[1];
  if (range[0] > ctrlFrom[0]) range[0] = ctrlFrom[0];
  if (range[1] > ctrlFrom[1]) range[1] = ctrlFrom[1];
  if (range[2] < ctrlFrom[0]) range[2] = ctrlFrom[0];
  if (range[3] < ctrlFrom[1]) range[3] = ctrlFrom[1];
  short l = 2 * (range[2] - range[0]) + 2 * (range[3] - range[1]);
  float dt = 1.0 / (float)l;
  float t = 0.0;
  short pos[2];
```

```
  pos[0] = from[0]; pos[1] = from[1];
  short prevPos[2];
  prevPos[0] = from[0]; prevPos[1] = from[1];
  TGASetPix(tga, from, pixFrom);
  while (t <= 1.0) {
    TGACurvePos(from, to, ctrlFrom, ctrlTo, t, pos);
    if (pos[0] != prevPos[0] || pos[1] != prevPos[1]) {
      for (int irgb = 0; irgb < 4; ++irgb)
        pix._rgba[irgb] = (unsigned char)round((1.0 - t) *
          (float)(pixFrom->_rgba[irgb]) +
          t * (float)(pixTo->_rgba[irgb]));
      TGASetPix(tga, pos, &pix);
      prevPos[0] = pos[0]; prevPos[1] = pos[1];
    }
    t += dt;
  }
  TGASetPix(tga, to, pixTo);
}

void TGADrawRect(TGA *tga, short *from, short *to, TGAPixel *pix) {
  if (tga == NULL || from == NULL || to == NULL || pix == NULL ||
    tga->_header == NULL || tga->_pixels == NULL)
    return;
  short cornA[2];
  short cornB[2];
  cornA[0] = from[0]; cornA[1] = from[1];
  cornB[0] = from[0]; cornB[1] = to[1];
  TGADrawLine(tga, cornA, cornB, pix);
  cornA[0] = from[0]; cornA[1] = from[1];
  cornB[0] = to[0]; cornB[1] = from[1];
  TGADrawLine(tga, cornA, cornB, pix);
  cornA[0] = to[0]; cornA[1] = to[1];
  cornB[0] = to[0]; cornB[1] = from[1];
  TGADrawLine(tga, cornA, cornB, pix);
  cornA[0] = to[0]; cornA[1] = to[1];
  cornB[0] = from[0]; cornB[1] = to[1];
  TGADrawLine(tga, cornA, cornB, pix);
}

void TGAFillRect(TGA *tga, short *from, short *to, TGAPixel *pix) {
  if (tga == NULL || from == NULL || to == NULL || pix == NULL ||
    tga->_header == NULL || tga->_pixels == NULL)
    return;
  short cornA[2];
  short cornB[2];
  if (from[0] < to[0]) {
    cornA[0] = from[0]; cornB[0] = to[0];
  } else {
    cornA[0] = to[0]; cornB[0] = from[0];
  }
  if (from[1] < to[1]) {
    cornA[1] = from[1]; cornB[1] = to[1];
  } else {
    cornA[1] = to[1]; cornB[1] = from[1];
  }
  short p[2];
  for (p[0] = cornA[0]; p[0] < cornB[0]; ++(p[0]))
    for (p[1] = cornA[1]; p[1] < cornB[1]; ++(p[1]))
      TGASetPix(tga, p, pix);
}

void TGADrawEllipse(TGA *tga, short *center, short *r, TGAPixel *pix) {
```

```
  if (tga == NULL || center == NULL || r == NULL || pix == NULL ||
    tga->_header == NULL || tga->_pixels == NULL ||
    r[0] < 0 || r[1] < 0)
    return;
  short cornA[2];
  short cornB[2];
  cornA[0] = center[0] - r[0]; cornA[1] = center[1] - r[1];
  cornB[0] = center[0] + r[0]; cornB[1] = center[1] + r[1];
  short p[2];
  float s = (float)(r[0]) / (float)(r[1]);
  for (p[0] = cornA[0]; p[0] <= cornB[0]; ++(p[0])) {
    for (p[1] = cornA[1]; p[1] <= cornB[1]; ++(p[1])) {
      short d = (short)round(sqrt(pow(p[0] - center[0], 2.0) +
        pow(s * (float)(p[1] - center[1]), 2.0)));
      if (d == r[0]) TGASetPix(tga, p, pix);
    }
  }
}

// Return the value of the gaussian (mean, sigma) at x
float TGAGauss(float x, float mean, float sigma) {
  float a = 1.0 / (sigma * sqrt(2.0 * 3.14159));
  float ret = a * exp(-1.0 * pow(x - mean, 2.0) /
    (2.0 * pow(sigma, 2.0)));
  return ret;
}

void TGAFillEllipse(TGA *tga, short *center, short *r, TGAPixel *pix) {
  if (tga == NULL || center == NULL || r == NULL || pix == NULL ||
    tga->_header == NULL || tga->_pixels == NULL)
    return;
  short cornA[2];
  short cornB[2];
  cornA[0] = center[0] - r[0]; cornA[1] = center[1] - r[1];
  cornB[0] = center[0] + r[0]; cornB[1] = center[1] + r[1];
  short p[2];
  float s = (float)(r[0]) / (float)(r[1]);
  for (p[0] = cornA[0]; p[0] <= cornB[0]; ++(p[0])) {
    for (p[1] = cornA[1]; p[1] <= cornB[1]; ++(p[1])) {
      short d = (short)round(sqrt(pow(p[0] - center[0], 2.0) +
        pow(s * (float)(p[1] - center[1]), 2.0)));
      if (d <= r[0]) TGASetPix(tga, p, pix);
    }
  }
}

void TGAGaussBlur(TGA *tga, float strength, float range) {
  if (tga == NULL || tga->_header == NULL || strength <= 0.0)
    return;
  float *drgb = (float*)malloc(tga->_header->_width *
    tga->_header->_height * 3 * sizeof(float));
  short px[2] = {0, 0};
  int irgb = 0;
  for (px[0] = tga->_header->_width; px[0]--;) {
    for (px[1] = tga->_header->_height; px[1]--;) {
      long int index = 3 * (px[1] * tga->_header->_width + px[0]);
      for (irgb = 3; irgb--;)
        drgb[index + irgb] = 0.0;
    }
  }
  for (px[0] = tga->_header->_width; px[0]--;) {
    for (px[1] = tga->_header->_height; px[1]--;) {
```

```
      long int indexp = 3 * (px[1] * tga->_header->_width + px[0]);
      for (irgb = 3; irgb--;) {
        short qx[2] = {0, 0};
        double sum = 0.0;
        double p = 0.0;
        short from[2] = {0, 0};
        short to[2] = {0, 0};
        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 (qx[0] = from[0]; qx[0] < to[0]; ++(qx[0])) {
          for (qx[1] = from[1]; qx[1] < to[1]; ++(qx[1])) {
            double dist = sqrt(pow(qx[0] - px[0], 2.0) +
              pow(qx[1] - px[1], 2.0));
            if (dist < range) {
              double g = TGAGauss(dist, 0.0, strength);
              sum += g;
              TGAPixel *pixelQ = TGAGetPix(tga, qx);
              p += g * (double)(pixelQ->_rgba[irgb]);
            }
          }
        }
        drgb[indexp + irgb] = p / sum;
      }
    }
  }
  for (px[0] = tga->_header->_width; px[0]--;) {
    for (px[1] = tga->_header->_height; px[1]--;) {
      long int index = 3 * (px[1] * tga->_header->_width + px[0]);
      TGAPixel *pixel = TGAGetPix(tga, px);
      for (irgb = 3; irgb--;) {
        pixel->_rgba[irgb] =
          (unsigned char)round(drgb[index + irgb]);
      }
    }
  }
  free(drgb);
  drgb = NULL;
}
```

3 Makefile

```
OPTIONS_DEBUG=-ggdb -g3 -Wall
OPTIONS_RELEASE=-O3 -s
OPTIONS=$(OPTIONS_RELEASE) -lm

all : main

main: main.o tga.o Makefile
        gcc -o main main.o tga.o $(OPTIONS)

main.o : main.c tga.h Makefile
        gcc -c main.c $(OPTIONS)

tga.o : tga.c tga.h Makefile
        gcc -c tga.c $(OPTIONS)
```

```
clean :
        rm -rf *.o main
```

4 Usage

```
#include "stdio.h"
#include "stdlib.h"
#include "tga.h"

int main() {
  int ret;
  TGA *theTGA;
  // Create the TGA
  short dim[2] = {120, 100};
  TGAPixel pix;
  pix._rgba[0] = pix._rgba[1] = pix._rgba[2] = 255;
  pix._rgba[3] = 255;
  theTGA = TGACreate(dim, &pix);
  if (theTGA == NULL) {
    fprintf(stderr, "Error while creating the tga\n");
    return 1;
  }
  // Set the color of some pixels
  short pos[2];
  pix._rgba[0] = pix._rgba[1] = pix._rgba[2] = 0;
  pos[0] = 60; pos[1] = 50;
  TGASetPix(theTGA, pos, &pix);
  pix._rgba[0] = 255; pix._rgba[1] = 0; pix._rgba[2] = 0;
  pos[0] = 90; pos[1] = 50;
  TGASetPix(theTGA, pos, &pix);
  pix._rgba[0] = 0; pix._rgba[1] = 0; pix._rgba[2] = 255;
  pos[0] = 60; pos[1] = 25;
  TGASetPix(theTGA, pos, &pix);
  pix._rgba[0] = 0; pix._rgba[1] = 255; pix._rgba[2] = 0;
  pos[0] = 30; pos[1] = 75;
  TGASetPix(theTGA, pos, &pix);
  // Draw some lines
  pix._rgba[0] = 0; pix._rgba[1] = 0; pix._rgba[2] = 0;
  short from[2];
  short to[2];
  from[0] = 50; from[1] = 40; to[0] = 50; to[1] = 60;
  TGADrawLine(theTGA, from, to, &pix);
  from[0] = 50; from[1] = 60; to[0] = 70; to[1] = 60;
  TGADrawLine(theTGA, from, to, &pix);
  pix._rgba[0] = 255; pix._rgba[1] = 0; pix._rgba[2] = 255;
  from[0] = -10; from[1] = 50; to[0] = 60; to[1] = -10;
  TGADrawLine(theTGA, from, to, &pix);
  from[0] = 60; from[1] = -10; to[0] = 130; to[1] = 50;
  TGADrawLine(theTGA, from, to, &pix);
  from[0] = 130; from[1] = 50; to[0] = 60; to[1] = 110;
  TGADrawLine(theTGA, from, to, &pix);
  from[0] = 60; from[1] = 110; to[0] = -10; to[1] = 50;
  TGADrawLine(theTGA, from, to, &pix);
  // Draw a rectangle
  pix._rgba[0] = 0; pix._rgba[1] = 255; pix._rgba[2] = 255;
  from[0] = 70; from[1] = 40; to[0] = 100; to[1] = 10;
  TGADrawRect(theTGA, from, to, &pix);
  // Draw a filled rectangle
  pix._rgba[0] = 255; pix._rgba[1] = 255; pix._rgba[2] = 0;
  from[0] = 75; from[1] = 35; to[0] = 95; to[1] = 15;
```

```
  TGAFillRect(theTGA, from, to, &pix);
  // Draw an ellipse
  pix._rgba[0] = 128; pix._rgba[1] = 128; pix._rgba[2] = 128;
  short center[2] = {30, 50};
  short radius[2] = {15, 20};
  TGADrawEllipse(theTGA, center, radius, &pix);
  // Draw a filled ellipse
  pix._rgba[0] = 200; pix._rgba[1] = 200; pix._rgba[2] = 200;
  center[0] = 60; center[1] = 75;
  radius[0] = 25; radius[1] = 10;
  TGAFillEllipse(theTGA, center, radius, &pix);
  // Draw a link
  from[0] = 30; from[1] = 25; to[0] = 90; to[1] = 75;
  TGAPixel pixb;
  pix._rgba[0] = pix._rgba[3] = 255;
  pix._rgba[1] = pix._rgba[2] = 0;
  pixb._rgba[2] = pixb._rgba[3] = 255;
  pixb._rgba[1] = pixb._rgba[0] = 0;
  TGADrawLink(theTGA, from, to, &pix, &pixb);
  // Draw a curve
  short ctrlFrom[2] = {40, 0};
  short ctrlTo[2] = {80, 50};
  TGADrawCurve(theTGA, from, to, ctrlFrom, ctrlTo, &pix, &pixb);
  // Apply gaussian blur
  TGAGaussBlur(theTGA, 0.5, 2.0);
  // 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 1;
  }
  // Print its header on standard output stream
  TGAPrintHeader(theTGA, stdout);
  //Free the tga
  TGAFree(theTGA);
  free(theTGA);
  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:            100
Bits per pixel:    32
Descriptor:        0
```

Resulting image (enlarge):