Shapoid

P. Baillehache

November 22, 2017

Contents

Introduction

Shapoid is a C library providing the `Shapoid` structure and its functions which can be used to manipulate Shapoid objects (see next section for details).

1 Definitions

A Shapoid is a geometry defined by its dimension $D \in \mathbb{N}_+^*$ equals to the number of dimensions of the space it exists in, its position $\overrightarrow{P}$, and its axis $(\overrightarrow{A_0}, \overrightarrow{A_1}, ..., \overrightarrow{A_{D-1}})$. A_i and P are vectors of dimension D. In what follows I'll use I as notation for the interval $[0, D-1]$ for simplification.

Shapoids are classified in three groups: Facoid, Pyramidoid and Spheroid. The volume of a Shapoid is defined by, for a Facoid:

$$\left\{\sum_{i\in I} v_i\overrightarrow{A_i} + \overrightarrow{P}\right\}, v_i \in [0.0, 1.0] \tag{1}$$

for a Pyramidoid:

$$\left\{\sum_{i\in I} v_i\overrightarrow{A_i} + \overrightarrow{P}\right\}, v_i \in [0.0, 1.0], \sum_{i\in I} v_i \leq 1.0 \tag{2}$$

and for a Spheroid:

$$\begin{aligned}\left\{\textstyle\sum_{i\in I}(v_i - 0.5)\overrightarrow{A_i} + \overrightarrow{P}\right\}, \\ v_i \in [0.0, 1.0], \quad \textstyle\sum_{i\in I}(v_i - 0.5)^2 \leq 0.25\end{aligned} \tag{3}$$

1.1 Transformation

A translation of a Shapoid by $\overrightarrow{T}$ is obtained as follow:

$$\left(\overrightarrow{P}, \left\{\overrightarrow{A_i}\right\}_{i\in I}\right) \mapsto \left(\overrightarrow{P} + \overrightarrow{T}, \left\{\overrightarrow{A_i}\right\}_{i\in I}\right) \tag{4}$$

A scale of a Shapoid by $\overrightarrow{S}$ is obtained as follow:

$$\left(\overrightarrow{P}, \left\{\overrightarrow{A_i}\right\}_{i\in I}\right) \mapsto \left(\overrightarrow{P}, \left\{\overrightarrow{A'_i}\right\}_{i\in I}\right) \tag{5}$$

where

$$\overrightarrow{A'_i} = S_i\overrightarrow{A_i} \tag{6}$$

For Shapoid whose dimension D is equal to 2, a rotation by angle θ is obtained as follow:

$$\left(\overrightarrow{P}, \overrightarrow{A_0}, \overrightarrow{A_1}\right) \mapsto \left(\overrightarrow{P}, \overrightarrow{A'_0}, \overrightarrow{A'_1}\right) \tag{7}$$

where

$$\overrightarrow{A'_i} = \left[\begin{array}{cc} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{array}\right]\overrightarrow{A_i} \tag{8}$$

1.2 Shapoid's coordinate system

The Shapoid's coordinate system is the system having $\overrightarrow{P}$ as origin and $\overrightarrow{A_i}$ as axis. One can change from the Shapoid's coordinate system ($\overrightarrow{X^S}$) to the standard coordinate system ($\overrightarrow{X}$) as follow:

$$\overrightarrow{X} = \left[\left(\overrightarrow{A_0}\right)\left(\overrightarrow{A_1}\right)...\left(\overrightarrow{A_{D-1}}\right)\right]\overrightarrow{X^S} + \overrightarrow{P} \tag{9}$$

and reciprocally, from the standard coordinate system to the Shapoid's coordinate system:

$$\overrightarrow{X^S} = \left[\left(\overrightarrow{A_0}\right)\left(\overrightarrow{A_1}\right)...\left(\overrightarrow{A_{D-1}}\right)\right]^{-1}\left(\overrightarrow{X} - \overrightarrow{P}\right) \tag{10}$$

1.3 Insideness

$\overrightarrow{X}$ is inside the Shapoid S if, for a Facoid:

$$\forall i \in I, 0.0 \leq X_i^S \leq 1.0 \tag{11}$$

for a Pyramidoid:

$$\begin{cases} \forall i \in I, 0.0 \leq X_i^S \leq 1.0 \\ \sum_{i \in I} X_i^S \leq 1.0 \end{cases} \tag{12}$$

for a Spheroid:

$$\left|\left|\overrightarrow{X^S}\right|\right| \leq 0.5 \tag{13}$$

1.4 Bounding box

A bounding box of a Shapoid is a Facoid whose axis are colinear to axis of the standard coordinate system, and including the Shapoid in its volume. While the smallest possible bounding box can be easily obtained for Facoid and Pyramidoid, it's more complicate for Spheroid. Then we will consider for the Spheroid the bounding box of the equivalent Facoid $\left(\overrightarrow{P} - \sum_{i \in I}\left(0.5 * \overrightarrow{A_i}\right), \left\{\overrightarrow{A_i}\right\}_{i \in I}\right)$ which gives the smallest bounding box when axis of the Spheroid are colinear to axis of the standard coordinate system and a bounding box slightly too large when not colinear.
The bounding box is defined as follow, for a Facoid:

$$\left(\overrightarrow{P'}, \left\{\overrightarrow{A'_i}\right\}_{i \in I}\right) \tag{14}$$

where

$$\left\{\begin{array}{l} P'_i = P_i + \sum_{j\in I^-} A_{ji} \\ A'_{ij} = 0.0, i \neq j \\ A'_{ij} = \sum_{k\in I^+} A_{kj} - \sum_{k\in I^-} A_{kj}, i = j \end{array}\right. \tag{15}$$

and, I^+ and I^- are the subsets of I such as $\forall j \in I^+, A_{ij} \geq 0.0$ and $\forall j \in I^-, A_{ij} < 0.0$.

for a Pyramidoid:

$$\left(\overrightarrow{P'}, \left\{\overrightarrow{A'_i}\right\}_{i\in I}\right) \tag{16}$$

where

$$\left\{\begin{array}{l} P'_i = P_i + Min\left(Min_{j\in I}(A_{ji}), 0.0\right) \\ A'_{ij} = 0.0, i \neq j \\ A'_{ij} = Max_{k\in I}(A_{kj}) - Min\left(Min_{k\in I}(A_{kj}), 0.0\right), i = j \end{array}\right. \tag{17}$$

1.5 Depth and Center

Depth $\mathbf{D}_S(\overrightarrow{X})$ of position $\overrightarrow{X}$ a Shapoid S is a value ranging from 0.0 if $\overrightarrow{X}$ is on the surface of the Shapoid, to 1.0 if $\overrightarrow{X}$ is at the farthest location from the surface inside the Shapoid. Depth is by definition equal to 0.0 if $\overrightarrow{X}$ is outside the Shapoid. Depth is continuous and derivable on the volume of the Shapoid. It is defined by, for a Facoid:

$$\mathbf{D}_S(\overrightarrow{X}) = \prod_{i\in I}\left(1.0 - 4.0 * (0.5 - X_i^S)^2\right) \tag{18}$$

for a Pyramidoid:

$$\mathbf{D}_S(\overrightarrow{X}) = \prod_{i\in I}\left(1.0 - 4.0 * \left(0.5 - \frac{X_i^S}{1.0 - \sum_{j\in I-\{i\}} X_j^S}\right)^2\right) \tag{19}$$

and for a Spheroid:

$$\mathbf{D}_S(\overrightarrow{X}) = 1.0 - 2.0 * \left|\left|\overrightarrow{X^S}\right|\right| \tag{20}$$

The maximum depth is obtained at $\overrightarrow{C}$ such as, for a Facoid:

$$\forall i \in I, C_i^S = 0.5 \tag{21}$$

for a Pyramidoid:

$$\forall i \in I, C_i^S = \frac{1}{D+1} \tag{22}$$

for a Spheroid:

$$\forall i \in I, C_i^S = 0.0 \tag{23}$$

$\overrightarrow{C}$ is called the center of the Shapoid.

2 Interface

```
// ============ SHAPOID.H ================

#ifndef SHAPOID_H
#define SHAPOID_H

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

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pbmath.h"
#include "gset.h"
struct Shapoid;
typedef struct Shapoid Shapoid;
struct SCurve;
typedef struct SCurve SCurve;
#include "bcurve.h"

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

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

void ShapoidGetBoundingBoxUnsupported(void*t, ...);
#define ShapoidGetBoundingBox(T) _Generic((T), \
  Shapoid*: ShapoidGetBoundingBoxThat, \
  GSet*: ShapoidGetBoundingBoxSet, \
  default: ShapoidGetBoundingBoxUnsupported)(T)


// -------------- Shapoid

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

#define SpheroidCreate(D) ShapoidCreate(D, ShapoidTypeSpheroid)
#define FacoidCreate(D) ShapoidCreate(D, ShapoidTypeFacoid)
#define PyramidoidCreate(D) ShapoidCreate(D, ShapoidTypePyramidoid)

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

typedef enum ShapoidType {
  ShapoidTypeInvalid, ShapoidTypeFacoid, ShapoidTypeSpheroid,
  ShapoidTypePyramidoid
} ShapoidType;
// Don’t forget to update ShapoidTypeString in pbmath.c when adding
// new type

typedef struct Shapoid {
  // Position of origin
  VecFloat *_pos;
  // Dimension
  int _dim;
  // Vectors defining faces
  VecFloat **_axis;
  // Type of Shapoid
  ShapoidType _type;
  // Linear sytem used to import coordinates
  EqLinSys *_eqLinSysImport;
} Shapoid;

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

// Create a Shapoid of dimension ’dim’ and type ’type’, default values:
// _pos = null vector
// _axis[d] = unit vector along dimension d
// Return NULL if arguments are invalid or malloc failed
Shapoid* ShapoidCreate(int dim, ShapoidType type);

// Clone a Shapoid
// Return NULL if couldn’t clone
Shapoid* ShapoidClone(Shapoid *that);

// Free memory used by a Shapoid
// Do nothing if arguments are invalid
void ShapoidFree(Shapoid **that);

// Load the Shapoid from the stream
// If the VecFloat is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can’t allocate memory
// 3: invalid data
// 4: fscanf error
int ShapoidLoad(Shapoid **that, FILE *stream);

// Save the Shapoid to the stream
// Return 0 upon success, or
// 1: invalid arguments
// 2: fprintf error
int ShapoidSave(Shapoid *that, FILE *stream);

// Print the Shapoid on ’stream’
// Do nothing if arguments are invalid
void ShapoidPrint(Shapoid *that, FILE *stream);

// Get the dimension of the Shapoid
// Return 0 if arguments are invalid
int ShapoidGetDim(Shapoid *that);

// Get the type of the Shapoid
// Return ShapoidTypeInvalid if arguments are invalid
ShapoidType ShapoidGetType(Shapoid *that);

// Get the type of the Shapoid as a string
// Return a pointer to a constant string (not to be freed)
// Return the string for ShapoidTypeInvalid if arguments are invalid
const char* ShapoidGetTypeAsString(Shapoid *that);

// Return a VecFloat equal to the position of the Shapoid
// Return NULL if arguments are invalid
VecFloat* ShapoidGetPos(Shapoid *that);

// Return a VecFloat equal to the 'dim'-th axis of the Shapoid
// Return NULL if arguments are invalid
VecFloat* ShapoidGetAxis(Shapoid *that, int dim);

// Set the position of the Shapoid to 'pos'
// Do nothing if arguments are invalid
void ShapoidSetPos(Shapoid *that, VecFloat *pos);

// Set the 'dim'-th axis of the Shapoid to 'v'
// Do nothing if arguments are invalid
void ShapoidSetAxis(Shapoid *that, int dim, VecFloat *v);

// Translate the Shapoid by 'v'
// Do nothing if arguments are invalid
void ShapoidTranslate(Shapoid *that, VecFloat *v);

// Scale the Shapoid by 'v' (each axis is multiplied by v[iAxis])
// Do nothing if arguments are invalid
void ShapoidScale(Shapoid *that, VecFloat *v);

// Scale the Shapoid by 'v' (each axis is multiplied by v[iAxis])
// and translate the Shapoid such as its center after scaling
// is at the same position than before scaling
// Do nothing if arguments are invalid
void ShapoidGrow(Shapoid *that, VecFloat *v);

// Rotate the Shapoid of dimension 2 by 'theta' (in radians, CCW)
// Do nothing if arguments are invalid
void ShapoidRotate2D(Shapoid *that, float theta);

// Convert the coordinates of 'pos' from standard coordinate system
// toward the Shapoid coordinates system
// Return null if the arguments are invalid
VecFloat* ShapoidImportCoord(Shapoid *that, VecFloat *pos);

// Convert the coordinates of 'pos' from the Shapoid coordinates system
// toward standard coordinate system
// Return null if the arguments are invalid
VecFloat* ShapoidExportCoord(Shapoid *that, VecFloat *pos);

// Return true if 'pos' is inside the Shapoid
// Else return false
bool ShapoidIsPosInside(Shapoid *that, VecFloat *pos);

// Get a bounding box of the Shapoid. The bounding box is aligned
// on the standard coordinate system (its axis are colinear with
// the axis of the standard coordinate system).
// The bounding box is returned as a Facoid, which position is
// at the minimum value along each axis.
// Return null if the argument are invalid.
Shapoid* ShapoidGetBoundingBoxThat(Shapoid *that);

// Get the bounding box of a set of Facoid. The bounding box is aligned
// on the standard coordinate system (its axis are colinear with
// the axis of the standard coordinate system).
// The bounding box is returned as a Facoid, which position is
// at the minimum value along each axis.
// Return null if the arguments are invalid or the shapoid in the set
// don’t have all the same dimension.
Shapoid* ShapoidGetBoundingBoxSet(GSet *set);

// Get a SCurve approximating the Shapoid ’that’
// ’that’ must be of dimension 2
// Return null if arguments are invalid
SCurve* Shapoid2SCurve(Shapoid *that);

// Get the depth value in the Shapoid of ’pos’
// The depth is defined as follow: the point with depth equals 1.0 is
// the farthest point from the surface of the Shapoid (inside it),
// points with depth equals to 0.0 are point on the surface of the
// Shapoid. Depth is continuous and derivable over the volume of the
// Shapoid
// Return 0.0 if arguments are invalid, or pos is outside the Shapoid
float ShapoidGetPosDepth(Shapoid *that, VecFloat *pos);

// Get the center of the shapoid in standard coordinate system
// Return null if arguments are invalid
VecFloat* ShapoidGetCenter(Shapoid *that);

// Get the percentage of ’tho’ included ’that’ (in [0.0, 1.0])
// 0.0 -> ’tho’ is completely outside of ’that’
// 1.0 -> ’tho’ is completely inside of ’that’
// ’that’ and ’tho’ must me of same dimensions
// Return 0.0 if the arguments are invalid or something went wrong
float ShapoidGetCoverage(Shapoid *that, Shapoid *tho);

#endif
```

3 Code

```
// ============ SHAPOID.C ================

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

#include "shapoid.h"

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

// -------------- Shapoid

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

const char *ShapoidTypeString[4] = {
  (const char*)"InvalidShapoid", (const char*)"Facoid",
  (const char*)"Spheroid", (const char*)"Pyramidoid"};

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

// Update the system of linear equation used to import coordinates
// Do nothing if arguments are invalid or memory allocation failed
void ShapoidUpdateEqLinSysImport(Shapoid *that);

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

```c
// Create a Shapoid of dimension 'dim' and type 'type', default values:
// _pos = null vector
// _axis[d] = unit vector along dimension d
// Return NULL if arguments are invalid or malloc failed
Shapoid* ShapoidCreate(int dim, ShapoidType type) {
  // Check argument
  if (dim < 0 || type == ShapoidTypeInvalid)
    return NULL;
  // Declare a vector used for initialisation
  VecShort *d = VecShortCreate(2);
  if (d == NULL)
    return NULL;
  // Declare a identity matrix used for initialisation
  VecSet(d, 0, dim);
  VecSet(d, 1, dim);
  MatFloat *mat = MatFloatCreate(d);
  VecFree(&d);
  if (mat == NULL)
    return NULL;
  MatFloatSetIdentity(mat);
  // Allocate memory
  Shapoid *that = (Shapoid*)malloc(sizeof(Shapoid));
  // If we could allocate memory
  if (that != NULL) {
    // Init pointers
    that->_pos = NULL;
    that->_axis = NULL;
    that->_eqLinSysImport = NULL;
    // Set the dimension and type
    that->_type = type;
    that->_dim = dim;
    // Allocate memory for position
    that->_pos = VecFloatCreate(dim);
    // If we couldn't allocate memory
    if (that->_pos == NULL) {
      MatFree(&mat);
      ShapoidFree(&that);
      return NULL;
    }
    // Allocate memory for array of axis
    that->_axis = (VecFloat**)malloc(sizeof(VecFloat*) * dim);
    if (that->_axis == NULL) {
      MatFree(&mat);
      ShapoidFree(&that);
      return NULL;
    }
    for (int iAxis = dim; iAxis--;)
      that->_axis[iAxis] = NULL;
    // Allocate memory for each axis
    for (int iAxis = 0; iAxis < dim; ++iAxis) {
      // Allocate memory for position
      that->_axis[iAxis] = VecFloatCreate(dim);
      // If we couldn't allocate memory
      if (that->_axis[iAxis] == NULL) {
        MatFree(&mat);
        ShapoidFree(&that);
        return NULL;
      }
      // Set value of the axis
      VecSet(that->_axis[iAxis], iAxis, 1.0);
    }
    // Create the linear system for coordinate importation
    that->_eqLinSysImport = EqLinSysCreate(mat, NULL);
    if (that->_eqLinSysImport == NULL) {
      MatFree(&mat);
      ShapoidFree(&that);
      return NULL;
    }
  }
  // Free memory
  MatFree(&mat);
  // Return the new Shapoid
  return that;
}

// Update the system of linear equation used to import coordinates
// Do nothing if arguments are invalid or memory allocation failed
void ShapoidUpdateEqLinSysImport(Shapoid *that) {
  // Check argument
  if (that == NULL)
    return;
  VecShort *dim = VecShortCreate(2);
  if (dim == NULL)
    return;
  // Set a pointer to the matrix in the EqLinSys
  MatFloat *mat = MatClone(that->_eqLinSysImport->_M);
  // Set the values of the matrix
  for (VecSet(dim, 0, 0); VecGet(dim, 0) < that->_dim;
    VecSet(dim, 0, VecGet(dim, 0) + 1)) {
    for (VecSet(dim, 1, 0); VecGet(dim, 1) < that->_dim;
      VecSet(dim, 1, VecGet(dim, 1) + 1)) {
      MatSet(mat, dim, VecGet(that->_axis[VecGet(dim, 0)],
        VecGet(dim, 1)));
    }
  }
  // Update the EqLinSys
  EqLinSysSetM(that->_eqLinSysImport, mat);
  // Free memory
  MatFree(&mat);
  VecFree(&dim);
}

// Clone a Shapoid
// Return NULL if couldn't clone
Shapoid* ShapoidClone(Shapoid *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Create a clone
  Shapoid *clone = ShapoidCreate(that->_dim, that->_type);
  if (clone != NULL) {
    // Set the position and axis of the clone
    ShapoidSetPos(clone, that->_pos);
    for (int iAxis = clone->_dim; iAxis--;)
      ShapoidSetAxis(clone, iAxis, that->_axis[iAxis]);
    // Clone the EqLinSys
    EqLinSysFree(&(clone->_eqLinSysImport));
    clone->_eqLinSysImport = EqLinSysClone(that->_eqLinSysImport);
  }
  // Return the clone
  return clone;
}

// Free memory used by a Shapoid
// Do nothing if arguments are invalid
void ShapoidFree(Shapoid **that) {
  // Check argument
  if (that == NULL || *that == NULL)
    return;
  // Free memory
  for (int iAxis = 0; iAxis < (*that)->_dim; ++iAxis)
    VecFree((*that)->_axis + iAxis);
  free((*that)->_axis);
  VecFree(&((*that)->_pos));
  EqLinSysFree(&((*that)->_eqLinSysImport));
  free(*that);
  *that = NULL;
}

// Load the Shapoid from the stream
// If the VecFloat is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
int ShapoidLoad(Shapoid **that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // If 'that' is already allocated
  if (*that != NULL) {
    // Free memory
    ShapoidFree(that);
  }
  // Read the dimension and type
  int dim;
  int ret = fscanf(stream, "%d", &dim);
  // If we coudln't fscanf
  if (ret == EOF)
    return 4;
  if (dim <= 0)
    return 3;
  ShapoidType type;
  ret = fscanf(stream, "%u", &type);
  // If we coudln't fscanf
  if (ret == EOF)
    return 4;
  // Allocate memory
  *that = ShapoidCreate(dim, type);
  // If we coudln't allocate memory
  if (*that == NULL) {
    return 2;
  }
  // Read the values
  ret = VecFloatLoad(&((*that)->_pos), stream);
  if (ret != 0)
    return ret;
  for (int iAxis = 0; iAxis < dim; ++iAxis) {
    ret = VecFloatLoad((*that)->_axis + iAxis, stream);
    if (ret != 0)
      return ret;
  }
  // Update the EqLinSys
  ShapoidUpdateEqLinSysImport(*that);
  // Return success code
  return 0;
}

// Save the Shapoid to the stream
// Return 0 upon success, or
// 1: invalid arguments
// 2: fprintf error
int ShapoidSave(Shapoid *that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // Save the dimension and type
  int ret = fprintf(stream, "%d %u\n", that->_dim, that->_type);
  // If we coudln't fprintf
  if (ret < 0)
    return 2;
  // Save the position and axis
  ret = VecFloatSave(that->_pos, stream);
  if (ret != 0)
    return ret;
  for (int iAxis = 0; iAxis < that->_dim; ++iAxis) {
    ret = VecFloatSave(that->_axis[iAxis], stream);
    if (ret != 0)
      return ret;
  }
  // Return success code
  return 0;
}

// Print the Shapoid on 'stream'
// Do nothing if arguments are invalid
void ShapoidPrint(Shapoid *that, FILE *stream) {
  // Check arugments
  if (that == NULL || stream == NULL)
    return;
  // Print the Shapoid
  fprintf(stream, "Type: %s\n", ShapoidTypeString[that->_type]);
  fprintf(stream, "Dim: %d\n", that->_dim);
  fprintf(stream, "Pos: ");
  VecPrint(that->_pos, stream);
  fprintf(stream, "\n");
  for (int iAxis = 0; iAxis < that->_dim; ++iAxis) {
    fprintf(stream, "Axis(%d): ", iAxis);
    VecPrint(that->_axis[iAxis], stream);
    fprintf(stream, "\n");
  }
}

// Get the dimension of the Shapoid
// Return 0 if arguments are invalid
int ShapoidGetDim(Shapoid *that) {
  // Check arugments
  if (that == NULL)
    return 0;
  // Return the dimension
  return that->_dim;
}

// Get the dimension of the Shapoid
// Return 0 if arguments are invalid
ShapoidType ShapoidGetType(Shapoid *that) {
  // Check arugments
  if (that == NULL)
    return 0;
  // Return the type
  return that->_type;
}

// Get the type of the Shapoid as a string
// Return a pointer to a constant string (not to be freed)
// Return the string for ShapoidTypeInvalid if arguments are invalid
const char* ShapoidGetTypeAsString(Shapoid *that) {
  // Check arugments
  if (that == NULL)
    return ShapoidTypeString[ShapoidTypeInvalid];
  // Return the type
  return ShapoidTypeString[that->_type];
}

// Return a VecFloat equal to the position of the Shapoid
// Return NULL if arguments are invalid
VecFloat* ShapoidGetPos(Shapoid *that) {
  // Check arugments
  if (that == NULL)
    return NULL;
  // Return a clone of the position
  return VecClone(that->_pos);
}

// Return a VecFloat equal to the 'dim'-th axis of the Shapoid
// Return NULL if arguments are invalid
VecFloat* ShapoidGetAxis(Shapoid *that, int dim) {
  // Check arugments
  if (that == NULL || dim < 0 || dim >= that->_dim)
    return NULL;
  // Return a clone of the axis
  return VecClone(that->_axis[dim]);
}

// Set the position of the Shapoid to 'pos'
// Do nothing if arguments are invalid
void ShapoidSetPos(Shapoid *that, VecFloat *pos) {
  // Check arugments
  if (that == NULL || pos == NULL)
    return;
  // Set the position
  VecCopy(that->_pos, pos);
}

// Set the 'dim'-th axis of the Shapoid to 'v'
// Do nothing if arguments are invalid
void ShapoidSetAxis(Shapoid *that, int dim, VecFloat *v) {
  // Check arugments
  if (that == NULL || v == NULL)
    return;
  // Set the axis
  VecCopy(that->_axis[dim], v);
  // Update the EqLinSys
  ShapoidUpdateEqLinSysImport(that);
}

// Translate the Shapoid by 'v'
// Do nothing if arguments are invalid
void ShapoidTranslate(Shapoid *that, VecFloat *v) {
  // Check arugments
  if (that == NULL || v == NULL)
    return;
  // Translate the position
  VecOp(that->_pos, 1.0, v, 1.0);
}

// Scale the Shapoid by 'v' (each axis is multiplied by v[iAxis])
// Do nothing if arguments are invalid
void ShapoidScale(Shapoid *that, VecFloat *v) {
  // Check arugments
  if (that == NULL || v == NULL)
    return;
  // Scale each axis
  for (int iAxis = that->_dim; iAxis--;)
    VecOp(that->_axis[iAxis], VecGet(v, iAxis), NULL, 0.0);
  // Update the EqLinSys
  ShapoidUpdateEqLinSysImport(that);
}

// Scale the Shapoid by 'v' (each axis is multiplied by v[iAxis])
// and translate the Shapoid such as its center after scaling
// is at the same position than before scaling
// Do nothing if arguments are invalid
void ShapoidGrow(Shapoid *that, VecFloat *v) {
  // Check arugments
  if (that == NULL || v == NULL)
    return;
  // Scale
  ShapoidScale(that, v);
  // If the shapoid is a Facoid or Pyramidoid
  if (that->_type == ShapoidTypeFacoid ||
    that->_type == ShapoidTypePyramidoid) {
    // Reposition to keep center at the same position
    for (int iAxis = that->_dim; iAxis--;)
      VecOp(that->_pos, 1.0,
        that->_axis[iAxis], -0.5 * (1.0 - 1.0 / VecGet(v, iAxis)));
  }
  // Update the EqLinSys
  ShapoidUpdateEqLinSysImport(that);
}


// Rotate the Shapoid of dimension 2 by 'theta' (in radians, CCW)
// Do nothing if arguments are invalid
void ShapoidRotate2D(Shapoid *that, float theta) {
  // Check arugments
  if (that == NULL)
    return;
  // Rotate each axis
  for (int iAxis = that->_dim; iAxis--;)
    VecRot2D(that->_axis[iAxis], theta);
  // Update the EqLinSys
  ShapoidUpdateEqLinSysImport(that);
}

// Convert the coordinates of 'pos' from standard coordinate system
// toward the Shapoid coordinates system
// Return null if the arguments are invalid
VecFloat* ShapoidImportCoord(Shapoid *that, VecFloat *pos) {
  // Check arguments
  if (that == NULL || pos == NULL || ShapoidGetDim(that) != VecDim(pos))
```

```
    return NULL;
  // Create a matrix for the linear system solver
  VecShort *dim = VecShortCreate(2);
  if (dim == NULL)
    return NULL;
  VecSet(dim, 0, that->_dim);
  VecSet(dim, 1, that->_dim);
  MatFloat *mat = MatFloatCreate(dim);
  if (mat == NULL) {
    VecFree(&dim);
    return NULL;
  }
  // Set the values of the matrix
  for (VecSet(dim, 0, 0); VecGet(dim, 0) < that->_dim;
    VecSet(dim, 0, VecGet(dim, 0) + 1)) {
    for (VecSet(dim, 1, 0); VecGet(dim, 1) < that->_dim;
      VecSet(dim, 1, VecGet(dim, 1) + 1)) {
      MatSet(mat, dim, VecGet(that->_axis[VecGet(dim, 0)],
        VecGet(dim, 1)));
    }
  }
  VecFloat *v = VecGetOp(pos, 1.0, that->_pos, -1.0);
  if (v == NULL) {
    VecFree(&dim);
    MatFree(&mat);
    return NULL;
  }
  // Create the linear system solver and solve it
  EqLinSys *sys = EqLinSysCreate(mat, v);
  VecFloat *res = EqLinSysSolve(sys);
  // Free memory
  VecFree(&v);
  VecFree(&dim);
  EqLinSysFree(&sys);
  MatFree(&mat);
  // return the result
  return res;
}

// Convert the coordinates of 'pos' from the Shapoid coordinates system
// toward standard coordinate system
// Return null if the arguments are invalid
VecFloat* ShapoidExportCoord(Shapoid *that, VecFloat *pos) {
  // Check arguments
  if (that == NULL || pos == NULL || ShapoidGetDim(that) != VecDim(pos))
    return NULL;
  // Allocate memory for the result
  VecFloat *res = VecClone(that->_pos);
  // If we could allocate memory
  if (res != NULL)
    for (int dim = that->_dim; dim--;)
      VecOp(res, 1.0, that->_axis[dim], VecGet(pos, dim));
  // Return the result
  return res;
}

// Return true if 'pos' is inside the Shapoid
// Else return false
bool ShapoidIsPosInside(Shapoid *that, VecFloat *pos) {
  // Check arguments
  if (that == NULL || pos == NULL || ShapoidGetDim(that) != VecDim(pos))
    return false;
  // Get the coordinates of pos in the Shapoid coordinate system
  VecFloat *coord = ShapoidImportCoord(that, pos);
  // If we couldn't get the coordinates
  if (coord == NULL)
    // Stop here
    return false;
  // Declare a variable to memorize the result
  bool ret = false;
  // If the Shapoid is a Facoid
  if (that->_type == ShapoidTypeFacoid) {
    // pos is in the Shapoid if all the coord in Shapoid coord
    // system are in [0.0, 1.0]
    ret = true;
    for (int dim = that->_dim; dim-- && ret == true;) {
      float v = VecGet(coord, dim);
      if (v < 0.0 || v > 1.0)
        ret = false;
    }
  // Else, if the Shapoid is a Pyramidoid
  } else if (that->_type == ShapoidTypePyramidoid) {
    // pos is in the Shapoid if all the coord in Shapoid coord
    // system are in [0.0, 1.0] and their sum is in [0.0, 1.0]
    ret = true;
    float sum = 0.0;
    for (int dim = that->_dim; dim-- && ret == true;) {
      float v = VecGet(coord, dim);
      sum += v;
      if (v < 0.0 || v > 1.0)
        ret = false;
    }
    if (ret == true && sum > 1.0)
      ret = false;
  // Else, if the Shapoid is a Spheroid
  } else if (that->_type == ShapoidTypeSpheroid) {
    // pos is in the Shapoid if its norm is in [0.0, 0.5]
    float norm = VecNorm(coord);
    if (norm <= 0.5)
      ret = true;
  }
  // Free memory
  VecFloatFree(&coord);
  // Return the result
  return ret;
}

// Get a bounding box of the Shapoid. The bounding box is aligned
// on the standard coordinate system (its axis are colinear with
// the axis of the standard coordinate system).
// The bounding box is returned as a Facoid, which position is
// at the minimum value along each axis.
// Return null if the argument are invalid.
Shapoid* ShapoidGetBoundingBoxThat(Shapoid *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Declare a variable to memorize the result
  Shapoid *res = FacoidCreate(ShapoidGetDim(that));
  if (res != NULL) {
    // If the Shapoid is a Facoid
    if (that->_type == ShapoidTypeFacoid) {
      // For each axis
      for (int dim = that->_dim; dim--;) {
```

```
    // Declare a variable to memorize the bound of the interval on
    // this axis
    float bound[2];
    bound[0] = bound[1] = VecGet(that->_pos, dim);
    // For each parameter
    for (int param = that->_dim; param--;) {
      // Get the value of the axis influencing the current dimension
      float v = VecGet(that->_axis[param], dim);
      // If the value is negative, update the minimum bound
      if (v < 0.0)
        bound[0] += v;
      // Else, if the value is negative, update the minimum bound
      else
        bound[1] += v;
    }
    // Memorize the result
    VecSet(res->_pos, dim, bound[0]);
    VecSet(res->_axis[dim], dim, bound[1] - bound[0]);
  }
// Else, if the Shapoid is a Pyramidoid
} else  if (that->_type == ShapoidTypePyramidoid) {
  // For each axis
  for (int dim = that->_dim; dim--;) {
    // Declare a variable to memorize the bound of the interval on
    // this axis
    float bound[2];
    bound[0] = bound[1] = VecGet(that->_axis[0], dim);
    // For each parameter
    for (int param = that->_dim; param--;) {
      // Get the value of the axis influencing the current dimension
      float v = VecGet(that->_axis[param], dim);
      // Search the min and max values
      if (v < bound[0])
        bound[0] = v;
      if (v > bound[1])
        bound[1] = v;
    }
    // Memorize the result
    if (bound[0] < 0.0) {
      VecSet(res->_pos, dim, VecGet(that->_pos, dim) + bound[0]);
      VecSet(res->_axis[dim], dim, bound[1] - bound[0]);
    } else {
      VecSet(res->_pos, dim, VecGet(that->_pos, dim));
      VecSet(res->_axis[dim], dim, bound[1]);
    }
  }
// Else, if the Shapoid is a Spheroid
} else  if (that->_type == ShapoidTypeSpheroid) {
  // In case of a Spheroid, things get complicate
  // We'll approximate the bounding box of the Spheroid
  // with the one of the same Spheroid viewed as a Facoid
  // and simply take care that the _pos is at the center of the
  // Spheroid
  // For each axis
  for (int dim = that->_dim; dim--;) {
    // Declare a variable to memorize the bound of the interval on
    // this axis
    float bound[2];
    bound[0] = VecGet(that->_pos, dim);
    // Correct position
    // For each parameter
    for (int param = that->_dim; param--;) {
          // Get the value of the axis influencing the current dimension
          float v = VecGet(that->_axis[param], dim);
          // Correct the pos
          bound[0] -= 0.5 * v;
        }
        bound[1] = bound[0];
        // For each parameter
        for (int param = that->_dim; param--;) {
          // Get the value of the axis influencing the current dimension
          float v = VecGet(that->_axis[param], dim);
          // If the value is negative, update the minimum bound
          if (v < 0.0)
            bound[0] += v;
          // Else, if the value is negative, update the minimum bound
          else
            bound[1] += v;
        }
        // Memorize the result
        VecSet(res->_pos, dim, bound[0]);
        VecSet(res->_axis[dim], dim, bound[1] - bound[0]);
      }
    } else {
      // In any case of invalid shapoid type return NULL
      ShapoidFree(&res);
    }
  }
  // Return the result
  return res;
}

// Get the bounding box of a set of Facoid. The bounding box is aligned
// on the standard coordinate system (its axis are colinear with
// the axis of the standard coordinate system).
// The bounding box is returned as a Facoid, which position is
// at the minimum value along each axis.
// Return null if the arguments are invalid or the shapoid in the set
// don't have all the same dimension.
Shapoid* ShapoidGetBoundingBoxSet(GSet *set) {
  // Check arguments
  if (set == NULL)
    return NULL;
  // Declare a variable for the result
  Shapoid *res = NULL;
  // Declare a pointer to the elements of the set
  GSetElem *elem = set->_head;
  // Loop on element of the set
  while (elem != NULL) {
    // Declare a pointer to the Facoid
    Shapoid *shapoid = (Shapoid*)(elem->_data);
    // If it's the first Facoid in the set
    if (res == NULL) {
      // Get the bounding box of this shapoid
      res = ShapoidGetBoundingBox(shapoid);
      // If we couldn't get the bounding box
      if (res == NULL)
        return NULL;
    // Else, this is not the first Shapoid in the set
    } else {
      // Ensure the Facoids have all the same dimension
      if (shapoid->_dim != res->_dim) {
        ShapoidFree(&res);
        return NULL;
      }
      // Get the bounding box of this shapoid
      Shapoid *bound = ShapoidGetBoundingBox(shapoid);
      // If we couldn't get the bounding box
      if (bound == NULL) {
        ShapoidFree(&res);
        return NULL;
      }
      // For each dimension
      for (int iDim = res->_dim; iDim--;) {
        // Update the bounding box
        if (VecGet(bound->_pos, iDim) < VecGet(res->_pos, iDim)) {
          VecSet(res->_axis[iDim], iDim,
            VecGet(res->_axis[iDim], iDim) +
            VecGet(res->_pos, iDim) -
            VecGet(bound->_pos, iDim));
          VecSet(res->_pos, iDim, VecGet(bound->_pos, iDim));
        }
        if (VecGet(bound->_pos, iDim) +
          VecGet(bound->_axis[iDim], iDim) >
          VecGet(res->_pos, iDim) +
          VecGet(res->_axis[iDim], iDim))
          VecSet(res->_axis[iDim], iDim,
          VecGet(bound->_pos, iDim) +
          VecGet(bound->_axis[iDim], iDim) -
          VecGet(res->_pos, iDim));
      }
      // Free memory used by the bounding box
      ShapoidFree(&bound);
    }
    // Move to the next element
    elem = elem->_next;
  }
  // Return the result
  return res;
}

// Get a SCurve approximating the Shapoid 'that'
// 'that' must be of dimension 2
// Return null if arguments are invalid
SCurve* Shapoid2SCurve(Shapoid *that) {
  // Check arguments
  if (that == NULL || ShapoidGetDim(that) != 2)
    return NULL;
  // Declare a SCurve to memorize the result
  SCurve *ret = SCurveCreate(ShapoidGetDim(that));
  // If we couldn't allocate memory
  if (ret == NULL)
    return NULL;
  // Declare a pointer to the GSet of the SCurve
  GSet *set = ret->_curves;
  // If the shapoid is a Facoid
  if (ShapoidGetType(that) == ShapoidTypeFacoid) {
    VecFloat *A = VecGetOp(that->_pos, 1.0, that->_axis[0], 1.0);
    VecFloat *B = VecGetOp(that->_pos, 1.0, that->_axis[1], 1.0);
    VecFloat *C = VecGetOp(A, 1.0, that->_axis[1], 1.0);
    BCurve *curve = NULL;
    if (A != NULL && B != NULL && C != NULL) {
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, that->_pos);
        BCurveSet(curve, 1, A);
        GSetAppend(set, curve);
      }
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, A);
        BCurveSet(curve, 1, C);
        GSetAppend(set, curve);
      }
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, C);
        BCurveSet(curve, 1, B);
        GSetAppend(set, curve);
      }
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, B);
        BCurveSet(curve, 1, that->_pos);
        GSetAppend(set, curve);
      }
    }
    VecFree(&A);
    VecFree(&B);
    VecFree(&C);
  // Else, if the shapoid is a Pyramidoid
  } else if (ShapoidGetType(that) == ShapoidTypePyramidoid) {
    VecFloat *A = VecGetOp(that->_pos, 1.0, that->_axis[0], 1.0);
    VecFloat *B = VecGetOp(that->_pos, 1.0, that->_axis[1], 1.0);
    BCurve *curve = NULL;
    if (A != NULL && B != NULL) {
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, that->_pos);
        BCurveSet(curve, 1, A);
        GSetAppend(set, curve);
      }
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, A);
        BCurveSet(curve, 1, B);
        GSetAppend(set, curve);
      }
      curve = BCurveCreate(1, 2);
      if (curve != NULL) {
        BCurveSet(curve, 0, B);
        BCurveSet(curve, 1, that->_pos);
        GSetAppend(set, curve);
      }
    }
    VecFree(&A);
    VecFree(&B);
  // Else, if the shapoid is a Spheroid
  } else if (ShapoidGetType(that) == ShapoidTypeSpheroid) {
    // Approximate each quarter of the Spheroid with BCurves
    // Declare a variable to memorize the angular position on the
    // Spheroid surface
    float theta = 0.0;
    // Declare a variable to memorize the delta of angular position
    float deltaTheta = PBMATH_HALFPI / 3.0;
    // Declare a GSet to memorize the 4 points of the point cloud
    // used to calculate the BCurve approximating the quarter of
    // Spheroid
```

```
    GSet *pointCloud = GSetCreate();
    if (pointCloud != NULL) {
      // Loop until we have made a full turn around the Spheroid
      for (int iCurve = 4; iCurve--;) {
        // For each point of the point cloud
        for (int iPoint = 4; iPoint--;) {
          // Declare a variable to memorize the coordinates of the point
          VecFloat *point = VecFloatCreate(2);
          // If we could allocate memory
          if (point != NULL) {
            // Calculate the coordinates of the current point in the
            // Spheroid coordinate system
            VecSet(point, 0, 0.5 * cos(theta));
            VecSet(point, 1, 0.5 * sin(theta));
            // Add the point converted to standard coordinate system
            // to the point cloud
            VecFloat *pointConvert = ShapoidExportCoord(that, point);
            GSetAppend(pointCloud, pointConvert);
            VecFree(&point);
          }
          // Increment the angular position
          theta += deltaTheta;
        }
        BCurve *curve = BCurveFromCloudPoint(pointCloud);
        // If we could get the BCurve
        if (curve != NULL)
          // Append the curve to the set of curve to be drawm
          GSetAppend(set, curve);
        // Free memory
        GSetElem *elem = pointCloud->_head;
        while (elem != NULL) {
          VecFloatFree((VecFloat**)(&(elem->_data)));
          elem = elem->_next;
        }
        // Empty the point cloud
        GSetFlush(pointCloud);
        // We need to decrement theta because the first point of the
        // next curve will be the last point of the current curve
        theta -= deltaTheta;
      }
      GSetFree(&pointCloud);
    }
  }
  // Return the result
  return ret;
}

// Get the depth value in the Shapoid of 'pos'
// The depth is defined as follow: the point with depth equals 1.0 is
// the farthest point from the surface of the Shapoid (inside it),
// points with depth equals to 0.0 are point on the surface of the
// Shapoid. Depth is continuous and derivable over the volume of the
// Shapoid
// Return 0.0 if arguments are invalid, or pos is outside the Shapoid
float ShapoidGetPosDepth(Shapoid *that, VecFloat *pos) {
  // Check arguments
  if (that == NULL || pos == NULL || ShapoidGetDim(that) != VecDim(pos))
    return 0.0;
  // Get the coordinates of pos in the Shapoid coordinate system
  VecFloat *coord = ShapoidImportCoord(that, pos);
  // If we couldn't get the coordinates
  if (coord == NULL)
```

```
    // Stop here
    return 0.0;
  // Declare a variable to memorize the result
  float ret = 0.0;
  // If the Shapoid is a Facoid
  if (that->_type == ShapoidTypeFacoid) {
    ret = 1.0;
    for (int dim = that->_dim; dim-- && ret > PBMATH_EPSILON;) {
      float v = VecGet(coord, dim);
      if (v < 0.0 || VecGet(coord, dim) > 1.0)
        ret = 0.0;
      else
        ret *= 1.0 - pow(0.5 - v, 2.0) * 4.0;
    }
  // Else, if the Shapoid is a Pyramidoid
  } else if (that->_type == ShapoidTypePyramidoid) {
    ret = 1.0;
    float sum = 0.0;
    bool flag = true;
    for (int dim = that->_dim; dim-- && ret > PBMATH_EPSILON;) {
      float v = VecGet(coord, dim);
      sum += v;
      if (v < 0.0 || v > 1.0)
        flag = false;
    }
    if (flag == true && sum > 1.0)
      flag = false;
    if (flag == false)
      ret = 0.0;
    else {
      ret = 1.0;
      for (int dim = ShapoidGetDim(that); dim--;) {
        float z = 0.0;
        for (int d = ShapoidGetDim(that); d--;)
          if (d != dim)
            z += VecGet(coord, d);
        ret *=
          (1.0 - 4.0 * pow(0.5 - VecGet(coord, dim) / (1.0 - z), 2.0));
      }
    }
  // Else, if the Shapoid is a Spheroid
  } else if (that->_type == ShapoidTypeSpheroid) {
    float norm = VecNorm(coord);
    if (norm <= 0.5)
      ret = 1.0 - norm * 2.0;
  }
  // Free memory
  VecFloatFree(&coord);
  // Return the result
  return ret;
}

// Get the center of the shapoid in standard coordinate system
// Return null if arguments are invalid
VecFloat* ShapoidGetCenter(Shapoid *that) {
  // Check arguments
  if (that == NULL)
    return NULL;
  // Declare a variable to memorize the result in Shapoid
  // coordinate system
  VecFloat *coord = VecFloatCreate(ShapoidGetDim(that));
  // If we could allocate memory
  if (coord != NULL) {
    // For each dimension
    for (int dim = ShapoidGetDim(that); dim--;) {
      if (ShapoidGetType(that) == ShapoidTypeFacoid)
        VecSet(coord, dim, 0.5);
      else if (ShapoidGetType(that) == ShapoidTypePyramidoid)
        VecSet(coord, dim, 1.0 / (1.0 + ShapoidGetDim(that)));
      else if (ShapoidGetType(that) == ShapoidTypeSpheroid)
        VecSet(coord, dim, 0.0);
    }
  }
  // Convert the cooridnate in standard cooridnate system
  VecFloat *res = ShapoidExportCoord(that, coord);
  // Return the result
  return res;
}

// Get the percentage of 'tho' included 'that' (in [0.0, 1.0])
// 0.0 -> 'tho' is completely outside of 'that'
// 1.0 -> 'tho' is completely inside of 'that'
// 'that' and 'tho' must me of same dimensions
// Return 0.0 if the arguments are invalid or something went wrong
float ShapoidGetCoverage(Shapoid *that, Shapoid *tho) {
  // Check arguments
  if (that == NULL || tho == NULL ||
    ShapoidGetDim(that) != ShapoidGetDim(tho))
    return 0.0;
  // Declare variables to compute the result
  float ratio = 0.0;
  float sum = 0.0;
  // Declare variables for the relative and absolute position in 'tho'
  VecFloat *pRel = VecFloatCreate(ShapoidGetDim(that));
  VecFloat *pAbs = NULL;
  // If we couldn't allocate memory
  if (pRel == NULL) {
    // Free memory and stop here
    VecFree(&pRel);
    return 0.0;
  }
  // Declare a variable to memorize the step in relative coordinates
  float delta = 0.1;
  // Declare a variable to memorize the last index in dimension
  int lastI = VecDim(pRel) - 1;
  // Declare a variable to memorize the max value of coordinates
  float max = 1.0;
  // If 'tho' is a spheroid, reposition the start coordinates
  if (tho->_type == ShapoidTypeSpheroid) {
    max = 0.5;
    for (int iDim = ShapoidGetDim(that); iDim--;)
      VecSet(pRel, iDim, -0.5);
  }
  // Loop on relative coordinates
  while (VecGet(pRel, lastI) <= max + PBMATH_EPSILON) {
    // Get the absolute coordinates
    pAbs = ShapoidExportCoord(tho, pRel);
    // If we could get the position
    if (pAbs != NULL) {
      // If this position is inside 'that'
      if (ShapoidIsPosInside(that, pAbs) == true)
        // Increment the ratio
        ratio += 1.0;
      sum += 1.0;
      // Free memory
      VecFree(&pAbs);
    }
    // Step the relative coordinates
    int iDim = 0;
    while (iDim >= 0) {
      VecSet(pRel, iDim, VecGet(pRel, iDim) + delta);
      if (iDim != lastI &&
        VecGet(pRel, iDim) > max + PBMATH_EPSILON) {
        VecSet(pRel, iDim, max - 1.0);
        ++iDim;
      } else {
        iDim = -1;
      }
    }
  }
  // Finish the computation of the ratio
  ratio /= sum;
  // Free memory
  VecFree(&pRel);
  // Return the result
  return ratio;
}
```

4 Makefile

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

all : main

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

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

shapoid.o : shapoid.c shapoid.h Makefile $(INCPATH)/bcurve.h $(INCPATH)/gset.h $(INCPATH)/pbmath.h
gcc $(OPTIONS) -I$(INCPATH) -c shapoid.c

clean :
rm -rf *.o main

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

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

5 Usage

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

```
#include <time.h>
#include <string.h>
#include "shapoid.h"

int main(int argc, char **argv) {
  // Initialise the random generator
  srandom(time(NULL));

  Shapoid* facoidA = FacoidCreate(2);
  ShapoidPrint(facoidA, stdout);
  VecFloat *v = VecFloatCreate(2);
  if (v == NULL) {
    fprintf(stderr, "malloc failed\n");
    return 28;
  }
  VecSet(v, 0, 2.0);
  VecSet(v, 1, 3.0);
  ShapoidScale(facoidA, v);
  fprintf(stdout, "scale by ");
  VecPrint(v, stdout);
  fprintf(stdout, "\n");
  ShapoidPrint(facoidA, stdout);
  ShapoidRotate2D(facoidA, PBMATH_PI * 0.5);
  fprintf(stdout, "rotate by %.3f\n", PBMATH_PI * 0.5);
  ShapoidPrint(facoidA, stdout);
  VecSet(v, 0, 1.0);
  VecSet(v, 1, -1.0);
  ShapoidTranslate(facoidA, v);
  fprintf(stdout, "translate by ");
  VecPrint(v, stdout);
  fprintf(stdout, "\n");
  ShapoidPrint(facoidA, stdout);
  VecSet(v, 0, 0.0);
  VecSet(v, 1, 1.0);
  ShapoidSetAxis(facoidA, 0, v);
  fprintf(stdout, "set axis 0 to ");
  VecPrint(v, stdout);
  fprintf(stdout, "\n");
  ShapoidPrint(facoidA, stdout);
  fprintf(stdout, "save shapoid to shapoid.txt\n");
  FILE *f = fopen("./shapoid.txt", "w");
  if (f == NULL) {
    fprintf(stderr, "fopen failed\n");
    return 29;
  }
  ShapoidSave(facoidA, f);
  fclose(f);
  fprintf(stdout, "load shapoid from shapoid.txt\n");
  Shapoid* facoidB = NULL;
  f = fopen("./shapoid.txt", "r");
  if (f == NULL) {
    fprintf(stderr, "fopen failed\n");
    return 30;
  }
  ShapoidLoad(&facoidB, f);
  fclose(f);
  ShapoidPrint(facoidB, stdout);
  VecSet(v, 0, 1.0);
  VecSet(v, 1, 1.0);
  VecFloat *coordEx = ShapoidExportCoord(facoidB, v);
  if (coordEx == NULL) {
    fprintf(stderr, "ShapoidExport failed\n");
    return 31;
  }
  fprintf(stdout,"coordinates ");
  VecPrint(v, stdout);
  fprintf(stdout," in the shapoid becomes ");
  VecPrint(coordEx, stdout);
  fprintf(stdout," in the standard coordinate system\n");
  VecCopy(v, coordEx);
  VecFloat *coordIm = ShapoidImportCoord(facoidB, v);
  if (coordIm == NULL) {
    fprintf(stderr, "ShapoidImport failed\n");
    return 32;
  }
  fprintf(stdout,"coordinates ");
  VecPrint(v, stdout);
  fprintf(stdout," in the standard coordinate system becomes ");
  VecPrint(coordIm, stdout);
  fprintf(stdout," in the shapoid\n");
  VecSet(v, 0, 0.0);
  VecSet(v, 1, 0.0);
  VecPrint(v, stdout);
  if (ShapoidIsPosInside(facoidB, v) == true)
    fprintf(stdout, " is in the facoid\n");
  else
    fprintf(stdout, " is not in the facoid\n");
  VecSet(v, 0, 1.0);
  VecSet(v, 1, -4.0);
  VecPrint(v, stdout);
  if (ShapoidIsPosInside(facoidB, v) == true)
    fprintf(stdout, " is in the facoid\n");
  else
    fprintf(stdout, " is not in the facoid\n");
  ShapoidRotate2D(facoidB, -PBMATH_QUARTERPI);
  Shapoid *bounding = ShapoidGetBoundingBox(facoidB);
  if (bounding == NULL) {
    fprintf(stderr, "ShapoidGetBoundingBox failed\n");
    return 33;
  }
  fprintf(stdout, "bounding box of\n");
  ShapoidPrint(facoidB, stdout);
  fprintf(stdout, "is\n");
  ShapoidPrint(bounding, stdout);
  ShapoidFree(&bounding);
  VecFloat *center = ShapoidGetCenter(facoidB);
  if (center == NULL) {
    fprintf(stderr, "ShapoidGetCenter failed\n");
    return 34;
  }
  fprintf(stdout, "center of the facoid is ");
  VecPrint(center, stdout);
  fprintf(stdout, "\n");
  VecFree(&center);
  facoidB->_type = ShapoidTypePyramidoid;
  bounding = ShapoidGetBoundingBox(facoidB);
  if (bounding == NULL) {
    fprintf(stderr, "ShapoidGetBoundingBox failed\n");
    return 35;
  }
  fprintf(stdout, "bounding box of\n");
  ShapoidPrint(facoidB, stdout);
  fprintf(stdout, "is\n");
  ShapoidPrint(bounding, stdout);
```

```
ShapoidFree(&bounding);
center = ShapoidGetCenter(facoidB);
if (center == NULL) {
  fprintf(stderr, "ShapoidGetCenter failed\n");
  return 36;
}
fprintf(stdout, "center of the facoid is ");
VecPrint(center, stdout);
fprintf(stdout, "\n");
VecFree(&center);
facoidB->_type = ShapoidTypeSpheroid;
bounding = ShapoidGetBoundingBox(facoidB);
if (bounding == NULL) {
  fprintf(stderr, "ShapoidGetBoundingBox failed\n");
  return 37;
}
fprintf(stdout, "bounding box of\n");
ShapoidPrint(facoidB, stdout);
fprintf(stdout, "is\n");
ShapoidPrint(bounding, stdout);
center = ShapoidGetCenter(facoidB);
if (center == NULL) {
  fprintf(stderr, "ShapoidGetCenter failed\n");
  return 38;
}
fprintf(stdout, "center of the shapoid is ");
VecPrint(center, stdout);
fprintf(stdout, "\n");
VecFree(&center);

GSet *setBounding = GSetCreate();
if (setBounding == NULL) {
  fprintf(stderr, "GSetCreate failed\n");
  return 39;
}
GSetAppend(setBounding, facoidA);
GSetAppend(setBounding, facoidB);
facoidB->_type = ShapoidTypeFacoid;
VecSet(facoidB->_pos, 0, 2.0);
fprintf(stdout, "bounding box of\n");
ShapoidPrint(facoidA, stdout);
fprintf(stdout, "and\n");
ShapoidPrint(facoidB, stdout);
fprintf(stdout, "is\n");
bounding = ShapoidGetBoundingBox(setBounding);
if (bounding == NULL) {
  fprintf(stderr, "ShapoidGetBoundingBox failed\n");
  return 40;
}
ShapoidPrint(bounding, stdout);
// Grow
fprintf(stdout, "Grow the facoid:\n");
ShapoidPrint(facoidA, stdout);
fprintf(stdout, "by 2.0:\n");
VecSet(v, 0, 2.0); VecSet(v, 1, 2.0);
ShapoidGrow(facoidA, v);
ShapoidPrint(facoidA, stdout);
// Coverage ratio
fprintf(stdout, "Percentage of :\n");
ShapoidPrint(facoidB, stdout);
fprintf(stdout, "included in :\n");
ShapoidPrint(facoidA, stdout);
```

```
  float ratio = ShapoidGetCoverage(facoidA, facoidB);
  fprintf(stdout, "is %f\n", ratio);
  // Free memory
  ShapoidFree(&bounding);
  GSetFree(&setBounding);
  VecFree(&coordEx);
  VecFree(&coordIm);
  VecFree(&v);
  ShapoidFree(&facoidA);
  ShapoidFree(&facoidB);
  ShapoidFree(&bounding);

  // Return success code
  return 0;
}
```

Output:

```
Type: Facoid
Dim: 2
Pos: <0.000,0.000>
Axis(0): <1.000,0.000>
Axis(1): <0.000,1.000>
scale by <2.000,3.000>
Type: Facoid
Dim: 2
Pos: <0.000,0.000>
Axis(0): <2.000,0.000>
Axis(1): <0.000,3.000>
rotate by 1.571
Type: Facoid
Dim: 2
Pos: <0.000,0.000>
Axis(0): <0.000,2.000>
Axis(1): <-3.000,0.000>
translate by <1.000,-1.000>
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.000,2.000>
Axis(1): <-3.000,0.000>
set axis 0 to <0.000,1.000>
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.000,1.000>
Axis(1): <-3.000,0.000>
save shapoid to shapoid.txt
load shapoid from shapoid.txt
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.000,1.000>
Axis(1): <-3.000,0.000>
coordinates <1.000,1.000> in the shapoid becomes <-2.000,0.000> in the standard coordinate system
coordinates <-2.000,0.000> in the standard coordinate system becomes <1.000,1.000> in the shapoid
<0.000,0.000> is in the facoid
<1.000,-4.000> is not in the facoid
bounding box of
Type: Facoid
```

```
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.707,0.707>
Axis(1): <-2.121,2.121>
is
Type: Facoid
Dim: 2
Pos: <-1.121,-1.000>
Axis(0): <2.828,0.000>
Axis(1): <0.000,2.828>
center of the facoid is <0.293,0.414>
bounding box of
Type: Pyramidoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.707,0.707>
Axis(1): <-2.121,2.121>
is
Type: Facoid
Dim: 2
Pos: <-1.121,-1.000>
Axis(0): <2.828,0.000>
Axis(1): <0.000,2.121>
center of the facoid is <0.529,-0.057>
bounding box of
Type: Spheroid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.707,0.707>
Axis(1): <-2.121,2.121>
is
Type: Facoid
Dim: 2
Pos: <-0.414,-2.414>
Axis(0): <2.828,0.000>
Axis(1): <0.000,2.828>
center of the shapoid is <1.000,-1.000>
bounding box of
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.000,1.000>
Axis(1): <-3.000,0.000>
and
Type: Facoid
Dim: 2
Pos: <2.000,-1.000>
Axis(0): <0.707,0.707>
Axis(1): <-2.121,2.121>
is
Type: Facoid
Dim: 2
Pos: <-2.000,-1.000>
Axis(0): <4.707,0.000>
Axis(1): <0.000,2.828>
Grow the facoid:
Type: Facoid
Dim: 2
Pos: <1.000,-1.000>
Axis(0): <0.000,1.000>
Axis(1): <-3.000,0.000>
by 2.0:
```

```
Type: Facoid
Dim: 2
Pos: <2.500,-1.500>
Axis(0): <0.000,2.000>
Axis(1): <-6.000,0.000>
Percentage of :
Type: Facoid
Dim: 2
Pos: <2.000,-1.000>
Axis(0): <0.707,0.707>
Axis(1): <-2.121,2.121>
included in :
Type: Facoid
Dim: 2
Pos: <2.500,-1.500>
Axis(0): <0.000,2.000>
Axis(1): <-6.000,0.000>
is 0.600000
```

shapoid.txt:

```
2 1
2 1.000000 -1.000000
2 0.000000 1.000000
2 -3.000000 0.000004
```