

Shapoid

P. Baillehache

May 22, 2018

Contents

1	Definitions	2
1.1	Transformation	2
1.2	Shapoid's coordinate system	3
1.3	Insideness	3
1.4	Bounding box	3
1.5	Depth and Center	4
1.6	Iterator on Spheroid	5
1.7	Collision detection of Spheroid	7
2	Interface	8
3	Code	26
3.1	shapoid.c	26
3.2	shapoid-inline.c	47
4	Makefile	76
5	Unit tests	77
6	Unit tests output	110

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).

It also provides the **ShapoidIter** structure and its functions which can be used to sequentially loop through the surface/volume/... of a **Shapoid**.

It uses the **PBErr**, **PBMath** and **GSet** libraries.

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 \vec{P} , and its axis $(\vec{A}_0, \vec{A}_1, \dots, \vec{A}_{D-1})$. 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:

$$\left\{ \sum_{i \in I} v_i \vec{A}_i + \vec{P} \right\}, v_i \in [0.0, 1.0] \quad (1)$$

for a Pyramidoid:

$$\left\{ \sum_{i \in I} v_i \vec{A}_i + \vec{P} \right\}, v_i \in [0.0, 1.0], \sum_{i \in I} v_i \leq 1.0 \quad (2)$$

and for a Spheroid:

$$\left\{ \sum_{i \in I} v_i \vec{A}_i + \vec{P} \right\}, \quad v_i \in [-0.5, 0.5], \quad \sum_{i \in I} v_i^2 \leq 0.25 \quad (3)$$

1.1 Transformation

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

$$(\vec{P}, \{\vec{A}_i\}_{i \in I}) \mapsto (\vec{P} + \vec{T}, \{\vec{A}_i\}_{i \in I}) \quad (4)$$

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

$$(\vec{P}, \{\vec{A}_i\}_{i \in I}) \mapsto (\vec{P}, \{\vec{A}'_i\}_{i \in I}) \quad (5)$$

where

$$\vec{A}'_i = S_i \vec{A}_i \quad (6)$$

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

$$(\vec{P}, \vec{A}_0, \vec{A}_1) \mapsto (\vec{P}, \vec{A}'_0, \vec{A}'_1) \quad (7)$$

where

$$\vec{A}'_i = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \vec{A}_i \quad (8)$$

1.2 Shapoid's coordinate system

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

$$\vec{X} = \left[\left(\vec{A}_0 \right) \left(\vec{A}_1 \right) \dots \left(\vec{A}_{D-1} \right) \right] \vec{X}^S + \vec{P} \quad (9)$$

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

$$\vec{X}^S = \left[\left(\vec{A}_0 \right) \left(\vec{A}_1 \right) \dots \left(\vec{A}_{D-1} \right) \right]^{-1} (\vec{X} - \vec{P}) \quad (10)$$

1.3 Insideness

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

$$\forall i \in I, 0.0 \leq X_i^S \leq 1.0 \quad (11)$$

for a Pyramidoid:

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

for a Spheroid:

$$\left\| \vec{X}^S \right\| \leq 0.5 \quad (13)$$

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 $\left(\vec{P} - \sum_{i \in I} (0.5 * \vec{A}_i), \{\vec{A}_i\}_{i \in I}\right)$ 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:

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

where

$$\begin{cases} P'_i = P_i + \sum_{j \in I^-} A_{ji} \\ A'_{ij} = 0.0, i \neq j \\ A'_{ij} = \sum_{k \in I^+} A_{kj} - \sum_{k \in I^-} A_{kj}, i = j \end{cases} \quad (15)$$

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:

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

where

$$\begin{cases} P'_i = P_i + \text{Min}(\text{Min}_{j \in I}(A_{ji}), 0.0) \\ A'_{ij} = 0.0, i \neq j \\ A'_{ij} = \text{Max}_{k \in I}(A_{kj}) - \text{Min}_{k \in I}(A_{kj}), i = j \end{cases} \quad (17)$$

1.5 Depth and Center

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

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

for a Pyramidoid:

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

and for a Spheroid:

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

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

$$\forall i \in I, C_i^S = 0.5 \quad (21)$$

for a Pyramidoid:

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

for a Spheroid:

$$\forall i \in I, C_i^S = 0.0 \quad (23)$$

\vec{C} is called the center of the Shapoid.

1.6 Iterator on Spheroid

While a sequential path through a Facoid and a Pyramidoid is obvious, path through a Spheroid is more complex. The solution implemented is described below.

Given a Spheroid of dimension N we start from an arbitrary position: $< 0, 0, \dots, -0.5 >$. From there we step the axis starting from the first one. If we could step an axis the step algorithm stops and return the new position as it could successfully step. However, if we could step on an axis other than the first one, it means we have modified the constraint for previous axis, the constraint being "is inside the spheroid". Then we reposition the axis before the stepped one to its lower possible value. It will allow it to step again at the next iteration on a new boundary defined by other axis values, and this scale up naturally to any dimension. Care must be care to the case when an axis reaches its upper value: the delta given by the user and the influence of other axis value make it jumps "over" the boundary in most cases. To keep things neat and clean we recalculate the exact value of the axis for its last step instead of using the delta given by the user.

The calculation of the lower and upper values of an axis given the values of other axis can be performed as follow:

Lets note $\vec{P} = \langle p_0, p_1, \dots, p_{N-1} \rangle$ the position in a Spheroid of dimension N . A position will be on the boundary of the Spheroid if and only if $\|\vec{P}\| = 0.5$. We want to calculate α which bring a position \vec{P}' inside the Spheroid to a position \vec{P} on the boundary of the Spheroid by modifying the axis i (i.e. $p_i = p'_i + \alpha$ and $p_j = p'_j, j \neq i$). Lets note $n = \|\vec{P}'\|$. We have:

$$p_0'^2 + p_1'^2 + \dots + p_i'^2 + \dots + p_{N-1}'^2 = n^2 \quad (24)$$

and

$$p_0^2 + p_1^2 + \dots + p_i^2 + \dots + p_{N-1}^2 = 0.25 \quad (25)$$

equivalent to

$$p_0'^2 + p_1'^2 + \dots + (p'_i + \alpha)^2 + \dots + p_{N-1}'^2 = 0.25 \quad (26)$$

by substracting (24) and (26) we have

$$p_i'^2 - (p'_i + \alpha)^2 = n^2 - 0.25 \quad (27)$$

equivalent to

$$p_i'^2 - (p_i'^2 + \alpha^2 + 2p'_i\alpha) = n^2 - 0.25 \quad (28)$$

equivalent to

$$-\alpha^2 - 2p'_i\alpha - (n^2 - 0.25) = 0 \quad (29)$$

simplified to

$$\alpha^2 + 2p'_i\alpha + (n^2 - 0.25) = 0 \quad (30)$$

This quadratic equation can be solved directly to obtain α :

$$\alpha = \frac{-2p'_i \pm \sqrt{4p_i'^2 - 4(n^2 - 0.25)}}{2} \quad (31)$$

Which gives the two solutions defining the lower and upper boundaries of the Spheroid on the axis i .

This result can then be used to solve our problem with what I'll call the "Wormy Algorithm":

```

flag := true
norm := Norm(P)
iDim := 0
loop until (iDim < N and flag == true)
  prevNorm := norm
  P_iDim := P_iDim + delta
  norm := Norm(P)
  if (prevNorm < 0.5 and norm > 0.5)
    P_iDim := 0
    norm := Norm(P)
    val := 0.5 * Sqrt(-4.0 * (norm^2 - 0.25))
    P_iDim := val
    norm := 0.5
    flag := false
    iDim := iDim - 1
  else
    if (norm > 0.5)
      P_iDim := 0
    else
      flag := false
      iDim := iDim - 1
    end if
  end if
  iDim := iDim + 1
end loop
if (flag == false)
  iDim := iDim - 1
  if (iDim >= 0)
    P_iDim := P_iDim + 0.5 * (-2.0 * P_iDim -
      Sqrt(4.0 * (P_iDim)^2 - 4.0 * (norm^2 - 0.25)))
  end if
end if
return Not(flag)

```

This algorithm step P to the next position in the path by δ and returns true if we haven't reached the end of the path, or false if we have reached the end of the path (i.e. if we have iterated through all the surface/volume/... of the Spheroid). Remember that P must be initialised to $\langle 0, 0, \dots, -0.5 \rangle$ as the beginning of the path, and that **ShapoidIter** iterates coordinates in the Shapoid's coordinate system.

1.7 Collision detection of Spheroid

The detection of collision between two Spheroids is done as follow. One of the spheroid is converted into the coordinates system of the other and checked against a circle of radius 0.5 centered at the origin of the system. By checking that the position of the converted Spheroid is less than its minor radius plus 0.5 or more than its major radius plus 0.5, the trivial cases of, respectively, interesection and non intersection can be performed. In other cases an incremental search from the center of the converted Spheroid toward the nearest

point to the origin inside this Spheroid is performed. The intersection can then be checked by testing if the distance of this point to the origin is less or equal than 0.5.

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 "pberr.h"
#include "pbmath.h"
#include "gset.h"

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

#define CloneShapoidType(T) typedef struct T {Shapoid _s;} T

#define SpheroidCreate(Dim) \
    (Spheroid*)ShapoidCreate(Dim, ShapoidTypeSpheroid)
#define FacoidCreate(Dim) \
    (Facoid*)ShapoidCreate(Dim, ShapoidTypeFacoid)
#define PyramidoidCreate(Dim) \
    (Pyramidoid*)ShapoidCreate(Dim, ShapoidTypePyramidoid)

#define ShapoidGetCoverage(ShapoidA, ShapoidB) \
    _ShapoidGetCoverageDelta((Shapoid* const)ShapoidA, \
    (Shapoid* const)ShapoidB, 0.1)
#define ShapoidGetCoverageDelta(ShapoidA, ShapoidB, Prec) \
    _ShapoidGetCoverageDelta((Shapoid* const)ShapoidA, \
    (Shapoid* const)ShapoidB, Prec)

#define ShapoidIterCreateStatic(Shap, Delta) \
    _ShapoidIterCreateStatic((Shapoid* const)(Shap), (VecFloat*)(Delta))

#define ShapoidIterSetShapoid(Iter, Shap) \
    _ShapoidIterSetShapoid(Iter, (Shapoid* const)(Shap))

#define ShapoidIterSetDelta(Iter, Delta) \
    _ShapoidIterSetDelta(Iter, (VecFloat* const)(Delta))

extern const char* ShapoidTypeString[3];

// ----- ShapoidIter

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

typedef enum ShapoidType {
    ShapoidTypeFacoid, ShapoidTypeSpheroid,
```



```

    ShapoidTypePyramidoid
} ShapoidType;
// Don't forget to update ShapoidTypeString in shapoid.c when adding
// new type

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

CloneShapoidType(Facoid);
CloneShapoidType(Pyramidoid);
typedef struct Spheroid {
    Shapoid _s;
    // Major and minor axis for optimization
    int _majAxis;
    int _minAxis;
} Spheroid;
//CloneShapoidType(Spheroid);

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

// Create a Shapoid of dimension 'dim' and type 'type', default values:
// _pos = null vector
// _axis[d] = unit vector along dimension d
Shapoid* ShapoidCreate(const int dim, const ShapoidType type);

// Clone a Shapoid
Shapoid* _ShapoidClone(const Shapoid* const that);
inline Facoid* FacoidClone(const Facoid* const that) {
    return (Facoid*)_ShapoidClone((const Shapoid*)that);
}
inline Pyramidoid* PyramidoidClone(const Pyramidoid* const that) {
    return (Pyramidoid*)_ShapoidClone((const Shapoid*)that);
}
inline Spheroid* SpheroidClone(const Spheroid* const that) {
    return (Spheroid*)_ShapoidClone((const Shapoid*)that);
}

// Free memory used by a Shapoid
void _ShapoidFree(Shapoid** that);

// Function which return the JSON encoding of 'that'
JSONNode* _ShapoidEncodeAsJSON(const Shapoid* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool _ShapoidDecodeAsJSON(Shapoid** that, const JSONNode* const json);

// Load the Shapoid of type 'type' from the stream
// If the Shapoid is already allocated, it is freed before loading
// Return true upon success else false
bool _ShapoidLoad(Shapoid** that, FILE* const stream);
#if BUILDMODE != 0
inline

```

```

#endif
bool FacoidLoad(Facoid** that, FILE* const stream);
#if BUILDMODE != 0
inline
#endif
bool PyramidoidLoad(Pyramidoid** that, FILE* const stream);
#if BUILDMODE != 0
inline
#endif
bool SpheroidLoad(Spheroid** that, FILE* const stream);

// Save the Shapoid to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true upon success else false
bool _ShapoidSave(const Shapoid* const that, FILE* const stream,
    const bool compact);

// Print the Shapoid on 'stream'
void _ShapoidPrintln(const Shapoid* const that, FILE* const stream);

// Get the dimension of the Shapoid
#if BUILDMODE != 0
inline
#endif
int _ShapoidGetDim(const Shapoid* const that);

// Get the type of the Shapoid
#if BUILDMODE != 0
inline
#endif
ShapoidType _ShapoidGetType(const Shapoid* const that);

// Get the type of the Shapoid as a string
// Return a pointer to a constant string (not to be freed)
#if BUILDMODE != 0
inline
#endif
const char* _ShapoidGetTypeAsString(const Shapoid* const that);

// Return a VecFloat equals to the position of the Shapoid
#if BUILDMODE != 0
inline
#endif
VecFloat* _ShapoidGetPos(const Shapoid* const that);

// Return a VecFloat equals to the 'dim'-th axis of the Shapoid
#if BUILDMODE != 0
inline
#endif
VecFloat* _ShapoidGetAxis(const Shapoid* const that, const int dim);

// Return the position of the Shapoid
#if BUILDMODE != 0
inline
#endif
const VecFloat* _ShapoidPos(const Shapoid* const that);

// Return the 'dim'-th axis of the Shapoid
#if BUILDMODE != 0
inline
#endif

```

```

const VecFloat* _ShapoidAxis(const Shapoid* const that, const int dim);

// Set the position of the Shapoid to 'pos'
#if BUILDMODE != 0
inline
#endif
void _ShapoidSetPos(Shapoid* const that, const VecFloat* const pos);

// Set the 'iElem'-th value of the position of the Shapoid to 'val'
#if BUILDMODE != 0
inline
#endif
void _ShapoidPosSet(Shapoid* const that, const int iElem,
    const float val);

// Set the 'iElem'-th value of the position of the Shapoid to 'val'
// added to its current value
#if BUILDMODE != 0
inline
#endif
void _ShapoidPosSetAdd(Shapoid* const that, const int iElem,
    const float val);

// Set the position of the Shapoid such as its center is at 'pos'
#if BUILDMODE != 0
inline
#endif
void _ShapoidSetCenterPos(Shapoid* const that,
    const VecFloat* const pos);

// Set the 'dim'-th axis of the Shapoid to 'v'
#if BUILDMODE != 0
inline
#endif
void _ShapoidSetAxis(Shapoid* const that, const int dim,
    const VecFloat* const v);

// Set the 'iElem'-th element of the 'dim'-th axis of the Shapoid to 'v'
#if BUILDMODE != 0
inline
#endif
void _ShapoidAxisSet(Shapoid* const that, const int dim,
    const int iElem, const float v);

// Set the 'iElem'-th element of the 'dim'-th axis of the Shapoid to
// 'v' added to its current value
#if BUILDMODE != 0
inline
#endif
void _ShapoidAxisSetAdd(Shapoid* const that, const int dim,
    const int iElem, const float v);

// Translate the Shapoid by 'v'
#if BUILDMODE != 0
inline
#endif
void _ShapoidTranslate(Shapoid* const that, const VecFloat* const v);

// Scale the Shapoid by 'v' (each axis is multiplied by v[iAxis])
#if BUILDMODE != 0
inline
#endif

```

```

void _ShapoidScaleVector(Shapoid* const that, const VecFloat* const v);

// Scale the Shapoid by 'c'
#if BUILDMODE != 0
inline
#endif
void _ShapoidScaleScalar(Shapoid* const that, const float c);

// 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
#if BUILDMODE != 0
inline
#endif
void _ShapoidGrowVector(Shapoid* const that, const VecFloat* const v);

// Scale the Shapoid by 'c'
// and translate the Shapoid such as its center after scaling
// is at the same position than before scaling
#if BUILDMODE != 0
inline
#endif
void _ShapoidGrowScalar(Shapoid* const that, const float c);

// Rotate the Shapoid of dimension 2 by 'theta' (in radians, CCW)
// relatively to its center
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotCenter(Shapoid* const that, const float theta);

// Rotate the Shapoid of dimension 2 by 'theta' (in radians, CCW)
// relatively to its position
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotStart(Shapoid* const that, const float theta);

// Rotate the Shapoid of dimension 2 by 'theta' (in radians, CCW)
// relatively to the origin of the global coordinates system
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotOrigin(Shapoid* const that, const float theta);

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its center around 'axis'
// 'axis' must be normalized
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotAxisCenter(Shapoid* const that,
    const VecFloat3D* const axis, const float theta);

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its position around 'axis'
// 'axis' must be normalized
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotAxisStart(Shapoid* const that,
    const VecFloat3D* const axis, const float theta);

```

```

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to the origin of the global coordinates system
// around 'axis'
// 'axis' must be normalized
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotAxisOrigin(Shapoid* const that,
    const VecFloat3D* const axis, const float theta);

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its center around X
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotXCenter(Shapoid* const that, const float theta);

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its position around X
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotXStart(Shapoid* const that, const float theta);

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to the origin of the global coordinates system
// around X
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotXOrigin(Shapoid* const that, const float theta);

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its center around Y
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotYCenter(Shapoid* const that, const float theta);

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its position around Y
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotYStart(Shapoid* const that, const float theta);

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to the origin of the global coordinates system
// around Y
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotYOrigin(Shapoid* const that, const float theta);

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its center around Z
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotZCenter(Shapoid* const that, const float theta);

```

```

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its position around Z
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotZStart(Shapoid* const that, const float theta);

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to the origin of the global coordinates system
// around Z
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotZOrigin(Shapoid* const that, const float theta);

// Convert the coordinates of 'pos' from standard coordinate system
// toward the Shapoid coordinates system
#if BUILDMODE != 0
inline
#endif
VecFloat* _ShapoidImportCoord(const Shapoid* const that,
    const VecFloat* const pos);

// Convert the coordinates of 'pos' from the Shapoid coordinates system
// toward standard coordinate system
#if BUILDMODE != 0
inline
#endif
VecFloat* _ShapoidExportCoord(const Shapoid* const that,
    const VecFloat* const pos);

// Return true if 'pos' (in standard coordinates system) is inside the
// Shapoid
// Else return false
#if BUILDMODE != 0
inline
#endif
bool _ShapoidIsPosInside(const Shapoid* const that,
    const VecFloat* const pos);
#if BUILDMODE != 0
inline
#endif
bool FacoidIsPosInside(const Facoid* const that,
    const VecFloat* const pos);
#if BUILDMODE != 0
inline
#endif
bool PyramidoidIsPosInside(const Pyramidoid* const that,
    const VecFloat* const pos);
#if BUILDMODE != 0
inline
#endif
bool SpheroidIsPosInside(const Spheroid* const that,
    const VecFloat* const 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.
Facoid* _ShapoidGetBoundingBox(const Shapoid* const that);
Facoid* FacoidGetBoundingBox(const Facoid* const that);

```

```

Facoid* PyramidoidGetBoundingBox(const Pyramidoid* const that);
Facoid* SpheroidGetBoundingBox(const Spheroid* const 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.
Facoid* ShapoidGetBoundingBoxSet(const GSetShapoid* const set);

// Get the depth value in the Shapoid of 'pos' in standard coordinate
// system
// 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 pos is outside the Shapoid
#ifdef BUILDMODE != 0
inline
#endif
float _ShapoidGetPosDepth(const Shapoid* const that,
    const VecFloat* const pos);
#ifdef BUILDMODE != 0
inline
#endif
float FacoidGetPosDepth(const Facoid* const that,
    const VecFloat* const pos);
#ifdef BUILDMODE != 0
inline
#endif
float PyramidoidGetPosDepth(const Pyramidoid* const that,
    const VecFloat* const pos);
#ifdef BUILDMODE != 0
inline
#endif
float SpheroidGetPosDepth(const Spheroid* const that,
    const VecFloat* const pos);

// Get the center of the shapoid in standard coordinate system
#ifdef BUILDMODE != 0
inline
#endif
VecFloat* _ShapoidGetCenter(const Shapoid* const that);
#ifdef BUILDMODE != 0
inline
#endif
VecFloat* FacoidGetCenter(const Facoid* const that);
#ifdef BUILDMODE != 0
inline
#endif
VecFloat* PyramidoidGetCenter(const Pyramidoid* const that);
#ifdef BUILDMODE != 0
inline
#endif
VecFloat* SpheroidGetCenter(const Spheroid* const that);

// Get the percentage of 'tho' included into '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
// delta is the step of the algorithm (in ]0.0, 1.0])

```

```

// small -> slow but precise
// big -> fast but rough
float _ShapoidGetCoverageDelta(const Shapoid* const that,
    const Shapoid* const tho, const float delta);

// Update the system of linear equation used to import coordinates
#if BUILDMODE != 0
inline
#endif
void ShapoidUpdateSysLinEqImport(Shapoid* const that);

// Check if shapoid 'that' and 'tho' are equals
#if BUILDMODE != 0
inline
#endif
bool _ShapoidIsEqual(const Shapoid* const that,
    const Shapoid* const tho);

// Add a copy of the Facoid 'that' to the GSet 'set' (containing
// other Facoid), taking care to avoid overlapping Facoid
// The copy of 'that' may be resized or divided
// The Facoid in the set and 'that' must be aligned with the
// coordinates system axis and have
// same dimensions
void FacoidAlignedAddClippedToSet(const Facoid* const that,
    GSetShapoid* const set);

// Check if the Facoid 'that' is completely included into the Facoid
// 'facoid'
// Both Facoid must be aligned with the coordinates system and have
// same dimensions
// Return true if it is included, false else
bool FacoidAlignedIsInsideFacoidAligned(const Facoid* const that,
    const Facoid* const facoid);

// Check if the Facoid 'that' is completely excluded from the Facoid
// 'facoid'
// Both Facoid must be aligned with the coordinates system and have
// same dimensions
// Return true if it is excluded, false else
bool FacoidAlignedIsOutsideFacoidAligned(const Facoid* const that,
    const Facoid* const facoid);

// Get a GSet of Facoid aligned with coordinates system covering the
// Facoid 'that' except for area in the Facoid 'facoid'
// Both Facoid must be aligned with the coordinates system and have
// same dimensions
GSetShapoid* FacoidAlignedSplitExcludingFacoidAligned(
    const Facoid* const that, const Facoid* const facoid);

// Return true if 'that' intersects 'tho'
// Return false else
// 'that' and 'tho' must have same dimension
bool _SpheroidIsInterSpheroid(const Spheroid* const that,
    const Spheroid* const tho);

// Update the major and minor axis of the Spheroid 'that'
void SpheroidUpdateMajMinAxis(Spheroid* const that);

// Get the maximum distance from the center of the Shapoid 'that' and
// its surface
// Currenty only defined for spheroid, return 0.0 else

```



```

float _ShapoidGetBoundingRadius(const Shapoid* const that);

// ----- ShapoidIter

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

typedef struct ShapoidIter {
    // Attached shapoid
    const Shapoid* _shap;
    // Delta step
    VecFloat* _delta;
    // Current position (in internal coordinates of the shapoid)
    VecFloat* _pos;
} ShapoidIter;

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

// Create a new iterator on the Shapoid 'shap' with a step of 'delta'
// (step on the internal coordinates of the Shapoid)
// The iterator is initialized and ready to be stepped
ShapoidIter _ShapoidIterCreateStatic(const Shapoid* const shap,
    const VecFloat* const delta);

// Free the memory used by the ShapoidIter 'that'
void ShapoidIterFreeStatic(ShapoidIter* const that);

// Reinitialise the ShapoidIter 'that' to its starting position
void ShapoidIterInit(ShapoidIter* const that);

// Step the ShapoidIter 'that'
// Return false if the iterator is at its end and couldn't be stepped
bool ShapoidIterStep(ShapoidIter* const that);

// Return the current position in Shapoid coordinates of the
// ShapoidIter 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* ShapoidIterGetInternalPos(const ShapoidIter* const that);

// Return the current position in standard coordinates of the
// ShapoidIter 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* ShapoidIterGetExternalPos(const ShapoidIter* const that);

// Set the attached Shapoid of the ShapoidIter 'that' to 'shap'
// The iterator is reset to its initial position
#if BUILDMODE != 0
inline
#endif
void _ShapoidIterSetShapoid(ShapoidIter* const that,
    const Shapoid* const shap);

// Get the Shapoid of the ShapoidIter 'that'
#if BUILDMODE != 0
inline
#endif
const Shapoid* ShapoidIterShapoid(const ShapoidIter* const that);

// Set the delta of the ShapoidIter 'that' to a copy of 'delta'

```

```

#if BUILDMODE != 0
inline
#endif
void _ShapoidIterSetDelta(ShapoidIter* const that,
    const VecFloat* const delta);

// Get the delta of the ShapoidIter 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* ShapoidIterDelta(const ShapoidIter* const that);

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

#define ShapoidClone(Shap) _Generic(Shap, \
    Shapoid*: _ShapoidClone, \
    Facoid*: FacoidClone, \
    Pyramidoid*: PyramidoidClone, \
    Spheroid*: SpheroidClone, \
    default: PBErrInvalidPolymorphism)(Shap)

#define ShapoidFree(ShapRef) _Generic(ShapRef, \
    Shapoid**: _ShapoidFree, \
    Facoid**: _ShapoidFree, \
    Pyramidoid**: _ShapoidFree, \
    Spheroid**: _ShapoidFree, \
    default: PBErrInvalidPolymorphism)((Shapoid**)(ShapRef))

#define ShapoidEncodeAsJSON(Shap) _Generic(Shap, \
    Shapoid*: _ShapoidEncodeAsJSON, \
    Facoid*: _ShapoidEncodeAsJSON, \
    Pyramidoid*: _ShapoidEncodeAsJSON, \
    Spheroid*: _ShapoidEncodeAsJSON, \
    const Shapoid*: _ShapoidEncodeAsJSON, \
    const Facoid*: _ShapoidEncodeAsJSON, \
    const Pyramidoid*: _ShapoidEncodeAsJSON, \
    const Spheroid*: _ShapoidEncodeAsJSON, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)Shap)

#define ShapoidDecodeAsJSON(ShapRef, Json) _Generic(ShapRef, \
    Shapoid**: _ShapoidDecodeAsJSON, \
    Facoid**: _ShapoidDecodeAsJSON, \
    Pyramidoid**: _ShapoidDecodeAsJSON, \
    Spheroid**: _ShapoidDecodeAsJSON, \
    default: PBErrInvalidPolymorphism)((Shapoid**)ShapRef, Json)

#define ShapoidLoad(ShapRef, Stream) _Generic(ShapRef, \
    Shapoid**: _ShapoidLoad, \
    Facoid**: FacoidLoad, \
    Pyramidoid**: PyramidoidLoad, \
    Spheroid**: SpheroidLoad, \
    default: PBErrInvalidPolymorphism)(ShapRef, Stream)

#define ShapoidSave(Shap, Stream, Compact) _Generic(Shap, \
    Shapoid*: _ShapoidSave, \
    Facoid*: _ShapoidSave, \
    Pyramidoid*: _ShapoidSave, \
    Spheroid*: _ShapoidSave, \
    const Shapoid*: _ShapoidSave, \
    const Facoid*: _ShapoidSave, \
    const Pyramidoid*: _ShapoidSave, \
    const Spheroid*: _ShapoidSave, \

```

```

default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap), \
    Stream, Compact)

#define ShapoidPrintln(Shap, Stream) _Generic(Shap, \
    Shapoid*: _ShapoidPrintln, \
    Facoid*: _ShapoidPrintln, \
    Pyramidoid*: _ShapoidPrintln, \
    Spheroid*: _ShapoidPrintln, \
    const Shapoid*: _ShapoidPrintln, \
    const Facoid*: _ShapoidPrintln, \
    const Pyramidoid*: _ShapoidPrintln, \
    const Spheroid*: _ShapoidPrintln, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap), Stream)

#define ShapoidGetType(Shap) _Generic(Shap, \
    Shapoid*: _ShapoidGetType, \
    Facoid*: _ShapoidGetType, \
    Pyramidoid*: _ShapoidGetType, \
    Spheroid*: _ShapoidGetType, \
    const Shapoid*: _ShapoidGetType, \
    const Facoid*: _ShapoidGetType, \
    const Pyramidoid*: _ShapoidGetType, \
    const Spheroid*: _ShapoidGetType, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap))

#define ShapoidGetTypeAsString(Shap) _Generic(Shap, \
    Shapoid*: _ShapoidGetTypeAsString, \
    Facoid*: _ShapoidGetTypeAsString, \
    Pyramidoid*: _ShapoidGetTypeAsString, \
    Spheroid*: _ShapoidGetTypeAsString, \
    const Shapoid*: _ShapoidGetTypeAsString, \
    const Facoid*: _ShapoidGetTypeAsString, \
    const Pyramidoid*: _ShapoidGetTypeAsString, \
    const Spheroid*: _ShapoidGetTypeAsString, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap))

#define ShapoidGetDim(Shap) _Generic(Shap, \
    Shapoid*: _ShapoidGetDim, \
    Facoid*: _ShapoidGetDim, \
    Pyramidoid*: _ShapoidGetDim, \
    Spheroid*: _ShapoidGetDim, \
    const Shapoid*: _ShapoidGetDim, \
    const Facoid*: _ShapoidGetDim, \
    const Pyramidoid*: _ShapoidGetDim, \
    const Spheroid*: _ShapoidGetDim, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap))

#define ShapoidGetPos(Shap) _Generic(Shap, \
    Shapoid*: _ShapoidGetPos, \
    Facoid*: _ShapoidGetPos, \
    Pyramidoid*: _ShapoidGetPos, \
    Spheroid*: _ShapoidGetPos, \
    const Shapoid*: _ShapoidGetPos, \
    const Facoid*: _ShapoidGetPos, \
    const Pyramidoid*: _ShapoidGetPos, \
    const Spheroid*: _ShapoidGetPos, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap))

#define ShapoidPos(Shap) _Generic(Shap, \
    Shapoid*: _ShapoidPos, \
    Facoid*: _ShapoidPos, \
    Pyramidoid*: _ShapoidPos, \

```

```

Spheroid*: _ShapoidPos, \
const Shapoid*: _ShapoidPos, \
const Facoid*: _ShapoidPos, \
const Pyramidoid*: _ShapoidPos, \
const Spheroid*: _ShapoidPos, \
default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap))

#define ShapoidSetAxis(Shap, Index, Vec) _Generic(Shap, \
    Shapoid*: _ShapoidSetAxis, \
    Facoid*: _ShapoidSetAxis, \
    Pyramidoid*: _ShapoidSetAxis, \
    Spheroid*: _ShapoidSetAxis, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Index, \
        (VecFloat*)Vec)

#define ShapoidSetPos(Shap, Vec) _Generic(Shap, \
    Shapoid*: _ShapoidSetPos, \
    Facoid*: _ShapoidSetPos, \
    Pyramidoid*: _ShapoidSetPos, \
    Spheroid*: _ShapoidSetPos, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), (VecFloat*)Vec)

#define ShapoidPosSet(Shap, Index, Val) _Generic(Shap, \
    Shapoid*: _ShapoidPosSet, \
    Facoid*: _ShapoidPosSet, \
    Pyramidoid*: _ShapoidPosSet, \
    Spheroid*: _ShapoidPosSet, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Index, Val)

#define ShapoidPosSetAdd(Shap, Index, Val) _Generic(Shap, \
    Shapoid*: _ShapoidPosSetAdd, \
    Facoid*: _ShapoidPosSetAdd, \
    Pyramidoid*: _ShapoidPosSetAdd, \
    Spheroid*: _ShapoidPosSetAdd, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Index, Val)

#define ShapoidSetCenterPos(Shap, Vec) _Generic(Shap, \
    Shapoid*: _ShapoidSetCenterPos, \
    Facoid*: _ShapoidSetCenterPos, \
    Pyramidoid*: _ShapoidSetCenterPos, \
    Spheroid*: _ShapoidSetCenterPos, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), (VecFloat*)Vec)

#define ShapoidGetAxis(Shap, Index) _Generic(Shap, \
    Shapoid*: _ShapoidGetAxis, \
    Facoid*: _ShapoidGetAxis, \
    Pyramidoid*: _ShapoidGetAxis, \
    Spheroid*: _ShapoidGetAxis, \
    const Shapoid*: _ShapoidGetAxis, \
    const Facoid*: _ShapoidGetAxis, \
    const Pyramidoid*: _ShapoidGetAxis, \
    const Spheroid*: _ShapoidGetAxis, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap), Index)

#define ShapoidAxis(Shap, Index) _Generic(Shap, \
    Shapoid*: _ShapoidAxis, \
    Facoid*: _ShapoidAxis, \
    Pyramidoid*: _ShapoidAxis, \
    Spheroid*: _ShapoidAxis, \
    const Shapoid*: _ShapoidAxis, \
    const Facoid*: _ShapoidAxis, \
    const Pyramidoid*: _ShapoidAxis, \

```

```

    const Spheroid*: _ShapoidAxis, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap), Index)

#define ShapoidAxisSet(Shap, Dim, Index, Val) _Generic(Shap, \
    Shapoid*: _ShapoidAxisSet, \
    Facoid*: _ShapoidAxisSet, \
    Pyramidoid*: _ShapoidAxisSet, \
    Spheroid*: _ShapoidAxisSet, \
    const Shapoid*: _ShapoidAxisSet, \
    const Facoid*: _ShapoidAxisSet, \
    const Pyramidoid*: _ShapoidAxisSet, \
    const Spheroid*: _ShapoidAxisSet, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Dim, Index, Val)

#define ShapoidAxisSetAdd(Shap, Dim, Index, Val) _Generic(Shap, \
    Shapoid*: _ShapoidAxisSetAdd, \
    Facoid*: _ShapoidAxisSetAdd, \
    Pyramidoid*: _ShapoidAxisSetAdd, \
    Spheroid*: _ShapoidAxisSetAdd, \
    const Shapoid*: _ShapoidAxisSetAdd, \
    const Facoid*: _ShapoidAxisSetAdd, \
    const Pyramidoid*: _ShapoidAxisSetAdd, \
    const Spheroid*: _ShapoidAxisSetAdd, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Dim, Index, Val)

#define ShapoidTranslate(Shap, Vec) _Generic(Shap, \
    Shapoid*: _ShapoidTranslate, \
    Facoid*: _ShapoidTranslate, \
    Pyramidoid*: _ShapoidTranslate, \
    Spheroid*: _ShapoidTranslate, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), (VecFloat*)Vec)

#define ShapoidIsEqual(ShapA, ShapB) _Generic(ShapA, \
    Shapoid*: _Generic(ShapB, \
        Shapoid*: _ShapoidIsEqual, \
        Facoid*: _ShapoidIsEqual, \
        Pyramidoid*: _ShapoidIsEqual, \
        Spheroid*: _ShapoidIsEqual, \
        const Shapoid*: _ShapoidIsEqual, \
        const Facoid*: _ShapoidIsEqual, \
        const Pyramidoid*: _ShapoidIsEqual, \
        const Spheroid*: _ShapoidIsEqual, \
        default: PBErrInvalidPolymorphism), \
    Facoid*: _Generic(ShapB, \
        Shapoid*: _ShapoidIsEqual, \
        Facoid*: _ShapoidIsEqual, \
        Pyramidoid*: _ShapoidIsEqual, \
        Spheroid*: _ShapoidIsEqual, \
        const Shapoid*: _ShapoidIsEqual, \
        const Facoid*: _ShapoidIsEqual, \
        const Pyramidoid*: _ShapoidIsEqual, \
        const Spheroid*: _ShapoidIsEqual, \
        default: PBErrInvalidPolymorphism), \
    Pyramidoid*: _Generic(ShapB, \
        Shapoid*: _ShapoidIsEqual, \
        Facoid*: _ShapoidIsEqual, \
        Pyramidoid*: _ShapoidIsEqual, \
        Spheroid*: _ShapoidIsEqual, \
        const Shapoid*: _ShapoidIsEqual, \
        const Facoid*: _ShapoidIsEqual, \
        const Pyramidoid*: _ShapoidIsEqual, \
        const Spheroid*: _ShapoidIsEqual, \

```

```

        default: PBErrInvalidPolymorphism), \
Spheroid*: _Generic(ShapB, \
    Shapoid*: _ShapoidIsEqual, \
    Facoid*: _ShapoidIsEqual, \
    Pyramidoid*: _ShapoidIsEqual, \
    Spheroid*: _ShapoidIsEqual, \
    const Shapoid*: _ShapoidIsEqual, \
    const Facoid*: _ShapoidIsEqual, \
    const Pyramidoid*: _ShapoidIsEqual, \
    const Spheroid*: _ShapoidIsEqual, \
    default: PBErrInvalidPolymorphism), \
default: PBErrInvalidPolymorphism)((const Shapoid*)(ShapA), \
    (const Shapoid*)(ShapB))

#define ShapoidGetBoundingBox(Shap) _Generic(Shap, \
    Shapoid*: _ShapoidGetBoundingBox, \
    Facoid*: FacoidGetBoundingBox, \
    Pyramidoid*: PyramidoidGetBoundingBox, \
    Spheroid*: SpheroidGetBoundingBox, \
    const Shapoid*: _ShapoidGetBoundingBox, \
    const Facoid*: FacoidGetBoundingBox, \
    const Pyramidoid*: PyramidoidGetBoundingBox, \
    const Spheroid*: SpheroidGetBoundingBox, \
    GSetShapoid*: ShapoidGetBoundingBoxSet, \
    const GSetShapoid*: ShapoidGetBoundingBoxSet, \
    default: PBErrInvalidPolymorphism)(Shap)

#define ShapoidScale(Shap, Scale) _Generic(Scale, \
    VecFloat*: _ShapoidScaleVector, \
    VecFloat2D*: _ShapoidScaleVector, \
    VecFloat3D*: _ShapoidScaleVector, \
    float: _ShapoidScaleScalar, \
    const VecFloat*: _ShapoidScaleVector, \
    const VecFloat2D*: _ShapoidScaleVector, \
    const VecFloat3D*: _ShapoidScaleVector, \
    const float: _ShapoidScaleScalar, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Scale)

#define ShapoidGrow(Shap, Scale) _Generic(Shap, \
    Shapoid*: _Generic(Scale, \
        VecFloat*: _ShapoidGrowVector, \
        float: _ShapoidGrowScalar, \
        default: PBErrInvalidPolymorphism), \
    Facoid*: _Generic(Scale, \
        VecFloat*: _ShapoidGrowVector, \
        float: _ShapoidGrowScalar, \
        default: PBErrInvalidPolymorphism), \
    Pyramidoid*: _Generic(Scale, \
        VecFloat*: _ShapoidGrowVector, \
        float: _ShapoidGrowScalar, \
        default: PBErrInvalidPolymorphism), \
    Spheroid*: _Generic(Scale, \
        VecFloat*: _ShapoidGrowVector, \
        float: _ShapoidGrowScalar, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Scale)

#define ShapoidGetCenter(Shap) _Generic(Shap, \
    Shapoid*: _ShapoidGetCenter, \
    Facoid*: FacoidGetCenter, \
    Pyramidoid*: PyramidoidGetCenter, \
    Spheroid*: SpheroidGetCenter, \

```

```

const Shapoid*: _ShapoidGetCenter, \
const Facoid*: FacoidGetCenter, \
const Pyramidoid*: PyramidoidGetCenter, \
const Spheroid*: SpheroidGetCenter, \
default: PBErrInvalidPolymorphism)(Shap)

#define ShapoidRotCenter(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotCenter, \
    Facoid*: _ShapoidRotCenter, \
    Pyramidoid*: _ShapoidRotCenter, \
    Spheroid*: _ShapoidRotCenter, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidRotOrigin(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotOrigin, \
    Facoid*: _ShapoidRotOrigin, \
    Pyramidoid*: _ShapoidRotOrigin, \
    Spheroid*: _ShapoidRotOrigin, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidRotStart(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotStart, \
    Facoid*: _ShapoidRotStart, \
    Pyramidoid*: _ShapoidRotStart, \
    Spheroid*: _ShapoidRotStart, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidRotAxisCenter(Shap, Axis, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotAxisCenter, \
    Facoid*: _ShapoidRotAxisCenter, \
    Pyramidoid*: _ShapoidRotAxisCenter, \
    Spheroid*: _ShapoidRotAxisCenter, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Axis, Theta)

#define ShapoidRotAxisOrigin(Shap, Axis, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotAxisOrigin, \
    Facoid*: _ShapoidRotAxisOrigin, \
    Pyramidoid*: _ShapoidRotAxisOrigin, \
    Spheroid*: _ShapoidRotAxisOrigin, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Axis, Theta)

#define ShapoidRotAxisStart(Shap, Axis, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotAxisStart, \
    Facoid*: _ShapoidRotAxisStart, \
    Pyramidoid*: _ShapoidRotAxisStart, \
    Spheroid*: _ShapoidRotAxisStart, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Axis, Theta)

#define ShapoidRotXCenter(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotXCenter, \
    Facoid*: _ShapoidRotXCenter, \
    Pyramidoid*: _ShapoidRotXCenter, \
    Spheroid*: _ShapoidRotXCenter, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidRotXOrigin(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotXOrigin, \
    Facoid*: _ShapoidRotXOrigin, \
    Pyramidoid*: _ShapoidRotXOrigin, \
    Spheroid*: _ShapoidRotXOrigin, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

```

```

#define ShapoidRotXStart(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotXStart, \
    Facoid*: _ShapoidRotXStart, \
    Pyramidoid*: _ShapoidRotXStart, \
    Spheroid*: _ShapoidRotXStart, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidRotYCenter(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotYCenter, \
    Facoid*: _ShapoidRotYCenter, \
    Pyramidoid*: _ShapoidRotYCenter, \
    Spheroid*: _ShapoidRotYCenter, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidRotYOrigin(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotYOrigin, \
    Facoid*: _ShapoidRotYOrigin, \
    Pyramidoid*: _ShapoidRotYOrigin, \
    Spheroid*: _ShapoidRotYOrigin, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidRotYStart(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotYStart, \
    Facoid*: _ShapoidRotYStart, \
    Pyramidoid*: _ShapoidRotYStart, \
    Spheroid*: _ShapoidRotYStart, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidRotZCenter(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotZCenter, \
    Facoid*: _ShapoidRotZCenter, \
    Pyramidoid*: _ShapoidRotZCenter, \
    Spheroid*: _ShapoidRotZCenter, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidRotZOrigin(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotZOrigin, \
    Facoid*: _ShapoidRotZOrigin, \
    Pyramidoid*: _ShapoidRotZOrigin, \
    Spheroid*: _ShapoidRotZOrigin, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidRotZStart(Shap, Theta) _Generic(Shap, \
    Shapoid*: _ShapoidRotZStart, \
    Facoid*: _ShapoidRotZStart, \
    Pyramidoid*: _ShapoidRotZStart, \
    Spheroid*: _ShapoidRotZStart, \
    default: PBErrInvalidPolymorphism)((Shapoid*)(Shap), Theta)

#define ShapoidImportCoord(Shap, VecPos) _Generic(Shap, \
    Shapoid*: _ShapoidImportCoord, \
    Facoid*: _ShapoidImportCoord, \
    Pyramidoid*: _ShapoidImportCoord, \
    Spheroid*: _ShapoidImportCoord, \
    const Shapoid*: _ShapoidImportCoord, \
    const Facoid*: _ShapoidImportCoord, \
    const Pyramidoid*: _ShapoidImportCoord, \
    const Spheroid*: _ShapoidImportCoord, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap), \
    (const VecFloat*)VecPos)

#define ShapoidExportCoord(Shap, VecPos) _Generic(Shap, \

```



```

    Shapoid*: _ShapoidExportCoord, \
    Facoid*: _ShapoidExportCoord, \
    Pyramidoid*: _ShapoidExportCoord, \
    Spheroid*: _ShapoidExportCoord, \
    const Shapoid*: _ShapoidExportCoord, \
    const Facoid*: _ShapoidExportCoord, \
    const Pyramidoid*: _ShapoidExportCoord, \
    const Spheroid*: _ShapoidExportCoord, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap), \
        (const VecFloat*)VecPos)

#define ShapoidIsPosInside(Shap, VecPos) _Generic(Shap, \
    Shapoid*: _ShapoidIsPosInside, \
    Facoid*: FacoidIsPosInside, \
    Pyramidoid*: PyramidoidIsPosInside, \
    Spheroid*: SpheroidIsPosInside, \
    const Shapoid*: _ShapoidIsPosInside, \
    const Facoid*: FacoidIsPosInside, \
    const Pyramidoid*: PyramidoidIsPosInside, \
    const Spheroid*: SpheroidIsPosInside, \
    default: PBErrInvalidPolymorphism)(Shap, VecPos)

#define ShapoidGetPosDepth(Shap, VecPos) _Generic(Shap, \
    Shapoid*: _ShapoidGetPosDepth, \
    Facoid*: FacoidGetPosDepth, \
    Pyramidoid*: PyramidoidGetPosDepth, \
    Spheroid*: SpheroidGetPosDepth, \
    const Shapoid*: _ShapoidGetPosDepth, \
    const Facoid*: FacoidGetPosDepth, \
    const Pyramidoid*: PyramidoidGetPosDepth, \
    const Spheroid*: SpheroidGetPosDepth, \
    default: PBErrInvalidPolymorphism)(Shap, (VecFloat*)VecPos)

#define ShapoidIsInter(ShapA, ShapB) _Generic(ShapA, \
    Spheroid*: _Generic(ShapB, \
        Spheroid*: _SpheroidIsInterSpheroid, \
        const Spheroid*: _SpheroidIsInterSpheroid, \
        default: PBErrInvalidPolymorphism), \
    const Spheroid*: _Generic(ShapB, \
        Spheroid*: _SpheroidIsInterSpheroid, \
        const Spheroid*: _SpheroidIsInterSpheroid, \
        default: PBErrInvalidPolymorphism), \
    default: PBErrInvalidPolymorphism)(ShapA, ShapB)

#define ShapoidGetBoundingRadius(Shap) _Generic(Shap, \
    Shapoid*: _ShapoidGetBoundingRadius, \
    Pyramidoid*: _ShapoidGetBoundingRadius, \
    Facoid*: _ShapoidGetBoundingRadius, \
    Spheroid*: _ShapoidGetBoundingRadius, \
    const Shapoid*: _ShapoidGetBoundingRadius, \
    const Pyramidoid*: _ShapoidGetBoundingRadius, \
    const Facoid*: _ShapoidGetBoundingRadius, \
    const Spheroid*: _ShapoidGetBoundingRadius, \
    default: PBErrInvalidPolymorphism)((const Shapoid*)(Shap))

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

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

```

```
#endif
```

3 Code

3.1 shapoid.c

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

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

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

// ----- Shapoid

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

const char* ShapoidTypeString[3] = {
    (const char*)"Facoid", (const char*)"Spheroid",
    (const char*)"Pyramidoid"};

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

// Create a Shapoid of dimension 'dim' and type 'type', default values:
// _pos = null vector
// _axis[d] = unit vector along dimension d
Shapoid* ShapoidCreate(const int dim, const ShapoidType type) {
#if BUILDMODE == 0
    if (dim <= 0) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "Invalid dimension (%d>0)", dim);
        PBErrCatch(ShapoidErr);
    }
#endif
    // Declare a vector used for initialisation
    VecShort2D d = VecShortCreateStatic2D();
    // Declare a identity matrix used for initialisation
    VecSet(&d, 0, dim);
    VecSet(&d, 1, dim);
    MatFloat* mat = MatFloatCreate(&d);
    MatSetIdentity(mat);
    // Allocate memory
    Shapoid* that = NULL;
    if (type == ShapoidTypeSpheroid)
        that = PBErrMalloc(ShapoidErr, sizeof(Spheroid));
    else
        that = PBErrMalloc(ShapoidErr, sizeof(Shapoid));
    // Init pointers
    that->_pos = NULL;
    that->_axis = NULL;
    that->_sysLinEqImport = NULL;
    // Set the dimension and type
    that->_type = type;
    that->_dim = dim;
```

```

// Allocate memory for position
that->_pos = VecFloatCreate(dim);
// Allocate memory for array of axis
that->_axis = PBErrMalloc(ShapoidErr, sizeof(VecFloat*) * dim);
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);
    // Set value of the axis
    VecSet(that->_axis[iAxis], iAxis, 1.0);
}
// Create the linear system for coordinate importation
that->_sysLinEqImport = SysLinEqCreate(mat, (VecFloat*)NULL);
// Free memory
MatFree(&mat);
// Specific properties of Spheroid
if (type == ShapoidTypeSpheroid) {
    ((Spheroid*)that)->_majAxis = 0;
    ((Spheroid*)that)->_minAxis = 0;
}
// Return the new Shapoid
return that;
}

// Clone a Shapoid
Shapoid* _ShapoidClone(const Shapoid* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Create a clone
    Shapoid* clone = ShapoidCreate(that->_dim, that->_type);
    // Set the position and axis of the clone
    ShapoidSetPos(clone, that->_pos);
    for (int iAxis = clone->_dim; iAxis--;)
        VecCopy(clone->_axis[iAxis], that->_axis[iAxis]);
    ShapoidUpdateSysLinEqImport(clone);
    // Clone the SysLinEq
    SysLinEqFree(&(clone->_sysLinEqImport));
    clone->_sysLinEqImport = SysLinEqClone(that->_sysLinEqImport);
    // If it's a spheroid, copy the spheroid properties too
    if (that->_type == ShapoidTypeSpheroid) {
        ((Spheroid*)clone)->_majAxis = ((Spheroid*)that)->_majAxis;
        ((Spheroid*)clone)->_minAxis = ((Spheroid*)that)->_minAxis;
    }
    // Return the clone
    return clone;
}

// Free memory used by a Shapoid
void _ShapoidFree(Shapoid** that) {
    // Check argument
    if (that == NULL || *that == NULL)
        return;
    // Free memory
    for (int iAxis = (*that)->_dim; iAxis--;)
        VecFree((*that)->_axis + iAxis);
}

```

```

    free((*that)->_axis);
    VecFree(&((*that)->_pos));
    SysLinEqFree(&((*that)->_sysLinEqImport));
    free(*that);
    *that = NULL;
}

// Function which return the JSON encoding of 'that'
JSONNode* _ShapoidEncodeAsJSON(const Shapoid* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Create the JSON structure
    JSONNode* json = JSONCreate();
    // Declare a buffer to convert value into string
    char val[100];
    // Encode the dimension
    sprintf(val, "%d", that->_dim);
    JSONAddProp(json, "_dim", val);
    // Encode the type
    sprintf(val, "%u", that->_type);
    JSONAddProp(json, "_type", val);
    // Encode the position
    JSONAddProp(json, "_pos", VecEncodeAsJSON(ShapoidPos(that)));
    // Encode the axis
    JSONArrayStruct setAxis = JSONArrayStructCreateStatic();
    // For each axis
    for (int iAxis = 0; iAxis < that->_dim; ++iAxis)
        JSONArrayStructAdd(&setAxis,
            VecEncodeAsJSON(ShapoidAxis(that, iAxis)));
    JSONAddProp(json, "_axis", &setAxis);
    // Free memory
    JSONArrayStructFlush(&setAxis);
    // Return the created JSON
    return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool _ShapoidDecodeAsJSON(Shapoid** that, const JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (json == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'json' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // If 'that' is already allocated
    if (*that != NULL)
        // Free memory
        ShapoidFree(that);
    // Get the dim from the JSON
    JSONNode* prop = JSONProperty(json, "_dim");
    if (prop == NULL) {

```

```

        return false;
    }
    int dim = atoi(JSONLabel(JSONValue(prop, 0)));
    // Get the type from the JSON
    prop = JSONProperty(json, "_type");
    if (prop == NULL) {
        return false;
    }
    ShapoidType type = atoi(JSONLabel(JSONValue(prop, 0)));
    // If data are invalid
    if (dim <= 0)
        return false;
    // Allocate memory
    *that = ShapoidCreate(dim, type);
    // Get the position from the JSON
    prop = JSONProperty(json, "_pos");
    if (prop == NULL) {
        return false;
    }
    if (!VecDecodeAsJSON(&((*that)->_pos), prop)) {
        return false;
    }
    // Decode the axis
    prop = JSONProperty(json, "_axis");
    if (prop == NULL) {
        return false;
    }
    if (JSONGetNbValue(prop) != dim) {
        return false;
    }
    for (int iAxis = 0; iAxis < dim; ++iAxis) {
        JSONNode* axis = JSONValue(prop, iAxis);
        if (!VecDecodeAsJSON((*that)->_axis + iAxis, axis))
            return false;
        // If the axis is not of the correct dimension
        if (VecGetDim((*that)->_axis[iAxis]) != (*that)->_dim)
            return false;
    }
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(*that);
    // If it's a Spheroid
    if ((*that)->_type == ShapoidTypeSpheroid)
        // Update the major and minor axis
        SpheroidUpdateMajMinAxis((Spheroid*)*that);
    // Return the success code
    return true;
}

// Load the Shapoid from the stream
// If the Shapoid is already allocated, it is freed before loading
// Return true upon success else false
bool _ShapoidLoad(Shapoid** that, FILE* const stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PErrCatch(ShapoidErr);
    }
    if (stream == NULL) {
        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'stream' is null");
        PErrCatch(ShapoidErr);
    }
#endif
}

```

```

    }
#endif
    // Declare a json to load the encoded data
    JSONNode* json = JSONCreate();
    // Load the whole encoded data
    if (!JSONLoad(json, stream)) {
        return false;
    }
    // Decode the data from the JSON
    if (!ShapoidDecodeAsJSON(that, json)) {
        return false;
    }
    // Free the memory used by the JSON
    JSONFree(&jjson);
    // Return success code
    return true;
}

// Save the Shapoid to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true upon success else false
bool _ShapoidSave(const Shapoid* const that, FILE* const stream,
    const bool compact) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (stream == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'stream' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Get the JSON encoding
    JSONNode* json = ShapoidEncodeAsJSON(that);
    // Save the JSON
    if (!JSONSave(json, stream, compact)) {
        return false;
    }
    // Free memory
    JSONFree(&jjson);
    // Return success code
    return true;
}

// Print the Shapoid on 'stream'
void _ShapoidPrintln(const Shapoid* const that, FILE* const stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (stream == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'stream' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
}

```

```

// 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 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.
Facoid* _ShapoidGetBoundingBox(const Shapoid* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (that->_type != ShapoidTypeFacoid &&
        that->_type != ShapoidTypeSpheroid &&
        that->_type != ShapoidTypePyramidoid) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
        PBErrCatch(ShapoidErr);
    }
#endif
    // If the Shapoid is a Facoid
    if (that->_type == ShapoidTypeFacoid) {
        return FacoidGetBoundingBox((Facoid*)that);
    }
    // Else, if the Shapoid is a Pyramidoid
    else if (that->_type == ShapoidTypePyramidoid) {
        return PyramidoidGetBoundingBox((Pyramidoid*)that);
    }
    // Else, if the Shapoid is a Spheroid
    else if (that->_type == ShapoidTypeSpheroid) {
        return SpheroidGetBoundingBox((Spheroid*)that);
    }
    else {
        return NULL;
    }
}

Facoid* FacoidGetBoundingBox(const Facoid* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Declare a variable to memorize the result
    Facoid* res = FacoidCreate(ShapoidGetDim(that));
    // For each axis
    for (int dim = ShapoidGetDim(that); dim--;) {
        // Declare a variable to memorize the bound of the interval on
        // this axis
        float bound[2];

```

```

    bound[0] = bound[1] = VecGet(((Shapoid*)that)->_pos, dim);
    // For each parameter
    for (int param = ShapoidGetDim(that); param--;) {
        // Get the value of the axis influencing the current dimension
        float v = VecGet(((Shapoid*)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(((Shapoid*)res)->_pos, dim, bound[0]);
    VecSet(((Shapoid*)res)->_axis[dim], dim, bound[1] - bound[0]);
}
// Return the result
return res;
}

Facoid* PyramidoidGetBoundingBox(const Pyramidoid* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Declare a variable to memorize the result
    Facoid* res = FacoidCreate(ShapoidGetDim(that));
    // For each axis
    for (int dim = ShapoidGetDim(that); dim--;) {
        // Declare a variable to memorize the bound of the interval on
        // this axis
        float bound[2];
        bound[0] = bound[1] = 0.0;
        // For each parameter
        for (int param = ShapoidGetDim(that); param--;) {
            // Get the value of the axis influencing the current dimension
            float v = VecGet(((Shapoid*)that)->_axis[param], dim);
            // Search the min and max values
            if (v < bound[0])
                bound[0] = v;
            if (v > bound[1])
                bound[1] = v;
        }
        if (bound[0] > 0.0)
            bound[0] = 0.0;
        if (bound[1] < 0.0)
            bound[1] = 0.0;
        // Memorize the result
        VecSet(((Shapoid*)res)->_pos, dim,
            VecGet(ShapoidPos(that), dim) + bound[0]);
        VecSet(((Shapoid*)res)->_axis[dim], dim, bound[1] - bound[0]);
    }
    // Return the result
    return res;
}

Facoid* SpheroidGetBoundingBox(const Spheroid* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {

```



```

        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PErrCatch(ShapoidErr);
    }
#endif
    // Declare a variable to memorize the result
    Facoid* res = FacoidCreate(ShapoidGetDim(that));
    // 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 = ShapoidGetDim(that); dim--;) {
        // Declare a variable to memorize the bound of the interval on
        // this axis
        float bound[2];
        bound[0] = VecGet(((Shapoid*)that)->_pos, dim);
        // Correct position
        // For each parameter
        for (int param = ShapoidGetDim(that); param--;) {
            // Get the value of the axis influencing the current dimension
            float v = VecGet(((Shapoid*)that)->_axis[param], dim);
            // Correct the pos
            bound[0] -= 0.5 * v;
        }
        bound[1] = bound[0];
        // For each parameter
        for (int param = ShapoidGetDim(that); param--;) {
            // Get the value of the axis influencing the current dimension
            float v = VecGet(((Shapoid*)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(((Shapoid*)res)->_pos, dim, bound[0]);
        VecSet(((Shapoid*)res)->_axis[dim], dim, bound[1] - bound[0]);
    }
    // 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.
Facoid* ShapoidGetBoundingBoxSet(const GSetShapoid* const set) {
    #if BUILDMODE == 0
        if (set == NULL) {
            ShapoidErr->_type = PErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'set' is null");
            PErrCatch(ShapoidErr);
        }
        GSetElem* elemCheck = GSetGetElem(set, 0);
        int dim = ((Shapoid*)(elemCheck->_data))->_dim;
        while (elemCheck != NULL) {
            if (((Shapoid*)(elemCheck->_data))->_dim != dim) {

```

```

        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'set' contains Shapoids of various dimensions");
        PBErrCatch(ShapoidErr);
    }
    elemCheck = elemCheck->_next;
}
#endif
// Declare a variable for the result
Facoid* res = NULL;
// Declare a pointer to the elements of the set
GSetElem* elem = GSetGetElem(set, 0);
// 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);
    // Else, this is not the first Shapoid in the set
    } else {
        // Get the bounding box of this shapoid
        Facoid* bound = ShapoidGetBoundingBox(shapoid);
        // For each dimension
        for (int iDim = ShapoidGetDim(res); iDim--;) {
            // Update the bounding box
            if (VecGet(((Shapoid*)bound)->_pos, iDim) <
                VecGet(((Shapoid*)res)->_pos, iDim)) {
                VecSetAdd(((Shapoid*)res)->_axis[iDim], iDim,
                    VecGet(((Shapoid*)res)->_pos, iDim) -
                    VecGet(((Shapoid*)bound)->_pos, iDim));
                VecSet(((Shapoid*)res)->_pos, iDim,
                    VecGet(((Shapoid*)bound)->_pos, iDim));
            }
            if (VecGet(((Shapoid*)bound)->_pos, iDim) +
                VecGet(((Shapoid*)bound)->_axis[iDim], iDim) >
                VecGet(((Shapoid*)res)->_pos, iDim) +
                VecGet(((Shapoid*)res)->_axis[iDim], iDim))
                VecSetAdd(((Shapoid*)res)->_axis[iDim], iDim,
                    VecGet(((Shapoid*)bound)->_axis[iDim], iDim) -
                    VecGet(((Shapoid*)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 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
// delta is the step of the algorithm (in ]0.0, 1.0])
// small -> slow but precise
// big -> fast but rough
float _ShapoidGetCoverageDelta(const Shapoid* const that,
    const Shapoid* const tho, const float delta) {

```

```

#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PErrCatch(ShapoidErr);
    }
    if (tho == NULL) {
        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'tho' is null");
        PErrCatch(ShapoidErr);
    }
    if (ShapoidGetDim(that) != ShapoidGetDim(tho)) {
        ShapoidErr->_type = PErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "Shapoids dimensions are different (%d==%d)",
            ShapoidGetDim(that), ShapoidGetDim(tho));
        PErrCatch(ShapoidErr);
    }
    if (that->_type != ShapoidTypeFacoid &&
        that->_type != ShapoidTypeSpheroid &&
        that->_type != ShapoidTypePyramidoid) {
        ShapoidErr->_type = PErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
        PErrCatch(ShapoidErr);
    }
#endif
    // 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* pStd = NULL;
    // Declare a variable to memorize the last index in dimension
    int lastI = VecGetDim(pRel) - 1;
    // Declare a variable to memorize the max value of coordinates
    float max = 1.0;
    // If 'tho' is a spheroid, correct the start coordinates and range
    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 coordinates in standard system
        pStd = ShapoidExportCoord(tho, pRel);
        // If this position is inside 'tho'
        if (ShapoidIsPosInside(tho, pStd) == true) {
            // If this position is inside 'that'
            if (ShapoidIsPosInside(that, pStd) == true)
                // Increment the ratio
                ratio += 1.0;
            sum += 1.0;
        }
        // Free memory
        VecFree(&pStd);
        // Step the relative coordinates
        int iDim = 0;
        while (iDim >= 0) {
            VecSetAdd(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;
}

// Add a copy of the Facoid 'that' to the GSet 'set' (containing
// other Facoid), taking care to avoid overlapping Facoid
// The copy of 'that' made be resized or divided
// The Facoid in the set and 'that' must be aligned with the
// coordinates system axis and have
// same dimensions
void FacoidAlignedAddClippedToSet(const Facoid* const that,
    GSetShapoid* const set) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (set == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'set' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
// If the set is empty
if (GSetNbElem(set) == 0) {
    // Add a clone of the facoid
    GSetAppend(set, FacoidClone(that));
// Else, the set is not empty
} else {
    // Create a set of sub facoid to be added and initialize it with a
    // clone of 'that'
    GSetShapoid setToAdd = GSetShapoidCreateStatic();
    GSetAppend(&setToAdd, FacoidClone(that));
    // For each sub facoid to add
    GSetIterForward iterToAdd = GSetIterForwardCreateStatic(&setToAdd);
    do {
        // Get the current facoid to add
        Facoid* facoidToAdd = GSetIterGet(&iterToAdd);
        // Declare a flag to skip the loop when possible
        bool flagSkip = false;
        // For each facoid in the set
        GSetIterForward iter = GSetIterForwardCreateStatic(set);
        do {
            // Get the current facoid
            Facoid* facoid = GSetIterGet(&iter);
            // If the facoid to be added is completely included into this
            // facoid
            if (FacoidAlignedIsInsideFacoidAligned(facoidToAdd, facoid)) {
                // This facoid doesn't need to be added, delete it
                ShapoidFree(&facoidToAdd);
            }
        } while (GSetIterForwardCreateStatic(set));
    } while (GSetIterForwardCreateStatic(&setToAdd));
}
}

```

```

        GSetIterGetElem(&iterToAdd)->_data = NULL;
        // And skip the other facoids in the set
        flagSkip = true;
    // Else, if this facoid is completely include in the facoid to
    // be added
    } else if (FacoidAlignedIsInsideFacoidAligned(facoid,
        facoidToAdd)) {
        // Remove the facoid in the set
        ShapoidFree(&facoid);
        GSetIterGetElem(&iter)->_data = NULL;
    // Else, if both facoid are in intersection
    } else if (!FacoidAlignedIsOutsideFacoidAligned(facoidToAdd,
        facoid)) {
        // Split the facoid to be added into new facoids
        // which cover the non intersecting area
        GSetShapoid* split =
            FacoidAlignedSplitExcludingFacoidAligned(facoidToAdd,
                facoid);
        GSetAppendSet(&setToAdd, split);
        GSetFree(&split);
        // Delete the splitted facoid
        ShapoidFree(&facoidToAdd);
        GSetIterGetElem(&iterToAdd)->_data = NULL;
        // And skip the other facoids in the set
        flagSkip = true;
    }
    // Else the facoid to add is completely outside, leave it as
    // it is
    } while (!flagSkip && GSetIterStep(&iter));
} while (GSetIterStep(&iterToAdd));
// When we arrive here the set 'setToAdd' contains the facoids
// to be added to 'set'
GSetAppendSet(set, &setToAdd);
// There may have been deleted facoid, ensure the resulting set
// is clean by removing null pointer
GSetRemoveAll(set, (Shapoid*)NULL);
// Free memory used by the set of sub facoid to add
GSetFlush(&setToAdd);
}
}

// Check if the Facoid 'that' is completely included into the Facoid
// 'facoid'
// Both Facoid must be aligned with the coordinates system
// 'that' and 'facoid' must have same dimensions and have
// same dimensions
// Return true if it is included, false else
bool FacoidAlignedIsInsideFacoidAligned(const Facoid* const that,
    const Facoid* const facoid) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PErrCatch(ShapoidErr);
    }
    if (facoid == NULL) {
        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'facoid' is null");
        PErrCatch(ShapoidErr);
    }
    if (ShapoidGetDim(that) != ShapoidGetDim(facoid)) {
        ShapoidErr->_type = PErrTypeInvalidArg;

```

```

        sprintf(ShapoidErr->_msg,
            "'that' and 'facoid' have different dimensions (%d==%d)",
            ShapoidGetDim(that), ShapoidGetDim(facoid));
        PBErrCatch(ShapoidErr);
    }
#endif
    // Check inclusion for each axis
    for (int iAxis = ShapoidGetDim(that); iAxis--;)
        // If 'that' is outside 'facoid' for this axis
        if (VecGet(ShapoidPos(that), iAxis) <
            VecGet(ShapoidPos(facoid), iAxis) ||
            VecGet(ShapoidPos(that), iAxis) +
            VecGet(ShapoidAxis(that, iAxis), iAxis) >
            VecGet(ShapoidPos(facoid), iAxis) +
            VecGet(ShapoidAxis(facoid, iAxis), iAxis))
            // Return false
            return false;
    // If we reach here it means 'that' is inside 'facoid', return true
    return true;
}

// Check if the Facoid 'that' is completely excluded from the Facoid
// 'facoid'
// Both Facoid must be aligned with the coordinates system and have
// same dimensions
// Return true if it is excluded, false else
bool FacoidAlignedIsOutsideFacoidAligned(const Facoid* const that,
    const Facoid* const facoid) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (facoid == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'facoid' is null");
        PBErrCatch(ShapoidErr);
    }
    if (ShapoidGetDim(that) != ShapoidGetDim(facoid)) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'that' and 'facoid' have different dimensions (%d==%d)",
            ShapoidGetDim(that), ShapoidGetDim(facoid));
        PBErrCatch(ShapoidErr);
    }
#endif
    // Check exclusion for each axis
    for (int iAxis = ShapoidGetDim(that); iAxis--;)
        // If 'that' is outside 'facoid' for this axis
        if (VecGet(ShapoidPos(that), iAxis) >
            VecGet(ShapoidPos(facoid), iAxis) +
            VecGet(ShapoidAxis(facoid, iAxis), iAxis) - PB_MATH_EPSILON ||
            VecGet(ShapoidPos(that), iAxis) +
            VecGet(ShapoidAxis(that, iAxis), iAxis) <
            VecGet(ShapoidPos(facoid), iAxis) + PB_MATH_EPSILON)
            // Return true
            return true;
    // If we reach here it means 'that' intersects 'facoid', return false
    return false;
}

```

```

// Get a GSet of Facoid aligned with coordinates system covering the
// Facoid 'that' except for area in the Facoid 'facoid'
// Both Facoid must be aligned with the coordinates system and have
// same dimensions
GSetShapoid* FacoidAlignedSplitExcludingFacoidAligned(
    const Facoid* const that, const Facoid* const facoid) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (facoid == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'facoid' is null");
        PBErrCatch(ShapoidErr);
    }
    if (ShapoidGetDim(that) != ShapoidGetDim(facoid)) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'that' and 'facoid' have different dimensions (%d==%d)",
            ShapoidGetDim(that), ShapoidGetDim(facoid));
        PBErrCatch(ShapoidErr);
    }
#endif
    // Ladies and Gentleman, here comes the infamous "Gruyere Algorithm"
    // Declare the result GSet
    GSetShapoid* set = GSetShapoidCreate();
    // Declare a clone of the original facoid
    Facoid* src = FacoidClone(that);
    // For each axis
    for (int iAxis = ShapoidGetDim(that); iAxis--;) {
        // If 'src' has area on the left of 'facoid' along this axis
        if (VecGet(ShapoidPos(src), iAxis) <
            VecGet(ShapoidPos(facoid), iAxis)) {
            // Create the facoid made of this area
            Facoid* sub = FacoidClone(src);
            ShapoidAxisSet(sub, iAxis, iAxis,
                VecGet(ShapoidPos(facoid), iAxis) -
                VecGet(ShapoidPos(src), iAxis));
            // Add it to the result set
            GSetAppend(set, sub);
            // Chop the added area from 'src'
            ShapoidAxisSetAdd(src, iAxis, iAxis,
                -1.0 * VecGet(ShapoidAxis(sub, iAxis), iAxis));
            ShapoidPosSet(src, iAxis,
                VecGet(ShapoidPos(facoid), iAxis));
        }
        // If 'src' has area on the right of 'facoid' along this axis
        if (VecGet(ShapoidPos(src), iAxis) +
            VecGet(ShapoidAxis(src, iAxis), iAxis) >
            VecGet(ShapoidPos(facoid), iAxis) +
            VecGet(ShapoidAxis(facoid, iAxis), iAxis)) {
            // Create the facoid made of this area
            Facoid* sub = FacoidClone(src);
            ShapoidAxisSet(sub, iAxis, iAxis,
                (VecGet(ShapoidPos(src), iAxis) +
                 VecGet(ShapoidAxis(src, iAxis), iAxis)) -
                (VecGet(ShapoidPos(facoid), iAxis) +
                 VecGet(ShapoidAxis(facoid, iAxis), iAxis)));
            ShapoidPosSet(sub, iAxis,
                VecGet(ShapoidPos(facoid), iAxis) +

```

```

        VecGet(ShapoidAxis(facoid, iAxis), iAxis));
    // Add it to the result set
    GSetAppend(set, sub);
    // Chop the added area from 'src'
    ShapoidAxisSetAdd(src, iAxis, iAxis,
        -1.0 * VecGet(ShapoidAxis(sub, iAxis), iAxis));
}
// If 'src' is empty
if (ISEQUALF(VecGet(ShapoidAxis(src, iAxis), iAxis), 0.0))
    // End the loop
    iAxis = 0;
}
// Free memory
ShapoidFree(&src);
// Return the result set
return set;
}

// Return true if 'that' intersects 'tho'
// Return false else
// 'that' and 'tho' must have same dimension
// https://hal.inria.fr/hal-00646511/PDF/CCD.3.0.pdf
bool _SpheroidIsInterSpheroid(const Spheroid* const that,
    const Spheroid* const tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (tho == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'tho' is null");
        PBErrCatch(ShapoidErr);
    }
    if (ShapoidGetDim(that) != ShapoidGetDim(tho)) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'that' and 'tho' have different dimensions (%d==%d)",
            ShapoidGetDim(that), ShapoidGetDim(tho));
        PBErrCatch(ShapoidErr);
    }
#endif
    // Create the projection of 'tho' in 'that' 's coordinates space
    Spheroid* proj = SpheroidCreate(ShapoidGetDim(tho));
    VecFloat* v = ShapoidImportCoord(that, ShapoidPos(tho));
    ShapoidSetPos(proj, v);
    VecFree(&v);
    for (int iAxis = ShapoidGetDim(tho); iAxis--;) {
        v = ShapoidImportCoord(that, ShapoidAxis(tho, iAxis));
        ShapoidSetAxis(proj, iAxis, v);
        VecFree(&v);
    }
    // Declare a variable to memorize the distance to the origin of
    // 'that' 's coordinate system
    float dist = VecNorm(ShapoidPos(proj));
    // Check for trivial cases
    float majRadius = 0.5 * VecNorm(ShapoidAxis(proj, proj->_majAxis));
    if (dist > majRadius + 0.5) {
        ShapoidFree(&proj);
        VecFree(&v);
        return false;
    }
}

```



```

} else if (proj->_majAxis == proj->_minAxis) {
    ShapoidFree(&proj);
    VecFree(&v);
    return true;
}
float minRadius = 0.5 * VecNorm(ShapoidAxis(proj, proj->_minAxis));
if (dist <= minRadius + 0.5) {
    ShapoidFree(&proj);
    VecFree(&v);
    return true;
}
// Non trivial case
// Search a position in the projection of 'tho' less than 1.0 units
// from the origin in 'that' 's coordinates space
// Declare a variable to move in the projection's coordinates space
VecFloat* pos = VecFloatCreate(ShapoidGetDim(tho));
// Declare a variable to memorize the derivative
VecFloat* dPos = VecFloatCreate(ShapoidGetDim(tho));
// Declare a variable to memorize the step for derivate calculation
float delta = 0.01;
// Declare a flag to stop the loop in case of deadlock
bool flag = false;
// Loop until we find a solution or deadlock
while (dist > 0.5 && !flag) {
    // Calculate the derivative along each axis
    v = VecFloatCreate(VecGetDim(pos));
    for (int iAxis = ShapoidGetDim(tho); iAxis--;) {
        // Copy the current position
        VecCopy(v, pos);
        // Move a delta along the current axis
        VecSetAdd(v, iAxis, delta);
        // Get the coordinate in 'that' 's coordinates system
        VecFloat* w = ShapoidExportCoord(proj, v);
        // Calculate the distance ot origin of 'that' 's coordinates
        // system
        float dp = VecNorm(w);
        // Free memory
        VecFree(&w);
        // Do the same thing with minus delta
        VecSetAdd(v, iAxis, -2.0 * delta);
        w = ShapoidExportCoord(proj, v);
        float dm = VecNorm(w);
        VecFree(&w);
        // Calculate the derivative along the current axis
        VecSet(dPos, iAxis, (dp - dm) / (2.0 * delta));
    }
    // Free memory
    VecFree(&v);
    // Move toward better solution
    // Declare a variable to memorize the next position
    VecFloat* nPos = VecGetOp(pos, 1.0, dPos, -1.0);
    // Ensure the position stay inside the Spheroid
    if (VecNorm(nPos) > 0.5) {
        VecNormalise(nPos);
        VecScale(nPos, 0.5);
    }
    // If we are stuck to the same position
    if (VecDist(pos, nPos) < PBMath_EPSILON)
        // Stop the loop
        flag = true;
    // Else we keep moving
} else {

```

```

        VecCopy(pos, nPos);
        // Update the current distance
        v = ShapoidExportCoord(proj, pos);
        dist = VecNorm(v);
        VecFree(&v);
    }
    // Free memory
    VecFree(&nPos);
}
// Free memory
ShapoidFree(&proj);
VecFree(&pos);
VecFree(&dPos);
// If we have found a position less than one unit from the origin
// of 'that' 's coordinates system
if (dist <= 0.5)
    // The spheroids intersect
    return true;
else
    // The spheroids do not intersect
    return false;
}

// Update the major and minor axis of the Spheroid 'that'
void SpheroidUpdateMajMinAxis(Spheroid* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    that->_majAxis = 0;
    float maj = VecNorm(ShapoidAxis(that, 0));
    that->_minAxis = 0;
    float min = maj;
    for (int iAxis = ShapoidGetDim(that); iAxis-- && iAxis != 0;) {
        float n = VecNorm(ShapoidAxis(that, iAxis));
        if (n > maj) {
            maj = n;
            that->_majAxis = iAxis;
        } else if (n < min) {
            min = n;
            that->_minAxis = iAxis;
        }
    }
}

// Get the maximum distance from the center of the Shapoid 'that' and
// its surface
// Currently only defined for spheroid, return 0.0 else
float _ShapoidGetBoundingRadius(const Shapoid* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    if (ShapoidGetType(that) == ShapoidTypeSpheroid) {
        return
            VecNorm(ShapoidAxis(that, ((Spheroid*)that)->_majAxis)) * 0.5;
    }
}

```

```

    }
    return 0.0;
}

// ----- ShapoidIter

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

// Step the ShapoidIter 'that' for a Facoid
// Return false if the iterator is at its end and couldn't be stepped
bool _ShapoidIterStepFacoid(ShapoidIter* const that);

// Step the ShapoidIter 'that' for a Pyramidoid
// Return false if the iterator is at its end and couldn't be stepped
bool _ShapoidIterStepPyramidoid(ShapoidIter* const that);

// Step the ShapoidIter 'that' for a Spheroid
// Return false if the iterator is at its end and couldn't be stepped
bool _ShapoidIterStepSpheroid(ShapoidIter* const that);

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

// Create a new iterator on the Shapoid 'shap' with a step of 'delta'
// (step on the internal coordinates of the Shapoid)
// The iterator is initialized and ready to be stepped
ShapoidIter _ShapoidIterCreateStatic(const Shapoid* const shap,
    const VecFloat* const delta) {
    #if BUILDMODE == 0
        if (shap == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'shap' is null");
            PBErrCatch(ShapoidErr);
        }
        if (delta == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'delta' is null");
            PBErrCatch(ShapoidErr);
        }
        if (VecGetDim(delta) != ShapoidGetDim(shap)) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg,
                "'delta' dimensions and 'shap' dimensions don't match (%d==%d)",
                VecGetDim(delta), ShapoidGetDim(shap));
            PBErrCatch(ShapoidErr);
        }
    }
    #endif
    // Declare the new iterator
    ShapoidIter iter;
    // Set properties
    iter._shap = shap;
    iter._delta = VecClone(delta);
    iter._pos = VecFloatCreate(VecGetDim(delta));
    // Init the position
    ShapoidIterInit(&iter);
    // Return the new iterator
    return iter;
}

// Free the memory used by the ShapoidIter 'that'
void ShapoidIterFreeStatic(ShapoidIter* const that) {
    // Check argument
    if (that == NULL)

```

```

        // Nothing to do
        return;
    // Free memory
    VecFree(&(that->_delta));
    VecFree(&(that->_pos));
}

// Reinitialise the ShapoidIter 'that' to its starting position
void ShapoidIterInit(ShapoidIter* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Initialise according to the type of Shapoid
    switch(ShapoidGetType(that->_shap)) {
        case ShapoidTypeFacoid:
        case ShapoidTypePyramidoid:
            VecSetNull(that->_pos);
            break;
        case ShapoidTypeSpheroid:
            VecSetNull(that->_pos);
            VecSet(that->_pos, VecGetDim(that->_pos) - 1, -0.5);
            break;
        default:
            break;
    }
}

// Step the ShapoidIter 'that'
// Return false if the iterator is at its end and couldn't be stepped
bool ShapoidIterStep(ShapoidIter* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Declare a flag for the return value
    bool flag = true;
    // Step according to the type of Shapoid
    switch(ShapoidGetType(that->_shap)) {
        case ShapoidTypeFacoid:
            flag = _ShapoidIterStepFacoid(that);
            break;
        case ShapoidTypePyramidoid:
            flag = _ShapoidIterStepPyramidoid(that);
            break;
        case ShapoidTypeSpheroid:
            flag = _ShapoidIterStepSpheroid(that);
            break;
        default:
            break;
    }
    return flag;
}

// Step the ShapoidIter 'that' for a Facoid
// Return false if the iterator is at its end and couldn't be stepped

```

```

bool _ShapoidIterStepFacoid(ShapoidIter* const that) {
    // Declare a variable for the returned flag
    bool ret = true;
    // Declare a variable to memorise the dimension currently increasing
    int iDim = VecGetDim(that->_pos) - 1;
    // Declare a flag for the loop condition
    bool flag = true;
    // Increment
    do {
        VecSetAdd(that->_pos, iDim, VecGet(that->_delta, iDim));
        if (VecGet(that->_pos, iDim) > 1.0 + PBMath_EPSILON) {
            VecSet(that->_pos, iDim, 0.0);
            --iDim;
        } else {
            flag = false;
        }
    } while (iDim >= 0 && flag == true);
    if (iDim == -1)
        ret = false;
    // Return the flag
    return ret;
}

// Step the ShapoidIter 'that' for a Pyramidoid
// Return false if the iterator is at its end and couldn't be stepped
bool _ShapoidIterStepPyramidoid(ShapoidIter* const that) {
    // Declare a variable for the returned flag
    bool ret = true;
    // Declare a variable to memorise the dimension currently increasing
    int iDim = VecGetDim(that->_pos) - 1;
    // Declare a flag for the loop condition
    bool flag = true;
    // Increment
    do {
        VecSetAdd(that->_pos, iDim, VecGet(that->_delta, iDim));
        float sum = 0.0;
        for (int iAxis = VecGetDim(that->_pos); iAxis--;)
            sum += VecGet(that->_pos, iAxis);
        if (sum > 1.0 + PBMath_EPSILON) {
            VecSet(that->_pos, iDim, 0.0);
            --iDim;
        } else {
            flag = false;
        }
    } while (iDim >= 0 && flag == true);
    if (iDim == -1)
        ret = false;
    // Return the flag
    return ret;
}

// Step the ShapoidIter 'that' for a Spheroid
// Return false if the iterator is at its end and couldn't be stepped
bool _ShapoidIterStepSpheroid(ShapoidIter* const that) {
    // Declare a variable to memorise the dimension currently increasing
    int iDim = 0;
    // Declare a flag for the loop condition
    bool flag = true;
    // Declare a variable to memorize the norm of the current position
    float norm = VecNorm(that->_pos);
    // Ladies and Gentleman, here comes the infamous "Worm Algorithm"
    // Increment from the first axis

```

```

for (iDim = 0; iDim < VecGetDim(that->_pos) && flag == true; ++iDim) {
    float prevNorm = norm;
    // Try to step in this axis
    VecSetAdd(that->_pos, iDim, VecGet(that->_delta, iDim));
    // Get the norm of the new position
    norm = VecNorm(that->_pos);
    // If we have just jumped over the boundary
    if (prevNorm < 0.5 - PBMath_EPSILON &&
        norm >= 0.5 + PBMath_EPSILON) {
        // Correct the step to reach exactly the boundary
        // Set the current axis to relax the constraint
        VecSet(that->_pos, iDim, 0.0);
        // Calculate the value for this axis which put back the position
        // at the boundary of the Spheroid (on positive side as we want
        // the end of the boundary)
        norm = VecNorm(that->_pos);
        float val = 0.5 * sqrt(-4.0 * (fastpow(norm, 2) - 0.25));
        VecSet(that->_pos, iDim, val);
        // Correct the norm
        norm = 0.5;
        // We could step on this axis, stop here
        flag = false;
        // To cancel the increment in the loop
        --iDim;
    } else {
        // If the new position is out of bound it means we reach the
        // boundary
        if (norm >= 0.5 + PBMath_EPSILON) {
            // Set the current axis to 0.0 to relax the constraint on
            // other axis
            VecSet(that->_pos, iDim, 0.0);
        } else {
            // We could step on this axis, stop here
            flag = false;
            // To cancel the increment in the loop
            --iDim;
        }
    }
}
// If we could step, it has modified the constraint on the previous
// axis which must then be updated
if (flag == false) {
    --iDim;
    // If there is actually a previous axis
    if (iDim >= 0) {
        // Calculate the value for this axis which put back the position
        // at the boundary of the Spheroid (on negative side as we will
        // increment from there)
        float val = VecGet(that->_pos, iDim) +
            0.5 * (-2.0 * VecGet(that->_pos, iDim) -
                sqrt(4.0 * (fastpow(VecGet(that->_pos, iDim), 2) -
                    fastpow(norm, 2) + 0.25)));
        VecSet(that->_pos, iDim, val);
    }
}
// Return the negative of the flag
return !flag;
}

```

3.2 shapoid-inline.c

```
// ===== SHAPOID-INLINE.C =====

// ----- Shapoid

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

// Get the dimension of the Shapoid
#if BUILDMODE != 0
inline
#endif
int _ShapoidGetDim(const Shapoid* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Return the dimension
    return that->_dim;
}

// Get the dimension of the Shapoid
#if BUILDMODE != 0
inline
#endif
ShapoidType _ShapoidGetType(const Shapoid* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // 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)
#if BUILDMODE != 0
inline
#endif
const char* _ShapoidGetTypeAsString(const Shapoid* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Return the type
    return ShapoidTypeString[that->_type];
}

// Return a VecFloat equals to the position of the Shapoid
#if BUILDMODE != 0
inline
#endif
VecFloat* _ShapoidGetPos(const Shapoid* const that) {
```

```

#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Return a clone of the position
    return VecClone(that->_pos);
}

// Return a VecFloat equals to the 'dim'-th axis of the Shapoid
#if BUILDMODE != 0
inline
#endif
VecFloat* _ShapoidGetAxis(const Shapoid* const that, const int dim) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (dim < 0 || dim >= that->_dim) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "Axis' index is invalid (0<=%d<%d)",
            dim, that->_dim);
        PBErrCatch(ShapoidErr);
    }
#endif
    // Return a clone of the axis
    return VecClone(that->_axis[dim]);
}

// Return the position of the Shapoid
#if BUILDMODE != 0
inline
#endif
const VecFloat* _ShapoidPos(const Shapoid* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Return the position
    return that->_pos;
}

// Return the 'dim'-th axis of the Shapoid
#if BUILDMODE != 0
inline
#endif
const VecFloat* _ShapoidAxis(const Shapoid* const that, const int dim) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (dim < 0 || dim >= that->_dim) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
    }
#endif
}

```



```

        sprintf(ShapoidErr->_msg, "Axis' index is invalid (0<=%d<%d)",
            dim, that->_dim);
        PBErriCatch(ShapoidErr);
    }
#endif
    // Return the axis
    return that->_axis[dim];
}

// Set the position of the Shapoid to 'pos'
#if BUILDMODE != 0
inline
#endif
void _ShapoidSetPos(Shapoid* const that, const VecFloat* const pos) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErriTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErriCatch(ShapoidErr);
    }
    if (pos == NULL) {
        ShapoidErr->_type = PBErriTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'pos' is null");
        PBErriCatch(ShapoidErr);
    }
    if (VecGetDim(pos) != that->_dim) {
        ShapoidErr->_type = PBErriTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
            VecGetDim(pos), that->_dim);
        PBErriCatch(ShapoidErr);
    }
#endif
    // Set the position
    VecCopy(that->_pos, pos);
}

// Set the 'iElem'-th value of the position of the Shapoid to 'val'
#if BUILDMODE != 0
inline
#endif
void _ShapoidPosSet(Shapoid* const that, const int iElem,
    const float val) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErriTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErriCatch(ShapoidErr);
    }
    if (iElem < 0 || iElem >= that->_dim) {
        ShapoidErr->_type = PBErriTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'iElem' is invalid (0<=%d<%d)",
            iElem, that->_dim);
        PBErriCatch(ShapoidErr);
    }
#endif
    // Set the position
    VecSet(that->_pos, iElem, val);
}

// Set the 'iElem'-th value of the position of the Shapoid to 'val'
// added to its current value
#if BUILDMODE != 0

```

```

inline
#endif
void _ShapoidPosSetAdd(Shapoid* const that, const int iElem,
    const float val) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (iElem < 0 || iElem >= that->_dim) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'iElem' is invalid (0<=%d<=%d)",
            iElem, that->_dim);
        PBErrCatch(ShapoidErr);
    }
#endif
    // Set the position
    VecSetAdd(that->_pos, iElem, val);
}

// Set the position of the Shapoid such as its center is at 'pos'
#ifdef BUILDMODE != 0
inline
#endif
void _ShapoidSetCenterPos(Shapoid* const that,
    const VecFloat* const pos) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (pos == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'pos' is null");
        PBErrCatch(ShapoidErr);
    }
    if (VecGetDim(pos) != that->_dim) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
            VecGetDim(pos), that->_dim);
        PBErrCatch(ShapoidErr);
    }
#endif
    VecFloat* v = ShapoidGetCenter(that);
    VecOp(v, -1.0, ShapoidPos(that), 1.0);
    VecOp(v, 1.0, pos, 1.0);
    ShapoidSetPos(that, v);
    VecFree(&v);
}

// Set the 'dim'-th axis of the Shapoid to 'v'
#ifdef BUILDMODE != 0
inline
#endif
void _ShapoidSetAxis(Shapoid* const that, const int dim,
    const VecFloat* const v) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;

```

```

        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErCatch(ShapoidErr);
    }
    if (v == NULL) {
        ShapoidErr->_type = PBErTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'v' is null");
        PBErCatch(ShapoidErr);
    }
    if (dim < 0 || dim >= that->_dim) {
        ShapoidErr->_type = PBErTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "Axis' index is invalid (0<=%d<%d)",
            dim, that->_dim);
        PBErCatch(ShapoidErr);
    }
    if (VecGetDim(v) != that->_dim) {
        ShapoidErr->_type = PBErTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'v' 's dimension is invalid (%d=%d)",
            dim, VecGetDim(v));
        PBErCatch(ShapoidErr);
    }
}
#endif
// Set the axis
VecCopy(that->_axis[dim], v);
// Update the SysLinEq
ShapoidUpdateSysLinEqImport(that);
// If it's a Spheroid
if (that->_type == ShapoidTypeSpheroid)
    // Update the major and minor axis
    SpheroidUpdateMajMinAxis((Spheroid*)that);
}

// Set the 'iElem'-th element of the 'dim'-th axis of the Shapoid to 'v'
#if BUILDMODE != 0
inline
#endif
void _ShapoidAxisSet(Shapoid* const that, const int dim,
    const int iElem, const float v) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErCatch(ShapoidErr);
        }
        if (dim < 0 || dim >= that->_dim) {
            ShapoidErr->_type = PBErTypeInvalidArg;
            sprintf(ShapoidErr->_msg, "Axis' index is invalid (0<=%d<%d)",
                dim, that->_dim);
            PBErCatch(ShapoidErr);
        }
        if (iElem < 0 || iElem >= VecGetDim(ShapoidAxis(that, dim))) {
            ShapoidErr->_type = PBErTypeInvalidArg;
            sprintf(ShapoidErr->_msg, "iElem is invalid (0<=%d<%d)",
                iElem, VecGetDim(ShapoidAxis(that, dim)));
            PBErCatch(ShapoidErr);
        }
    #endif
    // Set the axis
    VecSet(that->_axis[dim], iElem, v);
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
    // If it's a Spheroid
    if (that->_type == ShapoidTypeSpheroid)

```

```

        // Update the major and minor axis
        SpheroidUpdateMajMinAxis((Spheroid*)that);
    }

    // Set the 'iElem'-th element of the 'dim'-th axis of the Shapoid to
    // 'v' added to its current value
    #if BUILDMODE != 0
    inline
    #endif
    void _ShapoidAxisSetAdd(Shapoid* const that, const int dim,
        const int iElem, const float v) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (dim < 0 || dim >= that->_dim) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg, "Axis' index is invalid (0<=%d<%d)",
                dim, that->_dim);
            PBErrCatch(ShapoidErr);
        }
        if (iElem < 0 || iElem >= VecGetDim(ShapoidAxis(that, dim))) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg, "iElem is invalid (0<=%d<%d)",
                iElem, VecGetDim(ShapoidAxis(that, dim)));
            PBErrCatch(ShapoidErr);
        }
    #endif
        // Set the axis
        VecSetAdd(that->_axis[dim], iElem, v);
        // Update the SysLinEq
        ShapoidUpdateSysLinEqImport(that);
        // If it's a Spheroid
        if (that->_type == ShapoidTypeSpheroid)
            // Update the major and minor axis
            SpheroidUpdateMajMinAxis((Spheroid*)that);
    }

    // Translate the Shapoid by 'v'
    #if BUILDMODE != 0
    inline
    #endif
    void _ShapoidTranslate(Shapoid* const that, const VecFloat* const v) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (v == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'v' is null");
            PBErrCatch(ShapoidErr);
        }
        if (VecGetDim(v) != that->_dim) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg, "'v' 's dimension is invalid (%d==%d)",
                that->_dim, VecGetDim(v));
            PBErrCatch(ShapoidErr);
        }
    #endif

```

```

    }
#endif
    // Translate the position
    VecOp(that->_pos, 1.0, v, 1.0);
}

// Scale the Shapoid by 'v' (each axis is multiplied by v[iAxis])
#if BUILDMODE != 0
inline
#endif
void _ShapoidScaleVector(Shapoid* const that, const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (v == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'v' is null");
        PBErrCatch(ShapoidErr);
    }
    if (VecGetDim(v) != that->_dim) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'v' 's dimension is invalid (%d==%d)",
            that->_dim, VecGetDim(v));
        PBErrCatch(ShapoidErr);
    }
#endif
    // Scale each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecScale(that->_axis[iAxis], VecGet(v, iAxis));
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
    // If it's a Spheroid
    if (that->_type == ShapoidTypeSpheroid)
        // Update the major and minor axis
        SpheroidUpdateMajMinAxis((Spheroid*)that);
}

// Scale the Shapoid by 'c'
#if BUILDMODE != 0
inline
#endif
void _ShapoidScaleScalar(Shapoid* const that, const float c) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Scale each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecScale(that->_axis[iAxis], c);
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
    // If it's a Spheroid
    if (that->_type == ShapoidTypeSpheroid)
        // Update the major and minor axis
        SpheroidUpdateMajMinAxis((Spheroid*)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
#if BUILDMODE != 0
inline
#endif
void _ShapoidGrowVector(Shapoid* const that, const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErCatch(ShapoidErr);
    }
    if (v == NULL) {
        ShapoidErr->_type = PBErTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'v' is null");
        PBErCatch(ShapoidErr);
    }
    if (VecGetDim(v) != that->_dim) {
        ShapoidErr->_type = PBErTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'v' 's dimension is invalid (%d==%d)",
            that->_dim, VecGetDim(v));
        PBErCatch(ShapoidErr);
    }
    if (that->_type != ShapoidTypeFacoid &&
        that->_type != ShapoidTypeSpheroid &&
        that->_type != ShapoidTypePyramidoid) {
        ShapoidErr->_type = PBErTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
        PBErCatch(ShapoidErr);
    }
#endif
    // If the shapoid is a spheroid
    if (that->_type == ShapoidTypeSpheroid) {
        // Scale
        ShapoidScale(that, v);
        // Update the major and minor axis
        SpheroidUpdateMajMinAxis((Spheroid*)that);
    } else {
        // Memorize the center
        VecFloat* centerA = ShapoidGetCenter(that);
        // Scale
        ShapoidScale(that, v);
        // Reposition to keep center at the same position
        VecFloat* centerB = ShapoidGetCenter(that);
        VecOp(centerA, 1.0, centerB, -1.0);
        VecOp(that->_pos, 1.0, centerA, 1.0);
        VecFree(&centerA);
        VecFree(&centerB);
    }
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Scale the Shapoid by 'c'
// and translate the Shapoid such as its center after scaling
// is at the same position than before scaling
#if BUILDMODE != 0
inline
#endif

```

```

void _ShapoidGrowScalar(Shapoid* const that, const float c) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (that->_type != ShapoidTypeFacoid &&
        that->_type != ShapoidTypeSpheroid &&
        that->_type != ShapoidTypePyramidoid) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
        PBErrCatch(ShapoidErr);
    }
#endif
    // If the shapoid is a spheroid
    if (that->_type == ShapoidTypeSpheroid) {
        // Scale
        ShapoidScale(that, c);
        // Update the major and minor axis
        SpheroidUpdateMajMinAxis((Spheroid*)that);
    } // Else, the shapoid is not a spheroid
    else {
        // Memorize the center
        VecFloat* centerA = ShapoidGetCenter(that);
        // Scale
        ShapoidScale(that, c);
        // Reposition to keep center at the same position
        VecFloat* centerB = ShapoidGetCenter(that);
        VecOp(centerA, 1.0, centerB, -1.0);
        VecOp(that->_pos, 1.0, centerA, 1.0);
        VecFree(&centerA);
        VecFree(&centerB);
    }
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 2 by 'theta' (in radians, CCW)
// relatively to its center
#ifdef BUILDMODE != 0
inline
#endif
void _ShapoidRotCenter(Shapoid* const that, const float theta) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (that->_dim != 2) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'that' 's dimension is invalid (%d!=2)", that->_dim);
        PBErrCatch(ShapoidErr);
    }
    if (that->_type != ShapoidTypeFacoid &&
        that->_type != ShapoidTypeSpheroid &&
        that->_type != ShapoidTypePyramidoid) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
        PBErrCatch(ShapoidErr);
    }
#endif
}

```

```

    }
#endif
    // If it's a spheroid
    if (that->_type == ShapoidTypeSpheroid) {
        // Rotate each axis
        for (int iAxis = that->_dim; iAxis--;)
            VecRot(that->_axis[iAxis], theta);
        // Else, it's not a spheroid
    } else {
        VecFloat* center = ShapoidGetCenter(that);
        // Rotate each axis
        for (int iAxis = that->_dim; iAxis--;)
            VecRot(that->_axis[iAxis], theta);
        // Reposition the origin
        VecFloat* v = VecGetOp(that->_pos, 1.0, center, -1.0);
        VecRot(v, theta);
        VecOp(v, 1.0, center, 1.0);
        VecCopy(that->_pos, v);
        VecFree(&center);
        VecFree(&v);
    }
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 2 by 'theta' (in radians, CCW)
// relatively to its position
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotStart(Shapoid* const that, const float theta) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PErrCatch(ShapoidErr);
        }
        if (that->_dim != 2) {
            ShapoidErr->_type = PErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg,
                "'that' 's dimension is invalid (%d==2)", that->_dim);
            PErrCatch(ShapoidErr);
        }
    #endif
    // Rotate each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecRot(that->_axis[iAxis], theta);
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 2 by 'theta' (in radians, CCW)
// relatively to the origin of the global coordinates system
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotOrigin(Shapoid* const that, const float theta) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PErrCatch(ShapoidErr);
        }
    #endif

```



```

    }
    if (that->_dim != 2) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'that' 's dimension is invalid (%d==2)", that->_dim);
        PBErrCatch(ShapoidErr);
    }
#endif
    // Rotate each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecRot(that->_axis[iAxis], theta);
    // Reposition the origin
    VecRot(that->_pos, theta);
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its center around 'axis'
// 'axis' must be normalized
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotAxisCenter(Shapoid* const that,
    const VecFloat3D* const axis, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (axis == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'axis' is null");
        PBErrCatch(ShapoidErr);
    }
    if (VecGetDim(axis) != 3) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'axis' 's dimension is invalid (%d==3)", VecGetDim(axis));
        PBErrCatch(ShapoidErr);
    }
    if (that->_dim != 3) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'that' 's dimension is invalid (%d==3)", that->_dim);
        PBErrCatch(ShapoidErr);
    }
    if (that->_type != ShapoidTypeFacoid &&
        that->_type != ShapoidTypeSpheroid &&
        that->_type != ShapoidTypePyramidoid) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
        PBErrCatch(ShapoidErr);
    }
#endif
    // If it's a spheroid
    if (that->_type == ShapoidTypeSpheroid) {
        // Rotate each axis
        for (int iAxis = that->_dim; iAxis--;)
            VecRotAxis(that->_axis[iAxis], axis, theta);
        // Else, it's not a spheroid
    }
}

```

```

    } else {
        VecFloat* center = ShapoidGetCenter(that);
        // Rotate each axis
        for (int iAxis = that->_dim; iAxis--;)
            VecRotAxis(that->_axis[iAxis], axis, theta);
        // Reposition the origin
        VecFloat* v = VecGetOp(that->_pos, 1.0, center, -1.0);
        VecRotAxis(v, axis, theta);
        VecOp(v, 1.0, center, 1.0);
        VecCopy(that->_pos, v);
        VecFree(&center);
        VecFree(&v);
    }
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its position around 'axis'
// 'axis' must be normalized
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotAxisStart(Shapoid* const that,
    const VecFloat3D* const axis, const float theta) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (axis == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'axis' is null");
            PBErrCatch(ShapoidErr);
        }
        if (that->_dim != 3) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg,
                "'that' 's dimension is invalid (%d==3)", that->_dim);
            PBErrCatch(ShapoidErr);
        }
        if (VecGetDim(axis) != 3) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg,
                "'axis' 's dimension is invalid (%d==3)", VecGetDim(axis));
            PBErrCatch(ShapoidErr);
        }
    #endif
    // Rotate each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecRotAxis(that->_axis[iAxis], axis, theta);
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to the origin of the global coordinates system
// around 'axis'
// 'axis' must be normalized
#if BUILDMODE != 0
inline

```

```

#endif
void _ShapoidRotAxisOrigin(Shapoid* const that,
    const VecFloat3D* const axis, const float theta) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (axis == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'axis' is null");
            PBErrCatch(ShapoidErr);
        }
        if (that->_dim != 3) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg,
                "'that' 's dimension is invalid (%d==3)", that->_dim);
            PBErrCatch(ShapoidErr);
        }
        if (VecGetDim(axis) != 3) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg,
                "'axis' 's dimension is invalid (%d==3)", VecGetDim(axis));
            PBErrCatch(ShapoidErr);
        }
    #endif
    // Rotate each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecRotAxis(that->_axis[iAxis], axis, theta);
    // Reposition the origin
    VecRotAxis(that->_pos, axis, theta);
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its center around X
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotXCenter(Shapoid* const that, const float theta) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (that->_dim != 3) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg,
                "'that' 's dimension is invalid (%d==3)", that->_dim);
            PBErrCatch(ShapoidErr);
        }
        if (that->_type != ShapoidTypeFacoid &&
            that->_type != ShapoidTypeSpheroid &&
            that->_type != ShapoidTypePyramidoid) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
            PBErrCatch(ShapoidErr);
        }
    #endif
}

```

```

// If it's a spheroid
if (that->_type == ShapoidTypeSpheroid) {
// Rotate each axis
for (int iAxis = that->_dim; iAxis--;)
    VecRotX(that->_axis[iAxis], theta);
// Else, it's not a spheroid
} else {
    VecFloat* center = ShapoidGetCenter(that);
    // Rotate each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecRotX(that->_axis[iAxis], theta);
    // Reposition the origin
    VecFloat* v = VecGetOp(that->_pos, 1.0, center, -1.0);
    VecRotX(v, theta);
    VecOp(v, 1.0, center, 1.0);
    VecCopy(that->_pos, v);
    VecFree(&center);
    VecFree(&v);
}
// Update the SysLinEq
ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its position around X
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotXStart(Shapoid* const that,
    float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (that->_dim != 3) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'that' 's dimension is invalid (%d==3)", that->_dim);
        PBErrCatch(ShapoidErr);
    }
}
#endif
// Rotate each axis
for (int iAxis = that->_dim; iAxis--;)
    VecRotX(that->_axis[iAxis], theta);
// Update the SysLinEq
ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to the origin of the global coordinates system
// around X
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotXOrigin(Shapoid* const that,
    float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
    }
}

```

```

    PBErCatch(ShapoidErr);
}
if (that->_dim != 3) {
    ShapoidErr->_type = PBErTypeInvalidArg;
    sprintf(ShapoidErr->_msg,
        "'that' 's dimension is invalid (%d==3)", that->_dim);
    PBErCatch(ShapoidErr);
}
#endif
// Rotate each axis
for (int iAxis = that->_dim; iAxis--;)
    VecRotX(that->_axis[iAxis], theta);
// Reposition the origin
VecRotX(that->_pos, theta);
// Update the SysLinEq
ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its center around Y
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotYCenter(Shapoid* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErCatch(ShapoidErr);
    }
    if (that->_dim != 3) {
        ShapoidErr->_type = PBErTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'that' 's dimension is invalid (%d==3)", that->_dim);
        PBErCatch(ShapoidErr);
    }
    if (that->_type != ShapoidTypeFacoid &&
        that->_type != ShapoidTypeSpheroid &&
        that->_type != ShapoidTypePyramidoid) {
        ShapoidErr->_type = PBErTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
        PBErCatch(ShapoidErr);
    }
#endif
    // If it's a spheroid
    if (that->_type == ShapoidTypeSpheroid) {
        // Rotate each axis
        for (int iAxis = that->_dim; iAxis--;)
            VecRotY(that->_axis[iAxis], theta);
        // Else, it's not a spheroid
    } else {
        VecFloat* center = ShapoidGetCenter(that);
        // Rotate each axis
        for (int iAxis = that->_dim; iAxis--;)
            VecRotY(that->_axis[iAxis], theta);
        // Reposition the origin
        VecFloat* v = VecGetOp(that->_pos, 1.0, center, -1.0);
        VecRotY(v, theta);
        VecOp(v, 1.0, center, 1.0);
        VecCopy(that->_pos, v);
        VecFree(&center);
        VecFree(&v);
    }
}

```

```

    }
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its position around Y
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotYStart(Shapoid* const that,
    float theta) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PErrCatch(ShapoidErr);
        }
        if (that->_dim != 3) {
            ShapoidErr->_type = PErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg,
                "'that' 's dimension is invalid (%d==3)", that->_dim);
            PErrCatch(ShapoidErr);
        }
    #endif
    // Rotate each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecRotY(that->_axis[iAxis], theta);
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to the origin of the global coordinates system
// around Y
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotYOrigin(Shapoid* const that,
    float theta) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PErrCatch(ShapoidErr);
        }
        if (that->_dim != 3) {
            ShapoidErr->_type = PErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg,
                "'that' 's dimension is invalid (%d==3)", that->_dim);
            PErrCatch(ShapoidErr);
        }
    #endif
    // Rotate each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecRotY(that->_axis[iAxis], theta);
    // Reposition the origin
    VecRotY(that->_pos, theta);
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

```

```

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its center around Z
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotZCenter(Shapoid* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PErrCatch(ShapoidErr);
    }
    if (that->_dim != 3) {
        ShapoidErr->_type = PErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'that' 's dimension is invalid (%d==3)", that->_dim);
        PErrCatch(ShapoidErr);
    }
    if (that->_type != ShapoidTypeFacoid &&
        that->_type != ShapoidTypeSpheroid &&
        that->_type != ShapoidTypePyramidoid) {
        ShapoidErr->_type = PErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
        PErrCatch(ShapoidErr);
    }
#endif
    // If it's a spheroid
    if (that->_type == ShapoidTypeSpheroid) {
        // Rotate each axis
        for (int iAxis = that->_dim; iAxis--;)
            VecRotZ(that->_axis[iAxis], theta);
        // Else, it's not a spheroid
    } else {
        VecFloat* center = ShapoidGetCenter(that);
        // Rotate each axis
        for (int iAxis = that->_dim; iAxis--;)
            VecRotZ(that->_axis[iAxis], theta);
        // Reposition the origin
        VecFloat* v = VecGetOp(that->_pos, 1.0, center, -1.0);
        VecRotZ(v, theta);
        VecOp(v, 1.0, center, 1.0);
        VecCopy(that->_pos, v);
        VecFree(&center);
        VecFree(&v);
    }
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to its position around Z
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotZStart(Shapoid* const that,
    float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PErrCatch(ShapoidErr);
    }
}

```

```

    if (that->_dim != 3) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg,
            "'that' 's dimension is invalid (%d==3)", that->_dim);
        PBErrCatch(ShapoidErr);
    }
#endif
    // Rotate each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecRotZ(that->_axis[iAxis], theta);
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Rotate the Shapoid of dimension 3 by 'theta' (in radians, CCW)
// relatively to the origin of the global coordinates system
// around Z
#if BUILDMODE != 0
inline
#endif
void _ShapoidRotZOrigin(Shapoid* const that,
    float theta) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (that->_dim != 3) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg,
                "'that' 's dimension is invalid (%d==3)", that->_dim);
            PBErrCatch(ShapoidErr);
        }
    #endif
    // Rotate each axis
    for (int iAxis = that->_dim; iAxis--;)
        VecRotZ(that->_axis[iAxis], theta);
    // Reposition the origin
    VecRotZ(that->_pos, theta);
    // Update the SysLinEq
    ShapoidUpdateSysLinEqImport(that);
}

// Convert the coordinates of 'pos' from standard coordinate system
// toward the Shapoid coordinates system
#if BUILDMODE != 0
inline
#endif
VecFloat* _ShapoidImportCoord(const Shapoid* const that,
    const VecFloat* const pos) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (pos == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'pos' is null");
            PBErrCatch(ShapoidErr);
        }
    #endif
}

```



```

    if (VecGetDim(pos) != that->_dim) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
            that->_dim, VecGetDim(pos));
        PBErrCatch(ShapoidErr);
    }
#endif
    // Update the system solver for the requested position
    VecFloat* v = VecGetOp(pos, 1.0, that->_pos, -1.0);
    SysLinEqSetV(that->_sysLinEqImport, v);
    // Solve the system
    VecFloat* res = SysLinEqSolve(that->_sysLinEqImport);
    // Free memory
    VecFree(&v);
    // return the result
    return res;
}

#if BUILDMODE != 0
inline
#endif

// Convert the coordinates of 'pos' from the Shapoid coordinates system
// toward standard coordinate system
#if BUILDMODE != 0
inline
#endif
VecFloat* _ShapoidExportCoord(const Shapoid* const that,
    const VecFloat* const pos) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (pos == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'pos' is null");
        PBErrCatch(ShapoidErr);
    }
    if (VecGetDim(pos) != that->_dim) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
            that->_dim, VecGetDim(pos));
        PBErrCatch(ShapoidErr);
    }
#endif
    // Allocate memory for the result
    VecFloat* res = VecClone(that->_pos);
    for (int dim = that->_dim; dim--;)
        VecOp(res, 1.0, that->_axis[dim], VecGet(pos, dim));
    // Return the result
    return res;
}

// Get the center of the shapoid in standard coordinate system
#if BUILDMODE != 0
inline
#endif
VecFloat* _ShapoidGetCenter(const Shapoid* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
    }
#endif
}

```

```

        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErriCatch(ShapoidErr);
    }
    if (that->_type != ShapoidTypeFacoid &&
        that->_type != ShapoidTypeSpheroid &&
        that->_type != ShapoidTypePyramidoid) {
        ShapoidErr->_type = PBErriTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
        PBErriCatch(ShapoidErr);
    }
#endif
    if (ShapoidGetType(that) == ShapoidTypeFacoid)
        return FacoidGetCenter((Facoid*)that);
    else if (ShapoidGetType(that) == ShapoidTypePyramidoid)
        return PyramidoidGetCenter((Pyramidoid*)that);
    else if (ShapoidGetType(that) == ShapoidTypeSpheroid)
        return SpheroidGetCenter((Spheroid*)that);
    else
        return NULL;
}

#if BUILDMODE != 0
inline
#endif
VecFloat* FacoidGetCenter(const Facoid* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErriTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErriCatch(ShapoidErr);
        }
    #endif
    // Declare a variable to memorize the result in Shapoid
    // coordinate system
    VecFloat* coord = VecFloatCreate(ShapoidGetDim(that));
    // For each dimension
    for (int dim = ShapoidGetDim(that); dim--;)
        VecSet(coord, dim, 0.5);
    // Convert the coordinates in standard coordinate system
    VecFloat* res = ShapoidExportCoord(that, coord);
    // Free memory
    VecFree(&coord);
    // Return the result
    return res;
}

#if BUILDMODE != 0
inline
#endif
VecFloat* PyramidoidGetCenter(const Pyramidoid* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErriTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErriCatch(ShapoidErr);
        }
    #endif
    // Declare a variable to memorize the result in Shapoid
    // coordinate system
    VecFloat* coord = VecFloatCreate(ShapoidGetDim(that));
    // For each dimension
    for (int dim = ShapoidGetDim(that); dim--;)

```

```

    VecSet(coord, dim, 1.0 / (1.0 + ShapoidGetDim(that)));
    // Convert the coordinates in standard coordinate system
    VecFloat* res = ShapoidExportCoord(that, coord);
    // Free memory
    VecFree(&coord);
    // Return the result
    return res;
}

#if BUILDMODE != 0
inline
#endif
VecFloat* ShapoidGetCenter(const Shapoid* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Declare a variable to memorize the result in Shapoid
    // coordinate system
    VecFloat* coord = VecFloatCreate(ShapoidGetDim(that));
    // Convert the coordinates in standard coordinate system
    VecFloat* res = ShapoidExportCoord(that, coord);
    // Free memory
    VecFree(&coord);
    // Return the result
    return res;
}

// Check if shapoid 'that' and 'tho' are equals
#if BUILDMODE != 0
inline
#endif
bool _ShapoidIsEqual(const Shapoid* const that,
                    const Shapoid* const tho) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (tho == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'tho' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    // Check the dimension, type and position
    if (that->_dim != tho->_dim ||
        that->_type != tho->_type ||
        VecIsEqual(that->_pos, tho->_pos) == false)
        return false;
    // Check the axis
    for (int i = that->_dim; i--;)
        if (VecIsEqual(that->_axis[i], tho->_axis[i]) == false)
            return false;
    // If the Shapoid is a Spheroid, check Spheroid properties
    if (that->_type == ShapoidTypeSpheroid) {
        if (((Spheroid*)that)->_majAxis != ((Spheroid*)tho)->_majAxis ||
            ((Spheroid*)that)->_minAxis != ((Spheroid*)tho)->_minAxis)
    }
}

```

```

        return false;
    }
    // Return the success code
    return true;
}

// Update the system of linear equation used to import coordinates
#if BUILDMODE != 0
inline
#endif
void ShapoidUpdateSysLinEqImport(Shapoid* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
#endif
    VecShort2D dim = VecShortCreateStatic2D();
    // Set a pointer to the matrix in the SysLinEq
    MatFloat* mat = MatClone(that->_sysLinEqImport->_M);
    // Set the values of the matrix
    for (VecSet(&dim, 0, 0); VecGet(&dim, 0) < that->_dim;
        VecSetAdd(&dim, 0, 1)) {
        for (VecSet(&dim, 1, 0); VecGet(&dim, 1) < that->_dim;
            VecSetAdd(&dim, 1, 1)) {
            MatSet(mat, &dim, VecGet(that->_axis[VecGet(&dim, 0)],
                VecGet(&dim, 1)));
        }
    }
    // Update the SysLinEq
    SysLinEqSetM(that->_sysLinEqImport, mat);
    // Free memory
    MatFree(&mat);
}

// Return true if 'pos' (in stand coordinate system) is inside the
// Shapoid
// Else return false
#if BUILDMODE != 0
inline
#endif
bool _ShapoidIsPosInside(const Shapoid* const that,
    const VecFloat* const pos) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (pos == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'pos' is null");
        PBErrCatch(ShapoidErr);
    }
    if (VecGetDim(pos) != that->_dim) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
            that->_dim, VecGetDim(pos));
        PBErrCatch(ShapoidErr);
    }
    if (that->_type != ShapoidTypeFacoid &&

```

```

        that->_type != ShapoidTypeSpheroid &&
        that->_type != ShapoidTypePyramidoid) {
        ShapoidErr->_type = PBErTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
        PBErCatch(ShapoidErr);
    }
#endif
    // If the Shapoid is a Facoid
    if (that->_type == ShapoidTypeFacoid) {
        return FacoidIsPosInside((Facoid*)that, pos);
    } // Else, if the Shapoid is a Pyramidoid
    } else if (that->_type == ShapoidTypePyramidoid) {
        return PyramidoidIsPosInside((Pyramidoid*)that, pos);
    } // Else, if the Shapoid is a Spheroid
    } else if (that->_type == ShapoidTypeSpheroid) {
        return SpheroidIsPosInside((Spheroid*)that, pos);
    } else
        return false;
}

#if BUILDMODE != 0
inline
#endif
bool FacoidIsPosInside(const Facoid* const that,
    const VecFloat* const pos) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErCatch(ShapoidErr);
    }
    if (pos == NULL) {
        ShapoidErr->_type = PBErTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'pos' is null");
        PBErCatch(ShapoidErr);
    }
    if (VecGetDim(pos) != ShapoidGetDim(that)) {
        ShapoidErr->_type = PBErTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
            ShapoidGetDim(that), VecGetDim(pos));
        PBErCatch(ShapoidErr);
    }
#endif
    // Get the coordinates of pos in the Shapoid coordinate system
    VecFloat* coord = ShapoidImportCoord(that, pos);
    // Declare a variable to memorize the result
    bool ret = false;
    // pos is in the Shapoid if all the coord in Shapoid coord
    // system are in [0.0, 1.0]
    ret = true;
    for (int dim = ShapoidGetDim(that); dim-- && ret == true;) {
        float v = VecGet(coord, dim);
        if (v < 0.0 || v > 1.0)
            ret = false;
    }
    // Free memory
    VecFree(&coord);
    // Return the result
    return ret;
}

#if BUILDMODE != 0

```

```

inline
#endif
bool PyramidoidIsPosInside(const Pyramidoid* const that,
    const VecFloat* const pos) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (pos == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'pos' is null");
        PBErrCatch(ShapoidErr);
    }
    if (VecGetDim(pos) != ShapoidGetDim(that)) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
            ShapoidGetDim(that), VecGetDim(pos));
        PBErrCatch(ShapoidErr);
    }
#endif
    // Get the coordinates of pos in the Shapoid coordinate system
    VecFloat* coord = ShapoidImportCoord(that, pos);
    // Declare a variable to memorize the result
    bool ret = false;
    // 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 = ShapoidGetDim(that); 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;
    // Free memory
    VecFree(&coord);
    // Return the result
    return ret;
}

#if BUILDMODE != 0
inline
#endif
bool SpheroidIsPosInside(const Spheroid* const that,
    const VecFloat* const pos) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErrCatch(ShapoidErr);
    }
    if (pos == NULL) {
        ShapoidErr->_type = PBErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'pos' is null");
        PBErrCatch(ShapoidErr);
    }
    if (VecGetDim(pos) != ShapoidGetDim(that)) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
    }
#endif
}

```

```

        sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
            ShapoidGetDim(that), VecGetDim(pos));
        PBErrCatch(ShapoidErr);
    }
#endif
    // Get the coordinates of pos in the Shapoid coordinate system
    VecFloat* coord = ShapoidImportCoord(that, pos);
    // Declare a variable to memorize the result
    bool ret = false;
    // 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
    VecFree(&coord);
    // 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
#if BUILDMODE != 0
inline
#endif
float _ShapoidGetPosDepth(const Shapoid* const that,
    const VecFloat* const pos) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (pos == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'pos' is null");
            PBErrCatch(ShapoidErr);
        }
        if (VecGetDim(pos) != that->_dim) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
                that->_dim, VecGetDim(pos));
            PBErrCatch(ShapoidErr);
        }
        if (that->_type != ShapoidTypeFacoid &&
            that->_type != ShapoidTypeSpheroid &&
            that->_type != ShapoidTypePyramidoid) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg, "No implementation for 'that' 's type");
            PBErrCatch(ShapoidErr);
        }
    #endif
    // If the Shapoid is a Facoid
    if (that->_type == ShapoidTypeFacoid) {
        return FacoidGetPosDepth((Facoid*)that, pos);
    }
    // Else, if the Shapoid is a Pyramidoid
    else if (that->_type == ShapoidTypePyramidoid) {
        return PyramidoidGetPosDepth((Pyramidoid*)that, pos);
    }
    // Else, if the Shapoid is a Spheroid

```

```

    } else if (that->_type == ShapoidTypeSpheroid) {
        return SpheroidGetPosDepth((Spheroid*)that, pos);
    } else {
        return 0.0;
    }
}

#if BUILDMODE != 0
inline
#endif
float FacoidGetPosDepth(const Facoid* const that,
    const VecFloat* const pos) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (pos == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'pos' is null");
            PBErrCatch(ShapoidErr);
        }
        if (VecGetDim(pos) != ShapoidGetDim(that)) {
            ShapoidErr->_type = PBErrTypeInvalidArg;
            sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
                ShapoidGetDim(that), VecGetDim(pos));
            PBErrCatch(ShapoidErr);
        }
    #endif
    // Get the coordinates of pos in the Shapoid coordinate system
    VecFloat* coord = ShapoidImportCoord(that, pos);
    // Declare a variable to memorize the result
    float ret = 1.0;
    for (int dim = ShapoidGetDim(that); 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;
    }
    // Free memory
    VecFree(&coord);
    // Return the result
    return ret;
}

#if BUILDMODE != 0
inline
#endif
float PyramidoidGetPosDepth(const Pyramidoid* const that,
    const VecFloat* const pos) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (pos == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'pos' is null");
            PBErrCatch(ShapoidErr);
        }
    #endif
}

```



```

    }
    if (VecGetDim(pos) != ShapoidGetDim(that)) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
            ShapoidGetDim(that), VecGetDim(pos));
        PBErrCatch(ShapoidErr);
    }
#endif
    // Get the coordinates of pos in the Shapoid coordinate system
    VecFloat* coord = ShapoidImportCoord(that, pos);
    // Declare a variable to memorize the result
    float ret = 1.0;
    float sum = 0.0;
    bool flag = true;
    for (int dim = ShapoidGetDim(that); 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));
        }
    }
    // Free memory
    VecFree(&coord);
    // Return the result
    return ret;
}

#if BUILDMODE != 0
inline
#endif
float SpheroidGetPosDepth(const Spheroid* const that,
    const VecFloat* const pos) {
    #if BUILDMODE == 0
        if (that == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'that' is null");
            PBErrCatch(ShapoidErr);
        }
        if (pos == NULL) {
            ShapoidErr->_type = PBErrTypeNullPointer;
            sprintf(ShapoidErr->_msg, "'pos' is null");
            PBErrCatch(ShapoidErr);
        }
    #endif
    if (VecGetDim(pos) != ShapoidGetDim(that)) {
        ShapoidErr->_type = PBErrTypeInvalidArg;
        sprintf(ShapoidErr->_msg, "'pos' 's dimension is invalid (%d==%d)",
            ShapoidGetDim(that), VecGetDim(pos));
        PBErrCatch(ShapoidErr);
    }
}

```

```

    }
#endif
    // Get the coordinates of pos in the Shapoid coordinate system
    VecFloat* coord = ShapoidImportCoord(that, pos);
    // Declare a variable to memorize the result
    float ret = 0.0;
    float norm = VecNorm(coord);
    if (norm <= 0.5)
        ret = 1.0 - norm * 2.0;
    // Free memory
    VecFree(&coord);
    // Return the result
    return ret;
}

#if BUILDMODE != 0
inline
#endif
bool FacoidLoad(Facoid** that, FILE* const stream) {
    bool ret = _ShapoidLoad((Shapoid**)that, stream);
    if (!ret || ShapoidGetType(*that) != ShapoidTypeFacoid) {
        ShapoidFree(that);
        return false;
    }
    return true;
}

#if BUILDMODE != 0
inline
#endif
bool PyramidoidLoad(Pyramidoid** that, FILE* const stream) {
    bool ret = _ShapoidLoad((Shapoid**)that, stream);
    if (!ret || ShapoidGetType(*that) != ShapoidTypePyramidoid) {
        ShapoidFree(that);
        return false;
    }
    return true;
}

#if BUILDMODE != 0
inline
#endif
bool SpheroidLoad(Spheroid** that, FILE* const stream) {
    bool ret = _ShapoidLoad((Shapoid**)that, stream);
    if (!ret || ShapoidGetType(*that) != ShapoidTypeSpheroid) {
        ShapoidFree(that);
        return false;
    }
    return true;
}

// ----- ShapoidIter

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

// Return the current position in Shapoid coordinates of the
// ShapoidIter 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* ShapoidIterGetInternalPos(const ShapoidIter* const that) {
#if BUILDMODE == 0

```

```

    if (that == NULL) {
        ShapoidErr->_type = PBErTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErCatch(ShapoidErr);
    }
#endif
    return VecClone(that->_pos);
}

// Return the current position in standard coordinates of the
// ShapoidIter 'that'
#if BUILDMODE != 0
inline
#endif
VecFloat* ShapoidIterGetExternalPos(const ShapoidIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErCatch(ShapoidErr);
    }
#endif
    return ShapoidExportCoord(that->_shap, that->_pos);
}

// Set the attached Shapoid of the ShapoidIter 'that' to 'shap'
// The iterator is reset to its initial position
#if BUILDMODE != 0
inline
#endif
void _ShapoidIterSetShapoid(ShapoidIter* const that,
    const Shapoid* const shap) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErCatch(ShapoidErr);
    }
#endif
    that->_shap = shap;
}

// Get the Shapoid of the ShapoidIter 'that'
#if BUILDMODE != 0
inline
#endif
const Shapoid* ShapoidIterShapoid(const ShapoidIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PBErTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PBErCatch(ShapoidErr);
    }
#endif
    return that->_shap;
}

// Set the delta of the ShapoidIter 'that' to a copy of 'delta'
#if BUILDMODE != 0
inline
#endif
void _ShapoidIterSetDelta(ShapoidIter* const that,

```

```

    const VecFloat* const delta) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PErrCatch(ShapoidErr);
    }
#endif
    VecFree(&(that->_delta));
    that->_delta = VecClone(delta);
}

// Get the delta of the ShapoidIter 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* ShapoidIterDelta(const ShapoidIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        ShapoidErr->_type = PErrTypeNullPointer;
        sprintf(ShapoidErr->_msg, "'that' is null");
        PErrCatch(ShapoidErr);
    }
#endif
    return that->_delta;
}

```

4 Makefile

```

#directory
PBERRDIR=../PErr
PBMATHDIR=../PBMath
GSETDIR=../GSet
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

include $(PBERRDIR)/Makefile.inc

INCPATH=-I./ -I$(PBERRDIR)/ -I$(PBMATHDIR)/ -I$(GSETDIR)/ -I$(PBJSONDIR)/ -I$(GTREEDIR)/
BUILDOPTIONS=$(BUILDPARAM) $(INCPATH)

# compiler
COMPILER=gcc

#rules
all : main

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

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

```

```

shapoid.o : shapoid.c shapoid.h shapoid-inline.c $(PBMATHDIR)/pbmath.h $(GSETDIR)/gset.h $(PBERRDIR)/pberr.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c shapoid.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 $(GSE
$(COMPILER) $(BUILDOPTIONS) -c $(GTREEDIR)/gtree.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 "pberr.h"
#include "shapoid.h"

#define RANDOMSEED 0

void UnitTestCreateCloneIsEqualFree() {
    int dim = 3;
    Shapoid* facoid = ShapoidCreate(dim, ShapoidTypeFacoid);
    if (facoid == NULL || facoid->_dim != dim ||
        facoid->_type != ShapoidTypeFacoid || facoid->_pos == NULL ||
        VecGetDim(facoid->_pos) != dim || facoid->_sysLinEqImport == NULL ||
        facoid->_axis == NULL) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidCreate failed");
        PBErrCatch(ShapoidErr);
    }
    for (int iDim = dim; iDim--;) {
        if (ISEQUALF(VecGet(facoid->_pos, iDim), 0.0) == false ||
            facoid->_axis[iDim] == NULL ||
            VecGetDim(facoid->_axis[iDim]) != dim) {
            ShapoidErr->_type = PBErrTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidCreate failed");
            PBErrCatch(ShapoidErr);
        }
    }
}

```

```

}
for (int jDim = dim; jDim--;) {
    if ((iDim == jDim &&
        ISEQUALF(VecGet(facoid->_axis[iDim], jDim), 1.0) == false) ||
        (iDim != jDim &&
        ISEQUALF(VecGet(facoid->_axis[iDim], jDim), 0.0) == false)) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidCreate failed");
        PBErrCatch(ShapoidErr);
    }
}
if (ISEQUALF(VecGet(facoid->_sysLinEqImport->_V, iDim),
0.0) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidCreate failed");
    PBErrCatch(ShapoidErr);
}
}
VecShort2D u = VecShortCreateStatic2D();
VecSet(&u, 0, dim); VecSet(&u, 1, dim);
VecShort2D v = VecShortCreateStatic2D();
do {
    if ((VecGet(&v, 0) == VecGet(&v, 1) &&
        ISEQUALF(MatGet(facoid->_sysLinEqImport->_M, &v), 1.0) == false) ||
        (VecGet(&v, 0) != VecGet(&v, 1) &&
        ISEQUALF(MatGet(facoid->_sysLinEqImport->_M, &v), 0.0) == false) ||
        (VecGet(&v, 0) == VecGet(&v, 1) &&
        ISEQUALF(MatGet(facoid->_sysLinEqImport->_Minv, &v),
            1.0) == false) ||
        (VecGet(&v, 0) != VecGet(&v, 1) &&
        ISEQUALF(MatGet(facoid->_sysLinEqImport->_Minv, &v),
            0.0) == false)) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidCreate failed");
        PBErrCatch(ShapoidErr);
    }
} while (VecStep(&v, &u));
Shapoid* clone = ShapoidClone(facoid);
if (facoid->_dim != clone->_dim ||
    facoid->_type != clone->_type ||
    VecIsEqual(facoid->_pos, clone->_pos) == false ||
    MatIsEqual(facoid->_sysLinEqImport->_M,
        clone->_sysLinEqImport->_M) == false ||
    MatIsEqual(facoid->_sysLinEqImport->_Minv,
        clone->_sysLinEqImport->_Minv) == false ||
    VecIsEqual(facoid->_sysLinEqImport->_V,
        clone->_sysLinEqImport->_V) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidClone failed");
    PBErrCatch(ShapoidErr);
}
for (int i = dim; i--;) {
    if (VecIsEqual(facoid->_axis[i], clone->_axis[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidClone failed");
        PBErrCatch(ShapoidErr);
    }
}
}
if (ShapoidIsEqual(facoid, clone) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidIsEqual failed");
    PBErrCatch(ShapoidErr);
}

```

```

    }
    clone->_type = ShapoidTypePyramidoid;
    if (ShapoidIsEqual(facoid, clone) == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsEqual failed");
        PBErrCatch(ShapoidErr);
    }
    clone->_type = facoid->_type;
    clone->_dim = dim + 1;
    if (ShapoidIsEqual(facoid, clone) == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsEqual failed");
        PBErrCatch(ShapoidErr);
    }
    clone->_dim = facoid->_dim;
    VecSet(clone->_pos, 0, 1.0);
    if (ShapoidIsEqual(facoid, clone) == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsEqual failed");
        PBErrCatch(ShapoidErr);
    }
    VecSet(clone->_pos, 0, 0.0);
    VecSet(clone->_axis[0], 0, 2.0);
    if (ShapoidIsEqual(facoid, clone) == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsEqual failed");
        PBErrCatch(ShapoidErr);
    }
    VecSet(clone->_axis[0], 0, 1.0);
    ShapoidFree(&facoid);
    if (facoid != NULL) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidFree failed");
        PBErrCatch(ShapoidErr);
    }
    ShapoidFree(&clone);
    printf("UnitTestCreateCloneIsEqualFree OK\n");
}

void UnitTestLoadSavePrint() {
    int dim = 3;
    Facoid* facoid = FacoidCreate(dim);
    FILE* file = fopen("./facoid.txt", "w");
    if (ShapoidSave(facoid, file, false) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidSave failed");
        PBErrCatch(ShapoidErr);
    }
    fclose(file);
    file = fopen("./facoid.txt", "r");
    Facoid* load = FacoidCreate(dim);
    if (ShapoidLoad(&load, file) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidLoad failed");
        PBErrCatch(ShapoidErr);
    }
    fclose(file);
    if (ShapoidIsEqual(facoid, load) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidLoad/Save failed");
        PBErrCatch(ShapoidErr);
    }
}

```

```

    ShapoidPrintln(facoid, stdout);
    ShapoidFree(&facoid);
    ShapoidFree(&load);
    printf("UnitTestLoadSavePrint OK\n");
}

void UnitTestGetSetTypeDimPosAxis() {
    int dim = 3;
    Facoid* facoid = FacoidCreate(dim);
    Pyramidoid* pyramidoid = PyramidoidCreate(dim);
    Spheroid* spheroid = SpheroidCreate(dim);
    if (ShapoidGetType(facoid) != ShapoidTypeFacoid ||
        ShapoidGetType(pyramidoid) != ShapoidTypePyramidoid ||
        ShapoidGetType(spheroid) != ShapoidTypeSpheroid) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetType failed");
        PBErrCatch(ShapoidErr);
    }
    if (ShapoidGetDim(facoid) != dim) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetDim failed");
        PBErrCatch(ShapoidErr);
    }
    VecFloat* v = VecFloatCreate(dim);
    VecFloat* u = ShapoidGetPos(facoid);
    if (VecIsEqual(v, u) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetPos failed");
        PBErrCatch(ShapoidErr);
    }
    VecFree(&u);
    for (int i = dim; i--;) {
        u = ShapoidGetAxis(facoid, i);
        for (int j = dim; j--;)
            if ((i == j && ISEQUALF(VecGet(u, j), 1.0) == false) ||
                (i != j && ISEQUALF(VecGet(u, j), 0.0) == false)) {
                ShapoidErr->_type = PBErrTypeUnitTestFailed;
                sprintf(ShapoidErr->_msg, "ShapoidGetAxis failed");
                PBErrCatch(ShapoidErr);
            }
        VecFree(&u);
    }
    for (int i = dim; i--;)
        VecSet(v, i, (float)i);
    ShapoidSetPos(facoid, v);
    if (VecIsEqual(v, ShapoidPos(facoid)) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidSetPos failed");
        PBErrCatch(ShapoidErr);
    }
    for (int i = dim; i--;) {
        VecSetNull(v);
        VecSet(v, i, 2.0);
        ShapoidSetAxis(facoid, i, v);
        if (VecIsEqual(v, ShapoidAxis(facoid, i)) == false) {
            ShapoidErr->_type = PBErrTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidSetAxis failed");
            PBErrCatch(ShapoidErr);
        }
    }
    for (int i = dim; i--;)
        VecSet(v, i, i);
}

```



```

ShapoidSetCenterPos(facoid, v);
VecFloat* center = ShapoidGetCenter(facoid);
if (VecIsEqual(v, center) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidSetCenterPos failed");
    PBErrCatch(ShapoidErr);
}
VecFree(&center);
VecSet(v, 0, 1.0); VecSet(v, 1, 2.0); VecSet(v, 2, 0.5);
ShapoidScale(spheroid, v);
if (!ISEQUALF(ShapoidGetBoundingRadius(spheroid), 1.0)) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidGetBoundingRadius failed");
    PBErrCatch(ShapoidErr);
}
VecFree(&v);
ShapoidPosSet(facoid, 1, -1.0);
if (!ISEQUALF(((Shapoid*)facoid)->_pos->_val[1], -1.0)) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidPosSet failed");
    PBErrCatch(ShapoidErr);
}
ShapoidPosSetAdd(facoid, 1, -1.0);
if (!ISEQUALF(((Shapoid*)facoid)->_pos->_val[1], -2.0)) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidPosSet failed");
    PBErrCatch(ShapoidErr);
}
ShapoidAxisSet(facoid, 2, 1, -1.0);
if (!ISEQUALF(((Shapoid*)facoid)->_axis[2]->_val[1], -1.0)) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidAxisSet failed");
    PBErrCatch(ShapoidErr);
}
ShapoidAxisSetAdd(facoid, 2, 1, -1.0);
if (!ISEQUALF(((Shapoid*)facoid)->_axis[2]->_val[1], -2.0)) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidAxisSet failed");
    PBErrCatch(ShapoidErr);
}

ShapoidFree(&facoid);
ShapoidFree(&pyramidoid);
ShapoidFree(&spheroid);
printf("UnitTestGetSetTypeDimPosAxis OK\n");
}

void UnitTestTranslateScaleGrow() {
    int dim = 2;
    Facoid* facoid = FacoidCreate(dim);
    VecFloat* v = VecFloatCreate(dim);
    for (int i = dim; i--;)
        VecSet(v, i, 1.0);
    ShapoidTranslate(facoid, v);
    if (VecIsEqual(v, ((Shapoid*)facoid)->_pos) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidTranslate failed");
        PBErrCatch(ShapoidErr);
    }
    float scale = 2.0;
    ShapoidScale(facoid, scale);
    VecSetNull(v);
}

```

```

VecSetNull(((Shapoid*)facoid)->_pos);
if (VecIsEqual(v, ((Shapoid*)facoid)->_pos) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "_ShapoidScaleScalar failed");
    PBErCatch(ShapoidErr);
}
for (int i = dim; i--;) {
    for (int j = dim; j--;)
        if (i == j)
            VecSet(v, j, scale);
        else
            VecSet(v, j, 0.0);
    if (VecIsEqual(v, ((Shapoid*)facoid)->_axis[i]) == false) {
        ShapoidErr->_type = PBErTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "_ShapoidScaleScalar failed");
        PBErCatch(ShapoidErr);
    }
}
for (int i = dim; i--;)
    VecSet(v, i, 1.0 + (float)i);
ShapoidScale(facoid, v);
VecSetNull(v);
if (VecIsEqual(v, ((Shapoid*)facoid)->_pos) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "_ShapoidScaleVector failed");
    PBErCatch(ShapoidErr);
}
for (int i = dim; i--;) {
    for (int j = dim; j--;)
        if (i == j)
            VecSet(v, j, scale * (1.0 + (float)i));
        else
            VecSet(v, j, 0.0);
    if (VecIsEqual(v, ((Shapoid*)facoid)->_axis[i]) == false) {
        ShapoidErr->_type = PBErTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "_ShapoidScaleVector failed");
        PBErCatch(ShapoidErr);
    }
}
ShapoidFree(&facoid);
facoid = FacoidCreate(dim);
scale = 2.0;
ShapoidGrow(facoid, scale);
for (int i = dim; i--;)
    VecSet(v, i, -0.5);
if (VecIsEqual(v, ((Shapoid*)facoid)->_pos) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "_ShapoidGrowScalar failed");
    PBErCatch(ShapoidErr);
}
for (int i = dim; i--;) {
    for (int j = dim; j--;)
        if (i == j)
            VecSet(v, j, scale);
        else
            VecSet(v, j, 0.0);
    if (VecIsEqual(v, ((Shapoid*)facoid)->_axis[i]) == false) {
        ShapoidErr->_type = PBErTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "_ShapoidGrowScalar failed");
        PBErCatch(ShapoidErr);
    }
}
}

```

```

Pyramidoid* pyramidoid = PyramidoidCreate(dim);
VecFloat* centerA = ShapoidGetCenter(pyramidoid);
ShapoidGrow(pyramidoid, scale);
VecFloat* centerB = ShapoidGetCenter(pyramidoid);
if (VecIsEqual(centerA, centerB) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "_ShapoidGrowScalar failed");
    PBErrCatch(ShapoidErr);
}
for (int i = dim; i--;) {
    for (int j = dim; j--;)
        if (i == j)
            VecSet(v, j, scale);
        else
            VecSet(v, j, 0.0);
    if (VecIsEqual(v, ((Shapoid*)pyramidoid)->_axis[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "_ShapoidGrowScalar failed");
        PBErrCatch(ShapoidErr);
    }
}
VecFree(&centerA);
VecFree(&centerB);
Spheroid* spheroid = SpheroidCreate(dim);
ShapoidGrow(spheroid, scale);
VecSetNull(v);
if (VecIsEqual(v, ((Shapoid*)spheroid)->_pos) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "_ShapoidGrowScalar failed");
    PBErrCatch(ShapoidErr);
}
for (int i = dim; i--;) {
    for (int j = dim; j--;)
        if (i == j)
            VecSet(v, j, scale);
        else
            VecSet(v, j, 0.0);
    if (VecIsEqual(v, ((Shapoid*)spheroid)->_axis[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "_ShapoidGrowScalar failed");
        PBErrCatch(ShapoidErr);
    }
}
VecFloat* scalev = VecFloatCreate(dim);
for (int i = dim; i--;)
    VecSet(scalev, i, 1.0 + (float)i);
ShapoidFree(&facoid);
ShapoidFree(&pyramidoid);
ShapoidFree(&spheroid);
facoid = FacoidCreate(dim);
ShapoidGrow(facoid, scalev);
float pa[2] = {0.000, -0.500};
for (int i = dim; i--;)
    VecSet(v, i, pa[i]);
if (VecIsEqual(v, ((Shapoid*)facoid)->_pos) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "_ShapoidGrowVector failed");
    PBErrCatch(ShapoidErr);
}
for (int i = dim; i--;) {
    for (int j = dim; j--;)
        if (i == j)

```

```

        VecSet(v, j, VecGet(scalev, i));
    else
        VecSet(v, j, 0.0);
    if (VecIsEqual(v, ((Shapoid*)facoid)->_axis[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "_ShapoidGrowVector failed");
        PBErrCatch(ShapoidErr);
    }
}
pyramidoid = PyramidoidCreate(dim);
centerA = ShapoidGetCenter(pyramidoid);
ShapoidGrow(pyramidoid, scalev);
centerB = ShapoidGetCenter(pyramidoid);
if (VecIsEqual(centerA, centerB) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "_ShapoidGrowVector failed");
    PBErrCatch(ShapoidErr);
}
for (int i = dim; i--;) {
    for (int j = dim; j--;)
        if (i == j)
            VecSet(v, j, VecGet(scalev, i));
        else
            VecSet(v, j, 0.0);
    if (VecIsEqual(v, ((Shapoid*)pyramidoid)->_axis[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "_ShapoidGrowVector failed");
        PBErrCatch(ShapoidErr);
    }
}
VecFree(&centerA);
VecFree(&centerB);
spheroid = SpheroidCreate(dim);
ShapoidGrow(spheroid, scalev);
VecSetNull(v);
if (VecIsEqual(v, ((Shapoid*)spheroid)->_pos) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "_ShapoidGrowVector failed");
    PBErrCatch(ShapoidErr);
}
for (int i = dim; i--;) {
    for (int j = dim; j--;)
        if (i == j)
            VecSet(v, j, VecGet(scalev, i));
        else
            VecSet(v, j, 0.0);
    if (VecIsEqual(v, ((Shapoid*)spheroid)->_axis[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "_ShapoidGrowVector failed");
        PBErrCatch(ShapoidErr);
    }
}
VecFree(&scalev);
VecFree(&v);
ShapoidFree(&facoid);
ShapoidFree(&pyramidoid);
ShapoidFree(&spheroid);
printf("UnitTestTranslateScaleGrow OK\n");
}

void UnitTestRotate() {
    int dim = 2;

```

```

Facoid* facoid = FacoidCreate(dim);
Pyramidoid* pyramidoid = PyramidoidCreate(dim);
Spheroid* spheroid = SpheroidCreate(dim);
float theta = PBMath_HALFPI;
ShapoidRotCenter(facoid, theta);
float pb[2] = {1.0, 0.0};
float pc[2] = {0.0, 1.0};
float pd[2] = {-1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pb[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
            pc[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
            pd[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotCenter(pyramidoid, theta);
float pe[2] = {0.6666667, 0.0};
float pf[2] = {0.0, 1.0};
float pg[2] = {-1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pe[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
            pf[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
            pg[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotCenter(spheroid, theta);
float ph[2] = {0.0, 0.0};
float pi[2] = {0.0, 1.0};
float pj[2] = {-1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        ph[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
            pi[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
            pj[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotOrigin(facoid, theta);
float pk[2] = {0.0, 1.0};
float pl[2] = {-1.0, 0.0};
float pm[2] = {0.0, -1.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pk[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
            pl[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
            pm[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
    }
}

```

```

        sprintf(ShapoidErr->_msg, "ShapoidRotOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotOrigin(pyramidoid, theta);
float pn[2] = {0.0, 0.6666667};
float po[2] = {-1.0, 0.0};
float pp[2] = {0.0, -1.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pn[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
        po[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
        pp[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotOrigin(spheroid, theta);
float pq[2] = {0.0, 0.0};
float pr[2] = {-1.0, 0.0};
float ps[2] = {0.0, -1.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        pq[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
        pr[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
        ps[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotStart(facoid, theta);
float pt[2] = {0.0, 1.0};
float pu[2] = {0.0, -1.0};
float pv[2] = {1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pt[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
        pu[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
        pv[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotStart failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotStart(pyramidoid, theta);
float pw[2] = {0.0, 0.6666667};
float px[2] = {0.0, -1.0};
float py[2] = {1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pw[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
        px[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
        py[i]) == false) {

```

```

        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotStart failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotStart(spheroid, theta);
float pz[2] = {0.0, 0.0};
float paa[2] = {0.0, -1.0};
float pab[2] = {1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        pz[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
        paa[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
        pab[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotStart failed");
        PBErrCatch(ShapoidErr);
    }
}
}
ShapoidFree(&facoid);
ShapoidFree(&pyramidoid);
ShapoidFree(&spheroid);
printf("UnitTestRotate OK\n");
}

void UnitTestRotateAxis() {
    int dim = 3;
    Facoid* facoid = FacoidCreate(dim);
    Pyramidoid* pyramidoid = PyramidoidCreate(dim);
    Spheroid* spheroid = SpheroidCreate(dim);
    float theta = PBMath_HALFPI;
    VecFloat3D axis = VecFloatCreateStatic3D();
    VecSet(&axis, 0, 1.0); VecSet(&axis, 1, 1.0); VecSet(&axis, 2, 1.0);
    VecNormalise(&axis);
    ShapoidRotAxisCenter(facoid, &axis, theta);
    float pb[3] = {0.0, 0.0, 0.0};
    float pc[3] = {0.333333, 0.910684, -0.244017};
    float pd[3] = {-0.244017, 0.333333, 0.910684};
    for (int i = dim; i--;) {
        if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pb[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
            pc[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
            pd[i]) == false) {
            ShapoidErr->_type = PBErrTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidRotAxisCenter failed");
            PBErrCatch(ShapoidErr);
        }
    }
}
ShapoidRotAxisCenter(pyramidoid, &axis, theta);
float pe[3] = {0.000000, 0.000000, 0.000000};
float pf[3] = {0.333333, 0.910684, -0.244017};
float pg[3] = {-0.244017, 0.333333, 0.910684};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pe[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
        pf[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
        pg[i]) == false) {

```

```

        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotAxisCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotAxisCenter(spheroid, &axis, theta);
float ph[3] = {0.0, 0.0, 0.0};
float pi[3] = {0.333333, 0.910684, -0.244017};
float pj[3] = {-0.244017, 0.333333, 0.910684};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        ph[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
        pi[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
        pj[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotAxisCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
}
ShapoidRotAxisOrigin(facoid, &axis, theta);
float pk[3] = {0.0, 0.0, 0.0};
float pl[3] = {-0.333333, 0.666667, 0.666667};
float pm[3] = {0.666667, -0.333333, 0.666667};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pk[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
        pl[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
        pm[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotAxisOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
}
ShapoidRotAxisOrigin(pyramidoid, &axis, theta);
float pn[3] = {0.0, 0.0, 0.0};
float po[3] = {-0.333333, 0.666667, 0.666667};
float pp[3] = {0.666667, -0.333333, 0.666667};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pn[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
        po[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
        pp[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotAxisOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
}
ShapoidRotAxisOrigin(spheroid, &axis, theta);
float pq[3] = {0.0, 0.0, 0.0};
float pr[3] = {-0.333333, 0.666667, 0.666667};
float ps[3] = {0.666667, -0.333333, 0.666667};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        pq[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
        pr[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),

```



```

        ps[i]) == false) {
            ShapoidErr->_type = PBErrTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidRotAxisOrigin failed");
            PBErrCatch(ShapoidErr);
        }
    }
    ShapoidRotAxisStart(facoid, &axis, theta);
    float pt[3] = {0.0, 0.0, 0.0};
    float pu[3] = {0.333333, -0.244017, 0.910683};
    float pv[3] = {0.910683, 0.333333, -0.244017};
    for (int i = dim; i--;) {
        if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pt[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
                pu[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
                pv[i]) == false) {
            ShapoidErr->_type = PBErrTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidRotAxisStart failed");
            PBErrCatch(ShapoidErr);
        }
    }
    ShapoidRotAxisStart(pyramidoid, &axis, theta);
    float pw[3] = {0.0, 0.0, 0.0};
    float px[3] = {0.333333, -0.244017, 0.910683};
    float py[3] = {0.910683, 0.333333, -0.244017};
    for (int i = dim; i--;) {
        if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
            pw[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
                px[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
                py[i]) == false) {
            ShapoidErr->_type = PBErrTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidRotAxisStart failed");
            PBErrCatch(ShapoidErr);
        }
    }
    ShapoidRotAxisStart(spheroid, &axis, theta);
    float pz[3] = {0.0, 0.0, 0.0};
    float paa[3] = {0.333333, -0.244017, 0.910683};
    float pab[3] = {0.910683, 0.333333, -0.244017};
    for (int i = dim; i--;) {
        if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
            pz[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
                paa[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
                pab[i]) == false) {
            ShapoidErr->_type = PBErrTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidRotAxisStart failed");
            PBErrCatch(ShapoidErr);
        }
    }
    ShapoidFree(&facoid);
    ShapoidFree(&pyramidoid);
    ShapoidFree(&spheroid);
    printf("UnitTestRotateAxis OK\n");
}

void UnitTestRotateX() {
    int dim = 3;
    Facoid* facoid = FacoidCreate(dim);

```

```

Pyramidoid* pyramidoid = PyramidoidCreate(dim);
Spheroid* spheroid = SpheroidCreate(dim);
float theta = PBMATH_HALFPI;
ShapoidRotXCenter(facoid, theta);
float pb[3] = {0.0, 1.0, 0.0};
float pc[3] = {1.0, 0.0, 0.0};
float pd[3] = {0.0, 0.0, 1.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pb[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
            pc[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
            pd[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotXCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotXCenter(pyramidoid, theta);
float pe[3] = {0.0, 0.5, 0.0};
float pf[3] = {1.0, 0.0, 0.0};
float pg[3] = {0.0, 0.0, 1.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pe[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
            pf[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
            pg[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotXCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotXCenter(spheroid, theta);
float ph[3] = {0.0, 0.0, 0.0};
float pi[3] = {1.0, 0.0, 0.0};
float pj[3] = {0.0, 0.0, 1.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        ph[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
            pi[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
            pj[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotXCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotXOrigin(facoid, theta);
float pk[3] = {0.0, 0.0, 1.0};
float pl[3] = {1.0, 0.0, 0.0};
float pm[3] = {0.0, -1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pk[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
            pl[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
            pm[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotXOrigin failed");
    }
}

```

```

        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotXOrigin(pyramidoid, theta);
float pn[3] = {0.0, 0.0, 0.5};
float po[3] = {1.0, 0.0, 0.0};
float pp[3] = {0.0, -1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pn[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
        po[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
        pp[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotXOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotXOrigin(spheroid, theta);
float pq[3] = {0.0, 0.0, 0.0};
float pr[3] = {1.0, 0.0, 0.0};
float ps[3] = {0.0, -1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        pq[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
        pr[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
        ps[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotXOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotXStart(facoid, theta);
float pt[3] = {0.0, 0.0, 1.0};
float pu[3] = {1.0, 0.0, 0.0};
float pv[3] = {0.0, 0.0, -1.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pt[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
        pu[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
        pv[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotXStart failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotXStart(pyramidoid, theta);
float pw[3] = {0.0, 0.0, 0.5};
float px[3] = {1.0, 0.0, 0.0};
float py[3] = {0.0, 0.0, -1.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pw[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
        px[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
        py[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
    }
}

```

```

        sprintf(ShapoidErr->_msg, "ShapoidRotXStart failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotXStart(spheroid, theta);
float pz[3] = {0.0, 0.0, 0.0};
float paa[3] = {1.0, 0.0, 0.0};
float pab[3] = {0.0, 0.0, -1.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        pz[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
        paa[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
        pab[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotXStart failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidFree(&facoid);
ShapoidFree(&pyramidoid);
ShapoidFree(&spheroid);
printf("UnitTestRotateX OK\n");
}

void UnitTestRotateY() {
    int dim = 3;
    Facoid* facoid = FacoidCreate(dim);
    Pyramidoid* pyramidoid = PyramidoidCreate(dim);
    Spheroid* spheroid = SpheroidCreate(dim);
    float theta = PBMath_HALFPI;
    ShapoidRotYCenter(facoid, theta);
    float pb[3] = {0.0, 0.0, 1.0};
    float pc[3] = {0.0, 0.0, -1.0};
    float pd[3] = {0.0, 1.0, 0.0};
    for (int i = dim; i--;) {
        if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pb[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
            pc[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
            pd[i]) == false) {
            ShapoidErr->_type = PBErrTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidRotYCenter failed");
            PBErrCatch(ShapoidErr);
        }
    }
    ShapoidRotYCenter(pyramidoid, theta);
    float pe[3] = {0.0, 0.0, 0.5};
    float pf[3] = {0.0, 0.0, -1.0};
    float pg[3] = {0.0, 1.0, 0.0};
    for (int i = dim; i--;) {
        if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
            pe[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
            pf[i]) == false ||
            ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
            pg[i]) == false) {
            ShapoidErr->_type = PBErrTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidRotYCenter failed");
            PBErrCatch(ShapoidErr);
        }
    }
}

```

```

}
ShapoidRotYCenter(spheroid, theta);
float ph[3] = {0.0, 0.0, 0.0};
float pi[3] = {0.0, 0.0, -1.0};
float pj[3] = {0.0, 1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        ph[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
        pi[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
        pj[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotYCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotYOrigin(facoid, theta);
float pk[3] = {1.0, 0.0, 0.0};
float pl[3] = {-1.0, 0.0, 0.0};
float pm[3] = {0.0, 1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pk[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
        pl[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
        pm[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotYOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotYOrigin(pyramidoid, theta);
float pn[3] = {0.5, 0.0, 0.0};
float po[3] = {-1.0, 0.0, 0.0};
float pp[3] = {0.0, 1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pn[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
        po[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
        pp[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotYOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotYOrigin(spheroid, theta);
float pq[3] = {0.0, 0.0, 0.0};
float pr[3] = {-1.0, 0.0, 0.0};
float ps[3] = {0.0, 1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        pq[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
        pr[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
        ps[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotYOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}

```

```

    }
}
ShapoidRotYStart(facoid, theta);
float pt[3] = {1.0, 0.0, 0.0};
float pu[3] = {0.0, 0.0, 1.0};
float pv[3] = {0.0, 1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pt[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
            pu[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
            pv[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotYStart failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotYStart(pyramidoid, theta);
float pw[3] = {0.5, 0.0, 0.0};
float px[3] = {0.0, 0.0, 1.0};
float py[3] = {0.0, 1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pw[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
            px[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
            py[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotYStart failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotYStart(spheroid, theta);
float pz[3] = {0.0, 0.0, 0.0};
float paa[3] = {0.0, 0.0, 1.0};
float pab[3] = {0.0, 1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        pz[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
            paa[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
            pab[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotYStart failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidFree(&facoid);
ShapoidFree(&pyramidoid);
ShapoidFree(&spheroid);
printf("UnitTestRotateY OK\n");
}

void UnitTestRotateZ() {
    int dim = 3;
    Facoid* facoid = FacoidCreate(dim);
    Pyramidoid* pyramidoid = PyramidoidCreate(dim);
    Spheroid* spheroid = SpheroidCreate(dim);
    float theta = PBMATH_HALFPI;
    ShapoidRotZCenter(facoid, theta);
}

```

```

float pb[3] = {1.0, 0.0, 0.0};
float pc[3] = {0.0, 1.0, 0.0};
float pd[3] = {-1.0, 0.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pb[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
            pc[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
            pd[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotZCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotZCenter(pyramidoid, theta);
float pe[3] = {0.5, 0.0, 0.0};
float pf[3] = {0.0, 1.0, 0.0};
float pg[3] = {-1.0, 0.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pe[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
            pf[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
            pg[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotZCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotZCenter(spheroid, theta);
float ph[3] = {0.0, 0.0, 0.0};
float pi[3] = {0.0, 1.0, 0.0};
float pj[3] = {-1.0, 0.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        ph[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
            pi[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
            pj[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotZCenter failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotZOrigin(facoid, theta);
float pk[3] = {0.0, 1.0, 0.0};
float pl[3] = {-1.0, 0.0, 0.0};
float pm[3] = {0.0, -1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pk[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
            pl[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
            pm[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotZOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotZOrigin(pyramidoid, theta);

```

```

float pn[3] = {0.0, 0.5, 0.0};
float po[3] = {-1.0, 0.0, 0.0};
float pp[3] = {0.0, -1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pn[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
        po[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
        pp[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotZOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotZOrigin(spheroid, theta);
float pq[3] = {0.0, 0.0, 0.0};
float pr[3] = {-1.0, 0.0, 0.0};
float ps[3] = {0.0, -1.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        pq[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
        pr[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
        ps[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotZOrigin failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotZStart(facoid, theta);
float pt[3] = {0.0, 1.0, 0.0};
float pu[3] = {0.0, -1.0, 0.0};
float pv[3] = {1.0, 0.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)facoid)->_pos, i), pt[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[0], i),
        pu[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)facoid)->_axis[1], i),
        pv[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotZStart failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidRotZStart(pyramidoid, theta);
float pw[3] = {0.0, 0.5, 0.0};
float px[3] = {0.0, -1.0, 0.0};
float py[3] = {1.0, 0.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_pos, i),
        pw[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[0], i),
        px[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)pyramidoid)->_axis[1], i),
        py[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotZStart failed");
        PBErrCatch(ShapoidErr);
    }
}
}

```



```

ShapoidRotZStart(spheroid, theta);
float pz[3] = {0.0, 0.0, 0.0};
float paa[3] = {0.0, -1.0, 0.0};
float pab[3] = {1.0, 0.0, 0.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)spheroid)->_pos, i),
        pz[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[0], i),
            paa[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)spheroid)->_axis[1], i),
            pab[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidRotZStart failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidFree(&facoid);
ShapoidFree(&pyramidoid);
ShapoidFree(&spheroid);
printf("UnitTestRotateZ OK\n");
}

void UnitTestImportExportCoordIsPosInside() {
    int dim = 2;
    Facoid* facoid = FacoidCreate(dim);
    Pyramidoid* pyramidoid = PyramidoidCreate(dim);
    Spheroid* spheroid = SpheroidCreate(dim);
    VecFloat* v = VecFloatCreate(dim);
    for (int i = dim; i--;)
        VecSet(v, i, 1.0 + (float)i);
    ShapoidTranslate(facoid, v);
    ShapoidTranslate(pyramidoid, v);
    ShapoidTranslate(spheroid, v);
    float scale = -2.0;
    ShapoidScale(facoid, scale);
    ShapoidScale(pyramidoid, scale);
    ShapoidScale(spheroid, scale);
    int nbTest = 100;
    srandom(RANDOMSEED);
    for (int iTest = nbTest; iTest--;) {
        VecFloat* posReal = VecFloatCreate(dim);
        for (int i = dim; i--;)
            VecSet(posReal, i, (rnd() - 0.5) * 10.0);
        VecFloat* posShapoidA = ShapoidImportCoord(facoid, posReal);
        bool isInside = ShapoidIsPosInside(facoid, posReal);
        if (VecGet(posShapoidA, 0) >= 0.0 &&
            VecGet(posShapoidA, 1) <= 1.0 &&
            VecGet(posShapoidA, 1) >= 0.0 &&
            VecGet(posShapoidA, 1) <= 1.0) {
            if (isInside == false) {
                ShapoidErr->_type = PBErrTypeUnitTestFailed;
                sprintf(ShapoidErr->_msg, "ShapoidIsPosInside failed");
                PBErrCatch(ShapoidErr);
            }
        } else {
            if (isInside == true) {
                ShapoidErr->_type = PBErrTypeUnitTestFailed;
                sprintf(ShapoidErr->_msg, "ShapoidIsPosInside failed");
                PBErrCatch(ShapoidErr);
            }
        }
    }
    VecOp(posShapoidA, scale, v, 1.0);
}

```

```

if (VecIsEqual(posReal, posShapoidA) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidImportCoord failed");
    PBErrCatch(ShapoidErr);
}
VecFree(&posShapoidA);
VecFloat* posShapoidB = ShapoidImportCoord(pyramidoid, posReal);
isInside = ShapoidIsPosInside(pyramidoid, posReal);
if (VecGet(posShapoidB, 0) >= 0.0 &&
    VecGet(posShapoidB, 0) <= 1.0 &&
    VecGet(posShapoidB, 1) >= 0.0 &&
    VecGet(posShapoidB, 1) <= 1.0 &&
    VecGet(posShapoidB, 0) + VecGet(posShapoidB, 1) <= 1.0) {
    if (isInside == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsPosInside failed");
        PBErrCatch(ShapoidErr);
    }
} else {
    if (isInside == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsPosInside failed");
        PBErrCatch(ShapoidErr);
    }
}
VecOp(posShapoidB, scale, v, 1.0);
if (VecIsEqual(posReal, posShapoidB) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidImportCoord failed");
    PBErrCatch(ShapoidErr);
}
VecFree(&posShapoidB);
VecFloat* posShapoidC = ShapoidImportCoord(spheroid, posReal);
isInside = ShapoidIsPosInside(spheroid, posReal);
if (VecGet(posShapoidC, 0) >= -0.5 &&
    VecGet(posShapoidC, 0) <= 0.5 &&
    VecGet(posShapoidC, 1) >= -0.5 &&
    VecGet(posShapoidC, 1) <= 0.5 &&
    pow(VecGet(posShapoidC, 0), 2.0) +
    pow(VecGet(posShapoidC, 1), 2.0) <= 0.25) {
    if (isInside == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsPosInside failed");
        PBErrCatch(ShapoidErr);
    }
} else {
    if (isInside == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsPosInside failed");
        PBErrCatch(ShapoidErr);
    }
}
VecOp(posShapoidC, scale, v, 1.0);
if (VecIsEqual(posReal, posShapoidC) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidImportCoord failed");
    PBErrCatch(ShapoidErr);
}
VecFree(&posShapoidC);
VecFree(&posReal);
}
for (int iTest = nbTest; iTest--;) {

```

```

VecFloat* posShapoid = VecFloatCreate(dim);
for (int i = dim; i--;)
    VecSet(posShapoid, i, (rnd() - 0.5) * 10.0);
VecFloat* posRealA = ShapoidExportCoord(facoid, posShapoid);
VecOp(posRealA, 1.0, v, -1.0);
VecScale(posRealA, 1.0 / scale);
if (VecIsEqual(posRealA, posShapoid) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidExportCoord failed");
    PBErrCatch(ShapoidErr);
}
VecFree(&posRealA);
VecFloat* posRealB = ShapoidExportCoord(pyramidoid, posShapoid);
VecOp(posRealB, 1.0, v, -1.0);
VecScale(posRealB, 1.0 / scale);
if (VecIsEqual(posRealB, posShapoid) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidExportCoord failed");
    PBErrCatch(ShapoidErr);
}
VecFree(&posRealB);
VecFloat* posRealC = ShapoidExportCoord(facoid, posShapoid);
VecOp(posRealC, 1.0, v, -1.0);
VecScale(posRealC, 1.0 / scale);
if (VecIsEqual(posRealC, posShapoid) == false) {
    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidExportCoord failed");
    PBErrCatch(ShapoidErr);
}
VecFree(&posRealC);
VecFree(&posShapoid);
}
VecFree(&v);
ShapoidFree(&facoid);
ShapoidFree(&pyramidoid);
ShapoidFree(&spheroid);
printf("UnitTestImportExportCoordIsPosInside OK\n");
}

void UnitTestGetBoundingBox() {
    int dim = 2;
    Facoid* facoid = FacoidCreate(dim);
    Pyramidoid* pyramidoid = PyramidoidCreate(dim);
    Spheroid* spheroid = SpheroidCreate(dim);
    VecFloat* v = VecFloatCreate(dim);
    for (int i = dim; i--;)
        VecSet(v, i, 1.0 + (float)i);
    ShapoidTranslate(facoid, v);
    ShapoidTranslate(pyramidoid, v);
    ShapoidTranslate(spheroid, v);
    float scale = -2.0;
    ShapoidScale(facoid, scale);
    ShapoidScale(pyramidoid, scale);
    ShapoidScale(spheroid, scale);
    float theta = PBMath_QUARTERPI;
    ShapoidRotCenter(facoid, theta);
    ShapoidRotCenter(pyramidoid, theta);
    ShapoidRotCenter(spheroid, theta);
    Facoid* boundA = ShapoidGetBoundingBox(facoid);
    float pa[2] = {-1.414214, -0.414213};
    float pb[2] = {2.828427, 0.0};
    float pc[2] = {0.0, 2.828427};

```

```

for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)boundA)->_pos, i),
        pa[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundA)->_axis[0], i),
        pb[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundA)->_axis[1], i),
        pc[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetBoundingBox failed");
        PBErrCatch(ShapoidErr);
    }
}
}
Facoid* boundB = ShapoidGetBoundingBox(pyramidoid);
float pd[2] = {-1.08088, 0.86193};
float pe[2] = {2.82843, 0.0};
float pf[2] = {0.0, 1.41421};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)boundB)->_pos, i), pd[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundB)->_axis[0], i),
        pe[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundB)->_axis[1], i),
        pf[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetBoundingBox failed");
        PBErrCatch(ShapoidErr);
    }
}
}
Facoid* boundC = ShapoidGetBoundingBox(spheroid);
float pg[2] = {-0.414214, 0.585786};
float ph[2] = {2.828427, 0.0};
float pi[2] = {0.0, 2.828427};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)boundC)->_pos, i), pg[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundC)->_axis[0], i),
        ph[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundC)->_axis[1], i),
        pi[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetBoundingBox failed");
        PBErrCatch(ShapoidErr);
    }
}
}
GSetShapoid set = GSetShapoidCreateStatic();
GSetPush(&set, facoid);
GSetPush(&set, pyramidoid);
GSetPush(&set, spheroid);
Facoid* boundD = ShapoidGetBoundingBox(&set);
float pj[2] = {-1.41421, -0.41421};
float pk[2] = {3.828427, 0.0};
float pl[2] = {0.0, 3.828427};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)boundD)->_pos, i), pj[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundD)->_axis[0], i),
        pk[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundD)->_axis[1], i),
        pl[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetBoundingBox failed");
        PBErrCatch(ShapoidErr);
    }
}
}
GSetFlush(&set);

```

```

VecSet(v, 0, 2.0); VecSet(v, 1, 4.0);
ShapoidSetPos(facoid, v);
ShapoidSetPos(pyramidoid, v);
VecSet(v, 0, 7.0); VecSet(v, 1, 0.0);
ShapoidSetAxis(facoid, 0, v);
ShapoidSetAxis(pyramidoid, 0, v);
VecSet(v, 0, 0.0); VecSet(v, 1, 4.0);
ShapoidSetAxis(facoid, 1, v);
ShapoidSetAxis(pyramidoid, 1, v);
Facoid* boundE = ShapoidGetBoundingBox(facoid);
float pm[2] = {2.0, 4.0};
float pn[2] = {7.0, 0.0};
float po[2] = {0.0, 4.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)boundE)->_pos, i), pm[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundE)->_axis[0], i),
            pn[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundE)->_axis[1], i),
            po[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetBoundingBox failed");
        PBErrCatch(ShapoidErr);
    }
}
Facoid* boundF = ShapoidGetBoundingBox(pyramidoid);
float pp[2] = {2.0, 4.0};
float pq[2] = {7.0, 0.0};
float pr[2] = {0.0, 4.0};
for (int i = dim; i--;) {
    if (ISEQUALF(VecGet(((Shapoid*)boundF)->_pos, i), pp[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundF)->_axis[0], i),
            pq[i]) == false ||
        ISEQUALF(VecGet(((Shapoid*)boundF)->_axis[1], i),
            pr[i]) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetBoundingBox failed");
        PBErrCatch(ShapoidErr);
    }
}
ShapoidFree(&boundA);
ShapoidFree(&boundB);
ShapoidFree(&boundC);
ShapoidFree(&boundD);
ShapoidFree(&boundE);
ShapoidFree(&boundF);
VecFree(&v);
ShapoidFree(&facoid);
ShapoidFree(&pyramidoid);
ShapoidFree(&spheroid);
printf("UnitTestGetBoundingBox OK\n");
}

void UnitTestGetPosDepthCenterCoverage() {
    int dim = 2;
    Facoid* facoid = FacoidCreate(dim);
    Pyramidoid* pyramidoid = PyramidoidCreate(dim);
    Spheroid* spheroid = SpheroidCreate(dim);
    VecFloat* center = ShapoidGetCenter(facoid);
    if (ISEQUALF(VecGet(center, 0), 0.5) == false ||
        ISEQUALF(VecGet(center, 1), 0.5) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetCenter failed");
    }
}

```

```

    PBErCatch(ShapoidErr);
}
VecFree(&center);
center = ShapoidGetCenter(pyramidoid);
if (ISEQUALF(VecGet(center, 0), 0.333333) == false ||
    ISEQUALF(VecGet(center, 1), 0.333333) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidGetCenter failed");
    PBErCatch(ShapoidErr);
}
VecFree(&center);
center = ShapoidGetCenter(spheroid);
if (ISEQUALF(VecGet(center, 0), 0.0) == false ||
    ISEQUALF(VecGet(center, 1), 0.0) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidGetCenter failed");
    PBErCatch(ShapoidErr);
}
VecFree(&center);
float coverage = ShapoidGetCoverageDelta(facoid, pyramidoid, 0.001);
if (ISEQUALF(coverage, 1.0) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidGetCoverage failed");
    PBErCatch(ShapoidErr);
}
coverage = ShapoidGetCoverageDelta(pyramidoid, facoid, 0.001);
if (ISEQUALF(coverage, 0.500499) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidGetCoverage failed");
    PBErCatch(ShapoidErr);
}
coverage = ShapoidGetCoverageDelta(pyramidoid, spheroid, 0.001);
if (ISEQUALF(coverage, 0.24937) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidGetCoverage failed");
    PBErCatch(ShapoidErr);
}
coverage = ShapoidGetCoverageDelta(spheroid, pyramidoid, 0.001);
if (ISEQUALF(coverage, 0.39251) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidGetCoverage failed");
    PBErCatch(ShapoidErr);
}
coverage = ShapoidGetCoverageDelta(facoid, spheroid, 0.001);
if (ISEQUALF(coverage, 0.24937) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidGetCoverage failed");
    PBErCatch(ShapoidErr);
}
coverage = ShapoidGetCoverageDelta(spheroid, facoid, 0.001);
if (ISEQUALF(coverage, 0.196451) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidGetCoverage failed");
    PBErCatch(ShapoidErr);
}
VecFloat2D pos = VecFloatCreateStatic2D();
VecSet(&pos, 0, 0.333333); VecSet(&pos, 1, 0.333333);
float depth = ShapoidGetPosDepth(facoid, (VecFloat*)&pos);
if (ISEQUALF(depth, 0.790123) == false) {
    ShapoidErr->_type = PBErTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidGetPosDepth failed");
    PBErCatch(ShapoidErr);
}

```

```

    }
    depth = ShapoidGetPosDepth(pyramidoid, (VecFloat*)&pos);
    if (ISEQUALF(depth, 1.0) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetPosDepth failed");
        PBErrCatch(ShapoidErr);
    }
    depth = ShapoidGetPosDepth(spheroid, (VecFloat*)&pos);
    if (ISEQUALF(depth, 0.057192) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidGetPosDepth failed");
        PBErrCatch(ShapoidErr);
    }
    ShapoidFree(&facoid);
    ShapoidFree(&pyramidoid);
    ShapoidFree(&spheroid);
    printf("UnitTestGetPosDepthCenterCoverage OK\n");
}

void UnitTestFacoidAlignedIsInsideFacoidAligned() {
    Facoid* facA = FacoidCreate(2);
    Facoid* facB = FacoidCreate(2);
    VecFloat2D p = VecFloatCreateStatic2D();
    VecFloat2D u = VecFloatCreateStatic2D();
    VecFloat2D v = VecFloatCreateStatic2D();
    VecSet(&p, 0, 0.0); VecSet(&p, 1, 0.0);
    VecSet(&u, 0, 1.0); VecSet(&v, 1, 1.0);
    ShapoidSetPos(facA, &p);
    ShapoidSetAxis(facA, 0, &u);
    ShapoidSetAxis(facA, 1, &v);
    VecSet(&p, 0, 2.0); VecSet(&p, 1, 2.0);
    VecSet(&u, 0, 0.5); VecSet(&v, 1, 0.5);
    ShapoidSetPos(facB, &p);
    ShapoidSetAxis(facB, 0, &u);
    ShapoidSetAxis(facB, 1, &v);
    if (FacoidAlignedIsInsideFacoidAligned(facA, facB) == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg,
            "FacoidAlignedIsInsideFacoidAligned failed");
        PBErrCatch(ShapoidErr);
    }
    VecSet(&p, 0, 0.1); VecSet(&p, 1, 0.1);
    ShapoidSetPos(facB, &p);
    if (FacoidAlignedIsInsideFacoidAligned(facB, facA) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg,
            "FacoidAlignedIsInsideFacoidAligned failed");
        PBErrCatch(ShapoidErr);
    }
    VecSet(&u, 0, 1.0); VecSet(&v, 1, 1.0);
    ShapoidSetAxis(facB, 0, &u);
    ShapoidSetAxis(facB, 1, &v);
    if (FacoidAlignedIsInsideFacoidAligned(facB, facA) == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg,
            "FacoidAlignedIsInsideFacoidAligned failed");
        PBErrCatch(ShapoidErr);
    }
    ShapoidFree(&facA);
    ShapoidFree(&facB);
    printf("UnitTestFacoidAlignedIsInsideFacoidAligned OK\n");
}

```

```

void UnitTestFacoidAlignedIsOutsideFacoidAligned() {
    Facoid* facA = FacoidCreate(2);
    Facoid* facB = FacoidCreate(2);
    VecFloat2D p = VecFloatCreateStatic2D();
    VecFloat2D u = VecFloatCreateStatic2D();
    VecFloat2D v = VecFloatCreateStatic2D();
    VecSet(&p, 0, 0.0); VecSet(&p, 1, 0.0);
    VecSet(&u, 0, 1.0); VecSet(&v, 1, 1.0);
    ShapoidSetPos(facA, &p);
    ShapoidSetAxis(facA, 0, &u);
    ShapoidSetAxis(facA, 1, &v);
    VecSet(&p, 0, 2.0); VecSet(&p, 1, 2.0);
    VecSet(&u, 0, 0.5); VecSet(&v, 1, 0.5);
    ShapoidSetPos(facB, &p);
    ShapoidSetAxis(facB, 0, &u);
    ShapoidSetAxis(facB, 1, &v);
    if (FacoidAlignedIsOutsideFacoidAligned(facA, facB) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg,
            "FacoidAlignedIsOutsideFacoidAligned failed");
        PBErrCatch(ShapoidErr);
    }
    VecSet(&p, 0, 0.1); VecSet(&p, 1, 0.1);
    ShapoidSetPos(facB, &p);
    if (FacoidAlignedIsOutsideFacoidAligned(facB, facA) == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg,
            "FacoidAlignedIsOutsideFacoidAligned failed");
        PBErrCatch(ShapoidErr);
    }
    VecSet(&u, 0, 1.0); VecSet(&v, 1, 1.0);
    ShapoidSetAxis(facB, 0, &u);
    ShapoidSetAxis(facB, 1, &v);
    if (FacoidAlignedIsOutsideFacoidAligned(facB, facA) == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg,
            "FacoidAlignedIsOutsideFacoidAligned failed");
        PBErrCatch(ShapoidErr);
    }
    ShapoidFree(&facA);
    ShapoidFree(&facB);
    printf("UnitTestFacoidAlignedIsOutsideFacoidAligned OK\n");
}

void UnitTestFacoidAlignedSplitExcludingFacoidAligned() {
    Facoid* facA = FacoidCreate(2);
    Facoid* facB = FacoidCreate(2);
    VecFloat2D p = VecFloatCreateStatic2D();
    VecFloat2D u = VecFloatCreateStatic2D();
    VecFloat2D v = VecFloatCreateStatic2D();
    VecSet(&p, 0, 0.0); VecSet(&p, 1, 0.0);
    VecSet(&u, 0, 1.0); VecSet(&v, 1, 1.0);
    ShapoidSetPos(facA, &p);
    ShapoidSetAxis(facA, 0, &u);
    ShapoidSetAxis(facA, 1, &v);
    VecSet(&p, 0, 0.5); VecSet(&p, 1, 0.5);
    VecSet(&u, 0, 1.0); VecSet(&v, 1, 1.0);
    ShapoidSetPos(facB, &p);
    ShapoidSetAxis(facB, 0, &u);
    ShapoidSetAxis(facB, 1, &v);
    GSetShapoid* split =

```



```

    FacoidAlignedSplitExcludingFacoidAligned(facA, facB);
    Facoid* facC = (Facoid*)GSetPop(split);
    VecSet(&p, 0, 0.0); VecSet(&p, 1, 0.0);
    VecSet(&u, 0, 1.0); VecSet(&v, 1, 0.5);
    if (VecIsEqual(ShapoidPos(facC), &p) == false ||
        VecIsEqual(ShapoidAxis(facC, 0), &u) == false ||
        VecIsEqual(ShapoidAxis(facC, 1), &v) == false) {
        ShapoidErr->_type = PErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg,
            "FacoidAlignedSplitExcludingFacoidAligned failed");
        PErrCatch(ShapoidErr);
    }
    Facoid* facD = (Facoid*)GSetPop(split);
    VecSet(&p, 0, 0.0); VecSet(&p, 1, 0.5);
    VecSet(&u, 0, 0.5); VecSet(&v, 1, 0.5);
    if (VecIsEqual(ShapoidPos(facD), &p) == false ||
        VecIsEqual(ShapoidAxis(facD, 0), &u) == false ||
        VecIsEqual(ShapoidAxis(facD, 1), &v) == false) {
        ShapoidErr->_type = PErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg,
            "FacoidAlignedSplitExcludingFacoidAligned failed");
        PErrCatch(ShapoidErr);
    }
    GSetFree(&split);
    ShapoidFree(&facA);
    ShapoidFree(&facB);
    ShapoidFree(&facC);
    ShapoidFree(&facD);
    printf("UnitTestFacoidAlignedSplitExcludingFacoidAligned OK\n");
}

void UnitTestFacoidAlignedAddClippedToSet() {
    Facoid* facA = FacoidCreate(2);
    VecFloat2D p = VecFloatCreateStatic2D();
    VecFloat2D u = VecFloatCreateStatic2D();
    VecFloat2D v = VecFloatCreateStatic2D();
    VecSet(&p, 0, 0.0); VecSet(&p, 1, 0.0);
    VecSet(&u, 0, 10.0); VecSet(&v, 1, 10.0);
    ShapoidSetPos(facA, &p);
    ShapoidSetAxis(facA, 0, &u);
    ShapoidSetAxis(facA, 1, &v);
    GSetShapoid set = GSetShapoidCreateStatic();
    FacoidAlignedAddClippedToSet(facA, &set);
    VecSet(&p, 0, 15.0); VecSet(&p, 1, 15.0);
    ShapoidSetPos(facA, &p);
    FacoidAlignedAddClippedToSet(facA, &set);
    VecSet(&p, 0, 8.0); VecSet(&p, 1, 8.0);
    ShapoidSetPos(facA, &p);
    FacoidAlignedAddClippedToSet(facA, &set);
    VecSet(&p, 0, 12.0); VecSet(&p, 1, 9.0);
    ShapoidSetPos(facA, &p);
    VecSet(&u, 0, 1.0); VecSet(&v, 1, 10.0);
    ShapoidSetAxis(facA, 0, &u);
    ShapoidSetAxis(facA, 1, &v);
    FacoidAlignedAddClippedToSet(facA, &set);
    VecSet(&p, 0, 5.0); VecSet(&p, 1, 5.0);
    ShapoidSetPos(facA, &p);
    VecSet(&u, 0, 1.0); VecSet(&v, 1, 1.0);
    ShapoidSetAxis(facA, 0, &u);
    ShapoidSetAxis(facA, 1, &v);
    FacoidAlignedAddClippedToSet(facA, &set);
    if (GSetNbElem(&set) != 6) {

```

```

    ShapoidErr->_type = PBErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg,
        "FacoidAlignedAddClippedToSet failed");
    PBErrCatch(ShapoidErr);
}
int iCheck = 0;
float checkp[12] = {
    0.0, 0.0, 15.0, 15.0, 10.0, 8.0, 8.0,
    10.0, 8.0, 15.0, 12.0, 18.0};
float checku[6] = {10.0, 10.0, 8.0, 10.0, 7.0, 1.0};
float checkv[6] = {10.0, 10.0, 2.0, 5.0, 3.0, 1.0};
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,
            "FacoidAlignedAddClippedToSet failed");
        PBErrCatch(ShapoidErr);
    }
    ShapoidFree(&fac);
    ++iCheck;
} while(GSetNbElem(&set) > 0);
ShapoidFree(&facA);
printf("UnitTestFacoidAlignedAddClippedToSet OK\n");
}

void UnitTestIsInter() {
    Spheroid* spheroidA = SpheroidCreate(3);
    Spheroid* spheroidB = SpheroidCreate(3);
    VecFloat3D v = VecFloatCreateStatic3D();
    if (ShapoidIsInter(spheroidA, spheroidB) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsInter failed");
        PBErrCatch(ShapoidErr);
    }
    VecSet(&v, 0, 1.1);
    ShapoidSetPos(spheroidB, &v);
    if (ShapoidIsInter(spheroidB, spheroidA) == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsInter failed");
        PBErrCatch(ShapoidErr);
    }
    VecSet(&v, 1, 1.0);
    ShapoidSetPos(spheroidB, &v);
    if (ShapoidIsInter(spheroidA, spheroidB) == true) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsInter failed");
        PBErrCatch(ShapoidErr);
    }
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 1.1);
    ShapoidSetPos(spheroidB, &v);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 2.0); VecSet(&v, 2, 1.0);
    ShapoidScale(spheroidB, (VecFloat*)&v);
    if (ShapoidIsInter(spheroidA, spheroidB) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsInter failed");
        PBErrCatch(ShapoidErr);
    }
}

```

```

    }
    ShapoidRotZCenter(spheroidB, -PBMath_QUARTERPI);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 0.0);
    ShapoidSetPos(spheroidB, &v);
    if (ShapoidIsInter(spheroidA, spheroidB) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIsInter failed");
        PBErrCatch(ShapoidErr);
    }
    ShapoidFree(&spheroidA);
    ShapoidFree(&spheroidB);
    printf("UnitTestIsInter OK\n");
}

void UnitTestShapoidIterCreateFree() {
    Facoid* facoid = FacoidCreate(2);
    VecFloat2D delta = VecFloatCreateStatic2D();
    ShapoidIter iter = ShapoidIterCreateStatic(facoid, &delta);
    if (iter._shap != (Shapoid*)facoid ||
        iter._pos == NULL ||
        iter._delta == NULL ||
        VecGetDim(iter._pos) != 2 ||
        VecGetDim(iter._delta) != 2) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIterFreeStatic failed");
        PBErrCatch(ShapoidErr);
    }
    ShapoidFree(&facoid);
    ShapoidIterFreeStatic(&iter);
    if (iter._pos != NULL ||
        iter._delta != NULL) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIterFreeStatic failed");
        PBErrCatch(ShapoidErr);
    }
    printf("UnitTestShapoidIterCreateFree OK\n");
}

void UnitTestShapoidIterGetSet() {
    Facoid* facoidA = FacoidCreate(2);
    Facoid* facoidB = FacoidCreate(2);
    VecFloat2D deltaA = VecFloatCreateStatic2D();
    VecFloat2D deltaB = VecFloatCreateStatic2D();
    for (int i = 2; i--;) {
        VecSet(&deltaA, i, 0.1);
        VecSet(&deltaB, i, 0.2);
    }
    ShapoidIter iter = ShapoidIterCreateStatic(facoidA, &deltaA);
    if (ShapoidIterShapoid(&iter) != (Shapoid*)facoidA) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIterShapoid failed");
        PBErrCatch(ShapoidErr);
    }
    if (VecIsEqual(ShapoidIterDelta(&iter), &deltaA) == false) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIterDelta failed");
        PBErrCatch(ShapoidErr);
    }
    ShapoidIterSetShapoid(&iter, facoidB);
    ShapoidIterSetDelta(&iter, &deltaB);
    if (ShapoidIterShapoid(&iter) != (Shapoid*)facoidB) {
        ShapoidErr->_type = PBErrTypeUnitTestFailed;
    }
}

```

```

        sprintf(ShapoidErr->_msg, "ShapoidIterSetShapoid failed");
        PBErCatch(ShapoidErr);
    }
    if (VecIsEqual(ShapoidIterDelta(&iter), &deltaB) == false) {
        ShapoidErr->_type = PBErTypeUnitTestFailed;
        sprintf(ShapoidErr->_msg, "ShapoidIterSetDelta failed");
        PBErCatch(ShapoidErr);
    }
    ShapoidFree(&facoidA);
    ShapoidFree(&facoidB);
    ShapoidIterFreeStatic(&iter);
    printf("UnitTestShapoidIterGetSet OK\n");
}

void UnitTestShapoidIterStepFacoid() {
    Facoid* facoid = FacoidCreate(2);
    VecFloat2D delta = VecFloatCreateStatic2D();
    for (int i = 2; i--;)
        VecSet(&delta, i, 0.25);
    ShapoidIter iter = ShapoidIterCreateStatic(facoid, &delta);
    int iCheck = 0;
    float check[50] = {0.000,0.000,0.000,0.250,0.000,0.500,0.000,0.750,
        0.000,1.000,0.250,0.000,0.250,0.250,0.250,0.500,0.250,0.750,0.250,
        1.000,0.500,0.000,0.500,0.250,0.500,0.500,0.500,0.750,0.500,1.000,
        0.750,0.000,0.750,0.250,0.750,0.500,0.750,0.750,0.750,1.000,1.000,
        0.000,1.000,0.250,1.000,0.500,1.000,0.750,1.000,1.000
    };
    do {
        VecFloat* v = ShapoidIterGetInternalPos(&iter);
        if (ISEQUALF(VecGet(v, 0), check[2 * iCheck]) == false ||
            ISEQUALF(VecGet(v, 1), check[2 * iCheck + 1]) == false) {
            ShapoidErr->_type = PBErTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidIterStep failed");
            PBErCatch(ShapoidErr);
        }
        VecFree(&v);
        ++iCheck;
    } while (ShapoidIterStep(&iter));
    ShapoidFree(&facoid);
    ShapoidIterFreeStatic(&iter);
    printf("UnitTestShapoidIterStepFacoid OK\n");
}

void UnitTestShapoidIterStepPyramidoid() {
    Pyramidoid* pyramidoid = PyramidoidCreate(3);
    VecFloat3D delta = VecFloatCreateStatic3D();
    for (int i = 3; i--;)
        VecSet(&delta, i, 0.25);
    ShapoidIter iter = ShapoidIterCreateStatic(pyramidoid, &delta);
    int iCheck = 0;
    float check[105] = {0.000,0.000,0.000,0.000,0.000,0.250,0.000,0.000,
        0.500,0.000,0.000,0.750,0.000,0.000,1.000,0.000,0.250,0.000,0.000,
        0.250,0.250,0.000,0.250,0.500,0.000,0.250,0.750,0.000,0.500,0.000,
        0.000,0.500,0.250,0.000,0.500,0.500,0.000,0.750,0.000,0.000,0.750,
        0.250,0.000,1.000,0.000,0.250,0.000,0.000,0.250,0.000,0.250,0.250,
        0.000,0.500,0.250,0.000,0.750,0.250,0.250,0.000,0.250,0.250,0.250,
        0.250,0.250,0.500,0.250,0.500,0.000,0.250,0.500,0.250,0.250,0.750,
        0.000,0.500,0.000,0.000,0.500,0.000,0.250,0.500,0.000,0.500,0.500,
        0.250,0.000,0.500,0.250,0.250,0.500,0.500,0.000,0.750,0.000,0.000,
        0.750,0.000,0.250,0.750,0.250,0.000,1.000,0.000,0.000
    };
    do {

```

```

VecFloat* v = ShapoidIterGetInternalPos(&iter);
if (ISEQUALF(VecGet(v, 0), check[3 * iCheck]) == false ||
    ISEQUALF(VecGet(v, 1), check[3 * iCheck + 1]) == false ||
    ISEQUALF(VecGet(v, 2), check[3 * iCheck + 2]) == false) {
    ShapoidErr->_type = PErrTypeUnitTestFailed;
    sprintf(ShapoidErr->_msg, "ShapoidIterStep failed");
    PErrCatch(ShapoidErr);
}
VecFree(&v);
++iCheck;
} while (ShapoidIterStep(&iter));
ShapoidFree(&pyramidoid);
ShapoidIterFreeStatic(&iter);
printf("UnitTestShapoidIterStepPyramidoid OK\n");
}

void UnitTestShapoidIterStepSpheroid() {
    int dim = 3;
    Spheroid* spheroid = SpheroidCreate(dim);
    VecFloat* delta = VecFloatCreate(dim);
    for (int i = dim; i--;)
        VecSet(delta, i, 0.25);
    ShapoidIter iter = ShapoidIterCreateStatic(spheroid, delta);
    int iCheck = 0;
    float check[147] = {
        0.00000,0.00000,-0.50000,0.00000,-0.43301,-0.25000,-0.39244,
        -0.18301,-0.25000,-0.14244,-0.18301,-0.25000,0.10756,-0.18301,
        -0.25000,0.35756,-0.18301,-0.25000,0.39244,-0.18301,-0.25000,
        -0.42780,0.06699,-0.25000,-0.17780,0.06699,-0.25000,0.07220,
        0.06699,-0.25000,0.32220,0.06699,-0.25000,0.42780,0.06699,
        -0.25000,-0.29499,0.31699,-0.25000,-0.04499,0.31699,-0.25000,
        0.20501,0.31699,-0.25000,0.29499,0.31699,-0.25000,0.00000,
        -0.50000,0.00000,-0.43301,-0.25000,0.00000,-0.18301,-0.25000,
        0.00000,0.06699,-0.25000,0.00000,0.31699,-0.25000,0.00000,0.43301,
        -0.25000,0.00000,-0.50000,0.00000,0.00000,-0.25000,0.00000,0.00000,
        0.00000,0.00000,0.00000,0.25000,0.00000,0.00000,0.50000,0.00000,
        0.00000,-0.43301,0.25000,0.00000,-0.18301,0.25000,0.00000,0.06699,
        0.25000,0.00000,0.31699,0.25000,0.00000,0.43301,0.25000,0.00000,
        0.00000,0.50000,0.00000,0.00000,-0.43301,0.25000,-0.39244,-0.18301,
        0.25000,-0.14244,-0.18301,0.25000,0.10756,-0.18301,0.25000,0.35756,
        -0.18301,0.25000,0.39244,-0.18301,0.25000,-0.42780,0.06699,0.25000,
        -0.17780,0.06699,0.25000,0.07220,0.06699,0.25000,0.32220,0.06699,
        0.25000,0.42780,0.06699,0.25000,-0.29499,0.31699,0.25000,-0.04499,
        0.31699,0.25000,0.20501,0.31699,0.25000,0.29499,0.31699,0.25000,
        0.00000,0.00000,0.50000
    };
    do {
        VecFloat* v = ShapoidIterGetInternalPos(&iter);
        if (ISEQUALF(VecGet(v, 0), check[3 * iCheck]) == false ||
            ISEQUALF(VecGet(v, 1), check[3 * iCheck + 1]) == false ||
            ISEQUALF(VecGet(v, 2), check[3 * iCheck + 2]) == false) {
            ShapoidErr->_type = PErrTypeUnitTestFailed;
            sprintf(ShapoidErr->_msg, "ShapoidIterStep failed");
            PErrCatch(ShapoidErr);
        }
        VecFree(&v);
        ++iCheck;
    } while (ShapoidIterStep(&iter));
    ShapoidFree(&spheroid);
    ShapoidIterFreeStatic(&iter);
    VecFree(&delta);
    printf("UnitTestShapoidIterStepSpheroid OK\n");
}

```

```

}

void UnitTestShapoidIter() {
    UnitTestShapoidIterCreateFree();
    UnitTestShapoidIterGetSet();
    UnitTestShapoidIterStepFacoid();
    UnitTestShapoidIterStepPyramidoid();
    UnitTestShapoidIterStepSpheroid();

    printf("UnitTestShapoidIter OK\n");
}

void UnitTestAll() {
    UnitTestCreateCloneIsEqualFree();
    UnitTestLoadSavePrint();
    UnitTestGetSetTypeDimPosAxis();
    UnitTestTranslateScaleGrow();
    UnitTestRotate();
    UnitTestRotateAxis();
    UnitTestRotateX();
    UnitTestRotateY();
    UnitTestRotateZ();
    UnitTestImportExportCoordIsPosInside();
    UnitTestGetBoundingBox();
    UnitTestGetPosDepthCenterCoverage();
    UnitTestFacoidAlignedIsInsideFacoidAligned();
    UnitTestFacoidAlignedIsOutsideFacoidAligned();
    UnitTestFacoidAlignedSplitExcludingFacoidAligned();
    UnitTestFacoidAlignedAddClippedToSet();
    UnitTestIsInter();
    UnitTestShapoidIter();
    printf("UnitTestAll OK\n");
}

int main() {
    UnitTestAll();
    // Return success code
    return 0;
}

```

6 Unit tests output

```

UnitTestCreateCloneIsEqualFree OK
Type: Facoid
Dim: 3
Pos: <0.000,0.000,0.000>
Axis(0): <1.000,0.000,0.000>
Axis(1): <0.000,1.000,0.000>
Axis(2): <0.000,0.000,1.000>
UnitTestLoadSavePrint OK
UnitTestGetSetTypeDimPosAxis OK
UnitTestTranslateScaleGrow OK
UnitTestRotate OK
UnitTestRotateAxis OK
UnitTestRotateX OK
UnitTestRotateY OK
UnitTestRotateZ OK
UnitTestImportExportCoordIsPosInside OK

```

```

UnitTestGetBoundingBox OK
UnitTestGetPosDepthCenterCoverage OK
UnitTestFacoidAlignedIsInsideFacoidAligned OK
UnitTestFacoidAlignedIsOutsideFacoidAligned OK
UnitTestFacoidAlignedSplitExcludingFacoidAligned OK
UnitTestFacoidAlignedAddClippedToSet OK
UnitTestIsInter OK
UnitTestShapoidIterCreateFree OK
UnitTestShapoidIterGetSet OK
UnitTestShapoidIterStepFacoid OK
UnitTestShapoidIterStepPyramidoid OK
UnitTestShapoidIterStepSpheroid OK
UnitTestShapoidIter OK
UnitTestAll OK

```

facoid.txt

```

{
  "_dim": "3",
  "_type": "0",
  "_pos": {
    "_dim": "3",
    "_val": ["0.000000", "0.000000", "0.000000"]
  },
  "_axis": [
    {
      "_dim": "3",
      "_val": ["1.000000", "0.000000", "0.000000"]
    },
    {
      "_dim": "3",
      "_val": ["0.000000", "1.000000", "0.000000"]
    },
    {
      "_dim": "3",
      "_val": ["0.000000", "0.000000", "1.000000"]
    }
  ]
}

```

Example of path on a 2D Spheroid using the ShapoidIterator:

