GenBrush

P. Baillehache

May 1, 2018

Contents

Introduction

GenBrush is a C library providing structures and functions for creating 2D/3D, bitmap/vector, still/animated graphics on various supports.

It can be use as a monolithic library, or as an interface with another graphical library (Cairo/GTK). The choice can be made through a single compilation option.

GenBrush is based on the following model: the scene to be represented graphically is composed of Objects viewed by an Eye, interpreted by a Hand and recreated with a Tool applying an Ink on a Surface made of Layers.

The currently available implementations for these entities are:

- Object: Point (VecFloat), SCurve, Shapoid

- Eye: Orthographic, Isometric

- Hand: Default

- Tool: PixelPlotter

- Ink: Solid

- Surface: Image (TGA), GBApp, GBWidget

- Layer: divided into background, inside and foreground layers; blending modes: Default (overwritting), Normal, Over

For details refer to the following sections.

The library can easily be extended to match the user needs. Foreseen extensions of the entities are for example:

- Object: BBody, ...

- Eye: Perspective, ...

- Hand: Human (approximation of the viewed object), ...

- Tool: Pen, Brush, ...

- Ink: Generic (BBody), UVMapping

- Surface: Full support of TGA, other image formats, innterface with Cairo only, ...

- Layer: other blending modes, ...

A graphic can then be created with the GenBrush library by describing the scene as a set of Pods, which are combinations of an Object, an Eye, a Hand, a Tool, an Ink and a Layer. The rendering of each Pod processes sequentially the original Object through the other entities to generates pixels into the Layer. Once all the Pods have been processed, the Layers are blended to generate RGBA final pixels in the Surface. The scene description can be modified after creation. Pods use references to entities, meaning that if a parameter of, for example, an Ink is modified all the Pods attached to this Ink will be recalculated at the next rendering.

Several rendering of the same scene are optimized to recalculate only the final pixels affected by the entities which has been modified since the previous rendering. It is the reponsibility of the user to notify GenBrush when an entity has been modified.

Finally, the user can apply post processing to the surface at the end of each rendering. Currently implemented post processing are:

- Normalization by hue

The GenBrush library uses the `PBErr`, `PBMath`, `GSet`, `Shapoid`, `BCurve` libraries, and GTK if compiled as an interface for this library.

1 Definitions

1.1 Object

The scene to be represented graphically can be described using Points, SCurves and Shapoids.

1.1.1 Point

Points are a single point in space (any dimensions from 2). After projection through the Eye, the first two dimensions are the coordinates in the Layer, the third dimension is the depth, and the following dimensions are parameters for the other entities. If a dimension is needed but not provided by the Point its value is considered to be 0.0.

1.1.2 SCurve

SCurves are curves in space (any dimensions from 2). After projection through the Eye, the first two dimensions are the coordinates in the Layer, the third dimension is the depth, and the following dimensions are parameters for the other entities. If a dimension is needed but not provided by the Point its value is considered to be 0.0.

1.1.3 Shapoid

Shapoids are surfaces (if dimension equals 2) or volumes (if dimension is greater than 2) in space. After projection through the Eye, the first two dimensions are the coordinates in the Layer, the third dimension is the depth, and the following dimensions are parameters for the other entities. If a dimension is needed but not provided by the Point its value is considered to be 0.0.

1.2 Eye

The Eye create a projection of an Object from the original space coordinates system toward the Layer space coordinates system. The projected Object is always of the same type as the original Object, only its component are

affected by the projection.

The Eye always sees the Object in 3 dimensions. If the Object has only two dimensions, the Eye considers it has a third one with values equal to 0.0. If the Object has more than 3 dimensions, the Eye applies the projection on the first 3 ones and leaves the other dimensions unchanged.

All the Eyes have a translation, a scale and rotation parameters. The first, second and third components of the projected Object are multiplied repectively by the first, second and third components of the scale vector. The rotation angle (in radians) is a 2D rotation applied on the two first components of the scaled projected Object. Finally the translation 2D vector is added to the rotated scaled projected Object. The translation acts as a translation of the viewed object in the plane of the Layer, the scale acts like a zoom on the viewed object, and the rotation like a tilt of the Eye. Given $\overrightarrow{t}$, $\overrightarrow{s}$ and θ the translation, scale and rotation of the Eye the translated rotated scaled version O' of the projected version of Object O can be obtained as follow:

$$O' = \begin{bmatrix} s_x cos\theta & -s_y sin\theta & 0.0 \\ s_x sin\theta & s_y cos\theta & 0.0 \\ 0.0 & 0.0 & s_z \end{bmatrix} O + \begin{pmatrix} t_x \\ t_y \\ 0.0 \end{pmatrix} \tag{1}$$

1.2.1 Orthographic

The Orthographic Eye creates projection of Objects onto planes aligned with the original coordinates system. It has 6 views: front, rear, left, right, top, bottom. For each view the translated rotated scaled projected version O' of the Object O can be obtained as follow:

- front:

$$O' = \begin{bmatrix} s_x cos\theta & -s_y sin\theta & 0.0 \\ s_x sin\theta & s_y cos\theta & 0.0 \\ 0.0 & 0.0 & s_z \end{bmatrix} O + \begin{pmatrix} t_x \\ t_y \\ 0.0 \end{pmatrix} \tag{2}$$

- rear:

$$O' = \begin{bmatrix} -s_x cos\theta & -s_y sin\theta & 0.0 \\ -s_x sin\theta & s_y cos\theta & 0.0 \\ 0.0 & 0.0 & -s_z \end{bmatrix} O + \begin{pmatrix} t_x \\ t_y \\ 0.0 \end{pmatrix} \tag{3}$$

- top:

$$O' = \begin{bmatrix} s_x cos\theta & 0.0 & -s_y sin\theta \\ s_x sin\theta & 0.0 & s_y cos\theta \\ 0.0 & -s_z & 0.0 \end{bmatrix} O + \begin{pmatrix} t_x \\ t_y \\ 0.0 \end{pmatrix} \tag{4}$$

- bottom:

$$O' = \begin{bmatrix} s_x cos\theta & 0.0 & s_y sin\theta \\ s_x sin\theta & 0.0 & -s_y cos\theta \\ 0.0 & s_z & 0.0 \end{bmatrix} O + \begin{pmatrix} t_x \\ t_y \\ 0.0 \end{pmatrix} \tag{5}$$

- left:

$$O' = \begin{bmatrix} 0.0 & -s_y sin\theta & -s_x cos\theta \\ 0.0 & s_y cos\theta & -s_x sin\theta \\ s_z & 0.0 & 0.0 \end{bmatrix} O + \begin{pmatrix} t_x \\ t_y \\ 0.0 \end{pmatrix} \tag{6}$$

- right:

$$O' = \begin{bmatrix} 0.0 & -s_y sin\theta & s_x cos\theta \\ 0.0 & s_y cos\theta & s_x sin\theta \\ -s_z & 0.0 & 0.0 \end{bmatrix} O + \begin{pmatrix} t_x \\ t_y \\ 0.0 \end{pmatrix} \tag{7}$$

1.2.2 Isometric

The Isometric Eye creates 3D projection of Objects without simulation of the perspective. It adds two other rotations to the Eye: the rotation around the second dimension in the original coordinates system, and the rotation around the first dimension in the Layer coordinates system. Given θ_y and θ_r the angle in radians of these two rotations, the translated rotated scaled projected version O' of the Object O can be obtained as follow:

$$O' = \begin{bmatrix} M_{00} & M_{10} & M_{20} \\ M_{01} & M_{11} & M_{21} \\ M_{02} & M_{12} & M_{22} \end{bmatrix} O + \begin{pmatrix} t_x \\ t_y \\ 0.0 \end{pmatrix} \tag{8}$$

where

$$M_{00} = s_x cos\theta cos\theta_y - s_y sin\theta sin\theta_y sin\theta_r$$
$$M_{01} = s_x sin\theta cos\theta_y + s_y cos\theta sin\theta_y sin\theta_r$$
$$M_{02} = s_z sin\theta_y cos\theta_r$$
$$M_{10} = -s_y sin\theta cos\theta_r$$
$$M_{11} = s_y cos\theta cos\theta_r \qquad (9)$$
$$M_{12} = -s_z sin\theta_r$$
$$M_{20} = -s_x cos\theta sin\theta_y - s_y sin\theta cos\theta_y sin\theta_r$$
$$M_{21} = -s_x sin\theta sin\theta_y + s_y cos\theta cos\theta_y sin\theta_r$$
$$M_{22} = s_z cos\theta_y cos\theta_r$$

1.3   Hand

The Hand interprets each viewed Object and creates a new instance of one or several Objects (possibly of types differents from the one of the original Object). This interpreted version of the Object is the one which will actually be traced by the Tool. For example a Hand may convert a volume object into its outline curve version.

The Hand may be parameterized by the components of dimensions from the 4th one of the Object.

A given Hand type may not be defined for all types of Object. If the user tries to apply a Hand on an unsupported Object, the Hand simply does nothing and the Object won't appear in the Surface.

1.3.1   Default

The default hand simply creates a copy of the Object created by the Eye.

1.4   Tool

The Tool translates each Object created by the Hand into pixels of Ink stacked into the Layers. It generally runs through the Object's internal coordinates, calculates the rounded coordinates of the Object (coordinates in the Layer coordinates system) and the coordinates of the original Object (coordinates in the global coordinates system), then get the Inks value at theses coordinates and add the corresponding pixels in the Layers.

The Tool may be parameterized by the components of dimensions from the 4th one of the Object(s).

A given Tool type may not be defined for all types of Object. If the user tries to apply a Tool on an unsupported Object, the Tool simply does nothing and the Object won't appear in the Surface.

1.4.1 Plotter

If the Plotter draws a Point, it adds one pixel at the rounded position in the Layer. The depth of the pixel is equals to the 3rd dimension of the Point if it has one, 0.0 else. The ink value is calculated by using a 3D vector null for the internal position as a Point doesn't have internal position.

If the Plotter draws a SCurve, it runs through the SCurve from beginning to end by small increment of its internal parameter equals to $0.5/l$ where l is the length (or its approximation) of the SCurve. If the SCurve has only 2 dimensions, the Plotter considers the 3rd as being always equal to 0.0. At each step the Plotter checks if it has move to a different voxel, i.e. if at least one of the rounded value of the 3 first dimensions has changed. If so, the Plotter calculates the Ink value at this position and adds the equivalent pixel to the Layer.

If the Plotter draws a Shapoid, it checks for each voxel of the bounding box of the Shapoid if the center of the voxel is inside the Shapoid, and if so adds a pixel into the Layer at the position (2 first dimension) and depth (3rd dimension) of this voxel. If the Shapoid has only 2 dimensions, a 3rd one always equals to 0.0 is considered. The internal coordinates for a given voxel can be calculated thanks to the Shapoid functions.

1.5 Ink

The Ink converts coordinates into RGBA values. It takes as input internal coordinates (in the Object internal coordinates system), external coordinates (in the global coordinates system) and layer coordinates. Layer coordinates have integer values whlie the others have floating values. An Ink may ignore some or all of the coordinates given as parameter, and to be able to proces any kind of object should be resilient to lower than expected dimensions for these

coordinates (typically by assuming the missing dimensions are equals to 0.0).

The Ink may be parameterized by the components of dimensions from the 4th one of the Object(s).

1.5.1 Solid

The Solid Ink returns a RGBA value set by the user, regardless of the coordinates given as parameters.

1.6 Surface

A Surface is a set of Layers plus an array of RGBA values (the final pixels). It is an abstraction level which allows to manipulate any kind of output for the rendering process: a file on the hard drive, a window on the screen, a widget in another application, ...

The result of the rendering of a scene is stored as individual pixels in the Layers of the Surface, and when the Surface is requested for update, these pixels are blended into the final pixels.

A Surface has a background color which is used has a fallback when blending pixels of Layers (see below for detail).

1.6.1 Image

An Image Surface is an image file saved on a physical media when the Surface is requested for update. Currently the Image Surface supports the following formats:

- TGA: supports only 16, 24 or 32 bits per pixel, uncompressed or RLE RGB (data types 2 and 10)

An Image Surface can also load an image from a physical media toward a new Layer. The new Layer is then created with the dimensions of the image, depth of pixels are set to 0.0, parameters of the new Layer to their default value.

1.6.2 GBWidget

A GBWidget Surface is a Surface compatible with the GTK library. It can return a GtkWidget drawing area to be used on the GTK side (added to a container, attached to an idle function, ...). The rendering of the Surface automatically update the GTK drawing area.

1.6.3 GBApp

A GBApp Surface is a full GtkApplication allowing the user to simply display graphics content on screen. The GTK windows created has a fixed size defined by the user. The rendering of the Surface refreshes the content of the window. An idle function can be set by the user to create animation. The GBApp Surface can return a GtkWidget on which the user could attached more event handler to extend the behaviour of the window.

1.7 Layer

A Layer is a an array of stacked pixels (RGBA value, depth value and blend mode). A Layer has a position in its Surface and dimensions eventually larger or smaller than its Surface's dimensions. The Layer content is always clipped to its Surface when rendering.

The Layers are stacked in the Surface in three levels:

- Background: Pixels of Layers in Background level are rendered in the order defined by the stack, or in the order their original Object has been added to the Layer for the case of several Objects in the same Layer. For example, pixels of layerA will be hided by pixels of layerB if layerA is before layerB in the stack of Layers. Another example, pixels of objectA will be hided by pixels of objectB, both in the same Layer, if objectA has been added to the Layer before objectB.

- Inside: Pixels of Inside Layers always hide pixels of Background Layers. The order of addition into the scene of Inside Layers doesn't matter. Pixels are always reordered according to their depth accross all Inside Layers upon rendering. Pixels with lower depth hide pixels with higher depth.

- Foreground: Pixels of Foreground layers always hide pixels of Inside and Background Layers. Pixels of Layers in Background level are rendered

in the order defined by the stack, or in the order their original Object has been added to the Layer for the case of several Objects in the same Layer. For example, pixels of layerA will be hided by pixels of layerB if layerA is before layerB in the stack of Layers. Another example, pixels of objectA will be hided by pixels of objectB, both in the same Layer, if objectA has been added to the Layer before objectB.

The position of a Layer in the stack, or its level, can be modified by the user after creation.

To create the final pixels, stacked pixels are blended according to their blending mode. The blending mode of a given stacked pixel is the one of its Layer at the time of rendering. The blending occurs from top to bottom (from the end of the Foreground stack toward the beginning of the Background stack). The blending starts with the highest stacked pixel, and then blend it with its blending mode on the stacked pixel below, and so on downward. Blending stops as soon as the alpha value of the resulting final pixel reaches its maximal value (255) for optimization. If the blending operation reaches the bottom of the stack and the alpha value has still not reached its maximum, one last blend occurs with a pixel equals to the background color of the Surface.

The blending modes are detailed below (where $rgba$ is the higher pixel, $rgba'$ is the lower pixel. res is the result pixel and the blending mode is the blending mode of the higher pixel.

1.7.1   Default

The Default blending mode simply overwrites the lower pixel to the final pixel.

1.7.2   Normal

The Normal blending mode calculates the RGBA value as follow:

$$\begin{cases} res(red) = \lfloor (1.0 - u)rgba(red) + u * rgba'(red) + 0.5 \rfloor \\ res(green) = \lfloor (1.0 - u)rgba(green) + u * rgba'(green) + 0.5 \rfloor \\ res(blue) = \lfloor (1.0 - u)rgba(blue) + u * rgba'(blue) + 0.5 \rfloor \\ res(alpha) = \lfloor MIN(255.0, rgba(alpha) + rgba'(alpha)) + 0.5 \rfloor \end{cases} \quad (10)$$

where

$$u = 0.5 * (1.0 + (rgba(alpha) - rgba'(alpha))/255.0) \quad (11)$$

1.7.3 Over

The Over blending mode calculates the RGBA value as follow:

$$\begin{cases} res(red) = \lfloor (1.0 - u)rgba(red) + u * rgba'(red) + 0.5 \rfloor \\ res(green) = \lfloor (1.0 - u)rgba(green) + u * rgba'(green) + 0.5 \rfloor \\ res(blue) = \lfloor (1.0 - u)rgba(blue) + u * rgba'(blue) + 0.5 \rfloor \\ res(alpha) = \lfloor MIN(255.0, rgba(alpha) + rgba'(alpha)) + 0.5 \rfloor \end{cases} \quad (12)$$

where

$$u = rgba(alpha)/255.0 \quad (13)$$

2 Interface

2.1 genbrush.h

```
// ============ GENBRUSH.H ================

#ifndef GENBRUSH_H
#define GENBRUSH_H


// About the coordinates systems:
// In input, the coordinates system must be a left handed
// coordinates system, with x toward the right, y toward the top and
// z toward the rear.
//    y
//    ^ z
//    |/
//    -->x
// So the front view of a GBEyeOrtho has x to the right, y to the top
// and z as the depth component (positive is away from the viewer).
//    y
//    ^
//    |
//   z-->x
// For a GBEyeIsometric, x is toward the top-right, y is toward the top
// and z is toward the top-left.
//    y
// z ^ x
//   \|/
// In output, the coordinates system of the surface is x toward the
// right and y toward the top, with the origin at the bottom-left of
// the surface.
//    y
//    ^
//    |
//    -->x
// In GBSurface, the final pixels are stored in rows from left to
// right, the first row is the top row in the surface.

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

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
```

```c
#include <string.h>
#include <stdbool.h>
#include "pberr.h"
#include "pbmath.h"
#include "gset.h"
#include "shapoid.h"
#include "bcurve.h"

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

// Color index in _rgba
// for CAIRO_FORMAT_ARGB32 :
#define GBPixelRed 2
#define GBPixelGreen 1
#define GBPixelBlue 0
#define GBPixelAlpha 3

#define GBColorWhite (GBPixel){ \
    ._rgba[GBPixelAlpha]=255,._rgba[GBPixelRed]=255, \
    ._rgba[GBPixelGreen]=255,._rgba[GBPixelBlue]=255}
#define GBColorBlack (GBPixel){ \
    ._rgba[GBPixelAlpha]=255,._rgba[GBPixelRed]=0, \
    ._rgba[GBPixelGreen]=0,._rgba[GBPixelBlue]=0}
#define GBColorTransparent (GBPixel){ \
    ._rgba[GBPixelAlpha]=0,._rgba[GBPixelRed]=0, \
    ._rgba[GBPixelGreen]=0,._rgba[GBPixelBlue]=0}
#define GBColorRed (GBPixel){ \
    ._rgba[GBPixelAlpha]=255,._rgba[GBPixelRed]=255, \
    ._rgba[GBPixelGreen]=0,._rgba[GBPixelBlue]=0}
#define GBColorGreen (GBPixel){ \
    ._rgba[GBPixelAlpha]=255,._rgba[GBPixelRed]=0, \
    ._rgba[GBPixelGreen]=255,._rgba[GBPixelBlue]=0}
#define GBColorBlue (GBPixel){ \
    ._rgba[GBPixelAlpha]=255,._rgba[GBPixelRed]=0, \
    ._rgba[GBPixelGreen]=0,._rgba[GBPixelBlue]=255}

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

typedef struct GBPixel {
  // Pixel values
  unsigned char _rgba[4];
} GBPixel;

typedef enum GBLayerBlendMode {
  GBLayerBlendModeDefault, // Simple overwritting
  GBLayerBlendModeNormal, // Blending according to relative alpha
  GBLayerBlendModeOver // Blending according to alpha of top pix
} GBLayerBlendMode;

typedef enum GBLayerStackPosition {
  GBLayerStackPosBg, // Layers' pixels are stacked according to their
                     // index
  GBLayerStackPosInside, // Layers' pixels are stacked according to
                         // their depth
  GBLayerStackPosFg // Layers' pixels are stacked according to their
                    // index
} GBLayerStackPosition;

typedef struct GBStackedPixel {
  // Pixel values
  GBPixel _val;
  // Depth
```

```
  float _depth;
  // BlendMode
  GBLayerBlendMode _blendMode;
}  GBStackedPixel;

typedef struct GBLayer {
  // Position in the GBSurface of the bottom left corner of the GBLayer
  // (coordinates of the bottom left corner in the GBSurface are 0,0)
  // (x toward right, y toward top)
  VecShort2D _pos;
  // Previous position
  VecShort2D _prevPos;
  // Dimension
  VecShort2D _dim;
  // _dim[0] x _dim[1] GSet of StackedPixel (stored by rows)
  GSet* _pix;
  // Composition
  GBLayerBlendMode _blendMode;
  // Modified flag
  bool _modified;
  // Position in stack
  GBLayerStackPosition _stackPos;
} GBLayer;

typedef enum GBPPType {
  GBPPTypeNormalizeHue
} GBPPType;

typedef struct GBPostProcessing {
  enum GBPPType _type;
} GBPostProcessing;

typedef enum GBSurfaceType {
  GBSurfaceTypeDefault,
  GBSurfaceTypeImage
#if BUILDWITHGRAPHICLIB == 1
    , GBSurfaceTypeApp, GBSurfaceTypeWidget
#endif
} GBSurfaceType;

typedef struct GBSurface {
  // Type of the surface
  GBSurfaceType _type;
  // Dimension (x toward right, y toward top)
  VecShort2D _dim;
  // Background color
  GBPixel _bgColor;
  // Set of GBLayer
  GSet _layers;
  // Final pixels
  GBPixel* _finalPix;
} GBSurface;

typedef struct GBSurfaceImage {
  // Parent GBSurface
  GBSurface _surf;
  // File name
  char* _fileName;
} GBSurfaceImage;

typedef enum GBEyeType {
  GBEyeTypeOrtho,
```

```
    GBEyeTypeIsometric
} GBEyeType;

typedef struct GBEye {
  // Type
  GBEyeType _type;
  // Scale
  VecFloat3D _scale;
  // (0,0) in graphical element is translated at _orig in the surface
  VecFloat2D _orig;
  // Rotation of the eye (clockwise, radians)
  float _theta;
  // Projection matrix from real space to surface space
  // Transformation are applied in the following order in the
  // surface space:
  // scale, rotation, translation
  MatFloat* _proj;
} GBEye;

typedef enum GBEyeOrthoView {
  // View from ...
  GBEyeOrthoViewFront,
  GBEyeOrthoViewRear,
  GBEyeOrthoViewTop,
  GBEyeOrthoViewBottom,
  GBEyeOrthoViewLeft,
  GBEyeOrthoViewRight
} GBEyeOrthoView;

typedef struct GBEyeOrtho {
  // GBEye data
  GBEye _eye;
  // Orientation of the orthographic projection
  GBEyeOrthoView _view;
} GBEyeOrtho;

typedef struct GBEyeIsometric {
  // GBEye data
  GBEye _eye;
  // Rotation right handed around y (in radians, initially 0.0)
  float _thetaY;
  // Rotation right handed around the right direction in the surface
  // (in radians, in [-pi/2, pi/2], initially pi/4)
  float _thetaRight;
} GBEyeIsometric;

typedef enum GBHandType {
  GBHandTypeDefault
} GBHandType;

typedef struct GBHand {
  // Type
  GBHandType _type;
} GBHand;

typedef struct GBHandDefault {
  // Parent
  GBHand _hand;
} GBHandDefault;

typedef enum GBToolType {
  GBToolTypePlotter
```

```
} GBToolType;

typedef struct GBTool {
  // Type
  GBToolType _type;
} GBTool;

typedef struct GBToolPlotter {
  // Parent
  GBTool _tool;
} GBToolPlotter;

typedef enum GBInkType {
  GBInkTypeSolid
} GBInkType;

typedef struct GBInk {
  // Type
  GBInkType _type;
} GBInk;

typedef struct GBInkSolid {
  // Parent
  GBInk _ink;
  // Color
  GBPixel _color;
} GBInkSolid;

typedef enum GBObjType {
  GBObjTypePoint,
  GBObjTypeShapoid,
  GBObjTypeSCurve
} GBObjType;

typedef struct GBObjPod {
  // Type
  GBObjType _type;
  // Source object
  union {
    VecFloat* _srcPoint;
    Shapoid* _srcShapoid;
    SCurve* _srcSCurve;
  };
  // Object projected through eye
  union {
    VecFloat* _eyePoint;
    Shapoid* _eyeShapoid;
    SCurve* _eyeSCurve;
  };
  // GSet of projected Points (VecFloat) through eye + hand
  GSetVecFloat _handPoints;
  // GSet of projected Shapoids through eye + hand
  GSetShapoid _handShapoids;
  // GSet of projected Shapoids through eye + hand
  GSetSCurve _handSCurves;
  // Eye
  GBEye* _eye;
  // Hand
  GBHand* _hand;
  // Tool
  GBTool* _tool;
  // Ink
```

```
  GBInk* _ink;
  // Layer
  GBLayer* _layer;
} GBObjPod;

typedef struct GenBrush {
  // Surface of the GenBrush
  GBSurface* _surf;
  // Set of GBObjPod to be drawn
  GSet _pods;
  // Set of GBPostProcessing to be apply at the end of GBUpdate
  GSet _postProcs;
} GenBrush;

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

// --------------- GBPixel -------------------------

// Blend the pixel 'pix' into the pixel 'that'
// BlendNormal mixes colors according to their relative alpha value
// and add the alpha values
void GBPixelBlendNormal(GBPixel* that, GBPixel* pix);

// Blend the pixel 'pix' into the pixel 'that'
// BlendOver mixes colors according to the alpha value of 'pix'
// and add the alpha values
void GBPixelBlendOver(GBPixel* that, GBPixel* pix);

// Return the blend result of the stack of Pixel 'stack'
// If there is transparency down to the bottom of the stack, use the
// background color 'bgColor'
// If the stack is empty, return a transparent pixel
GBPixel GBPixelStackBlend(GSet* stack, GBPixel* bgColor);

// --------------- GBLayer -------------------------

// Create a new GBLayer with dimensions 'dim'
// The layer is initialized with empty stacks of pixel
// _pos = (0,0)
// blendMode = GBLayerBlendModeDefault
// stackPos = GBLayerStackPosBg
GBLayer* GBLayerCreate(VecShort2D* dim);

// Free the GBLayer 'that'
void GBLayerFree(GBLayer** that);

// Get the area of the layer (width * height)
#if BUILDMODE != 0
inline
#endif
int GBLayerArea(GBLayer* that);

// Get the position of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D* GBLayerPos(GBLayer* that);

// Get a copy of the position of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
```

```
VecShort2D GBLayerGetPos(GBLayer* that);

// Set the position of the GBLayer 'that' to 'pos'
// If the flag _modified==false _prevPos is first set to _pos
// and _modified is set to true
#if BUILDMODE != 0
inline
#endif
void GBLayerSetPos(GBLayer* that, VecShort2D* pos);

// Get the previous position of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D* GBLayerPrevPos(GBLayer* that);

// Get a copy of the previous position of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D GBLayerGetPrevPos(GBLayer* that);

// Get the dimensions of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D* GBLayerDim(GBLayer* that);

// Get a copy of the dimensions of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D GBLayerGetDim(GBLayer* that);

// Get a copy of the blend mode of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
GBLayerBlendMode GBLayerGetBlendMode(GBLayer* that);

// Set the blend mode of the GBLayer 'that' to 'blend'
// Set the flag _modified to true
#if BUILDMODE != 0
inline
#endif
void GBLayerSetBlendMode(GBLayer* that, GBLayerBlendMode blend);

// Get a copy of the modified flag of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
bool GBLayerIsModified(GBLayer* that);

// Set the modified flag of the GBLayer 'that' to 'flag'
#if BUILDMODE != 0
inline
#endif
void GBLayerSetModified(GBLayer* that, bool flag);

// Get a copy of the stack position of the GBLayer 'that'
#if BUILDMODE != 0
inline
```

```
#endif
GBLayerStackPosition GBLayerGetStackPos(GBLayer* that);

// Set the stack position of the GBLayer 'that' to 'pos'
// Set the flag _modified to true
#if BUILDMODE != 0
inline
#endif
void GBLayerSetStackPos(GBLayer* that, GBLayerStackPosition pos);

// Get the stacked pixels of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
GSet* GBLayerPixels(GBLayer* that);

// Get the stacked pixels of the GBLayer 'that' at position 'pos'
// 'pos' must be inside the layer
#if BUILDMODE != 0
inline
#endif
GSet* GBLayerPixel(GBLayer* that, VecShort2D* pos);

// Get the stacked pixels of the GBLayer 'that' at position 'pos'
// If 'pos' is out of the layer return NULL
#if BUILDMODE != 0
inline
#endif
GSet* GBLayerPixelSafe(GBLayer* that, VecShort2D* pos);

// Add the pixel 'pix' with depth 'depth' on top of the stack at
// position 'pos' of GBLayer 'that'
// Set the flag _modified to true
// 'pos' must be inside the layer
// If the pixel is completely transparent (_rgba[GBPixelAlpha]==0)
// do nothing
#if BUILDMODE != 0
inline
#endif
void GBLayerAddPixel(GBLayer* that, VecShort2D* pos, GBPixel* pix,
  float depth);

// Add the pixel 'pix' with depth 'depth' on top of the stack at
// position 'pos' of GBLayer 'that'
// Set the flag _modified to true
// If 'pos' is out of the layer do nothing
// If the pixel is completely transparent (_rgba[GBPixelAlpha]==0)
// do nothing
#if BUILDMODE != 0
inline
#endif
void GBLayerAddPixelSafe(GBLayer* that, VecShort2D* pos, GBPixel* pix,
  float depth);

// Return true if the position 'pos' is inside the layer 'that'
// boundary, false else
#if BUILDMODE != 0
inline
#endif
bool GBLayerIsPosInside(GBLayer* that, VecShort2D *pos);

// Delete all the stacked pixels in the GBLayer 'that'
```

```
#if BUILDMODE != 0
inline
#endif
void GBLayerFlush(GBLayer* that);

// Create a new GBLayer with dimensions and content given by the
// image on disk at location 'fileName'
// Return NULL if we couldn't create the layer
GBLayer* GBLayerCreateFromFile(char* fileName);

// Get the boundary of the GBLayer 'that' inside the GBSurface 'surf'
// The boundaries are given as a Facoid
// If the flag 'prevPos' is true, gives the bounday at the previous
// position
// Return NULL if the layer is completely out of the surface
Facoid* GBLayerGetBoundaryInSurface(GBLayer* that, GBSurface* surf,
  bool prevPos);

// --------------- GBPostProcessing -------------------------

// Create a static GBPostProcessing with type 'type'
GBPostProcessing* GBPostProcessingCreate(GBPPType type);

// Create a new static GBPostProcessing with type 'type'
GBPostProcessing GBPostProcessingCreateStatic(GBPPType type);

// Free the memory used by the GBPostProcessing 'that'
void GBPostProcessingFree(GBPostProcessing** that);

// Return the type of the GBPostProcessing 'that'
#if BUILDMODE != 0
inline
#endif
GBPPType GBPostProcessingGetType(GBPostProcessing* that);

// --------------- GBSurface -------------------------

// Create a new static GBSurface with dimension 'dim' and type 'type'
// _finalPix is set to 0
// _bgColor is set to white
GBSurface GBSurfaceCreateStatic(GBSurfaceType type, VecShort2D* dim);

// Create a static GBSurface with dimension 'dim' and type 'type'
// _finalPix is set to 0
// _bgColor is set to white
GBSurface* GBSurfaceCreate(GBSurfaceType type, VecShort2D* dim);

// Free the memory used by the GBSurface 'that'
void GBSurfaceFree(GBSurface** that);

// Free the memory used by the properties of the GBSurface 'that'
void GBSurfaceFreeStatic(GBSurface* that);

// Clone the GBSurface 'that'
GBSurface GBSurfaceClone(GBSurface* that);

// Return true if the GBSurface 'that' has same dimension and same
// values for _finalPix as GBSurface 'surf'
// Else, return false
#if BUILDMODE != 0
inline
#endif
```

```
bool GBSurfaceIsSameAs(GBSurface* that, GBSurface* surf);

// Get the type of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
GBSurfaceType GBSurfaceGetType(GBSurface* that);

// Get a copy of the dimensions of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D GBSurfaceGetDim(GBSurface* that);

// Get the dimensions of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D* GBSurfaceDim(GBSurface* that);

// Get the final pixels of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel* GBSurfaceFinalPixels(GBSurface* that);

// Get the final pixel at position 'pos' of the GBSurface 'that'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
GBPixel* GBSurfaceFinalPixel(GBSurface* that, VecShort2D* pos);

// Get a copy of the final pixel at position 'pos' of the GBSurface
// 'that'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
GBPixel GBSurfaceGetFinalPixel(GBSurface* that, VecShort2D* pos);

// Set the final pixel at position 'pos' of the GBSurface 'that' to
// the pixel 'pix'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
void GBSurfaceSetFinalPixel(GBSurface* that, VecShort2D* pos,
  GBPixel* pix);

// Get the final pixel at position 'pos' of the GBSurface 'that'
// If 'pos' is out of the surface return NULL
#if BUILDMODE != 0
inline
#endif
GBPixel* GBSurfaceFinalPixelSafe(GBSurface* that, VecShort2D* pos);

// Get a copy of the final pixel at position 'pos' of the GBSurface
// 'that'
// If 'pos' is out of the surface return a transparent pixel
#if BUILDMODE != 0
inline
```

```
#endif
GBPixel GBSurfaceGetFinalPixelSafe(GBSurface* that, VecShort2D* pos);

// Set the final pixel at position 'pos' of the GBSurface 'that' to
// the pixel 'pix'
// If 'pos' is out of the surface do nothing
#if BUILDMODE != 0
inline
#endif
void GBSurfaceSetFinalPixelSafe(GBSurface* that, VecShort2D* pos,
  GBPixel* pix);

// Get the area of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
int GBSurfaceArea(GBSurface* that);

// Get the background color of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel* GBSurfaceBgColor(GBSurface* that);

// Get a copy of the background color of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel GBSurfaceGetBgColor(GBSurface* that);

// Set the background color of the GBSurface 'that' to 'col'
#if BUILDMODE != 0
inline
#endif
void GBSurfaceSetBgColor(GBSurface* that, GBPixel* col);

// Return true if the position 'pos' is inside the GBSurface 'that'
// boundary, false else
#if BUILDMODE != 0
inline
#endif
bool GBSurfaceIsPosInside(GBSurface* that, VecShort2D *pos);

// Get the set of layers of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
GSet* GBSurfaceLayers(GBSurface* that);

// Add a new GBLayer with dimensions 'dim' to the top of the stack
// of layers of the GBSurface 'that'
// Return the new GBLayer
#if BUILDMODE != 0
inline
#endif
GBLayer* GBSurfaceAddLayer(GBSurface* that, VecShort2D* dim);

// Add a new GBLayer with the content of the image located at
// 'fileName' to the top of the stack of layers of the GBSurface 'that'
// Return the new GBLayer
#if BUILDMODE != 0
inline
```

```
#endif
GBLayer* GBSurfaceAddLayerFromFile(GBSurface* that, char* fileName);

// Get the 'iLayer'-th layer of the GBSurface 'that'
// 'iLayer' must be valid
#if BUILDMODE != 0
inline
#endif
GBLayer* GBSurfaceGetLayer(GBSurface* that, int iLayer);

// Get the number of layer of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
int GBSurfaceNbLayer(GBSurface* that);

// Set the _modified flag of all layers of the GBSurface 'that'
// to 'flag'
#if BUILDMODE != 0
inline
#endif
void GBSurfaceSetLayersModified(GBSurface* that, bool flag);

// Set the position of the GBLayer 'layer' into the set of layers of
// the GBSurface 'that' to 'pos'
// If 'layer' can't be found in the surface do nothing
// 'pos' must be valid (0<='pos'<nbLayers)
#if BUILDMODE != 0
inline
#endif
void GBSurfaceSetLayerPos(GBSurface* that, GBLayer* layer, int pos);

// Remove the GBLayer 'layer' from the set of layers of the
// GBSurface 'that'
// The memory used by 'layer' is freed
#if BUILDMODE != 0
inline
#endif
void GBSurfaceRemoveLayer(GBSurface* that, GBLayer* layer);

// Get a GSet of Facoid representing the sub areas of the GBSurface
// 'that' affected by layers with _modified flag equal to true
// If there is no modified sub area return an empty GSet
GSetShapoid* GBSurfaceGetModifiedArea(GBSurface* that);

// Update the final pixels according to layers of the GBSurface 'that'
// Update only pixels affected by layers with the _modified flag
// equals to true
void GBSurfaceUpdate(GBSurface* that);

// Reset all the final pix of the GBSurface 'that' to its
// background color, and reset all the modified flag of layers to true
void GBSurfaceFlush(GBSurface* that);

// Apply the post processing 'post' to the final pixels in the
// GBSurface 'that'
void GBSurfacePostProcess(GBSurface* that, GBPostProcessing* post);

// ---------------- GBSurfaceImage -------------------------

// Create a new GBSurfaceImage with dimension 'dim'
GBSurfaceImage* GBSurfaceImageCreate(VecShort2D* dim);
```

```
// Free the memory used by the GBSurfaceImage 'that'
void GBSurfaceImageFree(GBSurfaceImage** that);

// Clone the GBSurfaceImage 'that'
GBSurfaceImage* GBSurfaceImageClone(GBSurfaceImage* that);

// Get the filename of the GBSurfaceImage 'that'
#if BUILDMODE != 0
inline
#endif
char* GBSurfaceImageFileName(GBSurfaceImage* that);

// Set the filename of the GBSurfaceImage 'that' to 'fileName'
#if BUILDMODE != 0
inline
#endif
void GBSurfaceImageSetFileName(GBSurfaceImage* that, char* fileName);

// Save a GBSurfaceImage 'that'
// If the filename is not set do nothing and return false
// Return true if it could save the surface, false else
bool GBSurfaceImageSave(GBSurfaceImage* that);

// Create a new GBSurfaceImage with one layer containing the content
// of the image located at 'fileName' and dimensions equal to the
// dimensions of the image
// If the image couldn't be loaded return NULL
GBSurfaceImage* GBSurfaceImageCreateFromFile(char* fileName);

// --------------- GBEye -------------------------

// Create a new GBEye with type 'type'
GBEye GBEyeCreateStatic(GBEyeType type);

// Free the memory used by the GBEye 'that'
void GBEyeFreeStatic(GBEye* that);

// Return the type of the GBEye 'that'
#if BUILDMODE != 0
inline
#endif
GBEyeType _GBEyeGetType(GBEye* that);

// Get the scale of the GBEye
#if BUILDMODE != 0
inline
#endif
VecFloat3D* _GBEyeScale(GBEye* that);

// Get a copy of the scale of the GBEye
#if BUILDMODE != 0
inline
#endif
VecFloat3D _GBEyeGetScale(GBEye* that);

// Get the translation of the GBEye
#if BUILDMODE != 0
inline
#endif
VecFloat2D* _GBEyeOrig(GBEye* that);
```

```
// Get a copy of the translation of the GBEye
#if BUILDMODE != 0
inline
#endif
VecFloat2D _GBEyeGetOrig(GBEye* that);

// Get the rotation of the GBEye (in radians)
#if BUILDMODE != 0
inline
#endif
float _GBEyeGetRot(GBEye* that);

// Set the scale of the GBEye
#if BUILDMODE != 0
inline
#endif
void GBEyeSetScaleVec(GBEye* that, VecFloat3D* scale);
#if BUILDMODE != 0
inline
#endif
void GBEyeSetScaleFloat(GBEye* that, float scale);

// Set the translation of the GBEye
#if BUILDMODE != 0
inline
#endif
void _GBEyeSetOrig(GBEye* that, VecFloat2D* trans);

// Set the rotation of the GBEye (in radians)
#if BUILDMODE != 0
inline
#endif
void _GBEyeSetRot(GBEye* that, float theta);

// Update the projection matrix of the GBEye according to scale, rot
// and origin
void GBEyeUpdateProj(GBEye* that);

// Get the matrix projection of the eye
#if BUILDMODE != 0
inline
#endif
MatFloat* _GBEyeProj(GBEye* that);

// Call the appropriate GBEye<>Process according to the type of the
// GBEye 'that'
#if BUILDMODE != 0
inline
#endif
void _GBEyeProcess(GBEye* that, GBObjPod* pod);

// Return the projection through the GBEye 'that' of the Point 'point'
VecFloat* GBEyeGetProjectedPoint(GBEye* that, VecFloat* point);

// Return the projection through the GBEye 'that' of the SCurve 'curve'
SCurve* GBEyeGetProjectedCurve(GBEye* that, SCurve* curve);

// Return the projection through the GBEye 'that' of the Shapoid 'shap'
Shapoid* GBEyeGetProjectedShapoid(GBEye* that, Shapoid* shap);

// ---------------- GBEyeOrtho ------------------------
```

```
// Return a new GBEyeOrtho with orientation 'view'
// scale is initialized to (1,1), trans to (0,0) and rot to 0
GBEyeOrtho* GBEyeOrthoCreate(GBEyeOrthoView view);

// Free the memory used by the GBEyeOrho 'that'
void GBEyeOrthoFree(GBEyeOrtho** that);

// Set the orientation the GBEyeOrtho 'that'
#if BUILDMODE != 0
inline
#endif
void GBEyeOrthoSetView(GBEyeOrtho* that, GBEyeOrthoView view);

// Process the object of the GBObjPod 'pod' to update the viewed object
// through the GBEyeOrtho 'that'
void GBEyeOrthoProcess(GBEyeOrtho* that, GBObjPod* pod);

// --------------- GBEyeIsometric ------------------------

// Return a new GBEyeIsometric with orientation 'view'
// scale is initialized to (1,1), trans to (0,0) and rot to 0
// thetaY is initialized to pi/4 and thetaRight to pi/4
GBEyeIsometric* GBEyeIsometricCreate();

// Free the memory used by the GBEyeIsometric 'that'
void GBEyeIsometricFree(GBEyeIsometric** that);

// Set the angle around Y of the GBEyeOrtho to 'theta' (in radians)
#if BUILDMODE != 0
inline
#endif
void GBEyeIsometricSetRotY(GBEyeIsometric* that, float theta);

// Set the angle around Right of the GBEyeOrtho to 'theta'
// (in radians, in [-pi/2, pi/2])
// If 'theta' is out of range it is automatically bounded
// (ex: pi -> pi/2)
#if BUILDMODE != 0
inline
#endif
void GBEyeIsometricSetRotRight(GBEyeIsometric* that, float theta);

// Get the angle around Y of the GBEyeOrtho 'that'
#if BUILDMODE != 0
inline
#endif
float GBEyeIsometricGetRotY(GBEyeIsometric* that);

// Get the angle around Right of the GBEyeOrtho 'that'
#if BUILDMODE != 0
inline
#endif
float GBEyeIsometricGetRotRight(GBEyeIsometric* that);

// Process the object of the GBObjPod 'pod' to update the viewed object
// through the GBEyeIsometric 'that'
void GBEyeIsometricProcess(GBEyeIsometric* that, GBObjPod* pod);

// --------------- GBHand ------------------------

// Create a new GBHand with type 'type'
GBHand GBHandCreateStatic(GBHandType type);
```

```
// Free the memory used by the GBHand 'that'
void GBHandFreeStatic(GBHand* that);

// Return the type of the GBHand 'that'
#if BUILDMODE != 0
inline
#endif
GBHandType _GBHandGetType(GBHand* that);

// Call the appropriate GBHand<>Process according to the type of the
// GBHand 'that'
#if BUILDMODE != 0
inline
#endif
void _GBHandProcess(GBHand* that, GBObjPod* pod);

// --------------- GBHandDefault ------------------------

// Create a new GBHandDefault
GBHandDefault* GBHandDefaultCreate();

// Free the memory used by the GBHandDefault 'that'
void GBHandDefaultFree(GBHandDefault** that);

// Process the viewed projection of the object in GBObjPod 'pod' into
// its handed projection through the GBHandDefault 'that'
void GBHandDefaultProcess(GBHandDefault* that, GBObjPod* pod);

// --------------- GBTool ------------------------

// Create a static GBTool with GBToolType 'type'
GBTool GBToolCreateStatic(GBToolType type);

// Free the memory used by the GBTool 'that'
void GBToolFreeStatic(GBTool* that);

// Return a copy of the type of the GBTool 'that'
#if BUILDMODE != 0
inline
#endif
GBToolType GBToolGetType(GBTool* that);

// Function to call the appropriate GBTool<>Draw function according to
// type of GBTool 'that'
#if BUILDMODE != 0
inline
#endif
void _GBToolDraw(GBTool* that, GBObjPod* pod);

// --------------- GBToolPlotter ------------------------

// Create a new GBToolPlotter
GBToolPlotter* GBToolPlotterCreate();

// Free the memory used by the GBToolPlotter 'that'
void GBToolPlotterFree(GBToolPlotter** that);

// Draw the object in the GBObjPod 'pod' with the GBToolPlotter 'that'
void GBToolPlotterDraw(GBToolPlotter* that, GBObjPod* pod);

// --------------- GBInk ------------------------
```

```
// Return the type of the GBInk 'that'
#if BUILDMODE != 0
inline
#endif
GBInkType _GBInkGetType(GBInk* that);

// Free the memory used by the GBInk 'that'
void GBInkFree(GBInk* that);

// Entry point for the GBTool<>Draw function to get the color of the
// appropriate GBInk according to the type of 'that'
// posInternal represents the position in the object internal space
// posExternal represents the position in the global coordinates system
// posLayer represents the position in the destination layer
GBPixel _GBInkGet(GBInk* that, GBTool* tool, GBObjPod* pod,
  VecFloat* posInternal, VecFloat* posExternal, VecShort* posLayer);

// ---------------- GBInkSolid -------------------------

// Create a new GBInkSolid with color 'col'
GBInkSolid* GBInkSolidCreate(GBPixel* col);

// Free the memory used by the GBInkSolid 'that'
void GBInkSolidFree(GBInkSolid** that);

// Get the color of the GBInkSolid 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel GBInkSolidGet(GBInkSolid* that);

// Set the color of the GBInkSolid 'that' to 'col'
#if BUILDMODE != 0
inline
#endif
void GBInkSolidSet(GBInkSolid* that, GBPixel* col);

// ---------------- GBObjPod -------------------------

// Create a new GBObjPod for a Point at position 'pos'
// drawn with 'eye', 'hand', 'tool' and 'ink' in layer 'layer'
// 'pos' must be a vector of 2 or more dimensions
// If 'eye', 'hand, 'tool', 'ink' or 'layer' is null return null
GBObjPod* _GBObjPodCreatePoint(VecFloat* pos, GBEye* eye, GBHand* hand,
  GBTool* tool, GBInk* ink, GBLayer* layer);

// Create a new GBObjPod for the Shapoid 'shap'
// drawn with 'eye', 'hand', 'tool' and 'ink' in layer 'layer'
// 'shap' 's dimension must be 2 or more
// If 'eye', 'hand', 'tool', 'ink' or 'layer' is null return null
GBObjPod* _GBObjPodCreateShapoid(Shapoid* shap, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer);

// Create a new GBObjPod for the SCurve 'curve'
// drawn with 'eye', 'hand', 'tool' and 'ink' in layer 'layer'
// 'curve' 's dimension must be 2 or more
// If 'eye', 'hand', 'tool', 'ink' or 'layer' is null return null
GBObjPod* _GBObjPodCreateSCurve(SCurve* curve, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer);

// Free the memory used by the GBObjPod 'that'
```

```
void GBObjPodFree(GBObjPod** that);

// Return the type of the object in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBObjType GBObjPodGetType(GBObjPod* that);

// Return the object in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
void* GBObjPodGetObj(GBObjPod* that);

// Return the object viewed by its attached eye in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
void* GBObjPodGetEyeObj(GBObjPod* that);

// Return the object processed as Points by its attached hand in the
// GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GSetVecFloat* GBObjPodGetHandObjAsPoints(GBObjPod* that);

// Return the object processed as Shapoids by its attached hand in the
// GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GSetShapoid* GBObjPodGetHandObjAsShapoids(GBObjPod* that);

// Return the object processed as SCurves by its attached hand in the
// GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GSetSCurve* GBObjPodGetHandObjAsSCurves(GBObjPod* that);

// Return the eye in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBEye* GBObjPodGetEye(GBObjPod* that);

// Return the hand in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBHand* GBObjPodGetHand(GBObjPod* that);

// Return the tool in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBTool* GBObjPodGetTool(GBObjPod* that);

// Return the ink in the GBObjPod 'that'
#if BUILDMODE != 0
inline
```

```
#endif
GBInk* GBObjPodGetInk(GBObjPod* that);

// Return the layer in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBLayer* GBObjPodGetLayer(GBObjPod* that);

// Set the Point viewed by its attached eye in the GBObjPod 'that'
// to 'point'
// If 'point' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetEyePoint(GBObjPod* that, VecFloat* point);

// Set the Shapoid viewed by its attached eye in the GBObjPod 'that'
// to 'shap'
// If 'shap' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetEyeShapoid(GBObjPod* that, Shapoid* shap);

// Set the SCurve viewed by its attached eye in the GBObjPod 'that'
// to 'curve'
// If 'curve' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetEyeSCurve(GBObjPod* that, SCurve* curve);

// Set the eye in the GBObjPod 'that' to 'eye'
// If 'eye' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetEye(GBObjPod* that, GBEye* eye);

// Set the hand in the GBObjPod 'that' to 'hand'
// If 'hand' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetHand(GBObjPod* that, GBHand* hand);

// Set the tool in the GBObjPod 'that' to 'tool'
// If 'tool' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetTool(GBObjPod* that, GBTool* tool);

// Set the ink in the GBObjPod 'that' to 'ink'
// If 'ink' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetInk(GBObjPod* that, GBInk* ink);

// Set the layer in the GBObjPod 'that' to 'layer'
```

```
// If 'layer' is null do nothing
#if BUILDMODE != 0
inline
#endif
void GBObjPodSetLayer(GBObjPod* that, GBLayer* layer);

// ---------------- GenBrush ------------------------

// Create a new GenBrush with a GBSurface of type GBSurfaceTypeImage
// and dimensions 'dim'
GenBrush* GBCreateImage(VecShort2D* dim);

// Free memory used by the GenBrush 'that'
void GBFree(GenBrush** that);

// Create a new GenBrush with one layer containing the content
// of the image located at 'fileName' and dimensions equal to the
// dimensions of the image
// If the image couldn't be loaded return NULL
GenBrush* GBCreateFromFile(char* fileName);

// Get the GBSurface of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBSurface* GBSurf(GenBrush* that);

// Get the dimensions of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D* GBDim(GenBrush* that);

// Get a copy of the dimensions of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D GBGetDim(GenBrush* that);

// Get the final pixels of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel* GBFinalPixels(GenBrush* that);

// Get the final pixel at position 'pos' of the GBSurface of the GB
// 'that'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
GBPixel* GBFinalPixel(GenBrush* that, VecShort2D* pos);

// Get a copy of the final pixel at position 'pos' of the GBSurface
// of the GB 'that'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
GBPixel GBGetFinalPixel(GenBrush* that, VecShort2D* pos);

// Set the final pixel at position 'pos' of the GBSurface of the GB
```

```
// 'that' to the pixel 'pix'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
void GBSetFinalPixel(GenBrush* that, VecShort2D* pos, GBPixel* pix);

// Get the final pixel at position 'pos' of the GBSurface of the GB
// 'that'
// If 'pos' is out of the surface return NULL
#if BUILDMODE != 0
inline
#endif
GBPixel* GBFinalPixelSafe(GenBrush* that, VecShort2D* pos);

// Get a copy of the final pixel at position 'pos' of the GBSurface
// of the GB 'that'
// If 'pos' is out of the surface return a transparent pixel
#if BUILDMODE != 0
inline
#endif
GBPixel GBGetFinalPixelSafe(GenBrush* that, VecShort2D* pos);

// Set the final pixel at position 'pos' of the GBSurface of the GB
// 'that' to the pixel 'pix'
// If 'pos' is out of the surface do nothing
#if BUILDMODE != 0
inline
#endif
void GBSetFinalPixelSafe(GenBrush* that, VecShort2D* pos, GBPixel* pix);

// Get the type of the GBSurface of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBSurfaceType GBGetType(GenBrush* that);

// Get the background color of the GBSurface of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel* GBBgColor(GenBrush* that);

// Get a copy of the background color of the GBSurface of the
// GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel GBGetBgColor(GenBrush* that);

// Set the background color of the GBSurface of the GenBrush
//  'that' to 'col'
#if BUILDMODE != 0
inline
#endif
void GBSetBgColor(GenBrush* that, GBPixel* col);

// Get the filename of the GBSurfaceImage of the GenBrush 'that'
// Return NULL if the surface is not a GBSurfaceImage
#if BUILDMODE != 0
inline
#endif
```

32

```
char* GBFileName(GenBrush* that);

// Set the filename of the GBSurfaceImage of the GenBrush 'that'
// to 'fileName'
// Do nothing if the surface is not a GBSurfaceImage
#if BUILDMODE != 0
inline
#endif
void GBSetFileName(GenBrush* that, char* fileName);

// Update the GBSurface of the GenBrush 'that'
void GBUpdate(GenBrush* that);

// Render the GBSurface (save on disk, display on screen, ...) of
// the GenBrush 'that'
bool GBRender(GenBrush* that);

// Get the area of the GBSurface of the Genbrush 'that'
#if BUILDMODE != 0
inline
#endif
int GBArea(GenBrush* that);

// Return true if the position 'pos' is inside the GBSurface of the
// GenBrush 'that'
// boundary, false else
#if BUILDMODE != 0
inline
#endif
bool GBIsPosInside(GenBrush* that, VecShort2D *pos);

// Return the layers of the surface of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GSet* GBLayers(GenBrush* that);

// Add a new GBLayer with dimensions 'dim' to the top of the stack
// of layers of the GBSurface of the GenBrush 'that'
// Return the new GBLayer
#if BUILDMODE != 0
inline
#endif
GBLayer* GBAddLayer(GenBrush* that, VecShort2D* dim);

// Add a new GBLayer with content equals to the image located at
// 'fileName' to the top of the stack
// of layers of the GBSurface of the GenBrush' that'
// Return the new GBLayer
#if BUILDMODE != 0
inline
#endif
GBLayer* GBAddLayerFromFile(GenBrush* that, char* fileName);

// Get the 'iLayer'-th layer of the GBSurface of the GenBrush 'that'
// 'iLayer' must be valid
#if BUILDMODE != 0
inline
#endif
GBLayer* GBGetLayer(GenBrush* that, int iLayer);

// Get the number of layer of the GBSurface of the GenBrush 'that'
```

```
#if BUILDMODE != 0
inline
#endif
int GBGetNbLayer(GenBrush* that);

// Set the position of the GBLayer 'layer' into the set of layers of
// the GBSurface of the GenBrush'that' to 'pos'
// If 'layer' can't be found in the surface do nothing
// 'pos' must be valid (0<='pos'<nbLayers)
#if BUILDMODE != 0
inline
#endif
void GBSetLayerPos(GenBrush* that, GBLayer* layer, int pos);

// Remove the GBLayer 'layer' from the set of layers of the
// GBSurface of the GenBrush 'that'
// The memory used by 'layer' is freed
#if BUILDMODE != 0
inline
#endif
void GBRemoveLayer(GenBrush* that, GBLayer* layer);

// Get the number of objects in the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
int GBGetNbPod(GenBrush* that);

// Add a GBObjPod for the Point at position 'pos' to the GenBrush 'that'
// drawn with 'eye', 'hand' and 'tool' in layer 'layer'
// 'pos' must be a vector of 2 or more dimensions
#if BUILDMODE != 0
inline
#endif
void _GBAddPoint(GenBrush* that, VecFloat* pos, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer);

// Add a GBObjPod for the Shapoid 'shap' to the GenBrush 'that'
// drawn with 'eye', 'hand' and 'tool' in layer 'layer'
// 'shap' 's dimension must be 2 or more
#if BUILDMODE != 0
inline
#endif
void _GBAddShapoid(GenBrush* that, Shapoid* shap, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer);

// Add a GBObjPod for the SCurve 'curve' to the GenBrush 'that'
// drawn with 'eye', 'hand' and 'tool' in layer 'layer'
// 'curve' 's dimension must be 2 or more
#if BUILDMODE != 0
inline
#endif
void _GBAddSCurve(GenBrush* that, SCurve* curve, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer);

// Remove from the list of pods of the GenBrush 'that' those who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBRemovePod(that, elemA, NULL, NULL, NULL, NULL, NULL) removes all
// the pods related to the object 'elemA'
// GBRemovePod(that, elemA, NULL, handA, NULL, NULL, NULL) removes
// all the pods related to both the object 'elemA' AND 'handA'
```

```
// If all the filters are null it removes all the pods
void _GBRemovePod(GenBrush* that, void* obj, GBEye* eye, GBHand* hand,
  GBTool* tool, GBInk* ink, GBLayer* layer);
// TODO: the filtering arguments should be in a struct

// Set the eye to 'toEye' in the pods of the GenBrush 'that' who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBSetPodEye(that, toEye, elemA, NULL, NULL, NULL, NULL, NULL)
// affects all the pods related to the object 'elemA'
// GBSetPodEye(that, toEye, elemA, NULL, handA, NULL, NULL, NULL)
// affects all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it affects all the pods
// If 'toEye' is null, do nothing
void _GBSetPodEye(GenBrush* that, GBEye* toEye, void* obj, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer);
// TODO: the filtering arguments should be in a struct

// Set the hand to 'toHand' in the pods of the GenBrush 'that' who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBSetPodEye(that, toHand, elemA, NULL, NULL, NULL, NULL, NULL)
// affects all the pods related to the object 'elemA'
// GBSetPodEye(that, toHand, elemA, NULL, handA, NULL, NULL, NULL)
// affects all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it affects all the pods
// If 'toHand' is null, do nothing
void _GBSetPodHand(GenBrush* that, GBHand* toHand, void* obj,
  GBEye* eye, GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer);
// TODO: the filtering arguments should be in a struct

// Set the tool to 'toTool' in the pods of the GenBrush 'that' who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBSetPodTool(that, toTool, elemA, NULL, NULL, NULL, NULL) affects
// all the pods related to the object 'elemA'
// GBSetPodTool(that, toTool, elemA, NULL, handA, NULL, NULL) affects
// all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it affects all the pods
// If 'toTool' is null, do nothing
void _GBSetPodTool(GenBrush* that, GBTool* toTool, void* obj,
  GBEye* eye, GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer);
// TODO: the filtering arguments should be in a struct

// Set the ink to 'toInk' in the pods of the GenBrush 'that' who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBSetPodTool(that, toTool, elemA, NULL, NULL, NULL, NULL) affects
// all the pods related to the object 'elemA'
// GBSetPodTool(that, toTool, elemA, NULL, handA, NULL, NULL) affects
// all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it affects all the pods
// If 'toInk' is null, do nothing
void _GBSetPodInk(GenBrush* that, GBInk* toInk, void* obj,
  GBEye* eye, GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer);
// TODO: the filtering arguments should be in a struct

// Set the layer to 'toLayer' in the pods of the GenBrush 'that' who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBSetPodLayer(that, toLayer, elemA, NULL, NULL, NULL, NULL, NULL)
// affects all the pods related to the object 'elemA'
```

```
// GBSetPodLayer(that, toLayer, elemA, NULL, handA, NULL, NULL, NULL)
// affects all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it affects all the pods
// If 'toLayer' is null, do nothing
void _GBSetPodLayer(GenBrush* that, GBLayer* toLayer, void* obj,
  GBEye* eye, GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer);
// TODO: the filtering arguments should be in a struct

// Reset all the final pix of the surface of the GenBrush 'that' to its
// background color, and reset all the modified flag of layers to true
#if BUILDMODE != 0
inline
#endif
void GBFlush(GenBrush* that);

// Set to true the modified flag of the layers of pods attached to the
// object 'obj' in the list of pods of the GenBrush 'that'
void _GBNotifyChangeFromObj(GenBrush* that, void* obj);

// Set to true the modified flag of the layers of pods attached to the
// GBEye 'eye' in the list of pods of the GenBrush 'that'
void _GBNotifyChangeFromEye(GenBrush* that, GBEye* eye);

// Set to true the modified flag of the layers of pods attached to the
// GBInk 'ink' in the list of pods of the GenBrush 'that'
void _GBNotifyChangeFromInk(GenBrush* that, GBInk* ink);

// Set to true the modified flag of the layers of pods attached to the
// GBHand 'hand' in the list of pods of the GenBrush 'that'
void _GBNotifyChangeFromHand(GenBrush* that, GBHand* hand);

// Set to true the modified flag of the layers of pods attached to the
// GBTool 'tool' in the list of pods of the GenBrush 'that'
void _GBNotifyChangeFromTool(GenBrush* that, GBTool* tool);

// Return true if the surface of the GenBrush 'that' is same as the
// surface of the GenBrush 'gb'
// Else, return false
#if BUILDMODE != 0
inline
#endif
bool GBIsSameAs(GenBrush* that, GenBrush* gb);

// Add a GBPostProcessing of type 'type' to the GenBrush 'that'
// Return the GBPostProcessing
#if BUILDMODE != 0
inline
#endif
GBPostProcessing* GBAddPostProcess(GenBrush* that, GBPPType type);

// Remove the GBPostProcessing 'post' from the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
void GBRemovePostProcess(GenBrush* that, GBPostProcessing* post);

// Remove all the GBPostProcessing from the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
void GBRemoveAllPostProcess(GenBrush* that);
```

```
// Get the 'iPost'-th post process of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBPostProcessing* GBGetPostProcess(GenBrush* that, int iPost);

// Get the GSet of GBPostProcessing of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GSet* GBPostProcs(GenBrush* that);

// Get the number of GBPostProcessing of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
int GBGetNbPostProcs(GenBrush* that);

// Remove all the GBObjPods from the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
void GBRemoveAllPod(GenBrush* that);

#if BUILDWITHGRAPHICLIB == 1
#include "genbrush-GTK.h"
#endif

// ================= Polymorphism =================

#define GBPosIndex(Pos, Dim) \
  (VecGet(Dim, 0) * VecGet(Pos, 1) + VecGet(Pos, 0))

#define GBInkGetType(Ink) _Generic(Ink, \
  GBInk*: _GBInkGetType, \
  GBInkSolid*: _GBInkGetType, \
  default: PBErrInvalidPolymorphism) ((GBInk*)(Ink))

#define GBInkGet(Ink, Tool, Pod, PosInternal, PosExternal, PosLayer) \
  _GBInkGet((GBInk*)(Ink), (GBTool*)(Tool), Pod, \
  (VecFloat*)(PosInternal), (VecFloat*)(PosExternal), \
  (VecShort*)(PosLayer))

#define GBObjPodCreatePoint(Pos, Eye, Hand, Tool, Ink, Layer) \
  _Generic(Pos, \
  VecFloat*: _GBObjPodCreatePoint, \
  VecFloat2D*: _GBObjPodCreatePoint, \
  VecFloat3D*: _GBObjPodCreatePoint, \
  default: PBErrInvalidPolymorphism) ((VecFloat*)(Pos), \
  (GBEye*)(Eye), (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBObjPodCreateShapoid(Shap, Eye, Hand, Tool, Ink, Layer) \
  _Generic(Shap, \
  Shapoid*: _GBObjPodCreateShapoid, \
  Facoid*: _GBObjPodCreateShapoid, \
  Spheroid*: _GBObjPodCreateShapoid, \
  Pyramidoid*: _GBObjPodCreateShapoid, \
  default: PBErrInvalidPolymorphism) ((Shapoid*)(Shap), \
  (GBEye*)(Eye), (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBObjPodCreateSCurve(Curve, Eye, Hand, Tool, Ink, Layer) \
  _GBObjPodCreateSCurve(Curve, \
```

```
  (GBEye*)(Eye), (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBAddPoint(GB, Pos, Eye, Hand, Tool, Ink, Layer) \
  _Generic(Pos, \
  VecFloat*: _GBAddPoint, \
  VecFloat2D*: _GBAddPoint, \
  VecFloat3D*: _GBAddPoint, \
  default: PBErrInvalidPolymorphism) (GB, (VecFloat*)(Pos), \
  (GBEye*)(Eye), (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBAddShapoid(GB, Shap, Eye, Hand, Tool, Ink, Layer) \
  _Generic(Shap, \
  Shapoid*: _GBAddShapoid, \
  Facoid*: _GBAddShapoid, \
  Spheroid*: _GBAddShapoid, \
  Pyramidoid*: _GBAddShapoid, \
  default: PBErrInvalidPolymorphism) (GB, (Shapoid*)(Shap), \
  (GBEye*)(Eye), (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBAddSCurve(GB, Curve, Eye, Hand, Tool, Ink, Layer) \
  _GBAddSCurve(GB, Curve, \
  (GBEye*)(Eye), (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBRemovePod(GB, Obj, Eye, Hand, Tool, Ink, Layer) \
  _GBRemovePod(GB, (void*)(Obj), (GBEye*)(Eye), (GBHand*)(Hand), \
  (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBObjPodGetObjAsPoint(Pod) (VecFloat*)GBObjPodGetObj(Pod)
#define GBObjPodGetObjAsShapoid(Pod) (Shapoid*)GBObjPodGetObj(Pod)
#define GBObjPodGetObjAsSCurve(Pod) (SCurve*)GBObjPodGetObj(Pod)

#define GBObjPodGetEyeObjAsPoint(Pod) (VecFloat*)GBObjPodGetEyeObj(Pod)
#define GBObjPodGetEyeObjAsShapoid(Pod) (Shapoid*)GBObjPodGetEyeObj(Pod)
#define GBObjPodGetEyeObjAsSCurve(Pod) (SCurve*)GBObjPodGetEyeObj(Pod)

#define GBObjPodSetEyePoint(Pod, Point) \
  _GBObjPodSetEyePoint(Pod, (VecFloat*)Point)
#define GBObjPodSetEyeShapoid(Pod, Shap) \
  _GBObjPodSetEyeShapoid(Pod, (Shapoid*)Shap)
#define GBObjPodSetEyeSCurve(Pod, Curve) \
  _GBObjPodSetEyeSCurve(Pod, (SCurve*)Curve)

#define GBSetPodEye(GB, ToEye, Obj, Eye, Hand, Tool, Ink, Layer) \
  _GBSetPodEye(GB, (GBEye*)(ToEye), (void*)(Obj), (GBEye*)(Eye), \
  (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBSetPodHand(GB, ToHand, Obj, Eye, Hand, Tool, Ink, Layer) \
  _GBSetPodHand(GB, (GBHand*)(ToHand), (void*)(Obj), (GBEye*)(Eye), \
  (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBSetPodTool(GB, ToTool, Obj, Eye, Hand, Tool, Ink, Layer) \
  _GBSetPodTool(GB, (GBTool*)(ToTool), (void*)(Obj), (GBEye*)(Eye), \
  (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBSetPodInk(GB, ToInk, Obj, Eye, Hand, Tool, Ink, Layer) \
  _GBSetPodInk(GB, (GBInk*)(ToInk), (void*)(Obj), (GBEye*)(Eye), \
  (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)

#define GBSetPodLayer(GB, ToLayer, Obj, Eye, Hand, Tool, Ink, Layer) \
  _GBSetPodLayer(GB, ToLayer, (void*)(Obj), (GBEye*)(Eye), \
  (GBHand*)(Hand), (GBTool*)(Tool), (GBInk*)(Ink), Layer)
```

```
#define GBNotifyChangeFromObj(GB, Obj) \
  _GBNotifyChangeFromObj(GB, (void*)(Obj))

#define GBNotifyChangeFromEye(GB, Eye) \
  _GBNotifyChangeFromEye(GB, (GBEye*)(Eye))

#define GBNotifyChangeFromHand(GB, Hand) \
  _GBNotifyChangeFromHand(GB, (GBHand*)(Hand))

#define GBNotifyChangeFromTool(GB, Tool) \
  _GBNotifyChangeFromTool(GB, (GBTool*)(Tool))

#define GBNotifyChangeFromInk(GB, Ink) \
  _GBNotifyChangeFromInk(GB, (GBInk*)(Ink))

#define GBHandGetType(Hand) _Generic(Hand, \
  GBHand*: _GBHandGetType, \
  GBHandDefault*: _GBHandGetType, \
  default: PBErrInvalidPolymorphism) ((GBHand*)(Hand))

#define GBHandProcess(Hand, Pod) _Generic(Hand, \
  GBHand*: _GBHandProcess, \
  GBHandDefault*: GBHandDefaultProcess, \
  default: PBErrInvalidPolymorphism) (Hand, Pod)

#define GBObjPodSetHand(Pod, Hand) \
  _GBObjPodSetHand(Pod, (GBHand*)(Hand))

#define GBObjPodSetEye(Pod, Eye) \
  _GBObjPodSetEye(Pod, (GBEye*)(Eye))

#define GBObjPodSetTool(Pod, Tool) \
  _GBObjPodSetTool(Pod, (GBTool*)(Tool))

#define GBObjPodSetInk(Pod, Ink) \
  _GBObjPodSetInk(Pod, (GBInk*)(Ink))

#define GBEyeGetType(Eye) _Generic(Eye, \
  GBEye*: _GBEyeGetType, \
  GBEyeOrtho*: _GBEyeGetType, \
  GBEyeIsometric*: _GBEyeGetType, \
  default: PBErrInvalidPolymorphism)((GBEye*)(Eye))

#define GBEyeScale(Eye) _Generic(Eye, \
  GBEye*: _GBEyeScale, \
  GBEyeOrtho*: _GBEyeScale, \
  GBEyeIsometric*: _GBEyeScale, \
  default: PBErrInvalidPolymorphism)((GBEye*)(Eye))

#define GBEyeGetScale(Eye) _Generic(Eye, \
  GBEye*: _GBEyeGetScale, \
  GBEyeOrtho*: _GBEyeGetScale, \
  GBEyeIsometric*: _GBEyeGetScale, \
  default: PBErrInvalidPolymorphism)((GBEye*)(Eye))

#define GBEyeSetScale(Eye, Scale) _Generic(Scale, \
  float: GBEyeSetScaleFloat, \
  VecFloat3D*: GBEyeSetScaleVec, \
  default: PBErrInvalidPolymorphism)( \
    _Generic(Eye, \
      GBEye*: Eye, \
      GBEyeOrtho*: (GBEye*)Eye, \
```

```
        GBEyeIsometric*: (GBEye*)Eye, \
        default: Eye), \
      _Generic(Scale, \
        float: Scale, \
        VecFloat3D*: Scale, \
        default: Scale))

#define GBEyeOrig(Eye) _Generic(Eye, \
  GBEye*: _GBEyeOrig, \
  GBEyeOrtho*: _GBEyeOrig, \
  GBEyeIsometric*: _GBEyeOrig, \
  default: PBErrInvalidPolymorphism)((GBEye*)(Eye))

#define GBEyeGetOrig(Eye) _Generic(Eye, \
  GBEye*: _GBEyeGetOrig, \
  GBEyeOrtho*: _GBEyeGetOrig, \
  GBEyeIsometric*: _GBEyeGetOrig, \
  default: PBErrInvalidPolymorphism)((GBEye*)(Eye))

#define GBEyeSetOrig(Eye, Orig) _Generic(Eye, \
  GBEye*: _GBEyeSetOrig, \
  GBEyeOrtho*: _GBEyeSetOrig, \
  GBEyeIsometric*: _GBEyeSetOrig, \
  default: PBErrInvalidPolymorphism)((GBEye*)(Eye), Orig)

#define GBEyeGetRot(Eye) _Generic(Eye, \
  GBEye*: _GBEyeGetRot, \
  GBEyeOrtho*: _GBEyeGetRot, \
  GBEyeIsometric*: _GBEyeGetRot, \
  default: PBErrInvalidPolymorphism)((GBEye*)(Eye))

#define GBEyeSetRot(Eye, Theta) _Generic(Eye, \
  GBEye*: _GBEyeSetRot, \
  GBEyeOrtho*: _GBEyeSetRot, \
  GBEyeIsometric*: _GBEyeSetRot, \
  default: PBErrInvalidPolymorphism)((GBEye*)(Eye), Theta)

#define GBEyeProj(Eye) _Generic(Eye, \
  GBEye*: _GBEyeProj, \
  GBEyeOrtho*: _GBEyeProj, \
  GBEyeIsometric*: _GBEyeProj, \
  default: PBErrInvalidPolymorphism)((GBEye*)(Eye))

#define GBEyeProcess(Eye, Pod) _Generic(Eye, \
  GBEye*: _GBEyeProcess, \
  GBEyeOrtho*: _GBEyeProcess, \
  GBEyeIsometric*: _GBEyeProcess, \
  default: PBErrInvalidPolymorphism)((GBEye*)(Eye), Pod)

#define GBEyeFree(EyeRef) _Generic(EyeRef, \
  GBEyeOrtho**: GBEyeOrthoFree, \
  GBEyeIsometric**: GBEyeIsometricFree, \
  default: PBErrInvalidPolymorphism)(EyeRef)

#define GBToolDraw(Tool, Pod) _Generic(Tool, \
  GBTool*: _GBToolDraw, \
  GBToolPlotter*: GBToolPlotterDraw, \
  default: PBErrInvalidPolymorphism) (Tool, Pod)

#if BUILDWITHGRAPHICLIB == 0
```

```
#elif BUILDWITHGRAPHICLIB == 1


#endif

// =============== Inliner ===================

#if BUILDMODE != 0
#include "genbrush-inline.c"
#endif



#endif
```

2.2   genbrush-GTK.h

```
// ============ GENBRUSH-GTK.H ===============

#include <gtk/gtk.h>
#include <cairo.h>

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


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

typedef struct GBSurfaceApp {
  GBSurface _surf;
  // Title of the window
  char* _title;
  // Application
  GtkApplication *_app;
  // Window
  GtkWidget *_window;
  // Widget
  GtkWidget *_drawingArea;
  // cairo surface
  cairo_surface_t *_cairoSurf;
  // Idle function
  gint (*_idleFun)(gpointer);
  // Idle function time out
  int _idleMs;
  // Returned status when the application is killed
  int _returnStatus;
} GBSurfaceApp;

typedef struct GBSurfaceWidget {
  GBSurface _surf;
  // Widget
  GtkWidget *_drawingArea;
  // cairo surface
  cairo_surface_t *_cairoSurf;
} GBSurfaceWidget;

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

// Create a new GBSurfaceApp with title 'title' and
// dimensions 'dim'
GBSurfaceApp* GBSurfaceAppCreate(VecShort2D* dim, char* title);
```

```
// Free the GBSurfaceApp 'that'
void GBSurfaceAppFree(GBSurfaceApp** that);

// Render the GBSurfaceApp 'that'
// This function block the execution until the app is killed
// Return true if the status of the app when closed was 0, false else
bool GBSurfaceAppRender(GBSurfaceApp* that);

// Close the GBSurfaceApp 'that'
#if BUILDMODE != 0
inline
#endif
void GBSurfaceAppClose(GBSurfaceApp* that);

// Refresh the content of the GBSurfaceApp 'that'
#if BUILDMODE != 0
inline
#endif
void GBSurfaceAppRefresh(GBSurfaceApp* that);

// Set the idle function of the GBSurfaceApp 'that' to 'idleFun'
// with a timeout of 'idleMs'
// The interface of the 'idleFun' is
// gint tick(gpointer data)
// the argument of 'idleFun' is a pointer to GBSurfaceApp
void GBSurfaceSetIdle(GBSurfaceApp* that,
  gint (*idleFun)(gpointer), int idleMs);

// Create a new GBSurfaceWidget with dimensions 'dim'
GBSurfaceWidget* GBSurfaceWidgetCreate(VecShort2D* dim);

// Free the GBSurfaceWidget 'that'
void GBSurfaceWidgetFree(GBSurfaceWidget** that);

// Return the GtkWidget of the GBSurfaceWidget 'that'
#if BUILDMODE != 0
inline
#endif
GtkWidget* GBSurfaceWidgetGetGtkWidget(GBSurfaceWidget* that);

// Return the GtkWidget of the GBSurfaceApp 'that'
#if BUILDMODE != 0
inline
#endif
GtkWidget* GBSurfaceAppGetGtkWidget(GBSurfaceApp* that);

// Render the GBSurfaceWidget 'that'
void GBSurfaceWidgetRender(GBSurfaceWidget* that);

// Create a GenBrush with a blank GBSurfaceApp
GenBrush* GBCreateApp(VecShort2D* dim, char* title);

// Set the idle function of the GBSurfaceApp of the GenBrush 'that'
// to 'idleFun' with a timeout of 'idleMs'
// The interface of the 'idleFun' is
// gint tick(gpointer data)
// the argument of 'idleFun' is a pointer to GBSurfaceApp
// If the surface of the app is not a GBSurfaceTypeApp, do nothing
void GBSetIdle(GenBrush* that, gint (*idleFun)(gpointer), int idleMs);

// Create a GenBrush with a blank GBSurfaceWidget
GenBrush* GBCreateWidget(VecShort2D* dim);
```

```
// Return the GtkWidget of the GenBrush 'that', or NULL if the surface
// of the GenBursh is not a GBSurfaceWidget
#if BUILDMODE != 0
inline
#endif
GtkWidget* GBGetGtkWidget(GenBrush* that);
```

3 Code

3.1 genbrush.c

```
// ============ GENBRUSH.C ===============

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

#include "genbrush.h"
#if BUILDMODE == 0
#include "genbrush-inline.c"
#endif

// =============== Structure declaration ===================

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

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

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

// Convert the indice of the TGA file data to the indice in layer's
```

```
// pixel according to origin position
int GBConvertToIndexLayerForTGA(int n, GBTGAHeader* header);

// Read the header of a TGA file from the stream 'stream' and store
// it into 'header'
// Return true if it could read the header, else false
bool GBReadTGAHeader(FILE* stream, GBTGAHeader* header);

// Read the body of a TGA file from the stream 'stream' and store
// it into 'pix', given the header 'header' of the file and the
// blend mode 'blendMode'
// Return true if it could read the body, else false
bool GBReadTGABody(FILE *stream, GSet* pix, GBTGAHeader* header,
  GBLayerBlendMode blendMode);

// Create a new GBLayer with dimensions and content given by the
// TGA image on disk at location 'fileName'
// Return NULL if we couldn't create the layer
GBLayer* GBLayerCreateFromFileTGA(char* fileName);

// Save the GBSurfaceImage 'that' as a TGA file
// Return false if the image couldn't be saved, true else
bool GBSurfaceImageSaveAsTGA(GBSurfaceImage* that);

// Create a new GBSurfaceImage with one layer containing the content
// of the TGA image located at 'fileName' and dimensions equal to the
// dimensions of the image
// If the image couldn't be loaded return NULL
GBSurfaceImage* GBSurfaceImageCreateFromTGA(char* fileName);

// Draw the Point 'point' in the GBObjPod 'pod' with the
// GBToolPlotter 'that'
void GBToolPlotterDrawPoint(GBToolPlotter* that, VecFloat* point,
  GBObjPod* pod);

// Draw the Shapoid 'shap' in the GBObjPod 'pod' with the
// GBToolPlotter 'that'
void GBToolPlotterDrawShapoid(GBToolPlotter* that, Shapoid* shap,
  GBObjPod* pod);

// Draw the SCurve 'curve' in the GBObjPod 'pod' with the
// GBToolPlotter 'that'
void GBToolPlotterDrawSCurve(GBToolPlotter* that, SCurve* curve,
  GBObjPod* pod);

// Update the content of the layer 'layer' of the GenBrush 'that'
// Flush the content of the layer and redraw all the pod attached to it
void GBUpdateLayer(GenBrush* that, GBLayer* layer);

// Get the value for the projection matrix of a GBEyeOrtho
void GBEyeOrthoGetProjVal(GBEyeOrtho* that, float* val);

// Get the value for the projection matrix of a GBEyeIsometric
void GBEyeIsometricGetProjVal(GBEyeIsometric* that, float* val);

// Normalize in hue the final pixels of the GBSurface 'that'
void GBSurfaceNormalizeHue(GBSurface* that);

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

// Function to decode rgba values when loading a TGA file
// Do nothing if arguments are invalid
```

```
void GBTGAMergeBytes(GBPixel* pixel, unsigned char* p, int bytes) {
  // Check arguments
  if (pixel == NULL || p == NULL)
    return;
  // Merge bytes
  if (bytes == 4) {
    pixel->_rgba[GBPixelRed] = p[2];
    pixel->_rgba[GBPixelGreen] = p[1];
    pixel->_rgba[GBPixelBlue] = p[0];
    pixel->_rgba[GBPixelAlpha] = p[3];
  } else if (bytes == 3) {
    pixel->_rgba[GBPixelRed] = p[2];
    pixel->_rgba[GBPixelGreen] = p[1];
    pixel->_rgba[GBPixelBlue] = p[0];
    pixel->_rgba[GBPixelAlpha] = 255;
  } else if (bytes == 2) {
    pixel->_rgba[GBPixelRed] = (p[1] & 0x7c) << 1;
    pixel->_rgba[GBPixelGreen] =
      ((p[1] & 0x03) << 6) | ((p[0] & 0xe0) >> 2);
    pixel->_rgba[GBPixelBlue] = (p[0] & 0x1f) << 3;
    pixel->_rgba[GBPixelAlpha] = (p[1] & 0x80);
  }
}

// Convert the indice of the TGA file data to the indice in layer's
// pixel according to origin position
int GBConvertToIndexLayerForTGA(int n, GBTGAHeader* header) {
  if ((header->_imageDescriptor & 32) == 0)
    // lower left hand corner
    return n;
  else
    // upper left hand corner
    return (header->_height - 1 -
      ((int)floor((float)n / (float)(header->_width)))) *
      header->_width + n % header->_width;
}

// Read the header of a TGA file from the stream 'stream' and store
// it into 'header'
// Return true if it could read the header, else false
bool GBReadTGAHeader(FILE* stream, GBTGAHeader* header) {
  // Declare a variable to memorize the return value of fread
  size_t ret = 0;
  // Read the header's values
  header->_idLength = fgetc(stream);
  header->_colorMapType = fgetc(stream);
  // If the color map type is not supported
  if (header->_colorMapType != 0 &&
    header->_colorMapType != 1) {
    // Stop here
    return false;
  }
  header->_dataTypeCode = fgetc(stream);
  // If the data type is not supported
  if (header->_dataTypeCode != 2 && header->_dataTypeCode != 10) {
    // Stop here
    return false;
  }
  ret = fread(&(header->_colorMapOrigin), 2, 1, stream);
  if (ret == 0) {
    // Stop here
    return false;
```

```
  }
  ret = fread(&(header->_colorMapLength), 2, 1, stream);
  if (ret == 0) {
    // Stop here
    return false;
  }
  header->_colorMapDepth = fgetc(stream);
  ret = fread(&(header->_xOrigin), 2, 1, stream);
  if (ret == 0) {
    // Stop here
    return false;
  }
  ret = fread(&(header->_yOrigin), 2, 1, stream);
  if (ret == 0) {
    // Stop here
    return false;
  }
  ret = fread(&(header->_width), 2, 1, stream);
  if (ret == 0 || header->_width <= 0) {
    // Stop here
    return false;
  }
  ret = fread(&(header->_height), 2, 1, stream);
  if (ret == 0 || header->_height <= 0) {
    // Stop here
    return false;
  }
  header->_bitsPerPixel = fgetc(stream);
  header->_imageDescriptor = fgetc(stream);
  // Origin of the screen
  // TODO: Override the initial _yOrigin
  /*if ((header->_imageDescriptor & 32) == 0)
    // lower left hand corner
    header->_yOrigin = header->_height;
  else
    // upper left hand corner
    header->_yOrigin = 0;*/
  // Return the success code
  return true;
}

// Read the body of a TGA file from the stream 'stream' and store
// it into 'pix', given the header 'header' of the file and the
// blend mode 'blendMode'
// Return true if it could read the body, else false
bool GBReadTGABody(FILE *stream, GSet* pix, GBTGAHeader* header,
  GBLayerBlendMode blendMode) {
  // Declare variables used during decoding
  int i = 0, j = 0;
  unsigned int bytes2read = 0, skipover = 0;
  unsigned char p[5] = {0};
  // If the number of byte per pixel is not supported
  if (header->_bitsPerPixel != 16 &&
    header->_bitsPerPixel != 24 &&
    header->_bitsPerPixel != 32)
    return false;
  // Skip the unused information
  skipover += header->_idLength;
  skipover += header->_colorMapType * header->_colorMapLength;
  fseek(stream,skipover,SEEK_CUR);
  // Calculate the number of byte per pixel
  bytes2read = header->_bitsPerPixel / 8;
```

46

```
// For each pixel
int n = 0;
while (n < header->_height * header->_width) {
  int nLayer = GBConvertToIndexLayerForTGA(n, header);
  // Read the pixel according to the data type, merge and
  // move to the next pixel
  if (header->_dataTypeCode == 2) {
    if (fread(p, 1, bytes2read, stream) != bytes2read)
      return false;
    GBPixel mergedPix;
    GBTGAMergeBytes(&mergedPix, p, bytes2read);
    // Create a stacked pixel
    GBStackedPixel* stacked = PBErrMalloc(GenBrushErr,
      sizeof(GBStackedPixel));
    memcpy(&(stacked->_val), &mergedPix, sizeof(GBPixel));
    stacked->_depth = 0.0;
    stacked->_blendMode = blendMode;
    // Add the stacked pixel to the stack
    GSetAddSort(pix + nLayer, stacked, stacked->_depth);
    // Increment the index
    ++n;
    nLayer = GBConvertToIndexLayerForTGA(n, header);
  } else if (header->_dataTypeCode == 10) {
    if (fread(p, 1, bytes2read + 1, stream) != bytes2read + 1)
      return false;
    j = p[0] & 0x7f;
    GBPixel mergedPix;
    GBTGAMergeBytes(&mergedPix, &(p[1]), bytes2read);
    // Create a stacked pixel
    GBStackedPixel* stacked = PBErrMalloc(GenBrushErr,
      sizeof(GBStackedPixel));
    memcpy(&(stacked->_val), &mergedPix, sizeof(GBPixel));
    stacked->_depth = 0.0;
    stacked->_blendMode = blendMode;
    // Add the stacked pixel to the stack
    GSetAddSort(pix + nLayer, stacked, stacked->_depth);
    // Increment the index
    ++n;
    nLayer = GBConvertToIndexLayerForTGA(n, header);
    if (p[0] & 0x80) {
      for (i = 0; i < j; ++i) {
        GBPixel mergedPix;
        GBTGAMergeBytes(&mergedPix, &(p[1]), bytes2read);
        // Create a stacked pixel
        GBStackedPixel* stacked = PBErrMalloc(GenBrushErr,
          sizeof(GBStackedPixel));
        memcpy(&(stacked->_val), &mergedPix, sizeof(GBPixel));
        stacked->_depth = 0.0;
        stacked->_blendMode = blendMode;
        // Add the stacked pixel to the stack
        GSetAddSort(pix + nLayer, stacked, stacked->_depth);
        // Increment the index
        ++n;
        nLayer = GBConvertToIndexLayerForTGA(n, header);
      }
    } else {
      for (i = 0; i < j; ++i) {
        if (fread(p, 1, bytes2read, stream) != bytes2read)
          return false;
        GBPixel mergedPix;
        GBTGAMergeBytes(&mergedPix, p, bytes2read);
        // Create a stacked pixel
```

```
            GBStackedPixel* stacked = PBErrMalloc(GenBrushErr,
              sizeof(GBStackedPixel));
            memcpy(&(stacked->_val), &mergedPix, sizeof(GBPixel));
            stacked->_depth = 0.0;
            stacked->_blendMode = blendMode;
            // Add the stacked pixel to the stack
            GSetAddSort(pix + nLayer, stacked, stacked->_depth);
            // Increment the index
            ++n;
            nLayer = GBConvertToIndexLayerForTGA(n, header);
          }
        }
      }
    }
  }
  // Return the success code
  return true;
}


// --------------- GBPixel ------------------------

// Blend the pixel 'pix' into the pixel 'that'
// BlendNormal mixes colors according to their relative alpha value
// and add the alpha values
void GBPixelBlendNormal(GBPixel* that, GBPixel* pix) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pix == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pix' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare a variable for computation
  float u = 0.5 *
    (1.0 + ((float)(that->_rgba[GBPixelAlpha]) -
    (float)(pix->_rgba[GBPixelAlpha])) / 255.0);
  for (int iRgb = 4; iRgb--;)
    // Add 0.5 to the conversion to float ot avoid numeric instability
    if (iRgb == GBPixelAlpha)
      that->_rgba[iRgb] = (unsigned char)floor(
        MIN(255.0, (float)(that->_rgba[iRgb]) +
        (float)(pix->_rgba[iRgb])) + 0.5);
    else
      that->_rgba[iRgb] = (unsigned char)floor(
        (1.0 - u) * (float)(that->_rgba[iRgb]) +
        u * (float)(pix->_rgba[iRgb]) + 0.5);
}

// Blend the pixel 'pix' into the pixel 'that'
// BlendOver mixes colors according to the alpha value of 'pix'
// and add the alpha values
void GBPixelBlendOver(GBPixel* that, GBPixel* pix) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
```

```
    if (pix == NULL) {
      GenBrushErr->_type = PBErrTypeNullPointer;
      sprintf(GenBrushErr->_msg, "'pix' is null");
      PBErrCatch(GenBrushErr);
    }
#endif
  // Declare a variable for computation
  float u = (float)(that->_rgba[GBPixelAlpha]) / 255.0;
  for (int iRgb = 4; iRgb--;)
    // Add 0.5 to the conversion to float ot avoid numeric instability
    if (iRgb == GBPixelAlpha)
      that->_rgba[iRgb] = (unsigned char)floor(
        MIN(255.0, (float)(that->_rgba[iRgb]) +
        (float)(pix->_rgba[iRgb])) + 0.5);
    else
      that->_rgba[iRgb] = (unsigned char)floor(
        (1.0 - u) * (float)(that->_rgba[iRgb]) +
        u * (float)(pix->_rgba[iRgb]) + 0.5);
}

// Return the blend result of the stack of Pixel 'stack'
// If there is transparency down to the bottom of the stack, use the
// background color 'bgColor'
// If the stack is empty, return a transparent pixel
GBPixel GBPixelStackBlend(GSet* stack, GBPixel* bgColor) {
#if BUILDMODE == 0
  if (stack == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'stack' is null");
    PBErrCatch(GenBrushErr);
  }
  if (bgColor == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'bgColor' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare the result pixel
  GBPixel res = GBColorTransparent;
  // Declare a variable to memorize the blend mode
  GBLayerBlendMode blendMode = GBLayerBlendModeNormal;
  // If the stack is not empty
  if (GSetNbElem(stack) > 0) {
    // Initalise the result pixel with the last pixel
    GSetIterBackward iter = GSetIterBackwardCreateStatic(stack);
    GBStackedPixel* pix = GSetIterGet(&iter);
    blendMode = pix->_blendMode;
    res = pix->_val;
    // For each following pixel until we reach the bottom of the stack
    // or the opacity reaches 255
    while (res._rgba[GBPixelAlpha] < 255 && GSetIterStep(&iter)) {
      pix = GSetIterGet(&iter);
      switch (blendMode) {
        case GBLayerBlendModeDefault:
          break;
        case GBLayerBlendModeNormal:
          GBPixelBlendNormal(&res, &(pix->_val));
          break;
        case GBLayerBlendModeOver:
          GBPixelBlendOver(&res, &(pix->_val));
          break;
        default:
```

```
        break;
      }
      blendMode = pix->_blendMode;
    }
    // If we've reached the bottom of the stack and there is still
    // transparency
    if (res._rgba[GBPixelAlpha] < 255) {
      // Add the background color
      switch (blendMode) {
        case GBLayerBlendModeDefault:
          break;
        case GBLayerBlendModeNormal:
          GBPixelBlendNormal(&res, bgColor);
          break;
        case GBLayerBlendModeOver:
          GBPixelBlendOver(&res, bgColor);
          break;
        default:
          break;
      }
    }
  // Else, the stack is empty
  } else {
    // Simply copy the background pixel;
    res = *bgColor;
  }
  // Return the result pixel
  return res;
}


// --------------- GBLayer ------------------------

// Create a new GBLayer with dimensions 'dim'
// The layer is initialized with empty stacks of pixel
// _pos = (0,0)
// blendMode = GBLayerBlendModeDefault
// stackPos = GBLayerStackPosBg
GBLayer* GBLayerCreate(VecShort2D* dim) {
#if BUILDMODE == 0
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
  }
  if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'dim' is invalid (%d>0,%d>0)",
      VecGet(dim, 0), VecGet(dim, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare a variable to memorize the new layer
  GBLayer* that = PBErrMalloc(GenBrushErr, sizeof(GBLayer));
  // Set properties
  that->_dim = *dim;
  that->_pos = VecShortCreateStatic2D();
  that->_prevPos = VecShortCreateStatic2D();
  that->_pix = PBErrMalloc(GenBrushErr,
    sizeof(GSet) * GBLayerArea(that));
  for (int iPix = GBLayerArea(that); iPix--;)
    that->_pix[iPix] = GSetCreateStatic();
  that->_blendMode = GBLayerBlendModeDefault;
```

```
    that->_modified = true;
    that->_stackPos = GBLayerStackPosBg;
    // Return the new layer
    return that;
}

// Free the GBLayer 'that'
void GBLayerFree(GBLayer** that) {
    if (that == NULL || *that == NULL) {
        return;
    }
    // Free the memory
    if ((*that)->_pix != NULL) {
        GBLayerFlush(*that);
        free((*that)->_pix);
    }
    free(*that);
    *that = NULL;
}

// Create a new GBLayer with dimensions and content given by the
// image on disk at location 'fileName'
// Return NULL if we couldn't create the layer
GBLayer* GBLayerCreateFromFile(char* fileName) {
    // If the fileName is NULL
    if (fileName == NULL)
        // Nothing to do
        return NULL;
    // Call the appropriate function depending on file extension
    if (strcmp(fileName + strlen(fileName) - 4, ".tga") == 0 ||
        strcmp(fileName + strlen(fileName) - 4, ".TGA") == 0)
        return GBLayerCreateFromFileTGA(fileName);
    return NULL;
}

// Create a new GBLayer with dimensions and content given by the
// TGA image on disk at location 'fileName'
// Return NULL if we couldn't create the layer
GBLayer* GBLayerCreateFromFileTGA(char* fileName) {
    // Open the file
    FILE* fptr = fopen(fileName,"r");
    // If we couldn't open the file
    if (fptr == NULL)
        // Stop here
        return NULL;
    // Declare a variable to memorize the header
    GBTGAHeader h;
    // Read the header
    if (GBReadTGAHeader(fptr, &h) == false) {
        // Stop here
        fclose(fptr);
        return NULL;
    }
    // Create the layer
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, h._width);
    VecSet(&dim, 1, h._height);
    GBLayer* layer = GBLayerCreate(&dim);
    // Set a pointer to the pixels of the new layer
    GSet* pix = GBLayerPixels(layer);
    // Read the body of the TGA file
    if (GBReadTGABody(fptr, pix, &h, layer->_blendMode) == false) {
```

```c
    // Stop here
    GBLayerFree(&layer);
    fclose(fptr);
    return NULL;
  }
  // Close the file
  fclose(fptr);
  // Return the new layer
  return layer;
}

// Get the boundary of the GBLayer 'that' inside the GBSurface 'surf'
// The boundaries are given as a Facoid
// If the flag 'prevPos' is true, gives the bounday at the previous
// position
// Return NULL if the layer is completely out of the surface
Facoid* GBLayerGetBoundaryInSurface(GBLayer* that, GBSurface* surf,
  bool prevPos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (surf == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'surf' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare a pointer toward the relevant position accoridng to
  // 'prevPos'
  VecShort2D* pos =
    (prevPos == false ? GBLayerPos(that) : GBLayerPrevPos(that));
  // Get the values of the clip area made by the intersection of the
  // layer and the surface
  VecFloat2D min = VecFloatCreateStatic2D();
  VecFloat2D max = VecFloatCreateStatic2D();
  for (int i = 2; i--;) {
    VecSet(&min, i, MAX(0, VecGet(pos, i)));
    VecSet(&max, i, MIN(VecGet(GBSurfaceDim(surf), i),
      VecGet(GBLayerDim(that), i) + VecGet(pos, i)));
  }
  // Declare a Facoid to memorize the result
  Facoid* bound = NULL;
  // If the layer is in intersection with the surface
  VecOp(&max, 1, &min, -1);
  if (VecGet(&max, 0) > 0.0 &&
    VecGet(&max, 1) > 0.0) {
    // Set the values of the Facoid
    bound = FacoidCreate(2);
    ShapoidSetPos(bound, (VecFloat*)&min);
    ShapoidScale(bound, (VecFloat*)&max);
  }
  // Return the result
  return bound;
}

// --------------- GBPostProcessing ------------------------

// Create a static GBPostProcessing with type 'type'
GBPostProcessing* GBPostProcessingCreate(GBPPType type) {
```

```c
  // Declare the new GBPostProcessing
  GBPostProcessing* ret = PBErrMalloc(GenBrushErr,
    sizeof(GBPostProcessing));
  // Set the properties
  ret->_type = type;
  // Return the new post processing
  return ret;
}

// Create a new static GBPostProcessing with type 'type'
GBPostProcessing GBPostProcessingCreateStatic(GBPPType type) {
  // Declare the new GBPostProcessing
  GBPostProcessing ret;
  // Set the properties
  ret._type = type;
  // Return the new post processing
  return ret;
}

// Free the memory used by the GBPostProcessing 'that'
void GBPostProcessingFree(GBPostProcessing** that) {
  // Check argument
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free memory
  free(*that);
  *that = NULL;
}


// --------------- GBSurface -------------------------

// Create a new static GBSurface with dimension 'dim' and type 'type'
// _finalPix is set to 0
// _bgColor is set to white
GBSurface GBSurfaceCreateStatic(GBSurfaceType type, VecShort2D* dim) {
#if BUILDMODE == 0
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
  }
  if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'dim' is invalid (%d>0,%d>0)",
      VecGet(dim, 0), VecGet(dim, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare the new GBSurface
  GBSurface ret;
  // Set properties
  ret._type = type;
  ret._dim = *dim;
  ret._bgColor = GBColorWhite;
  ret._layers = GSetCreateStatic();
  // Allocate memory for the final pixels
  ret._finalPix = PBErrMalloc(GenBrushErr,
    sizeof(GBPixel) * VecGet(dim, 0) * VecGet(dim, 1));
  memset(ret._finalPix, 0,
    sizeof(GBPixel) * VecGet(dim, 0) * VecGet(dim, 1));
  // Flush the surface to initialize the _finalPix with the _bgColor
```

53

```
    GBSurfaceFlush(&ret);
    // Return the new GBSurface
    return ret;
}

// Create a new GBSurface with dimension 'dim' and type 'type'
// _finalPix is set to 0
// _bgColor is set to white
GBSurface* GBSurfaceCreate(GBSurfaceType type, VecShort2D* dim) {
#if BUILDMODE == 0
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
  }
  if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'dim' is invalid (%d>0,%d>0)",
      VecGet(dim, 0), VecGet(dim, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare the new GBSurface
  GBSurface *ret = PBErrMalloc(GenBrushErr, sizeof(GBSurface));
  // Set properties
  *ret = GBSurfaceCreateStatic(type, dim);
  // Return the new surface
  return ret;
}

// Free the memory used by the properties of the GBSurface 'that'
void GBSurfaceFreeStatic(GBSurface* that) {
  // If the pointer is null
  if (that == NULL)
    // Nothing to do
    return;
  // Free the memory
  while (GSetNbElem(&(that->_layers)) > 0) {
    GBLayer* layer = (GBLayer*)GSetPop(&(that->_layers));
    GBLayerFree(&layer);
  }
  free(that->_finalPix);
  that->_finalPix = NULL;
}

// Free the memory used by the GBSurface 'that'
void GBSurfaceFree(GBSurface** that) {
  // If the pointer is null
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free the memory
  GBSurfaceFreeStatic(*that);
  free(*that);
  *that = NULL;
}

// Clone the GBSurface 'that'
GBSurface GBSurfaceClone(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
```

```
      sprintf(GenBrushErr->_msg, "'that' is null");
      PBErrCatch(GenBrushErr);
  }
#endif
  // Declare the clone
  GBSurface clone = *that;
  // Clone properties
  clone._finalPix = PBErrMalloc(GenBrushErr,
    sizeof(GBPixel) * GBSurfaceArea(that));
  memcpy(clone._finalPix, GBSurfaceFinalPixels(that),
    sizeof(GBPixel) * GBSurfaceArea(that));
  // Return the clone
  return clone;
}

// Get a GSet of Facoid representing the sub areas of the GBSurface
// 'that' affected by layers with _modified flag equal to true
// If there is no modified sub area return an empty GSet
GSetShapoid* GBSurfaceGetModifiedArea(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare the GSet to memorize the areas
  GSetShapoid* areas = GSetShapoidCreate();
  // If the surface has layers
  if (GBSurfaceNbLayer(that) > 0) {
    // For each layer of the surface
    GSetIterForward iter =
      GSetIterForwardCreateStatic(GBSurfaceLayers(that));
    do {
      // Get the current layer
      GBLayer* layer = GSetIterGet(&iter);
      // If the layer is modified
      if (GBLayerIsModified(layer)) {
        // Get the boundary in surface of this layer
        Facoid* bound =
          GBLayerGetBoundaryInSurface(layer, that, false);
        // If the boundary is not empty
        if (bound != NULL) {
          // Add the boundary to the areas taking care there is no
          // overlapping areas
          FacoidAlignedAddClippedToSet(bound, areas);
          // Free memory used by the boundary
          ShapoidFree(&bound);
        }
        // If the layer has been moved we also need to repaint the
        // area at its previous position
        if (!VecIsEqual(GBLayerPos(layer), GBLayerPrevPos(layer))) {
          // Get the boundary in surface of this layer at previous
          // position
          bound = GBLayerGetBoundaryInSurface(layer, that, true);
          // If the boundary is not empty
          if (bound != NULL) {
            // Add the boundary to the areas taking care there is no
            // overlapping areas
            FacoidAlignedAddClippedToSet(bound, areas);
            // Free memory used by the boundary
            ShapoidFree(&bound);
```

```
          }
        }
      }
    } while (GSetIterStep(&iter));
  }
  // Return the areas
  return areas;
}


// Update the final pixels according to layers of the GBSurface 'that'
// Update only pixels affected by layers with the _modified flag
// equals to true
void GBSurfaceUpdate(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If the surface has layers
  if (GBSurfaceNbLayer(that) > 0) {
    // Get the areas needing to be updated
    GSetShapoid* areas = GBSurfaceGetModifiedArea(that);
    // Declare some vectors to memorize pixel coordinates
    VecShort2D from = VecShortCreateStatic2D();
    VecShort2D to = VecShortCreateStatic2D();
    VecShort2D pCoord = VecShortCreateStatic2D();
    VecShort2D pLayer = VecShortCreateStatic2D();
    // Declare 3 sets for the blending of pixels
    GSet setBg = GSetCreateStatic();
    GSet setIn = GSetCreateStatic();
    GSet setFg = GSetCreateStatic();
    // For each area
    while (GSetNbElem(areas) > 0) {
      // Get the current area
      Facoid* area = (Facoid*)GSetPop(areas);
      // Facoid values are float, we need to convert to short for
      // pixel coordinates
      for (int i = 2; i--;) {
        VecSet(&from, i, (short)round(VecGet(ShapoidPos(area), i)));
        VecSet(&to, i, (short)round(VecGet(ShapoidPos(area), i) +
          VecGet(ShapoidAxis(area, i), i)));
      }
      // For each pixel in the current area
      VecCopy(&pCoord, &from);
      do {
        // For each layer
        GSetIterForward iter =
          GSetIterForwardCreateStatic(GBSurfaceLayers(that));
        do {
          // Get the current layer
          GBLayer* layer = GSetIterGet(&iter);
          // Get the position of the current pixel in layer coordinates
          VecCopy(&pLayer, &pCoord);
          VecOp(&pLayer, 1.0, GBLayerPos(layer), -1.0);
          // If the current pixel is inside the layer
          if (GBLayerIsPosInside(layer, &pLayer)) {
            // Add the stacked pixels of the layer to the appropriate
            // set
            switch (GBLayerGetStackPos(layer)) {
              case GBLayerStackPosBg:
```

```
                  GSetAppendSet(&setBg, GBLayerPixel(layer, &pLayer));
                  break;
               case GBLayerStackPosInside:
                  GSetAppendSortedSet(&setIn,
                    GBLayerPixel(layer, &pLayer));
                  break;
               case GBLayerStackPosFg:
                  GSetAppendSet(&setFg, GBLayerPixel(layer, &pLayer));
                  break;
               default:
                  break;
            }
          }
        } while (GSetIterStep(&iter));
        // Merge the 3 stacks
        GSetAppendSet(&setBg, &setIn);
        GSetAppendSet(&setBg, &setFg);
        // Update the final pixel with the blend of the stack
        GBPixel* finalPix = GBSurfaceFinalPixel(that, &pCoord);
        *finalPix = GBPixelStackBlend(&setBg, GBSurfaceBgColor(that));
        // Empty the sets of stacked pixels
        GSetFlush(&setBg);
        GSetFlush(&setIn);
        GSetFlush(&setFg);
      } while (VecShiftStep(&pCoord, &from ,&to));
      // Free the memory used by the area
      ShapoidFree(&area);
    }
    // Free memory
    GSetFree(&areas);
    // Reset the modified flag of layers
    GBSurfaceSetLayersModified(that, false);
  }
}

// Reset all the final pix of the GBSurface 'that' to its
// background color, and reset all the modified flag of layers to true
void GBSurfaceFlush(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Set all the pixel to the background color
  for (int iPix = GBSurfaceArea(that); iPix--;)
    memcpy(that->_finalPix + iPix, &(that->_bgColor), sizeof(GBPixel));
  // Reset the modified flag of layers
  GBSurfaceSetLayersModified(that, true);
}

// Apply the post processing 'post' to the final pixels in the
// GBSurface 'that'
void GBSurfacePostProcess(GBSurface* that, GBPostProcessing* post) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (post == NULL) {
```

```
      GenBrushErr->_type = PBErrTypeNullPointer;
      sprintf(GenBrushErr->_msg, "'post' is null");
      PBErrCatch(GenBrushErr);
  }
#endif
  // Call the appropriate post processing
  switch(GBPostProcessingGetType(post)) {
    case GBPPTypeNormalizeHue:
      GBSurfaceNormalizeHue(that);
      break;
    default:
      return;
  }
}


// Normalize in hue the final pixels of the GBSurface 'that'
void GBSurfaceNormalizeHue(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare a variable to memorize the maximum possible intensity
  float maxIntensity = 255.0;
  // Declare two variables to memorize the minimum and maximum hue
  // intensity
  float min = maxIntensity;
  float max = 0.0;
  // Search the minimum and maximum intensity
  VecShort2D p = VecShortCreateStatic2D();
  do {
    GBPixel* pix = GBSurfaceFinalPixel(that, &p);
    float intensity = (float)MAX(pix->_rgba[GBPixelRed],
      MAX(pix->_rgba[GBPixelGreen], pix->_rgba[GBPixelBlue]));
    if (intensity > max)
      max = intensity;
    intensity = (float)MIN(pix->_rgba[GBPixelRed],
      MIN(pix->_rgba[GBPixelGreen], pix->_rgba[GBPixelBlue]));
    if (intensity < min)
      min = intensity;
  } while (VecStep(&p, GBSurfaceDim(that)));
  // If the image is not already normalized and is normalizable
  if (!ISEQUALF(min, max) && (min >= 1.0 || max <= maxIntensity)) {
    // Declare a variable for optimization
    float c = maxIntensity / (max - min);
    // Normalize each final pixel color
    VecSetNull(&p);
    do {
      GBPixel* pix = GBSurfaceFinalPixel(that, &p);
      float intensity = ((float)(pix->_rgba[GBPixelRed]) - min) * c;
      pix->_rgba[GBPixelRed] =
        (unsigned char)round(MIN(255.0, MAX(0.0, intensity)));
      intensity = ((float)(pix->_rgba[GBPixelGreen]) - min) * c;
      pix->_rgba[GBPixelGreen] =
        (unsigned char)round(MIN(255.0, MAX(0.0, intensity)));
      intensity = ((float)(pix->_rgba[GBPixelBlue]) - min) * c;
      pix->_rgba[GBPixelBlue] =
        (unsigned char)round(MIN(255.0, MAX(0.0, intensity)));
    } while (VecStep(&p, GBSurfaceDim(that)));
  }
```

```
  }

// ---------------- GBSurfaceImage -------------------------

// Create a new GBSurfaceImage with dimension 'dim'
GBSurfaceImage* GBSurfaceImageCreate(VecShort2D* dim) {
#if BUILDMODE == 0
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
  }
  if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'dim' is invalid (%d>0,%d>0)",
      VecGet(dim, 0), VecGet(dim, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare the new GBSurfaceImage
  GBSurfaceImage* ret = PBErrMalloc(GenBrushErr,
    sizeof(GBSurfaceImage));
  // Set properties
  ret->_surf = GBSurfaceCreateStatic(GBSurfaceTypeImage, dim);
  ret->_fileName = NULL;
  // Return the new surface
  return ret;
}

// Free the memory used by the GBSurfaceImage 'that'
void GBSurfaceImageFree(GBSurfaceImage** that) {
  // If the pointer is null
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free the memory
  GBSurfaceFreeStatic((GBSurface*)(*that));
  if ((*that)->_fileName != NULL) {
    free((*that)->_fileName);
    (*that)->_fileName = NULL;
  }
  free(*that);
  *that = NULL;
}

// Clone the GBSurfaceImage 'that'
GBSurfaceImage* GBSurfaceImageClone(GBSurfaceImage* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare the clone
  GBSurfaceImage* clone =
    GBSurfaceImageCreate(GBSurfaceDim((GBSurface*)that));
  // Declare a variable ot memorize he newly allocated stack of pixels
  GBPixel* nFinal = clone->_surf._finalPix;
  // Clone properties
  memcpy(clone, that, sizeof(GBSurface));
  clone->_surf._finalPix = nFinal;
```

```
    memcpy(clone->_surf._finalPix,
      GBSurfaceFinalPixels((GBSurface*)that),
      sizeof(GBPixel) * GBSurfaceArea((GBSurface*)that));
    clone->_fileName = NULL;
    GBSurfaceImageSetFileName(clone, GBSurfaceImageFileName(that));
    // Return the clone
    return clone;
}

// Save a GBSurfaceImage 'that'
// If the filename is not set do nothing and return false
// Return true if it could save the surface, false else
bool GBSurfaceImageSave(GBSurfaceImage* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (that->_fileName == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that->_fileName' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If the file name is null, do nothing
  if (that->_fileName == NULL)
    return false;
  // Call the appropriate function depending on file extension
  if (strcmp(that->_fileName + strlen(that->_fileName) - 4,
    ".tga") == 0 ||
    strcmp(that->_fileName + strlen(that->_fileName) - 4,
    ".TGA") == 0)
    return GBSurfaceImageSaveAsTGA(that);
  return false;
}

// Save the GBSurfaceImage 'that' as a TGA file
// Return false if the image couldn't be saved, true else
bool GBSurfaceImageSaveAsTGA(GBSurfaceImage* that) {
  int ret;
  // Open the file
  FILE* fptr = fopen(that->_fileName, "w");
  // If we couln't open the file
  if (fptr == NULL)
    // Stop here
    return false;
  // Declare a header
  GBTGAHeader h;
  // Create the header
  h._idLength = 0;
  h._colorMapType = 0;
  h._dataTypeCode = 2;
  h._colorMapOrigin = 0;
  h._colorMapLength = 0;
  h._colorMapDepth = 0;
  h._width = VecGet(GBSurfaceDim((GBSurface*)that), 0);
  h._height = VecGet(GBSurfaceDim((GBSurface*)that), 1);
  h._xOrigin = 0;
  h._yOrigin = 0;
  h._bitsPerPixel = 32;
  h._imageDescriptor = 0;
```

```
// Write the header
ret = putc(h._idLength, fptr);
if (ret == EOF) {
  fclose(fptr);
  return false;
}
ret = putc(h._colorMapType, fptr);
if (ret == EOF) {
  fclose(fptr);
  return false;
}
ret = putc(h._dataTypeCode, fptr);
if (ret == EOF) {
  fclose(fptr);
  return false;
}
ret = fwrite(&(h._colorMapOrigin), 2, 1, fptr);
if (ret == 0) {
  fclose(fptr);
  return false;
}
ret = fwrite(&(h._colorMapLength), 2, 1, fptr);
if (ret == 0) {
  fclose(fptr);
  return false;
}
ret = putc(h._colorMapDepth, fptr);
if (ret == EOF) {
  fclose(fptr);
  return false;
}
ret = fwrite(&(h._xOrigin), 2, 1, fptr);
if (ret == 0) {
  fclose(fptr);
  return false;
}
ret = fwrite(&(h._yOrigin), 2, 1, fptr);
if (ret == 0) {
  fclose(fptr);
  return false;
}
ret = fwrite(&(h._width), 2, 1, fptr);
if (ret == 0) {
  fclose(fptr);
  return false;
}
ret = fwrite(&(h._height), 2, 1, fptr);
if (ret == 0) {
  fclose(fptr);
  return false;
}
ret = putc(h._bitsPerPixel, fptr);
if (ret == EOF) {
  fclose(fptr);
  return false;
}
ret = putc(h._imageDescriptor, fptr);
if (ret == EOF) {
  fclose(fptr);
  return false;
}
```

```
ret = putc(h._idLength, fptr);
```

```
    // Declare a variable to memorize the final layer's pixel
    GBPixel* pix = GBSurfaceFinalPixels((GBSurface*)that);
    // For each pixel
    for (int y = 0; y < h._height; ++y) {
      for (int x = 0; x < h._width; ++x) {
        // Get the index of the pixel
        int i = y * h._width + x;
        // Write the pixel values
        ret = putc(pix[i]._rgba[GBPixelBlue], fptr);
        if (ret == EOF) {
          fclose(fptr);
          return false;
        }
        ret = putc(pix[i]._rgba[GBPixelGreen], fptr);
        if (ret == EOF) {
          fclose(fptr);
          return false;
        }
        ret = putc(pix[i]._rgba[GBPixelRed], fptr);
        if (ret == EOF) {
          fclose(fptr);
          return false;
        }
        ret = putc(pix[i]._rgba[GBPixelAlpha], fptr);
        if (ret == EOF) {
          fclose(fptr);
          return false;
        }
      }
    }
    // Close the file
    fclose(fptr);
    // Return the success code
    return true;
}

// Create a new GBSurfaceImage with one layer containing the content
// of the image located at 'fileName' and dimensions equal to the
// dimensions of the image
// If the image couldn't be loaded return NULL
GBSurfaceImage* GBSurfaceImageCreateFromFile(char* fileName) {
#if BUILDMODE == 0
  if (fileName == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'fileName' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If the file name is null, return NULL
  if (fileName == NULL)
    return NULL;
  // Call the appropriate function depending on file extension
  if (strcmp(fileName + strlen(fileName) - 4, ".tga") == 0 ||
    strcmp(fileName + strlen(fileName) - 4, ".TGA") == 0)
    return GBSurfaceImageCreateFromTGA(fileName);
  return NULL;
}

// Create a new GBSurfaceImage with one layer containing the content
// of the TGA image located at 'fileName' and dimensions equal to the
// dimensions of the image
// If the image couldn't be loaded return NULL
```

```
GBSurfaceImage* GBSurfaceImageCreateFromTGA(char* fileName) {
#if BUILDMODE == 0
  if (fileName == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'fileName' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Load the image into a GBLayer
  GBLayer* layer = GBLayerCreateFromFileTGA(fileName);
  // If we couldn't load the image
  if (layer == NULL)
    // Return null
    return NULL;
  // Create a new GBSurfaceImage with same dimension as the layer
  GBSurfaceImage* surf = GBSurfaceImageCreate(GBLayerDim(layer));
  // Set the background color to transparent
  GBPixel transparent = GBColorTransparent;
  GBSurfaceSetBgColor((GBSurface*)surf, &transparent);
  // Add the layer to the surf
  GSetAppend(GBSurfaceLayers((GBSurface*)surf), layer);
  // Update the surface
  GBSurfaceUpdate((GBSurface*)surf);
  // Return the new surface
  return surf;
}

// ---------------- GBEye ------------------------

// Return a new GBEye with type 'type'
// scale is initialized to (1,1), trans to (0,0) and rot to 0
GBEye GBEyeCreateStatic(GBEyeType type) {
  // Declare a GBEye
  GBEye that;
  // Set properties
  that._type = type;
  that._scale = VecFloatCreateStatic3D();
  VecSet(&(that._scale), 0, 1.0); VecSet(&(that._scale), 1, 1.0);
  VecSet(&(that._scale), 2, 1.0);
  that._orig = VecFloatCreateStatic2D();
  VecSet(&(that._orig), 0, 0.0); VecSet(&(that._orig), 1, 0.0);
  that._theta = 0.0;
  // Allocate memory for the projection matrix
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3); VecSet(&dim, 1, 3);
  that._proj = MatFloatCreate(&dim);
  // Return the new GBEye
  return that;
}

// Free the memory used by the GBEye
void GBEyeFreeStatic(GBEye* that) {
  if (that == NULL)
    return;
  // Free memory
  MatFree(&(that->_proj));
}

// Update the projection matrix of the GBEye according to scale, rot
// and origin
void GBEyeUpdateProj(GBEye* that) {
#if BUILDMODE == 0
```

```
    if (that == NULL) {
      GenBrushErr->_type = PBErrTypeNullPointer;
      sprintf(GenBrushErr->_msg, "'that' is null");
      PBErrCatch(GenBrushErr);
    }
#endif
  // Declare a variable to memorize the values of the projection matrix
  // order is: (0,0),(0,1),(0,2),(1,0),...
  float val[12] = {0.0};
  // Set the values according to the eye type
  switch (that->_type) {
    case GBEyeTypeOrtho:
      GBEyeOrthoGetProjVal((GBEyeOrtho*)that, val);
      break;
    case GBEyeTypeIsometric:
      GBEyeIsometricGetProjVal((GBEyeIsometric*)that, val);
      break;
    default:
      break;
  }
  // Set the values to the matrix
  VecShort2D index = VecShortCreateStatic2D();
  VecShort2D dim = MatGetDim(that->_proj);
  int iVal = 0;
  do {
    MatSet(that->_proj, &index, val[iVal]);
    ++iVal;
  } while (VecStep(&index, &dim));
}

// Return the projection through the GBEye 'that' of the Point 'point'
VecFloat* GBEyeGetProjectedPoint(GBEye* that, VecFloat* point) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (point == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'point' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Allocate memory for the output vector
  VecFloat* ret = VecClone(point);
  // Create a VecFloat3D to perform the product with the projection
  // matrix
  VecFloat3D v = VecFloatCreateStatic3D();
  for (int i = MIN(VecGetDim(point), 3); i--;)
    VecSet(&v, i, VecGet(point, i));
  // Perform the product
  VecFloat* w = MatGetProdVec(that->_proj, &v);
  // Translate to origin
  for (int i = 2; i--;)
    VecSet(w, i, VecGet(w, i) + VecGet(&(that->_orig), i));
  // Copy the values into the result
  for (int i = MIN(VecGetDim(point), 3); i--;)
    VecSet(ret, i, VecGet(w, i));
  // Free the memory used for the product
  VecFree(&w);
  // Return the result
```

```
    return ret;
}

// Return the projection through the GBEye 'that' of the SCurve 'curve'
SCurve* GBEyeGetProjectedCurve(GBEye* that, SCurve* curve) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (curve == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'curve' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Get a copy of the curve
  SCurve* ret = SCurveClone(curve);
  // Replace each curve's control point by its projection
  GSetIterForward iter =
    GSetIterForwardCreateStatic(SCurveCtrls(ret));
  do {
    VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
    VecFloat* v = GBEyeGetProjectedPoint(that, ctrl);
    VecCopy(ctrl ,v);
    VecFree(&v);
  } while (GSetIterStep(&iter));
  // Return the projected curve
  return ret;
}

// Return the projection through the GBEye 'that' of the Shapoid 'shap'
Shapoid* GBEyeGetProjectedShapoid(GBEye* that, Shapoid* shap) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (shap == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'shap' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Clone the shapoid
  Shapoid* ret = ShapoidClone(shap);
  // Project the origin of the shapoid
  VecFloat* proj = GBEyeGetProjectedPoint(that, ShapoidPos(ret));
  ShapoidSetPos(ret, proj);
  VecFree(&proj);
  // Project each axis
  for (int iAxis = ShapoidGetDim(ret); iAxis--;) {
    proj = GBEyeGetProjectedPoint(that, ShapoidAxis(ret, iAxis));
    VecCopy(ShapoidAxis(ret, iAxis), proj);
    VecFree(&proj);
  }
  // Return the new shapoid
  return ret;
}
```

```
// --------------- GBEyeOrtho ------------------------

// Return a new GBEyeOrtho with orientation 'view'
// scale is initialized to (1,1), trans to (0,0) and rot to 0
GBEyeOrtho* GBEyeOrthoCreate(GBEyeOrthoView view) {
  // Allocate memory
  GBEyeOrtho* that = PBErrMalloc(GenBrushErr, sizeof(GBEyeOrtho));
  // Set properties
  that->_view = view;
  that->_eye = GBEyeCreateStatic(GBEyeTypeOrtho);
  GBEyeUpdateProj((GBEye*)that);
  // Return the new GBEyeOrtho
  return that;
}

// Free the memory used by the GBEyeOrho
void GBEyeOrthoFree(GBEyeOrtho** that) {
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  GBEyeFreeStatic((GBEye*)*that);
  free(*that);
  *that = NULL;
}

// Get the value for the projection matrix of a GBEyeOrtho
void GBEyeOrthoGetProjVal(GBEyeOrtho* that, float* val) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (val == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'val' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare variable for optimization
  float cosTheta = cos(GBEyeGetRot(that));
  float sinTheta = sin(GBEyeGetRot(that));
  float scaleX = VecGet(GBEyeScale(that), 0);
  float scaleY = VecGet(GBEyeScale(that), 1);
  float scaleZ = VecGet(GBEyeScale(that), 2);
  // Set the values according to the orientation
  switch (that->_view) {
    case GBEyeOrthoViewFront:
      val[0] = scaleX * cosTheta;
      val[1] = scaleX * sinTheta;
      val[2] = 0.0;
      val[3] = scaleY * -sinTheta;
      val[4] = scaleY * cosTheta;
      val[5] = 0.0;
      val[6] = 0.0;
      val[7] = 0.0;
      val[8] = scaleZ;
      break;
    case GBEyeOrthoViewRear:
      val[0] = scaleX * -cosTheta;
      val[1] = scaleX * -sinTheta;
      val[2] = 0.0;
```

```
      val[3] = scaleY * -sinTheta;
      val[4] = scaleY * cosTheta;
      val[5] = 0.0;
      val[6] = 0.0;
      val[7] = 0.0;
      val[8] = -scaleZ;
      break;
    case GBEyeOrthoViewTop:
      val[0] = scaleX * cosTheta;
      val[1] = scaleX * sinTheta;
      val[2] = 0.0;
      val[3] = 0.0;
      val[4] = 0.0;
      val[5] = -scaleZ;
      val[6] = scaleY * -sinTheta;
      val[7] = scaleY * cosTheta;
      val[8] = 0.0;
      break;
    case GBEyeOrthoViewBottom:
      val[0] = scaleX * cosTheta;
      val[1] = scaleX * sinTheta;
      val[2] = 0.0;
      val[3] = 0.0;
      val[4] = 0.0;
      val[5] = scaleZ;
      val[6] = scaleY * sinTheta;
      val[7] = scaleY * -cosTheta;
      val[8] = 0.0;
      break;
    case GBEyeOrthoViewLeft:
      val[0] = 0.0;
      val[1] = 0.0;
      val[2] = scaleZ;
      val[3] = scaleY * -sinTheta;
      val[4] = scaleY * cosTheta;
      val[5] = 0.0;
      val[6] = scaleX * -cosTheta;
      val[7] = scaleX * -sinTheta;
      val[8] = 0.0;
      break;
    case GBEyeOrthoViewRight:
      val[0] = 0.0;
      val[1] = 0.0;
      val[2] = -scaleZ;
      val[3] = scaleY * -sinTheta;
      val[4] = scaleY * cosTheta;
      val[5] = 0.0;
      val[6] = scaleX * cosTheta;
      val[7] = scaleX * sinTheta;
      val[8] = 0.0;
      break;
    default:
      break;
  }
}

// Process the object of the GBObjPod 'pod' to update the viewed object
// through the GBEyeOrtho 'that'
void GBEyeOrthoProcess(GBEyeOrtho* that, GBObjPod* pod) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
```

```
      sprintf(GenBrushErr->_msg, "'that' is null");
      PBErrCatch(GenBrushErr);
  }
  if (pod == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pod' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  VecFloat* point = NULL;
  SCurve* curve = NULL;
  Shapoid* shap = NULL;
  switch (GBObjPodGetType(pod)) {
    case GBObjTypePoint:
      point = GBEyeGetProjectedPoint((GBEye*)that,
        GBObjPodGetObjAsPoint(pod));
      GBObjPodSetEyePoint(pod, point);
      break;
    case GBObjTypeSCurve:
      curve = GBEyeGetProjectedCurve((GBEye*)that,
        GBObjPodGetObjAsSCurve(pod));
      GBObjPodSetEyeSCurve(pod, curve);
      break;
    case GBObjTypeShapoid:
      shap = GBEyeGetProjectedShapoid((GBEye*)that,
        GBObjPodGetObjAsShapoid(pod));
      GBObjPodSetEyeShapoid(pod,shap);
      break;
    default:
      break;
  }
}


// --------------- GBEyeIsometric -------------------------

// Return a new GBEyeIsometric with orientation 'view'
// scale is initialized to (1,1), trans to (0,0) and rot to 0
// thetaY is initialized to pi/4 and thetaRight to pi/4
GBEyeIsometric* GBEyeIsometricCreate() {
  // Allocate memory
  GBEyeIsometric* that = PBErrMalloc(GenBrushErr,
    sizeof(GBEyeIsometric));
  // Set properties
  that->_thetaY = PBMATH_QUARTERPI;
  that->_thetaRight = PBMATH_QUARTERPI;
  that->_eye = GBEyeCreateStatic(GBEyeTypeIsometric);
  GBEyeUpdateProj((GBEye*)that);
  // Return the new GBEyeIsometric
  return that;
}


// Free the memory used by the GBEyeIsometric 'that'
void GBEyeIsometricFree(GBEyeIsometric** that) {
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  GBEyeFreeStatic((GBEye*)*that);
  free(*that);
  *that = NULL;
}


// Process the object of the GBObjPod 'pod' to update the viewed object
```

```c
// through the GBEyeIsometric 'that'
void GBEyeIsometricProcess(GBEyeIsometric* that, GBObjPod* pod) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pod == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pod' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  VecFloat* point = NULL;
  SCurve* curve = NULL;
  Shapoid* shap = NULL;
  switch (GBObjPodGetType(pod)) {
    case GBObjTypePoint:
      point = GBEyeGetProjectedPoint((GBEye*)that,
        GBObjPodGetObjAsPoint(pod));
      GBObjPodSetEyePoint(pod, point);
      break;
    case GBObjTypeSCurve:
      curve = GBEyeGetProjectedCurve((GBEye*)that,
        GBObjPodGetObjAsSCurve(pod));
      GBObjPodSetEyeSCurve(pod, curve);
      break;
    case GBObjTypeShapoid:
      shap = GBEyeGetProjectedShapoid((GBEye*)that,
        GBObjPodGetObjAsShapoid(pod));
      GBObjPodSetEyeShapoid(pod,shap);
      break;
    default:
      break;
  }
}

// Get the value for the projection matrix of a GBEyeIsometric
void GBEyeIsometricGetProjVal(GBEyeIsometric* that, float* val) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (val == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'val' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare variables for optimization
  float scaleX = VecGet(GBEyeScale(that), 0);
  float scaleY = VecGet(GBEyeScale(that), 1);
  float scaleZ = VecGet(GBEyeScale(that), 2);
  float cosTheta = cos(GBEyeGetRot(that));
  float sinTheta = sin(GBEyeGetRot(that));
  float cosThetaY = cos(GBEyeIsometricGetRotY(that));
  float sinThetaY = sin(GBEyeIsometricGetRotY(that));
  float cosThetaR = cos(GBEyeIsometricGetRotRight(that));
  float sinThetaR = sin(GBEyeIsometricGetRotRight(that));
```

```
  // Set the values
  val[0] = scaleX * cosTheta * cosThetaY -
    scaleY * sinTheta * sinThetaY * sinThetaR;
  val[1] = scaleX * sinTheta * cosThetaY +
    scaleY * cosTheta * sinThetaY * sinThetaR;
  val[2] = scaleZ * sinThetaY * cosThetaR;

  val[3] = -1.0 * scaleY * sinTheta * cosThetaR;
  val[4] = scaleY * cosTheta * cosThetaR;
  val[5] = -scaleZ * sinThetaR;

  val[6] = -1.0 * (scaleX * cosTheta * sinThetaY +
    scaleY * sinTheta * cosThetaY * sinThetaR);
  val[7] = -1.0 * scaleX * sinTheta * sinThetaY +
    scaleY * cosTheta * cosThetaY * sinThetaR;
  val[8] = scaleZ * cosThetaY * cosThetaR;
}

// --------------- GBHand ------------------------

// Create a new GBHand with type 'type'
GBHand GBHandCreateStatic(GBHandType type) {
  // Declare the new hand
  GBHand hand;
  // Set properties
  hand._type = type;
  // Return the new hand
  return hand;
}

// Free the memory used by the GBHand 'that'
void GBHandFreeStatic(GBHand* that) {
  // Nothing to do
  (void)that;
}

// --------------- GBHandDefault ------------------------

// Create a new GBHandDefault
GBHandDefault* GBHandDefaultCreate() {
  // Declare the new hand
  GBHandDefault* hand = PBErrMalloc(GenBrushErr, sizeof(GBHandDefault));
  // Set properties
  hand->_hand = GBHandCreateStatic(GBHandTypeDefault);
  // Return the new hand
  return hand;
}

// Free the memory used by the GBHandDefault 'that'
void GBHandDefaultFree(GBHandDefault** that) {
  // Check argument
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  GBHandFreeStatic((GBHand*)(*that));
  free(*that);
  *that = NULL;
}

// Process the viewed projection of the object in GBObjPod 'pod' into
// its handed projection through the GBHandDefault 'that'
void GBHandDefaultProcess(GBHandDefault* that, GBObjPod* pod) {
```

```c
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pod == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pod' is null");
    PBErrCatch(GenBrushErr);
  }
  if (GBObjPodGetEyeObj(pod) == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "Viewed object of 'pod' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  (void)that;
  // Empty all the sets
  GSetVecFloat* setPoints = GBObjPodGetHandObjAsPoints(pod);
  while (GSetNbElem(setPoints) > 0) {
    VecFloat* point = GSetPop(setPoints);
    VecFree(&point);
  }
  GSetShapoid* setShapoids = GBObjPodGetHandObjAsShapoids(pod);
  while (GSetNbElem(setShapoids) > 0) {
    Shapoid* shap = GSetPop(setShapoids);
    ShapoidFree(&shap);
  }
  GSetSCurve* setSCurves = GBObjPodGetHandObjAsSCurves(pod);
  while (GSetNbElem(setPoints) > 0) {
    SCurve* curve = GSetPop(setSCurves);
    SCurveFree(&curve);
  }
  // Simply copy the viewed object as it is according to its type
  switch (GBObjPodGetType(pod)) {
    case GBObjTypePoint:
      // Add a copy of the viewed point
      GSetAppend(setPoints, VecClone(GBObjPodGetEyeObjAsPoint(pod)));
      break;
    case GBObjTypeShapoid:
      // Add a copy of the viewed shapoid
      GSetAppend(setShapoids,
        ShapoidClone(GBObjPodGetEyeObjAsShapoid(pod)));
      break;
    case GBObjTypeSCurve:
      // Add a copy of the viewed scurve
      GSetAppend(setSCurves,
        SCurveClone(GBObjPodGetEyeObjAsSCurve(pod)));
      break;
    default:
      break;
  }
}

// --------------- GBTool --------------------------

// Create a static GBTool with GBToolType 'type'
GBTool GBToolCreateStatic(GBToolType type) {
  // Declare the GBTool
  GBTool that;
  // Set properties
```

```
  that._type = type;
  // Return the tool
  return that;
}

// Free the memory used by the GBTool 'that'
void GBToolFreeStatic(GBTool* that) {
  // Nothing to do
  (void)that;
}


// --------------- GBToolPlotter ------------------------

// Create a new GBToolPlotter
GBToolPlotter* GBToolPlotterCreate() {
  // Declare the new GBToolPlotter
  GBToolPlotter* that = PBErrMalloc(GenBrushErr, sizeof(GBToolPlotter));
  // Set properties
  that->_tool = GBToolCreateStatic(GBToolTypePlotter);
  // Return the new tool
  return that;
}

// Free the memory used by the GBToolPlotter 'that'
void GBToolPlotterFree(GBToolPlotter** that) {
  // Check argument
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free memory
  GBToolFreeStatic(&((*that)->_tool));
  free(*that);
  *that = NULL;
}

// Draw the object in the GBObjPod 'pod' with the GBToolPlotter 'that'
void GBToolPlotterDraw(GBToolPlotter* that, GBObjPod* pod) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pod == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pod' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Loop on the Points drawn by hand
  if (GSetNbElem(&(pod->_handPoints)) > 0) {
    GSetIterForward iter =
      GSetIterForwardCreateStatic(&(pod->_handPoints));
    do {
      // Get the point
      VecFloat* point = GSetIterGet(&iter);
      // Draw the point
      GBToolPlotterDrawPoint(that, point, pod);
    } while (GSetIterStep(&iter));
  }
  // Loop on the Shapoids drawn by hand
  if (GSetNbElem(&(pod->_handShapoids)) > 0) {
```

```
        GSetIterForward iter =
          GSetIterForwardCreateStatic(&(pod->_handShapoids));
        do {
          // Get the shapoid
          Shapoid* shap = GSetIterGet(&iter);
          // Draw the shapoid
          GBToolPlotterDrawShapoid(that, shap, pod);
        } while (GSetIterStep(&iter));
      }
    // Loop on the SCurves drawn by hand
    if (GSetNbElem(&(pod->_handSCurves)) > 0) {
      GSetIterForward iter =
        GSetIterForwardCreateStatic(&(pod->_handSCurves));
      do {
        // Get the curve
        SCurve* curve = GSetIterGet(&iter);
        // Draw the curve
        GBToolPlotterDrawSCurve(that, curve, pod);
      } while (GSetIterStep(&iter));
    }
}

// Draw the Point 'point' in the GBObjPod 'pod' with the
// GBToolPlotter 'that'
void GBToolPlotterDrawPoint(GBToolPlotter* that, VecFloat* point,
  GBObjPod* pod) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pod == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pod' is null");
    PBErrCatch(GenBrushErr);
  }
  if (point == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'point' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Get the position in layer
  VecShort2D posLayer = VecShortCreateStatic2D();
  for (int i = 2; i--;)
    VecSet(&posLayer, i, (short)round(VecGet(point, i)));
  // Get the color from the ink
  // As the point as no internal parameters create a null 3d vector
  // instead
  VecFloat3D null = VecFloatCreateStatic3D();
  GBPixel pixColor = GBInkGet(GBObjPodGetInk(pod), that, pod, &null,
    point, &posLayer);
  // Get the depth
  float depth = 0.0;
  if (VecGetDim(point) > 2)
    depth = VecGet(point, 2);
  // Add the pixel at the point position in the layer
  GBLayerAddPixelSafe(GBObjPodGetLayer(pod), &posLayer,
    &pixColor, depth);
}
```

```
// Draw the Shapoid 'shap' in the GBObjPod 'pod' with the
// GBToolPlotter 'that'
void GBToolPlotterDrawShapoid(GBToolPlotter* that, Shapoid* shap,
  GBObjPod* pod) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pod == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pod' is null");
    PBErrCatch(GenBrushErr);
  }
  if (shap == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'shap' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Get the bounding box of the shapoid in layer
  Facoid* bound = ShapoidGetBoundingBox(shap);
  // Declare two vectors for the VecShiftStep
  VecShort* from = VecShortCreate(ShapoidGetDim(bound));
  VecShort* to = VecShortCreate(ShapoidGetDim(bound));
  // Initialise the values of the vectors for the VecShiftStep
  for (int iAxis = ShapoidGetDim(bound); iAxis--;) {
    if (iAxis < 2) {
      VecSet(from, iAxis,
        (short)round(MAX(VecGet(ShapoidPos(bound), iAxis), 0.0)));
      short v = (short)round(VecGet(ShapoidPos(bound), iAxis) +
        VecGet(ShapoidAxis(bound, iAxis), iAxis)) + 1;
      VecSet(to, iAxis,
        MIN(v, VecGet(GBLayerDim(GBObjPodGetLayer(pod)),iAxis)));
    } else {
      VecSet(from, iAxis,
        (short)round(VecGet(ShapoidPos(bound), iAxis)));
      short v = (short)round(VecGet(ShapoidPos(bound), iAxis) +
        VecGet(ShapoidAxis(bound, iAxis), iAxis)) + 1;
      VecSet(to, iAxis, v);
    }
  }
  // Loop on pixels in layers inside the bounding box
  VecShort* posLayer = VecClone(from);
  VecFloat* posLayerFloat = VecFloatCreate(VecGetDim(posLayer));
  do {
    for (int iAxis = VecGetDim(posLayer); iAxis--;)
      VecSet(posLayerFloat, iAxis,
        (float)VecGet(posLayer, iAxis) + 0.5);
    // If this pixel is inside the Shapoid
    if (ShapoidIsPosInside(shap, posLayerFloat)) {
      // Get the current internal coordinates
      VecFloat* posIn = ShapoidImportCoord(shap, posLayerFloat);
      // Get the current external coordinates
      VecFloat* posExt =
        ShapoidExportCoord(GBObjPodGetObjAsShapoid(pod), posIn);
      // Get the ink at this position
      GBPixel pixColor = GBInkGet(GBObjPodGetInk(pod), that, pod,
        posIn, posExt, posLayer);
      // Get the depth
      float depth = 0.0;
```

74

```c
      if (VecGetDim(posLayerFloat) > 2)
        depth = VecGet(posLayerFloat, 2);
      // Add the pixel to the layer
      GBLayerAddPixelSafe(GBObjPodGetLayer(pod), (VecShort2D*)posLayer,
        &pixColor, depth);
      // Free memory
      VecFree(&posIn);
      VecFree(&posExt);
    }
  } while (VecShiftStep(posLayer, from, to));
  // Free memory
  VecFree(&from);
  VecFree(&to);
  VecFree(&posLayer);
  VecFree(&posLayerFloat);
  ShapoidFree(&bound);
}

// Draw the SCurve 'curve' in the GBObjPod 'pod' with the
// GBToolPlotter 'that'
void GBToolPlotterDrawSCurve(GBToolPlotter* that, SCurve* curve,
  GBObjPod* pod) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pod == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pod' is null");
    PBErrCatch(GenBrushErr);
  }
  if (curve == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'curve' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Det the approximate length of the curve
  float length = SCurveGetApproxLen(curve);
  // Calculate the delta step based on teh apporximate length
  float delta = 0.5 / length;
  // Create an iterator on the curve
  SCurveIter iter = SCurveIterCreateStatic(curve, delta);
  // Declare a vector to store the internal position as a vector
  VecFloat* posIn = VecFloatCreate(1);
  // Declare a vector to memorize the position in layer as short
  VecShort* posLayer = VecShortCreate(SCurveGetDim(curve));
  // Declare a vector to memorize the
  VecShort* prevPosLayer = VecShortCreate(SCurveGetDim(curve));
  // Loop on the curve internal position
  do {
    // Get the internal position
    float p = SCurveIterGetPos(&iter);
    // Get the position in layer
    VecFloat* posLayerFloat = SCurveIterGet(&iter);
    for (int iAxis = VecGetDim(posLayer); iAxis--;)
      VecSet(posLayer, iAxis,
        (short)round(VecGet(posLayerFloat, iAxis)));
    // Ensure the position is different (at pixel level) with the
    // previous one, except for the first step (p==0.0)
```

```
      if (ISEQUALF(p, 0.0) == true ||
        VecIsEqual(posLayer, prevPosLayer) == false) {
        // Get the internal position as a vector
        VecSet(posIn, 0, p);
        // Get the external position
        VecFloat* posExt = SCurveGet(GBObjPodGetObjAsSCurve(pod), p);
        // Get the ink at this position
        GBPixel pixColor = GBInkGet(GBObjPodGetInk(pod), that, pod,
          posIn, posExt, posLayer);
        // Get the depth
        float depth = 0.0;
        if (VecGetDim(posLayerFloat) > 2)
          depth = VecGet(posLayerFloat, 2);
        // Add the pixel to the layer
        GBLayerAddPixelSafe(GBObjPodGetLayer(pod), (VecShort2D*)posLayer,
          &pixColor, depth);
        // Free memory
        VecFree(&posExt);
        // Set the previous position
        VecCopy(prevPosLayer, posLayer);
      }
      // Free memory
      VecFree(&posLayerFloat);
    } while (SCurveIterStep(&iter));
    // Free memory
    VecFree(&posIn);
    VecFree(&posLayer);
    VecFree(&prevPosLayer);
}

// --------------- GBInk ------------------------

// Entry point for the GBTool<>Draw function to get the color of the
// appropriate GBInk according to the type of 'that'
// posInternal represents the position in the object internal space
// posExternal represents the position in the global coordinates system
// posLayer represents the position in the destination layer
GBPixel _GBInkGet(GBInk* that, GBTool* tool, GBObjPod* pod,
  VecFloat* posInternal, VecFloat* posExternal, VecShort* posLayer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Currently unused, voided and left for forward compatibility
  (void)tool; (void)pod;
  (void)posInternal; (void)posExternal; (void)posLayer;
  // Declare the result pixel
  GBPixel pix = GBColorTransparent;
  // Call the appropriate function according to the type of ink
  switch (GBInkGetType(that)) {
    case GBInkTypeSolid:
      pix = GBInkSolidGet((GBInkSolid*)that);
      break;
    default:
      break;
  }
  // Return the result pixel
  return pix;
}
```

```
// --------------- GBInk ------------------------

// Free the memory used by the GBInk 'that'
void GBInkFree(GBInk* that) {
  // Nothing to do
  (void)that;
}

// --------------- GBInkSolid ------------------------

// Create a new GBInkSolid with color 'col'
GBInkSolid* GBInkSolidCreate(GBPixel* col) {
#if BUILDMODE == 0
  if (col == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'col' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare the new GBInkSolid
  GBInkSolid* ink = PBErrMalloc(GenBrushErr, sizeof(GBInkSolid));
  // Set properties
  ink->_ink._type = GBInkTypeSolid;
  ink->_color = *col;
  // Return the new ink
  return ink;
}

// Free the memory used by the GBInkSolid 'that'
void GBInkSolidFree(GBInkSolid** that) {
  // Check argument
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free memory
  GBInkFree(&((*that)->_ink));
  free(*that);
  *that = NULL;
}

// --------------- GBObjPod ------------------------

// Create a new GBObjPod for a Point at position 'pos'
// drawn with 'eye', 'hand', 'tool' and 'ink' in layer 'layer'
// 'pos' must be a vector of 2 or more dimensions
// If 'eye', 'hand, 'tool', 'ink' or 'layer' is null return null
GBObjPod* _GBObjPodCreatePoint(VecFloat* pos, GBEye* eye, GBHand* hand,
  GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (VecGetDim(pos) < 2) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg,
    "'pos' 's dimension is invalid (%d>=2)",
    VecGetDim(pos));
    PBErrCatch(GenBrushErr);
  }
```

```
#endif
  // Check that the parameters are not null
  if (eye == NULL || hand == NULL || tool == NULL ||
    ink == NULL || layer == NULL)
    return NULL;
  // Declare the new GBObjPod
  GBObjPod* pod = PBErrMalloc(GenBrushErr, sizeof(GBObjPod));
  // Set properties
  pod->_type = GBObjTypePoint;
  pod->_srcPoint = pos;
  pod->_eyePoint = NULL;
  pod->_handPoints = GSetVecFloatCreateStatic();
  pod->_handShapoids = GSetShapoidCreateStatic();
  pod->_handSCurves = GSetSCurveCreateStatic();
  pod->_eye = eye;
  pod->_hand = hand;
  pod->_tool = tool;
  pod->_ink = ink;
  pod->_layer = layer;
  // Process the Point through eye and hand
  GBEyeProcess(eye, pod);
  GBHandProcess(hand, pod);
  // Return the new pod
  return pod;
}

// Create a new GBObjPod for the Shapoid 'shap'
// drawn with 'eye', 'hand', 'tool' and 'ink' in layer 'layer'
// 'shap' 's dimension must be 2 or more
// If 'eye', 'hand, 'tool', 'ink' or 'layer' is null return null
GBObjPod* _GBObjPodCreateShapoid(Shapoid* shap, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (shap == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'shap' is null");
    PBErrCatch(GenBrushErr);
  }
  if (ShapoidGetDim(shap) < 2) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg,
    "'shap' 's dimension is invalid (%d>=2)",
    ShapoidGetDim(shap));
    PBErrCatch(GenBrushErr);
  }
#endif
  // Check that the parameters are not null
  if (eye == NULL || hand == NULL || tool == NULL ||
    ink == NULL || layer == NULL)
    return NULL;
  // Declare the new GBObjPod
  GBObjPod* pod = PBErrMalloc(GenBrushErr, sizeof(GBObjPod));
  // Set properties
  pod->_type = GBObjTypeShapoid;
  pod->_srcShapoid = shap;
  pod->_eyeShapoid = NULL;
  pod->_handPoints = GSetVecFloatCreateStatic();
  pod->_handShapoids = GSetShapoidCreateStatic();
  pod->_handSCurves = GSetSCurveCreateStatic();
  pod->_eye = eye;
  pod->_hand = hand;
  pod->_tool = tool;
```

78

```c
  pod->_ink = ink;
  pod->_layer = layer;
  // Process the Shapoid through eye and hand
  GBEyeProcess(eye, pod);
  GBHandProcess(hand, pod);
 // Return the new pod
  return pod;
}


// Create a new GBObjPod for the SCurve 'curve'
// drawn with 'eye', 'hand', 'tool' and 'ink' in layer 'layer'
// 'curve' 's dimension must be 2 or more
// If 'eye', 'hand, 'tool', 'ink' or 'layer' is null return null
GBObjPod* _GBObjPodCreateSCurve(SCurve* curve, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (curve == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'curve' is null");
    PBErrCatch(GenBrushErr);
  }
  if (SCurveGetDim(curve) < 2) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg,
    "'curve' 's dimension is invalid (%d>=2)",
    SCurveGetDim(curve));
    PBErrCatch(GenBrushErr);
  }
#endif
  // Check that the parameters are not null
  if (eye == NULL || hand == NULL || tool == NULL ||
    ink == NULL || layer == NULL)
    return NULL;
  // Declare the new GBObjPod
  GBObjPod* pod = PBErrMalloc(GenBrushErr, sizeof(GBObjPod));
  // Set properties
  pod->_type = GBObjTypeSCurve;
  pod->_srcSCurve = curve;
  pod->_eyeSCurve = NULL;
  pod->_handPoints = GSetVecFloatCreateStatic();
  pod->_handShapoids = GSetShapoidCreateStatic();
  pod->_handSCurves = GSetSCurveCreateStatic();
  pod->_eye = eye;
  pod->_hand = hand;
  pod->_tool = tool;
  pod->_ink = ink;
  pod->_layer = layer;
  // Process the SCurve through eye and hand
  GBEyeProcess(eye, pod);
  GBHandProcess(hand, pod);
  // Return the new pod
  return pod;
}


// Free the memory used by the GBObjPod 'that'
void GBObjPodFree(GBObjPod** that) {
  // Check argument
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free memory
  switch (GBObjPodGetType(*that)) {
```

```
      case GBObjTypePoint:
        VecFree(&((*that)->_eyePoint));
        break;
      case GBObjTypeShapoid:
        ShapoidFree(&((*that)->_eyeShapoid));
        break;
      case GBObjTypeSCurve:
        SCurveFree(&((*that)->_eyeSCurve));
        break;
      default:
        break;
    }
    while (GSetNbElem(&((*that)->_handPoints)) > 0) {
      VecFloat* point = GSetPop(&((*that)->_handPoints));
      VecFree(&point);
    }
    while (GSetNbElem(&((*that)->_handShapoids)) > 0) {
      Shapoid* shap = GSetPop(&((*that)->_handShapoids));
      ShapoidFree(&shap);
    }
    while (GSetNbElem(&((*that)->_handSCurves)) > 0) {
      SCurve* curve = GSetPop(&((*that)->_handSCurves));
      SCurveFree(&curve);
    }
    free(*that);
    *that = NULL;
}

// --------------- GenBrush ------------------------

// Create a new GenBrush with a GBSurface of type GBSurfaceTypeImage
// and dimensions 'dim'
GenBrush* GBCreateImage(VecShort2D* dim) {
#if BUILDMODE == 0
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
  }
  if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'dim' is invalid (%d>0,%d>0)",
      VecGet(dim, 0), VecGet(dim, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare the new GenBrush
  GenBrush* ret = PBErrMalloc(GenBrushErr, sizeof(GenBrush));
  // Set properties
  ret->_surf = (GBSurface*)GBSurfaceImageCreate(dim);
  ret->_pods = GSetCreateStatic();
  ret->_postProcs = GSetCreateStatic();
  // Return the new GenBrush
  return ret;
}

// Create a new GenBrush with one layer containing the content
// of the image located at 'fileName' and dimensions equal to the
// dimensions of the image
// If the image couldn't be loaded return NULL
GenBrush* GBCreateFromFile(char* fileName) {
  // Declare the new GenBrush
```

```
    GenBrush* ret = NULL;
    // Load the image in a surface
    GBSurfaceImage *img = GBSurfaceImageCreateFromFile(fileName);
    if (img != NULL) {
      // Declare the new GenBrush
      ret = PBErrMalloc(GenBrushErr, sizeof(GenBrush));
      // Set properties
      ret->_surf = (GBSurface*)img;
      ret->_pods = GSetCreateStatic();
      ret->_postProcs = GSetCreateStatic();
    }
    // Return the new GenBrush
    return ret;

}

// Free memory used by the GenBrush 'that'
void GBFree(GenBrush** that) {
  // If the pointer is null
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free memory
  switch (GBGetType(*that)) {
    case GBSurfaceTypeDefault:
      GBSurfaceFree(&((*that)->_surf));
      break;
    case GBSurfaceTypeImage:
      GBSurfaceImageFree((GBSurfaceImage**)&((*that)->_surf));
      break;
    default:
      break;
  }
  GBRemoveAllPod(*that);
  GBRemoveAllPostProcess(*that);
  free(*that);
  *that = NULL;
}

// Update the GBSurface of the GenBrush 'that'
void GBUpdate(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Refresh the layers that have been modified
  // If there is layers
  if (GBGetNbLayer(that) > 0) {
    // Declare an iterator on the layers
    GSetIterForward iter =
      GSetIterForwardCreateStatic(GBSurfaceLayers(GBSurf(that)));
    // Loop on the layers
    do {
      // Get the current layer
      GBLayer* layer = GSetIterGet(&iter);
      // If this layer has been modified
      if (GBLayerIsModified(layer))
        // Refresh the layer content
        GBUpdateLayer(that, layer);
```

81

```
    } while (GSetIterStep(&iter));
  }
  // Request the update of the surface
  GBSurfaceUpdate(GBSurf(that));
  // Apply the post processing
  if (GSetNbElem(GBPostProcs(that)) > 0) {
    GSetIterForward iter =
      GSetIterForwardCreateStatic(GBPostProcs(that));
    do {
      GBPostProcessing* post = GSetIterGet(&iter);
      GBSurfacePostProcess(GBSurf(that), post);
    } while (GSetIterStep(&iter));
  }
}

// Update the content of the layer 'layer' of the GenBrush 'that'
// Flush the content of the layer and redraw all the pod attached to it
void GBUpdateLayer(GenBrush* that, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (layer == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'layer' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Flush the content of the layer
  GBLayerFlush(layer);
  // If there is no pods
  if (GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Loop on pods
  do {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod is attahced ot the requested layer
    if (pod->_layer == layer)
      // Redraw the pod
      GBToolDraw(GBObjPodGetTool(pod), pod);
  } while (GSetIterStep(&iter));
}

// Render the GBSurface (save on disk, display on screen, ...) of
// the GenBrush 'that'
bool GBRender(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Declare a variable to memorize the success of rendering
  bool status = false;
  switch (GBSurfaceGetType(that->_surf)) {
```

```
      case GBSurfaceTypeImage:
        status = GBSurfaceImageSave((GBSurfaceImage*)(that->_surf));
        break;
#if BUILDWITHGRAPHICLIB == 1
      case GBSurfaceTypeWidget:
        GBSurfaceWidgetRender((GBSurfaceWidget*)(that->_surf));
        status = true;
        break;
      case GBSurfaceTypeApp:
        status = GBSurfaceAppRender((GBSurfaceApp*)(that->_surf));
        break;
#endif
      default:
        break;
  }
  // Return the status of rendering
  return status;
}


// Remove from the list of pods of the GenBrush 'that' those who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBRemovePod(that, elemA, NULL, NULL, NULL, NULL, NULL) removes all
// the pods related to the object 'elemA'
// GBRemovePod(that, elemA, NULL, handA, NULL, NULL, NULL) removes
// all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it removes all the pods
void _GBRemovePod(GenBrush* that, void* obj, GBEye* eye, GBHand* hand,
  GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If there is no pods
  if (GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Declare a variable for the end condition of the loop
  bool flag = true;
  // Loop on pods
  while (flag) {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod matches the filters
    if ((obj == NULL || (void*)(pod->_srcPoint) == obj) &&
      (eye == NULL || pod->_eye == eye) &&
      (hand == NULL || pod->_hand == hand) &&
      (tool == NULL || pod->_tool == tool) &&
      (ink == NULL || pod->_ink == ink) &&
      (layer == NULL || pod->_layer == layer)) {
      // Remove the pod and move to the next one
      GBObjPodFree(&pod);
      flag = GSetIterRemoveElem(&iter);
    // Else the pod doesn't match the filters
    } else {
      // Move to the next pod
      flag = GSetIterStep(&iter);
```

```
    }
  }
}

// Set the eye to 'toEye' in the pods of the GenBrush 'that' who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBSetPodEye(that, toEye, elemA, NULL, NULL, NULL, NULL, NULL)
// affects all the pods related to the object 'elemA'
// GBSetPodEye(that, toEye, elemA, NULL, handA, NULL, NULL, NULL)
// affects all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it affects all the pods
// If 'toEye' is null, do nothing
void _GBSetPodEye(GenBrush* that, GBEye* toEye, void* obj, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If 'toEye' is null or there is no pods
  if (toEye == NULL || GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Loop on pods
  do {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod matches the filters
    if ((obj == NULL || (void*)(pod->_srcPoint) == obj) &&
      (eye == NULL || pod->_eye == eye) &&
      (hand == NULL || pod->_hand == hand) &&
      (tool == NULL || pod->_tool == tool) &&
      (ink == NULL || pod->_ink == ink) &&
      (layer == NULL || pod->_layer == layer))
      // Set the eye of the pod
      pod->_eye = toEye;
  } while (GSetIterStep(&iter));
}

// Set the hand to 'toHand' in the pods of the GenBrush 'that' who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBSetPodEye(that, toHand, elemA, NULL, NULL, NULL, NULL, NULL)
// affects all the pods related to the object 'elemA'
// GBSetPodEye(that, toHand, elemA, NULL, handA, NULL, NULL, NULL)
// affects all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it affects all the pods
// If 'toHand' is null, do nothing
void _GBSetPodHand(GenBrush* that, GBHand* toHand, void* obj,
  GBEye* eye, GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
```

```
  // If 'toHand' is null or there is no pods
  if (toHand == NULL || GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Loop on pods
  do {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod matches the filters
    if ((obj == NULL || (void*)(pod->_srcPoint) == obj) &&
      (eye == NULL || pod->_eye == eye) &&
      (hand == NULL || pod->_hand == hand) &&
      (tool == NULL || pod->_tool == tool) &&
      (ink == NULL || pod->_ink == ink) &&
      (layer == NULL || pod->_layer == layer))
      // Set the hand of the pod
      pod->_hand = toHand;
  } while (GSetIterStep(&iter));
}

// Set the tool to 'toTool' in the pods of the GenBrush 'that' who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBSetPodTool(that, toTool, elemA, NULL, NULL, NULL, NULL) affects
// all the pods related to the object 'elemA'
// GBSetPodTool(that, toTool, elemA, NULL, handA, NULL, NULL) affects
// all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it affects all the pods
// If 'toTool' is null, do nothing
void _GBSetPodTool(GenBrush* that, GBTool* toTool, void* obj,
  GBEye* eye, GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If 'toTool' is null or there is no pods
  if (toTool == NULL || GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Loop on pods
  do {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod matches the filters
    if ((obj == NULL || (void*)(pod->_srcPoint) == obj) &&
      (eye == NULL || pod->_eye == eye) &&
      (hand == NULL || pod->_hand == hand) &&
      (tool == NULL || pod->_tool == tool) &&
      (ink == NULL || pod->_ink == ink) &&
      (layer == NULL || pod->_layer == layer))
      // Set the tool of the pod
      pod->_tool = toTool;
  } while (GSetIterStep(&iter));
}
```

```
// Set the ink to 'toInk' in the pods of the GenBrush 'that' who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBSetPodTool(that, toTool, elemA, NULL, NULL, NULL, NULL) affects
// all the pods related to the object 'elemA'
// GBSetPodTool(that, toTool, elemA, NULL, handA, NULL, NULL) affects
// all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it affects all the pods
// If 'toInk' is null, do nothing
void _GBSetPodInk(GenBrush* that, GBInk* toInk, void* obj,
  GBEye* eye, GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If 'toInk' is null or there is no pods
  if (toInk == NULL || GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Loop on pods
  do {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod matches the filters
    if ((obj == NULL || (void*)(pod->_srcPoint) == obj) &&
      (eye == NULL || pod->_eye == eye) &&
      (hand == NULL || pod->_hand == hand) &&
      (tool == NULL || pod->_tool == tool) &&
      (ink == NULL || pod->_ink == ink) &&
      (layer == NULL || pod->_layer == layer))
      // Set the tool of the pod
      pod->_ink = toInk;
  } while (GSetIterStep(&iter));
}

// Set the layer to 'toLayer' in the pods of the GenBrush 'that' who
// match the 'obj', 'eye', 'hand', 'tool', 'ink' and 'layer'
// Null arguments are ignored. For example:
// GBSetPodLayer(that, toLayer, elemA, NULL, NULL, NULL, NULL, NULL)
// affects all the pods related to the object 'elemA'
// GBSetPodLayer(that, toLayer, elemA, NULL, handA, NULL, NULL, NULL)
// affects all the pods related to both the object 'elemA' AND 'handA'
// If all the filters are null it affects all the pods
// If 'toLayer' is null, do nothing
void _GBSetPodLayer(GenBrush* that, GBLayer* toLayer, void* obj,
  GBEye* eye, GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If 'toLayer' is null or there is no pods
  if (toLayer == NULL || GBGetNbPod(that) == 0)
    // Nothing to do
    return;
```

```
    // Declare an iterator on the set of pods
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
    // Loop on pods
    do {
      // Get the current pod
      GBObjPod* pod = GSetIterGet(&iter);
      // If the pod matches the filters
      if ((obj == NULL || (void*)(pod->_srcPoint) == obj) &&
        (eye == NULL || pod->_eye == eye) &&
        (hand == NULL || pod->_hand == hand) &&
        (tool == NULL || pod->_tool == tool) &&
        (ink == NULL || pod->_ink == ink) &&
        (layer == NULL || pod->_layer == layer))
        // Set the layer of the pod
        pod->_layer = toLayer;
    } while (GSetIterStep(&iter));
}

// Set to true the modified flag of the layers of pods attached to the
// object 'obj' in the list of pods of the GenBrush 'that'
void _GBNotifyChangeFromObj(GenBrush* that, void* obj) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (obj == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'obj' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If there is no pods
  if (GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Loop on pods
  do {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod matches the filter
    if (GBObjPodGetObj(pod) == obj) {
      // Reprocess the object
      GBEyeProcess(GBObjPodGetEye(pod), pod);
      GBHandProcess(GBObjPodGetHand(pod), pod);
      // Set the layer flag
      GBLayerSetModified(pod->_layer, true);
    }
  } while (GSetIterStep(&iter));
}

// Set to true the modified flag of the layers of pods attached to the
// GBEye 'eye' in the list of pods of the GenBrush 'that'
void _GBNotifyChangeFromEye(GenBrush* that, GBEye* eye) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
```

```
  }
  if (eye == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'eye' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If there is no pods
  if (GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Loop on pods
  do {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod matches the filter
    if (GBObjPodGetEye(pod) == eye) {
      // Reprocess the object
      GBEyeProcess(GBObjPodGetEye(pod), pod);
      GBHandProcess(GBObjPodGetHand(pod), pod);
      // Set the layer flag
      GBLayerSetModified(pod->_layer, true);
    }
  } while (GSetIterStep(&iter));
}

// Set to true the modified flag of the layers of pods attached to the
// GBInk 'ink' in the list of pods of the GenBrush 'that'
void _GBNotifyChangeFromInk(GenBrush* that, GBInk* ink) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (ink == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'ink' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If there is no pods
  if (GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Loop on pods
  do {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod matches the filter
    if (pod->_ink == ink)
      // Set the layer flag
      GBLayerSetModified(pod->_layer, true);
  } while (GSetIterStep(&iter));
}

// Set to true the modified flag of the layers of pods attached to the
// GBHand 'hand' in the list of pods of the GenBrush 'that'
```

```
void _GBNotifyChangeFromHand(GenBrush* that, GBHand* hand) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (hand == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'hand' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If there is no pods
  if (GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Loop on pods
  do {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod matches the filter
    if (GBObjPodGetHand(pod) == hand) {
      GBHandProcess(GBObjPodGetHand(pod), pod);
      // Set the layer flag
      GBLayerSetModified(pod->_layer, true);
    }
  } while (GSetIterStep(&iter));
}

// Set to true the modified flag of the layers of pods attached to the
// GBTool 'tool' in the list of pods of the GenBrush 'that'
void _GBNotifyChangeFromTool(GenBrush* that, GBTool* tool) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (tool == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'tool' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If there is no pods
  if (GBGetNbPod(that) == 0)
    // Nothing to do
    return;
  // Declare an iterator on the set of pods
  GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_pods));
  // Loop on pods
  do {
    // Get the current pod
    GBObjPod* pod = GSetIterGet(&iter);
    // If the pod matches the filter
    if (pod->_tool == tool)
      // Set the layer flag
      GBLayerSetModified(pod->_layer, true);
  } while (GSetIterStep(&iter));
```

```
}

// =============== GTK Functions ==================
#if BUILDWITHGRAPHICLIB == 1
  #include "genbrush-GTK.c"
#endif
```

3.2   genbrush-inline.c

```
// =========== GENBRUSH-INLINE.C ===============

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

// --------------- GBLayer -------------------------

// Get the area of the layer (width * height)
#if BUILDMODE != 0
inline
#endif
int GBLayerArea(GBLayer* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return (int)VecGet(&(that->_dim), 0) *
    (int)VecGet(&(that->_dim), 1);
}

// Get the position of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D* GBLayerPos(GBLayer* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return &(that->_pos);
}

// Get a copy of the position of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D GBLayerGetPos(GBLayer* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_pos;
}
```

```c
// Set the position of the GBLayer 'that' to 'pos'
// If the flag _modified==false _prevPos is first set to _pos
// and _modified is set to true
#if BUILDMODE != 0
inline
#endif
void GBLayerSetPos(GBLayer* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (!(that->_modified)) {
    that->_modified = true;
    that->_prevPos = that->_pos;
  }
  that->_pos = *pos;
}

// Get the previous position of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D* GBLayerPrevPos(GBLayer* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return &(that->_prevPos);
}

// Get a copy of the previous position of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D GBLayerGetPrevPos(GBLayer* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_prevPos;
}

// Get the dimensions of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D* GBLayerDim(GBLayer* that) {
```

```
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return &(that->_dim);
}

// Get a copy of the dimensions of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D GBLayerGetDim(GBLayer* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_dim;
}

// Get a copy of the blend mode of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
GBLayerBlendMode GBLayerGetBlendMode(GBLayer* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_blendMode;
}

// Set the blend mode of the GBLayer 'that' to 'blend'
// Set the flag _modified to true
#if BUILDMODE != 0
inline
#endif
void GBLayerSetBlendMode(GBLayer* that, GBLayerBlendMode blend) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  that->_modified = true;
  that->_blendMode = blend;
}

// Get a copy of the modified flag of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
bool GBLayerIsModified(GBLayer* that) {
```

```
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_modified;
}

// Set the modified flag of the GBLayer 'that' to 'flag'
#if BUILDMODE != 0
inline
#endif
void GBLayerSetModified(GBLayer* that, bool flag) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  that->_modified = flag;
}

// Get a copy of the stack position of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
GBLayerStackPosition GBLayerGetStackPos(GBLayer* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_stackPos;
}

// Set the stack position of the GBLayer 'that' to 'pos'
// Set the flag _modified to true
#if BUILDMODE != 0
inline
#endif
void GBLayerSetStackPos(GBLayer* that, GBLayerStackPosition pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  that->_modified = true;
  that->_stackPos = pos;
}

// Get the stacked pixels of the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
GSet* GBLayerPixels(GBLayer* that) {
```

```
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_pix;
}

// Get the stacked pixels of the GBLayer 'that' at position 'pos'
// 'pos' must be inside the layer
#if BUILDMODE != 0
inline
#endif
GSet* GBLayerPixel(GBLayer* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (!GBLayerIsPosInside(that, pos)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is out of the layer (%d,%d)",
      VecGet(pos, 0), VecGet(pos, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  int iPix = GBPosIndex(pos, GBLayerDim(that));
  return that->_pix + iPix;
}

// Get the stacked pixels of the GBLayer 'that' at position 'pos'
// If 'pos' is out of the layer return NULL
#if BUILDMODE != 0
inline
#endif
GSet* GBLayerPixelSafe(GBLayer* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (!GBLayerIsPosInside(that, pos))
    return NULL;
  else
    return GBLayerPixel(that, pos);
}
```

```
// Add the pixel 'pix' with depth 'depth' on top of the stack at
// position 'pos' of GBLayer 'that'
// Set the flag _modified to true
// 'pos' must be inside the layer
// If the pixel is completely transparent (_rgba[GBPixelAlpha]==0)
// do nothing
#if BUILDMODE != 0
inline
#endif
void GBLayerAddPixel(GBLayer* that, VecShort2D* pos, GBPixel* pix,
  float depth) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pix == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pix' is null");
    PBErrCatch(GenBrushErr);
  }
  if (!GBLayerIsPosInside(that, pos)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is out of the layer (%d,%d)",
      VecGet(pos, 0), VecGet(pos, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  // If the pixel is transparent
  if (pix->_rgba[GBPixelAlpha] == 0)
    // Do nothing
    return;
  int iPix = GBPosIndex(pos, GBLayerDim(that));
  GBStackedPixel* clone = PBErrMalloc(GenBrushErr,
    sizeof(GBStackedPixel));
  clone->_val = *pix;
  clone->_depth = depth;
  clone->_blendMode = GBLayerGetBlendMode(that);
  GSetAppend(that->_pix + iPix, clone);
}

// Add the pixel 'pix' with depth 'depth' on top of the stack at
// position 'pos' of GBLayer 'that'
// Set the flag _modified to true
// If 'pos' is out of the layer do nothing
// If the pixel is completely transparent (_rgba[GBPixelAlpha]==0)
// do nothing
#if BUILDMODE != 0
inline
#endif
void GBLayerAddPixelSafe(GBLayer* that, VecShort2D* pos, GBPixel* pix,
  float depth) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
```

```
      PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pix == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pix' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (GBLayerIsPosInside(that, pos))
    GBLayerAddPixel(that, pos, pix, depth);
}

// Return true if the position 'pos' is inside the layer 'that'
// boundary, false else
#if BUILDMODE != 0
inline
#endif
bool GBLayerIsPosInside(GBLayer* that, VecShort2D *pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (VecGet(pos, 0) >= 0 &&
    VecGet(pos, 1) >= 0 &&
    VecGet(pos, 0) < VecGet(GBLayerDim(that), 0) &&
    VecGet(pos, 1) < VecGet(GBLayerDim(that), 1))
    return true;
  else
    return false;
}

// Delete all the stacked pixels in the GBLayer 'that'
#if BUILDMODE != 0
inline
#endif
void GBLayerFlush(GBLayer* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  for (int iPix = GBLayerArea(that); iPix--;) {
    GSet *stack = that->_pix + iPix;
    while (GSetNbElem(stack) > 0) {
      GBStackedPixel *pix = GSetPop(stack);
      free(pix);
    }
```

```
  }
}

// --------------- GBPostProcessing ------------------------

// Return the type of the GBPostProcessing 'that'
#if BUILDMODE != 0
inline
#endif
GBPPType GBPostProcessingGetType(GBPostProcessing* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_type;
}

// --------------- GBSurface ------------------------

// Return true if the GBSurface 'that' has same dimension and same
// values for _finalPix as GBSurface 'surf'
// Else, return false
#if BUILDMODE != 0
inline
#endif
bool GBSurfaceIsSameAs(GBSurface* that, GBSurface* surf) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (surf == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'surf' is null");
    PBErrCatch(GenBrushErr);
  }
#endif

  // Display different pixels, for debugging purpose
  /*VecShort2D p = VecShortCreateStatic2D();
  do {
    GBPixel* pA = GBSurfaceFinalPixel(that, &p);
    GBPixel* pB = GBSurfaceFinalPixel(surf, &p);
    if (pA->_rgba[0] != pB->_rgba[0] ||
      pA->_rgba[1] != pB->_rgba[1] ||
      pA->_rgba[2] != pB->_rgba[2] ||
      pA->_rgba[3] != pB->_rgba[3]) {
      VecPrint(&p, stdout);
      printf(" %d,%d,%d,%d %d,%d,%d,%d\n",
        pA->_rgba[0], pA->_rgba[1], pA->_rgba[2], pA->_rgba[3],
        pB->_rgba[0], pB->_rgba[1], pB->_rgba[2], pB->_rgba[3]);
    }
  } while (VecStep(&p, GBSurfaceDim(that)));*/

  if (VecIsEqual(GBSurfaceDim(that), GBSurfaceDim(surf)) &&
    memcmp(GBSurfaceFinalPixels(that), GBSurfaceFinalPixels(surf),
    sizeof(GBPixel) * VecGet(GBSurfaceDim(that), 0) *
    VecGet(GBSurfaceDim(that), 1)) == 0)
```

97

```
    return true;
  else
    return false;
}

// Get the type of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
GBSurfaceType GBSurfaceGetType(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_type;
}

// Get a copy of the dimensions of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D GBSurfaceGetDim(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_dim;
}

// Get the dimensions of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D* GBSurfaceDim(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return &(that->_dim);
}

// Get the final pixels of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel* GBSurfaceFinalPixels(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
```

```
    return that->_finalPix;
}

// Get the final pixel at position 'pos' of the GBSurface 'that'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
GBPixel* GBSurfaceFinalPixel(GBSurface* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (!GBSurfaceIsPosInside(that, pos)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is out of the surface (%d,%d)",
      VecGet(pos, 0),VecGet(pos, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  int iPix = GBPosIndex(pos, GBSurfaceDim(that));
  return that->_finalPix + iPix;
}

// Get a copy of the final pixel at position 'pos' of the GBSurface
// 'that'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
GBPixel GBSurfaceGetFinalPixel(GBSurface* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (!GBSurfaceIsPosInside(that, pos)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is out of the surface (%d,%d)",
      VecGet(pos, 0),VecGet(pos, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  int iPix = GBPosIndex(pos, GBSurfaceDim(that));
  return that->_finalPix[iPix];
}

// Set the final pixel at position 'pos' of the GBSurface 'that' to
// the pixel 'pix'
```

99

```
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
void GBSurfaceSetFinalPixel(GBSurface* that, VecShort2D* pos,
  GBPixel* pix) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (!GBSurfaceIsPosInside(that, pos)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is out of the surface (%d,%d)",
      VecGet(pos, 0),VecGet(pos, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  int iPix = GBPosIndex(pos, GBSurfaceDim(that));
  that->_finalPix[iPix] = *pix;
}

// Get the final pixel at position 'pos' of the GBSurface 'that'
// If 'pos' is out of the surface return NULL
#if BUILDMODE != 0
inline
#endif
GBPixel* GBSurfaceFinalPixelSafe(GBSurface* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (GBSurfaceIsPosInside(that, pos))
    return GBSurfaceFinalPixel(that, pos);
  else
    return NULL;
}

// Get a copy of the final pixel at position 'pos' of the GBSurface
// 'that'
// If 'pos' is out of the surface return a transparent pixel
#if BUILDMODE != 0
inline
#endif
GBPixel GBSurfaceGetFinalPixelSafe(GBSurface* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
```

```c
      sprintf(GenBrushErr->_msg, "'that' is null");
      PBErrCatch(GenBrushErr);
    }
    if (pos == NULL) {
      GenBrushErr->_type = PBErrTypeNullPointer;
      sprintf(GenBrushErr->_msg, "'pos' is null");
      PBErrCatch(GenBrushErr);
    }
#endif
    if (GBSurfaceIsPosInside(that, pos))
      return GBSurfaceGetFinalPixel(that, pos);
    else
      return GBColorTransparent;
}

// Set the final pixel at position 'pos' of the GBSurface 'that' to
// the pixel 'pix'
// If 'pos' is out of the surface do nothing
#if BUILDMODE != 0
inline
#endif
void GBSurfaceSetFinalPixelSafe(GBSurface* that, VecShort2D* pos,
  GBPixel* pix) {
#if BUILDMODE == 0
    if (that == NULL) {
      GenBrushErr->_type = PBErrTypeNullPointer;
      sprintf(GenBrushErr->_msg, "'that' is null");
      PBErrCatch(GenBrushErr);
    }
    if (pos == NULL) {
      GenBrushErr->_type = PBErrTypeNullPointer;
      sprintf(GenBrushErr->_msg, "'pos' is null");
      PBErrCatch(GenBrushErr);
    }
    if (pix == NULL) {
      GenBrushErr->_type = PBErrTypeNullPointer;
      sprintf(GenBrushErr->_msg, "'pix' is null");
      PBErrCatch(GenBrushErr);
    }
#endif
    if (GBSurfaceIsPosInside(that, pos))
      GBSurfaceSetFinalPixel(that, pos, pix);
}

// Get the area of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
int GBSurfaceArea(GBSurface* that) {
#if BUILDMODE == 0
    if (that == NULL) {
      GenBrushErr->_type = PBErrTypeNullPointer;
      sprintf(GenBrushErr->_msg, "'that' is null");
      PBErrCatch(GenBrushErr);
    }
#endif
    return VecGet(&(that->_dim), 0) * VecGet(&(that->_dim), 1);
}

// Get the background color of the GBSurface 'that'
#if BUILDMODE != 0
inline
```

```
#endif
GBPixel* GBSurfaceBgColor(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return &(that->_bgColor);
}

// Get a copy of the background color of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel GBSurfaceGetBgColor(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_bgColor;
}

// Set the background color of the GBSurface 'that' to 'col'
#if BUILDMODE != 0
inline
#endif
void GBSurfaceSetBgColor(GBSurface* that, GBPixel* col) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (col == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'col' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  that->_bgColor = *col;
}

// Return true if the position 'pos' is inside the GBSurface 'that'
// boundary, false else
#if BUILDMODE != 0
inline
#endif
bool GBSurfaceIsPosInside(GBSurface* that, VecShort2D *pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
```

```
      PBErrCatch(GenBrushErr);
  }
#endif
  if (VecGet(pos, 0) >= 0 &&
    VecGet(pos, 1) >= 0 &&
    VecGet(pos, 0) < VecGet(GBSurfaceDim(that), 0) &&
    VecGet(pos, 1) < VecGet(GBSurfaceDim(that), 1))
    return true;
  else
    return false;
}

// Get the seet of layers of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
GSet* GBSurfaceLayers(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return &(that->_layers);
}

// Add a new GBLayer with dimensions 'dim' to the top of the stack
// of layers of the GBSurface 'that'
// Return the new GBLayer
#if BUILDMODE != 0
inline
#endif
GBLayer* GBSurfaceAddLayer(GBSurface* that, VecShort2D* dim) {
#if BUILDMODE == 0
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
  }
  if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'dim' is invalid (%d>0,%d>0)",
      VecGet(dim, 0), VecGet(dim, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  // Create the layer
  GBLayer* layer = GBLayerCreate(dim);
  // Add the new layer to the set of layers
  GSetAppend(&(that->_layers), layer);
  // Return the new layer
  return layer;
}

// 'fileName' to the top of the stack of layers of the GBSurface
// 'that'
// Return the new GBLayer
#if BUILDMODE != 0
inline
#endif
GBLayer* GBSurfaceAddLayerFromFile(GBSurface* that, char* fileName) {
```

```
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (fileName == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'fileName' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Create the layer from the file
  GBLayer* layer = GBLayerCreateFromFile(fileName);
  // If we could create the layer
  if (layer != NULL)
    // Add the layer to the surface
    GSetAppend(&(that->_layers), layer);
  // Return the new layer
  return layer;
}

// Get the 'iLayer'-th layer of the GBSurface 'that'
// 'iLayer' must be valid
#if BUILDMODE != 0
inline
#endif
GBLayer* GBSurfaceGetLayer(GBSurface* that, int iLayer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (iLayer < 0 || iLayer >= GSetNbElem(&(that->_layers))) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'iLayer' is invalid (0<=%d<%d)",
      iLayer, GSetNbElem(&(that->_layers)));
    PBErrCatch(GenBrushErr);
  }
#endif
  return (GBLayer*)GSetGet(&(that->_layers), iLayer);
}

// Get the number of layer of the GBSurface 'that'
#if BUILDMODE != 0
inline
#endif
int GBSurfaceNbLayer(GBSurface* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GSetNbElem(&(that->_layers));
}

// Set the _modified flag of all layers of the GBSurface 'that'
// to 'flag'
#if BUILDMODE != 0
```

104

```
inline
#endif
void GBSurfaceSetLayersModified(GBSurface* that, bool flag) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If there are layers
  if (GBSurfaceNbLayer(that) > 0) {
    // Declare an iterator on the set of layers
    GSetIterForward iter =
      GSetIterForwardCreateStatic(GBSurfaceLayers(that));
    do {
      GBLayer* layer = GSetIterGet(&iter);
      GBLayerSetModified(layer, flag);
    } while (GSetIterStep(&iter));
  }
}

// Set the position of the GBLayer 'layer' into the set of layers of
// the GBSurface 'that' to 'pos'
// If 'layer' can't be found in the surface do nothing
// 'pos' must be valid (0<='pos'<nbLayers)
#if BUILDMODE != 0
inline
#endif
void GBSurfaceSetLayerPos(GBSurface* that, GBLayer* layer, int pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (layer == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'layer' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos < 0 || pos >= GSetNbElem(&(that->_layers))) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is invalid (0<=%d<%d)",
      pos, GSetNbElem(&(that->_layers)));
    PBErrCatch(GenBrushErr);
  }
#endif
  // Get the current index of the layer to move
  int curPos = GSetGetIndexFirst(&(that->_layers), layer);
  // If we could find the layer
  if (curPos != -1) {
    // Move the layer
    GSetMoveElem(&(that->_layers), curPos, pos);
    // Set the _modified flag
    GBLayerSetModified(layer, true);
  }
}

// Remove the GBLayer 'layer' from the set of layers of the
// GBSurface 'that'
// The memory used by 'layer' is freed
```

```
#if BUILDMODE != 0
inline
#endif
void GBSurfaceRemoveLayer(GBSurface* that, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (layer == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'layer' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Remove the layer from the set
  GSetRemoveAll(&(that->_layers), layer);
  // Free the memory
  GBLayerFree(&layer);
}

// --------------- GBSurfaceImage -------------------------

// Get the filename of the GBSurfaceImage 'that'
#if BUILDMODE != 0
inline
#endif
char* GBSurfaceImageFileName(GBSurfaceImage* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_fileName;
}

// Set the filename of the GBSurfaceImage 'that' to 'fileName'
#if BUILDMODE != 0
inline
#endif
void GBSurfaceImageSetFileName(GBSurfaceImage* that, char* fileName) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If the filename is already allocated, free memory
  if (that->_fileName != NULL) {
    free(that->_fileName);
  }
  // If the new filename is not null
  if (fileName != NULL) {
    // Allocate memory and copy the new filename
    that->_fileName = PBErrMalloc(GenBrushErr,
      sizeof(char) * (strlen(fileName) + 1));
    strcpy(that->_fileName, fileName);
  }
```

```
}

// ---------------- GBEye ------------------------

// Return the type of the GBEye 'that'
#if BUILDMODE != 0
inline
#endif
GBEyeType _GBEyeGetType(GBEye* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_type;
}

// Get the scale of the GBEye
#if BUILDMODE != 0
inline
#endif
VecFloat3D* _GBEyeScale(GBEye* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return &(that->_scale);
}

// Get a copy of the scale of the GBEye
#if BUILDMODE != 0
inline
#endif
VecFloat3D _GBEyeGetScale(GBEye* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_scale;
}


// Get the translation of the GBEye
#if BUILDMODE != 0
inline
#endif
VecFloat2D* _GBEyeOrig(GBEye* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
```

```c
    return &(that->_orig);
}


// Get a copy of the translation of the GBEye
#if BUILDMODE != 0
inline
#endif
VecFloat2D _GBEyeGetOrig(GBEye* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_orig;
}


// Get the rotation of the GBEye (in radians)
#if BUILDMODE != 0
inline
#endif
float _GBEyeGetRot(GBEye* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_theta;
}

// Set the scale of the GBEye
#if BUILDMODE != 0
inline
#endif
void GBEyeSetScaleVec(GBEye* that, VecFloat3D* scale) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (scale == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'scale' is null");
    PBErrCatch(GenBrushErr);
  }
  if (ISEQUALF(VecGet(scale, 0), 0.0) == true ||
    ISEQUALF(VecGet(scale, 1), 0.0) == true) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'scale' is invalid (%f!=0 && %f!=0)",
      VecGet(scale, 0), VecGet(scale, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  VecCopy(&(that->_scale), scale);
  // Update the projection matrix
  GBEyeUpdateProj(that);
```

```
}
#if BUILDMODE != 0
inline
#endif
void GBEyeSetScaleFloat(GBEye* that, float scale) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (ISEQUALF(scale, 0.0) == true) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'scale' is invalid (%f!=0)", scale);
    PBErrCatch(GenBrushErr);
  }
#endif
  for (int i = 3; i--;)
    VecSet(&(that->_scale), i, scale);
  // Update the projection matrix
  GBEyeUpdateProj(that);
}


// Set the translation of the GBEye
#if BUILDMODE != 0
inline
#endif
void _GBEyeSetOrig(GBEye* that, VecFloat2D* trans) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (trans == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'trans' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  that->_orig = *trans;
  // Update the projection matrix
  GBEyeUpdateProj(that);
}


// Set the rotation of the GBEye (in radians)
#if BUILDMODE != 0
inline
#endif
void _GBEyeSetRot(GBEye* that, float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  that->_theta = theta;
  // Update the projection matrix
  GBEyeUpdateProj(that);
}
```

```c
// Get the matrix projection of the eye
#if BUILDMODE != 0
inline
#endif
MatFloat* _GBEyeProj(GBEye* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_proj;
}

// Call the appropriate GBEye<>Process according to the type of the
// GBEye 'that'
#if BUILDMODE != 0
inline
#endif
void _GBEyeProcess(GBEye* that, GBObjPod* pod) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pod == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pod' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Call the appropriate function according to the type of eye
  switch (that->_type) {
    case GBEyeTypeOrtho:
      GBEyeOrthoProcess((GBEyeOrtho*)that, pod);
      break;
    case GBEyeTypeIsometric:
      GBEyeIsometricProcess((GBEyeIsometric*)that, pod);
      break;
    default:
      switch (GBObjPodGetType(pod)) {
        case GBObjTypePoint:
          GBObjPodSetEyePoint(pod,
            VecClone(GBObjPodGetObjAsPoint(pod)));
          break;
        case GBObjTypeSCurve:
          GBObjPodSetEyeSCurve(pod,
            SCurveClone(GBObjPodGetObjAsSCurve(pod)));
          break;
        case GBObjTypeShapoid:
          GBObjPodSetEyeShapoid(pod,
            ShapoidClone(GBObjPodGetObjAsShapoid(pod)));
          break;
        default:
          break;
      }
      break;
  }
}
```

110

```
// --------------- GBEyeOrtho ------------------------

// Set the orientation the GBEyeOrtho
#if BUILDMODE != 0
inline
#endif
void GBEyeOrthoSetView(GBEyeOrtho* that, GBEyeOrthoView view) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  that->_view = view;
  // Update the projection matrix
  GBEyeUpdateProj((GBEye*)that);
}

// --------------- GBEyeIsometric ------------------------

// Set the angle around Y of the GBEyeOrtho to 'theta' (in radians)
#if BUILDMODE != 0
inline
#endif
void GBEyeIsometricSetRotY(GBEyeIsometric* that, float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  that->_thetaY = theta;
  GBEyeUpdateProj((GBEye*)that);
}

// Set the angle around Right of the GBEyeOrtho to 'theta'
// (in radians, in [-pi/2, pi/2])
// If 'theta' is out of range it is automatically bounded
// (ex: pi -> pi/2)
#if BUILDMODE != 0
inline
#endif
void GBEyeIsometricSetRotRight(GBEyeIsometric* that, float theta) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Ensure 'theta' is in bounds
  if (theta > PBMATH_HALFPI)
    theta = PBMATH_HALFPI;
  else if (theta < -PBMATH_HALFPI)
    theta = -PBMATH_HALFPI;
  that->_thetaRight = theta;
  GBEyeUpdateProj((GBEye*)that);
}
```

```
// Get the angle around Y of the GBEyeOrtho 'that'
#if BUILDMODE != 0
inline
#endif
float GBEyeIsometricGetRotY(GBEyeIsometric* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_thetaY;
}

// Get the angle around Right of the GBEyeOrtho 'that'
#if BUILDMODE != 0
inline
#endif
float GBEyeIsometricGetRotRight(GBEyeIsometric* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_thetaRight;
}

// --------------- GBHand -------------------------

// Return the type of the GBHand 'that'
#if BUILDMODE != 0
inline
#endif
GBHandType _GBHandGetType(GBHand* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_type;
}

// Call the appropriate GBHand<>Process according to the type of the
// GBHand 'that'
#if BUILDMODE != 0
inline
#endif
void _GBHandProcess(GBHand* that, GBObjPod* pod) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // If 'pod' has no viewed object do nothing
  if (GBObjPodGetEyeObj(pod) == NULL)
```

112

```
    return;
  // Call the appropriate function depending on the type of hand
  switch (that->_type) {
    case GBHandTypeDefault:
      GBHandDefaultProcess((GBHandDefault*)that, pod);
      break;
    default:
      break;
  }
}

// --------------- GBTool ------------------------

// Return a copy of the type of the GBTool 'that'
#if BUILDMODE != 0
inline
#endif
GBToolType GBToolGetType(GBTool* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_type;
}

// --------------- GBInkSolid ------------------------

// Return the type of the GBInk 'that'
#if BUILDMODE != 0
inline
#endif
GBInkType _GBInkGetType(GBInk* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_type;
}

// --------------- GInkSolid ------------------------

// Get the color of the GBInkSolid 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel GBInkSolidGet(GBInkSolid* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_color;
}
```

```
// Set the color of the GBInkSolid 'that' to 'col'
#if BUILDMODE != 0
inline
#endif
void GBInkSolidSet(GBInkSolid* that, GBPixel* col) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (col == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'col' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  that->_color = *col;
}


// --------------- GBTool -------------------------

// Function to call the appropriate GBTool<>Draw function according to
// type of GBTool 'that'
#if BUILDMODE != 0
inline
#endif
void _GBToolDraw(GBTool* that, GBObjPod* pod) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pod == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pod' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  switch (GBToolGetType(that)) {
    case GBToolTypePlotter:
      GBToolPlotterDraw((GBToolPlotter*)that, pod);
      break;
    default:
      break;
  }
}

// --------------- GBObjPod -------------------------

// Return the type of the object in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBObjType GBObjPodGetType(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
```

```
  }
#endif
  return that->_type;
}

// Return the object in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
void* GBObjPodGetObj(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_srcPoint;
}

// Return the object viewed by its attached eye in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
void* GBObjPodGetEyeObj(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_eyePoint;
}

// Return the object processed as Points by its attached hand in the
// GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GSetVecFloat* GBObjPodGetHandObjAsPoints(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return &(that->_handPoints);
}

// Return the object processed as Shapoids by its attached hand in the
// GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GSetShapoid* GBObjPodGetHandObjAsShapoids(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
```

```
  }
#endif
  return &(that->_handShapoids);
}


// Return the object processed as SCurves by its attached hand in the
// GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GSetSCurve* GBObjPodGetHandObjAsSCurves(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return &(that->_handSCurves);
}


// Return the eye in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBEye* GBObjPodGetEye(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_eye;
}

// Return the hand in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBHand* GBObjPodGetHand(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_hand;
}

// Return the tool in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBTool* GBObjPodGetTool(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
```

116

```
      sprintf(GenBrushErr->_msg, "'that' is null");
      PBErrCatch(GenBrushErr);
  }
#endif
  return that->_tool;
}

// Return the ink in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBInk* GBObjPodGetInk(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_ink;
}

// Return the layer in the GBObjPod 'that'
#if BUILDMODE != 0
inline
#endif
GBLayer* GBObjPodGetLayer(GBObjPod* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_layer;
}

// Set the Point viewed by its attached eye in the GBObjPod 'that'
// to 'point'
// If 'point' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetEyePoint(GBObjPod* that, VecFloat* point) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (point != NULL) {
    if (that->_eyePoint != NULL)
      VecFree(&(that->_eyePoint));
    that->_eyePoint = point;
  }
}

// Set the Shapoid viewed by its attached eye in the GBObjPod 'that'
// to 'shap'
// If 'shap' is null do nothing
#if BUILDMODE != 0
```

```
inline
#endif
void _GBObjPodSetEyeShapoid(GBObjPod* that, Shapoid* shap) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (shap != NULL) {
    if (that->_eyeShapoid != NULL)
      ShapoidFree(&(that->_eyeShapoid));
    that->_eyeShapoid = shap;
  }
}

// Set the SCurve viewed by its attached eye in the GBObjPod 'that'
// to 'curve'
// If 'curve' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetEyeSCurve(GBObjPod* that, SCurve* curve) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (curve != NULL) {
    if (that->_eyeSCurve != NULL)
      SCurveFree(&(that->_eyeSCurve));
    that->_eyeSCurve = curve;
  }
}

// Set the eye in the GBObjPod 'that' to 'eye'
// If 'eye' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetEye(GBObjPod* that, GBEye* eye) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (eye != NULL)
    that->_eye = eye;
}

// Set the hand in the GBObjPod 'that' to 'hand'
// If 'hand' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetHand(GBObjPod* that, GBHand* hand) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (hand != NULL)
    that->_hand = hand;
}

// Set the tool in the GBObjPod 'that' to 'tool'
// If 'tool' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetTool(GBObjPod* that, GBTool* tool) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (tool != NULL)
    that->_tool = tool;
}

// Set the ink in the GBObjPod 'that' to 'ink'
// If 'ink' is null do nothing
#if BUILDMODE != 0
inline
#endif
void _GBObjPodSetInk(GBObjPod* that, GBInk* ink) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (ink != NULL)
    that->_ink = ink;
}

// Set the layer in the GBObjPod 'that' to 'layer'
// If 'layer' is null do nothing
#if BUILDMODE != 0
inline
#endif
void GBObjPodSetLayer(GBObjPod* that, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (layer != NULL)
    that->_layer = layer;
}

// --------------- GenBrush -------------------------
```

```c
// Get the GBSurface of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBSurface* GBSurf(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_surf;
}

// Get the dimensions of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D* GBDim(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceDim(that->_surf);
}

// Get a copy of the dimensions of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
VecShort2D GBGetDim(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceGetDim(that->_surf);
}

// Get the final pixels of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel* GBFinalPixels(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceFinalPixels(that->_surf);
}

// Get the final pixel at position 'pos' of the GBSurface of the GB
```

```
// 'that'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
GBPixel* GBFinalPixel(GenBrush* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (!GBIsPosInside(that, pos)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is out of the surface (%d,%d)",
      VecGet(pos, 0), VecGet(pos, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceFinalPixel(GBSurf(that), pos);
}

// Get a copy of the final pixel at position 'pos' of the GBSurface
// of the GB 'that'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
GBPixel GBGetFinalPixel(GenBrush* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (!GBIsPosInside(that, pos)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is out of the surface (%d,%d)",
      VecGet(pos, 0), VecGet(pos, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceGetFinalPixel(GBSurf(that), pos);
}

// Set the final pixel at position 'pos' of the GBSurface of the GB
// 'that' to the pixel 'pix'
// 'pos' must be in the surface
#if BUILDMODE != 0
inline
#endif
void GBSetFinalPixel(GenBrush* that, VecShort2D* pos, GBPixel* pix) {
```

```c
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pix == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pix' is null");
    PBErrCatch(GenBrushErr);
  }
  if (!GBIsPosInside(that, pos)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is out of the surface (%d,%d)",
      VecGet(pos, 0), VecGet(pos, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  GBSurfaceSetFinalPixel(GBSurf(that), pos, pix);
}

// Get the final pixel at position 'pos' of the GBSurface of the GB
// 'that'
// If 'pos' is out of the surface return NULL
#if BUILDMODE != 0
inline
#endif
GBPixel* GBFinalPixelSafe(GenBrush* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (GBIsPosInside(that, pos))
    return GBSurfaceFinalPixel(GBSurf(that), pos);
  else
    return NULL;
}

// Get a copy of the final pixel at position 'pos' of the GBSurface
// of the GB 'that'
// If 'pos' is out of the surface return a transparent pixel
#if BUILDMODE != 0
inline
#endif
GBPixel GBGetFinalPixelSafe(GenBrush* that, VecShort2D* pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
```

```
      PBErrCatch(GenBrushErr);
    }
    if (pos == NULL) {
      GenBrushErr->_type = PBErrTypeNullPointer;
      sprintf(GenBrushErr->_msg, "'pos' is null");
      PBErrCatch(GenBrushErr);
    }
#endif
    if (GBIsPosInside(that, pos))
      return GBSurfaceGetFinalPixel(GBSurf(that), pos);
    else
      return GBColorTransparent;
}

// Set the final pixel at position 'pos' of the GBSurface of the GB
// 'that' to the pixel 'pix'
// If 'pos' is out of the surface do nothing
#if BUILDMODE != 0
inline
#endif
void GBSetFinalPixelSafe(GenBrush* that, VecShort2D* pos, GBPixel* pix) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pix == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pix' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (GBIsPosInside(that, pos))
    GBSurfaceSetFinalPixel(GBSurf(that), pos, pix);
}

// Get the type of the GBSurface of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBSurfaceType GBGetType(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceGetType(that->_surf);
}

// Get the background color of the GBSurface of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel* GBBgColor(GenBrush* that) {
```

123

```
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceBgColor(GBSurf(that));
}

// Get a copy of the background color of the GBSurface of the
// GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBPixel GBGetBgColor(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceGetBgColor(GBSurf(that));
}

// Set the background color of the GBSurface of the GenBrush
//  'that' to 'col'
#if BUILDMODE != 0
inline
#endif
void GBSetBgColor(GenBrush* that, GBPixel* col) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (col == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'col' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  GBSurfaceSetBgColor(GBSurf(that), col);
}

// Get the filename of the GBSurfaceImage of the GenBrush 'that'
// Return NULL if the surface is not a GBSurfaceImage
#if BUILDMODE != 0
inline
#endif
char* GBFileName(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (GBGetType(that) == GBSurfaceTypeImage)
    return GBSurfaceImageFileName((GBSurfaceImage*)GBSurf(that));
```

```
    else
      return NULL;
}

// Set the filename of the GBSurfaceImage of the GenBrush 'that'
// to 'fileName'
// Do nothing if the surface is not a GBSurfaceImage
#if BUILDMODE != 0
inline
#endif
void GBSetFileName(GenBrush* that, char* fileName) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (GBGetType(that) == GBSurfaceTypeImage)
    GBSurfaceImageSetFileName((GBSurfaceImage*)GBSurf(that), fileName);
}

// Get the area of the GBSurface of the Genbrush 'that'
#if BUILDMODE != 0
inline
#endif
int GBArea(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceArea(GBSurf(that));
}

// Return true if the position 'pos' is inside the GBSurface of the
// GenBrush 'that'
// boundary, false else
#if BUILDMODE != 0
inline
#endif
bool GBIsPosInside(GenBrush* that, VecShort2D *pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'pos' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceIsPosInside(GBSurf(that), pos);
}

// Return the layers of the surface of the GenBrush 'that'
#if BUILDMODE != 0
inline
```

```
#endif
GSet* GBLayers(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceLayers(GBSurf(that));
}

// Add a new GBLayer with dimensions 'dim' to the top of the stack
// of layers of the GBSurface of the GenBrush' that'
// Return the new GBLayer
#if BUILDMODE != 0
inline
#endif
GBLayer* GBAddLayer(GenBrush* that, VecShort2D* dim) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
  }
  if (VecGet(dim, 0) <= 0 || VecGet(dim, 1) <= 0) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'dim' is invalid (%d>0,%d>0)",
      VecGet(dim, 0), VecGet(dim, 1));
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceAddLayer(GBSurf(that), dim);
}

// Add a new GBLayer with content equals to the image located at
// 'fileName' to the top of the stack
// of layers of the GBSurface of the GenBrush' that'
// Return the new GBLayer
#if BUILDMODE != 0
inline
#endif
GBLayer* GBAddLayerFromFile(GenBrush* that, char* fileName) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (fileName == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'fileName' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceAddLayerFromFile(GBSurf(that), fileName);
}
```

126

```
// Get the 'iLayer'-th layer of the GBSurface of the GenBrush 'that'
// 'iLayer' must be valid
#if BUILDMODE != 0
inline
#endif
GBLayer* GBGetLayer(GenBrush* that, int iLayer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (iLayer < 0 ||
    iLayer >= GSetNbElem(GBSurfaceLayers(GBSurf(that)))) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'iLayer' is invalid (0<=%d<%d)",
      iLayer, GSetNbElem(GBSurfaceLayers(GBSurf(that))));
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceGetLayer(GBSurf(that), iLayer);
}


// Get the number of layer of the GBSurface of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
int GBGetNbLayer(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceNbLayer(GBSurf(that));
}


// Set the position of the GBLayer 'layer' into the set of layers of
// the GBSurface of the GenBrush'that' to 'pos'
// If 'layer' can't be found in the surface do nothing
// 'pos' must be valid (0<='pos'<nbLayers)
#if BUILDMODE != 0
inline
#endif
void GBSetLayerPos(GenBrush* that, GBLayer* layer, int pos) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (layer == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'layer' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos < 0 || pos >= GBGetNbLayer(that)) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'pos' is invalid (0<=%d<%d)",
      pos, GBGetNbLayer(that));
```

```
    PBErrCatch(GenBrushErr);
  }
#endif
  GBSurfaceSetLayerPos(GBSurf(that), layer, pos);
}

// Remove the GBLayer 'layer' from the set of layers of the
// GBSurface of the GenBrush 'that'
// The memory used by 'layer' is freed
#if BUILDMODE != 0
inline
#endif
void GBRemoveLayer(GenBrush* that, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (layer == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'layer' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Remove the graphical elements bounded to this layer
  GBRemovePod(that, NULL, NULL, NULL, NULL, NULL, layer);
  // Remove the layer from the surface
  GBSurfaceRemoveLayer(GBSurf(that), layer);
}

// Get the number of pod in the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
int GBGetNbPod(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GSetNbElem(&(that->_pods));
}

// Add a GBObjPod for the Point at position 'pos' to the GenBrush 'that'
// drawn with 'eye', 'hand' and 'tool' in layer 'layer'
// 'pos' must be a vector of 2 or more dimensions
#if BUILDMODE != 0
inline
#endif
void _GBAddPoint(GenBrush* that, VecFloat* pos, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (pos == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
```

```
      sprintf(GenBrushErr->_msg, "'pos' is null");
      PBErrCatch(GenBrushErr);
  }
#endif
  GSetAppend(&(that->_pods),
    GBObjPodCreatePoint(pos, eye, hand, tool, ink, layer));
}

// Add a GBObjPod for the Shapoid 'shap' to the GenBrush 'that'
// drawn with 'eye', 'hand' and 'tool' in layer 'layer'
// 'shap' 's dimension must be 2 or more
#if BUILDMODE != 0
inline
#endif
void _GBAddShapoid(GenBrush* that, Shapoid* shap, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (shap == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'shap' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  GSetAppend(&(that->_pods),
    GBObjPodCreateShapoid(shap, eye, hand, tool, ink, layer));
}

// Add a GBObjPod for the SCurve 'curve' to the GenBrush 'that'
// drawn with 'eye', 'hand' and 'tool' in layer 'layer'
// 'curve' 's dimension must be 2 or more
#if BUILDMODE != 0
inline
#endif
void _GBAddSCurve(GenBrush* that, SCurve* curve, GBEye* eye,
  GBHand* hand, GBTool* tool, GBInk* ink, GBLayer* layer) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (curve == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'curve' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  GSetAppend(&(that->_pods),
    GBObjPodCreateSCurve(curve, eye, hand, tool, ink, layer));
}

// Reset all the final pix of the surface of the GenBrush 'that' to its
// background color, and reset all the modified flag of layers to true
#if BUILDMODE != 0
inline
#endif
void GBFlush(GenBrush* that) {
```

```
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  GBSurfaceFlush(GBSurf(that));
}

// Return true if the surface of the GenBrush 'that' is same as the
// surface of the GenBrush 'gb'
// Else, return false
#if BUILDMODE != 0
inline
#endif
bool GBIsSameAs(GenBrush* that, GenBrush* gb) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (gb == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'gb' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GBSurfaceIsSameAs(GBSurf(that), GBSurf(gb));
}

// Add a GBPostProcessing of type 'type' to the GenBrush 'that'
// Return the GBPostProcessing
#if BUILDMODE != 0
inline
#endif
GBPostProcessing* GBAddPostProcess(GenBrush* that, GBPPType type) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  GBPostProcessing* post = GBPostProcessingCreate(type);
  GSetAppend(&(that->_postProcs), post);
  return post;
}

// Remove the GBPostProcessing 'post' from the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
void GBRemovePostProcess(GenBrush* that, GBPostProcessing* post) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
```

```
  if (GBGetNbPostProcs(that) > 0) {
    GSetIterForward iter =
      GSetIterForwardCreateStatic(GBPostProcs(that));
    bool flag = true;
    do {
      GBPostProcessing* p = GSetIterGet(&iter);
      if (post == p) {
        GBPostProcessingFree(&p);
        GSetIterRemoveElem(&iter);
      }
    } while (flag == true && GSetIterStep(&iter));
  }
  GSetRemoveFirst(&(that->_postProcs), post);
}

// Remove all the GBPostProcessing from the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
void GBRemoveAllPostProcess(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  while (GBGetNbPostProcs(that) > 0) {
    GBPostProcessing* post = GSetPop(&(that->_postProcs));
    GBPostProcessingFree(&post);
  }
}

// Get the 'iPost'-th post process of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GBPostProcessing* GBGetPostProcess(GenBrush* that, int iPost) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GSetGet(&(that->_postProcs), iPost);
}

// Get the GSet of GBPostProcessing of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
GSet* GBPostProcs(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return &(that->_postProcs);
}
```

```
// Get the number of GBPostProcessing of the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
int GBGetNbPostProcs(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return GSetNbElem(&(that->_postProcs));
}

// Remove all the GBObjPods from the GenBrush 'that'
#if BUILDMODE != 0
inline
#endif
void GBRemoveAllPod(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeInvalidArg;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  while (GBGetNbPod(that) > 0) {
    GBObjPod* pod = GSetPop(&(that->_pods));
    GBObjPodFree(&pod);
  }
}


// =============== GTK Functions ===================

#if BUILDWITHGRAPHICLIB == 1
  #include "genbrush-inline-GTK.c"
#endif
```

3.3 genbrush-GTK.c

```
// ============ GENBRUSH-GTK.C ===============

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

// Callback for the 'activate' event on a GBSurfaceApp
void GBSurfaceAppCallbackActivate(GtkApplication* app,
  gpointer data);

// Callback for the 'draw' event on a GBSurfaceApp
gboolean GBSurfaceAppCallbackDraw(GtkWidget *widget, cairo_t *cr,
  gpointer data);

// Callback for the 'config' event on a GBSurfaceApp
gboolean GBSurfaceAppCallbackConfigEvt (GtkWidget *widget,
  GdkEventConfigure *event, gpointer data);

// Callback for the 'draw' event on a GBSurfaceWidget
```

```
gboolean GBSurfaceWidgetCallbackDraw(GtkWidget *widget, cairo_t *cr,
  gpointer data);

// Callback for the 'config' event on a GBSurfaceWidget
gboolean GBSurfaceWidgetCallbackConfigEvt(GtkWidget *widget,
  GdkEventConfigure *event, gpointer data);

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

// Callback for the 'draw' event on a GBSurfaceApp
gboolean GBSurfaceAppCallbackDraw(GtkWidget *widget, cairo_t *cr,
  gpointer data) {
  (void)widget;
  // Declare a variable to convert the data into the GBSurfaceApp
  GBSurfaceApp* GBApp = (GBSurfaceApp*)data;
  // Paint the final pixels of the GBSurface on the Cairo surface
  cairo_set_source_surface(cr, GBApp->_cairoSurf, 0, 0);
  cairo_paint(cr);
  return FALSE;
}

// Callback for the 'config' event on a GBSurfaceApp
gboolean GBSurfaceAppCallbackConfigEvt(GtkWidget *widget,
  GdkEventConfigure *event, gpointer data) {
  (void)event;
  // Declare a variable to convert the data into the GBSurfaceApp
  GBSurfaceApp* GBApp = (GBSurfaceApp*)data;
  // Attach the GBPixel to the cairo surface
  GBApp->_cairoSurf = cairo_image_surface_create_for_data(
    (unsigned char*)GBSurfaceFinalPixels(&(GBApp->_surf)),
    CAIRO_FORMAT_ARGB32,
    gtk_widget_get_allocated_width(widget),
    gtk_widget_get_allocated_height(widget),
    cairo_format_stride_for_width(CAIRO_FORMAT_ARGB32,
      gtk_widget_get_allocated_width(widget)));
  // Draw the surface
  cairo_t *cr = cairo_create(GBApp->_cairoSurf);
  GBSurfaceAppCallbackDraw(widget, cr, GBApp);
  cairo_destroy (cr);
  return TRUE;
}

// Callback for the activate event on a GBSurfaceApp
void GBSurfaceAppCallbackActivate(GtkApplication* app,
  gpointer data) {
  // Declare a variable to convert the data into the GBSurfaceApp
  GBSurfaceApp* GBApp = (GBSurfaceApp*)data;
  // Create the window
  GBApp->_window = gtk_application_window_new(app);
  // Set the title of the window
  gtk_window_set_title(GTK_WINDOW(GBApp->_window), GBApp->_title);
  // Set the size of the window
  gtk_window_set_default_size(GTK_WINDOW(GBApp->_window),
    VecGet(GBSurfaceDim(&(GBApp->_surf)), 0),
    VecGet(GBSurfaceDim(&(GBApp->_surf)), 1));
  // Set the window as non resizable
  gtk_window_set_resizable(GTK_WINDOW(GBApp->_window), false);
  // Create the drawing area
  GBApp->_drawingArea = gtk_drawing_area_new();
  // Set the size of the drawing area
  gtk_widget_set_size_request(GBApp->_drawingArea,
    VecGet(GBSurfaceDim(&(GBApp->_surf)), 0),
```

```
      VecGet(GBSurfaceDim(&(GBApp->_surf)), 1));
  // Insert the drawing area into the window
  gtk_container_add(GTK_CONTAINER(GBApp->_window), GBApp->_drawingArea);
  // Connect the events
  g_signal_connect(GBApp->_drawingArea, "draw",
    G_CALLBACK (GBSurfaceAppCallbackDraw), GBApp);
  g_signal_connect(GBApp->_drawingArea,"configure-event",
    G_CALLBACK (GBSurfaceAppCallbackConfigEvt), GBApp);
  if (GBApp->_idleFun != NULL)
    g_timeout_add(GBApp->_idleMs, GBApp->_idleFun, GBApp);
  // Display the window
  gtk_widget_show_all(GBApp->_window);
}


// Create a new GBSurfaceApp with title 'title' and
// dimensions 'dim'
// The surface is initialized with one layer of white pixels
// (rgba = {255})
GBSurfaceApp* GBSurfaceAppCreate(VecShort2D* dim, char* title) {
#if BUILDMODE == 0
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
  }
  if (title == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'title' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Allocate memory
  GBSurfaceApp* that = PBErrMalloc(GenBrushErr, sizeof(GBSurfaceApp));
  // Set the title of the application
  that->_title = PBErrMalloc(GenBrushErr,
    sizeof(char) * (strlen(title) + 1));
  strcpy(that->_title, title);
  // Create the surface
  that->_surf = GBSurfaceCreateStatic(GBSurfaceTypeApp, dim);
  // Create the application
  that->_app = gtk_application_new(NULL, G_APPLICATION_FLAGS_NONE);
  // Set the idle properties
  that->_idleFun = NULL;
  that->_idleMs = 0;
  // Set the return status
  that->_returnStatus = 0;
  // Connect the 'activate' event
  // Other properties will be set up by the activate event
  g_signal_connect (that->_app, "activate",
    G_CALLBACK (GBSurfaceAppCallbackActivate), that);
  // Return the GBSurfaceApp
  return that;
}


// Set the idle function of the GBSurfaceApp 'that' to 'idleFun'
// with a timeout of 'idleMs'
// The interface of the 'idleFun' is
// gint tick(gpointer data)
// the argument of 'idleFun' is a pointer to GBSurfaceApp
void GBSurfaceSetIdle(GBSurfaceApp* that,
  gint (*idleFun)(gpointer), int idleMs) {
#if BUILDMODE == 0
```

```c
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (idleFun == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'idleFun' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Memorize the idle properties
  that->_idleFun = idleFun;
  that->_idleMs = idleMs;
}

// Render the GBSurfaceApp 'that'
// This function block the execution until the app is killed
// Return true if the status of the app when closed was 0, false else
bool GBSurfaceAppRender(GBSurfaceApp* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Run the application
  int argc = 0;
  char* *argv = NULL;
  that->_returnStatus =
    g_application_run(G_APPLICATION(that->_app), argc, argv);
  // After the application has been killed
  // Return the success code
  return (that->_returnStatus == 0 ? true : false);
}

// Free the GBSurfaceApp 'that'
void GBSurfaceAppFree(GBSurfaceApp** that) {
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  g_object_unref((*that)->_app);
  free((*that)->_title);
}

// Free the GBSurfaceWidget 'that'
void GBSurfaceWidgetFree(GBSurfaceWidget** that) {
  (void)that;
}

// Create a GenBrush with a blank GBSurfaceApp
GenBrush* GBCreateApp(VecShort2D* dim, char* title) {
#if BUILDMODE == 0
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
  }
  if (title == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'title' is null");
```

```
      PBErrCatch(GenBrushErr);
  }
#endif
  // Allocate memory
  GenBrush* that = PBErrMalloc(GenBrushErr, sizeof(GenBrush));
  // Set properties
  that->_surf = (GBSurface*)GBSurfaceAppCreate(dim, title);
  that->_pods = GSetCreateStatic();
  // Return the new GenBrush
  return that;
}

// Set the idle function of the GBSurfaceApp of the GenBrush 'that'
// to 'idleFun' with a timeout of 'idleMs'
// The interface of the 'idleFun' is
// gint tick(gpointer data)
// the argument of 'idleFun' is a pointer to GBSurfaceApp
// If the surface of the app is not a GBSurfaceTypeApp, do nothing
void GBSetIdle(GenBrush* that, gint (*idleFun)(gpointer), int idleMs) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
  if (idleFun == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'idleFun' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (GBSurfaceGetType(that->_surf) == GBSurfaceTypeApp)
    GBSurfaceSetIdle((GBSurfaceApp*)(that->_surf), idleFun, idleMs);
}

// Create a GenBrush with a blank GBSurfaceWidget and an eye
// of type 'eyeType'
// If 'eyeType' is GBEyeTypeOrtho the view is front
GenBrush* GBCreateWidget(VecShort2D* dim) {
#if BUILDMODE == 0
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Allocate memory
  GenBrush* that = PBErrMalloc(GenBrushErr, sizeof(GenBrush));
  // Set properties
  that->_surf = (GBSurface*)GBSurfaceWidgetCreate(dim);
  that->_pods = GSetCreateStatic();
  // Return the new GenBrush
  return that;
}

// Create a new GBSurfaceWidget with dimensions 'dim'
GBSurfaceWidget* GBSurfaceWidgetCreate(VecShort2D* dim) {
#if BUILDMODE == 0
  if (dim == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'dim' is null");
    PBErrCatch(GenBrushErr);
```

```
  }
#endif
  // Allocate memory
  GBSurfaceWidget* that = PBErrMalloc(GenBrushErr,
    sizeof(GBSurfaceWidget));
  // Create the surface
  that->_surf = GBSurfaceCreateStatic(GBSurfaceTypeWidget, dim);
  // Create the drawing area
  that->_drawingArea = gtk_drawing_area_new();
  // Set the size of the drawing area
  gtk_widget_set_size_request(that->_drawingArea,
    VecGet(dim, 0), VecGet(dim, 1));
  // Connect to the draw event
  g_signal_connect(that->_drawingArea, "draw",
    G_CALLBACK (GBSurfaceWidgetCallbackDraw), that);
  // Connect to the configure event
  g_signal_connect(that->_drawingArea,"configure-event",
    G_CALLBACK (GBSurfaceWidgetCallbackConfigEvt), that);
  // Return the GBSurfaceWidget
  return that;
}


// Callback for the 'draw' event on a GBSurfaceWidget
gboolean GBSurfaceWidgetCallbackDraw(GtkWidget *widget, cairo_t *cr,
  gpointer data) {
  (void)widget;
  // Declare a variable to convert the data into the GBSurfaceApp
  GBSurfaceWidget* GBWidget = (GBSurfaceWidget*)data;
  // Paint the final pixels of the GBSurface on the Cairo surface
  cairo_set_source_surface(cr, GBWidget->_cairoSurf, 0, 0);
  cairo_paint (cr);
  return FALSE;
}


// Render the GBSurfaceWidget 'that'
void GBSurfaceWidgetRender(GBSurfaceWidget* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Request the update of the widget area
  gtk_widget_queue_draw_area(that->_drawingArea, 0.0, 0.0,
    gtk_widget_get_allocated_width(that->_drawingArea),
    gtk_widget_get_allocated_height(that->_drawingArea));
}


// Callback for the 'config' event on a GBSurfaceWidget
gboolean GBSurfaceWidgetCallbackConfigEvt(GtkWidget *widget,
  GdkEventConfigure *event, gpointer data) {
  (void)event;
  // Declare a variable to convert the data into the GBSurfaceApp
  GBSurfaceWidget* GBWidget = (GBSurfaceWidget*)data;
  // Attach the GBPixel to the cairo surface
  GBWidget->_cairoSurf = cairo_image_surface_create_for_data(
    (unsigned char* )GBSurfaceFinalPixels(&(GBWidget->_surf)),
    CAIRO_FORMAT_ARGB32,
    gtk_widget_get_allocated_width(widget),
    gtk_widget_get_allocated_height(widget),
    cairo_format_stride_for_width(CAIRO_FORMAT_ARGB32,
```

```
      gtk_widget_get_allocated_width(widget)));
  // Draw the surface
  cairo_t *cr = cairo_create(GBWidget->_cairoSurf);
  GBSurfaceWidgetCallbackDraw(widget, cr, GBWidget);
  cairo_destroy (cr);
  return TRUE;
}
```

3.4   genbrush-inline-GTK.c

```
// ============ GENBRUSH-INLINE-GTK.C ================

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

// Close the GBSurfaceApp 'that'
#if BUILDMODE != 0
inline
#endif
void GBSurfaceAppClose(GBSurfaceApp* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  // Request the killing of the drawing area
  gtk_window_close(GTK_WINDOW(that->_window));
}

// Refresh the content of the GBSurfaceApp 'that'
#if BUILDMODE != 0
inline
#endif
void GBSurfaceAppRefresh(GBSurfaceApp* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  gtk_widget_queue_draw_area(that->_drawingArea, 0.0, 0.0,
    gtk_widget_get_allocated_width(that->_drawingArea),
    gtk_widget_get_allocated_height(that->_drawingArea));
}

// Return the GtkWidget of the GBSurfaceWidget 'that'
#if BUILDMODE != 0
inline
#endif
GtkWidget* GBSurfaceWidgetGetGtkWidget(GBSurfaceWidget* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_drawingArea;
```

```
}

// Return the GtkWidget of the GBSurfaceApp 'that'
#if BUILDMODE != 0
inline
#endif
GtkWidget* GBSurfaceAppGetGtkWidget(GBSurfaceApp* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  return that->_drawingArea;
}

// Return the GtkWidget of the GenBrush 'that', or NULL if the surface
// of the GenBursh is not a GBSurfaceWidget
#if BUILDMODE != 0
inline
#endif
GtkWidget* GBGetGtkWidget(GenBrush* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GenBrushErr->_type = PBErrTypeNullPointer;
    sprintf(GenBrushErr->_msg, "'that' is null");
    PBErrCatch(GenBrushErr);
  }
#endif
  if (GBSurfaceGetType(that->_surf) == GBSurfaceTypeWidget)
    return GBSurfaceWidgetGetGtkWidget((GBSurfaceWidget*)(that->_surf));
  else
    return NULL;
}
```

4 Makefile

```
#directory
PBERRDIR=../PBErr
PBMATHDIR=../PBMath
GSETDIR=../GSet
SHAPOIDDIR=../Shapoid
BCURVEDIR=../BCurve
GTREEDIR=../GTree
PBJSONDIR=../PBJson


# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILDMODE=1

# 0: monolith version, the GBSurface is rendered toward a TGA image
# 1: GTK version, the GBSurface is rendered toward a TGA image or
#    a GtkWidget
BUILDWITHGRAPHICLIB=0

include $(PBERRDIR)/Makefile.inc
```

```
ifeq ($(BUILDWITHGRAPHICLIB),1)
GTKPARAM=`pkg-config --cflags gtk+-3.0` -DBUILDWITHGRAPHICLIB=$(BUILDWITHGRAPHICLIB)
GTKLINK=`pkg-config --libs gtk+-3.0`
CAIROPARAM=`pkg-config --cflags cairo`
CAIROLINK=`pkg-config --libs cairo`
endif

INCPATH=-I./ -I$(PBERRDIR)/ -I$(PBMATHDIR)/ -I$(GSETDIR)/ -I$(SHAPOIDDIR)/ -I$(BCURVEDIR)/ -I$(PBJSONDIR)/ -I$(GTREE
BUILDOPTIONS=$(BUILDPARAM) $(INCPATH) $(GTKPARAM) $(CAIROPARAM) -ggdb

# compiler
COMPILER=gcc

#rules
all : main

main: main.o pberr.o bcurve.o shapoid.o pbmath.o gset.o genbrush.o pbjson.o gtree.o Makefile
$(COMPILER) main.o pberr.o bcurve.o shapoid.o pbmath.o gset.o genbrush.o pbjson.o gtree.o $(LINKOPTIONS) -o main $(GT

main.o : main.c $(PBERRDIR)/pberr.h $(SHAPOIDDIR)/shapoid.h $(SHAPOIDDIR)/shapoid-inline.c $(PBMATHDIR)/pbmath.h $(PI
$(COMPILER) $(BUILDOPTIONS) -c main.c

genbrush.o : genbrush.c genbrush-GTK.c genbrush.h genbrush-GTK.h genbrush-inline-GTK.c genbrush-inline.c $(BCURVEDIR)
$(COMPILER) $(BUILDOPTIONS) -c genbrush.c

pbjson.o : $(PBJSONDIR)/pbjson.c $(PBJSONDIR)/pbjson-inline.c $(PBJSONDIR)/pbjson.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c $(PBJSONDIR)/pbjson.c

gtree.o : $(GTREEDIR)/gtree.c $(GTREEDIR)/gtree.h $(GTREEDIR)/gtree-inline.c Makefile $(GSETDIR)/gset-inline.c $(GSET
$(COMPILER) $(BUILDOPTIONS) -c $(GTREEDIR)/gtree.c

bcurve.o : $(BCURVEDIR)/bcurve.c $(BCURVEDIR)/bcurve-inline.c $(BCURVEDIR)/bcurve.h $(SHAPOIDDIR)/shapoid.h $(SHAPOII
$(COMPILER) $(BUILDOPTIONS) -c $(BCURVEDIR)/bcurve.c

shapoid.o : $(SHAPOIDDIR)/shapoid.c $(SHAPOIDDIR)/shapoid.h $(SHAPOIDDIR)/shapoid-inline.c $(PBMATHDIR)/pbmath.h $(PI
$(COMPILER) $(BUILDOPTIONS) -c $(SHAPOIDDIR)/shapoid.c

pberr.o : $(PBERRDIR)/pberr.c $(PBERRDIR)/pberr.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c $(PBERRDIR)/pberr.c

pbmath.o : $(PBMATHDIR)/pbmath.c $(PBMATHDIR)/pbmath-inline.c $(PBMATHDIR)/pbmath.h Makefile $(PBERRDIR)/pberr.h
$(COMPILER) $(BUILDOPTIONS) -c $(PBMATHDIR)/pbmath.c

gset.o : $(GSETDIR)/gset.c $(GSETDIR)/gset-inline.c $(GSETDIR)/gset.h Makefile $(PBERRDIR)/pberr.h
$(COMPILER) $(BUILDOPTIONS) -c $(GSETDIR)/gset.c

clean :
rm -rf *.o main

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

unitTest :
main > unitTest.txt; diff unitTest.txt unitTestRef.txt
```

5   Unit tests

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "genbrush.h"

#define RANDOMSEED 0


void UnitTestGBPixelBlendNormal() {
  GBPixel pixA;
  GBPixel pixB;
  pixA = (GBPixel){._rgba[GBPixelRed]=100, ._rgba[GBPixelGreen]=0,
    ._rgba[GBPixelBlue]=0, ._rgba[GBPixelAlpha]=100};
  pixB = (GBPixel){._rgba[GBPixelRed]=0, ._rgba[GBPixelGreen]=100,
    ._rgba[GBPixelBlue]=0, ._rgba[GBPixelAlpha]=100};
  GBPixelBlendNormal(&pixA, &pixB);
  if (pixA._rgba[GBPixelRed] != 50 ||
    pixA._rgba[GBPixelGreen] != 50 ||
    pixA._rgba[GBPixelBlue] != 0 ||
    pixA._rgba[GBPixelAlpha] != 200) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBPixelBlendNormal failed");
    PBErrCatch(GenBrushErr);
  }
  pixA = (GBPixel){._rgba[GBPixelRed]=100, ._rgba[GBPixelGreen]=0,
    ._rgba[GBPixelBlue]=0, ._rgba[GBPixelAlpha]=255};
  pixB = (GBPixel){._rgba[GBPixelRed]=0, ._rgba[GBPixelGreen]=100,
    ._rgba[GBPixelBlue]=0, ._rgba[GBPixelAlpha]=255};
  GBPixelBlendNormal(&pixA, &pixB);
  if (pixA._rgba[GBPixelRed] != 50 ||
    pixA._rgba[GBPixelGreen] != 50 ||
    pixA._rgba[GBPixelBlue] != 0 ||
    pixA._rgba[GBPixelAlpha] != 255) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBPixelBlendNormal failed");
    PBErrCatch(GenBrushErr);
  }
  pixA = (GBPixel){._rgba[GBPixelRed]=100, ._rgba[GBPixelGreen]=0,
    ._rgba[GBPixelBlue]=0, ._rgba[GBPixelAlpha]=255};
  pixB = (GBPixel){._rgba[GBPixelRed]=0, ._rgba[GBPixelGreen]=100,
    ._rgba[GBPixelBlue]=0, ._rgba[GBPixelAlpha]=25};
  GBPixelBlendNormal(&pixA, &pixB);
  if (pixA._rgba[GBPixelRed] != 5 ||
    pixA._rgba[GBPixelGreen] != 95 ||
    pixA._rgba[GBPixelBlue] != 0 ||
    pixA._rgba[GBPixelAlpha] != 255) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBPixelBlendNormal failed");
    PBErrCatch(GenBrushErr);
  }
  printf("UnitTestGBPixelBlendNormal OK\n");
}

void UnitTestGBPixelBlendOver() {
  GBPixel pixA;
  GBPixel pixB;
  pixA = (GBPixel){._rgba[GBPixelRed]=100, ._rgba[GBPixelGreen]=0,
    ._rgba[GBPixelBlue]=0, ._rgba[GBPixelAlpha]=100};
```

```
      pixB = (GBPixel){._rgba[GBPixelRed]=0, ._rgba[GBPixelGreen]=100,
        ._rgba[GBPixelBlue]=0, ._rgba[GBPixelAlpha]=25};
      GBPixelBlendOver(&pixA, &pixB);
      if (pixA._rgba[GBPixelRed] != 61 ||
        pixA._rgba[GBPixelGreen] != 39 ||
        pixA._rgba[GBPixelBlue] != 0 ||
        pixA._rgba[GBPixelAlpha] != 125) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBPixelBlendOver failed");
        PBErrCatch(GenBrushErr);
      }
      pixA = (GBPixel){._rgba[GBPixelRed]=100, ._rgba[GBPixelGreen]=0,
        ._rgba[GBPixelBlue]=0, ._rgba[GBPixelAlpha]=255};
      pixB = (GBPixel){._rgba[GBPixelRed]=0, ._rgba[GBPixelGreen]=100,
        ._rgba[GBPixelBlue]=0, ._rgba[GBPixelAlpha]=255};
      GBPixelBlendOver(&pixA, &pixB);
      if (pixA._rgba[GBPixelRed] != 0 ||
        pixA._rgba[GBPixelGreen] != 100 ||
        pixA._rgba[GBPixelBlue] != 0 ||
        pixA._rgba[GBPixelAlpha] != 255) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBPixelBlendOver failed");
        PBErrCatch(GenBrushErr);
      }
      printf("UnitTestGBPixelBlendOver OK\n");
    }

    void UnitTestGBPixel() {
      UnitTestGBPixelBlendNormal();
      UnitTestGBPixelBlendOver();
      printf("UnitTestGBPixel OK\n");
    }

    void UnitTestGBLayerCreateFree() {
      VecShort2D dim = VecShortCreateStatic2D();
      VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
      GBLayer* layer = GBLayerCreate(&dim);
      VecShort2D v = VecShortCreateStatic2D();
      if (VecIsEqual(&(layer->_pos), &v) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBLayerCreate failed");
        PBErrCatch(GenBrushErr);
      }
      if (VecIsEqual(&(layer->_prevPos), &v) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBLayerCreate failed");
        PBErrCatch(GenBrushErr);
      }
      if (VecIsEqual(&(layer->_dim), &dim) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBLayerCreate failed");
        PBErrCatch(GenBrushErr);
      }
      if (layer->_pix == NULL) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBLayerCreate failed");
        PBErrCatch(GenBrushErr);
      }
      if (layer->_blendMode != GBLayerBlendModeDefault) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBLayerCreate failed");
        PBErrCatch(GenBrushErr);
```

142

```
  }
  if (layer->_modified == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerCreate failed");
    PBErrCatch(GenBrushErr);
  }
  if (layer->_stackPos != GBLayerStackPosBg) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerCreate failed");
    PBErrCatch(GenBrushErr);
  }
  GBLayerFree(&layer);
  printf("UnitTestGBLayerCreateFree OK\n");
}

void UnitTestGBLayerArea() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
  GBLayer* layer = GBLayerCreate(&dim);
  if (GBLayerArea(layer) != 12) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerArea failed");
    PBErrCatch(GenBrushErr);
  }
  GBLayerFree(&layer);
  printf("UnitTestGBLayerArea OK\n");
}

void UnitTestGBLayerGetSet() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
  GBLayer* layer = GBLayerCreate(&dim);
  if (GBLayerPos(layer) != &(layer->_pos)) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerPos failed");
    PBErrCatch(GenBrushErr);
  }
  VecShort2D p = VecShortCreateStatic2D();
  VecSet(&p, 0, 1); VecSet(&p, 1, 2);
  GBLayerSetPos(layer, &p);
  if (VecIsEqual(GBLayerPos(layer), &p) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerSetPos failed");
    PBErrCatch(GenBrushErr);
  }
  VecSet(&p, 0, 0); VecSet(&p, 1, 0);
  if (VecIsEqual(&(layer->_prevPos), &p) == false ||
    layer->_modified == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerSetPos failed");
    PBErrCatch(GenBrushErr);
  }
  layer->_modified = false;
  VecSet(&p, 0, 3); VecSet(&p, 1, 4);
  GBLayerSetPos(layer, &p);
  VecSet(&p, 0, 1); VecSet(&p, 1, 2);
  if (VecIsEqual(&(layer->_prevPos), &p) == false ||
    layer->_modified == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerSetPos failed");
    PBErrCatch(GenBrushErr);
  }
```

```
p = GBLayerGetPos(layer);
if (VecIsEqual(GBLayerPos(layer), &p) == false) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerGetPos failed");
  PBErrCatch(GenBrushErr);
}
if (GBLayerPrevPos(layer) != &(layer->_prevPos)) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerPrevPos failed");
  PBErrCatch(GenBrushErr);
}
p = GBLayerGetPrevPos(layer);
if (VecIsEqual(GBLayerPrevPos(layer), &p) == false) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerGetPrevPos failed");
  PBErrCatch(GenBrushErr);
}
if (GBLayerDim(layer) != &(layer->_dim)) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerDim failed");
  PBErrCatch(GenBrushErr);
}
p = GBLayerGetDim(layer);
if (VecIsEqual(GBLayerDim(layer), &p) == false) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerGetDim failed");
  PBErrCatch(GenBrushErr);
}
if (GBLayerGetBlendMode(layer) != layer->_blendMode) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerGetBlendMode failed");
  PBErrCatch(GenBrushErr);
}
GBLayerSetBlendMode(layer, GBLayerBlendModeNormal);
if (GBLayerGetBlendMode(layer) != GBLayerBlendModeNormal) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerSetBlendMode failed");
  PBErrCatch(GenBrushErr);
}
if (GBLayerIsModified(layer) != true) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerIsModified failed");
  PBErrCatch(GenBrushErr);
}
GBLayerSetModified(layer, false);
if (GBLayerIsModified(layer) != false) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerSetModified failed");
  PBErrCatch(GenBrushErr);
}
if (GBLayerGetStackPos(layer) != GBLayerStackPosBg) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerGetStackPos failed");
  PBErrCatch(GenBrushErr);
}
GBLayerSetStackPos(layer, GBLayerStackPosFg);
if (GBLayerGetStackPos(layer) != GBLayerStackPosFg ||
  GBLayerIsModified(layer) == false) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerSetStackPos failed");
  PBErrCatch(GenBrushErr);
}
```

```
if (GBLayerPixels(layer) != layer->_pix) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerPixels failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, -1); VecSet(&p, 1, 2);
if (GBLayerPixelSafe(layer, &p) != NULL) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerPixelSafe failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, 1); VecSet(&p, 1, 2);
if (GBLayerPixel(layer, &p) != layer->_pix + 9) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerPixel failed");
  PBErrCatch(GenBrushErr);
}
if (GBLayerPixelSafe(layer, &p) != layer->_pix + 9) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerPixelSafe failed");
  PBErrCatch(GenBrushErr);
}
GBPixel pix = GBColorWhite;
float depth = 1.0;
GBLayerAddPixel(layer, &p, &pix, depth);
if (GSetNbElem(GBLayerPixel(layer, &p)) != 1) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerAddPixel failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, -1); VecSet(&p, 1, 2);
GBLayerAddPixelSafe(layer, &p, &pix, depth);
VecSet(&p, 0, 1); VecSet(&p, 1, 2);
GBLayerAddPixelSafe(layer, &p, &pix, depth);
if (GSetNbElem(GBLayerPixel(layer, &p)) != 2) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerAddPixelSafe failed");
  PBErrCatch(GenBrushErr);
}
if (GBLayerIsPosInside(layer, &p) == false) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerIsPosInside failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, 10); VecSet(&p, 1, 2);
if (GBLayerIsPosInside(layer, &p) == true) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerIsPosInside failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, 1); VecSet(&p, 1, -2);
if (GBLayerIsPosInside(layer, &p) == true) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerIsPosInside failed");
  PBErrCatch(GenBrushErr);
}
GBLayerFlush(layer);
VecSet(&p, 0, 1); VecSet(&p, 1, 2);
if (GSetNbElem(GBLayerPixel(layer, &p)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBLayerFlush failed");
  PBErrCatch(GenBrushErr);
```

```
  }
  GBLayerFree(&layer);
  printf("UnitTestGBLayerGetSet OK\n");
}

void UnitTestGBLayerCreateFromFile() {
  GBLayer* layer = GBLayerCreateFromFile("./testBottomLeft.tga");
  if (layer == NULL) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerCreateFromFile failed");
    PBErrCatch(GenBrushErr);
  }
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3); VecSet(&dim, 1, 4);
  if (VecIsEqual(&dim, GBLayerDim(layer)) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerCreateFromFile failed");
    PBErrCatch(GenBrushErr);
  }
  unsigned char check[48] = {
    91,0,163,192, 52,0,202,194, 23,0,231,205,
    153,0,102,200, 106,0,148,192, 64,0,190,193,
    211,0,43,221, 168,0,86,204, 121,0,133,193,
    255,0,0,255, 226,0,28,230, 184,0,70,209
  };
  for (int iPix = 0; iPix < 12; iPix++) {
    GBPixel pix =
      ((GBStackedPixel*)GSetGet(layer->_pix + iPix, 0))->_val;
    if (pix._rgba[GBPixelRed] != check[4 * iPix] ||
      pix._rgba[GBPixelGreen] != check[4 * iPix + 1] ||
      pix._rgba[GBPixelBlue] != check[4 * iPix + 2] ||
      pix._rgba[GBPixelAlpha] != check[4 * iPix + 3]) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBLayerCreateFromFile1 failed");
      PBErrCatch(GenBrushErr);
    }
  }
  GBLayerFree(&layer);
  layer = GBLayerCreateFromFile("./testTopLeft.tga");
  if (layer == NULL) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerCreateFromFile failed");
    PBErrCatch(GenBrushErr);
  }
  if (VecIsEqual(&dim, GBLayerDim(layer)) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerCreateFromFile failed");
    PBErrCatch(GenBrushErr);
  }
  for (int iPix = 0; iPix < 12; iPix++) {
    GBPixel pix =
      ((GBStackedPixel*)GSetGet(layer->_pix + iPix, 0))->_val;
    if (pix._rgba[GBPixelRed] != check[4 * iPix] ||
      pix._rgba[GBPixelGreen] != check[4 * iPix + 1] ||
      pix._rgba[GBPixelBlue] != check[4 * iPix + 2] ||
      pix._rgba[GBPixelAlpha] != check[4 * iPix + 3]) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBLayerCreateFromFile2 failed");
      PBErrCatch(GenBrushErr);
    }
  }
  GBLayerFree(&layer);
```

```
    printf("UnitTestGBLayerCreateFromFile OK\n");
}

void UnitTestGBLayerGetBoundaryInSurface() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 800); VecSet(&dim, 1, 600);
  GBSurface* surf = GBSurfaceCreate(GBSurfaceTypeImage, &dim);
  VecSet(&dim, 0, 200); VecSet(&dim, 1, 100);
  GBLayer* layer = GBLayerCreate(&dim);
  VecShort2D pos = VecShortCreateStatic2D();
  VecSet(&pos, 0, 10); VecSet(&pos, 1, 10);
  GBLayerSetPos(layer, &pos);
  Facoid* bound = GBLayerGetBoundaryInSurface(layer, surf, false);
  VecFloat2D p = VecFloatCreateStatic2D();
  VecFloat2D u = VecFloatCreateStatic2D();
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&p, 0, 10); VecSet(&p, 1, 10);
  VecSet(&u, 0, 200); VecSet(&u, 1, 0);
  VecSet(&v, 0, 0); VecSet(&v, 1, 100);
  if (VecIsEqual(ShapoidPos(bound), &p) == false ||
    VecIsEqual(ShapoidAxis(bound, 0), &u) == false ||
    VecIsEqual(ShapoidAxis(bound, 1), &v) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerGetBoundaryInSurface failed");
    PBErrCatch(GenBrushErr);
  }
  ShapoidFree(&bound);
  VecSet(&pos, 0, 700); VecSet(&pos, 1, 10);
  GBLayerSetPos(layer, &pos);
  bound = GBLayerGetBoundaryInSurface(layer, surf, false);
  VecSet(&p, 0, 700); VecSet(&p, 1, 10);
  VecSet(&u, 0, 100); VecSet(&u, 1, 0);
  VecSet(&v, 0, 0); VecSet(&v, 1, 100);
  if (VecIsEqual(ShapoidPos(bound), &p) == false ||
    VecIsEqual(ShapoidAxis(bound, 0), &u) == false ||
    VecIsEqual(ShapoidAxis(bound, 1), &v) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerGetBoundaryInSurface failed");
    PBErrCatch(GenBrushErr);
  }
  ShapoidFree(&bound);
  VecSet(&pos, 0, -100); VecSet(&pos, 1, 10);
  GBLayerSetPos(layer, &pos);
  bound = GBLayerGetBoundaryInSurface(layer, surf, false);
  VecSet(&p, 0, 0); VecSet(&p, 1, 10);
  VecSet(&u, 0, 100); VecSet(&u, 1, 0);
  VecSet(&v, 0, 0); VecSet(&v, 1, 100);
  if (VecIsEqual(ShapoidPos(bound), &p) == false ||
    VecIsEqual(ShapoidAxis(bound, 0), &u) == false ||
    VecIsEqual(ShapoidAxis(bound, 1), &v) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBLayerGetBoundaryInSurface failed");
    PBErrCatch(GenBrushErr);
  }
  ShapoidFree(&bound);
  VecSet(&pos, 0, 10); VecSet(&pos, 1, 550);
  GBLayerSetPos(layer, &pos);
  bound = GBLayerGetBoundaryInSurface(layer, surf, false);
  VecSet(&p, 0, 10); VecSet(&p, 1, 550);
  VecSet(&u, 0, 200); VecSet(&u, 1, 0);
  VecSet(&v, 0, 0); VecSet(&v, 1, 50);
  if (VecIsEqual(ShapoidPos(bound), &p) == false ||
```

```
      VecIsEqual(ShapoidAxis(bound, 0), &u) == false ||
      VecIsEqual(ShapoidAxis(bound, 1), &v) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBLayerGetBoundaryInSurface failed");
      PBErrCatch(GenBrushErr);
    }
    ShapoidFree(&bound);
    VecSet(&pos, 0, 10); VecSet(&pos, 1, -50);
    GBLayerSetPos(layer, &pos);
    bound = GBLayerGetBoundaryInSurface(layer, surf, false);
    VecSet(&p, 0, 10); VecSet(&p, 1, 0);
    VecSet(&u, 0, 200); VecSet(&u, 1, 0);
    VecSet(&v, 0, 0); VecSet(&v, 1, 50);
    if (VecIsEqual(ShapoidPos(bound), &p) == false ||
      VecIsEqual(ShapoidAxis(bound, 0), &u) == false ||
      VecIsEqual(ShapoidAxis(bound, 1), &v) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBLayerGetBoundaryInSurface failed");
      PBErrCatch(GenBrushErr);
    }
    ShapoidFree(&bound);
    VecSet(&pos, 0, -300); VecSet(&pos, 1, -150);
    GBLayerSetPos(layer, &pos);
    bound = GBLayerGetBoundaryInSurface(layer, surf, false);
    if (bound != NULL) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBLayerGetBoundaryInSurface failed");
      PBErrCatch(GenBrushErr);
    }
    VecSet(&pos, 0, 1000); VecSet(&pos, 1, 1000);
    GBLayerSetPos(layer, &pos);
    bound = GBLayerGetBoundaryInSurface(layer, surf, false);
    if (bound != NULL) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBLayerGetBoundaryInSurface failed");
      PBErrCatch(GenBrushErr);
    }
    VecSet(&pos, 0, 0); VecSet(&pos, 1, 1000);
    GBLayerSetPos(layer, &pos);
    bound = GBLayerGetBoundaryInSurface(layer, surf, false);
    if (bound != NULL) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBLayerGetBoundaryInSurface failed");
      PBErrCatch(GenBrushErr);
    }
    VecSet(&pos, 0, 1000); VecSet(&pos, 1, 0);
    GBLayerSetPos(layer, &pos);
    bound = GBLayerGetBoundaryInSurface(layer, surf, false);
    if (bound != NULL) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBLayerGetBoundaryInSurface failed");
      PBErrCatch(GenBrushErr);
    }
  GBLayerFree(&layer);
  GBSurfaceFree(&surf);
  printf("UnitTestGBLayerGetBoundaryInSurface OK\n");
}

void UnitTestGBLayer() {
  UnitTestGBLayerCreateFree();
  UnitTestGBLayerArea();
  UnitTestGBLayerGetSet();
```

```
  UnitTestGBLayerCreateFromFile();
  UnitTestGBLayerGetBoundaryInSurface();

  printf("UnitTestGBLayer OK\n");
}

void UnitTestGBPostProcessingCreateFree() {
  GBPostProcessing pp =
    GBPostProcessingCreateStatic(GBPPTypeNormalizeHue);
  if (pp._type != GBPPTypeNormalizeHue) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBPostProcessingCreateStatic failed");
    PBErrCatch(GenBrushErr);
  }
  GBPostProcessing* pp2 =
    GBPostProcessingCreate(GBPPTypeNormalizeHue);
  if (pp2->_type != GBPPTypeNormalizeHue) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBPostProcessingCreate failed");
    PBErrCatch(GenBrushErr);
  }
  GBPostProcessingFree(&pp2);
  if (pp2 != NULL) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBPostProcessingFree failed");
    PBErrCatch(GenBrushErr);
  }
  printf("UnitTestGBPostProcessingCreateFree OK\n");
}

void UnitTestGBPostProcessingGetSet() {
  GBPostProcessing pp =
    GBPostProcessingCreateStatic(GBPPTypeNormalizeHue);
  if (GBPostProcessingGetType(&pp) != GBPPTypeNormalizeHue) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBPostProcessingGetType failed");
    PBErrCatch(GenBrushErr);
  }
  printf("UnitTestGBPostProcessingGetSet OK\n");
}

void UnitTestGBPostProcessing() {
  UnitTestGBPostProcessingCreateFree();
  UnitTestGBPostProcessingGetSet();

  printf("UnitTestGBPostProcessing OK\n");
}

void UnitTestGBSurfaceCreateFree() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 800); VecSet(&dim, 1, 600);
  GBSurface* surf = GBSurfaceCreate(GBSurfaceTypeImage, &dim);
  if (surf->_type != GBSurfaceTypeImage) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceCreate failed");
    PBErrCatch(GenBrushErr);
  }
  if (VecIsEqual(&(surf->_dim), &dim) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceCreate failed");
    PBErrCatch(GenBrushErr);
  }
```

```
    GBPixel white = GBColorWhite;
    if (memcmp(&(surf->_bgColor), &white, sizeof(GBPixel)) != 0) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceCreate failed");
      PBErrCatch(GenBrushErr);
    }
    if (surf->_finalPix == NULL) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceCreate failed");
      PBErrCatch(GenBrushErr);
    }
    if (GSetNbElem(&(surf->_layers)) != 0) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceCreate failed");
      PBErrCatch(GenBrushErr);
    }
    GBSurfaceFree(&surf);
    printf("UnitTestGBSurfaceCreateFree OK\n");
}

void UnitTestGBSurfaceGetSet() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
    GBSurface surf = GBSurfaceCreateStatic(GBSurfaceTypeImage, &dim);
    if (GBSurfaceGetType(&surf) != GBSurfaceTypeImage) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceGetType failed");
      PBErrCatch(GenBrushErr);
    }
    VecShort2D d = GBSurfaceGetDim(&surf);
    if (VecIsEqual(&d, &dim) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceGetDim failed");
      PBErrCatch(GenBrushErr);
    }
    if (VecIsEqual(GBSurfaceDim(&surf), &dim) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceDim failed");
      PBErrCatch(GenBrushErr);
    }
    if (GBSurfaceFinalPixels(&surf) != surf._finalPix) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceFinalPix failed");
      PBErrCatch(GenBrushErr);
    }
    GBPixel black = GBColorBlack;
    GBSurfaceSetBgColor(&surf, &black);
    if (memcmp(&(surf._bgColor), &black, sizeof(GBPixel)) != 0) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceSetBgColor failed");
      PBErrCatch(GenBrushErr);
    }
    if (memcmp(GBSurfaceBgColor(&surf), &black, sizeof(GBPixel)) != 0) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceBgColor failed");
      PBErrCatch(GenBrushErr);
    }
    GBPixel bgCol = GBSurfaceGetBgColor(&surf);
    if (memcmp(&(surf._bgColor), &bgCol, sizeof(GBPixel)) != 0) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceGetBgColor failed");
      PBErrCatch(GenBrushErr);
```

150

```
}
VecShort2D p = VecShortCreateStatic2D();
VecSet(&p, 0, 1); VecSet(&p, 1, 2);
if (GBSurfaceFinalPixel(&surf, &p) != surf._finalPix + 9) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSurfaceFinalPixel failed");
  PBErrCatch(GenBrushErr);
}
if (GBSurfaceFinalPixelSafe(&surf, &p) != surf._finalPix + 9) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSurfaceFinalPixelSafe failed");
  PBErrCatch(GenBrushErr);
}
GBPixel white = GBColorWhite;
GBSurfaceSetFinalPixel(&surf, &p, &white);
if (memcmp(GBSurfaceFinalPixel(&surf, &p), &white,
  sizeof(GBPixel)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSurfaceSetFinalPixel failed");
  PBErrCatch(GenBrushErr);
}
GBPixel pix = GBSurfaceGetFinalPixel(&surf, &p);
if (memcmp(&pix, &white, sizeof(GBPixel)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSurfaceGetFinalPixel failed");
  PBErrCatch(GenBrushErr);
}
if (GBSurfaceIsPosInside(&surf, &p) == false) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSurfaceIsPosInside failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, -1); VecSet(&p, 1, 2);
if (GBSurfaceIsPosInside(&surf, &p) == true) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSurfaceIsPosInside failed");
  PBErrCatch(GenBrushErr);
}
GBPixel transparent = GBColorTransparent;
pix = GBSurfaceGetFinalPixelSafe(&surf, &p);
if (memcmp(&pix, &transparent, sizeof(GBPixel)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSurfaceGetFinalPixelSafe failed");
  PBErrCatch(GenBrushErr);
}
if (GBSurfaceFinalPixelSafe(&surf, &p) != NULL) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSurfaceFinalPixelSafe failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, 1); VecSet(&p, 1, 10);
if (GBSurfaceIsPosInside(&surf, &p) == true) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSurfaceIsPosInside failed");
  PBErrCatch(GenBrushErr);
}
GBSurfaceSetFinalPixelSafe(&surf, &p, &white);
VecSet(&p, 0, 2); VecSet(&p, 1, 1);
GBPixel blue = GBColorBlue;
GBSurfaceSetFinalPixelSafe(&surf, &p, &blue);
if (memcmp(GBSurfaceFinalPixel(&surf, &p), &blue,
  sizeof(GBPixel)) != 0) {
```

```
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceSetFinalPixelSafe failed");
    PBErrCatch(GenBrushErr);
  }
  if (GBSurfaceLayers(&surf) != &(surf._layers)) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceLayers failed");
    PBErrCatch(GenBrushErr);
  }
  GBSurfaceFreeStatic(&surf);
  printf("UnitTestGBSurfaceGetSet OK\n");
}

void UnitTestGBSurfaceArea() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
  GBSurface surf = GBSurfaceCreateStatic(GBSurfaceTypeImage, &dim);
  if (GBSurfaceArea(&surf) != 12) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceArea failed");
    PBErrCatch(GenBrushErr);
  }
  GBSurfaceFreeStatic(&surf);
  printf("UnitTestGBSurfaceArea OK\n");
}

void UnitTestGBSurfaceClone() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
  GBSurface surf = GBSurfaceCreateStatic(GBSurfaceTypeImage, &dim);
  GBSurface clone = GBSurfaceClone(&surf);
  if (GBSurfaceGetType(&surf) != GBSurfaceGetType(&clone)) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceClone failed");
    PBErrCatch(GenBrushErr);
  }
  if (VecIsEqual(GBSurfaceDim(&surf), GBSurfaceDim(&clone)) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceClone failed");
    PBErrCatch(GenBrushErr);
  }
  if (memcmp(&(surf._bgColor), &(clone._bgColor),
    sizeof(GBPixel)) != 0) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceClone failed");
    PBErrCatch(GenBrushErr);
  }
  if (surf._finalPix == clone._finalPix) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceClone failed");
    PBErrCatch(GenBrushErr);
  }
  if (memcmp(surf._finalPix, clone._finalPix,
    sizeof(GBPixel) * GBSurfaceArea(&surf)) != 0) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceClone failed");
    PBErrCatch(GenBrushErr);
  }
  GBSurfaceFreeStatic(&surf);
  GBSurfaceFreeStatic(&clone);
  printf("UnitTestGBSurfaceClone OK\n");
}
```

```
void UnitTestGBSurfaceAddRemoveLayer() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
  GBSurface surf = GBSurfaceCreateStatic(GBSurfaceTypeImage, &dim);
  GBLayer* layerA = GBSurfaceAddLayer(&surf, &dim);
  if (layerA == NULL) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceAddLayer failed");
    PBErrCatch(GenBrushErr);
  }
  if (GSetNbElem(GBSurfaceLayers(&surf)) != 1) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceAddLayer failed");
    PBErrCatch(GenBrushErr);
  }
  if (VecIsEqual(GBLayerDim(layerA), &dim) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceAddLayer failed");
    PBErrCatch(GenBrushErr);
  }
  GBLayer* layerB = GBSurfaceAddLayer(&surf, &dim);
  GBSurfaceRemoveLayer(&surf, layerA);
  if ((GBLayer*)(surf._layers._head->_data) != layerB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceRemoveLayer failed");
    PBErrCatch(GenBrushErr);
  }
  GBSurfaceFreeStatic(&surf);
  printf("UnitTestGBSurfaceAddRemoveLayer OK\n");
}

void UnitTestGBSurfaceGetLayerNbLayer() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
  GBSurface surf = GBSurfaceCreateStatic(GBSurfaceTypeImage, &dim);
  GBLayer* layers[3] = {NULL};
  for (int iLayer = 0; iLayer< 3; ++iLayer)
    layers[iLayer] = GBSurfaceAddLayer(&surf, &dim);
  if (GBSurfaceNbLayer(&surf) != 3) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceNbLayer failed");
    PBErrCatch(GenBrushErr);
  }
  for (int iLayer = 0; iLayer< 3; ++iLayer) {
    if (GBSurfaceGetLayer(&surf, iLayer) != layers[iLayer]) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceGetLayer failed");
      PBErrCatch(GenBrushErr);
    }
  }
  GBSurfaceFreeStatic(&surf);
  printf("UnitTestGBSurfaceGetLayer OK\n");
}

void UnitTestGBSurfaceSetLayersModified() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
  GBSurface surf = GBSurfaceCreateStatic(GBSurfaceTypeImage, &dim);
  GBLayer* layers[3] = {NULL};
  for (int iLayer = 0; iLayer< 3; ++iLayer)
    layers[iLayer] = GBSurfaceAddLayer(&surf, &dim);
```

153

```
    GBSurfaceSetLayersModified(&surf, false);
    for (int iLayer = 0; iLayer< 3; ++iLayer) {
      if (GBLayerIsModified(layers[iLayer]) != false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBSurfaceSetLayersModified failed");
        PBErrCatch(GenBrushErr);
      }
    }
    GBSurfaceSetLayersModified(&surf, true);
    for (int iLayer = 0; iLayer< 3; ++iLayer) {
      if (GBLayerIsModified(layers[iLayer]) != true) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBSurfaceSetLayersModified failed");
        PBErrCatch(GenBrushErr);
      }
    }
    GBSurfaceFreeStatic(&surf);
    printf("UnitTestGBSurfaceSetLayersModified OK\n");
}

void UnitTestGBSurfaceSetLayerPos() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
    GBSurface surf = GBSurfaceCreateStatic(GBSurfaceTypeImage, &dim);
    GBLayer* layers[3] = {NULL};
    for (int iLayer = 0; iLayer< 3; ++iLayer)
      layers[iLayer] = GBSurfaceAddLayer(&surf, &dim);
    GBSurfaceSetLayerPos(&surf, layers[2], 0);
    GBSurfaceSetLayerPos(&surf, layers[0], 2);
    if (GBSurfaceGetLayer(&surf, 0) != layers[2] ||
      GBSurfaceGetLayer(&surf, 1) != layers[1] ||
      GBSurfaceGetLayer(&surf, 2) != layers[0] ||
      GBLayerIsModified(layers[0]) != true ||
      GBLayerIsModified(layers[1]) != true ||
      GBLayerIsModified(layers[2]) != true) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceSetLayerPos failed");
      PBErrCatch(GenBrushErr);
    }
    GBSurfaceFreeStatic(&surf);
    printf("UnitTestGBSurfaceSetLayerPos OK\n");
}

void UnitTestGBSurfaceGetModifiedArea() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 100); VecSet(&dim, 1, 100);
    GBSurface surf = GBSurfaceCreateStatic(GBSurfaceTypeImage, &dim);
    GBLayer* layers[4] = {NULL};
    VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
    for (int iLayer = 0; iLayer < 4; ++iLayer)
      layers[iLayer] = GBSurfaceAddLayer(&surf, &dim);
    VecShort2D pos = VecShortCreateStatic2D();
    VecSet(&pos, 0, -2); VecSet(&pos, 1, 5);
    GBLayerSetPos(layers[0], &pos);
    VecSet(&pos, 0, 5); VecSet(&pos, 1, 5);
    GBLayerSetPos(layers[1], &pos);
    VecSet(&pos, 0, 10); VecSet(&pos, 1, 8);
    GBLayerSetPos(layers[2], &pos);
    VecSet(&pos, 0, 20); VecSet(&pos, 1, 20);
    GBLayerSetPos(layers[3], &pos);
    GBLayerSetModified(layers[3], false);
    GSetShapoid* set = GBSurfaceGetModifiedArea(&surf);
```

```
      if (GSetNbElem(set) != 5) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg,
          "GBSurfaceGetModifiedArea failed");
        PBErrCatch(ShapoidErr);
      }
      int iCheck = 0;
      float checkp[10] = {
          0.0, 5.0, 0.0, 0.0, 8.0, 5.0, 10.0, 15.0, 15.0, 8.0};
      float checku[5] = {8.0, 10.0, 7.0, 10.0, 5.0};
      float checkv[5] = {10.0, 5.0, 10.0, 3.0, 7.0};
      VecFloat2D p = VecFloatCreateStatic2D();
      VecFloat2D u = VecFloatCreateStatic2D();
      VecFloat2D v = VecFloatCreateStatic2D();
      do {
        Facoid* fac = (Facoid*)GSetPop(set);
        VecSet(&p, 0, checkp[2 * iCheck]);
        VecSet(&p, 1, checkp[2 * iCheck + 1]);
        VecSet(&u, 0, checku[iCheck]); VecSet(&v, 1, checkv[iCheck]);
        if (VecIsEqual(ShapoidPos(fac), &p) == false ||
          VecIsEqual(ShapoidAxis(fac, 0), &u) == false ||
          VecIsEqual(ShapoidAxis(fac, 1), &v) == false) {
          ShapoidErr->_type = PBErrTypeUnitTestFailed;
          sprintf(ShapoidErr->_msg,
            "GBSurfaceGetModifiedArea failed");
          PBErrCatch(ShapoidErr);
        }
        ShapoidFree(&fac);
        ++iCheck;
      } while(GSetNbElem(set) > 0);
      GSetFree(&set);
      GBSurfaceFreeStatic(&surf);
      printf("UnitTestGBSurfaceGetModifiedArea OK\n");
}

void UnitTestGBSurfaceUpdate() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
  GBSurface* surf = GBSurfaceCreate(GBSurfaceTypeImage, &dim);
  GBLayer* layers[4] = {NULL};
  VecSet(&dim, 0, 5); VecSet(&dim, 1, 5);
  for (int iLayer = 0; iLayer < 4; ++iLayer) {
    layers[iLayer] = GBSurfaceAddLayer(surf, &dim);
    GBLayerSetBlendMode(layers[iLayer], GBLayerBlendModeOver);
  }
  GBLayerSetStackPos(layers[1], GBLayerStackPosInside);
  GBLayerSetStackPos(layers[2], GBLayerStackPosInside);
  GBLayerSetStackPos(layers[3], GBLayerStackPosFg);
  VecShort2D pos = VecShortCreateStatic2D();
  VecSet(&pos, 0, 1); VecSet(&pos, 1, 1);
  GBLayerSetPos(layers[0], &pos);
  VecSet(&pos, 0, 3); VecSet(&pos, 1, 3);
  GBLayerSetPos(layers[1], &pos);
  VecSet(&pos, 0, 4); VecSet(&pos, 1, 4);
  GBLayerSetPos(layers[2], &pos);
  VecSet(&pos, 0, 7); VecSet(&pos, 1, 7);
  GBLayerSetPos(layers[3], &pos);
  VecSetNull(&pos);
  GBPixel col;
  float depth;
  do {
    col = GBColorGreen;
```

155

```
      depth = 0.0;
      GBLayerAddPixel(layers[0], &pos, &col, depth);
      col._rgba[GBPixelRed] = VecGet(&pos, 1) * 30;
      col._rgba[GBPixelGreen] = VecGet(&pos, 0) * 30;
      col._rgba[GBPixelBlue] = 0;
      col._rgba[GBPixelAlpha] = 255 - VecGet(&pos, 1) * 30;
      depth = VecGet(&pos, 0);
      GBLayerAddPixel(layers[1], &pos, &col, depth);
      col._rgba[GBPixelRed] = 0;
      col._rgba[GBPixelGreen] = VecGet(&pos, 0) * 30;
      col._rgba[GBPixelBlue] = VecGet(&pos, 1) * 30;
      col._rgba[GBPixelAlpha] = 255 - VecGet(&pos, 0) * 30;
      depth = VecGet(&pos, 1);
      GBLayerAddPixel(layers[2], &pos, &col, depth);
      col = GBColorBlue;
      depth = 0.0;
      GBLayerAddPixel(layers[3], &pos, &col, depth);
    } while (VecStep(&pos, &dim));
    GBSurfaceUpdate(surf);
    unsigned char check[400] = {
      255,255,255,255, 255,255,255,255, 255,255,255,
      255, 255,255,255,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 255,255,255,255, 0,255,0,255, 0,255,0,255, 0,255,0,255,
      0,255,0,255, 0,255,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      255,255,255,255, 0,255,0,255, 0,255,0,255, 0,255,0,255,
      0,255,0,255, 0,255,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      255,255,255,255, 0,255,0,255, 0,255,0,255, 0,0,0,255, 0,30,0,255,
      0,60,0,255, 0,90,0,255, 0,120,0,255, 0,0,0,0, 0,0,0,0,
      255,255,255,255, 0,255,0,255, 0,255,0,255, 26,30,0,255,
      0,0,0,255, 4,34,0,255, 7,67,0,255, 11,101,0,255, 120,184,120,255,
      0,0,0,0, 0,0,0,0, 0,255,0,255, 0,255,0,255, 46,60,0,255,
      0,0,30,255, 7,34,26,255, 14,67,23,255, 21,101,19,255,
      120,184,136,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      148,90,90,255, 0,0,60,255, 11,34,53,255, 21,67,46,255,
      32,101,39,255, 120,184,152,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 184,120,120,255, 0,0,90,255, 14,34,79,255, 28,67,69,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,120,255, 30,56,136,255, 60,106,152,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255
    };
    for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
      if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
        check[iPix * 4] ||
        surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
        check[iPix * 4 + 1] ||
        surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
        check[iPix * 4 + 2] ||
        surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
        check[iPix * 4 + 3]) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "GBSurfaceUpdate failed");
        //PBErrCatch(ShapoidErr);
      }
    }
    GBSurfaceFree(&surf);
    printf("UnitTestGBSurfaceUpdate OK\n");
}

void UnitTestGBSurfaceAddLayerFromFile() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
```

```
    GBSurface* surf = GBSurfaceCreate(GBSurfaceTypeImage, &dim);
    GBLayer* layer = GBSurfaceAddLayerFromFile(surf, "./ImageRef.tga");
    (void)layer;
    GBPixel transparent = GBColorTransparent;
    GBSurfaceSetBgColor(surf, &transparent);
    GBSurfaceUpdate(surf);
    unsigned char check[400] = {
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,255,0,255, 0,255,0,255,
      0,255,0,255, 0,255,0,255, 0,255,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,255,0,255, 0,255,0,255, 0,255,0,255,
      0,255,0,255, 0,255,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,255,0,255, 0,255,0,255, 0,0,0,255, 0,30,0,255,
      0,60,0,255, 0,90,0,255, 0,120,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,255,0,255, 0,255,0,255, 26,30,0,255, 0,0,0,255, 4,34,0,255,
      7,67,0,255, 11,101,0,255, 120,184,120,255, 0,0,0,0, 0,0,0,0,
      0,255,0,255, 0,255,0,255, 46,60,0,255, 0,0,30,255, 7,34,26,255,
      14,67,23,255, 21,101,19,255, 120,184,136,255, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 148,90,90,255, 0,0,60,255, 11,34,53,255,
      21,67,46,255, 32,101,39,255, 120,184,152,255, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 184,120,120,255, 0,0,90,255, 14,34,79,255,
      28,67,69,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,120,255, 30,56,136,255,
      60,106,152,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,255,255, 0,0,255,255, 0,0,255,255
    };
    for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
      if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
        check[iPix * 4] ||
        surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
        check[iPix * 4 + 1] ||
        surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
        check[iPix * 4 + 2] ||
        surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
        check[iPix * 4 + 3]) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "GBSurfaceAddLayerFromFile failed");
        PBErrCatch(ShapoidErr);
      }
    }
    GBSurfaceFree(&surf);
    printf("UnitTestGBSurfaceAddLayerFromFile OK\n");
}

void UnitTestGBSurfaceFlush() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
    GBSurface* surf = GBSurfaceCreate(GBSurfaceTypeImage, &dim);
    GBLayer* layer = GBSurfaceAddLayer(surf, &dim);
    GBLayerSetModified(layer, false);
    GBPixel blue = GBColorBlue;
    GBSurfaceSetBgColor(surf, &blue);
    GBSurfaceFlush(surf);
    if (GBLayerIsModified(layer) == false) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBSurfaceFlush failed");
      PBErrCatch(ShapoidErr);
    }
    for (int iPix = GBSurfaceArea(surf); iPix--;)
      if (memcmp(surf->_finalPix + iPix, &blue, sizeof(GBPixel)) != 0) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
```

```
        sprintf(ShapoidErr->_msg, "GBSurfaceFlush failed");
        PBErrCatch(ShapoidErr);
    }
  GBSurfaceFree(&surf);
  printf("UnitTestGBSurfaceFlush OK\n");
}

void UnitTestGBSurfaceIsSameAs() {
  char* fileName = "./ImageRef.tga";
  GBSurface* surfA = (GBSurface*)GBSurfaceImageCreateFromFile(fileName);
  GBSurface* surfB = (GBSurface*)GBSurfaceImageCreateFromFile(fileName);
  if (GBSurfaceIsSameAs(surfA, surfB) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBSurfaceIsSameAs failed");
    PBErrCatch(ShapoidErr);
  }
  VecSet(GBSurfaceDim(surfA), 0, 0);
  if (GBSurfaceIsSameAs(surfA, surfB) == true) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBSurfaceIsSameAs failed");
    PBErrCatch(ShapoidErr);
  }
  VecSet(GBSurfaceDim(surfA), 0, 10);
  GBSurfaceFinalPixels(surfA)->_rgba[GBPixelAlpha] = 255;
  if (GBSurfaceIsSameAs(surfA, surfB) == true) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBSurfaceIsSameAs failed");
    PBErrCatch(ShapoidErr);
  }
  GBSurfaceFree(&surfA);
  GBSurfaceFree(&surfB);
  printf("UnitTestGBSurfaceIsSameAs OK\n");
}

void UnitTestGBSurfaceNormalizeHue() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 3); VecSet(&dim, 1, 3);
  GBSurface* surf = GBSurfaceCreate(GBSurfaceTypeImage, &dim);
  unsigned char in[27] = {
    20,20,20, 10,20,30, 60,50,40,
    30,30,30, 20,30,40, 50,40,30,
    40,40,40, 30,40,50, 40,30,20
    };
  VecShort2D p = VecShortCreateStatic2D();
  int iPix = 0;
  do {
    GBPixel* pix = GBSurfaceFinalPixel(surf, &p);
    pix->_rgba[GBPixelRed] = in[iPix * 3];
    pix->_rgba[GBPixelGreen] = in[iPix * 3 + 1];
    pix->_rgba[GBPixelBlue] = in[iPix * 3 + 2];
    ++iPix;
  } while (VecStep(&p, GBSurfaceDim(surf)));
  GBPostProcessing post =
    GBPostProcessingCreateStatic(GBPPTypeNormalizeHue);
  GBSurfacePostProcess(surf, &post);
  VecSetNull(&p);
  iPix = 0;
  unsigned char out[27] = {
    51,51,51, 0,51,102, 255,204,153,
    102,102,102, 51,102,153, 204,153,102,
    153,153,153, 102,153,204, 153,102,51
    };
```

```
    do {
      GBPixel* pix = GBSurfaceFinalPixel(surf, &p);
      if (pix->_rgba[GBPixelRed] != out[iPix * 3] ||
        pix->_rgba[GBPixelGreen] != out[iPix * 3 + 1] ||
        pix->_rgba[GBPixelBlue] != out[iPix * 3 + 2]) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "UnitTestGBSurfaceNormalizeHue failed");
        PBErrCatch(ShapoidErr);
      }
      ++iPix;
    } while (VecStep(&p, GBSurfaceDim(surf)));
    GBSurfaceFree(&surf);
    GBSurfaceImage* img = GBSurfaceImageCreateFromFile(
      "./GBSurfaceNormalizeHueTest.tga");
    GBSurfacePostProcess((GBSurface*)img, &post);
    GBSurfaceImage* ref = GBSurfaceImageCreateFromFile(
      "./GBSurfaceNormalizeHueRef.tga");
    if (GBSurfaceIsSameAs((GBSurface*)img, (GBSurface*)ref) == false) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "UnitTestGBSurfaceNormalizeHue failed");
      PBErrCatch(ShapoidErr);
    }
    GBSurfaceFree(&surf);
    GBSurfaceImageFree(&img);
    GBSurfaceImageFree(&ref);
    printf("UnitTestGBSurfaceNormalizeHue OK\n");
}

void UnitTestGBSurface() {
  UnitTestGBSurfaceCreateFree();
  UnitTestGBSurfaceGetSet();
  UnitTestGBSurfaceArea();
  UnitTestGBSurfaceClone();
  UnitTestGBSurfaceAddRemoveLayer();
  UnitTestGBSurfaceGetLayerNbLayer();
  UnitTestGBSurfaceSetLayersModified();
  UnitTestGBSurfaceSetLayerPos();
  UnitTestGBSurfaceGetModifiedArea();
  UnitTestGBSurfaceUpdate();
  UnitTestGBSurfaceAddLayerFromFile();
  UnitTestGBSurfaceFlush();
  UnitTestGBSurfaceIsSameAs();
  UnitTestGBSurfaceNormalizeHue();
  printf("UnitTestGBSurface OK\n");
}

void UnitTestGBSurfaceImageCreateFree() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 800); VecSet(&dim, 1, 600);
  GBSurfaceImage* surf = GBSurfaceImageCreate(&dim);
  if (GBSurfaceGetType((GBSurface*)surf) != GBSurfaceTypeImage) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceImageCreate failed");
    PBErrCatch(GenBrushErr);
  }
  if (VecIsEqual(GBSurfaceDim((GBSurface*)surf), &dim) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceImageCreate failed");
    PBErrCatch(GenBrushErr);
  }
  if (surf->_fileName != NULL) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
```

```
      sprintf(GenBrushErr->_msg, "GBSurfaceImageCreate failed");
      PBErrCatch(GenBrushErr);
    }
    GBSurfaceImageFree(&surf);
    printf("UnitTestGBSurfaceImageCreateFree OK\n");
}

void UnitTestGBSurfaceImageGetSet() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 800); VecSet(&dim, 1, 600);
    GBSurfaceImage* surf = GBSurfaceImageCreate(&dim);
    if (GBSurfaceImageFileName(surf) != surf->_fileName) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceImageFileName failed");
      PBErrCatch(GenBrushErr);
    }
    char* fileName = "./";
    GBSurfaceImageSetFileName(surf, fileName);
    if (strcmp(surf->_fileName, fileName) != 0) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceImageSetFileName failed");
      PBErrCatch(GenBrushErr);
    }
    GBSurfaceImageFree(&surf);
    printf("UnitTestGBSurfaceImageGetSet OK\n");
}

void UnitTestGBSurfaceImageClone() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 800); VecSet(&dim, 1, 600);
    GBSurfaceImage* surf = GBSurfaceImageCreate(&dim);
    char* fileName = "./";
    GBSurfaceImageSetFileName(surf, fileName);
    GBSurfaceImage* clone = GBSurfaceImageClone(surf);
    if (GBSurfaceGetType((GBSurface*)clone) != GBSurfaceTypeImage) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceImageCreate failed");
      PBErrCatch(GenBrushErr);
    }
    if (VecIsEqual(GBSurfaceDim((GBSurface*)clone), &dim) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceImageCreate failed");
      PBErrCatch(GenBrushErr);
    }
    if (strcmp(clone->_fileName, fileName) != 0) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceImageClone failed");
      PBErrCatch(GenBrushErr);
    }
    GBSurfaceImageFree(&surf);
    GBSurfaceImageFree(&clone);
    printf("UnitTestGBSurfaceImageClone OK\n");
}

void UnitTestGBSurfaceImageSave() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
    GBSurfaceImage* surfImage = GBSurfaceImageCreate(&dim);
    GBSurface* surf = (GBSurface*)surfImage;

GBPixel t = GBColorTransparent;
GBSurfaceSetBgColor(surf, &t);
```

160

```
  GBSurfaceFlush(surf);

  GBLayer* layers[4] = {NULL};
  VecSet(&dim, 0, 5); VecSet(&dim, 1, 5);
  for (int iLayer = 0; iLayer < 4; ++iLayer) {
    layers[iLayer] = GBSurfaceAddLayer(surf, &dim);
    GBLayerSetBlendMode(layers[iLayer], GBLayerBlendModeOver);
  }
  GBLayerSetStackPos(layers[0], GBLayerStackPosBg);
  GBLayerSetStackPos(layers[1], GBLayerStackPosInside);
  GBLayerSetStackPos(layers[2], GBLayerStackPosInside);
  GBLayerSetStackPos(layers[3], GBLayerStackPosFg);
  VecShort2D pos = VecShortCreateStatic2D();
  VecSet(&pos, 0, 1); VecSet(&pos, 1, 1);
  GBLayerSetPos(layers[0], &pos);
  VecSet(&pos, 0, 3); VecSet(&pos, 1, 3);
  GBLayerSetPos(layers[1], &pos);
  VecSet(&pos, 0, 4); VecSet(&pos, 1, 4);
  GBLayerSetPos(layers[2], &pos);
  VecSet(&pos, 0, 7); VecSet(&pos, 1, 7);
  GBLayerSetPos(layers[3], &pos);
  VecSetNull(&pos);
  GBPixel col;
  float depth;
  do {
    col = GBColorGreen;
    depth = 0.0;
    GBLayerAddPixel(layers[0], &pos, &col, depth);
    col._rgba[GBPixelRed] = VecGet(&pos, 1) * 30;
    col._rgba[GBPixelGreen] = VecGet(&pos, 0) * 30;
    col._rgba[GBPixelBlue] = 0;
    col._rgba[GBPixelAlpha] = 255 - VecGet(&pos, 1) * 30;
    depth = VecGet(&pos, 0);
    GBLayerAddPixel(layers[1], &pos, &col, depth);
    col._rgba[GBPixelRed] = 0;
    col._rgba[GBPixelGreen] = VecGet(&pos, 0) * 30;
    col._rgba[GBPixelBlue] = VecGet(&pos, 1) * 30;
    col._rgba[GBPixelAlpha] = 255 - VecGet(&pos, 0) * 30;
    depth = VecGet(&pos, 1);
    GBLayerAddPixel(layers[2], &pos, &col, depth);
    col = GBColorBlue;
    depth = 0.0;
    GBLayerAddPixel(layers[3], &pos, &col, depth);
  } while (VecStep(&pos, &dim));
  GBSurfaceUpdate(surf);
  GBSurfaceImageSetFileName(surfImage, "./GBSurfaceImageSave.tga");
  if (GBSurfaceImageSave(surfImage) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceImageSave failed");
    PBErrCatch(GenBrushErr);
  }
  GBSurfaceImageFree(&surfImage);
  printf("UnitTestGBSurfaceImageSave OK\n");
}

void UnitTestGBSurfaceImageCreateFromFile() {
  char* fileName = "./ImageRef.tga";
  GBSurfaceImage* surfImage = GBSurfaceImageCreateFromFile(fileName);
  if (surfImage == NULL) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceImageCreateFromFile failed");
    PBErrCatch(GenBrushErr);
```

161

```
    }
  GBSurface* surf = (GBSurface*)surfImage;
  unsigned char check[400] = {
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,255,0,255, 0,255,0,255,
    0,255,0,255, 0,255,0,255, 0,255,0,255, 0,255,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,255,0,255, 0,255,0,255, 0,255,0,255,
    0,255,0,255, 0,255,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,255,0,255, 0,255,0,255, 0,0,0,255, 0,30,0,255,
    0,60,0,255, 0,90,0,255, 0,120,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,255,0,255, 0,255,0,255, 26,30,0,255, 0,0,0,255, 4,34,0,255,
    7,67,0,255, 11,101,0,255, 120,184,120,255, 0,0,0,0, 0,0,0,0,
    0,255,0,255, 0,255,0,255, 46,60,0,255, 0,0,30,255, 7,34,26,255,
    14,67,23,255, 21,101,19,255, 120,184,136,255, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 148,90,90,255, 0,0,60,255, 11,34,53,255,
    21,67,46,255, 32,101,39,255, 120,184,152,255, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 184,120,120,255, 0,0,90,255, 14,34,79,255,
    28,67,69,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,120,255, 30,56,136,255,
    60,106,152,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,255,255, 0,0,255,255, 0,0,255,255
    };
  for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
    if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
      check[iPix * 4] ||
      surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
      check[iPix * 4 + 1] ||
      surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
      check[iPix * 4 + 2] ||
      surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
      check[iPix * 4 + 3]) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBSurfaceCreateFromFile failed");
      PBErrCatch(ShapoidErr);
    }
  }
  GBSurfaceImageFree(&surfImage);
  printf("UnitTestGBSurfaceImageCreateFromFile OK\n");
}

void UnitTestGBSurfaceImage() {
  UnitTestGBSurfaceImageCreateFree();
  UnitTestGBSurfaceImageGetSet();
  UnitTestGBSurfaceImageClone();
  UnitTestGBSurfaceImageSave();
  UnitTestGBSurfaceImageCreateFromFile();
  printf("UnitTestGBSurfaceImage OK\n");
}

void UnitTestGBEyeOrthoCreateFree() {
  GBEyeOrthoView view = GBEyeOrthoViewFront;
  GBEyeOrtho* eye = GBEyeOrthoCreate(view);
  if (eye->_view != GBEyeOrthoViewFront) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBEyeOrthoCreate failed");
    PBErrCatch(GenBrushErr);
  }
  float check[9] = {1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0};
  VecShort2D index = VecShortCreateStatic2D();
  VecShort2D dim = MatGetDim(GBEyeProj(eye));
  int iCheck = 0;
```

162

```
    do {
      if (ISEQUALF(MatGet(GBEyeProj(eye), &index),
        check[iCheck]) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeOrthoCreate failed");
        PBErrCatch(GenBrushErr);
      }
      ++iCheck;
    } while (VecStep(&index, &dim));
    GBEyeFree(&eye);
    if (eye != NULL) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBEyeOrthoFree failed");
      PBErrCatch(GenBrushErr);
    }
    printf("UnitTestGBEyeOrthoCreateFree OK\n");
}

void UnitTestGBEyeOrthoGetSet() {
  GBEyeOrthoView view = GBEyeOrthoViewFront;
  GBEyeOrtho* eye = GBEyeOrthoCreate(view);
  VecShort2D index = VecShortCreateStatic2D();
  VecFloat3D scale = VecFloatCreateStatic3D();
  VecSet(&scale, 0, 2.0); VecSet(&scale, 1, 3.0);
  VecSet(&scale, 2, 1.0);
  GBEyeSetScale(eye, &scale);
  VecFloat2D orig = VecFloatCreateStatic2D();
  VecSet(&orig, 0, 4.0); VecSet(&orig, 1, 5.0);
  GBEyeSetOrig(eye, &orig);
  float theta = PBMATH_QUARTERPI;
  GBEyeSetRot(eye, theta);
  VecShort2D dim = MatGetDim(GBEyeProj(eye));
  float check[6][9] = {
    {1.414214, 1.414214, 0.0, -2.121320, 2.121320, 0.0, 0.0,
      0.0, 1.0},
    {-1.414214, -1.414214, 0.0, -2.121320, 2.121320, 0.0, 0.0,
      0.0, -1.0},
    {1.414214, 1.414214, 0.0, 0.0, 0.0, -1.0, -2.121320, 2.121320,
      0.0},
    {1.414214, 1.414214, 0.0, 0.0, 0.0, 1.0, 2.121320, -2.121320,
      0.0},
    {0.0, 0.0, 1.0, -2.121320, 2.121320, 0.0, -1.414214, -1.414214,
      0.0},
    {0.0, 0.0, -1.0, -2.121320, 2.121320, 0.0, 1.414214, 1.414214,
      0.0}
    };
  for (int iView = 6; iView--;) {
    int iCheck = 0;
    VecSetNull(&index);
    GBEyeOrthoSetView(eye, (GBEyeOrthoView)iView);
    do {
      if (ISEQUALF(MatGet(GBEyeProj(eye), &index),
        check[iView][iCheck]) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeOrthoSetView failed");
        PBErrCatch(GenBrushErr);
      }
      ++iCheck;
    } while (VecStep(&index, &dim));
  }
  GBEyeFree(&eye);
  printf("UnitTestGBEyeOrthoGetSet OK\n");
```

```
}

void UnitTestGBEyeOrthoProcessPoint() {
  VecFloat2D v2d = VecFloatCreateStatic2D();
  VecSet(&v2d, 0, 0.0); VecSet(&v2d, 1, 1.0);
  VecFloat3D v3d = VecFloatCreateStatic3D();
  VecSet(&v3d, 0, 1.0); VecSet(&v3d, 1, 0.0); VecSet(&v3d, 2, 1.0);
  VecFloat* v4d = VecFloatCreate(4);
  VecSet(v4d, 0, 1.0); VecSet(v4d, 1, 2.0);
  VecSet(v4d, 2, 3.0); VecSet(v4d, 3, 4.0);
  float check2d[6][2] = {
    {1.0,5.0},
    {1.0,5.0},
    {4.0,5.0},
    {4.0,5.0},
    {1.0,5.0},
    {1.0,5.0}};
  float check3d[6][3] = {
    {4.0,7.0,1.0},
    {4.0,3.0,-1.0},
    {1.0,7.0,0.0},
    {7.0,7.0,0.0},
    {4.0,3.0,1.0},
    {4.0,7.0,-1.0}};
  float check4d[6][4] = {
    {-2.0,7.0,3.0,4.0},
    {-2.0,3.0,-3.0,4.0},
    {-5.0,7.0,-2.0,4.0},
    {13.0,7.0,2.0,4.0},
    {-2.0,-1.0,1.0,4.0},
    {-2.0,11.0,-1.0,4.0}};
  GBHandDefault* hand = GBHandDefaultCreate();
  for (int iView = 6; iView--;) {
    GBEyeOrthoView view = (GBEyeOrthoView)iView;
    GBEyeOrtho* eye = GBEyeOrthoCreate(view);
    VecFloat3D scale = VecFloatCreateStatic3D();
    VecSet(&scale, 0, 2.0); VecSet(&scale, 1, 3.0);
    VecSet(&scale, 2, 1.0);
    GBEyeSetScale(eye, &scale);
    VecFloat2D orig = VecFloatCreateStatic2D();
    VecSet(&orig, 0, 4.0); VecSet(&orig, 1, 5.0);
    GBEyeSetOrig(eye, &orig);
    float theta = PBMATH_HALFPI;
    GBEyeSetRot(eye, theta);
    GBLayer layer;
    GBInk ink;
    GBTool tool;
    GBObjPod* pod = GBObjPodCreatePoint((VecFloat*)&v2d, eye,
      hand, &tool, &ink, &layer);
    VecFloat* proj = GBObjPodGetEyeObjAsPoint(pod);
    if (VecGetDim(proj) != VecGetDim(pod->_srcPoint)) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBEyeOrthoProcess failed");
      PBErrCatch(GenBrushErr);
    }
    for (int i = VecGetDim(proj); i--;) {
      if (ISEQUALF(VecGet(proj, i), check2d[iView][i]) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeOrthoProcess failed");
        PBErrCatch(GenBrushErr);
      }
    }
```

164

```
      pod->_srcPoint = (VecFloat*)&v3d;
      GBEyeProcess((GBEye*)eye, pod);
      proj = GBObjPodGetEyeObjAsPoint(pod);
      if (VecGetDim(proj) != 3) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeOrthoProcess failed");
        PBErrCatch(GenBrushErr);
      }
      for (int i = VecGetDim(proj); i--;) {
        if (ISEQUALF(VecGet(proj, i), check3d[iView][i]) == false) {
          GenBrushErr->_type = PBErrTypeUnitTestFailed;
          sprintf(GenBrushErr->_msg, "GBEyeOrthoProcess failed");
          PBErrCatch(GenBrushErr);
        }
      }
      pod->_srcPoint = v4d;
      GBEyeProcess((GBEye*)eye, pod);
      proj = GBObjPodGetEyeObjAsPoint(pod);
      if (VecGetDim(proj) != 4) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeOrthoProcess failed");
        PBErrCatch(GenBrushErr);
      }
      for (int i = VecGetDim(proj); i--;) {
        if (ISEQUALF(VecGet(proj, i), check4d[iView][i]) == false) {
          GenBrushErr->_type = PBErrTypeUnitTestFailed;
          sprintf(GenBrushErr->_msg, "GBEyeOrthoProcess failed");
          PBErrCatch(GenBrushErr);
        }
      }
      GBEyeFree(&eye);
      GBObjPodFree(&pod);
    }
    GBHandDefaultFree(&hand);
    VecFree(&v4d);
    printf("UnitTestGBEyeOrthoProcessPoint OK\n");
}

void UnitTestGBEyeOrthoProcessCurve() {
  SCurve* curve = SCurveCreate(1, 2, 2);
  for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    VecSet(SCurveCtrl(curve, iCtrl), 0, (float)iCtrl);
    VecSet(SCurveCtrl(curve, iCtrl), 1, (float)iCtrl * 2.0);
  }
  GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
  float theta = PBMATH_HALFPI;
  GBEyeSetRot((GBEye*)eye, theta);
  SCurve* projCurve = GBEyeGetProjectedCurve((GBEye*)eye, curve);
  if (SCurveGetDim(projCurve) != SCurveGetDim(curve)) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBEyeOrthoGetProjectedCurve failed");
    PBErrCatch(GenBrushErr);
  }
  for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    if (ISEQUALF(VecGet(SCurveCtrl(projCurve, iCtrl), 0),
      (float)iCtrl * -2.0) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBEyeOrthoGetProjectedCurve failed");
      PBErrCatch(GenBrushErr);
    }
    if (ISEQUALF(VecGet(SCurveCtrl(projCurve, iCtrl), 1),
      (float)iCtrl) == false) {
```

```
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeOrthoGetProjectedCurve failed");
        PBErrCatch(GenBrushErr);
      }
    }
  }
  GBEyeFree(&eye);
  SCurveFree(&curve);
  SCurveFree(&projCurve);
  printf("UnitTestGBEyeOrthoProcessCurve OK\n");
}

void UnitTestGBEyeOrthoProcessShapoid() {
  Facoid* facoid = FacoidCreate(2);
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 2.0);
  ShapoidTranslate(facoid, &v);
  VecSet(&v, 0, 2.0); VecSet(&v, 1, 3.0);
  ShapoidScale(facoid, (VecFloat*)&v);
  GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
  Shapoid* shap =
    GBEyeGetProjectedShapoid((GBEye*)eye, (Shapoid*)facoid);
  VecFloat2D checka = VecFloatCreateStatic2D();
  VecSet(&checka, 0, 1.0); VecSet(&checka, 1, 2.0);
  VecFloat2D checkb = VecFloatCreateStatic2D();
  VecSet(&checkb, 0, 2.0); VecSet(&checkb, 1, 0.0);
  VecFloat2D checkc = VecFloatCreateStatic2D();
  VecSet(&checkc, 0, 0.0); VecSet(&checkc, 1, 3.0);
  if (VecIsEqual(ShapoidPos(shap), &checka) == false ||
    VecIsEqual(ShapoidAxis(shap, 0), &checkb) == false ||
    VecIsEqual(ShapoidAxis(shap, 1), &checkc) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBEyeOrthoGetProjectedShapoid failed");
    PBErrCatch(GenBrushErr);
  }
  ShapoidFree(&shap);
  GBEyeFree(&eye);
  ShapoidFree(&facoid);

  Spheroid* spheroid = SpheroidCreate(3);
  VecFloat3D u = VecFloatCreateStatic3D();
  VecSet(&u, 0, 1.0); VecSet(&u, 1, 2.0); VecSet(&u, 2, 3.0);
  ShapoidTranslate(spheroid, &u);
  VecSet(&u, 0, 2.0); VecSet(&u, 1, 3.0); VecSet(&u, 1, 4.0);
  ShapoidScale(spheroid, (VecFloat*)&u);
  VecFloat3D checkd[6];
  VecFloat3D checke[6];
  VecFloat3D checkf[6];
  VecFloat3D checkg[6];
  for (int iView = 6; iView--;) {
    checkd[iView] = VecFloatCreateStatic3D();
    checke[iView] = VecFloatCreateStatic3D();
    checkf[iView] = VecFloatCreateStatic3D();
    checkg[iView] = VecFloatCreateStatic3D();
  }
  VecSet(checkd, 0, 3.0); VecSet(checkd, 1, 2.0);
  VecSet(checkd, 2, -1.0);
  VecSet(checke, 0, 0.0); VecSet(checke, 1, 0.0);
  VecSet(checke, 2, -2.0);
  VecSet(checkf, 0, 0.0); VecSet(checkf, 1, 4.0);
  VecSet(checkf, 2, 0.0);
  VecSet(checkg, 0, 3.0); VecSet(checkg, 1, 0.0);
  VecSet(checkg, 2, 0.0);
```

166

```
      VecSet(checkd + 1, 0, -3.0); VecSet(checkd + 1, 1, 2.0);
      VecSet(checkd + 1, 2, 1.0);
      VecSet(checke + 1, 0, 0.0); VecSet(checke + 1, 1, 0.0);
      VecSet(checke + 1, 2, 2.0);
      VecSet(checkf + 1, 0, 0.0); VecSet(checkf + 1, 1, 4.0);
      VecSet(checkf + 1, 2, 0.0);
      VecSet(checkg + 1, 0, -3.0); VecSet(checkg + 1, 1, 0.0);
      VecSet(checkg + 1, 2, 0.0);
      VecSet(checkd + 2, 0, 1.0); VecSet(checkd + 2, 1, -3.0);
      VecSet(checkd + 2, 2, 2.0);
      VecSet(checke + 2, 0, 2.0); VecSet(checke + 2, 1, 0.0);
      VecSet(checke + 2, 2, 0.0);
      VecSet(checkf + 2, 0, 0.0); VecSet(checkf + 2, 1, 0.0);
      VecSet(checkf + 2, 2, 4.0);
      VecSet(checkg + 2, 0, 0.0); VecSet(checkg + 2, 1, -3.0);
      VecSet(checkg + 2, 2, 0.0);
      VecSet(checkd + 3, 0, 1.0); VecSet(checkd + 3, 1, 3.0);
      VecSet(checkd + 3, 2, -2.0);
      VecSet(checke + 3, 0, 2.0); VecSet(checke + 3, 1, 0.0);
      VecSet(checke + 3, 2, 0.0);
      VecSet(checkf + 3, 0, 0.0); VecSet(checkf + 3, 1, 0.0);
      VecSet(checkf + 3, 2, -4.0);
      VecSet(checkg + 3, 0, 0.0); VecSet(checkg + 3, 1, 3.0);
      VecSet(checkg + 3, 2, 0.0);
      VecSet(checkd + 4, 0, -1.0); VecSet(checkd + 4, 1, 2.0);
      VecSet(checkd + 4, 2, -3.0);
      VecSet(checke + 4, 0, -2.0); VecSet(checke + 4, 1, 0.0);
      VecSet(checke + 4, 2, 0.0);
      VecSet(checkf + 4, 0, 0.0); VecSet(checkf + 4, 1, 4.0);
      VecSet(checkf + 4, 2, 0.0);
      VecSet(checkg + 4, 0, 0.0); VecSet(checkg + 4, 1, 0.0);
      VecSet(checkg + 4, 2, -3.0);
      VecSet(checkd + 5, 0, 1.0); VecSet(checkd + 5, 1, 2.0);
      VecSet(checkd + 5, 2, 3.0);
      VecSet(checke + 5, 0, 2.0); VecSet(checke + 5, 1, 0.0);
      VecSet(checke + 5, 2, 0.0);
      VecSet(checkf + 5, 0, 0.0); VecSet(checkf + 5, 1, 4.0);
      VecSet(checkf + 5, 2, 0.0);
      VecSet(checkg + 5, 0, 0.0); VecSet(checkg + 5, 1, 0.0);
      VecSet(checkg + 5, 2, 3.0);
      for (int iView = 6; iView--;) {
        GBEyeOrthoView view = (GBEyeOrthoView)iView;
        GBEyeOrtho* eye = GBEyeOrthoCreate(view);
        Shapoid* shap =
          GBEyeGetProjectedShapoid((GBEye*)eye, (Shapoid*)spheroid);
        if (VecIsEqual(ShapoidPos(shap), checkd + 5 - iView) == false ||
          VecIsEqual(ShapoidAxis(shap, 0), checke + 5 - iView) == false ||
          VecIsEqual(ShapoidAxis(shap, 1), checkf + 5 - iView) == false ||
          VecIsEqual(ShapoidAxis(shap, 2), checkg + 5 - iView) == false) {
          GenBrushErr->_type = PBErrTypeUnitTestFailed;
          sprintf(GenBrushErr->_msg,
            "GBEyeOrthoGetProjectedShapoid failed");
          PBErrCatch(GenBrushErr);
        }
        ShapoidFree(&shap);
        GBEyeFree(&eye);
      }
      ShapoidFree(&spheroid);
      printf("UnitTestGBEyeOrthoProcessShapoid OK\n");
    }

    void UnitTestGBEyeOrtho() {
```

167

```
    UnitTestGBEyeOrthoCreateFree();
    UnitTestGBEyeOrthoGetSet();
    UnitTestGBEyeOrthoProcessPoint();
    UnitTestGBEyeOrthoProcessCurve();
    UnitTestGBEyeOrthoProcessShapoid();
    printf("UnitTestGBEyeOrtho OK\n");
}

void UnitTestGBEyeIsometricCreateFree() {
    GBEyeIsometric* eye = GBEyeIsometricCreate();
    if (GBEyeGetType(eye) != GBEyeTypeIsometric ||
        ISEQUALF(eye->_thetaRight, PBMATH_QUARTERPI) == false ||
        ISEQUALF(eye->_thetaY, PBMATH_QUARTERPI) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeIsometricCreate failed");
        PBErrCatch(GenBrushErr);
    }
    float check[9] = {
        0.707107, -0.00000, -0.707107,
        0.50000, 0.707107, 0.50000,
        0.50000, -0.707107, 0.50000
        };
    for (int iVal = 9; iVal--;) {
        if (ISEQUALF(eye->_eye._proj->_val[iVal], check[iVal]) == false) {
            GenBrushErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GenBrushErr->_msg, "GBEyeIsometricCreate failed");
            PBErrCatch(GenBrushErr);
        }
    }
    GBEyeFree(&eye);
    if (eye != NULL) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeIsometricFree failed");
        PBErrCatch(GenBrushErr);
    }
    printf("UnitTestGBEyeIsometricCreateFree OK\n");
}

void UnitTestGBEyeIsometricGetSet() {
    GBEyeIsometric* eye = GBEyeIsometricCreate();
    if (ISEQUALF(GBEyeIsometricGetRotY(eye), eye->_thetaY) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeIsometricGetRotY failed");
        PBErrCatch(GenBrushErr);
    }
    if (ISEQUALF(GBEyeIsometricGetRotRight(eye),
        eye->_thetaRight) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeIsometricGetRotRight failed");
        PBErrCatch(GenBrushErr);
    }
    GBEyeIsometricSetRotRight(eye, 0.5 * PBMATH_QUARTERPI);
    if (ISEQUALF(GBEyeIsometricGetRotRight(eye),
        0.5 * PBMATH_QUARTERPI) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeIsometricSetRotRight failed");
        PBErrCatch(GenBrushErr);
    }
    float checka[9] = {
        0.707107, -0.000000, -0.707107,
        0.270598, 0.923880, 0.270598,
        0.653282, -0.382683, 0.653281
```

```
      };
  for (int iVal = 9; iVal--;) {
    if (ISEQUALF(eye->_eye._proj->val[iVal], checka[iVal]) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBEyeIsometricSetRotRight failed");
      PBErrCatch(GenBrushErr);
    }
  }
  GBEyeIsometricSetRotY(eye, 0.5 * PBMATH_QUARTERPI);
  float checkb[9] = {
    0.923880, -0.000000, -0.382683,
    0.146447, 0.923880, 0.353553,
    0.353553, -0.382683, 0.853553
    };
  if (ISEQUALF(GBEyeIsometricGetRotY(eye),
    0.5 * PBMATH_QUARTERPI) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBEyeIsometricSetRotY failed");
    PBErrCatch(GenBrushErr);
  }
  for (int iVal = 9; iVal--;) {
    if (ISEQUALF(eye->_eye._proj->val[iVal], checkb[iVal]) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBEyeIsometricSetRotY failed");
      PBErrCatch(GenBrushErr);
    }
  }
  GBEyeFree(&eye);
  printf("UnitTestGBEyeIsometricGetSet OK\n");
}

void UnitTestGBEyeIsometricProcessPoint() {
  VecFloat2D v2d = VecFloatCreateStatic2D();
  VecSet(&v2d, 0, 0.0); VecSet(&v2d, 1, 1.0);
  VecFloat3D v3d = VecFloatCreateStatic3D();
  VecSet(&v3d, 0, 1.0); VecSet(&v3d, 1, 0.0); VecSet(&v3d, 2, 1.0);
  VecFloat* v4d = VecFloatCreate(4);
  VecSet(v4d, 0, 1.0); VecSet(v4d, 1, 2.0);
  VecSet(v4d, 2, 3.0); VecSet(v4d, 3, 4.0);
  float check2d[2] = {1.878680,5.0};
  float check3d[3] = {1.0,5.0,1.0};
  float check4d[4] = {-6.242640,2.171572,0.585786,4.0};
  GBHandDefault* hand = GBHandDefaultCreate();
  GBEyeIsometric* eye = GBEyeIsometricCreate();
  VecFloat3D scale = VecFloatCreateStatic3D();
  VecSet(&scale, 0, 2.0); VecSet(&scale, 1, 3.0);
  VecSet(&scale, 2, 1.0);
  GBEyeSetScale(eye, &scale);
  VecFloat2D orig = VecFloatCreateStatic2D();
  VecSet(&orig, 0, 4.0); VecSet(&orig, 1, 5.0);
  GBEyeSetOrig(eye, &orig);
  float theta = PBMATH_HALFPI;
  GBEyeSetRot(eye, theta);
  GBLayer layer;
  GBInk ink;
  GBTool tool;
  GBObjPod* pod = GBObjPodCreatePoint((VecFloat*)&v2d, eye,
    hand, &tool, &ink, &layer);
  VecFloat* proj = GBObjPodGetEyeObjAsPoint(pod);
  if (VecGetDim(proj) != VecGetDim(pod->_srcPoint)) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBEyeIsometricProcess failed");
```

169

```
      PBErrCatch(GenBrushErr);
    }
    for (int i = VecGetDim(proj); i--;) {
      if (ISEQUALF(VecGet(proj, i), check2d[i]) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeIsometricProcess failed");
        PBErrCatch(GenBrushErr);
      }
    }
    pod->_srcPoint = (VecFloat*)&v3d;
    GBEyeProcess((GBEye*)eye, pod);
    proj = GBObjPodGetEyeObjAsPoint(pod);
    if (VecGetDim(proj) != 3) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBEyeIsometricProcess failed");
      PBErrCatch(GenBrushErr);
    }
    for (int i = VecGetDim(proj); i--;) {
      if (ISEQUALF(VecGet(proj, i), check3d[i]) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeIsometricProcess failed");
        PBErrCatch(GenBrushErr);
      }
    }
    pod->_srcPoint = v4d;
    GBEyeProcess((GBEye*)eye, pod);
    proj = GBObjPodGetEyeObjAsPoint(pod);
    if (VecGetDim(proj) != 4) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBEyeIsometricProcess failed");
      PBErrCatch(GenBrushErr);
    }
    for (int i = VecGetDim(proj); i--;) {
      if (ISEQUALF(VecGet(proj, i), check4d[i]) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBEyeIsometricProcess failed");
        PBErrCatch(GenBrushErr);
      }
    }
  GBEyeFree(&eye);
  GBObjPodFree(&pod);
  GBHandDefaultFree(&hand);
  VecFree(&v4d);
  printf("UnitTestGBEyeIsometricProcessPoint OK\n");
}

void UnitTestGBEyeIsometricProcessCurve() {
  SCurve* curve = SCurveCreate(1, 2, 2);
  for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    VecSet(SCurveCtrl(curve, iCtrl), 0, (float)iCtrl);
    VecSet(SCurveCtrl(curve, iCtrl), 1, (float)iCtrl * 2.0);
  }
  GBEyeIsometric* eye = GBEyeIsometricCreate();
  SCurve* projCurve = GBEyeGetProjectedCurve((GBEye*)eye, curve);
  if (SCurveGetDim(projCurve) != SCurveGetDim(curve)) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg,
      "GBEyeIsometricGetProjectedCurve failed");
    PBErrCatch(GenBrushErr);
  }
  float check[3][2]={
    {0.0,0.0},
```

```
      {0.707107,1.914214},
      {1.414214,3.828427}
      };
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
      if (ISEQUALF(VecGet(SCurveCtrl(projCurve, iCtrl), 0),
        check[iCtrl][0]) == false ||
        ISEQUALF(VecGet(SCurveCtrl(projCurve, iCtrl), 1),
        check[iCtrl][1]) == false) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg,
          "GBEyeIsometricGetProjectedCurve failed");
        PBErrCatch(GenBrushErr);
      }
    }
  GBEyeFree(&eye);
  SCurveFree(&curve);
  SCurveFree(&projCurve);
  printf("UnitTestGBEyeIsometricProcessCurve OK\n");
}

void UnitTestGBEyeIsometricProcessShapoid() {
  Facoid* facoid = FacoidCreate(3);
  VecFloat3D v = VecFloatCreateStatic3D();
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 2.0); VecSet(&v, 2, 3.0);
  ShapoidTranslate(facoid, &v);
  VecSet(&v, 0, 3.0); VecSet(&v, 1, 2.0); VecSet(&v, 2, 1.0);
  ShapoidScale(facoid, (VecFloat*)&v);
  GBEyeIsometric* eye = GBEyeIsometricCreate();
  Shapoid* shap =
    GBEyeGetProjectedShapoid((GBEye*)eye, (Shapoid*)facoid);
  VecFloat3D checka = VecFloatCreateStatic3D();
  VecSet(&checka, 0, -1.414214); VecSet(&checka, 1, 3.414214);
  VecSet(&checka, 2, 0.585786);
  VecFloat3D checkb = VecFloatCreateStatic3D();
  VecSet(&checkb, 0, 2.121320); VecSet(&checkb, 1, 1.500000);
  VecSet(&checkb, 2, 1.500000);
  VecFloat3D checkc = VecFloatCreateStatic3D();
  VecSet(&checkc, 0, 0.000000); VecSet(&checkc, 1, 1.414214);
  VecSet(&checkc, 2, -1.414214);
  VecFloat3D checkd = VecFloatCreateStatic3D();
  VecSet(&checkd, 0, -0.707107); VecSet(&checkd, 1, 0.500000);
  VecSet(&checkd, 2, 0.500000);
  if (VecIsEqual(ShapoidPos(shap), &checka) == false ||
    VecIsEqual(ShapoidAxis(shap, 0), &checkb) == false ||
    VecIsEqual(ShapoidAxis(shap, 1), &checkc) == false ||
    VecIsEqual(ShapoidAxis(shap, 2), &checkd) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg,
      "GBEyeIsometricGetProjectedShapoid failed");
    PBErrCatch(GenBrushErr);
  }
  ShapoidFree(&shap);
  GBEyeFree(&eye);
  ShapoidFree(&facoid);
  printf("UnitTestGBEyeIsometricProcessShapoid OK\n");
}

void UnitTestGBEyeIsometric() {
  UnitTestGBEyeIsometricCreateFree();
  UnitTestGBEyeIsometricGetSet();
  UnitTestGBEyeIsometricProcessPoint();
  UnitTestGBEyeIsometricProcessCurve();
```

```
  UnitTestGBEyeIsometricProcessShapoid();
  printf("UnitTestGBEyeIsometric OK\n");
}

void UnitTestGBEyeCreateFree() {
  VecFloat3D v = VecFloatCreateStatic3D();
  VecSet(&v, 0, 1.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 1.0);
  VecFloat2D w = VecFloatCreateStatic2D();
  GBEye eye = GBEyeCreateStatic(GBEyeTypeOrtho);
  if (eye._type != GBEyeTypeOrtho ||
    VecIsEqual(&(eye._scale), &v) == false ||
    VecIsEqual(&(eye._orig), &w) == false ||
    eye._proj == NULL) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBEyeCreateStatic failed");
    PBErrCatch(ShapoidErr);
  }
  GBEyeFreeStatic(&eye);
  if (eye._proj != NULL) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBEyeFreeStatic failed");
    PBErrCatch(ShapoidErr);
  }
  printf("UnitTestGBEyeCreateFree OK\n");
}

void UnitTestGBEyeGetSet() {
  GBEye eye = GBEyeCreateStatic(GBEyeTypeOrtho);
  if (GBEyeGetType(&eye) != GBEyeTypeOrtho) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBEyeGetType failed");
    PBErrCatch(ShapoidErr);
  }
  if (GBEyeScale(&eye) != &(eye._scale)) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBEyeScale failed");
    PBErrCatch(ShapoidErr);
  }
  VecFloat3D v = GBEyeGetScale(&eye);
  if (VecIsEqual(&v, &(eye._scale)) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBEyeGetScale failed");
    PBErrCatch(ShapoidErr);
  }
  if (GBEyeOrig(&eye) != &(eye._orig)) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBEyeOrig failed");
    PBErrCatch(ShapoidErr);
  }
  VecFloat2D w = GBEyeGetOrig(&eye);
  if (VecIsEqual(&w, &(eye._orig)) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBEyeGetOrig failed");
    PBErrCatch(ShapoidErr);
  }
  if (GBEyeGetRot(&eye) != eye._theta) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBEyeGetRot failed");
    PBErrCatch(ShapoidErr);
  }
  VecSet(&v, 0, 2.0); VecSet(&v, 1, 3.0); VecSet(&v, 2, 4.0);
  GBEyeSetScale(&eye, &v);
```

```
    if (VecIsEqual(&v, &(eye._scale)) == false) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBEyeSetScale failed");
      PBErrCatch(ShapoidErr);
    }
    VecSet(&v, 0, 3.0); VecSet(&v, 1, 3.0); VecSet(&v, 2, 3.0);
    GBEyeSetScale(&eye, (float)3.0);
    if (VecIsEqual(&v, &(eye._scale)) == false) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBEyeSetScale failed");
      PBErrCatch(ShapoidErr);
    }
    GBEyeSetOrig(&eye, &w);
    if (VecIsEqual(&w, &(eye._orig)) == false) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBEyeSetOrig failed");
      PBErrCatch(ShapoidErr);
    }
    float theta = 1.0;
    GBEyeSetRot(&eye, theta);
    if (ISEQUALF(eye._theta, theta) == false) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBEyeSetRot failed");
      PBErrCatch(ShapoidErr);
    }
    if (GBEyeProj(&eye) != eye._proj) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBEyeProj failed");
      PBErrCatch(ShapoidErr);
    }
    GBEyeFreeStatic(&eye);
    printf("UnitTestGBEyeGetSet OK\n");
}

void UnitTestGBEye() {
    UnitTestGBEyeCreateFree();
    UnitTestGBEyeGetSet();
    UnitTestGBEyeOrtho();
    UnitTestGBEyeIsometric();
    printf("UnitTestGBEye OK\n");
}

void UnitTestGBHandDefaultCreateFree() {
    GBHandDefault* hand = GBHandDefaultCreate();
    if (GBHandGetType(hand) != GBHandTypeDefault) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBHandDefaultCreate failed");
      PBErrCatch(ShapoidErr);
    }
    GBHandDefaultFree(&hand);
    if (hand != NULL) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBHandDefaultFree failed");
      PBErrCatch(ShapoidErr);
    }
    printf("UnitTestGBHandDefaultCreateFree OK\n");
}

void UnitTestGBHandDefaultProcess() {
    VecFloat3D point = VecFloatCreateStatic3D();
    VecSet(&point, 0, 1.0); VecSet(&point, 1, 2.0);
    VecSet(&point, 2, 3.0);
```

```
    GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
    GBHandDefault* hand = GBHandDefaultCreate();
    GBTool tool;
    GBInk ink;
    GBLayer layer;
    GBObjPod* pod =
      GBObjPodCreatePoint(&point, &eye, hand, &tool, &ink, &layer);
    GBHandProcess(hand, pod);
    if (GSetNbElem(&(pod->_handPoints)) == 0 ||
      VecIsEqual(GSetGetFirst(&(pod->_handPoints)),
        &point) == false) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBHandDefaultProcess failed");
      PBErrCatch(ShapoidErr);
    }
    GBEyeOrthoFree(&eye);
    GBObjPodFree(&pod);
    GBHandDefaultFree(&hand);
    printf("UnitTestGBHandDefaultProcess OK\n");
}

void UnitTestGBHandDefault() {
  UnitTestGBHandDefaultCreateFree();
  UnitTestGBHandDefaultProcess();
  printf("UnitTestGBHandDefault OK\n");
}

void UnitTestGBHandCreateFree() {
  GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
  if (hand._type != GBHandTypeDefault) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBHandCreateStatic failed");
    PBErrCatch(ShapoidErr);
  }
  GBHandFreeStatic(&hand);
  printf("UnitTestGBHandCreateFree OK\n");
}

void UnitTestGBHandGetSet() {
  GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
  if (GBHandGetType(&hand) != GBHandTypeDefault) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBHandGetType failed");
    PBErrCatch(ShapoidErr);
  }
  GBHandFreeStatic(&hand);
  printf("UnitTestGBHandCreateFree OK\n");
}

void UnitTestGBHand() {
  UnitTestGBHandCreateFree();
  UnitTestGBHandGetSet();
  UnitTestGBHandDefault();
  printf("UnitTestGBHand OK\n");
}

void UnitTestGBToolPlotterCreateFree() {
  GBToolPlotter* tool = GBToolPlotterCreate();
  if (tool->_tool._type != GBToolTypePlotter) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBToolPlotterCreate failed");
    PBErrCatch(GenBrushErr);
```

```
    }
    GBToolPlotterFree(&tool);
    if (tool != NULL) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBToolPlotterFree failed");
      PBErrCatch(GenBrushErr);
    }
    printf("UnitTestGBToolPlotterCreateFree OK\n");
}

void UnitTestGBToolPlotterDrawPoint() {
  GBToolPlotter* tool = GBToolPlotterCreate();
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
  GBSurface* surf = (GBSurface*)GBSurfaceImageCreate(&dim);
  GBPixel transparent = GBColorTransparent;
  GBSurfaceSetBgColor(surf, &transparent);
  GBLayer* layer = GBSurfaceAddLayer(surf, &dim);
  GBPixel red = GBColorRed;
  GBInkSolid* ink = GBInkSolidCreate(&red);
  VecFloat* pos = VecFloatCreate(3);
  VecSet(pos, 0, 4.0); VecSet(pos, 1, 4.0); VecSet(pos, 2, 0.0);
  GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
  GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
  GBObjPod* pod =
    GBObjPodCreatePoint(pos, &eye, &hand, tool, ink, layer);
  GSetAppend(&(pod->_handPoints), VecClone(pos));
  GBToolPlotterDraw(tool, pod);
  GBSurfaceUpdate(surf);
  unsigned char check[400] = {
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 255,0,0,255,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
  };
  for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
    if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
      check[iPix * 4] ||
      surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
      check[iPix * 4 + 1] ||
      surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
      check[iPix * 4 + 2] ||
      surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
      check[iPix * 4 + 3]) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBToolPlotterDraw failed");
```

```
      PBErrCatch(ShapoidErr);
    }
  }
  GBSurfaceImageSetFileName((GBSurfaceImage*)surf,
    "./GBToolPlotterDrawPoint.tga");
  GBSurfaceImageSave((GBSurfaceImage*)surf);
  GBObjPodFree(&pod);
  GBToolPlotterFree(&tool);
  GBSurfaceImageFree((GBSurfaceImage**)&surf);
  GBInkSolidFree(&ink);
  GBEyeOrthoFree(&eye);
  VecFree(&pos);
  printf("UnitTestGBToolPlotterDrawPoint OK\n");
}

void UnitTestGBToolPlotterDrawFacoid() {
  GBToolPlotter* tool = GBToolPlotterCreate();
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
  GBSurface* surf = (GBSurface*)GBSurfaceImageCreate(&dim);
  GBPixel transparent = GBColorTransparent;
  GBSurfaceSetBgColor(surf, &transparent);
  GBLayer* layer = GBSurfaceAddLayer(surf, &dim);
  GBPixel blue = GBColorBlue;
  GBInkSolid* ink = GBInkSolidCreate(&blue);
  Facoid* facoid = FacoidCreate(2);
  Shapoid* shap = (Shapoid*)facoid;
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 7.0); VecSet(&v, 1, 4.0);
  ShapoidScale(shap, (VecFloat*)&v);
  VecSet(&v, 0, 2.0); VecSet(&v, 1, 4.0);
  ShapoidTranslate(shap, (VecFloat*)&v);
  ShapoidRotStart(shap, PBMATH_QUARTERPI);
  GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
  GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
  GBObjPod* pod =
    GBObjPodCreateShapoid(shap, &eye, &hand, tool, ink, layer);
  GBToolPlotterDraw(tool, pod);
  GBSurfaceUpdate(surf);
  unsigned char check[400] = {
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
    0,0,255,255, 0,0,255,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255,
    0,0,255,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
    };
  for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
```

```
    if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
      check[iPix * 4] ||
      surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
      check[iPix * 4 + 1] ||
      surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
      check[iPix * 4 + 2] ||
      surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
      check[iPix * 4 + 3]) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBToolPlotterDraw failed");
      PBErrCatch(ShapoidErr);
    }
  }
  GBSurfaceImageSetFileName((GBSurfaceImage*)surf,
    "./GBToolPlotterDrawFacoid.tga");
  GBSurfaceImageSave((GBSurfaceImage*)surf);
  GBObjPodFree(&pod);
  GBToolPlotterFree(&tool);
  GBSurfaceImageFree((GBSurfaceImage**)&surf);
  GBInkSolidFree(&ink);
  ShapoidFree(&facoid);
  GBEyeOrthoFree(&eye);
  printf("UnitTestGBToolPlotterDrawFacoid OK\n");
}

void UnitTestGBToolPlotterDrawPyramidoid() {
  GBToolPlotter* tool = GBToolPlotterCreate();
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
  GBSurface* surf = (GBSurface*)GBSurfaceImageCreate(&dim);
  GBPixel transparent = GBColorTransparent;
  GBSurfaceSetBgColor(surf, &transparent);
  GBLayer* layer = GBSurfaceAddLayer(surf, &dim);
  GBPixel blue = GBColorBlue;
  GBInkSolid* ink = GBInkSolidCreate(&blue);
  Pyramidoid* pyramidoid = PyramidoidCreate(2);
  Shapoid* shap = (Shapoid*)pyramidoid;
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 7.0); VecSet(&v, 1, 4.0);
  ShapoidScale(shap, (VecFloat*)&v);
  VecSet(&v, 0, 2.0); VecSet(&v, 1, 4.0);
  ShapoidTranslate(shap, (VecFloat*)&v);
  GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
  GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
  GBObjPod* pod =
    GBObjPodCreateShapoid(shap, &eye, &hand, tool, ink, layer);
  GSetAppend(&(pod->_handShapoids), shap);
  GBToolPlotterDraw(tool, pod);
  GBSurfaceUpdate(surf);
  unsigned char checka[400] = {
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255,
    0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255,
    0,0,255,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
```

```
      0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
    };
    for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
      if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
        checka[iPix * 4] ||
        surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
        checka[iPix * 4 + 1] ||
        surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
        checka[iPix * 4 + 2] ||
        surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
        checka[iPix * 4 + 3]) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "GBToolPlotterDraw failed");
        PBErrCatch(ShapoidErr);
      }
    }
    GBSurfaceImageSetFileName((GBSurfaceImage*)surf,
      "./GBToolPlotterDrawPyramidoid.tga");
    GBSurfaceImageSave((GBSurfaceImage*)surf);
    GBObjPodFree(&pod);
    GBToolPlotterFree(&tool);
    GBSurfaceImageFree((GBSurfaceImage**)&surf);
    GBInkSolidFree(&ink);
    GBEyeOrthoFree(&eye);
    printf("UnitTestGBToolPlotterDrawPyramidoid OK\n");
}

void UnitTestGBToolPlotterDrawSpheroid() {
  GBToolPlotter* tool = GBToolPlotterCreate();
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
  GBSurface* surf = (GBSurface*)GBSurfaceImageCreate(&dim);
  GBPixel transparent = GBColorTransparent;
  GBSurfaceSetBgColor(surf, &transparent);
  GBLayer* layer = GBSurfaceAddLayer(surf, &dim);
  GBPixel blue = GBColorBlue;
  GBInkSolid* ink = GBInkSolidCreate(&blue);
  Spheroid* spheroid = SpheroidCreate(2);
  Shapoid* shap = (Shapoid*)spheroid;
  VecFloat2D v = VecFloatCreateStatic2D();
  VecSet(&v, 0, 10.0); VecSet(&v, 1, 8.0);
  ShapoidScale(shap, (VecFloat*)&v);
  VecSet(&v, 0, 5.0); VecSet(&v, 1, 5.0);
  ShapoidTranslate(shap, (VecFloat*)&v);
  GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
  GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
  GBObjPod* pod =
    GBObjPodCreateShapoid(shap, &eye, &hand, tool, ink, layer);
  GSetAppend(&(pod->_handShapoids), shap);
  GBToolPlotterDraw(tool, pod);
  GBSurfaceUpdate(surf);
  unsigned char checka[400] = {
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255,
```

```
      0,0,255,255, 0,0,255,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
    };
  for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
    if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
      checka[iPix * 4] ||
      surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
      checka[iPix * 4 + 1] ||
      surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
      checka[iPix * 4 + 2] ||
      surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
      checka[iPix * 4 + 3]) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBToolPlotterDraw failed");
      PBErrCatch(ShapoidErr);
    }
  }
  GBSurfaceImageSetFileName((GBSurfaceImage*)surf,
    "./GBToolPlotterDrawSpheroid.tga");
  GBSurfaceImageSave((GBSurfaceImage*)surf);
  GBObjPodFree(&pod);
  GBToolPlotterFree(&tool);
  GBSurfaceImageFree((GBSurfaceImage**)&surf);
  GBInkSolidFree(&ink);
  GBEyeOrthoFree(&eye);
  printf("UnitTestGBToolPlotterDrawSpheroid OK\n");
}

void UnitTestGBToolPlotterDrawFacoid3D() {
  GBToolPlotter* tool = GBToolPlotterCreate();
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
  GBSurface* surf = (GBSurface*)GBSurfaceImageCreate(&dim);
  GBPixel transparent = GBColorTransparent;
  GBSurfaceSetBgColor(surf, &transparent);
  GBLayer* layer = GBSurfaceAddLayer(surf, &dim);
  GBLayerSetStackPos(layer, GBLayerStackPosInside);
  GBLayerSetBlendMode(layer, GBLayerBlendModeOver);
  GBPixel blue = GBColorBlue;
  blue._rgba[GBPixelAlpha] = 10;
  GBInkSolid* ink = GBInkSolidCreate(&blue);
  Facoid* facoid = FacoidCreate(3);
  Shapoid* shap = (Shapoid*)facoid;
  VecFloat3D v = VecFloatCreateStatic3D();
  VecSet(&v, 0, 7.0); VecSet(&v, 1, 4.0); VecSet(&v, 2, 4.0);
  ShapoidScale(shap, (VecFloat*)&v);
  VecSet(&v, 0, 2.0); VecSet(&v, 1, 4.0); VecSet(&v, 2, 0.0);
```

```
      ShapoidTranslate(shap, (VecFloat*)&v);
      ShapoidRotYStart(shap, PBMATH_QUARTERPI);
      GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
      GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
      GBObjPod* pod =
        GBObjPodCreateShapoid(shap, &eye, &hand, tool, ink, layer);
      GBToolPlotterDraw(tool, pod);
      GBSurfaceUpdate(surf);
      unsigned char checka[400] = {
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,235,20, 0,0,235,20, 0,0,195,60,
        0,0,195,60, 0,0,205,50, 0,0,215,40, 0,0,235,20, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,235,20, 0,0,235,20, 0,0,195,60,
        0,0,195,60, 0,0,205,50, 0,0,215,40, 0,0,235,20, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,235,20, 0,0,235,20, 0,0,195,60,
        0,0,195,60, 0,0,205,50, 0,0,215,40, 0,0,235,20, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,235,20, 0,0,235,20, 0,0,195,60,
        0,0,195,60, 0,0,205,50, 0,0,215,40, 0,0,235,20, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
      };
      for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
        if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
          checka[iPix * 4] ||
          surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
          checka[iPix * 4 + 1] ||
          surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
          checka[iPix * 4 + 2] ||
          surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
          checka[iPix * 4 + 3]) {
          ShapoidErr->_type = PBErrTypeUnitTestFailed;
          sprintf(ShapoidErr->_msg, "GBToolPlotterDraw failed");
          PBErrCatch(ShapoidErr);
        }
      }
      GBSurfaceImageSetFileName((GBSurfaceImage*)surf,
        "./GBToolPlotterDrawFacoid3D.tga");
      GBSurfaceImageSave((GBSurfaceImage*)surf);
      GBObjPodFree(&pod);
      GBToolPlotterFree(&tool);
      GBSurfaceImageFree((GBSurfaceImage**)&surf);
      GBInkSolidFree(&ink);
      ShapoidFree(&facoid);
      GBEyeOrthoFree(&eye);
      printf("UnitTestGBToolPlotterDrawFacoid3D OK\n");
    }

    void UnitTestGBToolPlotterDrawSCurve() {
      GBToolPlotter* tool = GBToolPlotterCreate();
      VecShort2D dim = VecShortCreateStatic2D();
      VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
      GBSurface* surf = (GBSurface*)GBSurfaceImageCreate(&dim);
      GBPixel transparent = GBColorTransparent;
```

```
GBSurfaceSetBgColor(surf, &transparent);
GBLayer* layer = GBSurfaceAddLayer(surf, &dim);
GBLayerSetStackPos(layer, GBLayerStackPosInside);
GBLayerSetBlendMode(layer, GBLayerBlendModeOver);
GBPixel blue = GBColorBlue;
blue._rgba[GBPixelAlpha] = 10;
GBInkSolid* ink = GBInkSolidCreate(&blue);
SCurve* curve = SCurveCreate(3, 3, 1);
VecFloat3D v = VecFloatCreateStatic3D();
SCurveSetCtrl(curve, 0, (VecFloat*)&v);
VecSet(&v, 0, 10.0); VecSet(&v, 1, 10.0); VecSet(&v, 2, 10.0);
SCurveSetCtrl(curve, 1, (VecFloat*)&v);
VecSet(&v, 0, 10.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, -10.0);
SCurveSetCtrl(curve, 2, (VecFloat*)&v);
VecSet(&v, 0, 0.0); VecSet(&v, 1, 10.0); VecSet(&v, 2, 0.0);
SCurveSetCtrl(curve, 3, (VecFloat*)&v);
GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
GBObjPod* pod =
  GBObjPodCreateSCurve(curve, &eye, &hand, tool, ink, layer);
GBToolPlotterDraw(tool, pod);
GBSurfaceUpdate(surf);
unsigned char checka[400] = {
  0,0,245,10, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,235,20, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,245,10, 0,0,245,10, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,235,20, 0,0,245,10,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,245,10,
  0,0,245,10, 0,0,245,10, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,215,40, 0,0,195,60, 0,0,245,10, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,245,10,
  0,0,245,10, 0,0,245,10, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,235,20, 0,0,245,10,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,245,10, 0,0,245,10, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,235,20, 0,0,0,0, 0,0,0,0, 0,0,0,0,
  0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
  };
for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
  if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
    checka[iPix * 4] ||
    surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
    checka[iPix * 4 + 1] ||
    surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
    checka[iPix * 4 + 2] ||
    surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
    checka[iPix * 4 + 3]) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBToolPlotterDraw failed");
    PBErrCatch(ShapoidErr);
  }
}
GBSurfaceImageSetFileName((GBSurfaceImage*)surf,
  "./GBToolPlotterDrawSCurve.tga");
GBSurfaceImageSave((GBSurfaceImage*)surf);
GBObjPodFree(&pod);
```

```
    GBToolPlotterFree(&tool);
    GBSurfaceImageFree((GBSurfaceImage**)&surf);
    GBInkSolidFree(&ink);
    SCurveFree(&curve);
    GBEyeOrthoFree(&eye);
    printf("UnitTestGBToolPlotterDrawSCurve OK\n");
}

void UnitTestGBToolPlotter() {
    UnitTestGBToolPlotterCreateFree();
    UnitTestGBToolPlotterDrawPoint();
    UnitTestGBToolPlotterDrawFacoid();
    UnitTestGBToolPlotterDrawPyramidoid();
    UnitTestGBToolPlotterDrawSpheroid();
    UnitTestGBToolPlotterDrawFacoid3D();
    UnitTestGBToolPlotterDrawSCurve();

    printf("UnitTestGBToolPlotter OK\n");
}

void UnitTestGBToolCreateFree() {
    GBTool tool = GBToolCreateStatic(GBToolTypePlotter);
    if (tool._type != GBToolTypePlotter) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBToolCreateStatic failed");
        PBErrCatch(GenBrushErr);
    }
    GBToolFreeStatic(&tool);
    printf("UnitTestGBToolCreateFree OK\n");
}

void UnitTestGBToolGetSet() {
    GBTool tool = GBToolCreateStatic(GBToolTypePlotter);
    if (GBToolGetType(&tool) != GBToolTypePlotter) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBToolGetType failed");
        PBErrCatch(GenBrushErr);
    }
    GBToolFreeStatic(&tool);
    printf("UnitTestGBToolGetSet OK\n");
}

void UnitTestGBTool() {
    UnitTestGBToolCreateFree();
    UnitTestGBToolGetSet();
    UnitTestGBToolPlotter();
    printf("UnitTestGBTool OK\n");
}

void UnitTestGBInkSolidCreateFree() {
    GBPixel green = GBColorGreen;
    GBInkSolid* ink = GBInkSolidCreate(&green);
    if (memcmp(&(ink->_color), &green, sizeof(GBPixel)) != 0) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBInkSolidCreate failed");
        PBErrCatch(GenBrushErr);
    }
    if (ink->_ink._type != GBInkTypeSolid) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBInkSolidCreate failed");
        PBErrCatch(GenBrushErr);
    }
```

```
    GBInkSolidFree(&ink);
    if (ink != NULL) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBInkSolidFree failed");
      PBErrCatch(GenBrushErr);
    }
    printf("UnitTestGBInkSolidCreateFree OK\n");
}

void UnitTestGBInkSolidGetSet() {
  GBPixel green = GBColorGreen;
  GBInkSolid* ink = GBInkSolidCreate(&green);
  GBPixel col = GBInkSolidGet(ink);
  if (memcmp(&col, &green, sizeof(GBPixel)) != 0) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBInkSolidGet failed");
    PBErrCatch(GenBrushErr);
  }
  GBPixel blue = GBColorBlue;
  GBInkSolidSet(ink, &blue);
  if (memcmp(&(ink->_color), &blue, sizeof(GBPixel)) != 0) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBInkSolidSet failed");
    PBErrCatch(GenBrushErr);
  }
  col = GBInkGet(ink, NULL, NULL, NULL, NULL, NULL);
  if (memcmp(&col, &blue, sizeof(GBPixel)) != 0) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBInkGet failed");
    PBErrCatch(GenBrushErr);
  }
  if (GBInkGetType(ink) != GBInkTypeSolid) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBInkGetType failed");
    PBErrCatch(GenBrushErr);
  }
  GBInkSolidFree(&ink);
  printf("UnitTestGBInkSolidGetSet OK\n");
}

void UnitTestGBInkSolid() {
  UnitTestGBInkSolidCreateFree();
  UnitTestGBInkSolidGetSet();
  printf("UnitTestGBInkSolid OK\n");
}

void UnitTestGBInk() {
  UnitTestGBInkSolid();
  printf("UnitTestGBInk OK\n");
}

void UnitTestGBObjPodCreateFree() {
  GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
  GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
  GBTool tool;
  GBLayer layer;
  GBInk ink;
  VecFloat3D pos = VecFloatCreateStatic3D();
  GBObjPod* pod =
    GBObjPodCreatePoint(&pos, eye, &hand, &tool, &ink, &layer);
  if (pod->_type != GBObjTypePoint ||
    pod->_srcPoint != (VecFloat*)&pos ||
```

```
      pod->_eyePoint == NULL ||
      GSetNbElem(&(pod->_handPoints)) != 1 ||
      GSetNbElem(&(pod->_handShapoids)) != 0 ||
      GSetNbElem(&(pod->_handSCurves)) != 0 ||
      pod->_eye != (GBEye*)eye ||
      pod->_hand != &hand ||
      pod->_tool != &tool ||
      pod->_ink != &ink ||
      pod->_layer != &layer) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBObjPodCreatePoint failed");
      PBErrCatch(GenBrushErr);
    }
    GBObjPodFree(&pod);
    if (pod != NULL) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBObjPodFree failed");
      PBErrCatch(GenBrushErr);
    }
    Facoid* facoid = FacoidCreate(2);
    pod = GBObjPodCreateShapoid(facoid, eye, &hand, &tool, &ink, &layer);
    if (pod->_type != GBObjTypeShapoid ||
      pod->_srcShapoid != (Shapoid*)facoid ||
      pod->_eyeShapoid == NULL ||
      GSetNbElem(&(pod->_handPoints)) != 0 ||
      GSetNbElem(&(pod->_handShapoids)) != 1 ||
      GSetNbElem(&(pod->_handSCurves)) != 0 ||
      pod->_eye != (GBEye*)eye ||
      pod->_hand != &hand ||
      pod->_tool != &tool ||
      pod->_ink != &ink ||
      pod->_layer != &layer) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBObjPodCreateShapoid failed");
      PBErrCatch(GenBrushErr);
    }
    GBObjPodFree(&pod);
    SCurve* curve = SCurveCreate(2, 2, 1);
    pod = GBObjPodCreateSCurve(curve, eye, &hand, &tool, &ink, &layer);
    if (pod->_type != GBObjTypeSCurve ||
      pod->_srcSCurve != curve ||
      pod->_eyeSCurve == NULL ||
      GSetNbElem(&(pod->_handPoints)) != 0 ||
      GSetNbElem(&(pod->_handShapoids)) != 0 ||
      GSetNbElem(&(pod->_handSCurves)) != 1 ||
      pod->_eye != (GBEye*)eye ||
      pod->_hand != &hand ||
      pod->_tool != &tool ||
      pod->_ink != &ink ||
      pod->_layer != &layer) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBObjPodCreateSCurve failed");
      PBErrCatch(GenBrushErr);
    }
    GBObjPodFree(&pod);
    SCurveFree(&curve);
    ShapoidFree(&facoid);
    GBEyeOrthoFree(&eye);
    printf("UnitTestGBObjPodCreateFree OK\n");
}

void UnitTestGBObjPodGetSet() {
```

```
GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
GBHandDefault* hand = GBHandDefaultCreate();
GBTool* tool = PBErrMalloc(GenBrushErr, sizeof(GBTool));
GBLayer* layer = PBErrMalloc(GenBrushErr, sizeof(GBLayer));
GBInk* ink = PBErrMalloc(GenBrushErr, sizeof(GBInk));
VecFloat2D pos = VecFloatCreateStatic2D();
GBObjPod* pod =
  GBObjPodCreatePoint(&pos, eye, hand, tool, ink, layer);
if (GBObjPodGetType(pod) != GBObjTypePoint) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBObjPodGetType failed");
  PBErrCatch(GenBrushErr);
}
if (GBObjPodGetObjAsPoint(pod) != (VecFloat*)&pos) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBObjPodGetObjAsPoint failed");
  PBErrCatch(GenBrushErr);
}
if (GBObjPodGetHandObjAsPoints(pod) != &(pod->_handPoints)) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBObjPodGetHandObjAsPoints failed");
  PBErrCatch(GenBrushErr);
}
if (GBObjPodGetHandObjAsShapoids(pod) != &(pod->_handShapoids)) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBObjPodGetHandObjAsShapoids failed");
  PBErrCatch(GenBrushErr);
}
if (GBObjPodGetHandObjAsSCurves(pod) != &(pod->_handSCurves)) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBObjPodGetHandObjAsSCurves failed");
  PBErrCatch(GenBrushErr);
}
if (GBObjPodGetEyeObjAsPoint(pod) != pod->_eyePoint) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBObjPodGetEyeObjAsPoint failed");
  PBErrCatch(GenBrushErr);
}
VecFloat* posEyeB = VecFloatCreate(2);
GBObjPodSetEyePoint(pod, posEyeB);
if (GBObjPodGetEyeObjAsPoint(pod) != posEyeB) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBObjPodSetEyePoint failed");
  PBErrCatch(GenBrushErr);
}
GBObjPodFree(&pod);
Facoid* shap = FacoidCreate(2);
pod = GBObjPodCreateShapoid(shap, eye, hand, tool, ink, layer);
if (GBObjPodGetType(pod) != GBObjTypeShapoid) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBObjPodGetType failed");
  PBErrCatch(GenBrushErr);
}
if (GBObjPodGetObjAsShapoid(pod) != (Shapoid*)shap) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBObjPodGetObjAsShapoid failed");
  PBErrCatch(GenBrushErr);
}
if (GBObjPodGetEyeObjAsShapoid(pod) != pod->_eyeShapoid) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBObjPodGetEyeObjAsShapoid failed");
  PBErrCatch(GenBrushErr);
```

185

```
  }
  if (GBObjPodGetEye(pod) != (GBEye*)eye) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBObjPodGetEye failed");
    PBErrCatch(GenBrushErr);
  }
  if (GBObjPodGetHand(pod) != (GBHand*)hand) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBObjPodGetHand failed");
    PBErrCatch(GenBrushErr);
  }
  if (GBObjPodGetTool(pod) != tool) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBObjPodGetTool failed");
    PBErrCatch(GenBrushErr);
  }
  if (GBObjPodGetInk(pod) != ink) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBObjPodGetInk failed");
    PBErrCatch(GenBrushErr);
  }
  if (GBObjPodGetLayer(pod) != layer) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBObjPodGetLayer failed");
    PBErrCatch(GenBrushErr);
  }
  Facoid* shapEyeB = FacoidCreate(2);
  GBEyeOrtho* eyeB = GBEyeOrthoCreate(GBEyeOrthoViewFront);
  GBHandDefault* handB = GBHandDefaultCreate();
  GBTool* toolB = PBErrMalloc(GenBrushErr, sizeof(GBTool));
  GBLayer* layerB = PBErrMalloc(GenBrushErr, sizeof(GBLayer));
  GBInk* inkB = PBErrMalloc(GenBrushErr, sizeof(GBInk));
  GBObjPodSetEyeShapoid(pod, shapEyeB);
  if (GBObjPodGetEyeObjAsShapoid(pod) != (Shapoid*)shapEyeB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBObjPodSetEyeShap failed");
    PBErrCatch(GenBrushErr);
  }
  GBObjPodSetEye(pod, eyeB);
  GBObjPodSetHand(pod, handB);
  GBObjPodSetTool(pod, toolB);
  GBObjPodSetInk(pod, inkB);
  GBObjPodSetLayer(pod, layerB);
  if (GBObjPodGetEye(pod) != (GBEye*)eyeB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBObjPodSetEye failed");
    PBErrCatch(GenBrushErr);
  }
  if (GBObjPodGetHand(pod) != (GBHand*)handB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBObjPodSetHand failed");
    PBErrCatch(GenBrushErr);
  }
  if (GBObjPodGetTool(pod) != toolB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBObjPodSetTool failed");
    PBErrCatch(GenBrushErr);
  }
  if (GBObjPodGetInk(pod) != inkB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBObjPodSetInk failed");
    PBErrCatch(GenBrushErr);
```

```
    }
    if (GBObjPodGetLayer(pod) != layerB) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBObjPodSetLayer failed");
      PBErrCatch(GenBrushErr);
    }
    GBEyeOrthoFree(&eye);
    GBHandDefaultFree(&hand);
    free(tool);
    free(ink);
    free(layer);
    GBEyeOrthoFree(&eyeB);
    GBHandDefaultFree(&handB);
    free(toolB);
    free(inkB);
    free(layerB);
    ShapoidFree(&shap);
    GBObjPodFree(&pod);
    printf("UnitTestGBObjPodGetSet OK\n");
}

void UnitTestGBObjPod() {
    UnitTestGBObjPodCreateFree();
    UnitTestGBObjPodGetSet();

    printf("UnitTestGBObjPod OK\n");
}

void UnitTestGenBrushCreateFree() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 800); VecSet(&dim, 1, 600);
    GenBrush* gb = GBCreateImage(&dim);
    if (VecIsEqual(GBSurfaceDim(gb->_surf), &dim) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBCreateImage failed");
      PBErrCatch(GenBrushErr);
    }
    if (GBSurfaceGetType(gb->_surf) != GBSurfaceTypeImage) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBCreateImage failed");
      PBErrCatch(GenBrushErr);
    }
    if (GSetNbElem(&(gb->_pods)) != 0) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBCreateImage failed");
      PBErrCatch(GenBrushErr);
    }
    GBFree(&gb);
    if (gb != NULL) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBFree failed");
      PBErrCatch(GenBrushErr);
    }
    printf("UnitTestGenBrushCreateFree OK\n");
}

void UnitTestGenBrushGetSet() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
    GenBrush* gb = GBCreateImage(&dim);
    if (VecIsEqual(GBDim(gb), &dim) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
```

```
    sprintf(GenBrushErr->_msg, "GBDim failed");
    PBErrCatch(GenBrushErr);
}
VecShort2D d = GBGetDim(gb);
if (VecIsEqual(&d, &dim) == false) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBGetDim failed");
  PBErrCatch(GenBrushErr);
}
if (GBSurf(gb) != gb->_surf) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSurf failed");
  PBErrCatch(GenBrushErr);
}
if (GBFinalPixels(gb) != gb->_surf->_finalPix) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBFinalPix failed");
  PBErrCatch(GenBrushErr);
}
if (GBGetType(gb) != GBSurfaceTypeImage) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBGetType failed");
  PBErrCatch(GenBrushErr);
}
VecFloat2D v = VecFloatCreateStatic2D();
GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
GBTool tool;
GBLayer layer;
GBInk ink;
GSetAppend(&(gb->_pods), GBObjPodCreatePoint(&v, &eye, &hand,
  &tool, &ink, &layer));
if (GBGetNbPod(gb) != 1) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBGetNbPod failed");
  PBErrCatch(GenBrushErr);
}
GBPixel black = GBColorBlack;
GBSetBgColor(gb, &black);
if (memcmp(&(gb->_surf->_bgColor), &black, sizeof(GBPixel)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSetBgColor failed");
  PBErrCatch(GenBrushErr);
}
if (memcmp(GBBgColor(gb), &black, sizeof(GBPixel)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBBgColor failed");
  PBErrCatch(GenBrushErr);
}
GBPixel bgCol = GBGetBgColor(gb);
if (memcmp(&(gb->_surf->_bgColor), &bgCol, sizeof(GBPixel)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBGetBgColor failed");
  PBErrCatch(GenBrushErr);
}
char* fileName = "./";
GBSetFileName(gb, fileName);
if (strcmp(((GBSurfaceImage*)(gb->_surf))->_fileName,
  fileName) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSetFileName failed");
  PBErrCatch(GenBrushErr);
```

```
}
if (strcmp(GBFileName(gb), fileName) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBFileName failed");
  PBErrCatch(GenBrushErr);
}
VecShort2D p = VecShortCreateStatic2D();
VecSet(&p, 0, 1); VecSet(&p, 1, 2);
if (GBFinalPixel(gb, &p) != gb->_surf->_finalPix + 9) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBFinalPixel failed");
  PBErrCatch(GenBrushErr);
}
if (GBFinalPixelSafe(gb, &p) != gb->_surf->_finalPix + 9) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBFinalPixelSafe failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, 1); VecSet(&p, 1, -2);
if (GBFinalPixelSafe(gb, &p) != NULL) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBFinalPixelSafe failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, 1); VecSet(&p, 1, 2);
GBPixel white = GBColorWhite;
GBSetFinalPixel(gb, &p, &white);
if (memcmp(GBFinalPixel(gb, &p), &white,
  sizeof(GBPixel)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSetFinalPixel failed");
  PBErrCatch(GenBrushErr);
}
GBPixel pix = GBGetFinalPixel(gb, &p);
if (memcmp(&pix, &white, sizeof(GBPixel)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBGetFinalPixel failed");
  PBErrCatch(GenBrushErr);
}
GBPixel blue = GBColorBlue;
GBSetFinalPixelSafe(gb, &p, &blue);
pix = GBGetFinalPixel(gb, &p);
if (memcmp(&pix, &blue, sizeof(GBPixel)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBSetFinalPixelSafe failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, 1); VecSet(&p, 1, -2);
GBSetFinalPixelSafe(gb, &p, &white);
pix = GBGetFinalPixelSafe(gb, &p);
GBPixel transparent = GBColorTransparent;
if (memcmp(&pix, &transparent, sizeof(GBPixel)) != 0) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBGetFinalPixelSafe failed");
  PBErrCatch(GenBrushErr);
}
if (GBArea(gb) != 12) {
  GenBrushErr->_type = PBErrTypeUnitTestFailed;
  sprintf(GenBrushErr->_msg, "GBArea failed");
  PBErrCatch(GenBrushErr);
}
VecSet(&p, 0, 1); VecSet(&p, 1, 2);
```

189

```
    if (GBIsPosInside(gb, &p) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBIsPosInside failed");
      PBErrCatch(GenBrushErr);
    }
    VecSet(&p, 0, 10); VecSet(&p, 1, 2);
    if (GBIsPosInside(gb, &p) == true) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBIsPosInside failed");
      PBErrCatch(GenBrushErr);
    }
    VecSet(&p, 0, 1); VecSet(&p, 1, -2);
    if (GBIsPosInside(gb, &p) == true) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBIsPosInside failed");
      PBErrCatch(GenBrushErr);
    }
    if (GBLayers(gb) != &(gb->_surf->_layers)) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSurfaceLayers failed");
      PBErrCatch(GenBrushErr);
    }
    GBEyeOrthoFree(&eye);
    GBFree(&gb);
    printf("UnitTestGenBrushGetSet OK\n");
  }

  void UnitTestGenBrushUpdate() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
    GenBrush* gb = GBCreateImage(&dim);
    GBSurface* surf = GBSurf(gb);
    GBLayer* layer = GBAddLayer(gb, &dim);
    GBToolPlotter* tool = GBToolPlotterCreate();
    GBPixel transparent = GBColorTransparent;
    GBSurfaceSetBgColor(surf, &transparent);
    GBPixel blue = GBColorBlue;
    GBInkSolid* ink = GBInkSolidCreate(&blue);
    Facoid* facoid = FacoidCreate(2);
    Shapoid* shap = (Shapoid*)facoid;
    VecFloat2D v = VecFloatCreateStatic2D();
    VecSet(&v, 0, 7.0); VecSet(&v, 1, 4.0);
    ShapoidScale(shap, (VecFloat*)&v);
    VecSet(&v, 0, 2.0); VecSet(&v, 1, 4.0);
    ShapoidTranslate(shap, (VecFloat*)&v);
    GBEyeOrtho* eye = GBEyeOrthoCreate(GBEyeOrthoViewFront);
    GBHand hand = GBHandCreateStatic(GBHandTypeDefault);
    GBAddShapoid(gb, shap, &eye, &hand, tool, ink, layer);
    GBUpdate(gb);
    unsigned char check[400] = {
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255,
      0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
```

```
        0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255,
        0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,255,255, 0,0,255,255, 0,0,255,255,
        0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
      };
    for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
      if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
        check[iPix * 4] ||
        surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
        check[iPix * 4 + 1] ||
        surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
        check[iPix * 4 + 2] ||
        surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
        check[iPix * 4 + 3]) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "GBUpdate failed");
        PBErrCatch(ShapoidErr);
      }
    }
  GBToolPlotterFree(&tool);
  GBInkSolidFree(&ink);
  ShapoidFree(&shap);
  GBEyeOrthoFree(&eye);
  GBFree(&gb);
  printf("UnitTestGenBrushUpdate OK\n");
}

void UnitTestGenBrushRender() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
  GenBrush* gb = GBCreateImage(&dim);
  GBSurface* surf = GBSurf(gb);
  GBLayer* layers[4] = {NULL};
  VecSet(&dim, 0, 5); VecSet(&dim, 1, 5);
  for (int iLayer = 0; iLayer < 4; ++iLayer) {
    layers[iLayer] = GBSurfaceAddLayer(surf, &dim);
    GBLayerSetBlendMode(layers[iLayer], GBLayerBlendModeOver);
  }
  GBLayerSetStackPos(layers[1], GBLayerStackPosInside);
  GBLayerSetStackPos(layers[2], GBLayerStackPosInside);
  GBLayerSetStackPos(layers[3], GBLayerStackPosFg);
  VecShort2D pos = VecShortCreateStatic2D();
  VecSet(&pos, 0, 1); VecSet(&pos, 1, 1);
  GBLayerSetPos(layers[0], &pos);
  VecSet(&pos, 0, 3); VecSet(&pos, 1, 3);
  GBLayerSetPos(layers[1], &pos);
  VecSet(&pos, 0, 4); VecSet(&pos, 1, 4);
  GBLayerSetPos(layers[2], &pos);
  VecSet(&pos, 0, 7); VecSet(&pos, 1, 7);
  GBLayerSetPos(layers[3], &pos);
  VecSetNull(&pos);
  GBPixel col;
  float depth;
  do {
    col = GBColorGreen;
    depth = 0.0;
    GBLayerAddPixel(layers[0], &pos, &col, depth);
    col._rgba[GBPixelRed] = VecGet(&pos, 1) * 30;
```

```
    col._rgba[GBPixelGreen] = VecGet(&pos, 0) * 30;
    col._rgba[GBPixelBlue] = 0;
    col._rgba[GBPixelAlpha] = 255 - VecGet(&pos, 1) * 30;
    depth = VecGet(&pos, 0);
    GBLayerAddPixel(layers[1], &pos, &col, depth);
    col._rgba[GBPixelRed] = 0;
    col._rgba[GBPixelGreen] = VecGet(&pos, 0) * 30;
    col._rgba[GBPixelBlue] = VecGet(&pos, 1) * 30;
    col._rgba[GBPixelAlpha] = 255 - VecGet(&pos, 0) * 30;
    depth = VecGet(&pos, 1);
    GBLayerAddPixel(layers[2], &pos, &col, depth);
    col = GBColorBlue;
    depth = 0.0;
    GBLayerAddPixel(layers[3], &pos, &col, depth);
  } while (VecStep(&pos, &dim));
  GBUpdate(gb);
  GBSetFileName(gb, "./UnitTestGenBrushRender.tga");
  GBRender(gb);
  GBFree(&gb);
  printf("UnitTestGenBrushRender OK\n");
}

void UnitTestGenBrushAddRemoveLayer() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
  GenBrush* gb = GBCreateImage(&dim);
  GBLayer* layerA = GBAddLayer(gb, &dim);
  if (layerA == NULL) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBAddLayer failed");
    PBErrCatch(GenBrushErr);
  }
  if (GSetNbElem(GBSurfaceLayers(GBSurf(gb))) != 1) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBAddLayer failed");
    PBErrCatch(GenBrushErr);
  }
  if (VecIsEqual(GBLayerDim(layerA), &dim) == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBAddLayer failed");
    PBErrCatch(GenBrushErr);
  }
  GBLayer* layerB = GBAddLayer(gb, &dim);
  GBRemoveLayer(gb, layerA);
  if ((GBLayer*)(GBSurf(gb)->_layers._head->_data) != layerB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBRemoveLayer failed");
    PBErrCatch(GenBrushErr);
  }
  GBFree(&gb);
  printf("UnitTestGenBrushAddRemoveLayer OK\n");
}

void UnitTestGenBrushGetLayerNbLayer() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
  GenBrush* gb = GBCreateImage(&dim);
  GBLayer* layers[3] = {NULL};
  for (int iLayer = 0; iLayer< 3; ++iLayer)
    layers[iLayer] = GBAddLayer(gb, &dim);
  if (GBGetNbLayer(gb) != 3) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
```

```
      sprintf(GenBrushErr->_msg, "GBGetNbLayer failed");
      PBErrCatch(GenBrushErr);
    }
    for (int iLayer = 0; iLayer< 3; ++iLayer) {
      if (GBGetLayer(gb, iLayer) != layers[iLayer]) {
        GenBrushErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GenBrushErr->_msg, "GBGetLayer failed");
        PBErrCatch(GenBrushErr);
      }
    }
    GBFree(&gb);
    printf("UnitTestGenBrushGetLayer OK\n");
}

void UnitTestGenBrushSetLayerPos() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 4); VecSet(&dim, 1, 3);
    GenBrush* gb = GBCreateImage(&dim);
    GBLayer* layers[3] = {NULL};
    for (int iLayer = 0; iLayer< 3; ++iLayer)
      layers[iLayer] = GBAddLayer(gb, &dim);
    GBSetLayerPos(gb, layers[2], 0);
    GBSetLayerPos(gb, layers[0], 2);
    if (GBGetLayer(gb, 0) != layers[2] ||
      GBGetLayer(gb, 1) != layers[1] ||
      GBGetLayer(gb, 2) != layers[0] ||
      GBLayerIsModified(layers[0]) != true ||
      GBLayerIsModified(layers[1]) != true ||
      GBLayerIsModified(layers[2]) != true) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBSetLayerPos failed");
      PBErrCatch(GenBrushErr);
    }
    GBFree(&gb);
    printf("UnitTestGenBrushSetLayerPos OK\n");
}

void UnitTestGenBrushAddLayerFromFile() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
    GenBrush* gb = GBCreateImage(&dim);
    GBSurface* surf = GBSurf(gb);
    GBLayer* layer = GBAddLayerFromFile(gb, "./ImageRef.tga");
    (void)layer;
    GBPixel transparent = GBColorTransparent;
    GBSurfaceSetBgColor(surf, &transparent);
    GBSurfaceUpdate(surf);
    unsigned char check[400] = {
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,255,0,255, 0,255,0,255,
      0,255,0,255, 0,255,0,255, 0,255,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 0,255,0,255, 0,255,0,255, 0,255,0,255,
      0,255,0,255, 0,255,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,255,0,255, 0,255,0,255, 0,0,0,255, 0,30,0,255,
      0,60,0,255, 0,90,0,255, 0,120,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
      0,255,0,255, 0,255,0,255, 26,30,0,255, 0,0,0,255, 4,34,0,255,
      7,67,0,255, 11,101,0,255, 120,184,120,255, 0,0,0,0, 0,0,0,0,
      0,255,0,255, 0,255,0,255, 46,60,0,255, 0,0,30,255, 7,34,26,255,
      14,67,23,255, 21,101,19,255, 120,184,136,255, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 148,90,90,255, 0,0,60,255, 11,34,53,255,
      21,67,46,255, 32,101,39,255, 120,184,152,255, 0,0,0,0, 0,0,0,0,
      0,0,0,0, 0,0,0,0, 184,120,120,255, 0,0,90,255, 14,34,79,255,
```

```
       28,67,69,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
       0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,120,255, 30,56,136,255,
       60,106,152,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
       0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
       0,0,255,255, 0,0,255,255, 0,0,255,255
     };
  for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
    if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
      check[iPix * 4] ||
      surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
      check[iPix * 4 + 1] ||
      surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
      check[iPix * 4 + 2] ||
      surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
      check[iPix * 4 + 3]) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBAddLayerFromFile failed");
      PBErrCatch(ShapoidErr);
    }
  }
  GBFree(&gb);
  printf("UnitTestGenBrushAddLayerFromFile OK\n");
}

void UnitTestGenBrushAddRemoveObj() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
  GenBrush* gb = GBCreateImage(&dim);
  VecFloat2D posA = VecFloatCreateStatic2D();
  VecFloat2D posB = VecFloatCreateStatic2D();
  GBEye eyeA = GBEyeCreateStatic(GBEyeTypeOrtho);
  GBEye eyeB = GBEyeCreateStatic(GBEyeTypeOrtho);
  GBHand handA = GBHandCreateStatic(GBHandTypeDefault);
  GBHand handB = GBHandCreateStatic(GBHandTypeDefault);
  GBTool toolA;
  GBTool toolB;
  GBLayer layerA;
  GBLayer layerB;
  GBInk inkA;
  GBInk inkB;
  GBAddPoint(gb, &posA, &eyeA, &handA, &toolA, &inkA, &layerA);
  GBAddPoint(gb, &posB, &eyeA, &handA, &toolA, &inkB, &layerA);
  if (GBGetNbPod(gb) != 2 ||
    ((GBObjPod*)(gb->_pods._head->data))->_srcPoint !=
    (VecFloat*)&posA ||
    ((GBObjPod*)(gb->_pods._head->_next->_data))->_srcPoint !=
    (VecFloat*)&posB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBAddPoint failed");
    PBErrCatch(GenBrushErr);
  }
  GBRemovePod(gb, &posA, NULL, NULL, NULL, NULL, NULL);
  if (GBGetNbPod(gb) != 1 ||
    ((GBObjPod*)(gb->_pods._head->_data))->_srcPoint !=
      (VecFloat*)&posB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBRemovePod failed");
    PBErrCatch(GenBrushErr);
  }
  GBAddPoint(gb, &posA, &eyeA, &handA, &toolA, &inkA, &layerA);
  GBRemovePod(gb, NULL, &eyeA, NULL, NULL, NULL, NULL);
  if (GBGetNbPod(gb) != 0) {
```

```
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBRemovePod failed");
      PBErrCatch(GenBrushErr);
  }
  GBAddPoint(gb, &posA, &eyeA, &handA, &toolA, &inkA, &layerA);
  GBAddPoint(gb, &posB, &eyeB, &handA, &toolA, &inkA, &layerA);
  GBRemovePod(gb, NULL, &eyeA, &handA, NULL, NULL, NULL);
  if (GBGetNbPod(gb) != 1 ||
    ((GBObjPod*)(gb->_pods._head->data))->_srcPoint !=
      (VecFloat*)&posB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBRemovePod failed");
    PBErrCatch(GenBrushErr);
  }
  GBAddPoint(gb, &posA, &eyeA, &handA, &toolB, &inkA, &layerA);
  GBRemovePod(gb, NULL, NULL, NULL, &toolB, NULL, NULL);
  if (GBGetNbPod(gb) != 1 ||
    ((GBObjPod*)(gb->_pods._head->data))->_srcPoint !=
      (VecFloat*)&posB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBRemovePod failed");
    PBErrCatch(GenBrushErr);
  }
  GBAddPoint(gb, &posA, &eyeA, &handB, &toolB, &inkA, &layerB);
  GBRemovePod(gb, NULL, NULL, NULL, NULL, NULL, &layerB);
  if (GBGetNbPod(gb) != 1 ||
    ((GBObjPod*)(gb->_pods._head->data))->_srcPoint !=
      (VecFloat*)&posB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBRemovePod failed");
    PBErrCatch(GenBrushErr);
  }
  GBRemovePod(gb, NULL, NULL, NULL, NULL, NULL, NULL);
  if (GBGetNbPod(gb) != 0) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBRemovePod failed");
    PBErrCatch(GenBrushErr);
  }
  GBAddPoint(gb, &posA, &eyeA, &handB, &toolB, &inkA, &layerA);
  GBAddPoint(gb, &posB, &eyeB, &handB, &toolB, &inkB, &layerB);
  GBRemovePod(gb, NULL, NULL, NULL, NULL, &inkB, NULL);
  if (GBGetNbPod(gb) != 1 ||
    ((GBObjPod*)(gb->_pods._head->data))->_srcPoint !=
      (VecFloat*)&posA) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBRemovePod failed");
    PBErrCatch(GenBrushErr);
  }
  GBFree(&gb);
  GBEyeFreeStatic(&eyeA);
  GBEyeFreeStatic(&eyeB);
  printf("UnitTestGenBrushAddRemoveObj OK\n");
}

void UnitTestGenBrushSetPod() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
  GenBrush* gb = GBCreateImage(&dim);
  VecFloat2D posA = VecFloatCreateStatic2D();
  VecFloat2D posB = VecFloatCreateStatic2D();
  GBEye eyeA = GBEyeCreateStatic(GBEyeTypeOrtho);
  GBEye eyeB = GBEyeCreateStatic(GBEyeTypeOrtho);
```

```
  GBHand handA = GBHandCreateStatic(GBHandTypeDefault);
  GBHand handB = GBHandCreateStatic(GBHandTypeDefault);
  GBTool toolA;
  GBTool toolB;
  GBLayer layerA;
  GBLayer layerB;
  GBInk inkA;
  GBInk inkB;
  GBAddPoint(gb, &posA, &eyeA, &handA, &toolA, &inkA, &layerA);
  GBAddPoint(gb, &posB, &eyeA, &handB, &toolB, &inkA, &layerB);
  GBSetPodEye(gb, &eyeB, &posA, NULL, NULL, NULL, NULL, NULL);
  if ((((GBObjPod*)(gb->_pods._head->_data))->_eye != &eyeB) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSetPodEye failed");
    PBErrCatch(GenBrushErr);
  }
  GBSetPodHand(gb, &handA, NULL, NULL, NULL, NULL, NULL, &layerB);
  if ((((GBObjPod*)(gb->_pods._head->_next->_data))->_hand != &handA) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSetPodHand failed");
    PBErrCatch(GenBrushErr);
  }
  GBSetPodTool(gb, &toolA, NULL, NULL, &handA, NULL, NULL, NULL);
  if ((((GBObjPod*)(gb->_pods._head->_data))->_tool != &toolA ||
    ((GBObjPod*)(gb->_pods._head->_next->_data))->_tool != &toolA) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSetPodTool failed");
    PBErrCatch(GenBrushErr);
  }
  GBSetPodInk(gb, &inkA, NULL, NULL, NULL, NULL, &inkB, NULL);
  if ((((GBObjPod*)(gb->_pods._head->_data))->_ink != &inkA ||
    ((GBObjPod*)(gb->_pods._head->_next->_data))->_ink != &inkA) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSetPodTool failed");
    PBErrCatch(GenBrushErr);
  }
  GBSetPodLayer(gb, &layerA, &posB, NULL, NULL, &toolA, NULL, NULL);
  if ((((GBObjPod*)(gb->_pods._head->_next->_data))->_layer != &layerA) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSetPodLayer failed");
    PBErrCatch(GenBrushErr);
  }
  GBFree(&gb);
  GBEyeFreeStatic(&eyeA);
  GBEyeFreeStatic(&eyeB);
  printf("UnitTestGenBrushSetPod OK\n");
}

void UnitTestGenBrushFlush() {
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
  GenBrush* gb = GBCreateImage(&dim);
  GBSurface* surf = GBSurf(gb);
  GBLayer* layer = GBSurfaceAddLayer(surf, &dim);
  GBLayerSetModified(layer, false);
  GBPixel blue = GBColorBlue;
  GBSurfaceSetBgColor(surf, &blue);
  GBFlush(gb);
  if (GBLayerIsModified(layer) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBSurfaceFlush failed");
    PBErrCatch(ShapoidErr);
```

196

```
    }
    for (int iPix = GBSurfaceArea(surf); iPix--;)
      if (memcmp(surf->_finalPix + iPix, &blue, sizeof(GBPixel)) != 0) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "GBSurfaceFlush failed");
        PBErrCatch(ShapoidErr);
      }
    GBFree(&gb);
    printf("UnitTestGenBrushFlush OK\n");
}

void UnitTestGenBrushTouchLayers() {
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 10); VecSet(&dim, 1, 10);
    GenBrush* gb = GBCreateImage(&dim);
    VecFloat2D posA = VecFloatCreateStatic2D();
    VecFloat2D posB = VecFloatCreateStatic2D();
    GBEye eyeA = GBEyeCreateStatic(GBEyeTypeOrtho);
    GBHand handA = GBHandCreateStatic(GBHandTypeDefault);
    GBHand handB = GBHandCreateStatic(GBHandTypeDefault);
    GBTool toolA;
    GBTool toolB;
    GBLayer* layerA = GBAddLayer(gb, &dim);
    GBLayer* layerB = GBAddLayer(gb, &dim);
    GBInk inkA;
    GBAddPoint(gb, &posA, &eyeA, &handA, &toolA, &inkA, layerA);
    GBAddPoint(gb, &posB, &eyeA, &handB, &toolB, &inkA, layerB);
    GBSurfaceSetLayersModified(GBSurf(gb), false);
    GBNotifyChangeFromObj(gb, &posA);
    if (GBLayerIsModified(layerA) == false ||
      GBLayerIsModified(layerB) == true) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBNotifyChangeFromObj failed");
      PBErrCatch(GenBrushErr);
    }
    GBSurfaceSetLayersModified(GBSurf(gb), false);
    GBNotifyChangeFromEye(gb, &eyeA);
    if (GBLayerIsModified(layerA) == false ||
      GBLayerIsModified(layerB) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBNotifyChangeFromEye failed");
      PBErrCatch(GenBrushErr);
    }
    GBSurfaceSetLayersModified(GBSurf(gb), false);
    GBNotifyChangeFromHand(gb, &handA);
    if (GBLayerIsModified(layerA) == false ||
      GBLayerIsModified(layerB) == true) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBNotifyChangeFromHand failed");
      PBErrCatch(GenBrushErr);
    }
    GBSurfaceSetLayersModified(GBSurf(gb), false);
    GBNotifyChangeFromTool(gb, &toolA);
    if (GBLayerIsModified(layerA) == false ||
      GBLayerIsModified(layerB) == true) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBNotifyChangeFromTool failed");
      PBErrCatch(GenBrushErr);
    }
    GBSurfaceSetLayersModified(GBSurf(gb), false);
    GBNotifyChangeFromInk(gb, &inkA);
    if (GBLayerIsModified(layerA) == false ||
```

```
    GBLayerIsModified(layerB) == false) {
      GenBrushErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GenBrushErr->_msg, "GBNotifyChangeFromInk failed");
      PBErrCatch(GenBrushErr);
    }
  GBFree(&gb);
  GBEyeFreeStatic(&eyeA);
  printf("UnitTestGenBrushTouchLayers OK\n");
}

void UnitTestGenBrushCreateFromFile() {
  char* fileName = "./ImageRef.tga";
  GenBrush* gb = GBCreateFromFile(fileName);
  if (gb == NULL) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBCreateFromFile failed");
    PBErrCatch(GenBrushErr);
  }
  GBSurface* surf = GBSurf(gb);
  unsigned char check[400] = {
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,255,0,255, 0,255,0,255,
    0,255,0,255, 0,255,0,255, 0,255,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,255,0,255, 0,255,0,255, 0,255,0,255,
    0,255,0,255, 0,255,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,255,0,255, 0,255,0,255, 0,0,0,255, 0,30,0,255,
    0,60,0,255, 0,90,0,255, 0,120,0,255, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,255,0,255, 0,255,0,255, 26,30,0,255, 0,0,0,255, 4,34,0,255,
    7,67,0,255, 11,101,0,255, 120,184,120,255, 0,0,0,0, 0,0,0,0,
    0,255,0,255, 0,255,0,255, 46,60,0,255, 0,0,30,255, 7,34,26,255,
    14,67,23,255, 21,101,19,255, 120,184,136,255, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 148,90,90,255, 0,0,60,255, 11,34,53,255,
    21,67,46,255, 32,101,39,255, 120,184,152,255, 0,0,0,0, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 184,120,120,255, 0,0,90,255, 14,34,79,255,
    28,67,69,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,120,255, 30,56,136,255,
    60,106,152,255, 0,0,255,255, 0,0,255,255, 0,0,255,255, 0,0,0,0,
    0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0,
    0,0,255,255, 0,0,255,255, 0,0,255,255
    };
  for (int iPix = 0; iPix < GBSurfaceArea(surf); ++iPix) {
    if (surf->_finalPix[iPix]._rgba[GBPixelRed] !=
      check[iPix * 4] ||
      surf->_finalPix[iPix]._rgba[GBPixelGreen] !=
      check[iPix * 4 + 1] ||
      surf->_finalPix[iPix]._rgba[GBPixelBlue] !=
      check[iPix * 4 + 2] ||
      surf->_finalPix[iPix]._rgba[GBPixelAlpha] !=
      check[iPix * 4 + 3]) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBCreateFromFile failed");
      PBErrCatch(ShapoidErr);
    }
  }
  GBFree(&gb);
  printf("UnitTestGenBrushCreateFromFile OK\n");
}

void UnitTestGenBrushIsSameAs() {
  char* fileName = "./ImageRef.tga";
  GenBrush* gbA = GBCreateFromFile(fileName);
  GenBrush* gbB = GBCreateFromFile(fileName);
```

```
    if (GBIsSameAs(gbA, gbB) == false) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBIsSameAs failed");
      PBErrCatch(ShapoidErr);
    }
    VecSet(GBSurfaceDim(GBSurf(gbA)), 0, 0);
    if (GBIsSameAs(gbA, gbB) == true) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBIsSameAs failed");
      PBErrCatch(ShapoidErr);
    }
    VecSet(GBSurfaceDim(GBSurf(gbA)), 0, 10);
    GBSurfaceFinalPixels(GBSurf(gbA))->_rgba[GBPixelAlpha] = 255;
    if (GBIsSameAs(gbA, gbB) == true) {
      ShapoidErr->_type = PBErrTypeUnitTestFailed;
      sprintf(ShapoidErr->_msg, "GBIsSameAs failed");
      PBErrCatch(ShapoidErr);
    }
    GBFree(&gbA);
    GBFree(&gbB);
    printf("UnitTestGenBrushIsSameAs OK\n");
}

void UnitTestGenBrushAddRemovePostProcessing() {
  GenBrush* gb = GBCreateFromFile("./GBSurfaceNormalizeHueTest.tga");
  if (GBPostProcs(gb) != &(gb->_postProcs)) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBPostProcs failed");
    PBErrCatch(ShapoidErr);
  }
  GBPostProcessing* postA = GBAddPostProcess(gb, GBPPTypeNormalizeHue);
  if (gb->_postProcs._head->_data != (void*)postA) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBAddPostProcess failed");
    PBErrCatch(ShapoidErr);
  }
  if (GBGetPostProcess(gb, 0) != (void*)postA) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBGetPostProcess failed");
    PBErrCatch(ShapoidErr);
  }
  GBUpdate(gb);
  GBSurfaceImage* ref = GBSurfaceImageCreateFromFile(
    "./GBSurfaceNormalizeHueRef.tga");
  if (GBSurfaceIsSameAs(GBSurf(gb), (GBSurface*)ref) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBUpdate failed");
    PBErrCatch(ShapoidErr);
  }
  GBRemovePostProcess(gb, postA);
  if (GSetNbElem(GBPostProcs(gb)) != 0) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBRemovePostProcess failed");
    PBErrCatch(ShapoidErr);
  }
  postA = GBAddPostProcess(gb, GBPPTypeNormalizeHue);
  postA = GBAddPostProcess(gb, GBPPTypeNormalizeHue);
  if (GBGetNbPostProcs(gb) != 2) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "GBGetNbPostProcs failed");
    PBErrCatch(ShapoidErr);
  }
```

```
      GBRemoveAllPostProcess(gb);
      if (GSetNbElem(GBPostProcs(gb)) != 0) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "GBRemoveAllPostProcess failed");
        PBErrCatch(ShapoidErr);
      }
      postA = GBAddPostProcess(gb, GBPPTypeNormalizeHue);
      GBFree(&gb);
      GBSurfaceImageFree(&ref);
      printf("UnitTestGenBrushAddRemovePostProcessing\n");
}
void UnitTestGenBrush() {
  UnitTestGenBrushCreateFree();
  UnitTestGenBrushGetSet();
  UnitTestGenBrushUpdate();
  UnitTestGenBrushRender();
  UnitTestGenBrushAddRemoveLayer();
  UnitTestGenBrushGetLayerNbLayer();
  UnitTestGenBrushSetLayerPos();
  UnitTestGenBrushUpdate();
  UnitTestGenBrushAddLayerFromFile();
  UnitTestGenBrushAddRemoveObj();
  UnitTestGenBrushSetPod();
  UnitTestGenBrushFlush();
  UnitTestGenBrushTouchLayers();
  UnitTestGenBrushCreateFromFile();
  UnitTestGenBrushIsSameAs();
  UnitTestGenBrushAddRemovePostProcessing();

  printf("UnitTestGenBrush OK\n");
}

void UnitTestAll() {
  UnitTestGBPixel();
  UnitTestGBLayer();
  UnitTestGBPostProcessing();
  UnitTestGBSurface();
  UnitTestGBSurfaceImage();
  UnitTestGBEye();
  UnitTestGBHand();
  UnitTestGBTool();
  UnitTestGBInk();
  UnitTestGBObjPod();
  UnitTestGenBrush();
  printf("UnitTestAll OK\n");
}

#if BUILDWITHGRAPHICLIB == 0

  int main() {
    UnitTestAll();
    // Return success code
    return 0;
  }

#elif BUILDWITHGRAPHICLIB == 1

  void UnitTestPaintSurface(GBSurface* surf, VecShort2D* dim,
    int green) {
    // Declare a vector to memorize position in the surface
    VecShort2D pos = VecShortCreateStatic2D();
    // Set the surface to a shade of red and blue
```

```
    do {
      GBPixel pixel;
      pixel._rgba[GBPixelAlpha] = 255;
      pixel._rgba[GBPixelRed] = MIN(255, VecGet(&pos, 0));
      pixel._rgba[GBPixelGreen] = MIN(255, green);
      pixel._rgba[GBPixelBlue] = MIN(255, VecGet(&pos, 1));
      GBSurfaceSetFinalPixel(surf, &pos, &pixel);
    } while (VecStep(&pos, dim));
}

gint UnitTestGBSurfaceAppCB(gpointer data) {
  // Declare a variable to convert the data into the GBSurfaceApp
  GBSurfaceApp* GBApp = (GBSurfaceApp*)data;
  // Paint again the surface
  UnitTestPaintSurface((GBSurface*)GBApp,
    GBSurfaceDim((GBSurface*)GBApp), 0);
  // Refresh the displayed surface
  GBSurfaceAppRefresh(GBApp);
  // Kill the app
  GBSurfaceAppClose(GBApp);
  // Return false to continue the callback chain
  return FALSE;
}

void UnitTestGBSurfaceApp() {
  // Declare variable ot memorize the parameters of the app
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 300); VecSet(&dim, 1, 200);
  char* title = "UnitTestGBSurfaceApp";
  // Create the app
  GenBrush* app = GBCreateApp(&dim, title);
  // Set the idle function of the surface
  GBSetIdle(app, &UnitTestGBSurfaceAppCB, 1000);
  // Paint some content on the surface
  UnitTestPaintSurface(app->_surf, &dim, 0);
  // Render the app
  bool status = GBRender(app);
  // If the render of the app failed
  if (status == false) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "GBSurfaceAppCreate failed");
    PBErrCatch(GenBrushErr);
  }
  // Free the memory used by the app
  GBFree(&app);
  printf("UnitTestGBSurfaceApp OK\n");
}

GtkWidget* UnitTestGBSurfaceWidgetWindow;
gint UnitTestGBSurfaceWidgetCB(gpointer data) {
  static int green = 0;
  // Declare a variable to convert the data into the GenBrush
  GenBrush* GBWidget = (GenBrush*)data;
  // Paint again the surface
  UnitTestPaintSurface(GBWidget->_surf,
    GBSurfaceDim(GBWidget->_surf), green);
  ++green;
  // Refresh the displayed surface
  GBRender(GBWidget);
  if (green == 255)
    gtk_window_close(GTK_WINDOW(UnitTestGBSurfaceWidgetWindow));
  // Return true to stop the callback chain
```

```
    return TRUE;
}

gboolean UnitTestGBSurfaceWidgetCloseCB(GtkWidget* widget,
  GdkEventConfigure* event, gpointer data) {
    (void)widget;(void)event;
  // Declare a variable to convert the data into the GenBrush
  GenBrush* GBWidget = (GenBrush*)data;
  // Free memory
  GBFree(&GBWidget);
  // Return false to continue the callback chain
  return FALSE;
}

void UnitTestGBSurfaceWidgetActivateCB(GtkApplication* app,
  gpointer user_data) {
  (void)user_data;
  // Create a GTK application
  GtkWidget* window = gtk_application_window_new(app);
  UnitTestGBSurfaceWidgetWindow = window;
  // Set the title
  gtk_window_set_title(GTK_WINDOW(window), "UnitTestGBSurfaceWidget");
  // Declare a variable to memorize the size
  VecShort2D dim = VecShortCreateStatic2D();
  VecSet(&dim, 0, 300); VecSet(&dim, 1, 200);
  // Set the size
  gtk_window_set_default_size(GTK_WINDOW(window), VecGet(&dim, 0),
    VecGet(&dim, 1));
  // Avoid window resizing
  gtk_window_set_resizable(GTK_WINDOW(window), false);
  // Create the GenBrush with a GBSurfaceWidget
  GenBrush* GBWidget = GBCreateWidget(&dim);
  // Get the widget of the drawing area
  GtkWidget* drawingArea = GBGetGtkWidget(GBWidget);
  // Add the widget to the window
  gtk_container_add(GTK_CONTAINER(window), drawingArea);
  // Connect to the delete event
  g_signal_connect(window, "delete-event",
    G_CALLBACK(UnitTestGBSurfaceWidgetCloseCB), GBWidget);
  // Connect the idle function
  g_timeout_add(40, UnitTestGBSurfaceWidgetCB, GBWidget);
  // Display the window
  gtk_widget_show_all(window);
}

void UnitTestGBSurfaceWidget(int argc, char** argv) {
  // Create a GTK application
  GtkApplication* app =
    gtk_application_new(NULL, G_APPLICATION_FLAGS_NONE);
  // Connect the activate callback
  g_signal_connect(app, "activate",
    G_CALLBACK(UnitTestGBSurfaceWidgetActivateCB), NULL);
  // Run the application
  int status = g_application_run(G_APPLICATION(app), argc, argv);
  // If the application failed
  if (status != 0) {
    GenBrushErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenBrushErr->_msg, "Application failed");
    PBErrCatch(GenBrushErr);
  }
  // Unreference the application
  g_object_unref(app);
```

```
    printf("UnitTestGBSurfaceWidget OK\n");
  }

  void UnitTestAllGTK(int argc, char** argv) {
    UnitTestGBSurfaceApp();
    UnitTestGBSurfaceWidget(argc, argv);
    printf("UnitTestAllGTK OK\n");
  }

  int main(int argc, char** argv) {
    UnitTestAll();
    UnitTestAllGTK(argc, argv);
    // Return success code
    return 0;
  }

#endif
```

6 Unit tests output

```
UnitTestGBPixelBlendNormal OK
UnitTestGBPixelBlendOver OK
UnitTestGBPixel OK
UnitTestGBLayerCreateFree OK
UnitTestGBLayerArea OK
UnitTestGBLayerGetSet OK
UnitTestGBLayerCreateFromFile OK
UnitTestGBLayerGetBoundaryInSurface OK
UnitTestGBLayer OK
UnitTestGBPostProcessingCreateFree OK
UnitTestGBPostProcessingGetSet OK
UnitTestGBPostProcessing OK
UnitTestGBSurfaceCreateFree OK
UnitTestGBSurfaceGetSet OK
UnitTestGBSurfaceArea OK
UnitTestGBSurfaceClone OK
UnitTestGBSurfaceAddRemoveLayer OK
UnitTestGBSurfaceGetLayer OK
UnitTestGBSurfaceSetLayersModified OK
UnitTestGBSurfaceSetLayerPos OK
UnitTestGBSurfaceGetModifiedArea OK
UnitTestGBSurfaceUpdate OK
UnitTestGBSurfaceAddLayerFromFile OK
UnitTestGBSurfaceFlush OK
UnitTestGBSurfaceIsSameAs OK
UnitTestGBSurfaceNormalizeHue OK
UnitTestGBSurface OK
UnitTestGBSurfaceImageCreateFree OK
UnitTestGBSurfaceImageGetSet OK
UnitTestGBSurfaceImageClone OK
UnitTestGBSurfaceImageSave OK
UnitTestGBSurfaceImageCreateFromFile OK
UnitTestGBSurfaceImage OK
UnitTestGBEyeCreateFree OK
UnitTestGBEyeGetSet OK
UnitTestGBEyeOrthoCreateFree OK
UnitTestGBEyeOrthoGetSet OK
UnitTestGBEyeOrthoProcessPoint OK
```

```
UnitTestGBEyeOrthoProcessCurve OK
UnitTestGBEyeOrthoProcessShapoid OK
UnitTestGBEyeOrtho OK
UnitTestGBEyeIsometricCreateFree OK
UnitTestGBEyeIsometricGetSet OK
UnitTestGBEyeIsometricProcessPoint OK
UnitTestGBEyeIsometricProcessCurve OK
UnitTestGBEyeIsometricProcessShapoid OK
UnitTestGBEyeIsometric OK
UnitTestGBEye OK
UnitTestGBHandCreateFree OK
UnitTestGBHandCreateFree OK
UnitTestGBHandDefaultCreateFree OK
UnitTestGBHandDefaultProcess OK
UnitTestGBHandDefault OK
UnitTestGBHand OK
UnitTestGBToolCreateFree OK
UnitTestGBToolGetSet OK
UnitTestGBToolPlotterCreateFree OK
UnitTestGBToolPlotterDrawPoint OK
UnitTestGBToolPlotterDrawFacoid OK
UnitTestGBToolPlotterDrawPyramidoid OK
UnitTestGBToolPlotterDrawSpheroid OK
UnitTestGBToolPlotterDrawFacoid3D OK
UnitTestGBToolPlotterDrawSCurve OK
UnitTestGBToolPlotter OK
UnitTestGBTool OK
UnitTestGBInkSolidCreateFree OK
UnitTestGBInkSolidGetSet OK
UnitTestGBInkSolid OK
UnitTestGBInk OK
UnitTestGBObjPodCreateFree OK
UnitTestGBObjPodGetSet OK
UnitTestGBObjPod OK
UnitTestGenBrushCreateFree OK
UnitTestGenBrushGetSet OK
UnitTestGenBrushUpdate OK
UnitTestGenBrushRender OK
UnitTestGenBrushAddRemoveLayer OK
UnitTestGenBrushGetLayer OK
UnitTestGenBrushSetLayerPos OK
UnitTestGenBrushUpdate OK
UnitTestGenBrushAddLayerFromFile OK
UnitTestGenBrushAddRemoveObj OK
UnitTestGenBrushSetPod OK
UnitTestGenBrushFlush OK
UnitTestGenBrushTouchLayers OK
UnitTestGenBrushCreateFromFile OK
UnitTestGenBrushIsSameAs OK
UnitTestGenBrushAddRemovePostProcessing
UnitTestGenBrush OK
UnitTestAll OK
```
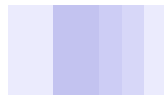
GBSurfaceImageSave.tga:
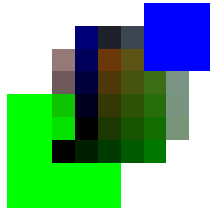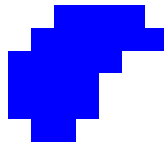
GBToolPlotterDrawSCurve.tga:
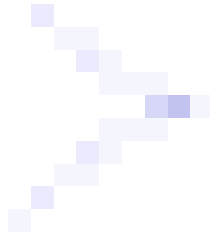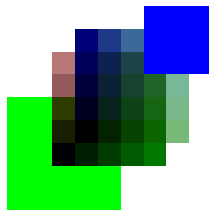

GBToolPlotterDrawFacoid3D.tga:

GBToolPlotterDrawSpheroid.tga:

GBToolPlotterDrawFacoid.tga:
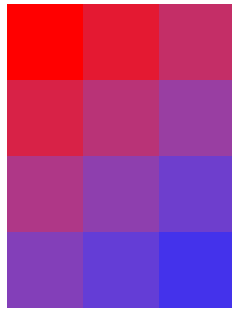

ImageRef.tga:
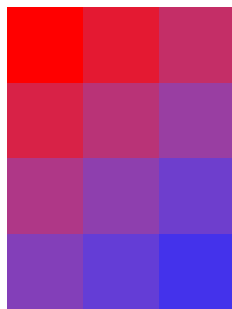

GBToolPlotterDrawPoint.tga:

GBToolPlotterDrawPyramidoid.tga:

testBottomLeft.tga:

testTopLeft.tga:

GBSurfaceNormalizeHueTest.tga:

GBSurfaceNormalizeHueRef.tga: