Cognex 3D-Locate

Class Reference

CVL 8.0

June 2016

Preface

This manual contains reference information for Cognex 3D-Locate, a collection of software tools for performing 3D machine vision.

Style Conventions Used in This Manual

This manual uses the style conventions described in this section for text and software diagrams.

Text Style Conventions

This manual uses the following style conventions for text:

boldface	Used for C/C++ keywords, function names, class names, structures, enumerations, types, and macros. Also used for user interface elements such as button names, dialog box names, and menu choices.
italic	Used for names of variables, data members, arguments, enumerations, constants, program names, file names. Used for names of books, chapters, and sections. Occasionally used for emphasis.
`courier`	Used for C/C++ code examples and for examples of program output.
`bold courier`	Used in illustrations of command sessions to show the commands that you would type.
<italic>	When enclosed in angle brackets, used to indicate keyboard keys such as *<Tab>* or *<Enter>*.

Microsoft Windows Support

Cognex CVL software runs on specific Microsoft Windows operating systems. In this documentation set, these are abbreviated to Windows unless there is a feature specific to one of the variants. Consult the *Getting Started* manual for your CVL release for details on the operating systems, hardware, and software supported by that release.

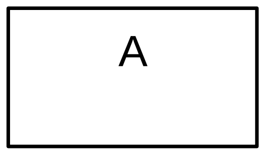
Software Diagramming Conventions

This manual uses the following symbols in class diagrams:

- **Classes** are shown as a box with the class name centered inside the box. For example, a class A with the C++ declaration

  ```
  class A{};
  ```

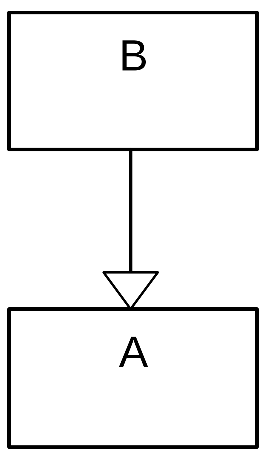
  is shown graphically as follows:



- **Inheritance** relationships between classes are shown using solid-line arrows from the derived class to the base class with a large, hollow triangle pointing toward the base class. For example, a class B that inherits from a class A with the declaration

  ```
  class B : public A {};
  ```

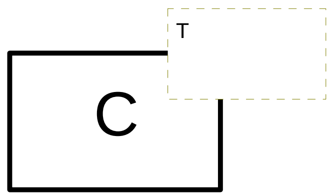
  is shown graphically as follows:



- **Template classes** are shown as a class box with a smaller, dotted-line rectangle representing the template parameter superimposed on the upper right corner of the class box. For example, a template class C with a parameter of type class T with the declaration:

```
template <class T>
class C{};
```

is shown graphically as follows:



These symbols are based on the Unified Modeling Language (UML), a standard graphical notation for object-oriented analysis and design. See the latest *OMG Unified Modeling Language Specification* (available from the Object Management Group at http://www.omg.org) for more information.

Cognex Offices

Cognex Corporation serves its customers from the following locations:

Corporate Headquarters	Cognex Corporation Corporate Headquarters One Vision Drive Natick, MA 01760-2059 (508) 650-3000
Web Site	http://www.cognex.com

3D Pose Functions

```
#include <ch_c3d/cdbpose.h>
```

For 3D applications, it is often useful to associate a pose with an image. Towards this end, this file defines an API for encoding/decoding a 3D/ 6DOF pose into a string and/or the comment field of a CDB record.

The encode/decode functions are named as follows:

- **cf3DEncodePoseIntoString()**
- **cf3DEncodePoseIntoCDBRecordComment()**
- **cf3DDecodePoseFromString()**
- **cf3DDecodePoseFromCDBRecordComment()**

Functions

cf3DEncodePoseIntoString

```
void cf3DEncodePoseIntoString(const cc3DXformRigid& pose,
  ccCvlString& comment);
```

Encodes the given pose into the given string.

Parameters

pose	The given pose.
comment	The given string.

cf3DEncodePoseIntoCDBRecordComment

```
void cf3DEncodePoseIntoCDBRecordComment(const
  cc3DXformRigid& pose, ccCDBRecord& cdbRecord);
```

Encodes the given pose into the comment field of the given record.

Parameters

pose	The given pose.
cdbRecord	The given record.

Notes

The encoding overwrites the comment in the cdbRecord.

■ 3D Pose Functions

cf3DDecodePoseFromString

```
void cf3DDecodePoseFromString(const ccCvlString& comment,
  cc3DXformRigid& pose);
```

Decodes the pose from the given string.

Parameters

pose The pose.

comment The given string.

Throws

ccInvariantFailure
The pose cannot be decoded from the string.

cf3DDecodePoseFromCDBRecordComment

```
void cf3DDecodePoseFromCDBRecordComment(const ccCDBRecord&
  cdbRecord, cc3DXformRigid& pose);
```

Decodes the pose from the comment field in the given cdb record.

Parameters

pose The pose.

cdbRecord The given cdb record.

Throws

ccInvariantFailure
The pose cannot be decoded from the cdb record.

cc3DAlignedBox

```
#include <ch_c3d/shapes3d.h>

class cc3DAlignedBox:
   public cc3DShape,
   public cc3DVertex,
   public cc3DCurve,
   public cc3DSurface,
   public cc3DVolume;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class represents a box that is aligned to the 3D coordinate system.

Constructors/Destructors

cc3DAlignedBox

```
cc3DAlignedBox();

cc3DAlignedBox(const cc3DVect& size, double transX,
  double transY, double transZ,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eVolume);

cc3DAlignedBox(const cc3DVect& originVertex,
  const cc3DVect& oppositeVertex,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eVolume);
```

•

```
cc3DAlignedBox();
```

Default constructor that constructs a degenerate aligned box with the following default values:

- **size()** is **cc3DVect(0,0,0)**
- **translation()** is **cc3DVect(0,0,0)**
- **stateType()** is *cc3DShapeDefs::eVolume*

-
```
cc3DAlignedBox(const cc3DVect& size, double transX,
  double transY, double transZ,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eVolume);
```

Constructs a 3D aligned box using the given size, translation, and state type.

Parameters

size	The size of the box
transX	The translation of the box in the X-direction
transY	The translation of the box in the Y-direction
transZ	The translation of the box in the Z-direction
type	The initial state type for this box. You must supply one of the following values for this parameter: *cc3DShapeDefs::eVertex* *cc3DShapeDefs::eCurve* *cc3DShapeDefs::eSurface* *cc3DShapeDefs::eVolume*

Throws

cc3DShapeDefs::BadParams
Any member of *size* is less than 0.

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface
cc3DShapeDefs::eVolume

-
```
cc3DAlignedBox(const cc3DVect& originVertex,
  const cc3DVect& oppositeVertex,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eVolume);
```

Constructs a 3D aligned box using two opposite vertices of the box and the given state type.

Parameters

originVertex	The position of the origin of the box.
oppositeVertex	The position of the opposite vertex from the origin.
type	The initial state type for this box. You must supply one of the following values for this parameter:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface
cc3DShapeDefs::eVolume

Throws

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface
cc3DShapeDefs::eVolume

Notes

The size of the constructed box is defined as follows:

```
size().x() = fabs(oppositeVertex.x() - originVertex.x())
size().y() = fabs(oppositeVertex.y() - originVertex.y())
size().z() = fabs(oppositeVertex.z() - originVertex.z())
```

The translation vector is defined by

```
min( originVertex.x(), oppositeVertex.x())
min( originVertex.y(), oppositeVertex.y())
min( originVertex.z(), oppositeVertex.z())
```

Public Member Functions

clone

```
virtual cc3DShapePtrh clone () const;
```

This is an override from class **cc3DShape**.

isFinite

```
virtual bool isFinite () const;
```

This is an override from class **cc3DShape**.

isEmpty

```
virtual bool isEmpty () const;
```

This is an override from class **cc3DShape**.

nearestPoint

```
virtual cc3DVect nearestPoint (const cc3DVect &pt) const;
```

This is an override from class **cc3DShape**.

Parameters

pt The point to which to determine the nearest point on this **cc3DAignedBox**.

boundingBox

```
virtual cc3DAlignedBox boundingBox() const;
```

This is an override from class **cc3DShape**.

mapShape

```
virtual cc3DShapePtrh mapShape(
   const cc3DXformBase& xform) const;

virtual void mapShape (const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const;
```

- ```
  virtual cc3DShapePtrh mapShape(
     const cc3DXformBase& xform) const;
  ```

Maps this shape with the supplied **cc3DXformBase**. The transformed shape is of type **cc3DBox** where *xform* is a rigid transform.

This is an override from class **cc3DShape**.

Parameters

xform The transform with which to map.

- ```
  virtual void mapShape (const cc3DXformBase& xform,
     cc3DShapePtrh& dst) const;
  ```

Maps this shape with the supplied **cc3DXformBase**. The transformed shape is of type **cc3DBox** where *xform* is a rigid transform.

This is an override from class **cc3DShape**.

Parameters

xform The transform with which to map.

dst The transformed shape. *dst* is set to a shape of type **cc3DBox** where *xform* is a rigid transform

nearestPointVertex

```
virtual cc3DVect nearestPointVertex(
   const cc3DVect &pt) const;
```

This is an override from class **cc3DVertex**.

Parameters

pt The point.

distanceVertex

```
virtual double distanceVertex(const cc3DVect &pt) const;
```

This is an override from class **cc3DVertex**.

Parameters

pt The point.

perimeter

```
virtual double perimeter() const;
```

This is an override from class **cc3DCurve**.

Regardless of whether the shape is degenerate, the perimeter is defined to be:

```
(size().x() + size.y() + size().z()) * 4
```

nearestPointCurve

```
virtual cc3DVect nearestPointCurve(
   const cc3DVect &pt) const;
```

This is an override from class **cc3DCurve**.

Parameters

pt The point.

area

```
virtual double area() const;
```

This is an override from class **cc3DSurface**.

Regardless of whether the shape is degenerate, the area is defined to be:

```
(size().x() * size.y() + size().z() * size.y() + size().z() * size.x()) * 2
```

nearestPointSurface

```
virtual cc3DVect nearestPointSurface(
   const cc3DVect &pt) const;
```

This is an override from class **cc3DSurface**.

Parameters

pt The point.

volume

```
virtual double volume() const;
```

This is an override from class **cc3DVolume**.

nearestPointVolume

```
virtual cc3DVect nearestPointVolume(
   const cc3DVect &pt) const;
```

This is an override from class **cc3DVolume**.

Parameters

pt The point.

stateType

```
virtual cc3DShapeDefs::StateType stateType() const;

void stateType(cc3DShapeDefs::StateType type);
```

- `virtual cc3DShapeDefs::StateType stateType() const;`

Returns the state type of this object.

- `void stateType(cc3DShapeDefs::StateType type);`

Sets the state type of this aligned box. The state type influences how various methods (such as **nearestPoint()**) inherited from **cc3DShape** class are interpreted.

Parameters

type The state type. *type* must be one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface
cc3DShapeDefs::eVolume

Notes

The default shape type is *cc3DShapeDefs::eVolume*.

Throws

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface
cc3DShapeDefs::eVolume

getSizeAndTranslation

```
void getSizeAndTranslation(
  cc3DVect& size, cc3DVect& trans) const;
```

Returns the size and translation of this **cc3DAlignedBox**.

Parameters

size A **cc3DVect** into which the size is placed.

trans A **cc3DVect** into which the translation is placed.

setSizeAndTranslation

```
void setSizeAndTranslation(const cc3DVect& size,
   const cc3DVect& trans);
```

Sets the size and translation of this **cc3DAlignedBox**.

Parameters

size A **cc3DVect** containing the new size.

trans A **cc3DVect** containing the new translation.

Throws

cc3DShapeDefs::BadParams
Any component of *size* is less than 0.

getCenterAndSize

```
void getCenterAndSize(cc3DVect& center,
   cc3DVect& size) const;
```

Returns the size and center of this **cc3DAlignedBox**.

Parameters

center A **cc3DVect** into which the center is placed.

size A **cc3DVect** into which the size is placed.

setCenterAndSize

```
void setCenterAndSize(const cc3DVect& center,
   const cc3DVect& size);
```

Sets the size and center of this **cc3DAlignedBox**.

Parameters

center A **cc3DVect** containing the new center.

size A **cc3DVect** containing the new size.

Throws

cc3DShapeDefs::BadParams
Any component of *size* is less than 0.

Notes

Calling this function may change the value of **translation()**.

size

```
cc3DVect size() const;

void size(const cc3DVect& newSize);
```

- ```
  cc3DVect size() const;
  ```

Returns the size of this **cc3DAlignedBox**.

- ```
  void size(const cc3DVect& newSize);
  ```

Sets the size of this **cc3DAlignedBox**.

Parameters

newSize A **cc3DVect** containing the x-, y-, and z- dimensions to set.

Throws

cc3DShapeDefs::BadParams
Any component of *newSize* is less than 0.

Notes

The default value is **cc3DVect(0,0,0)**.

The setter does not change the origin vertex (which corresponds to **cc3DVect(0,0,0)** in the unit box).

center

```
cc3DVect center() const;

void center(const cc3DVect& newCenter);
```

- ```
  cc3DVect center() const;
  ```

Returns the center of this **cc3DAlignedBox**

- ```
  void center(const cc3DVect& newCenter);
  ```

Sets the center of this **cc3DAlignedBox**.

Parameters

newCenter The new center.

Notes

Calling this function may change the value of **translation()**.

setSizeAndKeepCenterUnchanged

```
void setSizeAndKeepCenterUnchanged(
   const cc3DVect& newSize);
```

Sets the size of this **cc3DAlignedBox** while keeping its center fixed.

Parameters

newSize The new size.

Throws

cc3DShapeDefs::BadParams
Any component of *newSize* is less than 0.

Notes

This setter does not change the center of this aligned box, but might change the origin vertex (which corresponds to **cc3DVect(0,0,0)** in the unit box).

The setter might change the value of **shapeFromScaledUnit()**.

translation

```
cc3DVect translation() const;

void translation(const cc3DVect& newTrans);
```

- `cc3DVect translation() const;`

Returns the current translation of this **cc3DAlignedBox**.

- `void translation(const cc3DVect& newTrans);`

Sets the current translation of this **cc3DAlignedBox**.

The default value is **cc3DVect(0,0,0)**.

Parameters

newTrans The translation to set.

map

```
cc3DBox map(const cc3DXformRigid &xform) const;

void map(const cc3DXformRigid &xform, cc3DBox& dst) const;
```

- `cc3DBox map(const cc3DXformRigid &xform) const;`

Returns this **cc3DAlignedBox** mapped by the supplied transform.

Parameters

xform The transform.

- `void map(const cc3DXformRigid &xform, cc3DBox& dst) const;`

Sets the supplied **cc3DBox** to be the result of mapping this **cc3DAlignedBox** by the supplied transform.

Parameters

xform The transform

dst The **cc3DBox** into which to place the result.

mapTrans

```
cc3DAlignedBox mapTrans(const cc3DVect& trans) const;

void mapTrans(const cc3DVect& trans,
   cc3DAlignedBox& dst) const;
```

- `cc3DAlignedBox mapTrans(const cc3DVect& trans) const;`

Returns the result of translating this **cc3DAlignedBox** by the supplied values.

Parameters

trans The translation to apply.

- `void mapTrans(const cc3DVect& trans, cc3DAlignedBox& dst) const;`

Sets the supplied **cc3DAlignedBox** to be the result of translating this **cc3DAlignedBox** by the supplied values.

Parameters

trans The translation to apply.

dst The **cc3DAlignedBox** into which to place the result.

getOriginVertexAndOppositeVertex

```
void getOriginVertexAndOppositeVertex(
   cc3DVect& originVertex, cc3DVect& oppositeVertex) const;
```

Returns the position of the **cc3DAlignedBox** origin and the vertex opposite from the origin.

The origin vertex is the one that corresponds to **cc3DVect(0,0,0)** in the unit box while the opposite vector corresponds to **cc3DVect(1,1,1)** in the unit box. This function may not return the same vertices used in the corresponding constructor or **setOriginVertexAndOppositeVertex()**.

Parameters

originVertex	The origin.
oppositeVertex	The vertex opposite the origin.

setOriginVertexAndOppositeVertex

```
void setOriginVertexAndOppositeVertex(
   const cc3DVect& originVertex,
   const cc3DVect& oppositeVertex);
```

Sets the position of the **cc3DAlignedBox** origin and the vertex opposite from the origin.

Parameters

originVertex	The origin.
oppsiteVertex	The vertex opposite the origin.

Notes

For the setter, the size of the box is defined as follows:

```
size().x() = fabs(oppositeVertex.x() - originVertex.x())
size().y() = fabs(oppositeVertex.y() - originVertex.y())
size().z() = fabs(oppositeVertex.z() - originVertex.z())
```

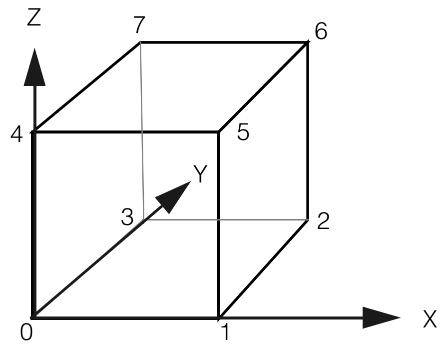
The translation vector is defined by

```
min( originVertex.x(), oppositeVertex.x())
min( originVertex.y(), oppositeVertex.y())
min( originVertex.z(), oppositeVertex.z())
```

■ cc3DAlignedBox

vertices `cmStd vector<cc3DVect> vertices() const;`

Returns the vertices of this box. The vertices are returned in the following order:



where vertex 0 corresponds to the origin vertex in the unit shape. More formally, the vertex order is given as follows, based on an (untransformed) unit square:

```
0   cc3DVect(0,0,0)
1   cc3DVect(1,0,0)
2   cc3DVect(1,1,0)
3   cc3DVect(0,1,0)
4   cc3DVect(0,0,1)
5   cc3DVect(1,0,1)
6   cc3DVect(1,1,1)
7   cc3DVect(0,1,1)
```

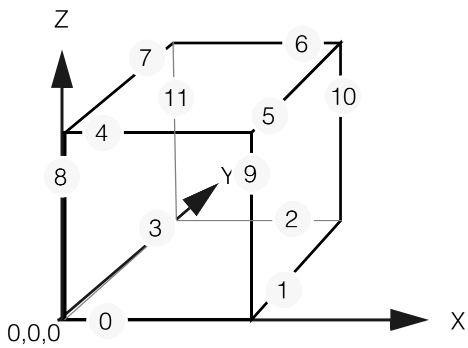
Notes

Some elements of the returned vector might be duplicate if this **cc3DAlignedBox** is degenerate.

lineSegs

```
cmStd vector<cc3DLineSeg> lineSegs() const;
```

Returns the line segments for the edges of this box. The segments are returned in the following order:



where the vertex marked *0,0,0* corresponds to the origin vertex in the unit shape. More formally, the edge order is given as follows, based on an (untransformed) unit square:

```
0   cc3DLineSeg(cc3DVect(0,0,0), cc3DVect(1,0,0))
1   cc3DLineSeg(cc3DVect(1,0,0), cc3DVect(1,1,0))
2   cc3DLineSeg(cc3DVect(1,1,0), cc3DVect(0,1,0))
3   cc3DLineSeg(cc3DVect(0,1,0), cc3DVect(0,0,0))
4   cc3DLineSeg(cc3DVect(0,0,1), cc3DVect(1,0,1))
5   cc3DLineSeg(cc3DVect(1,0,1), cc3DVect(1,1,1))
6   cc3DLineSeg(cc3DVect(1,1,1), cc3DVect(0,1,1))
7   cc3DLineSeg(cc3DVect(0,1,1), cc3DVect(0,0,1))
8   cc3DLineSeg(cc3DVect(0,0,0), cc3DVect(0,0,1))
9   cc3DLineSeg(cc3DVect(1,0,0), cc3DVect(1,0,1))
10  cc3DLineSeg(cc3DVect(1,1,0), cc3DVect(1,1,1))
11  cc3DLineSeg(cc3DVect(0,1,0), cc3DVect(0,1,1))
```

Notes

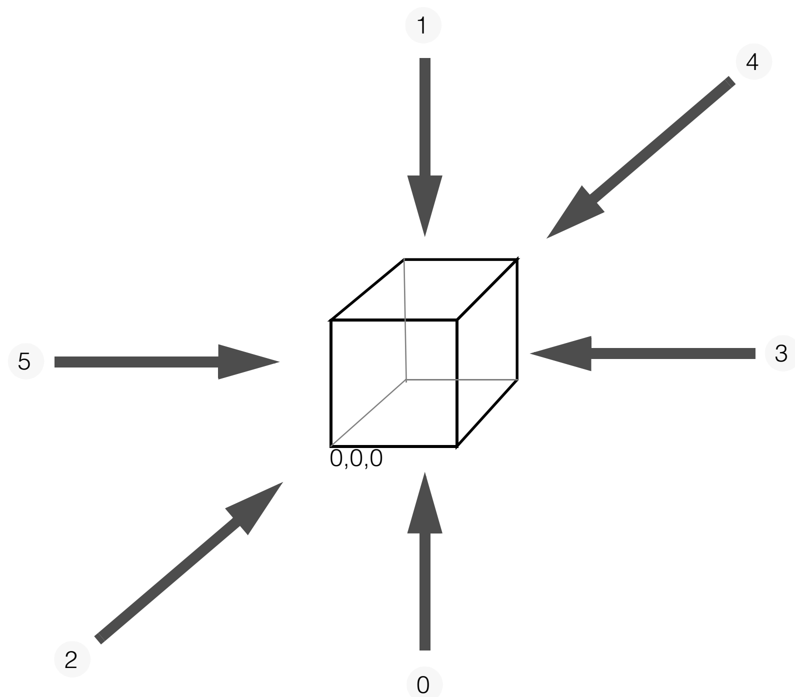
Some elements of the returned vector might be degenerate line segments and some elements might be duplicate if this **cc3DAlignedBox** itself is degenerate.

Each element of the returned vector has its state type set to *cc3DShapeDefs::eCurve*, regardless of the state type of the **cc3DAlignedBox**.

surfaces

```
cmStd vector<cc3DRect> surfaces() const;
```

Returns the surfaces of this box. The surfaces are returned in the following order:



where the vertex marked *0,0,0* corresponds to the origin vertex in the unit shape. More formally, the surface order is given as follows, based on an (untransformed) unit square (the vertex pairs define the opposite corners of each surface):

```
0   cc3DVect(0,0,0), cc3DVect(1,1,0)
1   cc3DVect(0,0,1), cc3DVect(1,1,1)
2   cc3DVect(0,0,0), cc3DVect(1,0,1)
3   cc3DVect(1,0,0), cc3DVect(1,1,1)
4   cc3DVect(0,1,0), cc3DVect(1,1,1)
5   cc3DVect(0,0,0), cc3DVect(0,1,1)
```

Notes

Some elements of the returned vector might be degenerate surfaces and some elements might be duplicate if this **cc3DAlignedBox** itself is degenerate.

Each element of the returned vector has the state type of *cc3DShapeDefs::eSurface* no matter the current state type of the box.

Operators

operator==

```
bool operator==(const cc3DAlignedBox& that) const;
```

Returns true if this object is exactly equal to *that*, and false otherwise.

Parameters

that The **cc3DAlignedBox** to compare to this one.

cc3DAxisAngle

```
#include <ch_c3d/axisang.h>
```

```
class cc3DAxisAngle;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class encapsulates the rotation axis and angle (around the rotation axis) representation. For more information on this angle representation, see

http://en.wikipedia.org/wiki/Rotation_representation_(mathematics)

Notes

This class is immutable.

Constructors/Destructors

cc3DAxisAngle

```
cc3DAxisAngle();

cc3DAxisAngle(const cc3DVect& axis, const ccRadian& angle);
```

- ```
  cc3DAxisAngle();
  ```

  Constructs a cc3DAxisAngle object using default parameters:

  ```
  angle() = ccRadian(0);
  axis() = cc3DVect(1, 0, 0);
  ```

- ```
  cc3DAxisAngle(const cc3DVect& axis, const ccRadian& angle);
  ```

  Constructs a cc3DAxisAngle object using the provided values.

  Parameters

axis	The axis of rotation, expressed as a vector.
angle	The amount of rotation.

Public Member Functions

angle

```
ccRadian angle() const;
```

Gets the rotation angle around the axis. For a default-constructed object, this is **ccRadian(0)**.

axis

```
cc3DVect axis() const;
```

Gets the rotation axis. For a default-constructed object, this is **cc3DVect(1,0,0)**.

Operators

operator==

```
bool operator==(const cc3DAxisAngle& that) const;
```

Returns true if this object is exactly equal to *that*, and false otherwise.

Parameters

that	The object to compare to this one.

cc3DBox

```
#include <ch_c3d/shapes3d.h>

class cc3DBox:
   public cc3DShape,
   public cc3DVertex,
   public cc3DCurve,
   public cc3DSurface,
   public cc3DVolume;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class represents an oriented box. The size of the box is defined by a single 3D vector, and the location and pose of the box is defined by a 3D rigid transformation.

Constructors/Destructors

cc3DBox

```
cc3DBox();

cc3DBox(const cc3DVect& size,
  const cc3DXformRigid& shapeFromScaledUnit,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eVolume);

cc3DBox(const cc3DAlignedBox& alignedBox);
```

•

```
cc3DBox();
```

Default constructor that creates a degenerate **cc3DBox** with the following default values:

- **size()** is **cc3DVect(0,0,0)**
- **shapeFromScaledUnit()** is identity
- **stateType()** is *cc3DShapeDefs::eVolume*

- ```
cc3DBox(const cc3DVect& size,
  const cc3DXformRigid& shapeFromScaledUnit,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eVolume);
```

Constructs a **cc3DBox** using the given size, the rigid transformation from the scaled unit box, and the state type.

**Parameters**

*size* — The size of the box (the **cc3DVect** contains the x-, y-, and z-dimensions of the box.

*shapeFromScaledUnit* — A rigid transformation giving the pose of the box.

*type* — The initial state type for this box. You must supply one of the following values for this parameter:

*cc3DShapeDefs::eVertex*
*cc3DShapeDefs::eCurve*
*cc3DShapeDefs::eSurface*
*cc3DShapeDefs::eVolume*

**Throws**

*cc3DShapeDefs::BadParams* — Any member of *size* is less than 0.

*cc3DShapeDefs::InvalidStateType* — *type* is not one of the following values:

*cc3DShapeDefs::eVertex*
*cc3DShapeDefs::eCurve*
*cc3DShapeDefs::eSurface*
*cc3DShapeDefs::eVolume*

- ```
cc3DBox(const cc3DAlignedBox& alignedBox);
```

Constructs a **cc3DBox** from the given **cc3DAlignedBox**.

Parameters

alignedBox — The **cc3DAlignedBox** from which to create this object.

Public Member Functions

clone

```
virtual cc3DShapePtrh clone () const;
```

This is an override from class **cc3DShape**.

isFinite

```
virtual bool isFinite () const;
```

This is an override from class **cc3DShape**.

isEmpty

```
virtual bool isEmpty () const;
```

This is an override from class **cc3DShape**.

nearestPoint

```
virtual cc3DVect nearestPoint (const cc3DVect &pt) const;
```

This is an override from class **cc3DShape**.

Parameters

pt	The point to which to determine the nearest point on this **cc3DBox**.

boundingBox

```
virtual cc3DAlignedBox boundingBox() const;
```

This is an override from class **cc3DShape**.

mapShape

```
virtual cc3DShapePtrh mapShape(
   const cc3DXformBase& xform) const;

virtual void mapShape (const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const;
```

- ```
  virtual cc3DShapePtrh mapShape(
     const cc3DXformBase& xform) const;
  ```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters

xform	The transform with which to map.

- ```
  virtual void mapShape (const cc3DXformBase& xform,
     cc3DShapePtrh& dst) const;
  ```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters

xform	The transform with which to map.

dst The transformed shape.

nearestPointVertex

```
virtual cc3DVect nearestPointVertex(
   const cc3DVect &pt) const;
```

This is an override from class **cc3DVertex**.

Parameters

pt The point.

distanceVertex

```
virtual double distanceVertex(const cc3DVect &pt) const;
```

This is an override from class **cc3DVertex**.

Parameters

pt The point.

perimeter

```
virtual double perimeter() const;
```

This is an override from class **cc3DCurve**.

Regardless of whether the shape is degenerate, the perimeter is defined to be:

```
(size().x() + size.y() + size().z()) * 4
```

nearestPointCurve

```
virtual cc3DVect nearestPointCurve(
   const cc3DVect &pt) const;
```

This is an override from class **cc3DCurve**.

Parameters

pt The point.

area

```
virtual double area() const;
```

This is an override from class **cc3DSurface**.

Regardless of whether the shape is degenerate, the area is defined to be:

```
(size().x() * size.y() + size().z() * size.y() + size().z() * size.x()) * 2
```

nearestPointSurface

```
virtual cc3DVect nearestPointSurface(
   const cc3DVect &pt) const;
```

This is an override from class **cc3DSurface**.

Parameters

pt	The point.

volume

```
virtual double volume() const;
```

This is an override from class **cc3DVolume**.

nearestPointVolume

```
virtual cc3DVect nearestPointVolume(
   const cc3DVect &pt) const;
```

This is an override from class **cc3DVolume**.

Parameters

pt	The point.

map

```
cc3DBox map(const cc3DXformRigid &xform) const;

void map(const cc3DXformRigid &xform, cc3DBox& dst) const;
```

- `cc3DBox map(const cc3DXformRigid &xform) const;`

Returns this shape mapped by the rigid transform xform.

Parameters

xform	The transform to map with.

- `void map(const cc3DXformRigid &xform, cc3DBox& dst) const;`

Maps this shape by the rigid transform *xform* and place the result in the supplied object.

Parameters

xform	The transform to map with.
dst	The object in which to place the result.

stateType

```
virtual cc3DShapeDefs::StateType stateType() const;

void stateType(cc3DShapeDefs::StateType type);
```

- `virtual cc3DShapeDefs::StateType stateType() const;`

Returns the state type of this object.

- `void stateType(cc3DShapeDefs::StateType type);`

Sets the state type of this aligned box. The state type influences how various methods (such as **nearestPoint()**) inherited from **cc3DShape** class are interpreted.

Parameters

type The state type. *type* must be one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface
cc3DShapeDefs::eVolume

Notes

The default shape type is *cc3DShapeDefs::eVolume*.

Throws

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface
cc3DShapeDefs::eVolume

getSizeAndShapeFromScaledUnit

```
void getSizeAndShapeFromScaledUnit (cc3DVect& size,
   cc3DXformRigid& shapeFromScaledUnit) const;
```

Gets the size and rigid transformation that maps from the scaled unit box.

Parameters

size A **cc3DVect** into which the size of the box is placed. The **cc3DVect** will contain the x-, y-, and z-dimensions of the box.

shapeFromScaledUnit
A **cc3DXformRigid** into which the rigid transformation giving the pose of the box is placed.

setSizeAndShapeFromScaledUnit

```
void setSizeAndShapeFromScaledUnit (const cc3DVect& size,
   const cc3DXformRigid& shapeFromScaledUnit);
```

Sets the size and rigid transformation that maps from the scaled unit box.

Parameters

size The size of the box (the **cc3DVect** contains the x-, y-, and z-dimensions of the box.

shapeFromScaledUnit
A rigid transformation giving the pose of the box.

Throws

cc3DShapeDefs::BadParams
Any member of *size* is less than 0.

getOriginVertexLengthVectorWidthVectorAndHeight

```
void getOriginVertexLengthVectorWidthVectorAndHeight (
   cc3DVect& vertex, cc3DVect& lengthVector,
   cc3DVect& widthVector, double& height);
```

Returns the origin vertex point, length vector, width vector, and height that define this **cc3DBox**. See **setOriginVertexLengthVectorWidthVectorAndHeight()** for a description of this parameterization.

Parameters

vertex The origin vertex.

lengthVector A **cc3DVect** giving the orientation and size of the length dimension of the box.

widthVector A **cc3DVect** giving the orientation and size of the width dimension of the box.

height The height of the box.

Throws

cc3DShapeDefs::DegenerateShape
This **cc3DBox** is degenerate.

setOriginVertexLengthVectorWidthVectorAndHeight

```
void setOriginVertexLengthVectorWidthVectorAndHeight (
   const cc3DVect& vertex, const cc3DVect& lengthVector,
   const cc3DVect& widthVector, double height);
```

Sets the origin vertex point, length vector, width vector, and height that define this **cc3DBox**. *lengthVector* and *widthVector* must be perpendicular to each other. If they are not, this function creates a new width vector using the following procedure:

1. Create a plane normal to the supplied *lengthVector*.
2. Project the supplied *widthVector* onto this plane.
3. Scale the projected vector so that it has the same length as the supplied *widthVector*. If the resulting vector has a length of zero (as would be the case if *lengthVector* and *widthVector* are parallel), an error is thrown.

The direction in which the supplied *height* is applied is determined by the cross product of *lengthVector* and *widthVector* (or the substitute width vector, if one is computed).

Parameters

vertex — The origin vertex of the box.

lengthVector — A **cc3DVect** giving the orientation and size of the length dimension of the box.

widthVector — A **cc3DVect** giving the orientation and size of the width dimension of the box.

height — The height of the box.

Throws

cc3DShapeDefs::BadParams
lengthVector, *widthVector*, or the computed width vector is **cc3DVect(0,0,0)** or *height* is less than 0.

Notes

Assuming that the internally generated width vector is named *widthVectorInternal* and the height vector is named as *heightVectorInternal*, then the center of the box will be

```
vertex + (lengthVector + widthVector +
heightVectorInternal.unit() * height)/2
```

Additionally, **shapeFromScaledUnit()** will map **cc3DVect (1,0,0)** to *lengthVector.unit()*, **cc3DVect(0,1,0)** to *widthVectorInternal.unit()*, **cc3DVect(0,0,1)** to *heightVectorInternal.unit(),* and **cc3DVect(0,0,0)** to *vertex*.

Also, **size().x()** will be *lengthVector.len()*, **size().y()** will be *widthVector.len()*, and **size().z()** will be *height*.

getCenterLengthVectorWidthVectorAndHeight

```
void getCenterLengthVectorWidthVectorAndHeight (
   cc3DVect& center, cc3DVect& lengthVector,
   cc3DVect& widthVector, double& height);
```

Returns the center point, length vector, width vector, and height that define this **cc3DBox**. See **setCenterLengthVectorWidthVectorAndHeight()** for a description of this parameterization.

Parameters

center	The center of the box.
lengthVector	A **cc3DVect** giving the orientation and size of the length dimension of the box.
widthVector	A **cc3DVect** giving the orientation and size of the width dimension of the box.
height	The height of the box.

Throws

cc3DShapeDefs::DegenerateShape
This **cc3DBox** is degenerate.

setCenterLengthVectorWidthVectorAndHeight

```
void setCenterLengthVectorWidthVectorAndHeight (
   const cc3DVect& center, const cc3DVect& lengthVector,
   const cc3DVect& widthVector, double height);
```

Sets the center point, length vector, width vector, and height that define this **cc3DBox**. *lengthVector* and *widthVector* must be perpendicular to each other. If they are not, this function creates a new width vector using the following procedure:

1. Create a plane normal to the supplied *lengthVector*.
2. Project the supplied *widthVector* onto this plane.
3. Scale the projected vector so that it has the same length as the supplied *widthVector*. If the resulting vector has a length of zero (as would be the case if *lengthVector* and *widthVector* are parallel), an error is thrown.

The direction in which the supplied *height* is applied is determined by the cross product of *lengthVector* and *widthVector* (or the substitute width vector, if one is computed).

Parameters

center	The center of the box.
lengthVector	A **cc3DVect** giving the orientation and size of the length dimension of the box.

widthVector A **cc3DVect** giving the orientation and size of the width dimension of the box.

height The height of the box.

Throws

cc3DShapeDefs::BadParams
lengthVector, *widthVector*, or the computed width vector is **cc3DVect(0,0,0)** or *height* is less than 0.

Notes

Assuming that the internally generated width vector is named *widthVectorInternal* and the height vector is named as *heightVectorInternal*, then the origin vertex of the box will be

```
center - (lengthVector + widthVector +
heightVectorInternal.unit() * height)/2
```

Additionally, **shapeFromScaledUnit()** will map **cc3DVect (1,0,0)** to *lengthVector.unit()*, **cc3DVect(0,1,0)** to *widthVectorInternal.unit()*, **cc3DVect(0,0,1)** to *heightVectorInternal.unit(),* and **cc3DVect(0,0,0)** to the origin vertex of the box.

Also, **size().x()** will be *lengthVector.len()*, **size().y()** will be *widthVector.len()*, and **size().z()** will be *height*.

size

```
cc3DVect size() const;

void size(const cc3DVect& newSize);
```

- `cc3DVect size() const;`

Returns the size of this **cc3DBox**, with the x-, y-, and z- components of the returned **cc3Vect** giving the length, width, and height of the box.

- `void size(const cc3DVect& newSize);`

Sets the size of this **cc3DBox**. The default size is **cc3Vect(0,0,0)**.

Parameters

newSize A **cc3Vect** giving the length, width, and height of the box.

Throws

cc3DShapeDefs::BadParams
Any element of *newSize* is less than 0.

Notes

The function does not change the origin vertex (which corresponds to cc3DVect(0,0,0) in the unit box).

setSizeAndKeepCenterUnchanged

```
void setSizeAndKeepCenterUnchanged(
   const cc3DVect& newSize);
```

Sets the size of this **cc3DBox** while preserving its center point. While the center is unchanged, the origin vertex (which corresponds to cc3DVect(0,0,0) in the unit box) may change. The rotation of the box does not change.

newSize A **cc3Vect** giving the length, width, and height of the box.

Throws

cc3DShapeDefs::BadParams
Any element of *newSize* is less than 0.

Notes

The setter might change the value of **shapeFromScaledUnit()**.

center

```
cc3DVect center() const;

void center(const cc3DVect& newCenter);
```

- `cc3DVect center() const;`

Returns the center of this **cc3DBox**.

- `void center(const cc3DVect& newCenter);`

Sets the center of this **cc3DBox**.

Parameters

newCenter The new center.

Notes

The setter might change the value of **shapeFromScaledUnit()**.

shapeFromScaledUnit

```
cc3DXformRigid shapeFromScaledUnit() const;

void shapeFromScaledUnit(const cc3DXformRigid& rigid);
```

- `cc3DXformRigid shapeFromScaledUnit() const;`

  Returns the **cc3DXformRigid** which maps this 3D box from the scaled unit box.

- `void shapeFromScaledUnit(const cc3DXformRigid& rigid);`

  Returns the **cc3DXformRigid** which maps this 3D box from the scaled unit box.

  Parameters

  rigid The transform that maps the scaled unit box to this **cc3DBox**.

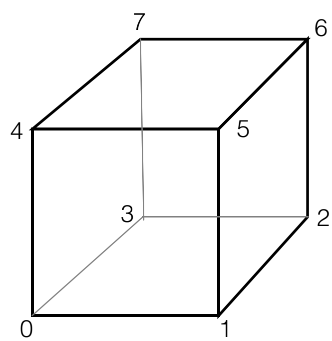
  Notes

  The default value is identity transform.

vertices

```
cmStd vector<cc3DVect> vertices() const;
```

Returns the vertices of this box. The vertices are returned in the following order:



where vertex 0 corresponds to the origin vertex in the unit shape. More formally, the vertex order is given as follows, based on an (untransformed) unit square:

```
0   cc3DVect(0,0,0)
1   cc3DVect(1,0,0)
2   cc3DVect(1,1,0)
3   cc3DVect(0,1,0)
4   cc3DVect(0,0,1)
5   cc3DVect(1,0,1)
6   cc3DVect(1,1,1)
7   cc3DVect(0,1,1)
```

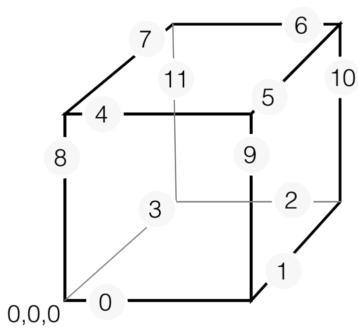
Notes

Some elements of the returned vector might be duplicate if this **cc3DBox** is degenerate.

lineSegs

```
cmStd vector<cc3DLineSeg> lineSegs() const;
```

Returns the line segments for the edges of this box. The segments are returned in the following order:



where the vertex marked *0,0,0* corresponds to the origin vertex in the unit shape. More formally, the edge order is given as follows, based on an (untransformed) unit square:

```
0   cc3DLineSeg(cc3DVect(0,0,0), cc3DVect(1,0,0))
1   cc3DLineSeg(cc3DVect(1,0,0), cc3DVect(1,1,0))
2   cc3DLineSeg(cc3DVect(1,1,0), cc3DVect(0,1,0))
3   cc3DLineSeg(cc3DVect(0,1,0), cc3DVect(0,0,0))
4   cc3DLineSeg(cc3DVect(0,0,1), cc3DVect(1,0,1))
5   cc3DLineSeg(cc3DVect(1,0,1), cc3DVect(1,1,1))
6   cc3DLineSeg(cc3DVect(1,1,1), cc3DVect(0,1,1))
7   cc3DLineSeg(cc3DVect(0,1,1), cc3DVect(0,0,1))
8   cc3DLineSeg(cc3DVect(0,0,0), cc3DVect(0,0,1))
9   cc3DLineSeg(cc3DVect(1,0,0), cc3DVect(1,0,1))
10  cc3DLineSeg(cc3DVect(1,1,0), cc3DVect(1,1,1))
11  cc3DLineSeg(cc3DVect(0,1,0), cc3DVect(0,1,1))
```

Notes

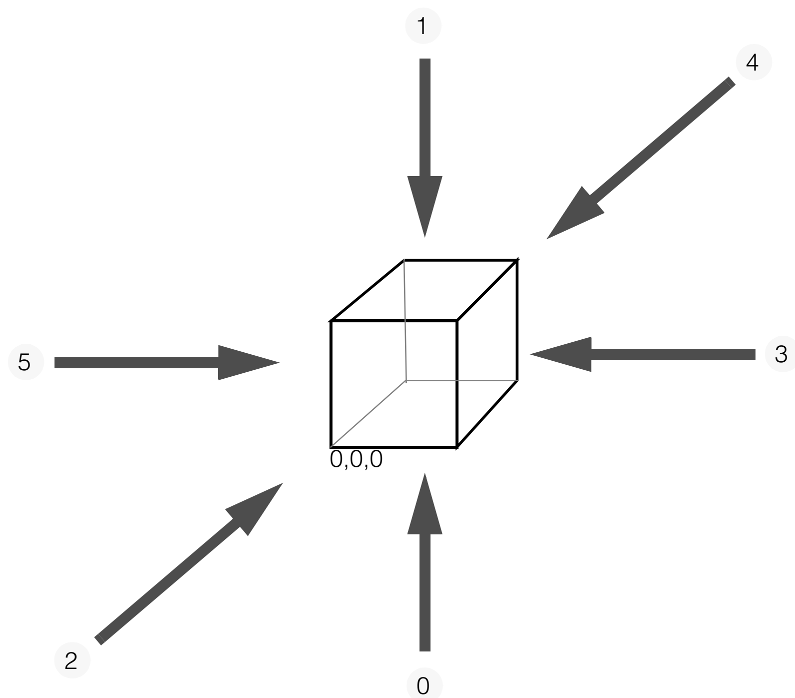
Some elements of the returned vector might be degenerate line segments and some elements might be duplicate if this **cc3DBox** itself is degenerate.

Each element of the returned vector has its state type set to *cc3DShapeDefs::eCurve*, regardless of the state type of the **cc3DBox**.

surfaces

```
cmStd vector<cc3DRect> surfaces() const;
```

Returns the surfaces of this box. The surfaces are returned in the following order:



where the vertex marked *0,0,0* corresponds to the origin vertex in the unit shape. More formally, the surface order is given as follows, based on an (untransformed) unit square (the vertex pairs define the opposite corners of each surface):

```
0   cc3DVect(0,0,0), cc3DVect(1,1,0)
1   cc3DVect(0,0,1), cc3DVect(1,1,1)
2   cc3DVect(0,0,0), cc3DVect(1,0,1)
3   cc3DVect(1,0,0), cc3DVect(1,1,1)
4   cc3DVect(0,1,0), cc3DVect(1,1,1)
5   cc3DVect(0,0,0), cc3DVect(0,1,1)
```

Notes

Some elements of the returned vector might be degenerate surfaces and some elements might be duplicate if this **cc3DBox** itself is degenerate.

Each element of the returned vector has the state type of *cc3DShapeDefs::eSurface* no matter the current state type of the box.

Operators

operator==

```
bool operator==(const cc3DBox& that) const;
```

Returns true if this object is exactly equal to *that*, and false otherwise.

Parameters

that The **cc3DBox** to compare to this one.

cc3DCameraCalib

```
#include <ch_c3d/ccalib3d.h>
```

```
class cc3DCameraCalib;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Main class for the 3D camera calibration tool.

3D camera calibration is a process that establishes a mathematical relationship between the 2D coordinate system associated with the pixels in an acquired image and a 3D coordinate system associated with the physical world in front of the camera. The following table lists the coordinate spaces associated with a 3D camera calibration.

Space	Description	Origin	Units	Handedness
Raw2D	Raw 2D image space defined by acquired pixels.	Upper-left corner of upper-left pixel in acquired image.	Pixels.	Left-handed. Positive-X extends to the right, positive-Y extends down.
Camera2D	Undistorted 2D space that removes effects of optical distortion and pixel aspect ratio.	0,0,1 in Camera3D space.	N/A	Left-handed. X- and Y-axes parallel to and in the same direction as Camera3D X- and Y-axes

Space	Description	Origin	Units	Handedness
Camera3D	Idealized 3D space with Z-axis corresponding to optical axis of camera.	Point of optical convergence within lens.	Physical units.	Right -handed. X- and Y-axis roughly parallel to Raw2D X- and Y-axis. Z-axis extends away from front of camera along optical axis.
Phys3D	3D physical space.	Defined by fiducial mark on calibration plate.	Physical units initially defined by calibration plate grid spacing.	Right handed. X- and Y-axis aligned to calibration grid, Z-axis normal to plate extending away from the camera.

For more information on 3D camera calibration, see chapters 1 and 4 of the *3D-Locate Developer's Guide*.

Constructors/Destructors

cc3DCameraCalib

```
cc3DCameraCalib();

cc3DCameraCalib(const cc2XformCalib2 &raw2DFromPhys3D,
   const ccPelRect &calibRoiRaw2D);

cc3DCameraCalib(
   const ccCalib2ParamsIntrinsic& raw2DFromCamera2D,
   const cc3DXformRigid& camera3DFromPhys3D,
   const ccPelRect &calibRoiRaw2D);

cc3DCameraCalib(const cc3DCameraCalib &rhs);
```

- `cc3DCameraCalib();`

  Return a default-constructed **cc3DCameraCalib**.

Notes

The default value for **camera3DFromPhys3D()** is an identity rotation matrix coupled with a translation of (0,0,1); **calibRoiRaw2D()** is initialized to an empty default-constructed **ccPelRect**.

- ```
cc3DCameraCalib(const cc2XformCalib2 &raw2DFromPhys3D,
   const ccPelRect &calibRoiRaw2D);
```

Construct a 3D camera calibration from a 2D image from 3D physical camera calibration transformation using the specified calibration region of interest.

**Parameters**

*raw2DFromPhys3D*
A transform from a calibrated 3D physical space to a 2D image space.

*calibRoiRaw2D* The calibration region of interest.

**Throws**

*cc3DCameraCalibDefs::InvalidCalibration*
*raw2DFromPhys3D* was computed from a single image.

*cc3DCameraCalibDefs::IncorrectCalibrationDirection*
*raw2DFromPhys3D* actually maps image space to physical 3D space.

*cc3DCameraCalibDefs::InvalidRegionOfInterest*
The width or height of *calibRoiRaw2D* is 0.

- ```
cc3DCameraCalib(
   const ccCalib2ParamsIntrinsic& raw2DFromCamera2D,
   const cc3DXformRigid& camera3DFromPhys3D,
   const ccPelRect &calibRoiRaw2D);
```

Construct a 3D camera calibration from the supplied camera intrinsics, camera 3D space from physical 3D space transformation, and calibration region of interest.

Parameters

raw2DFromCamera2D
A transform giving the camera intrinsics.

camera3DFromPhys3D
A transform giving the pose of camera 3D space in physical 3D space.

calibRoiRaw2D The calibration region of interest.

Throws

cc3DCameraCalibDefs::InvalidRegionOfInterest
The width or height of *calibRoiRaw2D* is 0.

- ```
  cc3DCameraCalib(const cc3DCameraCalib &rhs);
  ```

Copy constructor.

Parameters

rhs The source of the copy.

Public Member Functions

raw2DFromCamera2D

```
ccCalib2ParamsIntrinsic raw2DFromCamera2D() const;
```

Get the camera intrinsics. This transform corresponds to the mapping from camera2D space (the z=1 plane in front of the camera) to raw2D image space.

camera3DFromPhys3D

```
cc3DXformRigid camera3DFromPhys3D() const;
```

Get the camera extrinsics. This transform corresponds to the position of the camera with respect to phys3D space.

Notes

The default value for the camera3DFromPhys3D is an identity rotation matrix coupled with a translation of (0,0,1).

calibRoiRaw2D

```
ccPelRect calibRoiRaw2D() const;
```

Get the region of interest (the field of view) of the raw acquired image (the field of view of the camera).

pointRaw2DFromPointPhys3D

```
cc2Vect pointRaw2DFromPointPhys3D(
   const cc3Vect &pointPhys3D) const;
```

Compute the 2D image position from a given 3D physical position.

Parameters

pointPhys3D A point in physical 3D space.
```

Throws

cc3DShapeDefs::NoIntersection
The specified point is behind (or on the plane of) the camera

rayPhys3DFromPointRaw2D

```
cc3DRay rayPhys3DFromPointRaw2D(
   const cc2Vect &pointRaw2D) const;
```

Compute the 3D ray through the camera origin from the given 2D image position.

Parameters

pointRaw2D A point in raw 2D space.

cloneWithNewCamera3DFromPhys3D

```
cc3DCameraCalib cloneWithNewCamera3DFromPhys3D(
   const cc3DXformRigid &newCamera3DFromPhys3D) const;
```

Construct a new **cc3DCameraCalib** using the given camera3DFromPhys3D transform.

The newly constructed camera **cc3DCameraCalib** has the same camera intrinsics and region of interest rectangle as this one.

Parameters

newCamera3DFromPhys3D
The new physical space.

cloneComposeWithPhys3DFromAny3D

```
cc3DCameraCalib cloneComposeWithPhys3DFromAny3D(
   const cc3DXformRigid &phys3DFromAny3D) const;
```

Construct a new **cc3DCameraCalib** by composing the current **camera3DFromPhys3D()** with the specified phys3DFromAny3D transformation. This sets the new value of **camera3DFromPhys3D()** to the composition of its current value with the supplied transform.

The newly constructed camera **cc3DCameraCalib** has the same camera intrinsics and region of interest rectangle as this one.

Parameters

phys3DFromAny3D
The 3D transform to compose.

pointPhys2DFromRaw2D

```
cc2Vect pointPhys2DFromRaw2D(
   const cc2Vect& pointRaw2D) const;
```

Map a point in raw 2D (image) space into a 2D point on the xy plane in physical space.

Parameters

pointRaw2D

Throws

cc3DShapeDefs::NoIntersection
The specified point in raw 2D (image) space does not correspond to a 2D point in the xy plane in physical space (this can happen if the intersection of the line through the raw 2D point would intersect the xy plane in physical space behind the camera).

pointRaw2DFromPhys2D

```
cc2Vect pointRaw2DFromPhys2D(
   const cc2Vect& pointPhys2D) const;
```

Map a 2D point on the xy plane in physical space into a 2D point in raw 2D (image) space.

Parameters

pointPhys2D The point to map.

pointPhys3DFromPointRaw2D

```
void pointPhys3DFromPointRaw2D(const cc2Vect& pointRaw2D,
   const cc3DPlane& planePhys3D, cc3DVect& pointPhys3D,
   cc3DShapeDefs::IntersectionStatus& resultStatus) const;
```

Compute the 3D physical point associated with a 2D raw image point (the 3D physical point is computed by intersecting the ray corresponding to a 2D raw image point with the given plane).

Parameters

pointRaw2D The raw image point.

planePhys3D The 3D plane to which to constrain the 3D point

pointPhys3D A **cc3DVect** into which the 3D point is placed.

resultStatus Set to *cc3DShapeDefs::eIntersect* if a point is computed. If the specified plane is behind the camera, this is set to *cc3DShapeDefs::eNone*.

raw2DFromPhys2D

```
cc2XformBasePtrh_const raw2DFromPhys2D() const;
```

Return a new transform whose **mapPoint()** function will map a 2D point on the xy plane in physical space to a point in raw 2D (image) space.

phys2DFromRaw2D

```
cc2XformBasePtrh_const phys2DFromRaw2D() const;
```

Return a new transform whose **mapPoint()** function will map a point in raw 2D (image) space to a 2D point on the xy plane in physical space.

Operators

operator=

```
cc3DCameraCalib& operator=(const cc3DCameraCalib &rhs);
```

Assignment operator.

Parameters

rhs The source of the assignment.

operator==

```
bool operator==(const cc3DCameraCalib& that) const;
```

Return true if this **cc3DCameraCalib** is equal to the supplied object. False otherwise.

Parameters

that The **cc3DCameraCalib** to evaluate.

Notes

Two **cc3DCameraCalib** is objects are considered equal if and only if all their values are equal.

operator cc2XformCalib2

```
operator cc2XformCalib2() const;
```

Get the **cc2XformCalib2** corresponding to this 3D camera calibration

operator*

```
cc2Vect operator*(const cc3Vect &pointPhys3D) const;
```

Compute the 2D image position from a given 3D physical position

Parameters

The point to map.

Notes

This operator* overload is simply provided for convenience

Throws

cc3DShapeDefs::NoIntersection

The specified point is behind (or on the plane of) the camera.

cc3DCameraCalibCameraPlateResult

```
#include <ch_c3d/ccalib3d.h>
```

```
class cc3DCameraCalibCameraPlateResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class provides information, including error residuals, for a particular view of a calibration plate in an image. Multiple **cc3DCameraCalibCameraPlateResult** objects are generated during 3D camera calibration.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Returns true if this object contains computed data, false if it is a default-constructed object. Other methods of this class throw *cc3DCameraCalibDefs::NotComputed* **isComputed** is false.

calPlate3DFromCamera3D

```
cc3DXformRigid calPlate3DFromCamera3D() const;
```

Get the transform from the camera to the calibration plate.

Throws

cc3DCameraCalibDefs::NotComputed if this object is default-constructed.

residualsRaw2D

```
cc3DResiduals residualsRaw2D() const;
```

Get the residual statistics between the computed correspondence features and the actual correspondence features.

Notes

residualsRaw2D is measured in image pixels.

This residual is based on the actual features in one particular set of features (corresponding to one particular view and one particular camera), but the expected features are based on the calibration which was computed from all the features from all the views from all the cameras

Throws

cc3DCameraCalibDefs::NotComputed if this object is default-constructed.

residualsPhys3D

```
cc3DResiduals residualsPhys3D() const;
```

Get the residual statistics between the computed correspondence features and the actual correspondence features.

Notes

residualsPhys3D is measured in physical units.

This residual is based on the actual features in one particular set of features (corresponding to one particular view and one particular camera), but the expected features are based on the calibration which was computed from all the features from all the views from all the cameras

Throws

cc3DCameraCalibDefs::NotComputed if this object is default-constructed.

numCorrespondences

```
c_Int32 numCorrespondences() const;
```

Returns the number of correspondences found for this particular camera and this particular calibration plate pose.

Throws

cc3DCameraCalibDefs::NotComputed if this object is default-constructed.

featureCoverage

```
double featureCoverage() const;
```

Returns the proportion of the camera's field of view which was covered by the convex hull of the feature correspondences.

Throws

cc3DCameraCalibDefs::NotComputed if this object is default-constructed.

Operators

operator==

```
bool operator==(
   const cc3DCameraCalibCameraPlateResult& that) const;
```

Returns true if this **cc3DCameraCalibCameraPlateResult** has the same numerical values as supplied object; otherwise returns false,

Parameters

that — The **cc3DCameraCalibCameraPlateResult** to compare to this one.

■ cc3DCameraCalibCameraPlateResult

cc3DCameraCalibCameraResult

```
#include <ch_c3d/ccalib3d.h>
```

```
class cc3DCameraCalibCameraResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class provides information, including error residuals, for a particular camera.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Returns true if this object contains computed data, false if it is a default-constructed object. Other methods of this class throw *cc3DCameraCalibDefs::NotComputed* **isComputed** is false.

cameraPlateResults

```
const cmStd vector<cc3DCameraCalibCameraPlateResult>
   &cameraPlateResults() const;
```

Returns all of the **cc3DCameraCalibCameraPlateResult** objects for a camera. The returned vector is indexed by plate index. Each camera result includes multiple results corresponding to individual calibration plate poses

Throws

cc3DCameraCalibDefs::NotComputed if this object is default-constructed.

raw2DFromPhys3D

```
const cc3DCameraCalib &raw2DFromPhys3D() const;
```

Get this cameras computed calibration

Throws

cc3DCameraCalibDefs::NotComputed if this object is default-constructed.

featureCoverage

```
double featureCoverage() const;
```

Returns the proportion of the camera's field of view which was covered by the convex hull of the feature correspondences in all of the calibration views.

Throws

cc3DCameraCalibDefs::NotComputed if this object is default-constructed.

Operators

operator==

```
bool operator==(const cc3DCameraCalibCameraResult& that)
const;
```

Returns true if this **cc3DCameraCalibCameraResult** has the same numerical values as supplied object; otherwise returns false,

Parameters

that The **cc3DCameraCalibCameraResult** to compare to this one.

cc3DCameraCalibDefs

```
#include <ch_c3d/ccalib3d.h>
```

```
class cc3DCameraCalibDefs;
```

A name space that holds enumerations and constants used with 3D camera calibration.

Enumerations

PoseType

```
enum PoseType
```

This enumeration defines the calibration plate pose types. You must specify the pose type for each image of a calibration plate or set of plate correspondence pairs that you supply to the calibration system.

Value	Meaning
ePoseDefineWorldCoord = 0	The calibration plate pose defines the world coordinate system. The plate origin, as defined by the fiducial mark, defines the origin of world coordinate space.
ePoseElevated = 1	The calibration plate pose is precisely parallel to the pose that defined the world coordinate system (and the z-axis translation of the offset between the two calibration plate poses will be specified). The inter-plate spacing must be given in the same units used to specify the plate grid pitch. Note that the (x,y) translation and rotation in the plane are unconstrained.
ePoseTilted = 2	The calibration plate is arbitrarily positioned.
kDefaultPoseType	The default pose type (*ePoseDefineWorldCoord).*

cc3DCameraCalibFeatures

```
#include <ch_c3d/ccalib3d.h>
```

```
class cc3DCameraCalibFeatures;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Class that includes information about all of the calibration plate features from multiple cameras viewing a single plate pose.

Note You can specify the features using a vector of **cc3DPointSet2D3D** or a vector of **ccCrspPairWeightedVector**.

Constructors/Destructors

cc3DCameraCalibFeatures

```
cc3DCameraCalibFeatures(
   const cmStd vector<ccCrspPairWeightedVector> &Features =
   cmStd vector<ccCrspPairWeightedVector>(),
   const cmStd vector<ccPelRect> &CalibRoiRaw2Ds =
   cmStd vector<ccPelRect>(),
   cc3DCameraCalibDefs::PoseType pose_type =
   cc3DCameraCalibDefs::kDefaultPoseType,
   double ZPosition = 0.);

cc3DCameraCalibFeatures(
   const cmStd vector<cc3DPointSet2D3D>
   &FeaturesRaw2DPhys3D,
   const cmStd vector<ccPelRect> &CalibRoiRaw2Ds =
   cmStd vector<ccPelRect>(),
   cc3DCameraCalibDefs::PoseType pose_type =
   cc3DCameraCalibDefs::kDefaultPoseType,
   double ZPosition = 0.);
```

- ```
  cc3DCameraCalibFeatures(
     const cmStd vector<ccCrspPairWeightedVector> &Features =
     cmStd vector<ccCrspPairWeightedVector>(),
     const cmStd vector<ccPelRect> &CalibRoiRaw2Ds =
     cmStd vector<ccPelRect>(),
  ```
  ```

```
cc3DCameraCalibDefs::PoseType pose_type =
cc3DCameraCalibDefs::kDefaultPoseType,
double ZPosition = 0.);
```

Constructs a **cc3DCameraCalibFeatures** using the supplied vector of **ccCrspPairWeightedVector** objects that characterize the vertex location correspondence pairs and weights from a set of cameras viewing a given plate pose.

Notes

**cf3DCalibrateCameras(**) will throw *cc3DCameraCalibDefs::InvalidRegionOfInterest* if any of the calibRoiRaw2Ds pelRects are empty.

Parameters

features — A vector of **ccCrspPairWeightedVector** objects, where each element of the vector includes the vertex locations and weights from a single camera.

CalibRoiRaw2Ds — The regions of interest for the cameras. All the features in *features* must lie within these regions.

pose_type — The pose type of this calibration plate view. *pose_type* must be one of the following values:

cc3DCameraCalibDefs::ePoseDefineWorldCoord
cc3DCameraCalibDefs::ePoseElevated
cc3DCameraCalibDefs::ePoseTilted

ZPosition — If *pose_type* is *cc3DCameraCalibDefs::ePoseElevated*, then *ZPosition* must specify the offset in the Z-axis of the current plate pose from the plate that was specified for the world origin pose (*plate_type* of *cc3DCameraCalibDefs::ePoseDefineWorldCoord*). *ZPosition* must be a precise value, and it must be specified in the same units that are used to specify the grid pitch of the calibration plate.

Throws

cc3DCameraCalibDefs::BadParams
The size of *features* is not the same as the size of *CalibRoiRaw2Ds* or at least one feature does not fit within its corresponding *CalibRoiRaw2Ds* window.

•
```
cc3DCameraCalibFeatures(
   const cmStd vector<cc3DPointSet2D3D>
   &FeaturesRaw2DPhys3D,
   const cmStd vector<ccPelRect> &CalibRoiRaw2Ds =
   cmStd vector<ccPelRect>(),
```

```
cc3DCameraCalibDefs::PoseType pose_type =
cc3DCameraCalibDefs::kDefaultPoseType,
double ZPosition = 0.);
```

Constructs a **cc3DCameraCalibFeatures** using the supplied vector of **cc3DPointSet2D3D** objects that characterize the vertex location correspondence pairs from a set of cameras viewing a given plate pose.

Notes

**cf3DCalibrateCameras(**) will throw *cc3DCameraCalibDefs::InvalidRegionOfInterest* if any of the calibRoiRaw2Ds pelRects are empty.

Parameters

features — A vector of **cc3DPointSet2D3D** objects, where each element of the vector contains a 2D calibration plate vertex location in 2D image space and the corresponding physical location of the same vertex in 3D physical space.

CalibRoiRaw2Ds — The regions of interest for the cameras. All the features in *features* must lie within these regions.

pose_type — The pose type of this calibration plate view. *pose_type* must be one of the following values:

cc3DCameraCalibDefs::ePoseDefineWorldCoord
cc3DCameraCalibDefs::ePoseElevated
cc3DCameraCalibDefs::ePoseTilted

ZPosition — If *pose_type* is *cc3DCameraCalibDefs::ePoseElevated*, then *ZPosition* must specify the offset in the Z-axis of the current plate pose from the plate that was specified for the world origin pose (*plate_type* of *cc3DCameraCalibDefs::ePoseDefineWorldCoord*). *ZPosition* must be a precise value, and it must be specified in the same units that are used to specify the grid pitch of the calibration plate.

Throws

cc3DCameraCalibDefs::BadParams
The size of *features* is not the same as the size of *CalibRoiRaw2Ds* or at least one feature does not fit within its corresponding *CalibRoiRaw2Ds* window.

ccCalibCorrespondenceDefs::NegativeWeight
At least one of element of *features* has a negative weight.

Public Member Functions

features

```
const cmStd vector<ccCrspPairWeightedVector> &features()
  const;

void features(
  const cmStd vector<ccCrspPairWeightedVector> &features);
```

- ```
  const cmStd vector<ccCrspPairWeightedVector> &features()
    const;
  ```

Returns a vector of **ccCrspPairWeightedVector** that contains the feature pairs for this object. If this object was constructed with **cc3DPointSet2D3D** features, or if the **featuresRaw2DPhys3D()** setter was the last setter called, then this function will construct and compute an appropriate **ccCrspPairWeightedVector** object.

- ```
  void features(
    const cmStd vector<ccCrspPairWeightedVector> &features);
  ```

Sets this object's features to the supplied vector of **ccCrspPairWeightedVector** objects. The supplied vector should contain one element per camera.

Parameters

features A vector of **ccCrspPairWeightedVector** objects, where each element of the vector includes the vertex locations and weights from a single camera.

Notes

The features are indexed by the camera indices

It is acceptable for some of the **ccCrspPairWeightedVectors** to have size 0 (i.e., it is acceptable for some camera plate images to have no features)

The *features* are ordered as <image,physical>

The weights are taken into account by the **cf3DCalibratedCameras()** function when it determines the parameters which minimized the weighted sum squared image error.

featuresRaw2DPhys3D

```
cmStd vector<cc3DPointSet2D3D> featuresRaw2DPhys3D()
   const;

void featuresRaw2DPhys3D (
   const cmStd vector<cc3DPointSet2D3D>
   &featuresRaw2DPhys3D);
```

- ```
  cmStd vector<cc3DPointSet2D3D> featuresRaw2DPhys3D()
     const;
  ```

Returns a vector of **cc3DPointSet2D3D** that contains the feature pairs for this object. If this object was constructed with **ccCrspPairWeightedVector** features, or if the **features()** setter was the last setter called, then this function constructs and computes an appropriate **cc3DPointSet2D3D** object.

- ```
  void featuresRaw2DPhys3D (
     const cmStd vector<cc3DPointSet2D3D>
     &featuresRaw2DPhys3D);
  ```

Sets this object's features to the supplied vector of **cc3DPointSet2D3D** objects. The supplied vector should contain one element per camera.

Parameters

features A vector of **cc3DPointSet2D3D** objects, where each element of the vector includes the vertex locations and weights from a single camera.

Notes

It is acceptable for some of the vectors of *featuresRaw2DPhys3D* to have size 0 (i.e., it is acceptable for some camera plate images to have no features).

Throws

ccCalibCorrespondenceDefs::NegativeWeight
The weight of at least one element of *features* is negative.

calibRoiRaw2Ds

```
const cmStd vector<ccPelRect> &calibRoiRaw2Ds() const;

void calibRoiRaw2Ds(
   const cmStd vector<ccPelRect> &calibRoiRaw2Ds);
```

- ```
  const cmStd vector<ccPelRect> &calibRoiRaw2Ds() const;
  ```

Returns a vector of the regions of interest (fields of view) in the raw acquired images.
  ```

- `void calibRoiRaw2Ds(`
  `const cmStd vector<ccPelRect> &calibRoiRaw2Ds);`

Sets the regions of interest (fields of view) in the raw acquired images.

Parameters

calibRoiRaw2Ds
A vector of rectangles, indexed by camera.

Notes

The calibration function will throw an error if any of the features lie outside the corresponding specified windows

Throws

cc3DCameraCalibDefs::InvalidRegionOfInterest
One or more of the supplied rectangles are empty.

poseType

```
cc3DCameraCalibDefs::PoseType poseType() const;

void poseType(cc3DCameraCalibDefs::PoseType poseType);
```

- `cc3DCameraCalibDefs::PoseType poseType() const;`

Returns the pose type for the calibration plate pose of this object. The returned value is one of the following:

cc3DCameraCalibDefs::ePoseDefineWorldCoord
cc3DCameraCalibDefs::ePoseElevated
cc3DCameraCalibDefs::ePoseTilted

- `void poseType(cc3DCameraCalibDefs::PoseType poseType);`

Sets the pose type for the calibration plate pose of this object. See the description of *PoseType* on page 60 for a description of the pose types.

Parameters

poseType The pose type. *poseType* must be one of the following values:

cc3DCameraCalibDefs::ePoseDefineWorldCoord
cc3DCameraCalibDefs::ePoseElevated
cc3DCameraCalibDefs::ePoseTilted

Throws

cc3DCameraCalibDefs::BadParams
poseType is not a member of **cc3DCameraCalibDefs::PoseType**.

zPosition

```
double zPosition() const;

void zPosition(double zPosition);
```

- `double zPosition() const;`

Returns the Z-offset of this plate pose from the plate for which **poseType()** was set to *cc3DCameraCalibDefs::ePoseDefineWorldCoord.*

- `void zPosition(double zPosition);`

Sets the Z-offset of this plate pose from the plate for which **poseType()** was set to *cc3DCameraCalibDefs::ePoseDefineWorldCoord.* The **zPosition()** is only valid if the **platePose()** for this plate is *cc3DCameraCalibDefs::ePoseElevated.* The value must be specified in the same units used to provide the calibration plate grid pitch.

Parameters

zPosition The Z-offset of this plate from the plate that defined the 3D world coordinate system origin.

Notes

If **platePose()** for this plate is other than *cc3DCameraCalibDefs::ePoseElevated,* this value is ignored.

If **platePose()** for this plate is *cc3DCameraCalibDefs::ePoseDefineWorldCoord,* then *zPosition* must be 0.0; otherwise, **cf3DCalibrateCameras()** will throw a *cc3DCameraCalibDefs::BadParams* exception.

The z position corresponds to the z-axis value in the convention that the calibration plate uses a right handed coordinate system. The Cognex Checkerboard calibration plate induces a z-axis behind the plate - therefore positive z positions correspond to plate poses which are further from the camera than the world coordinate space pose.

checkConsistencyBetweenFeaturesAndCalibRoiRaw2Ds

```
void checkConsistencyBetweenFeaturesAndCalibRoiRaw2Ds()
   const;
```

Calling this function verifies that the number of feature sets and regions of interest is the same, and it verifies that no plate vertex positions lie outside of the associated region of interest.

Throws

cc3DCameraCalibDefs::BadParams
One of the tests noted above failed.

Operators

operator==

```
bool operator==(
   const cc3DCameraCalibFeatures& that) const;
```

Returns true if this **cc3DCameraCalibFeatures** has the same numerical values and enumeration values as the supplied object.

Parameters

that The object to evaluate.

cc3DCameraCalibParams

```
#include <ch_c3d/ccalib3d.h>

class cc3DCameraCalibParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Parameters for 3D camera calibration.

Constructors/Destructors

cc3DCameraCalibParams

```
cc3DCameraCalibParams(
  cc2XformCalib2Defs::ccCalib2DistortionModel
DistortionModel);
```

Constructs a **cc3DCameraCalibParams** using the supplied value.

Parameters

DistortionModel The distortion model for this calibration parameters object. The supplied value must be one of the following values:

cc2XformCalib2Defs::eSineTanLawProjection
cc2XformCalib2Defs::e3ParamRadial

The default is *cc2XformCalib2Defs::e3ParamRadial.* For information on selecting a distortion model, see the file *ch_cvl/ccalib.h*.

Public Member Functions

distortionModel

```
cc2XformCalib2Defs::ccCalib2DistortionModel
   distortionModel() const;

void distortionModel (
   cc2XformCalib2Defs::ccCalib2DistortionModel
   distortionModel);
```

- ```
  cc2XformCalib2Defs::ccCalib2DistortionModel
     distortionModel() const;
  ```

Returns the distortion model for this object. The returned value is one of the following:

cc2XformCalib2Defs::eSineTanLawProjection
cc2XformCalib2Defs::e3ParamRadial

- ```
  void distortionModel (
     cc2XformCalib2Defs::ccCalib2DistortionModel
     distortionModel);
  ```

Sets the distortion model for this object.

Parameters

distortionModel The distortion model to use. *distortionModel* must be one of the following values:

cc2XformCalib2Defs::eSineTanLawProjection
cc2XformCalib2Defs::e3ParamRadial

Throws

cc3DCameraCalibDefs::NotImplemented
distortionModel is not *cc2XformCalib2Defs::e3ParamRadial* or *cc2XformCalib2Defs::eSineTanLawProjection*.

callback

```
const ccProgressCallback& callback() const;

void callback(const ccProgressCallback& c);
```

- ```
  const ccProgressCallback& callback() const;
  ```

Returns the progress callback function. This function is called periodically during calibration.
  ```

- `void callback(const ccProgressCallback& c);`

Sets the progress callback function that will be called at various points during the camera calibration to indicate the amount of progress made. The intended use of this functionality is to allow the implementation of a progress indicator in a user interface.

The callback function is called with a progress value in the range of 0.0 through 1.0, with a value of 0.0 when the calibration starts and 1.0 when the calibration is nearly complete.

Parameters

c The callback function object.

Notes

The callback object is not serialized. For more information on **ccProgressCallback**, see the file *ch_cog/progress.h*.

Operators

operator== `bool operator==(const cc3DCameraCalibParams& that) const;`

Returns true if this **cc3DCameraCalibParams** has the same numerical values and enumeration values as the supplied object.

Parameters

that The object to evaluate.

■ cc3DCameraCalibParams

cc3DCameraCalibResult

```
#include <ch_c3d/ccalib3d.h>

class cc3DCameraCalibResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Results of 3D camera calibration. This object includes individual result objects for each camera and each plate pose. The results are indexed by camera indexed or by plate pose index.

Constructors/Destructors

cc3DCameraCalibResult

```
cc3DCameraCalibResult();
```

Compiler-generated default constructor.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Returns false if this object was default-constructed, true if it was computed.

Other members of this class will throw *cc3DCameraCalibDefs::NotComputed* if **isComputed()** is false.

raw2DFromPhys3Ds

```
const cmStd vector<cc3DCameraCalib> &raw2DFromPhys3Ds()
   const;
```

Returns the **cc3DCameraCalib** calibration objects computed by the tool.

Notes

The vector is indexed by camera index.

Throws

cc3DCameraCalibDefs::NotComputed
This object was default-constructed.

■ cc3DCameraCalibResult

cameraResults

```
const cmStd
  vector<cc3DCameraCalibCameraResult> &cameraResults()
  const;
```

Get the individual camera calibration results computed by the 3D camera calibration tool.

The vector is indexed by camera index. Each **cc3DCameraCalibCameraResult** includes information about multiple plate poses.

Throws

cc3DCameraCalibDefs::NotComputed
This object was default-constructed.

calPlateWorldCoord3DFromCalPlate3Ds

```
const cmStd
vector<cc3DXformRigid>
   &calPlateWorldCoord3DFromCalPlate3Ds() const;
```

Get the calibration plate poses computed by the 3D camera calibration algorithm. The poses are computed with respect to the calibration plate pose which defines the world coordinates.

The vector is indexed by pose index.

Throws

cc3DCameraCalibDefs::NotComputed
This object was default-constructed.

residualsRaw2D

```
cc3DResiduals residualsRaw2D(c_Int32 cameraIndex,
   c_Int32 plateIndex) const;

cc3DResiduals residualsRaw2D() const;
```

```
cc3DResiduals residualsRaw2D(c_Int32 cameraIndex,
   c_Int32 plateIndex) const;
```

Return the residual statistics for the specified camera and plate pose.

Parameters

cameraIndex The camera index. This is the order in which

plateIndex The plate index.

Notes

residualsRaw2D() is measured in image pels

This residual is based on the actual features in one particular set of features (corresponding to one particular view and one particular camera), but the expected features are based on the calibration which was computed from all the features from all the views from all the cameras

Throws

cc3DCameraCalibDefs::NotComputed
This object was default-constructed.

cc3DCameraCalibDefs::BadParams if cameraIndex is an invalid index (< 0 or >= number of cameras) or plateIndex is an invalid index (< 0 or >= number of plate poses)

- `cc3DResiduals residualsRaw2D() const;`

Return the residual statistics over all cameras over all poses

Notes

residualsRaw2D() is measured in image pels

This residual is based on the actual features in one particular set of features (corresponding to one particular view and one particular camera), but the expected features are based on the calibration which was computed from all the features from all the views from all the cameras

The **cf3DCalibratedCameras()** function minimizes the sum squared error in image coordinates. Therefore, if different cameras have different pixel resolutions, then the calibration may be dominated by the higher resolution cameras. Consequently, this function is most useful when all cameras have similar pixel resolution.

Throws

cc3DCameraCalibDefs::NotComputed
This object was default-constructed.

residualsPhys3D

```
cc3DResiduals residualsPhys3D(c_Int32 cameraIndex,
   c_Int32 plateIndex) const;

cc3DResiduals residualsPhys3D() const;
```

- ```
  cc3DResiduals residualsPhys3D(c_Int32 cameraIndex,
     c_Int32 plateIndex) const;
  ```

Return the residual statistics corresponding to the specified camera and the specified plate pose

Parameters

cameraIndex

plateIndex

Notes

residualsPhys3D() is measured in physical units.

This residual is based on the actual features in one particular set of features (corresponding to one particular view and one particular camera), but the expected features are based on the calibration which was computed from all the features from all the views from all the cameras

Throws

cc3DCameraCalibDefs::NotComputed
This object was default-constructed.

cc3DCameraCalibDefs::BadParams if cameraIndex is an invalid index (< 0 or >= number of cameras) or plateIndex is an invalid index (< 0 or >= number of plate poses)

- ```
  cc3DResiduals residualsPhys3D() const;
  ```

Return the residual statistics over all cameras over all poses

Notes

residualsPhys3D() is measured in physical units.

This residual is based on the actual features in one particular set of features (corresponding to one particular view and one particular camera), but the expected features are based on the calibration which was computed from all the features from all the views from all the cameras

The **cf3DCalibratedCameras()** function minimizes the sum squared error in image coordinates. Therefore, if different cameras have different pixel resolutions, then the calibration may be dominated by the higher resolution cameras. Consequently, this function is most useful when all cameras have similar pixel resolution.

Throws

cc3DCameraCalibDefs::NotComputed

This object was default-constructed.

camera3DFromPhys3D

```
cc3DXformRigid camera3DFromPhys3D(c_Int32 cameraIndex)
const;
```

Return the transform between the specified camera and the physical coordinate system. The physical coordinate system corresponds to the view having pose type *cc3DCameraCalibDefs::ePoseDefineWorldCoord*.

Parameters

cameraIndex

Throws

cc3DCameraCalibDefs::BadParams

cameraIndex is an invalid index (< 0 or >= number of cameras)

Throws

cc3DCameraCalibDefs::NotComputed

This object was default-constructed.

maximumTilt

```
ccRadian maximumTilt() const;
```

Return the maximum tilt of the calibration plate over all the measured poses.

Throws

cc3DCameraCalibDefs::NotComputed

This object was default-constructed.

Operators

operator==

```
bool operator==(const cc3DCameraCalibResult& that) const;
```

True if this **cc3DCameraCalibResult** has the same numerical values as the supplied object, false otherwise

Parameters

that The object to compare to this one.

cc3DCircle

```
#include <ch_c3d/shapes3d.h>

class cc3DCircle:
   public cc3DShape,
   public cc3DCurve,
   public cc3DSurface;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class represents a circle with an arbitrary orientation in 3D space.

Constructors/Destructors

cc3DCircle

```
cc3DCircle();

cc3DCircle(double radius,
  const cc3DXformRigid& shapeFromScaledUnit,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eSurface);

cc3DCircle(double radius, const cc3DVect& center,
  const cc3DVect& normal,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eSurface);
```

•

```
cc3DCircle();
```

Default constructor, and constructs a degenerate circle on the XY plane and whose center is at the origin.

Default values are as follows:

- **radius()** is 0
- **shapeFromScaledUnit()** is the identity transformation
- **stateType()** is *cc3DShapeDefs::eSurface*.

- ```
cc3DCircle(double radius,
  const cc3DXformRigid& shapeFromScaledUnit,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eSurface);
```

Constructs a 3D circle defined by a radius, a transform from the scaled unit circle, and state type.

**Parameters**

*radius* The radius of the circle

*shapeFromScaledUnit*
A transformation that maps the scaled unit circle to this circle.

*type* The initial state type for this circle. You must supply one of the following values for this parameter:

*cc3DShapeDefs::eCurve*
*cc3DShapeDefs::eSurface*

**Throws**

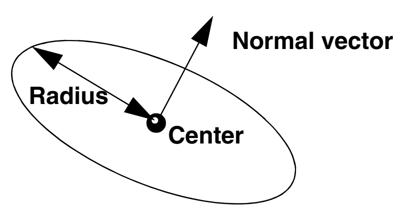
*cc3DShapeDefs::BadParams*
*radius* is less than 0.

*cc3DShapeDefs::InvalidStateType*
*type* is not one of the following values:

*cc3DShapeDefs::eCurve*
*cc3DShapeDefs::eSurface*

- ```
cc3DCircle(double radius, const cc3DVect& center,
  const cc3DVect& normal,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eSurface);
```

Constructs a 3D circle defined by a radius, center point, and a vector giving the circle's normal direction.



Calling **shapeFromScaledUnit()** on a **cc3DCircle** constructed using this constructor returns an internally computed **cc3DXformRigid .**

Parameters

radius — The radius.

center — The center of the circle.

normal — A **cc3DVect** giving the orientation of the circle (the circle lies in a plane normal to the supplied vector). If *normal* is not a unit vector, the tool normalizes it. Subsequent calls to **normal()** may not return the same vector used for construction.

type — The initial state type for this circle. You must supply one of the following values for this parameter:

cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface

Throws

cc3DShapeDefs::BadParams
radius is less than 0.

cc3DShapeDefs::InvalidDirection
normal is **cc3DVect(0,0,0)**.

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface

Public Member Functions

clone

```
virtual cc3DShapePtrh clone () const;
```

This is an override from class **cc3DShape**.

isFinite

```
virtual bool isFinite () const;
```

This is an override from class **cc3DShape**.

isEmpty

```
virtual bool isEmpty () const;
```

This is an override from class **cc3DShape**.

nearestPoint

```
virtual cc3DVect nearestPoint (const cc3DVect &pt) const;
```

This is an override from class **cc3DShape**.

Parameters

pt — The point to which to determine the nearest point on this **cc3DCircle**.

boundingBox

```
virtual cc3DAlignedBox boundingBox() const;
```

This is an override from class **cc3DShape**.

mapShape

```
virtual cc3DShapePtrh mapShape (const cc3DXformBase &xform)
   const;

virtual void mapShape (const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const;
```

- ```
  virtual cc3DShapePtrh mapShape (const cc3DXformBase &xform)
     const;
  ```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters

xform — The transform with which to map.

- ```
  virtual void mapShape (const cc3DXformBase& xform,
     cc3DShapePtrh& dst) const;
  ```

This is an override from class **cc3DShape**.

Parameters

xform — The transform with which to map.

dst — The transformed shape.

perimeter

```
virtual double perimeter() const;
```

This is an override from class **cc3DCurve**.

nearestPointCurve

```
virtual cc3DVect nearestPointCurve(const cc3DVect &pt)
const;
```

This is an override from class **cc3DCurve**.

Parameters

pt The point.

area

```
virtual double area() const;
```

This is an override from class **cc3DSurface**.

nearestPointSurface

```
virtual cc3DVect nearestPointSurface(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DSurface**.

Parameters

pt The point.

stateType

```
virtual cc3DShapeDefs::StateType stateType() const;

void stateType(cc3DShapeDefs::StateType type);
```

- `virtual cc3DShapeDefs::StateType stateType() const;`

Returns the state type of this **cc3DCircle**.

- `void stateType(cc3DShapeDefs::StateType type);`

Sets the state type of this **cc3DCircle**. The state type influences how various methods (such as **nearestPoint()**) inherited from **cc3DShape** class are interpreted.

Parameters

type The state type. *type* must be one of the following values:

cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface

Notes

The default shape type is *cc3DShapeDefs::eSurface*.

Throws

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface

map

```
cc3DCircle map(const cc3DXformRigid &xform) const;

void map(const cc3DXformRigid &xform, cc3DCircle& dst)
   const;
```

- `cc3DCircle map(const cc3DXformRigid &xform) const;`

Returns the result of mapping this **cc3DCircle** by the supplied rigid transformation.

Parameters

xform The transformation.

- `void map(const cc3DXformRigid &xform, cc3DCircle& dst) const;`

Places the result of mapping this **cc3DCircle** by the supplied rigid transformation in the supplied **cc3DCircle** instance.

Parameters

xform The transformation.

dst The **cc3DCircle** in which to place the result.

center

```
cc3DVect center() const;

void center(const cc3DVect& newCenter);
```

- `cc3DVect center() const;`

Returns the center of this **cc3DCircle.**

- `void center(const cc3DVect& newCenter);`

Sets the center of this **cc3DCircle.**

Calling this function may change the value returned by **shapeFromScaledUnit()**.

Parameters
newCenter The center.

normal

```
cc3DVect normal() const;

void normal(const cc3DVect& newNormal);
```

- `cc3DVect normal() const;`

Returns a **cc3DVect** that is normal to the plain containing this circle. The returned vector is a unit vector.

- `void normal(const cc3DVect& newNormal);`

Sets the orientation of this circle such that a plain containing the circle is normal to the supplied **cc3DVect**.

If the supplied vector is not a unit vector, this function normalizes it before storing it; the getter may not return the same **cc3DVect** as was passed to the setter.

Parameters
newNormal The normal vector to set.

Throws
cc3DShapeDefs::InvalidDirection
newNormal is **cc3DVect(0,0,0)**.

getRadiusAndShapeFromScaledUnit

```
void getRadiusAndShapeFromScaledUnit (
   double& r, cc3DXformRigid& shapeFromScaledUnit) const;
```

Returns the radius and rigid transform that maps this **cc3DCircle** from the scaled unit circle.

Parameters
r The returned radius.

shapeFromScaledUnit
The returned transformation.

setRadiusAndShapeFromScaledUnit

```
void setRadiusAndShapeFromScaledUnit (double r,
   const cc3DXformRigid& shapeFromScaledUnit);
```

Sets the radius and rigid transform that maps this **cc3DCircle** from the scaled unit circle.

Parameters

r — The radius.

shapeFromScaledUnit — The transformation.

Throws

cc3DShapeDefs::BadParams — *r* is less than 0.

getRadiusCenterAndNormalDirection

```
void getRadiusCenterAndNormalDirection (
   double& r, cc3DVect& center, cc3DVect& normal);
```

Returns the radius, center point, and normal vector that defines this **cc3DCircle**. The returned vector is a unit vector that is normal to the plane containing this shape.

Parameters

r — The returned radius.

center — The returned center point.

normal — The returned normal vector. The returned vector is normal to a plane containing the circle.

setRadiusCenterAndNormalDirection

```
void setRadiusCenterAndNormalDirection (double r,
   const cc3DVect& center, const cc3DVect& normal);
```

Sets this **cc3DCircle** to the supplied radius, center point, and normal vector.

The supplied normal vector is normalized to the unit vector internally; a subsequent call to **getRadiusCenterAndNormalDirection()** returns the normalized vector, not necessarily the one supplied to this function.

Parameters

r — The radius.

center — The center point.

normal — The normal vector. The circle is oriented so that it is contained within a plane that is normal to *normal*.

Throws

cc3DShapeDefs::BadParams — *r* is less than 0.

cc3DShapeDefs::InvalidDirection
normal is **cc3DVect(0,0,0)**.

radius

```
double radius() const;

void radius(double r);
```

- `double radius() const;`

Returns the radius of this **cc3DCircle**.

- `void radius(double r);`

Sets the radius of this **cc3DCircle**. The origin and orientation are unchanged.

The default value is 0.

Parameters
r The radius.

Throws
cc3DShapeDefs::BadParams
r is less than 0.

shapeFromScaledUnit

```
cc3DXformRigid shapeFromScaledUnit() const;

void shapeFromScaledUnit(const cc3DXformRigid& rigid);
```

- `cc3DXformRigid shapeFromScaledUnit() const;`

Returns a **cc3DXformRigid** that maps the unit circle to this **cc3DCircle**.

Notes
The returned transform is non-unique. Calling any of the setters in this class may cause an arbitrarily different, and correct, transformation to be returned.

- `void shapeFromScaledUnit(const cc3DXformRigid& rigid);`

Sets this **cc3DCircle** to the supplied **cc3DXformRigid** that maps the unit circle to the specified **cc3DCircle**.

Parameters
rigid The transformation.

Operators

operator==

```
bool operator==(const cc3DCircle& that) const;
```

Returns true if this **cc3DCircle** is exactly equal to the supplied object, and false otherwise.

Parameters

that	The object to compare to this one.

cc3DCircleFit2DDefs

```
#include <ch_c3d/fit2d.h>
```

```
class cc3DCircleFit2DDefs;
```

A name space that holds enumerations and constants used with the 3D circle pose estimation tool.

Enumerations

FitMode

```
enum FitMode;
```

This enumeration defines the fitting options for the global **cf3DFitCircle3DUsingPoints2D**() function.

Value	Meaning
eLeastSquaresComputeRadius = 0	Optimize least-squares fit unconstrained by expected radius.
eLeastSquaresUseSpecifiedRadius = 1	Optimize least-square fit for the given radius.
kDefaultFitMode	The default fit mode (*eLeastSquaresUseSpecifiedRadius*).

■ cc3DCircleFit2DDefs

cc3DCircleFit2DParams

```
#include <ch_c3d/fit2d.h>

class cc3DCircleFit2DParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Parameters class for the 3D circle pose estimation tool.

Constructors/Destructors

cc3DCircleFit2DParams

```
cc3DCircleFit2DParams(double Radius = 0.,
   enum cc3DCircleFit2DDefs::FitMode FitMode =
   cc3DCircleFit2DDefs::eLeastSquaresUseSpecifiedRadius);
```

Construct a cc3DCircleFit2DParams object with the given radius and the given value for fit mode.

Parameters

Radius — The radius of the circle. This value is expressed in the units of the 3D physical space.

FitMode — The fit mode. The supplied value must be one of the following values:

cc3DCircleFit2DDefs::eLeastSquaresComputeRadius
cc3DCircleFit2DDefs::eLeastSquaresUseSpecifiedRadius

Notes

The default values for **cc3DCircleFit2DParams** include a radius of 0 and a fit mode of *cc3DCircleFit2DDefs::eLeastSquaresUseSpecifiedRadius*; you must either specify a radius or a different fit mode before running the tool.

Throws

cc3DCircleFit2DDefs::BadParams
Radius is less than 0 or *FitMode* is not a valid member of **ccCircle2FitDefs::FitMode**.

Public Member Functions

radius

```
double radius() const;

void radius(double radius);
```

- `double radius() const;`

Returns the expected radius of the circle.

- `void radius(double radius);`

Sets the expected radius of the circle. This value is in the units of 3D physical space. If **fitMode()** is *cc3DCircleFit2DDefs::eLeastSquaresComputeRadius*, this value is ignored.

The default radius is 0.

Parameters

radius The expected radius.

Throws

cc3DCircleFit2DDefs::BadParams
radius is less than 0.

fitMode

```
enum cc3DCircleFit2DDefs::FitMode fitMode() const;

void fitMode(enum cc3DCircleFit2DDefs::FitMode fitMode);
```

- `enum cc3DCircleFit2DDefs::FitMode fitMode() const;`

Returns the current circle fitting method. The returned value is one of the following:

cc3DCircleFit2DDefs::eLeastSquaresComputeRadius
cc3DCircleFit2DDefs::eLeastSquaresUseSpecifiedRadius

- `void fitMode(enum cc3DCircleFit2DDefs::FitMode fitMode);`

Sets the circle fitting mode. If the supplied value is *cc3DCircleFit2DDefs::eLeastSquaresComputeRadius*, the tool ignores the value of **radius()** and computes the radius of the circle.

The default value is *cc3DCircleFit2DDefs::eLeastSquaresUseSpecifiedRadius*.

Parameters

fitMode The fitting mode. The supplied value must be one of:

cc3DCircleFit2DDefs::eLeastSquaresComputeRadius
cc3DCircleFit2DDefs::eLeastSquaresUseSpecifiedRadius

Throws

cc3DCircleFit2DDefs::BadParams
fitMode is not a valid member of **ccCircle2FitDefs::FitMode**.

Operators

operator==

```
bool operator==(const cc3DCircleFit2DParams& that) const;
```

Returns true if the supplied object has the same values as this one, false otherwise.

Parameters

that The object to compare to this one.

■ cc3DCircleFit2DParams

cc3DCircleFit2DResult

```
#include <ch_c3d/fit2d.h>
```

```
class cc3DCircleFit2DResult
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Class containing a single result from the 3D circle pose estimation tool.

Constructors/Destructors

cc3DCircleFit2DResult

```
cc3DCircleFit2DResult();
```

Construct a 3D circle pose estimation tool result object

Public Member Functions

circle

```
const cc3DCircle &circle() const;
```

Returns the found circle.

Throws

cc3DCircleFit2DDefs::NotComputed
This object is default constructed.

residualsRaw2D

```
cc3DResiduals residualsRaw2D() const;
```

Returns the residual error statistics (in image pixels) between this found circle and the 2D image data. The residual error statistics are computed by projecting the computed circle through each camera calibration, then computing the discrepancy between the project circle and the 2D points that were supplied to compute the circle.

Throws

cc3DCircleFit2DDefs::NotComputed
This object is default constructed.

found

```
bool found() const;
```

Returns true if this object was constructed to contain a found circle, false if it was default-constructed.

Operators

operator==

```
bool operator==(const cc3DCircleFit2DResult& that) const;
```

Returns true if the supplied object has the same values as this one, false otherwise.

Parameters

that The object to compare to this one.

cc3DCircleFit2DResultSet

```
class cc3DCircleFit2DResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Class containing a collection of results from the 3D circle pose estimation tool.

Constructors/Destructors

cc3DCircleFit2DResultSet

```
cc3DCircleFit2DResultSet() {}
```

Construct an empty result set object.

Public Member Functions

results

```
const cmStd vector<cc3DCircleFit2DResult> &results();
```

Returns all the computed found circles.

Operators

operator==

```
bool operator==(const cc3DCircleFit2DResultSet& that)
  const;
```

Returns true if the supplied object has the same values as this one, false otherwise.

Parameters

that The object to compare to this one.

cc3DCircleFitParams

```
#include <ch_c3d/fit3d.h>
```

```
class cc3DCircleFitParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Parameters class for the 3D circle fitting tool.

Note This tool supports the use of robust fitting algorithms such as RANSAC using an interface defined in *ch_cog/robstfit.h*. This interface is for beta-level use only with this release of 3D-Locate. By default, robust fitting is not enabled.

Constructors/Destructors

cc3DCircleFitParams

```
cc3DCircleFitParams() {}
```

Default constructor creates a params object with robust fit functionality disabled.

Public Member Functions

robustFitParams

```
void robustFitParams(ccRobustFitParams& params);
```

```
const ccRobustFitParams& robustFitParams()
```

Sets and gets the robust fitting parameters (beta).

cc3DCircleFitResult

```
#include <ch_c3d/fit3d.h>
```

```
class cc3DCircleFitResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Results class for the 3D circle fitting tool.

Note This tool supports the use of robust fitting algorithms such as RANSAC using an interface defined in *ch_cog/robstfit.h*. This interface is for beta-level use only with this release of 3D-Locate. By default, robust fitting is not enabled.

Constructors/Destructors

cc3DCircleFitResult

```
cc3DCircleFitResult();
```

Construct object using the following defaults:

- **found()** set to false

Public Member Functions

found

```
bool found() const;
```

Return whether a best-fitting circle was found.

circle

```
const cc3DCircle& circle() const;
```

Return the fit circle.

Throws

cc3DCircleFitDefs::NotComputed
This object is default-constructed.

residualsPhys3D

```
cc3DResiduals residualsPhys3D() const;
```

Return the residual statistics of the fitting result.

Throws

cc3DCircleFitDefs::NotComputed
This object is default-constructed.

outliers

```
const cmStd vector<c_UInt32> &outliers();
```

Outlier points not included in the fit. The returned vector is empty if robust fitting is not used.

Throws

cc3DCircleFitDefs::NotComputed
This object is default-constructed.

inliers

```
const cmStd vector<c_UInt32> &inliers() const;
```

Inlier points included in the fit. The returned vector includes the index of every supplied point if robust fitting is not used.

Throws

cc3DCircleFitDefs::NotComputed
This object is default-constructed.

reset

```
void reset();
```

Resets this object to its default-constructed state.

Operators

operator==

```
bool operator==(const cc3DCircleFitResult& that) const;
```

Returns true if the supplied object has the same values as this one, false otherwise.

Parameters

that The object to compare to this one.

cc3DCurve

```
#include <ch_c3d/shapes3d.h>

class cc3DCurve:
   public virtual ccPersistent,
   public virtual ccRepBase;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

Concrete base class for a 3D curve.

Public Member Functions

perimeter

```
virtual double perimeter() const = 0;
```

Returns the perimeter of this curve.

Throws

cc3DShapeDefs::NotFinite
This shape is not a finite curve shape in 3D space.

Notes

It returns 0 if this curve shape is empty or degenerate.

isDegenerateCurve

```
virtual bool isDegenerateCurve() const;
```

Gets whether this curve shape is degenerate, computed by checking whether **perimeter()** is 0 for finite curve shapes.

nearestPointCurve

```
virtual cc3DVect nearestPointCurve(const cc3DVect &pt)
   const = 0;
```

Returns the nearest point on this curve shape to the given point. If the nearest point is not unique, one of the nearest points is returned.

Parameters

pt The point.

Throws

cc3DShapeDefs::Empty3DShape
This curve shape is empty.

distanceCurve

```
virtual double distanceCurve(const cc3DVect &pt) const;
```

Returns the minimum distance from this curve to the supplied point.

Parameters

pt The point.

Throws

cc3DShapeDefs::Empty3DShape
This curve shape is empty.

cc3DEulerXYZ

```
#include <ch_c3d/eulerxyz.h>
```

```
class cc3DEulerXYZ;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class encapsulates the Euler angle representation of rotation. A rotation *R* of 3D coordinate axes is expressed as:

```
R = Rz * Ry * Rx
```

where

Rx is the rotation of yz-axes about fixed x-axis.

Ry is the rotation of zx-axes about fixed y-axis,

Rz is the rotation of xy-axes about fixed z-axis,

The order of applying rotations is: *Rx* first, *Ry* second and *Rz* third.

For more information on this angle representation, see

http://en.wikipedia.org/wiki/Euler_angles

Notes

This class is immutable.

Constructors/Destructors

cc3DEulerXYZ

```
cc3DEulerXYZ();

cc3DEulerXYZ(const ccRadian& angleX,
   const ccRadian& angleY, const ccRadian& angleZ);
```

- ```
  cc3DEulerXYZ();
  ```

Constructs a **cc3DEulerXYZ** object with **x()**, **y()**, and **z()** initialized to 0.
  ```

- ```
  cc3DEulerXYZ(const ccRadian& angleX,
     const ccRadian& angleY, const ccRadian& angleZ);
  ```

Constructs a **cc3DEulerXYZ** object using the provided values.

Compiler generated copy constructor, assignment and destructor OK.

Parameters

angleX The X angle

angleY The Y angle

angleZ The Z angle

Public Member Functions

x

```
ccRadian x() const;
```

Returns the yz-axes rotation angle about a fixed x-axis.

y

```
ccRadian y() const;
```

Returns the zx-axes rotation angle about a fixed y-axis.

z

```
ccRadian z() const;
```

Returns the xy-axes rotation angle about a fixed z-axis.

Operators

operator==

```
bool operator==(const cc3DEulerXYZ& that) const;
```

Returns true if this object is exactly equal to the supplied object, false otherwise.

Parameters

that The object to compare to this one.
```

cc3DEulerZYX

```
#include <ch_c3d/eulerzyx.h>
```

```
class cc3DEulerZYX;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class encapsulates the Euler angle representation of rotation. This class uses the Givens rotation convention. A rotation *R* of 3D coordinate axes is expressed as:

```
R = Rx * Ry * Rz
```

where

Rz is the rotation of xy-axes about fixed z-axis,

Ry is the rotation of zx-axes about fixed y-axis,

Rx is the rotation of yz-axes about fixed x-axis.

The order of applying rotations is: *Rz* first, *Ry* second and *Rx* third.

For more information on this angle representation, see

http://en.wikipedia.org/wiki/Euler_angles

Notes

This class is immutable.

Constructors/Destructors

cc3DEulerZYX

```
cc3DEulerZYX();

cc3DEulerZYX(const ccRadian& angleX,
   const ccRadian& angleY, const ccRadian& angleZ);
```

•

```
cc3DEulerZYX();
```

Constructs a **cc3DEulerZYX** object with **x()**, **y()**, and **z()** initialized to 0.

- ```
  cc3DEulerZYX(const ccRadian& angleX,
    const ccRadian& angleY, const ccRadian& angleZ);
  ```

  Constructs a **cc3DEulerZYX** object using the supplied values.

  Parameters

  angleX The X angle

  angleY The Y angle

  angleZ The Z angle

Public Member Functions

x

```
ccRadian x() const;
```

Returns the yz-axes rotation angle about a fixed x-axis.

y

```
ccRadian y() const;
```

Returns the zx-axes rotation angle about a fixed y-axis.

z

```
ccRadian z() const;
```

Returns the xy-axes rotation angle about a fixed z-axis.

Operators

operator==

```
bool operator==(const cc3DEulerZYX& that) const;
```

Returns true if this object is exactly equal to the supplied object, false otherwise.

Parameters

that The object to compare to this one.
```

cc3DHandEyeCalibrationInputData

```
#include <ch_c3d/handeye.h>

class cc3DHandEyeCalibrationInputData;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Class containing the data associated with a single robot station. You use this class, as opposed to **cc3DHandEyeCalibrationInputDataXO**, when you are either using hand-eye calibration to compute camera intrinsic parameters or you are supplying camera intrinsic parameters computed earlier.

This class is immutable.

Constructors/Destructors

cc3DHandEyeCalibrationInputData

```
cc3DHandEyeCalibrationInputData();

cc3DHandEyeCalibrationInputData(
   const cc3DXformRigid& robotBase3DFromHand3D,
   const ccCrspPairWeightedVector& calPlateFeatures);
```

- ```
  cc3DHandEyeCalibrationInputData();
  ```

  Constructs this object with an identity robot hand pose and an empty vector of calibration features. Such a default-constructed object cannot be used to perform calibration.

- ```
  cc3DHandEyeCalibrationInputData(
     const cc3DXformRigid& robotBase3DFromHand3D,
     const ccCrspPairWeightedVector& calPlateFeatures);
  ```

  Constructs this object with the specified robot hand pose and the calibration features observed at this pose.

Parameters

robotBase3DFromHand3D
The rigid transformation giving the pose of the robot hand in robot base 3D space.

calPlateFeatures A **ccCrspPairWeightedVector** containing the plate feature locations obtained at the given pose. Each 2D point in *calPlateFeatures* gives the 2D position of a plate vertex in 2D image space while the corresponding 3D point gives the position of the plate vertex in 3D calibration plate space.

Public Member Functions

robotBase3DFromHand3D

```
const cc3DXformRigid& robotBase3DFromHand3D() const;
```

Returns the rigid transformation giving the pose of the robot hand in robot base 3D space.

calPlateFeatures

```
const ccCrspPairWeightedVector& calPlateFeatures() const;
```

Returns a **ccCrspPairWeightedVector** containing the plate feature locations obtained at the given pose. Each 2D point in *calPlateFeatures* gives the 2D position of a plate vertex in 2D image space while the corresponding 3D point gives the position of the plate vertex in 3D calibration plate space.

Operators

operator==

```
bool operator== (
   const cc3DHandEyeCalibrationInputData& rhs) const;
```

Returns true if this object is exactly equal to the supplied object.

Parameters

rhs The object to compare to this one.

Typedefs

cc3DHandEyeCalibrationInputDataVector

```
typedef cmStd vector<cc3DHandEyeCalibrationInputData>
   cc3DHandEyeCalibrationInputDataVector;
```

Vector of **cc3DHandEyeCalibrationInputData** suitable for storing data from all of the stations associated with an individual robot.

cc3DHandEyeCalibrationInputDataXO

```
#include <ch_c3d/handeye.h>
```

```
class cc3DHandEyeCalibrationInputDataXO;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Class containing the data associated with a single robot station. You use this class, as opposed to **cc3DHandEyeCalibrationInputData**, when you are neither supplying nor computing intrinsic camera parameters. This class is suitable for robot heads where the camera or cameras, in the case of multi-camera heads, are already calibrated and you can provide the extrinsic parameters giving the pose of the calibration plate relative to the camera or the multi-camera head.

This class is immutable.

Constructors/Destructors

cc3DHandEyeCalibrationInputDataXO

```
cc3DHandEyeCalibrationInputDataXO();

cc3DHandEyeCalibrationInputDataXO(
   const cc3DXformRigid& robotBase3DFromHand3D,
   const cc3DXformRigid& camera3DFromCalPlate3D);
```

- ```
  cc3DHandEyeCalibrationInputDataXO();
  ```

  Constructs this object with an identity robot hand pose and extrinsic pose. Such a default-constructed object cannot be used to perform calibration.

- ```
  cc3DHandEyeCalibrationInputDataXO(const cc3DXformRigid&
  robotBase3DFromHand3D, const cc3DXformRigid&
  camera3DFromCalPlate3D);
  ```

  Constructs this object with the specified robot hand pose and corresponding extrinsic transform.

Parameters

robotBase3DFromHand3D
The rigid transformation giving the pose of the robot hand in robot base 3D space.

camera3DFromCalPlate3D
The rigid transformation that maps points from the coordinate system of the calibration plate to the camera's coordinates.

Public Member Functions

robotBase3DFromHand3D

```
const cc3DXformRigid& robotBase3DFromHand3D() const;
```

Returns the rigid transformation giving the pose of the robot hand in robot base 3D space.

camera3DFromCalPlate3D

```
const cc3DXformRigid& camera3DFromCalPlate3D() const;
```

The rigid transformation that maps points from the coordinate system of the calibration plate to the camera's coordinates.This pose is known as the extrinsic camera transform.

Notes

In some applications the "camera" is actually a multi-camera "head". For these applications the extrinsic transform describes the pose between the head and the calibration plate.

Operators

operator==

```
bool operator== (
   const cc3DHandEyeCalibrationInputDataXO& rhs) const;
```

Returns true if this object is exactly equal to the supplied object.

Parameters

rhs The object to compare to this one.

Typedefs

cc3DHandEyeCalibrationInputDataVectorXO

```
typedef cmStd vector<cc3DHandEyeCalibrationInputDataXO>
   cc3DHandEyeCalibrationInputDataVectorXO;
```

Vector of **cc3DHandEyeCalibrationInputDataXO** suitable for storing data from all of the stations associated with an individual robot.

cc3DHandEyeCalibrationResidualStatistics

```
#include <ch_c3d/handeye.h>

class cc3DHandEyeCalibrationResidualStatistics;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Class containing residual error statistics from hand-eye calibration.

Residual error statistics for hand-eye calibration are computed for a set of points on the surface of the calibration plate. You specify the sampling region and the number of samples in the **cc3DHandEyeCalibrationRunParams** object that you use for calibration.

For a moving camera/stationary plate hand-eye calibration result, the residual error for a given input station, *j*, is computed by transforming the points specified by the sampling parameters through the following transformations in this order:

1. The extrinsic camera parameters at station j
2. The hand-eye calibration result
3. The robot base from hand transformation for station j
4. The robot base from hand transformation for station 0
5. The hand-eye calibration result (again)
6. The extrinsic camera parameters at station 0

For a stationary camera/moving plate hand-eye calibration result, the residual error for a given station, *j*, is computed by transforming the points specified by the sampling parameters through the following transformations in this order:

1. The hand-plate calibration result
2. The pose of the robot hand at station zero
3. The pose of the robot hand at station j
4. The hand-plate calibration result (again)
5. The extrinsic camera parameters at station j
6. The extrinsic camera parameters at station 0

For both types of calibration, you can obtain residual error for all sampled points at all stations, individual points at all stations, or all points at a single station.

This class is immutable.

Constructors/Destructors

cc3DHandEyeCalibrationResidualStatistics

```
cc3DHandEyeCalibrationResidualStatistics();
```

Constructs this object with no data.

Public Member Functions

residualsOverallCalPlate3D

```
const cc3DPositionResiduals& residualsOverallCalPlate3D()
   const;
```

Returns the overall residual statistics. These statistics represent all mapped samples from all stations.

Throws

cc3DHandEyeCalibrationDefs::NoResidualStatistics
This object was default-constructed.

samplePointsCalPlate2D

```
const cmStd vector<cc2Vect>& samplePointsCalPlate2D()
   const;
```

Returns a vector of the 2D sample points, in 2D calibration plate space. The vector is stored in x-major order: the first **cc3DHandEyeCalibrationRunParams::numPlateSamplesY()** samples will all have the x coordinate of the **cc3DHandEyeCalibrationRunParams::plateRectangle()**'s origin.

meanMappedSamplePointsCalPlate3D

```
const cmStd
   vector<cc3DVect>& meanMappedSamplePointsCalPlate3D()
   const;
```

Returns a vector filled with the mean positions of the mapped 2D sample points, in the 3D calibration plate space observed at station zero. The vector is stored in the same order as the one returned by **samplePointsCalPlate2D()**.

residualsPerSampleCalPlate3D

```
const cmStd vector<cc3DPositionResiduals>
   &residualsPerSampleCalPlate3D() const;
```

Returns the residual statistics for a each individual sampling point. Each element of the returned vector holds statistics for a single sample, mapped from all stations. The vector is stored in the same order as the one returned by **samplePointsCalPlate2D()**.

residualsPerStationCalPlate3D

```
const cmStd vector<cc3DPositionResiduals>&
   residualsPerStationCalPlate3D() const;
```

Returns the "per station" residual statistics. Each element of the returned vector holds statistics for all of the samples mapped from a single station.

Operators

operator==

```
bool operator== (const
   cc3DHandEyeCalibrationResidualStatistics& rhs) const;
```

Returns true if this object is exactly equal to the supplied object.

Parameters

rhs The object to compare to this one.

cc3DHandEyeCalibrationResult

```
#include <ch_c3d/handeye.h>

class cc3DHandEyeCalibrationResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Class containing the results of a hand-eye calibration. You use this class, as opposed to **cc3DHandEyeCalibrationResultXO**, when you are either using hand-eye calibration to compute camera intrinsic parameters or you are supplying camera intrinsic parameters computed earlier.

Constructors/Destructors

cc3DHandEyeCalibrationResult

```
cc3DHandEyeCalibrationResult();

virtual ~cc3DHandEyeCalibrationResult();
```

- `cc3DHandEyeCalibrationResult();`

Constructs this object with **isCameraMoving()** set to true, an identity hand-eye transform, an identity base-plate transform, a default-constructed set of camera intrinsic parameters, an empty vector of camera calibrations, and a default constructed set of residual statistics.

Notes

An instance created by this constructor is of little use until it is initialized, assigned to, filled in by a call to **cf3DHandEyeCalibration()**, or loaded from an archive.

- `virtual ~cc3DHandEyeCalibrationResult();`

Ensures destructor is virtual (in case of derivation).

Public Member Functions

isCameraMoving

```
bool isCameraMoving() const;
```

Returns true if the calibration result was computed for the moving camera/stationary plate hand-eye calibration, false if the result was computed for the stationary camera/moving plate calibration.

movingCamera3DFromHand3D

```
const cc3DXformRigid& movingCamera3DFromHand3D() const;
```

Returns the hand-eye calibration transformation for a moving camera/stationary plate calibration. The returned transformation maps points from the robot hand 3D space to the calibrated camera 3D space.

Throws

cc3DHandEyeCalibrationDefs::StationaryCamera
The parameters used to compute this calibration specified a stationary camera/moving plate calibration.

robotBase3DFromStationaryCalPlate3D

```
const cc3DXformRigid&
   robotBase3DFromStationaryCalPlate3D() const;
```

Returns a transformation that maps points from the calibration plate 3D coordinate space to the robot base 3D space for a moving camera/stationary plate calibration.

Throws

cc3DHandEyeCalibrationDefs::StationaryCamera
The parameters used to compute this calibration specified a stationary camera/moving plate calibration.

robotBase3DFromStationaryCamera3D

```
const cc3DXformRigid& robotBase3DFromStationaryCamera3D()
   const;
```

Returns the hand-eye calibration transformation for a stationary camera/moving plate calibration. The returned transformation maps points from the calibrated camera 3D space to robot base 3D space.

Throws

cc3DHandEyeCalibrationDefs::MovingCamera
The parameters used to compute this calibration specified a moving camera/stationary plate calibration.

movingCalPlate3DFromHand3D

```
const cc3DXformRigid& movingCalPlate3DFromHand3D() const;
```

Returns a transformation that maps points from the robot hand 3D space to the calibration plate 3D coordinate space for a stationary camera/moving plate calibration.

Throws

cc3DHandEyeCalibrationDefs::MovingCamera
The parameters used to compute this calibration specified a moving camera/stationary plate calibration.

camera3DFromCalPlate3Ds

```
const cmStd
   vector<cc3DXformRigid>& camera3DFromCalPlate3Ds()
   const;
```

Returns a vector of cc3DXformRigid objects: one for each robot station. Each object contains the extrinsic camera parameters used at that station.

raw2DFromCamera2D

```
ccCalib2ParamsIntrinsic raw2DFromCamera2D() const;
```

Returns the camera intrinsics (which correspond to the mapping from camera2D space (the z=1 plane in front of the camera) to raw2D image space).

residualStatistics

```
const cc3DHandEyeCalibrationResidualStatistics&
   residualStatistics() const;
```

Returns all of the residual statistics computed during the hand-eye calibration procedure.

Operators

operator==

```
bool operator== (const cc3DHandEyeCalibrationResult& rhs)
   const;
```

Returns true if this object is exactly equal to the supplied object.

Parameters

rhs The object to compare to this one.

■ cc3DHandEyeCalibrationResult

cc3DHandEyeCalibrationResultXO

```
#include <ch_c3d/handeye.h>
```

```
class cc3DHandEyeCalibrationResultXO
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Class containing the results of a hand-eye calibration. You use this class, as opposed to **cc3DHandEyeCalibrationResult**, when you are neither supplying or computing intrinsic camera parameters. This class is suitable for robot heads where the camera or cameras, in the case of multi-camera heads, are already calibrated and you can provide the extrinsic parameters giving the pose of the calibration plate relative to the camera or the multi-camera head.

Constructors/Destructors

cc3DHandEyeCalibrationResultXO

```
cc3DHandEyeCalibrationResultXO();

virtual ~cc3DHandEyeCalibrationResultXO();
```

- ```
  cc3DHandEyeCalibrationResultXO();
  ```

  Constructs this object with **isCameraMoving()** set to true, an identity hand-eye transform, an identity base-plate transform, and a default constructed set of residual statistics.

  Notes

  An instance created by this constructor is of little use until it is initialized, assigned to, filled in by a call to **cf3DHandEyeCalibration()**, or loaded from an archive.

- ```
  virtual ~cc3DHandEyeCalibrationResultXO();
  ```

  Ensures destructor is virtual (in case of derivation).

Public Member Functions

isCameraMoving

```
bool isCameraMoving() const;
```

Returns true if the calibration result was computed for the moving camera/stationary plate hand-eye calibration, false if the result was computed for the stationary camera/moving plate calibration.

movingCamera3DFromHand3D

```
const cc3DXformRigid& movingCamera3DFromHand3D() const;
```

Returns the hand-eye calibration transformation for a moving camera/stationary plate calibration. The returned transformation maps points from the robot hand 3D space to the calibrated camera 3D space.

Throws

cc3DHandEyeCalibrationDefs::StationaryCamera
The parameters used to compute this calibration specified a stationary camera/moving plate calibration.

robotBase3DFromStationaryCalPlate3D

```
const cc3DXformRigid&
   robotBase3DFromStationaryCalPlate3D() const;
```

Returns a transformation that maps points from the calibration plate 3D coordinate space to the robot base 3D space for a moving camera/stationary plate calibration.

Throws

cc3DHandEyeCalibrationDefs::StationaryCamera
The parameters used to compute this calibration specified a stationary camera/moving plate calibration.

movingCalPlate3DFromHand3D

```
const cc3DXformRigid& movingCalPlate3DFromHand3D() const;
```

Returns a transformation that maps points from the robot hand 3D space to the calibration plate 3D coordinate space for a stationary camera/moving plate calibration.

Throws

cc3DHandEyeCalibrationDefs::MovingCamera
The parameters used to compute this calibration specified a moving camera/stationary plate calibration.

robotBase3DFromStationaryCamera3D

```
const cc3DXformRigid& robotBase3DFromStationaryCamera3D()
const;
```

Returns the hand-eye calibration transformation for a stationary camera/moving plate calibration. The returned transformation maps points from the calibrated camera 3D space to robot base 3D space.

Throws

cc3DHandEyeCalibrationDefs::MovingCamera
The parameters used to compute this calibration specified a moving camera/stationary plate calibration.

residualStatistics

```
const cc3DHandEyeCalibrationResidualStatistics&
   residualStatistics() const;
```

Returns all of the residual statistics computed during the hand-eye calibration procedure.

Operators

operator==

```
bool operator== (const cc3DHandEyeCalibrationResultXO& rhs)
const;
```

Returns true if *this exactly equals rhs.

Parameters

rhs

■ cc3DHandEyeCalibrationResultXO

cc3DHandEyeCalibrationRunParams

```
#include <ch_c3d/handeye.h>

class cc3DHandEyeCalibrationRunParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Class containing the run-time parameters for hand-eye calibration.

Constructors/Destructors

cc3DHandEyeCalibrationRunParams

```
cc3DHandEyeCalibrationRunParams();

cc3DHandEyeCalibrationRunParams(bool isCameraMoving,
   c_Int32 numPlateSamplesX, c_Int32 numPlateSamplesY,
   const ccRectangle<double>& plateRectangle);

virtual ~cc3DHandEyeCalibrationRunParams();
```

- `cc3DHandEyeCalibrationRunParams();`

  Constructs this object with the following values:

  - **isCameraMoving()** = true
  - **numPlateSamplesX()** = 0
  - **numPlateSamplesY()** = 0
  - **plateRectangle()** = **ccRectangle<double>()**

- ```
  cc3DHandEyeCalibrationRunParams(bool isCameraMoving,
     c_Int32 numPlateSamplesX, c_Int32 numPlateSamplesY,
     const ccRectangle<double>& plateRectangle);
  ```

  Constructs this object with the specified parameters.
  ```

Parameters

isCameraMoving
Specify *true* if the camera is mounted to the robot hand and the plate is stationary, false otherwise.

numPlateSamplesX
The number of sampling points for residual error computation in the X-direction.

numPlateSamplesY
The number of sampling points for residual error computation in the Y-direction.

plateRectange A rectangle specifying the part of the image to sample for residual error computation. The rectangle should be sized such that it is filled with calibration plate vertices when viewed at each robot hand station.

- `virtual ~cc3DHandEyeCalibrationRunParams();`

Ensures destructor is virtual (in case of derivation).

Public Member Functions

isCameraMoving

```
bool isCameraMoving() const;

void isCameraMoving(bool cameraMoving);
```

- `bool isCameraMoving() const;`

Returns true if this object is configured to specify that the camera is attached to the moving robot hand and the calibration plate is fixed, false if the camera is fixed and the calibration is attached to the moving arm.

- `void isCameraMoving(bool cameraMoving);`

Sets whether the camera is attached to, and moves with, the robot's hand.

Parameters

cameraMoving Specify true if the camera is attached to the moving robot hand, false if it is fixed and the calibration plate moves.

Notes

If the boolean is set to false, the hand-eye calibration routine will actually compute a hand-plate calibration.

numPlateSamplesX

```
c_Int32 numPlateSamplesX() const;

void numPlateSamplesX(c_Int32 numX);
```

- `c_Int32 numPlateSamplesX() const;`

  Returns the number of residual error sampling points in the X-direction of **plateRectangle()**.

- `void numPlateSamplesX(c_Int32 numX);`

  Sets the number of residual error sampling points in the X-direction of **plateRectangle()**.

  If either **numPlateSamplesX** or **numPlateSamplesY** is less than 1, no residual statistics will be computed.

  Parameters

  numX The number of samples.

numPlateSamplesY

```
c_Int32 numPlateSamplesY() const;

void numPlateSamplesY(c_Int32 numY);
```

- `c_Int32 numPlateSamplesY() const;`

  Returns the number of residual error sampling points in the Y-direction of **plateRectangle()**.

- `void numPlateSamplesY(c_Int32 numY);`

  Sets the number of residual error sampling points in the Y-direction of **plateRectangle()**.

  If either **numPlateSamplesX** or **numPlateSamplesY** is less than 1, no residual statistics will be computed.

  Parameters

  numY The number of samples.

plateRectangle

```
const ccRectangle<double>& plateRectangle() const;

void plateRectangle(const ccRectangle<double>& rect);
```

- `const ccRectangle<double>& plateRectangle() const;`

  Returns the sampling rectangle used for residual error computation.

- `void plateRectangle(const ccRectangle<double>& rect);`

  Sets the sampling rectangle for residual error computation. You specify the rectangle in 2D calibration plate space (the units are those in which the vertex pitch is specified, and the origin is defined by the fiducial marks on the plate).

  If the width or height of the rectangle is 0, no residual statistics will be computed.

  Parameters

  rect The sampling rectangle.

Operators

operator==

```
bool operator==
   (const cc3DHandEyeCalibrationRunParams& rhs) const;
```

Returns true if this object is exactly equal to the supplied object.

Parameters

rhs The object to compare to this one.

cc3DLine

```
#include <ch_c3d/shapes3d.h>

class cc3DLine:
   public cc3DShape,
   public cc3DCurve;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class represents a directed 3D line of infinite length.

For more information on the representations of 3D lines, see

http://en.wikipedia.org/wiki/Line_geometry

Enumerations

ParameterizationTypeOf3DLine

```
enum ParameterizationTypeOf3DLine;
```

This enumeration defines the supported parameterizations for a **cc3DLine**. Note that each parameterization uses two **cc3Vect** objects.

Value	Meaning
ePoints	Two points on the line.
ePointAndDirection	One point and a direction vector.
ePluckerUnitDirAndMoment	A Plücker direction vector and moment vector.

Constructors/Destructors

cc3DLine

```
cc3DLine();

cc3DLine(const cc3DVect& v1, const cc3DVect& v2,
   ParameterizationTypeOf3DLine type);

cc3DLine(const cc3DLineSeg& lineSeg);
```

- `cc3DLine();`

Default-constructs a degenerate line using the *cc3DLine::ePoints* parameterization with points **cc3DVect(0,0,0)** and **cc3DVect(0,0,0)**.

- ```
  cc3DLine(const cc3DVect& v1, const cc3DVect& v2,
     ParameterizationTypeOf3DLine type);
  ```

Constructs a 3D line using the supplied values. The meaning of the first two parameters depends on the specified parameterization type:

- If the parameterization is *cc3DLine::ePoints*, then *v1* and *v2* provide are two points on the line and the line direction is from *v1* to *v2*.
- If the parameterization is *cc3DLine::ePointAndDirection*, then *v1* is a point on the line and *v2* is the direction. It is not necessary to supply *v2* as a unit vector, but it will be scaled to a unit vector internally.
- If the parameterization is *cc3DLine::ePluckerUnitDirAndMoment*, then *v1* is the line direction and *v2* is the Plücker moment. It is not necessary to supply *v1* as a unit vector, but it will be scaled to a unit vector internally.

Parameters

v1 — The first vector.

v2 — The second vector.

type — The parameterization. *type* must be one of the following values:

cc3DLine::ePoints
cc3DLine::ePointAndDirection
cc3DLine::ePluckerUnitDirAndMoment

Throws

cc3DShapeDefs:::InvalidDirection
type is *cc3DLine::ePointAndDirection* and *v2* is **cc3DVect(0,0,0)** or *type* is *cc3DLine::ePluckerUnitDirAndMoment* and *v1* is **cc3DVect(0,0,0)**.
  ```

cc3DShapeDefs::InvalidLineParameterizationType
type is not a valid member of **cc3DLine::ParameterizationTypeOf3DLine**.

- `cc3DLine(const cc3DLineSeg& lineSeg);`

Constructs a 3D line from the supplied **cc3DLineSeg**. The line direction is the same as that of the supplied **cc3DLineSeg** (from **cc3DLineSeg::p1()** to **cc3DLineSeg::p2()**).

Parameters
lineSeg The line segment.

Throws
cc3DShapeDefs::DegenerateShape
lineSeg is degenerate.

Public Member Functions

clone `virtual cc3DShapePtrh clone () const;`

This is an override from class **cc3DShape**.

isFinite `virtual bool isFinite () const;`

This is an override from class **cc3DShape**.

isEmpty `virtual bool isEmpty () const;`

This is an override from class **cc3DShape**.

nearestPoint `virtual cc3DVect nearestPoint (const cc3DVect &pt) const;`

This is an override from class **cc3DShape**.

Parameters
pt The point.

boundingBox `virtual cc3DAlignedBox boundingBox() const;`

This is an override from class **cc3DShape**. Calling this member throws *cc3DShapeDefs::NotFinite*.

mapShape

```
virtual cc3DShapePtrh mapShape(const cc3DXformBase& xform)
   const;

virtual void mapShape (const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const;
```

- ```
  virtual cc3DShapePtrh mapShape(const cc3DXformBase& xform)
     const;
  ```

  Maps this shape with the supplied **cc3DXformBase**.

  This is an override from class **cc3DShape**.

  Parameters

xform	The transform with which to map.

- ```
  virtual void mapShape (const cc3DXformBase& xform,
     cc3DShapePtrh& dst) const;
  ```

  Maps this shape with the supplied **cc3DXformBase**.

  This is an override from class **cc3DShape**.

  Parameters

xform	The transform with which to map.
dst	The transformed shape.

stateType

```
virtual cc3DShapeDefs::StateType stateType() const;
```

This is an override from class **cc3DShape**.

perimeter

```
virtual double perimeter() const;
```

This is an override from class **cc3DCurve**.

isDegenerateCurve

```
virtual bool isDegenerateCurve() const;
```

This is an override from class **cc3DCurve**.

Returns true if this line is degenerate, and false otherwise.

nearestPointCurve

```
virtual cc3DVect nearestPointCurve(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DCurve**.

Parameters

pt The point.

map

```
cc3DLine map(const cc3DXformRigid &xform) const;

void map(const cc3DXformRigid &xform, cc3DLine& dst) const;
```

- `cc3DLine map(const cc3DXformRigid &xform) const;`

Returns this shape mapped by the rigid transform *xform*.

Parameters

xform The transform to map with.

- `void map(const cc3DXformRigid &xform, cc3DLine& dst) const;`

Maps this shape by the rigid transform xform and places the result in the supplied object.

Parameters

xform The transform to map with.

dst The object in which to place the result.

getPluckerUnitDirAndMoment

```
void getPluckerUnitDirAndMoment(cc3DVect& unitDir,
   cc3DVect& moment) const;
```

Returns the line direction and Plücker moment vector. The direction vector is a unit vector and may not be identical the vector supplied to the setter or constructor.

Parameters

unitDir The direction.

moment The Plücker moment.

Throws

cc3DShapeDefs::DegenerateShape
This line is degenerate.

setPluckerUnitDirAndMoment

```
void setPluckerUnitDirAndMoment(const cc3DVect& unitDir,
   const cc3DVect& moment);
```

Sets the line direction and Plücker moment vector. The direction vector does not need to be supplied as a unit vector; it will be scaled to a unit vector internally.

Parameters

unitDir The direction.

moment The Plücker moment.

pluckerUnitDir

```
cc3DVect pluckerUnitDir() const;
```

Gets the Plücker unit direction vector for this line. If the Plücker direction vector was not specified as a unit vector, the returned vector may not match that used with the setter or constructor.

Throws

cc3DShapeDefs::DegenerateShape
This line is degenerate.

pluckerMoment

```
cc3DVect pluckerMoment() const;
```

Gets the Plücker moment vector for this line.

Throws

cc3DShapeDefs::DegenerateShape
This line is degenerate.

getPointAndDirection

```
void getPointAndDirection(cc3DVect& pointNearestOrigin,
   cc3DVect& dir) const;
```

Returns a point and direction that defines this line. The returned point is the point on the line closest to the origin. This may not be the same point supplied to the setter or constructor.

Parameters

pointNearestOrigin
The point on the line closest to the origin.

dir A unit vector giving the line direction.

Throws

cc3DShapeDefs::DegenerateShape
This line is degenerate.

setPointAndDirection

```
void setPointAndDirection(const cc3DVect& point,
   const cc3DVect& dir);
```

Sets this line to the supplied point and direction. The supplied point may be any point on the line. The supplied direction need not be a unit vector; it will be scaled to a unit vector internally.

Parameters

point	A point on the line.
dir	The direction of the line.

Throws

cc3DShapeDefs::InvalidDirection
dir is **cc3DVect(0,0,0)**.

pointNearestOrigin

```
cc3DVect pointNearestOrigin() const;
```

Returns the point on this line nearest the origin.

unitDir

```
cc3DVect unitDir () const;
```

Returns the unit vector for the direction of this line. The returned vector is always a unit vector. The direction vector supplied to the various setters and constructors is normalized to a unit vector internally.

Throws

cc3DShapeDefs::DegenerateShape
This line is degenerate.

setPoints

```
void setPoints(const cc3DVect& pt1, const cc3DVect& pt2);
```

Sets the line the supplied points. The line will pass through the two points, and the line direction will be from the first point to the second.

Parameters

pt1	The first point.
pt2	The second point.

Notes

If the two points are the same, the line is degenerate.

Operators

operator==

```
bool operator==(const cc3DLine& that) const;
```

Return true if the supplied object is equal to this one, false otherwise.

Parameters

that The object to compare to this one.

cc3DLineFitParams

```
#include <ch_c3d/fit3d.h>
```

```
class cc3DLineFitParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Parameters class for the 3D line fitting tool.

Note This tool supports the use of robust fitting algorithms such as RANSAC using an interface defined in *ch_cog/robstfit.h*. This interface is for beta-level use only with this release of 3D-Locate. By default, robust fitting is not enabled.

Constructors/Destructors

cc3DLineFitParams

```
cc3DLineFitParams();
```

Default constructor creates a parameters object with robust fit functionality disabled.

Public Member Functions

robustFitParams

```
void robustFitParams(ccRobustFitParams& params);
```

```
const ccRobustFitParams& robustFitParams()
```

Sets and gets the robust fitting parameters (beta).

■ cc3DLineFitParams

cc3DLineFitResult

```
#include <ch_c3d/fit3d.h>
```

```
class cc3DLineFitResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Results class for the 3D line fitting tool.

Note This tool supports the use of robust fitting algorithms such as RANSAC using an interface defined in *ch_cog/robstfit.h*. This interface is for beta-level use only with this release of 3D-Locate. By default, robust fitting is not enabled.

Constructors/Destructors

cc3DLineFitResult

```
cc3DLineFitResult();
```

Construct object using the following defaults:

- **found()** set to false

Public Member Functions

found

```
bool found() const;
```

Return whether a best-fitting line was found.

line

```
const cc3DLine& line() const;
```

Return the fit line.

Throws

cc3DLineFitDefs::NotComputed
This object is default-constructed.

residualsPhys3D

```
cc3DResiduals residualsPhys3D() const;
```

Return the residual statistics of the fitting result.

Throws

cc3DLineFitDefs::NotComputed
This object is default-constructed.

outliers

```
const cmStd vector<c_UInt32> &outliers() const;
```

Outlier points not included in the fit. The returned vector is empty if robust fitting is not used.

Throws

cc3DLineFitDefs::NotComputed
This object is default-constructed.

inliers

```
const cmStd vector<c_UInt32> &inliers() const;
```

Inlier points included in the fit. The returned vector includes the index of every supplied point if robust fitting is not used.

Throws

cc3DLineFitDefs::NotComputed
This object is default-constructed.

reset

```
void reset();
```

Reset result object to its default-constructed state.

Operators

operator==

```
bool operator==(const cc3DLineFitResult& that) const;
```

Returns true if the supplied object has the same values as this one, false otherwise.

Parameters

that The object to compare to this one.

cc3DLineSeg

```
#include <ch_c3d/shapes3d.h>

class cc3DLineSeg:
   public cc3DShape,
   public cc3DVertex,
   public cc3DCurve;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class represents a directed 3D line segment.

Constructors/Destructors

cc3DLineSeg

```
cc3DLineSeg();

cc3DLineSeg (const cc3DVect& p1, const cc3DVect& p2,
   cc3DShapeDefs::StateType type = cc3DShapeDefs::eCurve);
```

- ```
  cc3DLineSeg();
  ```

  Default constructor produces a line segment with start and end points of **cc3DVect(0,0,0)**.

- ```
  cc3DLineSeg (const cc3DVect& p1, const cc3DVect& p2,
     cc3DShapeDefs::StateType type = cc3DShapeDefs::eCurve);
  ```

  Creates a line segment with the specified start point, end point, and state type

  Parameters

  p1 The start point.

  p2 The end point.

  type The initial state type for this line segment. You must supply one of the following values for this parameter:

  cc3DShapeDefs::eVertex
  cc3DShapeDefs::eCurve

Throws

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve

Public Member Functions

clone

```
virtual cc3DShapePtrh clone () const;
```

This is an override from class **cc3DShape**.

isFinite

```
virtual bool isFinite () const;
```

This is an override from class **cc3DShape**.

isEmpty

```
virtual bool isEmpty () const;
```

This is an override from class **cc3DShape**.

nearestPoint

```
virtual cc3DVect nearestPoint (const cc3DVect &pt) const;
```

This is an override from class **cc3DShape**.

Parameters

pt The point to which to determine the nearest point on this **cc3DLineSeg**.

boundingBox

```
virtual cc3DAlignedBox boundingBox() const;
```

This is an override from class **cc3DShape**.

mapShape

```
virtual cc3DShapePtrh mapShape(const cc3DXformBase& xform)
const;

virtual void mapShape (const cc3DXformBase& xform,
cc3DShapePtrh& dst) const;
```

- ```
  virtual cc3DShapePtrh mapShape(const cc3DXformBase& xform)
     const;
  ```

  Maps this shape with the supplied **cc3DXformBase**.

  This is an override from class **cc3DShape**.

  Parameters

  xform The transform with which to map.

- ```
  virtual void mapShape (const cc3DXformBase& xform,
     cc3DShapePtrh& dst) const;
  ```

  Maps this shape with the supplied **cc3DXformBase**.

  This is an override from class **cc3DShape**.

  Parameters

  xform The transform with which to map.

  dst The transformed shape.

nearestPointVertex

```
virtual cc3DVect nearestPointVertex(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DVertex**.

Parameters

pt The point.

distanceVertex

```
virtual double distanceVertex(const cc3DVect &pt) const;
```

This is an override from class **cc3DVertex**.

Parameters

pt The point.

perimeter

```
virtual double perimeter() const;
```

This is an override from class **cc3DCurve**.

■ cc3DLineSeg

nearestPointCurve

```
virtual cc3DVect nearestPointCurve(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DCurve**.

Parameters

pt	The point.

map

```
cc3DLineSeg map(const cc3DXformRigid &xform) const;

void map(const cc3DXformRigid &xform, cc3DLineSeg& dst)
const;
```

- `cc3DLineSeg map(const cc3DXformRigid &xform) const;`

Returns this shape mapped by the rigid transform *xform.*

Parameters

xform	The transform to map with.

- `void map(const cc3DXformRigid &xform, cc3DLineSeg& dst) const;`

Maps this shape by the rigid transform xform and places the result in the supplied object.

Parameters

xform	The transform to map with.
dst	The object in which to place the result.

stateType

```
virtual cc3DShapeDefs::StateType stateType() const;

void stateType(cc3DShapeDefs::StateType type);
```

- `virtual cc3DShapeDefs::StateType stateType() const;`

Returns the state type of this object.

- `void stateType(cc3DShapeDefs::StateType type);`

Sets the state type of this line segment. The state type influences how various methods (such as **nearestPoint()**) inherited from **cc3DShape** class are interpreted.

Parameters

type The state type. *type* must be one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve

Notes

The default shape type is *cc3DShapeDefs::eCurve*.

Throws

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve

p1

```
cc3DVect p1() const;
void p1(const cc3DVect& pt);
```

- `cc3DVect p1() const;`

Returns the start point of this line segment.

- `void p1(const cc3DVect& pt);`

Sets the start point of this line segment. The default start point is **cc3DVect(0,0,0)**.

Parameters

pt The point.

p2

```
cc3DVect p2() const;
void p2(const cc3DVect& pt);
```

- `cc3DVect p2() const;`

Returns the end point of this line segment.

- `void p2(const cc3DVect& pt);`

Sets the end point of this line segment.The default end point is **cc3DVect(0,0,0)**.

Parameters

pt The point.

len

```
double len() const;
```

Returns the length of this line segment.

line

```
cc3DLine line() const;
```

Returns a **cc3DLine** object which includes the start point and end point of this line segment, and has the same direction as this line segment,

Operators

operator==

```
bool operator==(const cc3DLineSeg& that) const;
```

Returns true if this object is exactly equal to

<that>, and false otherwise.

Parameters

that

cc3DPlane

```
#include <ch_c3d/shapes3d.h>

class cc3DPlane:
   public cc3DShape,
   public cc3DSurface;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class represents a directed 3D plane.

Constructors/Destructors

cc3DPlane

```
cc3DPlane();

cc3DPlane(const cc3DVect& pointOnPlane,
   const cc3DVect& normal);

cc3DPlane(const cc3DVect& pt1, const cc3DVect& pt2,
   const cc3DVect& pt3);

cc3DPlane(const cc3DVect& normal, double offset);
```

- `cc3DPlane();`

Default-constructs a degenerate plane by calling the three-point constructor with three **cc3DVect(0,0,0)**.

Notes

normal() throws *cc3DShapeDefs::DegenerateShape* for a degenerate plane.

- ```
  cc3DPlane(const cc3DVect& pointOnPlane,
     const cc3DVect& normal);
  ```

Constructs a 3D plane defined by a single point and the normal vector.

Parameters

pointOnPlane A point on the plane.
  ```

normal A vector normal to the plane. *normal* is scaled internally to a unit vector.

Throws

cc3DShapeDefs::InvalidDirection
normal is **cc3DVect(0,0,0)**.

- ```
cc3DPlane(const cc3DVect& pt1, const cc3DVect& pt2,
  const cc3DVect& pt3);
```

Constructs a 3D plane defined by three points. If the three points are collinear, the plane is degenerate.

**Parameters**

*pt1* The first point.

*pt2* The second point.

*pt3* The third point.

**Notes**

For a non-degenerate plane, the normal direction of the plane is defined as the cross product of (*pt1* - *pt2*) and (*pt3* - *pt1*).

- ```
cc3DPlane(const cc3DVect& normal, double offset);
```

Constructs a 3D plane based on the supplied normal vector and offset. The offset gives the translation of the plane from the origin along the normal vector.

The normal vector need not be supplied as a unit vector; it is scaled internally to a unit vector.

Parameters

normal The normal vector.

offset The distance from the origin to the plane in the direction of *normal*.

Notes

Multiplying the normalized normal vector by the offset produces a point on the plane.

Throws

cc3DShapeDefs::InvalidDirection
normal is **cc3DVect(0,0,0)**.

Public Member Functions

clone

```
virtual cc3DShapePtrh clone () const;
```

This is an override from class **cc3DShape**.

isFinite

```
virtual bool isFinite () const;
```

This is an override from class **cc3DShape**.

isEmpty

```
virtual bool isEmpty () const;
```

This is an override from class **cc3DShape**.

nearestPoint

```
virtual cc3DVect nearestPoint (const cc3DVect &pt) const;
```

This is an override from class **cc3DShape**.

Parameters

pt The point.

distance

```
virtual double distance(const cc3DVect &pt) const;
```

This is an override from class **cc3DShape**.

Parameters

pt The point.

boundingBox

```
virtual cc3DAlignedBox boundingBox() const;
```

This is an override from class **cc3DShape**.

mapShape

```
virtual cc3DShapePtrh mapShape(const cc3DXformBase &xform)
   const;

virtual void mapShape (const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const;
```

•

```
virtual cc3DShapePtrh mapShape(const cc3DXformBase &xform)
   const;
```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters
xform The transform with which to map.

-
```
virtual void mapShape (const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const;
```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters
xform The transform with which to map.

dst The transformed shape.

stateType
```
virtual cc3DShapeDefs::StateType stateType() const;
```

This is an override from class **cc3DShape**. The state type is always *cc3DShapeDefs::eSurface*.

area
```
virtual double area() const;
```

This is an override from class **cc3DSurface**.

Calling this member throws *cc3DShapeDefs::NotFinite*.

isDegenerateSurface
```
virtual bool isDegenerateSurface() const;
```

This is an override from class **cc3DSurface**.

Returns true if this plane is degenerate, and false otherwise.

nearestPointSurface
```
virtual cc3DVect nearestPointSurface(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DSurface**.

Parameters
pt The point.

distanceSurface
```
virtual double distanceSurface(const cc3DVect &pt) const;
```

This is an override from class **cc3DSurface**.

Parameters
pt The point.

map

```
cc3DPlane map(const cc3DXformRigid &xform) const;

void map(const cc3DXformRigid &xform, cc3DPlane& dst)
const;
```

- `cc3DPlane map(const cc3DXformRigid &xform) const;`

Returns the result of mapping this **cc3DCircle** by the supplied rigid transformation.

Parameters
xform The transformation.

- `void map(const cc3DXformRigid &xform, cc3DPlane& dst) const;`

Maps this shape by the rigid transform xform and places the result in the supplied object.

Parameters
xform The transform to map with.

dst The object in which to place the result.

normal

```
cc3DVect normal() const;
```

Gets the unit vector normal to this plane. This vector is a unit vector; it may differ from the normal vector used in a setter or constructor.

Throws
cc3DShapeDefs::DegenerateShape
This plane is degenerate.

setPoints

```
void setPoints(const cc3DVect& pt1, const cc3DVect& pt2,
   const cc3DVect& pt3);
```

Sets the plane using three points. If the three points are collinear, the plane is degenerate.

Parameters
pt1 The first point.

pt2 The second point.

pt3 The third point.

Notes

For a non-degenerate plane, the normal direction of the plane is defined as the cross product of (*pt1* - *pt2*) and (*pt3* - *pt1*).

getPointAndNormal

```
void getPointAndNormal(cc3DVect& pointNearestOrigin,
   cc3DVect& normal) const;
```

Returns a point and normal vector that defines this plane. The returned point is the point on this plane that is closest to the origin; it may differ from the point supplied to the setter. The returned normal vector is a unit vector; it may differ from the vector supplied to the setter.

Parameters

pointNearestOrigin
The point on this plane closest to the origin.

normal A unit vector that is normal to the plane.

Throws

cc3DShapeDefs::DegenerateShape
the plane is degenerate.

setPointAndNormal

```
void setPointAndNormal(const cc3DVect& pt,
   const cc3DVect& normal);
```

Sets this plane to the supplied point and normal vector. The supplied normal vector does not need to be a unit vector; it is normalized to a unit vector internally. The supplied point may be any point on the plane.

Parameters

pt A point on the plane.

normal A vector normal to the plane.

Throws

cc3DShapeDefs::InvalidDirection
normal is **cc3DVect(0,0,0)**.

getNormalAndOffset

```
void getNormalAndOffset(cc3DVect& normal, double& offset)
   const;
```

Return a vector normal to this plane and the offset of the plane from the origin in the normal direction. The returned vector is a unit vector; it may differ from the vector supplied to the setter or constructor.

Parameters

normal A unit vector normal to this plane.

offset The distance of the plane from the origin.

Throws

cc3DShapeDefs::DegenerateShape
This plane is degenerate.

Notes

Multiplying the normalized normal vector by the offset produces a point on the plane.

setNormalAndOffset

```
void setNormalAndOffset(const cc3DVect& normal,
   double offset);
```

Sets this plane to the supplied normal vector and offset. The offset is the translation of the plane from the origin in the normal direction.

Parameters

normal A vector normal to the plane. *normal* need not be a unit vector; it is scaled internally to be a unit vector.

offset The distance of the plane from the origin.

Throws

cc3DShapeDefs::InvalidDirection
normal is **cc3DVect(0,0,0)**

signedDistance

```
double signedDistance(const cc3DVect &pt) const;
```

Returned the signed distance from this plane to a point.

The distance is the dot product of point - pointNearestOrigin() and normal().

```
(pt - pointNearestOrigin()) dot product normal()
```

where *pt* is the supplied point.

The returned value is positive if the point is on the same side of the plane as the normal vector, and negative if it is on the opposite side.

Parameters
pt The point.

Throws
cc3DShapeDefs::DegenerateShape
This plane is degenerate.

offset

```
double offset() const;
```

Returns the translation of this plane from the origin in the direction of the normal vector.

Throws
cc3DShapeDefs::DegenerateShape
This plane is degenerate.

pointNearestOrigin

```
cc3DVect pointNearestOrigin() const
```

Returns the point on this plane nearest the origin.

projectVectorOntoPlane

```
cc3DVect projectVectorOntoPlane(const cc3DVect &vect)
   const;
```

Returns the projection of the supplied vector onto this plane.

Parameters
vect The vector to project.

Throws
cc3DShapeDefs::DegenerateShape
This plane is degenerate.

Notes
The returned vector is defined by the nearest point on the plane to the supplied vector minus the nearest point on the plane to **cc3DVect(0,0,0)**.

Operators

operator==

```
bool operator==(const cc3DPlane& that) const;
```

Return true if the supplied object is equal to this one, false otherwise.

Parameters

that The object to compare to this one.

cc3DPlaneFitParams

```
#include <ch_c3d/fit3d.h>
```

```
class cc3DPlaneFitParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Parameters class for the 3D plane fitting tool.

Note This tool supports the use of robust fitting algorithms such as RANSAC using an interface defined in *ch_cog/robstfit.h*. This interface is for beta-level use only with this release of 3D-Locate. By default, robust fitting is not enabled.

Constructors/Destructors

cc3DPlaneFitParams

```
cc3DPlaneFitParams() {}
```

Default constructor creates a params object with robust fit functionality disabled.

Public Member Functions

robustFitParams

```
void robustFitParams(ccRobustFitParams& params);
```

```
const ccRobustFitParams& robustFitParams()
```

Sets and gets the robust fitting parameters (beta).

cc3DPlaneFitParams

cc3DPlaneFitResult

```
#include <ch_c3d/fit3d.h>
```

```
class cc3DPlaneFitResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Results class for the 3D line fitting tool.

Note This tool supports the use of robust fitting algorithms such as RANSAC using an interface defined in *ch_cog/robstfit.h*. This interface is for beta-level use only with this release of 3D-Locate. By default, robust fitting is not enabled.

Constructors/Destructors

cc3DPlaneFitResult

```
cc3DPlaneFitResult() : found_(false), numPoints_(0) {}
```

Construct object using the following defaults:

- **found()** set to false

Public Member Functions

found

```
bool found() const;
```

Return whether a best-fitting plane was found.

plane

```
const cc3DPlane& plane() const;
```

Return the fit plane.

Throws

cc3DPlaneFitDefs::NotComputed
This object is default-constructed.

residualsPhys3D

```
cc3DResiduals residualsPhys3D() const;
```

Return the residual statistics of the fitting result.

Throws

cc3DPlaneFitDefs::NotComputed
This object is default-constructed.

inliers

```
const cmStd vector<c_UInt32> &inliers() const;
```

Inlier points included in the fit. The returned vector includes the index of every supplied point if robust fitting is not used.

Throws

cc3DPlaneFitDefs::NotComputed
This object is default-constructed.

outliers

```
const cmStd vector<c_UInt32> &outliers() const;
```

Outlier points not included in the fit. The returned vector is empty if robust fitting is not used.

Throws

cc3DPlaneFitDefs::NotComputed
This object is default-constructed.

reset

```
void reset();
```

Reset result object to its default-constructed state.

Operators

operator==

```
bool operator==(const cc3DPlaneFitResult& other) const;
```

Returns true if the supplied object has the same values as this one, false otherwise.

Parameters

that The object to compare to this one.

cc3DPoint

```
#include <ch_c3d/shapes3d.h>

class cc3DPoint:
   public cc3DShape,
   public cc3DVertex;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class represents a 3D point.

Constructors/Destructors

cc3DPoint

```
cc3DPoint(const cc3DVect& point = cc3DVect(0,0,0));

cc3DPoint(double x, double y, double z);
```

- `cc3DPoint(const cc3DVect& point = cc3DVect(0,0,0));`

Constructs a point with the specified position vector. The default location is 0,0,0.

Parameters

point The point location.

- `cc3DPoint(double x, double y, double z);`

Constructs a point with the specified 3D position.

Parameters

x The x-position of the point.

y The y-position of the point.

z The z-position of the point.

Public Member Functions

clone

```
virtual cc3DShapePtrh clone () const;
```

This is an override from class **cc3DShape**.

isFinite

```
virtual bool isFinite () const;
```

This is an override from class **cc3DShape**.

isEmpty

```
virtual bool isEmpty () const;
```

This is an override from class **cc3DShape**.

nearestPoint

```
virtual cc3DVect nearestPoint (const cc3DVect &pt) const;
```

This is an override from class **cc3DShape**.

Parameters

pt The point.

boundingBox

```
virtual cc3DAlignedBox boundingBox() const;
```

This is an override from class **cc3DShape**.

mapShape

```
virtual cc3DShapePtrh mapShape(const cc3DXformBase& xform)
   const;

virtual void mapShape(const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const;
```

•

```
virtual cc3DShapePtrh mapShape(const cc3DXformBase& xform)
   const;
```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters

xform The transform with which to map.

- ```
  virtual void mapShape (const cc3DXformBase& xform,
     cc3DShapePtrh& dst) const;
  ```

  Maps this shape with the supplied **cc3DXformBase**.

  This is an override from class **cc3DShape**.

  Parameters

xform	The transform with which to map.
dst	The transformed shape.

stateType

```
virtual cc3DShapeDefs::StateType stateType() const;
```

This is an override from class **cc3DShape**.

nearestPointVertex

```
virtual cc3DVect nearestPointVertex(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DVertex**.

Parameters

pt	The point.

map

```
cc3DPoint map(const cc3DXformRigid &xform) const;

void map(const cc3DXformRigid &xform, cc3DPoint& dst)
const;
```

- ```
  cc3DPoint map(const cc3DXformRigid &xform) const;
  ```

  Returns this shape mapped by the rigid transform xform.

  Parameters

xform	The transform to map with.

- ```
  void map(const cc3DXformRigid &xform, cc3DPoint& dst)
  const;
  ```

  Maps this shape by the rigid transform xform and places the result in the supplied object.

  Parameters

xform	The transform to map with.
dst	The object in which to place the result.
  ```

■ cc3DPoint

vect

```
cc3DVect vect() const;
void vect(const cc3DVect& newVect);
```

- `cc3DVect vect() const;`

  Returns a **cc3DVect** giving the position of this point.

- `void vect(const cc3DVect& newVect);`

  Sets this point's position to the supplied **cc3DVect**.

  Parameters

  newVect The new position.

x

```
double x() const;
void x(double newX);
```

- `double x() const;`

  Returns the x-coordinate of this point.

- `void x(double newX);`

  Sets the x-coordinate of this point. The default value is 0.

  Parameters

  newX The coordinate

y

```
double y() const;
void y(double newY);
```

- `double y() const;`

  Returns the y-coordinate of this point.

- `void y(double newY);`

  Sets the y-coordinate of this point. The default value is 0.

  Parameters

  newY The coordinate

z

```
double z() const;
void z(double newZ);
```

- `double z() const;`

  Returns the z-coordinate of this point.

- `void z(double newZ);`

  Sets the z-coordinate of this point. The default value is 0.

  Parameters
  newZ The coordinate

Operators

operator==

```
bool operator==(const cc3DPoint& that) const;
```

Return true if the supplied object is equal to this one, false otherwise.

Parameters
that The object to compare to this one.

operator cc3DVect

```
operator cc3DVect() const;
```

Cast operator.

operator=

```
cc3DPoint& operator= (const cc3DVect& vect);
```

Assignment operator for conversion to **cc3DVect**.

Parameters
vect The **cc3DVect** to assign to this **cc3DPoint**.

■ cc3DPoint

cc3DPointSet2D3D

```
#include <ch_c3d/corresp.h>
```

```
class cc3DPointSet2D3D;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Convenience class that holds a set of correspondences between 2D points in image space, 3D points in physical space, and the weight associated with each correspondence.

Note The compiler-generated constructor is used.

Public Member Functions

points2D

```
const cmStd vector<cc2Vect>& points2D() const;
void points2D(const cmStd vector<cc2Vect>& p2D);
```

- `const cmStd vector<cc2Vect>& points2D() const;`

Returns the vector of 2D points.

- `void points2D(const cmStd vector<cc2Vect>& p2D);`

Sets the vector of 2D points.

Parameters

p2D The points.

points3D

```
const cmStd vector<cc3DVect>& points3D() const;
void points3D(const cmStd vector<cc3DVect>& p3D);
```

- `const cmStd vector<cc3DVect>& points3D() const;`

Returns the vector of 3D points.

- `void points3D(const cmStd vector<cc3DVect>& p3D);`

Sets the vector of 3D points.

Parameters

p3D The points.

weights

```
const cmStd vector<double>& weights() const;

void weights(const cmStd vector<double>& w);
```

- `const cmStd vector<double>& weights() const;`

Returns the vector of weights.

- `void weights(const cmStd vector<double>& w);`

Sets the vector of weights.

If you supply a weights vector with no elements, all weights are presumed to be 1.0.

Parameters

w The weights.

points2D3D

```
void points2D3D(const cmStd vector<cc2Vect>& p2D,
   const cmStd vector<cc3DVect>& p3D,
   bool resetWeightsToEmpty = true);

void points2D3D(const cmStd vector<cc2Vect>& p2D,
   const cmStd vector<cc3DVect>& p3D,
   const cmStd vector<double>& weights);
```

-
```
void points2D3D(const cmStd vector<cc2Vect>& p2D,
   const cmStd vector<cc3DVect>& p3D,
   bool resetWeightsToEmpty = true);
```

Set the 2D and 3D points. Using this overload you can reset all weights to 1.0 (by emptying the weights vector) or you can leave the weights vector unchanged.

Parameters

p2D The 2D points.

p3D The 3D points.

resetWeightsToEmpty
If true, the weights vector is emptied, setting all weights to 1.0. If false, the weights vector is not changed.

-
```
void points2D3D(const cmStd vector<cc2Vect>& p2D,
   const cmStd vector<cc3DVect>& p3D,
   onst cmStd vector<double>& weights);
```

Set the 2D points, the 3D points, and the weights for all points.

Parameters

p2D The 2D points

p3D The 3D points.

weights The weights.

Operators

operator==

```
bool operator==(const cc3DPointSet2D3D& rhs) const;
```

Return true if the supplied object is equal to this one, false otherwise.

Parameters

rhs The object to compare to this one.

cc3DPositionResiduals

```
#include <ch_c3d/residual.h>
```

```
class cc3DPositionResiduals;
```

An immutable class that holds residual error statistics for a collection of 3D locations. This class lets you obtain both maximum and RMS error for differences in the x-, y-, and z-direction as well as for the Euclidean distance between expected and actual points.

Constructors/Destructors

cc3DPositionResiduals

```
cc3DPositionResiduals();

cc3DPositionResiduals(
   const cc3DResiduals& residualsX,
   const cc3DResiduals& residualsY,
   const cc3DResiduals& residualsZ,
   const cc3DResiduals& residualsDist);
```

- ```
  cc3DPositionResiduals();
  ```

Constructs this object with all values set to zero.

- ```
  cc3DPositionResiduals(
     const cc3DResiduals& residualsX,
     const cc3DResiduals& residualsY,
     const cc3DResiduals& residualsZ,
     const cc3DResiduals& residualsDist);
  ```

Constructs this object with the supplied residual error objects.

Parameters

residualsX Residual error statistics in the x-direction.

residualsY Residual error statistics in the y-direction.

residualsZ Residual error statistics in the z-direction.

residualsDist Residual error statistics for Euclidean distance.

Public Member Functions

residualsX

```
cc3DResiduals residualsX() const;
```

Returns the residual error statistics for the x-direction.

residualsY

```
cc3DResiduals residualsY() const;
```

Returns the residual error statistics for the y-direction.

residualsZ

```
cc3DResiduals residualsZ() const;
```

Returns the residual error statistics for the z-direction.

residualsDist

```
cc3DResiduals residualsDist() const;
```

Returns the residual error statistics for Euclidean distance.

Operators

operator==

```
bool operator== (const cc3DPositionResiduals& rhs) const;
```

Return true if the supplied object is equal to this one, false otherwise.

Parameters

that The object to compare to this one.

ccQuaternion

```
#include <ch_c3d/quatern.h>

class ccQuaternion;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

This class implements a quaternion, a mathematical construct that can be used to represent rigid 3D rotation.

ccQuaternion represents a quaternion as

$$w + xi + yj + zk$$

where *w,x,y*, and *z* are real numbers and

$$i^2 = j^2 = k^2 = ijk = -1$$

w is the scalar part of the quaternion while *x, y,* and *z* form the vector (imaginary) part.

For more information on quaternions, see

http://en.wikipedia.org/wiki/Quaternion

For general manipulations of 3D rotation values, use the **ccRotation** class. **ccQuaternion** is intended to allow you to obtain and set 3D rotation values using quaternions, not to provide a general facility for manipulating angles using quaternions.

Notes

This class is immutable.

Constructors/Destructors

ccQuaternion

```
ccQuaternion();
ccQuaternion(double w, double x, double y, double z);
```

- `ccQuaternion();`

  Default-constructs an object with the following values:

  - **w()** = 0;
  - **x()** = 0;
  - **y()** = 0;
  - **z()** = 0

- `ccQuaternion(double w, double x, double y, double z);`

  Constructs a quaternion using the given values.

  Parameters

w	The first component.
x	The second component.
y	The third component.
z	The fourth component.

Public Member Functions

w

```
double w() const;
```

Gets the scalar part (the first component) of this quaternion.

xyz

```
cc3DVect xyz() const;
```

Gets the vector (imaginary) part of this quaternion.

x

```
double x() const;
```

Gets the second component of this quaternion.

y

```
double y() const;
```

Gets the third component of this quaternion.

z

```
double z() const;
```

Gets the fourth component of this quaternion.

conjugate

```
ccQuaternion conjugate() const;
```

Gets the conjugate quaternion of this quaternion, which is defined as:

$$w - xi - yj - zk$$

normSquared

```
double normSquared() const;
```

Gets the norm squared value. The norm squared value is the Grassmann product of this quaternion and its conjugate:

$$normSquared = ww + xx + yy + zz$$

norm

```
double norm() const;
```

Gets the norm (magnitude) value of this quaternion (the square root of **normSquared()**).

unit

```
ccQuaternion unit() const;
```

Return a normalized (unit) quaternion based on this one. The **norm()** value of the returned quaternion is 1.

Throws

cc3DMathDefs::ZeroQuaternion
w(), **x()**, **y()**, and **z()** are all zero.

Notes

A unit quaternion is obtained by dividing the quaternion by its norm() value.

grassmannProduct

```
ccQuaternion grassmannProduct(const ccQuaternion& rhs)
   const;
```

Gets a new quaternion which is the Grassmann product of this quaternion and the supplied one.

Assume that a quaternion A is represented as

```
A.w() + A.vec
```

where A.vec is

```
A.x() i + A.y() j + A.z() k
```

then the Grassman product of A and B is

```
A.w() B.w() - A.vec dotProduct B.vec +
A.w() B.vec + B.w() A.vec +
A.vec crossProduct B.vec
```

Parameters

rhs The **ccQuaternion** to use to compute the Grassman product.

Notes

Grassmann product is non-commutative: A GrassmannProduct B does not equal B GrassmannProduct A.

innerProduct

```
double innerProduct(const ccQuaternion& rhs) const;
```

Gets a scalar which is the inner product of this quaternion and the supplied one.

Parameters

rhs The quaternion with which to compute the inner product.

Notes

Given quaternions A and B,

```
A innerProduct B = A.w() B.w() + A.xyz() dotProduct B.xyz()
```

Notes

For more information about quaternion products, see the section *Quaternion products* in the following page:

http://en.wikipedia.org/wiki/Quaternion

add

```
ccQuaternion add(const ccQuaternion& rhs) const;
```

Gets a new quaternion by adding this quaternion and the supplied one.

Parameters

rhs The quaternion to add to this one.

Notes

Given quaternions A and B

```
A + B = A.w() + B.w() +
(A.x() + B.x()) i +
```

```
(A.y() + B.y()) j +
(A.z() + B.z()) k
```

subtract

`ccQuaternion subtract(const ccQuaternion& rhs) const;`

Gets a new quaternion by subtracting the supplied quaternion from this one.

Parameters

rhs The quaternion to subtract from this one.

Notes

Given quaternions A and B

```
A - B = A.w() - B.w() +
(A.x() - B.x()) i +
(A.y() - B.y()) j +
(A.z() - B.z()) k
```

scale

`ccQuaternion scale(double s) const;`

Gets a new quaternion which is the multiplication of this quaternion with a supplied scale factor.

Parameters

s The scale factor.

Notes

Given a quaternion *A* and scale factor *s*, the scaled quaternion is:

```
s A.w() + ( s A.x() )i
+ ( s A.y() )j
+ ( s A.z() )k
```

Operators

operator==

`bool operator==(const ccQuaternion& that) const;`

Return true if the supplied object is equal to this one, false otherwise.

Parameters

that The object to compare to this one.

operator+

```
ccQuaternion operator+(const ccQuaternion& rhs) const;
```

Convenience operator overload. Add the supplied quaternion to this one and return the result.

Parameters

rhs The object to add.

operator+=

```
ccQuaternion& operator+=(const ccQuaternion& rhs);
```

Convenience operator overload. Add the supplied quaternion to this one.

Parameters

rhs The object to add.

operator-

```
ccQuaternion operator-(const ccQuaternion& rhs) const;
```

Convenience operator overload. Subtract the supplied quaternion from this one and return the result.

Parameters

rhs The object to subtract.

operator-=

```
ccQuaternion& operator-=(const ccQuaternion& rhs);
```

Convenience operator overload. Subtract the supplied quaternion from this one.

Parameters

rhs The object to subtract.

operator*

```
ccQuaternion operator*(double s) const;
```

Convenience operator overload. Multiply this quaternion by the supplied scale factor and return the result.

Parameters

s The scale factor.

operator*=

```
ccQuaternion& operator*=(double s);
```

Convenience operator overload. Multiply this quaternion by the supplied scale factor.

Parameters

s The scale factor.

cc3DRay

```
#include <ch_c3d/shapes3d.h>

class cc3DRay:
   public cc3DShape,
   public cc3DVertex,
   public cc3DCurve;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class represents a directed 3D ray with a start point, a direction, and an infinite length.

Enumerations

ParameterizationTypeOf3DRay

```
enum ParameterizationTypeOf3DRay;
```

This enumeration defines the supported parameterizations for a **cc3DRay**. Note that each parameterization uses two **cc3Vect** objects.

Value	Meaning
ePoints	One point defining the start of the ray and a second point that lies on the ray.
ePointAndDirection	A point defining the start of the ray and a vector giving its direction.

Constructors/Destructors

cc3DRay

```
cc3DRay();

cc3DRay (const cc3DVect& v1, const cc3DVect& v2,
   ParameterizationTypeOf3DRay constructionType,
   cc3DShapeDefs::StateType type = cc3DShapeDefs::eCurve);
```

- `cc3DRay();`

Default-constructs a degenerate ray constructed with two **cc3DVect(0,0,0)** and the *cc3DRay::ePoints* parameterization.

- ```
  cc3DRay (const cc3DVect& v1, const cc3DVect& v2,
     ParameterizationTypeOf3DRay constructionType,
     cc3DShapeDefs::StateType type = cc3DShapeDefs::eCurve);
  ```

Constructs a 3D ray using the specified parameters, parameterization type, and state type. The meaning of the first two parameters depends on the specified parameterization type:

- If the parameterization is *cc3DRay::ePoints*, then *v1* is the start point of the ray and *v2* is any other point on the ray. The ray direction is from *v1* to *v2*.
- If the parameterization is *cc3DRay::ePointAndDirection*, then *v1* is the start point of the ray and *v2* is the direction.

Parameters

v1 The first vector.

v2 The second vector.

constructionType
The parameterization for this constructor. *constructionType* must be one of the following values:

cc3DRay::ePoints
cc3DRay::ePointAndDirection

type The initial state type for this ray. *type* must be one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
  ```

Throws

cc3DShapeDefs::InvalidDirection
constructionType is *cc3DRay::ePointAndDirection* and *v2* is **cc3DVect(0,0,0)**.

cc3DShapeDefs::InvalidRayParameterizationType
constructionType is neither *cc3DRay::ePoints* nor *cc3DRay::ePointAndDirection.*

cc3DShapeDefs::InvalidStateType
type is neither *cc3DShapeDefs::eVertex* nor *cc3DShapeDefs::eCurve*.

Public Member Functions

clone

```
virtual cc3DShapePtrh clone () const;
```

This is an override from class **cc3DShape**.

isFinite

```
virtual bool isFinite () const;
```

This is an override from class **cc3DShape**.

isEmpty

```
virtual bool isEmpty () const;
```

This is an override from class **cc3DShape**.

nearestPoint

```
virtual cc3DVect nearestPoint (const cc3DVect &pt) const;
```

This is an override from class **cc3DShape**.

Parameters

pt The point.

boundingBox

```
virtual cc3DAlignedBox boundingBox() const;
```

This is an override from class **cc3DShape**.

mapShape

```
virtual cc3DShapePtrh mapShape(const cc3DXformBase& xform)
   const;

virtual void mapShape (const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const;
```

- ```
  virtual cc3DShapePtrh mapShape(const cc3DXformBase& xform)
     const;
  ```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters

xform The transform with which to map.

- ```
  virtual void mapShape (const cc3DXformBase& xform,
     cc3DShapePtrh& dst) const;
  ```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters

xform The transform with which to map.

dst The transformed shape.

nearestPointVertex

```
virtual cc3DVect nearestPointVertex(const cc3DVect &pt)
  const;
```

This is an override from class **cc3DVertex**.

Parameters

pt The point.

distanceVertex

```
virtual double distanceVertex(const cc3DVect &pt) const;
```

This is an override from class **cc3DVertex**.

Parameters

pt The point.

perimeter

```
virtual double perimeter() const;
```

This is an override from class **cc3DCurve**.

This function throws *cc3DShapeDefs::NotFinite*.

nearestPointCurve

```
virtual cc3DVect nearestPointCurve(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DCurve**.

Parameters

pt The point.

map

```
cc3DRay map(const cc3DXformRigid &xform) const;

void map(const cc3DXformRigid &xform, cc3DRay& dst) const;
```

- `cc3DRay map(const cc3DXformRigid &xform) const;`

Returns this shape mapped by the rigid transform xform.

Parameters

xform The transform to map with.

- `void map(const cc3DXformRigid &xform, cc3DRay& dst) const;`

Maps this shape by the rigid transform xform and places the result in the supplied object.

Parameters

xform The transform to map with.

dst The object in which to place the result.

stateType

```
virtual cc3DShapeDefs::StateType stateType() const;

void stateType(cc3DShapeDefs::StateType type);
```

- `virtual cc3DShapeDefs::StateType stateType() const;`

Returns the state type of this object.

• `void stateType(cc3DShapeDefs::StateType type);`

Sets the state type of this ray. The state type influences how various methods (such as **nearestPoint()**) inherited from **cc3DShape** class are interpreted.

Parameters

type The state type. *type* must be one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve

Notes

The default shape type is *cc3DShapeDefs::eVolume*.

Throws

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve

p1

```
cc3DVect p1() const;
void p1(const cc3DVect& pt);
```

• `cc3DVect p1() const;`

Returns the start point of this ray.

• `void p1(const cc3DVect& pt);`

Sets the start point of this ray.

The default value is **cc3DVect(0,0,0)**.

Parameters

pt The start point.

dir

```
cc3DVect dir() const;
void dir(const cc3DVect& newDir);
```

• `cc3DVect dir() const;`

Returns the direction of the ray.

• `void dir(const cc3DVect& newDir);`

Sets the direction of the ray.

The default value is **cc3DVect(0,0,0)**.

Parameters
newDir The new direction.

Throws
cc3DShapeDefs::InvalidDirection
newDir is **cc3DVect(0,0,0)**.

unitDir `cc3DVect unitDir() const;`

Gets the unit vector direction of the ray.

Throws
cc3DShapeDefs::DegenerateShape if this ray is degenerate.

line `cc3DLine line() const;`

Returns a **cc3DLine** object that includes the start point and has the same direction as this ray.

Operators

operator== `bool operator==(const cc3DRay& that) const;`

Return true if the supplied object is equal to this one, false otherwise.

Parameters
that The object to compare to this one.

■ cc3DRay

cc3DRect

```
#include <ch_c3d/shapes3d.h>

class cc3DRect:
   public cc3DShape,
   public cc3DVertex,
   public cc3DCurve,
   public cc3DSurface;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class represents a 3D rectangle.

Constructors/Destructors

cc3DRect

```
cc3DRect();

cc3DRect(const cc2Vect& size,
   const cc3DXformRigid& shapeFromScaledUnit,
   cc3DShapeDefs::StateType type);
```

•

```
cc3DRect();
```

Default-constructs a degenerate rectangle with the following values:

- **size()** is **cc2Vect(0,0)**.
- **shapeFromScaledUnit()** is identity.
- **stateType()** is *cc3DShapeDefs::eSurface*.

•

```
cc3DRect(const cc2Vect& size,
   const cc3DXformRigid& shapeFromScaledUnit,
   cc3DShapeDefs::StateType type);
```

Constructs a 3D rectangle using the supplied parameters. The *size* argument specifies the x- and y-dimensions of a scaled unit rectangle aligned to the x- and y-axes and with its minimum x- and y- values at the origin. The *shapeFromScaledUnit* rigid transform maps the rectangle to its 3D pose.

Parameters

size The size of the rectangle, as described above.

shapeFromScaledUnit
A rigid transformation describing the rectangle's pose in 3D.

type The state type. *type* must be one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface

Throws

cc3DShapeDefs::BadParams
The x- or y- dimension specified in *size* is less than 0.

Public Member Functions

clone

```
virtual cc3DShapePtrh clone () const;
```

This is an override from class **cc3DShape**.

isFinite

```
virtual bool isFinite () const;
```

This is an override from class **cc3DShape**.

isEmpty

```
virtual bool isEmpty () const;
```

This is an override from class **cc3DShape**.

nearestPoint

```
virtual cc3DVect nearestPoint (const cc3DVect &pt) const;
```

This is an override from class **cc3DShape**.

Parameters

pt The point.

boundingBox

```
virtual cc3DAlignedBox boundingBox() const;
```

This is an override from class **cc3DShape**.

mapShape

```
virtual cc3DShapePtrh mapShape (const cc3DXformBase &xform)
   const;

virtual void mapShape (const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const;
```

- ```
  virtual cc3DShapePtrh mapShape(const cc3DXformBase &xform)
     const;
  ```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters

xform	The transform with which to map.

- ```
  virtual void mapShape (const cc3DXformBase& xform,
     cc3DShapePtrh& dst) const;
  ```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters

xform	The transform with which to map.
dst	The transformed shape.

nearestPointVertex

```
virtual cc3DVect nearestPointVertex(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DVertex**.

Parameters

pt	The point.

distanceVertex

```
virtual double distanceVertex(const cc3DVect &pt) const;
```

This is an override from class **cc3DVertex**.

Parameters

pt	The point.

perimeter

```
virtual double perimeter() const;
```

This is an override from class **cc3DCurve**.

Regardless of whether the shape is degenerate, the perimeter is defined to be:

```
(size().x() + size.y()) * 2
```

nearestPointCurve

```
virtual cc3DVect nearestPointCurve(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DCurve**.

Parameters

pt The point.

area

```
virtual double area() const;
```

This is an override from class **cc3DSurface**.

nearestPointSurface

```
virtual cc3DVect nearestPointSurface(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DSurface**.

Parameters

pt The point.

stateType

```
virtual cc3DShapeDefs::StateType stateType() const;

void stateType(cc3DShapeDefs::StateType type);
```

- `virtual cc3DShapeDefs::StateType stateType() const;`

  Returns the state type of this object.

- `void stateType(cc3DShapeDefs::StateType type);`

  Sets the state type of this aligned box. The state type influences how various methods (such as **nearestPoint()**) inherited from **cc3DShape** class are interpreted.

  Parameters

  type The state type. *type* must be one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface

Notes

The default shape type is *cc3DShapeDefs::eSurface*.

Throws

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve
cc3DShapeDefs::eSurface

map

```
cc3DRect map(const cc3DXformRigid &xform) const;

void map(const cc3DXformRigid &xform, cc3DRect& dst) const;
```

- `cc3DRect map(const cc3DXformRigid &xform) const;`

Returns this shape mapped by the rigid transform xform.

Parameters

xform The transform to map with.

- `void map(const cc3DXformRigid &xform, cc3DRect& dst) const;`

Maps this shape by the rigid transform xform and places the result in the supplied object.

Parameters

xform The transform to map with.

dst The object in which to place the result.

getSizeAndShapeFromScaledUnit

```
void getSizeAndShapeFromScaledUnit (
   cc2Vect& size,cc3DXformRigid& shapeFromScaledUnit)
   const;
```

Gets the size of the rectangle and a rigid transform that maps the scaled unit rectangle to this one.

Parameters

size A **cc2Vect** giving the size of this rectangle.

shapeFromScaledUnit
A rigid transform that maps the scaled unit rectangle to this one.

setSizeAndShapeFromScaledUnit

```
void setSizeAndShapeFromScaledUnit (
   const cc2Vect& size,
   const cc3DXformRigid& shapeFromScaledUnit);
```

Sets this rectangle to the supplied parameters. The *size* argument specifies the x- and y-dimensions of a scaled unit rectangle aligned to the x- and y-axes and with its minimum x- and y- values at the origin. The *shapeFromScaledUnit* rigid transform maps the rectangle to its 3D pose.

rectangle from the scaled unit rectangle.

Parameters

size The size of the rectangle.

shapeFromScaledUnit
A rigid transform that maps the scaled unit rectangle to this one.

Throws

cc3DShapeDefs::BadParams
Either the x- or y-dimension of *size* is less than 0.

getCenterLengthVectorAndWidthVector

```
void getCenterLengthVectorAndWidthVector (
   cc3DVect& center, cc3DVect& lengthVector,
   cc3DVect& widthVector);
```

Gets the center point of this rectangle along with a pair of vectors giving the width and length of the rectangle.

Parameters

center The center point.

lengthVector A vector giving the orientation and size of the length dimension (x-dimension) of the rectangle.

widthVector A vector giving the orientation and size of the width dimension (y-dimension) of the rectangle.

Throws

cc3DShapeDefs::DegenerateShape
This rectangle is degenerate.

setCenterLengthVectorAndWidthVector

```
void setCenterLengthVectorAndWidthVector (
   const cc3DVect& center, const cc3DVect& lengthVector,
   const cc3DVect& widthVector);
```

Sets this rectangle based on the specified center point and length and width vectors.

If the supplied width and length vectors are not perpendicular, this function internally generates the width vector as follows:

1. Create a plane normal to the supplied length vector.
2. Project the supplied width vector onto this plane.
3. Scale the projected vector so that it has the same length as the supplied width. If the resulting vector has a length of zero (as would be the case if *lengthVector* and *widthVector* are parallel), an error is thrown.

Parameters

center The center point.

lengthVector A vector giving the orientation and size of the length dimension (x-dimension) of the rectangle.

widthVector A vector giving the orientation and size of the width dimension (y-dimension) of the rectangle.

Notes

shapeFromScaledUnit() will

- map the vector **cc3DVect (1,0,0)** to *lengthVector* scaled to a unit vector
- map the vector **cc3DVect(0,1,0)** to *widthVectorInternal* scaled to a unit vector,
- map the point **cc3Dvect(0,0,0)** to

  ```
  center - (lengthVector + widthVectorInternal)/2
  ```

  where *widthVectorInternal* is either the supplied width vector or the internally recomputed width vector described above.

Also **size().x()** will be equal to the length of *lenghtVector* and **size().y()** will be equal to the length of *widthVector.*

Throws

cc3DShapeDefs::BadParams
lengthVector or *widthVector* (or the internally generated width vector described above) is **cc3DVect(0,0,0)**.

cc3DShapeDefs::DegenerateShape
This rectangle is degenerate.

getOriginVertexLengthVectorAndWidthVector

```
void getOriginVertexLengthVectorAndWidthVector (
   cc3DVect& vertex, cc3DVect& lengthVector,
   cc3DVect& widthVector);
```

Gets the origin point of this rectangle along with a pair of vectors giving the width and length of the rectangle.

Parameters

vertex	The origin point.
lengthVector	A vector giving the orientation and size of the length dimension (x-dimension) of the rectangle.
widthVector	A vector giving the orientation and size of the width dimension (y-dimension) of the rectangle.

Throws

cc3DShapeDefs::DegenerateShape
This rectangle is degenerate.

setOriginVertexLengthVectorAndWidthVector

```
void setOriginVertexLengthVectorAndWidthVector (
   const cc3DVect& vertex, const cc3DVect& lengthVector,
   const cc3DVect& widthVector);
```

Sets this rectangle based on the specified origin point and length and width vectors.

If the supplied width and length vectors are not perpendicular, this function internally generates the width vector as follows:

1. Create a plane normal to the supplied length vector.
2. Project the supplied width vector onto this plane.
3. Scale the projected vector so that it has the same length as the supplied width. If the resulting vector has a length of zero (as would be the case if *lengthVector* and *widthVector* are parallel), an error is thrown.

Parameters

vertex	The origin point.
lengthVector	A vector giving the orientation and size of the length dimension (x-dimension) of the rectangle.
widthVector	A vector giving the orientation and size of the width dimension (y-dimension) of the rectangle.

Notes

shapeFromScaledUnit() will

- map the vector **cc3DVect (1,0,0)** to *lengthVector* scaled to a unit vector
- map the vector **cc3DVect(0,1,0)** to *widthVectorInternal* scaled to a unit vector,
- map the point **cc3Dvect(0,0,0)** to *vertex*

Also **size().x()** will be equal to the length of *lenghtVector* and **size().y()** will be equal to the length of *widthVector.* **center()** will return

```
vertex + (lengthVector + widthVectorInternal)/2
```

where *widthVectorInternal* is either the supplied width vector or the internally recomputed width vector described above.

Throws

cc3DShapeDefs::BadParams
lengthVector or *widthVector* (or the internally generated width vector described above) is **cc3DVect(0,0,0)**.

size

```
cc2Vect size() const;

void size(const cc2Vect& newSize);
```

- `cc2Vect size() const;`

Returns a **cc2Vect** giving the x- and y-dimensions of this rectangle.

- `void size(const cc2Vect& newSize);`

Sets the size of this rectangle.

The default size is **cc2vect(0,0)**.

Parameters

newSize The new size.

Throws

cc3DShapeDefs::BadParams
Either the x- or y-dimension specified in *newSize* is less than 0.

Notes

The setter does not change the origin vertex (which corresponds to **cc3DVect(0,0,0)** in the unit rectangle) but may change the value returned by **center()**.

setSizeAndKeepCenterUnchanged

```
void setSizeAndKeepCenterUnchanged(
   const cc2Vect& newSize);
```

Sets the size of this rectangle while keeping the center point unchanged.

Parameters

newSize The new size.

Throws

cc3DShapeDefs::BadParams

Either the x- or y-dimension specified in *newSize* is less than 0.

Notes

The setter does not change the center of this rectangle, but it may change the origin vertex (which corresponds to **cc3DVect(0,0,0)** in the unit rectangle). The setter may change the value returned by **shapeFromScaledUnit()**.

center

```
cc3DVect center() const;

void center(const cc3DVect& newCenter);
```

- `cc3DVect center() const;`

Returns the center point of this rectangle.

- `void center(const cc3DVect& newCenter);`

Sets the center point of this rectangle.

Parameters

newCenter The new center.

Notes

The setter may change the value returned by **shapeFromScaledUnit()**.

shapeFromScaledUnit

```
cc3DXformRigid shapeFromScaledUnit() const;

void shapeFromScaledUnit(const cc3DXformRigid& rigid);
```

- `cc3DXformRigid shapeFromScaledUnit() const;`

Returns the rigid transform that maps the scaled unit rectangle to this one.

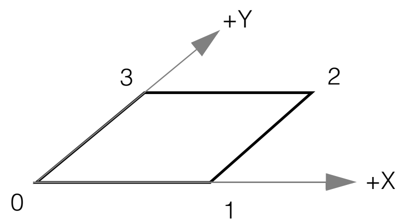
- `void shapeFromScaledUnit(const cc3DXformRigid& rigid);`

  Sets the rigid transform that maps the scaled unit rectangle to this one.

  Parameters
  rigid The transform.

vertices `cmStd vector<cc3DVect> vertices() const;`

Returns the vertices of this rectangle. The vertices are returned in the following order:



where vertex 0 corresponds to the origin vertex in the unit shape. More formally, the vertex order is given as follows, based on an (untransformed) unit square:

```
0   cc3DVect(0,0,0)
1   cc3DVect(1,0,0)
2   cc3DVect(1,1,0)
3   cc3DVect(0,1,0)
```

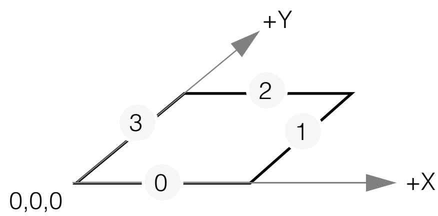
Notes

Some elements of the returned vector might be duplicate if this rectangle is degenerate.

lineSegs

```
cmStd vector<cc3DLineSeg> lineSegs() const;
```

Returns the line segments for the edges of this rectangle. The segments are returned in the following order:



where the vertex marked *0,0,0* corresponds to the origin vertex in the unit shape. More formally, the edge order is given as follows, based on an (untransformed) unit square:

```
0   cc3DLineSeg(cc3DVect(0,0,0), cc3DVect(1,0,0))
1   cc3DLineSeg(cc3DVect(1,0,0), cc3DVect(1,1,0))
2   cc3DLineSeg(cc3DVect(1,1,0), cc3DVect(0,1,0))
3   cc3DLineSeg(cc3DVect(0,1,0), cc3DVect(0,0,0))
```

Notes

Some elements of the returned vector might be degenerate line segments and some elements might be duplicate if this rectangle itself is degenerate.

Each element of the returned vector has its state type set to *cc3DShapeDefs::eCurve*, regardless of the state type of the **cc3DRect**.

Operators

operator==

```
bool operator==(const cc3DRect& that) const;
```

Returns true if this object is exactly equal to *that*, and false otherwise.

Parameters

that The object to compare to this one.

cc3DResiduals

```
#include <ch_c3d/residual.h>
```

```
class cc3DResiduals;
```

Class that holds residual error information. Residual error is the difference between an expected and a found value across a set of points. Error may be expressed as the maximum error (the largest difference) or the RMS error (the square root of the average of the differences squared).

Constructors/Destructors

cc3DResiduals

```
cc3DResiduals();
cc3DResiduals(double maximum, double rms);
```

- `cc3DResiduals();`

Constructs this object with all values set to zero.

- `cc3DResiduals(double maximum, double rms);`

Constructs this object with the specified values.

Parameters

maximum The maximum error.

rms The RMS error.

Throws

cc3DMathDefs::BadParams
maximum is less than *rms.*

Public Member Functions

maximum

```
double maximum() const;
void maximum(double m);
```

- `double maximum() const;`

Returns the maximum error.

- `void maximum(double m);`

Sets the maximum error.

Parameters

m The error.

rms

```
double rms() const;

void rms(double r);
```

- `double rms() const;`

Returns the RMS error.

- `void rms(double r);`

Sets the RMS error.

Parameters

r The error.

maximumAndRms

```
void maximumAndRms(double maximum, double rms);
```

Sets the maximum and RMS error values.

Parameters

maximum The maximum error.

rms The RMS error.

Throws

cc3DMathDefs::BadParams
maximum is less than *rms.*

Operators

operator==

```
bool operator== (const cc3DResiduals& rhs) const;
```

Return true if the supplied object is equal to this one, false otherwise.

Parameters

that The object to compare to this one.

cc3DSphere

```
#include <ch_c3d/shapes3d.h>

class cc3DSphere:
   public cc3DShape,
   public cc3DSurface,
   public cc3DVolume;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

This class represents a 3D sphere.

Constructors/Destructors

cc3DSphere

```
cc3DSphere();

cc3DSphere(double radius,
  const cc3DXformRigid& shapeFromScaledUnit,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eVolume);
```

•

```
cc3DSphere();
```

Default-constructs a degenerate sphere with these values:

- **radius()** is 0
- **shapeFromScaledUnit()** is identity
- **stateType()** is *cc3DShapeDefs::eVolume*

•

```
cc3DSphere(double radius,
  const cc3DXformRigid& shapeFromScaledUnit,
  cc3DShapeDefs::StateType type = cc3DShapeDefs::eVolume);
```

Constructs a sphere using the given radius, rigid transformation from the scaled unit sphere, and state type.

Parameters

radius The radius of the sphere.

shapeFromScaledUnit
A transformation from the scaled unit sphere.

type The state type. *type* must be one of the following values:

cc3DShapeDefs::eSurface
cc3DShapeDefs::eVolume

Throws

cc3DShapeDefs::BadParams
radius is less than 0.

cc3DShapeDefs::InvalidStateType
type is neither *cc3DShapeDefs::eSurface* nor *cc3DShapeDefs::eVolume*.

Public Member Functions

clone

```
virtual cc3DShapePtrh clone () const;
```

This is an override from class **cc3DShape**.

isFinite

```
virtual bool isFinite () const;
```

This is an override from class **cc3DShape**.

isEmpty

```
virtual bool isEmpty () const;
```

This is an override from class **cc3DShape**.

nearestPoint

```
virtual cc3DVect nearestPoint (const cc3DVect &pt) const;
```

This is an override from class **cc3DShape**.

Parameters

pt The point.

boundingBox

```
virtual cc3DAlignedBox boundingBox() const;
```

This is an override from class **cc3DShape**.

mapShape

```
virtual cc3DShapePtrh mapShape(const cc3DXformBase& xform)
   const;

virtual void mapShape (const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const;
```

- ```
  virtual cc3DShapePtrh mapShape(const cc3DXformBase& xform)
     const;
  ```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters

xform The transform with which to map.

- ```
  virtual void mapShape (const cc3DXformBase& xform,
     cc3DShapePtrh& dst) const;
  ```

Maps this shape with the supplied **cc3DXformBase**.

This is an override from class **cc3DShape**.

Parameters

xform The transform with which to map.

dst The transformed shape.

area

```
virtual double area() const;
```

This is an override from class **cc3DSurface**.

nearestPointSurface

```
virtual cc3DVect nearestPointSurface(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DSurface**.

Parameters

pt The point.

volume

```
virtual double volume() const;
```

This is an override from class **cc3DVolume**.

nearestPointVolume

```
virtual cc3DVect nearestPointVolume(const cc3DVect &pt)
   const;
```

This is an override from class **cc3DVolume**.

Parameters

pt The point.

map

```
cc3DSphere map(const cc3DXformRigid &xform) const;

void map(const cc3DXformRigid &xform, cc3DSphere& dst)
const;
```

- `cc3DSphere map(const cc3DXformRigid &xform) const;`

Returns the result of mapping this **cc3DSphere** by the supplied rigid transformation.

Parameters

xform The transformation.

- `void map(const cc3DXformRigid &xform, cc3DSphere& dst) const;`

Maps this shape by the rigid transform xform and places the result in the supplied object.

Parameters

xform The transform to map with.

dst The object in which to place the result.

stateType

```
virtual cc3DShapeDefs::StateType stateType() const;

void stateType(cc3DShapeDefs::StateType type);
```

- `virtual cc3DShapeDefs::StateType stateType() const;`

Returns the state type of this object.

- `void stateType(cc3DShapeDefs::StateType type);`

Sets the state type of this sphere. The state type influences how various methods (such as **nearestPoint()**) inherited from **cc3DShape** class are interpreted.

Parameters

type The state type. *type* must be one of the following values:

cc3DShapeDefs::eSurface
cc3DShapeDefs::eVolume

Notes

The default shape type is *cc3DShapeDefs::eVolume*.

Throws

cc3DShapeDefs::InvalidStateType
type is not one of the following values:

cc3DShapeDefs::eSurface
cc3DShapeDefs::eVolume

center

```
cc3DVect center() const;

void center(const cc3DVect& newCenter);
```

- ```
  cc3DVect center() const;
  ```

Returns the center of the sphere.

- ```
  void center(const cc3DVect& newCenter);
  ```

Sets the center of the sphere.

Parameters

newCenter The center point.

Notes

The setter might change the value returned by **shapeFromScaledUnit()**.

getRadiusAndShapeFromScaledUnit

```
void getRadiusAndShapeFromScaledUnit (
   double& r, cc3DXformRigid& shapeFromScaledUnit) const;
```

Gets the radius and rigid transformation from the scaled unit sphere that define this sphere.

Parameters

r The radius.

shapeFromScaledUnit
A rigid transform from the scaled unit sphere to this one.

setRadiusAndShapeFromScaledUnit

```
void setRadiusAndShapeFromScaledUnit (double r, const
cc3DXformRigid& shapeFromScaledUnit);
```

Sets this sphere to the supplied radius and rigid transformation from the scaled unit sphere.

Parameters

r The radius.

shapeFromScaledUnit
A rigid transform from the scaled unit sphere to this one.

Throws

cc3DShapeDefs::BadParams
radius is less than 0.

radius

```
double radius() const;

void radius(double r);
```

- `double radius() const;`

Returns the radius of this sphere.

- `void radius(double r);`

Sets the radius of this sphere.

The default radius is 0.

Parameters

r The radius to set.

Throws

cc3DShapeDefs::BadParams
radius is less than 0.

shapeFromScaledUnit

```
cc3DXformRigid shapeFromScaledUnit() const;
void shapeFromScaledUnit(const cc3DXformRigid& rigid);
```

- `cc3DXformRigid shapeFromScaledUnit() const;`

  Returns the rigid transform that maps the scaled unit sphere to this one.

- `void shapeFromScaledUnit(const cc3DXformRigid& rigid);`

  Sets the rigid transform that maps the scaled unit sphere to this one.

  The default transform is identity.

  Parameters
  rigid The transformation.

Operators

operator== `bool operator==(const cc3DSphere& that) const;`

Return true if the supplied object is equal to this one, false otherwise.

Parameters
that The object to compare to this one.

cc3DSphere

cc3DRotation

```
#include <ch_c3d/xform3d.h>
```

```
class cc3DRotation: public cc3DXformBase;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Complex

Main class for representing 3D rotations.

Constructors/Destructors

cc3DRotation

```
cc3DRotation();
cc3DRotation(const ccQuaternion& quaternion);
cc3DRotation(const cc3AngleVect& angleVect);
cc3DRotation(const cc3DEulerZYX& angles);
cc3DRotation(const cc3DEulerXYZ& angles);
cc3DRotation(const cc3DAxisAngle& axisAngle);
cc3DRotation(const cc3DMatrix& c);
```

- `cc3DRotation();`

Default constructor creates an identity transformation.

- `cc3DRotation(const ccQuaternion& quaternion);`

Constructs a **cc3DRotation** from the supplied quaternion.

Parameters

quaternion The quaternion.

Throws

cc3DMathDefs::ZeroQuaternion
quaternion has a norm of 0.

Notes

If *quaternion* is not a unit quaternion (norm of 1), it is scaled internally.

- `cc3DRotation(const cc3AngleVect& angleVect);`

Constructs a **cc3DRotation** from the supplied Euler angle vector.

Parameters

angleVect The Euler angle vector.

- `cc3DRotation(const cc3DEulerZYX& angles);`

Constructs a **cc3DRotation** from the supplied Euler ZYX angles.

Parameters

angles The Euler ZYX angles.

- `cc3DRotation(const cc3DEulerXYZ& angles);`

Constructs a **cc3DRotation** from the supplied Euler XYZ angles.

Parameters

angles The Euler XYZ angles.

- `cc3DRotation(const cc3DAxisAngle& axisAngle);`

Constructs a **cc3DRotation** from the supplied rotation axis and rotation angle.

Parameters

axisAngle The axis and angle.

Throws

cc3DMathDefs::InvalidAxis

axisAngle.axis is **cc3DVect(0,0,0)**.

Notes

If *axisAngle.axis* is not a unit vector, it is scaled internally.

- ```
  cc3DRotation(const cc3DMatrix& c);
  ```

Constructs a **cc3DRotation** from the supplied 3x3 rotation matrix.

It is a requirement that the supplied matrix be a rigid rotation, which implies that the determinant of the matrix must be 1 and that the composition of the matrix with its transposition be identity.

This requirement is not strictly enforced by this constructor: if the supplied matrix does not meet the requirement, the constructor attempts to compute a rigid rotation matrix that is close to the supplied matrix and uses that matrix internally.

Parameters
c The matrix.

Throws
cc3DMathDefs::NoRotationMatrix
c did not meet the requirements for a rigid rotation matrix and a substitute matrix could not be computed.

Public Member Functions

quaternion

```
ccQuaternion quaternion() const;

void quaternion(const ccQuaternion& quaternion);
```

- ```
  ccQuaternion quaternion() const;
  ```

Returns the quaternion representation of this rotation.

- ```
  void quaternion(const ccQuaternion& quaternion);
  ```

Sets this **cc3DRotation** to the rotation represented by the supplied quaternion.

If the supplied quaternion is not a unit quaternion, a corresponding unit quaternion is computed and used.

Parameters
quaternion The rotation to use.

Throws
cc3DDefs::ZeroQuaternion
The norm of *quaternion* is 0.
  ```

■ cc3DRotation

angleVect

```
cc3AngleVect angleVect() const;

void angleVect(const cc3AngleVect& vect);
```

- `cc3AngleVect angleVect() const;`

  Returns the Euler angle vector representation of this rotation.

- `void angleVect(const cc3AngleVect& vect);`

  Sets this **cc3DRotation** to the rotation represented by the supplied Euler angle vector.

  Parameters
  vect The rotation to use.

eulerXYZ

```
cc3DEulerXYZ eulerXYZ () const;

void eulerXYZ (const cc3DEulerXYZ& vect);
```

- `cc3DEulerXYZ eulerXYZ () const;`

  Returns the Euler XYZ angle representation of this rotation.

- `void eulerXYZ (const cc3DEulerXYZ& vect);`

  Sets this **cc3DRotation** to the rotation represented by the supplied Euler XYZ angle vector.

  Parameters
  vect The rotation to use.

eulerZYX

```
cc3DEulerZYX eulerZYX() const;

void eulerZYX (const cc3DEulerZYX& vect);
```

- `cc3DEulerZYX eulerZYX () const;`

  Returns the Euler ZYX angle representation of this rotation.

- `void eulerZYX (const cc3DEulerZYX& vect);`

  Sets this **cc3DRotation** to the rotation represented by the supplied Euler ZYX angle vector.

Parameters
vect The rotation to use.

axisAngle

```
cc3DAxisAngle axisAngle() const;
void axisAngle (const cc3DAxisAngle &axisAngle);
```

- `cc3DAxisAngle axisAngle() const;`

Returns the axis-angle representation of this rotation.

- `void axisAngle (const cc3DAxisAngle &axisAngle);`

Sets this **cc3DRotation** to the rotation represented by the supplied rotation axis and rotation angle.

Parameters
axisAngle The rotation to use.

Throws
cc3DMathDefs::InvalidAxis
axisAngle.axis is **cc3DVect(0,0,0)**.

Notes
If *axisAngle.axis* is not a unit vector, it is scaled internally.

matrix

```
cc3DMatrix matrix() const;
void matrix (const cc3DMatrix &c);
```

- `cc3DMatrix matrix() const;`

Returns the 3x3 matrix representation of this rotation.

- `void matrix (const cc3DMatrix &c);`

Sets this **cc3DRotation** to the rotation represented by the supplied 3x3 matrix.

It is a requirement that the supplied matrix be a rigid rotation, which implies that the determinant of the matrix must be 1 and that the composition of the matrix with its transposition be identity.

This requirement is not strictly enforced by this function: if the supplied matrix does not meet the requirement, the function attempts to compute a rigid rotation matrix that is close to the supplied matrix and uses that matrix internally.

Parameters

c The rotation to use.

Throws

cc3DMathDefs::NoRotationMatrix

c did not meet the requirements for a rigid rotation matrix and a substitute matrix could not be computed.

compose

```
cc3DRotation compose(const cc3DRotation& rhs) const;
```

Compose this **cc3DRotation** with the supplied one. The composition is from left to right.

Parameters

rhs The **cc3DRotation** to compose with this one.

inverse

```
cc3DRotation inverse () const;
```

Return the inverse of this **cc3DRotation**.

mapPoint

```
virtual cc3DVect mapPoint(const cc3DVect& pt) const;
```

Maps the given point through this **cc3DRotation**.

Parameters

pt The point.

invMapPoint

```
virtual cc3DVect invMapPoint(const cc3DVect& pt) const;
```

Maps the given point through the inverse of this **cc3DRotation**.

Parameters

pt The point.

mapPoints

```
virtual void mapPoints(
   const cmStd vector<cc3DVect>& points,
   cmStd vector<cc3DVect>& mappedPoints) const;
```

Maps the supplied points through this **cc3DRotation** and places them in the supplied value. The destination vector is resized if it is not the same size as the supplied source vector.

Parameters

points The points to map.

mappedPoints The mapped points.

invMapPoints

```
virtual void invMapPoints(
   const cmStd vector<cc3DVect>& points,
   cmStd vector<cc3DVect>& mappedPoints) const;
```

Maps the supplied points through the inverse of this **cc3DRotation** and places them in the supplied value. The destination vector is resized if it is not the same size as the supplied source vector.

Parameters

points The points to map.

mappedPoints The mapped points.

composeBase

```
virtual cc3DXformBasePtrh composeBase(
   const cc3DXformBase& rhs) const;
```

Return a 3D transform that is the composition of this **cc3DRotation** with the supplied transform. The composition is from left to right.

Parameters

rhs The transformation to compose with this **cc3DRotation**.

inverseBase

```
virtual cc3DXformBasePtrh inverseBase() const;
```

Return a 3D transform that is the inverse of this **cc3DRotation**.

clone

```
virtual cc3DXformBasePtrh clone() const;
```

Return a newly allocated copy of this **cc3DRotation**.

isIdentity

```
bool isIdentity() const;
```

Return true if this **cc3DRotation** is the identity transform, false otherwise.

Operators

operator*

```
cc3DRotation operator*(const cc3DRotation& rhs) const;
```

Convenience operator overload. Compose this cc3DRotation with the supplied one. The composition is from left to right.

operator==

```
bool operator==(const cc3DRotation& that) const;
```

Return true if the supplied object is equal to this one, false otherwise.

Parameters

that The object to compare to this one.

cc3DShape

```
#include <ch_c3d/shapes3d.h>

class cc3DShape:
   public virtual ccPersistent,
   public virtual ccRepBase;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	Complex

Base class for 3D shapes. Unless you intend to derive your own shapes, in most cases you should use the concrete shapes derived from this class.

Public Member Functions

clone

```
virtual cc3DShapePtrh clone () const =0;
```

Create a copy of this shape.

isFinite

```
virtual bool isFinite () const = 0;
```

Returns true if this shape has finite extent.

Notes

A **cc3DShape** has finite extent if it lies within a bounding box. Empty shapes are considered to have finite extent.

isEmpty

```
virtual bool isEmpty () const;
```

Returns true if the set of points that lie on the shape is empty.

nearestPoint

```
virtual cc3DVect nearestPoint(const cc3DVect &pt)
   const = 0;
```

Returns the nearest point on this shape to the given point. If the nearest point is not unique, one of the nearest points is returned.

The value returned for the nearest point depends on the value of **stateType()** for the concrete shape; the state type determines if the shapes surfaces, vertices, curves, or volume is used to compute the nearest point. Not all concrete shapes support all state types.

Parameters

pt The point.

Throws

cc3DShapeDefs::Empty3DShape
This shape is empty.

distance

```
virtual double distance(const cc3DVect &pt) const;
```

Returns the minimum distance from this shape to the supplied point. The nearest distance is the distance between the supplied point and the location returned by **nearestPoint()**.

The value returned for the nearest point depends on the value of **stateType()** for the concrete shape; the state type determines if the shapes surfaces, vertices, curves, or volume is used to compute the nearest point. Not all concrete shapes support all state types.

Parameters

pt The point.

Throws

cc3DShapeDefs::Empty3DShape
This shape is empty.

boundingBox

```
virtual cc3DAlignedBox boundingBox() const = 0;
```

Returns the bounding box of this shape.

Throws

cc3DShapeDefs::NotFinite
This shape is not finite.

cc3DShapeDefs::Empty3DShape
This shape is empty.

mapShape

```
virtual cc3DShapePtrh mapShape (const cc3DXformBase& xform)
   const =0;

virtual void mapShape (const cc3DXformBase& xform,
   cc3DShapePtrh& dst) const =0;
```

- ```
  virtual cc3DShapePtrh mapShape (const cc3DXformBase& xform)
     const =0;
  ```

Maps this shape with the supplied transformation and returns the result.

Parameters

xform The transform with which to map.

- ```
  virtual void mapShape (const cc3DXformBase& xform,
     cc3DShapePtrh& dst) const =0;
  ```

Maps this shape with the supplied transformation and places the result in the supplied shape pointer handle.

Parameters

xform The transform with which to map.

dst The transformed shape.

Notes

If the shape type of <dst> is not compatible with the mapped shape, a new shape will be created and assigned to *dst*.

stateType

```
virtual cc3DShapeDefs::StateType stateType() const = 0;
```

Returns the state type of the shape. The state type influences how various methods (such as **nearestPoint()**) are interpreted.

cc3DShapeDefs

```
#include <ch_c3d/shapes3d.h>
```

```
class cc3DShapeDefs;
```

A name space that holds enumerations and constants used with the 3D shapes.

Enumerations

StateType

```
enum StateType;
```

The state types for 3D shapes.The state type determines what point of the shape is used when computing the distance from the shape to another shape or point. Not all shapes support multiple state types, and different shapes support different types.

Value	Meaning
eVertex = 0x1	The shape is treated as a collection of vertices; the distance computation always uses a vertex of the shape.
eCurve = 0x2	The shape is treated as a curve; the distance computation always uses a point that lies on a curve.
eSurface = 0x4	The shape is treated as a surface; the distance computation always uses a point that lies on a surface of the shape.
eVolume = 0x8	The shape is treated as a volume; the distance computation always uses a point that lies within the volume of the shape.
eCollection = 0x10	The shape is treated as a shape collection.

cc3DShapeDefs

IntersectionStatus

```
enum IntersectionStatus;
```

Enumeration used by **cf3DIntersect()** to indicate how two shapes have intersected.

Value	Meaning
eIntersect = 0x1	A plane and line or plane and ray have a single intersection point. Two planes have a single intersection line.
eOverlap = 0x2	A plane and line or plane and ray or two planes are coincident.
eNone = 0x3	The shapes are parallel and do not intersect.

ProjectionStatus

```
enum ProjectionStatus;
```

Enumeration used by **cf3DProjectOntoPlane()** to indicate how a line or ray was projected onto a plane.

Value	Meaning
eStandardProjection = 0x1	The projection resulted in a line or ray.
eDegenerateProjection = 0x2	The projection resulted in a point (the line or ray was perpendicular to the plane).

cc3DShapeProjectParams

```
#include <ch_c3d/shapproj.h>
```

```
class cc3DShapeProjectParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Parameters class for shape projection.

Constructors/Destructors

cc3DShapeProjectParams

```
cc3DShapeProjectParams();
```

Public Member Functions

shapeReps

```
c_UInt32 shapeReps() const;
void shapeReps(c_UInt32 shapeReps);
```

- `c_UInt32 shapeReps() const;`

Returns the shape representations to project. The returned value is composed by ORing together one or more of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve

The default value for **shapeReps()** is *cc3DShapeDefs::eVertex* | *c3DShapeDefs::eCurve*.

- `void shapeReps(c_UInt32 shapeReps);`

Sets the shape representations to project. Only vertex and curve representations can be projected. To project multiple representations, OR their values together.

Parameters

shapeReps The representations to project. *shapeReps* must be formed by ORing together one or more of the following values:

cc3DShapeDefs::eVertex
cc3DShapeDefs::eCurve

Throws

ccShapeProjectDefs::BadParams
shapeReps is not a legal value.

Notes

Only the shape representations supported by the shape being projected are projected, regardless of the value specified here.

distortionToleranceRaw2D

```
double distortionToleranceRaw2D() const;

void distortionToleranceRaw2D(double tol);
```

- `double distortionToleranceRaw2D() const;`

Returns the distortion tolerance. This is the maximum Euclidean distance between the projected graphic and the true projected shape, in the units of raw 2D space.

- `void distortionToleranceRaw2D(double tol);`

Sets the distortion tolerance. This is the maximum Euclidean distance between the projected graphic and the true projected shape, in the units of raw 2D space. Smaller values produce more accurate projections while larger values use less memory and run faster.

The default value is 0.25.

Parameters

tol The tolerance in Raw 2D space units.

Throws

ccShapeProjectDefs::BadParams
tol is less than or equal to zero.

clipRectSource

```
cc3DShapeProjectDefs::ClipRectSource clipRectSource()
   const;

void clipRectSource(
   cc3DShapeProjectDefs::ClipRectSource src);
```

- ```
  cc3DShapeProjectDefs::ClipRectSource clipRectSource()
     const;
  ```

Returns the current source for the clipping rectangle. The returned value is one of the following:

cc3DShapeProjectDefs::eClipRectCalibration
cc3DShapeProjectDefs::eClipRectCustom

- ```
  void clipRectSource(
     cc3DShapeProjectDefs::ClipRectSource src);
  ```

Sets the source for the clipping rectangle. Only projected shapes within the clipping rectangle are added to the graphic list. To specify a custom clipping rectangle, you must set **clipRectSource** to *cc3DShapeProjectDefs::eClipRectCustom*.

If you specify *cc3DShapeProjectDefs::eClipRectCalibration*, clipping will occur at the limits of the image rectangle automatically obtained from the **ccCameraCalib** object passed into the projection function.

The default value is *cc3DShapeProjectDefs::eClipRectCalibration*.

Parameters

src — The clipping rectangle source. *src* must be one of the following values:

cc3DShapeProjectDefs::eClipRectCalibration
cc3DShapeProjectDefs::eClipRectCustom

■ cc3DShapeProjectParams

customClipRectRaw2D

```
const ccPelRect& customClipRectRaw2D() const;

void customClipRectRaw2D(ccPelRect& rect);
```

- `const ccPelRect& customClipRectRaw2D() const;`

Returns the current custom clipping rectangle. This rectangle is only used if **clipRectSource** is *cc3DShapeProjectDefs::eClipRectCustom*.

The default value is a **ccPelRect** with origin (0,0) and width and height of 0.

- `void customClipRectRaw2D(ccPelRect& rect);`

Sets the custom clipping rectangle. This rectangle is only used if **clipRectSource** is *cc3DShapeProjectDefs::eClipRectCustom*.The rectangle is specified in Raw 2D space.

Parameters

rect The clipping rectangle.

Notes

A convenient way to obtain an initial clipping rectangle is to call the **calibRoiRaw2D()** method of the same **cc3DCameraCalib** object that will be passed into the projection function. In fact, this is the rectangle that is automatically used when the **clipRectSource** is set to *cc3DProjefctionDefs::eClipRectCalibration*. This initial clipping rectangle can then be enlarged or reduced as desired to produce a custom clipping rectangle.

Throws

ccShapeProjectDefs::BadParams
The specified rectangle does not have a positive area.

Operators

operator== `bool operator==(const cc3DShapeProjectParams& other) const;`

Return true if the supplied object is equal to this one, false otherwise.

Parameters

other The object to compare to this one.

cc3DShapeProjectDefs

```
#include <ch_c3d/shapproj.h>
```

```
class cc3DShapeProjectDefs;
```

Namespace class containing enumerations for graphics projection.

Enumerations

ClipRectSource

```
enum ClipRectSource;
```

Enumeration defining the source for the clipping rectangle for graphics.

Value	Meaning
eClipRectCalibration = 0	Clip to the rectangle computed during camera calibration.
eClipRectCustom = 1	Use a user-specified rectangle.
kClipRectDefault	The default clipping rectangle source (*eClipRectCalibration).*

ClipStatus

```
enum ClipStatus;
```

Enumeration defining the type of clipping that occurred when the shapes were projected. Multiple clipping types may occur.

Value	Meaning
eClipNone = 0	No clipping occurred.
eClipZ = 0x1	The parts of the graphic that were behind the camera were clipped.
eClipXY = 0x2	The parts of the graphic that were outside of the clipping rectangle were clipped.

cc3DSurface

```
#include <ch_c3d/shapes3d.h>

class cc3DSurface:
   public virtual ccPersistent,
   public virtual ccRepBase;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	Complex

Base class that represents a 3D surface or a shape composed of a collection of 3D surfaces.

Public Member Functions

area

```
virtual double area() const = 0;
```

Returns the area of this surface.

Throws

cc3DShapeDefs::NotFinite
This shape is not finite.

Notes

This function returns 0 if this surface shape is empty or degenerate.

isDegenerateSurface

```
virtual bool isDegenerateSurface() const;
```

Returns true if this surface is degenerate (has an area of 0), false otherwise.

nearestPointSurface

```
virtual cc3DVect nearestPointSurface(const cc3DVect &pt)
   const = 0;
```

Returns the nearest point on this surface shape to the specified point. If the nearest point is not unique, one of the nearest points is returned.

Parameters

pt The point.

Throws

cc3DShapeDefs::*cc3DShapeDefs::Empty3DShape*
This shape is empty.

distanceSurface

```
virtual double distanceSurface(const cc3DVect &pt) const;
```

Returns the minimum distance from this surface to the supplied point.

Parameters

pt The point.

Throws

cc3DShapeDefs::*cc3DShapeDefs::Empty3DShape*
This shape is empty.

cc3DValidateCameraCalibDefs

```
#include <ch_c3d/calvalid.h>

class cc3DValidateCameraCalibDefs;
```

FeaturePositionsConstraints

```
enum FeaturePositionsConstraints;
```

An enumeration giving the constraint type to be applied when validating camera calibration.

Value	Meaning
eFeaturePositionsAccurateRelativePositions	The feature positions are accurately specified as relative positions.
kDefaultFeaturePositionsConstraints	The default constraint type (*eFeaturePositionsAccurateRelativePositions)*.

■ cc3DValidateCameraCalibDefs

cc3DValidateCameraCalibParams

```
#include <ch_c3d/calvalid.h>

class cc3DValidateCameraCalibParams;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Parameters class for 3D camera calibration validation.

Constructors/Destructors

cc3DValidateCameraCalibParams

```
cc3DValidateCameraCalibParams(
  cc3DValidateCameraCalibDefs::FeaturePositionsConstraints
  FeaturePositionsConstraints);
```

Parameters

FeaturePositionsConstraints

The constraint type. *FeaturePositionsConstraints* must be *cc3DValidateCameraCalibDefs::eFeaturePositionsAccurateRelativePositions*.

Public Member Functions

featurePositionsConstraints

```
cc3DValidateCameraCalibDefs::FeaturePositionsConstraints
   featurePositionsConstraints();

void featurePositionsConstraints (
  cc3DValidateCameraCalibDefs::FeaturePositionsConstraints
  featurePositionsConstraints);
```

•

```
cc3DValidateCameraCalibDefs::FeaturePositionsConstraints
   featurePositionsConstraints();
```

Returns the current constraint type.

-
```
void featurePositionsConstraints (
  cc3DValidateCameraCalibDefs::FeaturePositionsConstraints
  featurePositionsConstraints);
```

Sets the feature position constraint type.

Parameters

featurePositionsConstraints

The constraint type. *featurePositionsConstraints* must be *cc3DValidateCameraCalibDefs::eFeaturePositionsAccurateRelativePositions*.

Throws

cc3DValidateCameraCalibDefs::BadParams

featurePositionConstraints is not a valid member of **cc3DValidateCameraCalibDefs::FeaturePositionsConstraints.**

cc3DValidateCameraCalibResult

```
#include <ch_c3d/calvalid.h>

class cc3DValidateCameraCalibResult;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Results class for 3D camera calibration validation.

Constructors/Destructors

cc3DValidateCameraCalibResult

```
cc3DValidateCameraCalibResult();
```

Constructs a result object with no data.

Public Member Functions

cameraCalibResult

```
const cc3DCameraCalibResult &cameraCalibResult();
```

Returns a **cc3DCameraCalibResult** that provides calibration results for all cameras and all plate poses provided to the camera calibration function. You can use the returned object obtain information about the calibration for each camera and plate pose.

Throws

cc3DCameraCalibDefs::NotComputed
This is a default-constructed object.

residualsRMSStatisticsRaw2D

```
ccStatistics residualsRMSStatisticsRaw2D() const;
```

Returns a `ccStatistics` that provides RMS error residuals in 2D image units for all cameras and plate poses used for validation.

The **residualsRMSStatisticsRaw2D().rms()** and **residualsRMSStatisticsPhys3D.rms()** values provide a single numerical value that you can use to characterize the calibration quality. Cognex recommends that you compute and store these values immediately after initial system calibration and then periodically revalidate and recompute the values, comparing them with the baseline values.

Notes

The number of samples in the returned **ccStatistics** result is equal to the number of cameras times the number of plate poses.

Throws

cc3DCameraCalibDefs::NotComputed
This is a default-constructed object.

residualsRMSStatisticsPhys3D

```
ccStatistics residualsRMSStatisticsPhys3D() const;
```

Returns a `ccStatistics` that provides RMS error residuals in 3D physical units for all cameras and plate poses used for validation.

The **residualsRMSStatisticsRaw2D().rms()** and **residualsRMSStatisticsPhys3D.rms()** values provide a single numerical value that you can use to characterize the calibration quality. Cognex recommends that you compute and store these values immediately after initial system calibration and then periodically revalidate and recompute the values, comparing them with the baseline values.

Notes

The number of samples in the returned **ccStatistics** result is equal to the number of cameras times the number of plate poses.

Throws

cc3DCameraCalibDefs::NotComputed
This is a default-constructed object.

isComputed

```
bool isComputed() const;
```

Returns true if this object has computed results, false if it is default-constructed.

Operators

operator==

```
bool operator==(const cc3DValidateCameraCalibResult& that)
   const;
```

Returns true if this object has the same numerical values as the supplied object.

Parameters

that The object to compare to this one.

cc3DValidateCameraCalibResultSet

```
#include <ch_c3d/calvalid.h>
```

```
class cc3DValidateCameraCalibResultSet;
```

Class Properties

Copyable	Yes
Derivable	No
Archiveable	Simple

Results contain class for 3D camera calibration validation. This class contains two **cc3DValidateCameraCalibResult** objects. One object validates the calibration based on the camera extrinsic parameters in the calibration object being validated, the other recomputes the camera extrinsics based on the new correspondence data.

Constructors/Destructors

cc3DValidateCameraCalibResultSet

```
cc3DValidateCameraCalibResultSet();
```

Constructs a result object with no data.

Public Member Functions

isComputed

```
bool isComputed() const;
```

Returns true if this object has computed results, false if it is default-constructed.

validationResultsUsingOriginalCalibs

```
const cc3DValidateCameraCalibResult
   &validationResultsUsingOriginalCalibs() const;
```

Get the validation results corresponding to using the original camera calibrations

Throws

cc3DCameraCalibDefs::NotComputed
This is a default-constructed object.

validationResultsUsingRecomputedExtrinsics

```
const cc3DValidateCameraCalibResult
   &validationResultsUsingRecomputedExtrinsics() const;
```

Get the validation results corresponding to using the extrinsics based on the given features. The intrinsics from the calibration being validated are preserved and used.

Throws

cc3DCameraCalibDefs::NotComputed
This is a default-constructed object.

Operators

operator==

```
bool operator==(const
   cc3DValidateCameraCalibResultSet& that) const;
```

Returns true if this object has the same numerical values as the supplied object.

Parameters

that The object to compare to this one.

cc3DVolume

```
#include <ch_c3d/shapes3d.h>

class cc3DVolume:
   public virtual ccPersistent,
   public virtual ccRepBase;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	Complex

Base class that represents a 3D volume or a shape composed of a collection of 3D volumes.

Public Member Functions

volume

```
virtual double volume() const = 0;
```

Returns the volume.

Notes

This function returns 0 if this volume shape is empty or degenerate.

isDegenerateVolume

```
bool isDegenerateVolume() const;
```

Returns true if this volume is degenerate (has a volume of 0), false otherwise.

nearestPointVolume

```
virtual cc3DVect nearestPointVolume(const cc3DVect &pt)
   const = 0;
```

Returns the nearest point on this volume to the specified point. If the nearest point is not unique, one of the nearest points is returned.

Parameters

pt The point.

Throws

cc3DShapeDefs::*cc3DShapeDefs::Empty3DShape*
This shape is empty.

distanceVolume `virtual double distanceVolume(const cc3DVect &pt) const;`

Returns the minimum distance from this volume to the supplied point.

Parameters

pt The point.

Throws

cc3DShapeDefs::*cc3DShapeDefs::Empty3DShape*
This volume is empty.

cc3DVertex

```
#include <ch_c3d/shapes3d.h>

class cc3DVertex:
   public virtual ccPersistent,
   public virtual ccRepBase;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	Complex

Base class that represents a 3D vertex or a shape composed of a collection of 3D vertices.

Public Member Functions

nearestPointVertex

```
virtual cc3DVect nearestPointVertex(const cc3DVect &pt)
   const = 0;
```

Returns the nearest point on this vertex shape to the specified point. If the nearest point is not unique, one of the nearest points is returned.

Parameters

pt The point.

Throws

cc3DShapeDefs::*cc3DShapeDefs::Empty3DShape*
This shape is empty.

distanceVertex

```
virtual double distanceVertex(const cc3DVect &pt) const;
```

Returns the minimum distance from this vertex to the supplied point.

Parameters

pt The point.

Throws

cc3DShapeDefs::*cc3DShapeDefs::Empty3DShape*
This shape is empty.

■ cc3DVertex

cc3DXformBase

```
#include <ch_c3d/xform3d.h>

class cc3DXformBase:
   public ccPersistent,
   public ccRepBase;
```

Class Properties

Copyable	No
Derivable	Yes
Archiveable	Complex

Base class for 3D transformations.

Public Member Functions

mapPoint

```
virtual cc3DVect mapPoint(const cc3DVect& pt) const = 0;
```

Maps the given point.

Parameters

pt The point.

Notes

This maps the vector like a full 3D-point, with location as well as length and direction.

mapPoints

```
virtual void mapPoints(
   const cmStd vector<cc3DVect>& points,
   cmStd vector<cc3DVect>& mappedPoints) const = 0;
```

Maps the given points and places them in the supplied vector. If the output vector's size does not match the input vector's, it is resized.

Parameters

points The points to map.

mappedPoints The mapped points.

composeBase

```
virtual cc3DXformBasePtrh composeBase(
   const cc3DXformBase& rhs) const;
```

Returns a transformation that is the composition of this transformation with the supplied transformation. The composition order is left to right.

Parameters

rhs The transformation to compose with this one.

Notes

The run-time type of the returned result is not guaranteed to be the same across different CVL releases, so do not write code that depends on the exact run-time type (e.g. avoid using **dynamic_cast**).

In addition to this method which returns a pointer handler for a heap object, each derived class can have a **compose()** method which returns by value (not heap allocated), i.e. it takes a particular type of transformation and returns a particular type of transformation. For example,

```
cc3DRotation cc3DRotation::compose(
  const cc3DRotation& rhs) const
```

inverseBase

```
virtual cc3DXformBasePtrh inverseBase() const = 0;
```

Returns a transformation that is the inverse of this transformation.

Notes

It is expected that each derived class will define an **inverse()** function which returns a non-heap-allocated transformation of the appropriate type, e.g.

```
cc3DRotation cc3DRotation::inverse() const
```

Throws

ccMathError::Singular
This transform cannot be inverted because of singularity.

clone

```
virtual cc3DXformBasePtrh clone() const = 0;
```

Returns a newly allocated copy of this object.

operator*

```
cc3DVect operator*(const cc3DVect& point) const;

cc3DXformBasePtrh operator*(const cc3DXformBase& rhs)
   const;
```

- `cc3DVect operator*(const cc3DVect& point) const;`

  Maps the supplied point through this transformation.

- `cc3DXformBasePtrh operator*(const cc3DXformBase& rhs) const;`

  Returns the composition of the supplied transformation with this one. The order of composition is left to right.

cc3DXformRigid

```
#include <ch_c3d/xform3d.h>

class cc3DXformRigid:
   public cc3DXformBase;
```

Class Properties

Copyable	No
Derivable	No
Archiveable	Complex

Class that represents a rigid 3D transformation. A rigid transformation has only rotation and translation; it does not permit any scale change. A rigid transformation preserves distances; the distance between any two points is unchanged if the two points are transformed by the same rigid transformation.

Note The **cc3DXformRigid**'s order of operation is rotation followed by translation.

Constructors/Destructors

cc3DXformRigid

```
cc3DXformRigid();

cc3DXformRigid(const cc3DRotation& rotation,
   const cc3DVect& translation);
```

- `cc3DXformRigid();`

Constructs the identity transform.

- ```
  cc3DXformRigid(const cc3DRotation& rotation,
     const cc3DVect& translation);
  ```

Constructs a 3D rigid transform comprising the supplied rotation and translation.

Parameters

rotation The rotation component of the transformation.

translation The translation component of the transformation.
  ```

Public Member Functions

clone

```
virtual cc3DXformBasePtrh clone() const = 0;
```

Returns a newly allocated copy of this object.

composeBase

```
virtual cc3DXformBasePtrh composeBase(
   const cc3DXformBase& rhs) const;
```

Returns a transformation which is the composition of this one with the supplied transformation. The order of composition is left to right.

Parameters

rhs The transformation to compose with this one.

inverseBase

```
virtual cc3DXformBasePtrh inverseBase() const;
```

Returns a transformation that is the inverse of this one.

isIdentity

```
bool isIdentity() const;
```

Returns true if this transformation is exactly identity (identical to a default-constructed object).

rotation

```
cc3DRotation rotation() const;

void rotation(const cc3DRotation& r);
```

- ```
  cc3DRotation rotation() const;
  ```

  Returns the rotation component of this transformation.

- ```
  void rotation(const cc3DRotation& r);
  ```

  Sets the rotation component of this transformation.

  The default rotation is identity.

  Parameters

  r The rotation component.

trans

```
cc3DVect trans() const;
void trans(const cc3DVect& t);
```

- `cc3DVect trans() const;`

Returns the translation component of this transformation.

- `void trans(const cc3DVect& t);`

Sets the translation component of this transformation.

The default translation is **cc3DVect(0,0,0)**.

Parameters

t The translation component.

compose

```
cc3DXformRigid compose(const cc3DXformRigid& rhs) const;
```

Returns a transformation which is the composition of this one with the supplied transformation. The order of composition is left to right.

Parameters

rhs The transformation to compose with this one.

inverse

```
cc3DXformRigid inverse() const;
```

Returns a transformation that is the inverse of this one.

mapVector

```
cc3DVect mapVector(const cc3DVect& vect) const;
```

Rotates the supplied vector using the rotation component of this transformation.

Parameters

vect The vector to map.

invMapVector

```
cc3DVect invMapVector(const cc3DVect& vect) const;
```

Rotates the supplied vector using the rotation component of the inverse of this transformation.

Parameters

vect The vector to map.

mapVectors

```
void mapVectors(const cmStd vector<cc3DVect>& vects,
   cmStd vector<cc3DVect>& mappedVects) const;
```

Rotates the supplied vectors using the rotation component of this transformation, then stores the results in the supplied vector.

Parameters

vects The vectors to map.

mappedVects The mapped vectors. *mappedVects* is resized if it is a different size than *vects*.

invMapVectors

```
void invMapVectors(const cmStd vector<cc3DVect>& vects,
   cmStd vector<cc3DVect>& mappedVects) const;
```

Rotates the supplied vectors using the rotation component of the inverse of this transformation, then stores the results in the supplied vector.

Parameters

vects The vectors to map.

mappedVects The mapped vectors. *mappedVects* is resized if it is a different size than *vects*.

mapPoint

```
virtual cc3DVect mapPoint(const cc3DVect& pt) const;
```

Returns the result of mapping the supplied point through this transformation.

Parameters

pt The point.

invMapPoint

```
cc3DVect invMapPoint(const cc3DVect& pt) const;
```

Returns the result of mapping the supplied point through the inverse of this transformation.

Parameters

pt The point.

mapPoints

```
virtual void mapPoints(
   const cmStd vector<cc3DVect>& points,
   cmStd vector<cc3DVect>& mappedPoints) const;
```

Maps the supplied points with this transformation, then stores the results in the supplied vector.

Parameters

points	The points to map.
mappedPoints	The mapped points. *mappedPoints* is resized if it is a different size than *points*.

invMapPoints

```
void invMapPoints(const cmStd vector<cc3DVect> & points,
   cmStd vector<cc3DVect> & mappedPoints) const;
```

Maps the supplied points with the inverse of this transformation, then stores the results in the supplied vector.

Parameters

points	The points to map.
mappedPoints	The mapped points. *mappedPoints* is resized if it is a different size than *points*.

Operators

operator==

```
bool operator==(const cc3DRotation& that) const;
```

Returns true if the supplied object is identical to this one, false otherwise.

Parameters

that	The object to compare to this one.

operator*

```
cc3DXformRigid operator*(const cc3DXformRigid& rhs) const;
```

Convenience overload. Composes the supplied transformation with this one. The order of composition is left to right.

Parameters

rhs	The transformation to compose with this one.

cf3DCalibrateCameras()

```
#include <ch_c3d/cc3dcalib.h>
```

```
cf3DCalibrateCameras();
```

Global function to perform 3D camera calibration for one or more cameras. The following general requirements apply to **cf3DCalibrateCameras()**:

- All of the calibrate plate feature information provided to this function must be acquired using fixed cameras.
- All of the calibrate plate feature information provided to this function must be expressed using the same physical units. Cognex recommends using the same calibration plate for all images and viewsets.
- The optical configuration must be the same for all image acquisitions.

cf3DCalibrateCameras

```
void cf3DCalibrateCameras(
   const cmStd vector<cc3DCameraCalibFeatures>
   &calibrationPlatePoseFeatures,
   const cc3DCameraCalibParams &params,
   cc3DCameraCalibResult &result);

void cf3DCalibrateCameras(
   const cmStd vector<cc3DCameraCalibFeatures>
   &calibrationPlatePoseFeatures,
   const cmStd vector<ccCalib2ParamsIntrinsic>
   &intrinsicParams, cc3DCameraCalibResult &result);
```

-
```
void cf3DCalibrateCameras(
   const cmStd vector<cc3DCameraCalibFeatures>
   &calibrationPlatePoseFeatures,
   const cc3DCameraCalibParams &params,
   cc3DCameraCalibResult &result);
```

Compute the camera calibration (from the given vector of calibration features) and store the result in the supplied **cc3DCameraCalibResult** object.

This function can handle cameras of different camera intrinsics, such as different resolutions and cameras with different focal length lenses.

This function handles calibration features in 3-dimensional physical positions.

Parameters

calibrationPlatePoseFeatures

A vector **cc3DCameraCalibFeatures** objects. Each element of the vector contains feature correspondence pairs and weights for a single calibration plate pose as viewed from multiple cameras. The camera indexing must be the same in each element of the vector. This order is known as *camera index order*. The order of the elements in *calibrationPlatePoseFeatures* is known as the *plate pose index*.

params Parameters for the calibration.

result A **cc3DCameraCalibResult** object into which the result of the calibration is placed.

Notes

The tool can tolerate it if the plate pose defining world coordinates does not include features for some of the cameras - as long as the camera poses can be induced from other images.

The tool can tolerate some of the correspondences being empty so long as there are other correspondences which allow the camera calibrations to be computed.

The tool may take a relatively long time to run. This tool supports CVL timeouts.

This function minimizes the weighted sum squared error in image coordinates. Therefore, if different cameras have different pixel resolutions, then the calibration may be dominated by the higher resolution cameras. Consequently, this function is most useful when all cameras have similar pixel resolution.

Throws

cc3DCameraCalibDefs::BadParams

Either none of more than one of the calibration features has type *cc3DCameraCalibDefs::ePoseDefineWorldCoord*; not all of the elements in the calibration features vector characterize the same number of cameras (each element of *calibrationPlatePoseFeatures* must contain the same number of **ccCrspPairWeightedVector**s); none of the features are of type *cc3DCameraCalibDefs::ePoseTilted*; the z position associated with the *cc3DCameraCalibDefs::ePoseDefineWorldCoord* pose is non-zero; or the same number of region-of-interest rectangles is not specified for each plate pose.

cc3DCameraCalibDefs::InvalidRegionOfInterest

At least one element *calibrationPlatePoseFeatures* contains an empty *calibRoiRaw2Ds* pelRects are empty

cc3DCameraCalibDefs::TooFewCorrespondences

There are too few sets of correspondences for any of the cameras.

cc3DCameraCalibDefs::Singular
The input data are degenerate; the calibration cannot be computed.

•

```
void cf3DCalibrateCameras(
   const cmStd vector<cc3DCameraCalibFeatures>
   &calibrationPlatePoseFeatures,
   const cmStd vector<ccCalib2ParamsIntrinsic>
   &intrinsicParams, cc3DCameraCalibResult &result);
```

Compute the camera calibration (from the given vector of calibration features and camera intrinsic parameters) and store the result in the supplied **cc3DCameraCalibResult** object.

In this overload, the calibration function computes the camera poses and the calibration plate poses using the user-supplied intrinsics.

Parameters

calibrationPlatePoseFeatures
A vector **cc3DCameraCalibFeatures** objects. Each element of the vector contains feature correspondence pairs and weights for a single calibration plate pose as viewed from multiple cameras. The camera indexing must be the same in each element of the vector. This order is known as *camera index order*. The order of the elements in *calibrationPlatePoseFeatures* is known as the *plate pose index*.

intrinsicParams Intrinsic camera parameters for the cameras used to generate *calibrationPlatePoseFeatures*.

result A **cc3DCameraCalibResult** object into which the result of the calibration is placed.

Notes

The tool can tolerate it if the plate pose defining world coordinates does not include features for some of the cameras - as long as the camera poses can be induced from other images.

The tool can tolerate it if the plate pose defining world coordinates does not include features for some of the cameras - as long as the camera poses can be induced from other images.

The tool may take a relatively long time to run. This tool supports CVL timeouts.

This function minimizes the weighted sum squared error in image coordinates. Therefore, if different cameras have different pixel resolutions, then the calibration may be dominated by the higher resolution cameras. Consequently, this function is most useful when all cameras have similar pixel resolution.

The tool can tolerate it if the plate pose defining world coordinates does not include features for some of the cameras - as long as the camera poses can be induced from other images.

This overload, where the camera intrinsics are provided, can work with features from a single set of images. More images and more calibration plate poses will improve the accuracy of the calibration.

This function can handle cameras of different camera intrinsics such as different resolutions and cameras with different focal length lenses.

This function handles calibration features in 3-dimensional physical positions.

Throws

cc3DCameraCalibDefs::BadParams

An element of *calibrationPlatePoseFeatures* differs in size from *intrinsicParams*; either none of more than one of the calibration features has type *cc3DCameraCalibDefs::ePoseDefineWorldCoord*; not all of the elements in the calibration features vector characterize the same number of cameras (each element of *calibrationPlatePoseFeatures* must contain the same number of **ccCrspPairWeightedVector**s); none of the features are of type *cc3DCameraCalibDefs::ePoseTilted*; the z position associated with the *cc3DCameraCalibDefs::ePoseDefineWorldCoord* pose is non-zero; or the same number of region-of-interest rectangles is not specified for each plate pose.

cc3DCameraCalibDefs::InvalidRegionOfInterest

At least one element *calibrationPlatePoseFeatures* contains an empty *calibRoiRaw2Ds* pelRects are empty

cc3DCameraCalibDefs::TooFewCorrespondences
There are too few sets of correspondences for any of the cameras.

cc3DCameraCalibDefs::Singular
The input data are degenerate; the calibration cannot be computed.

■ cf3DCalibrateCameras()

cf3DComputePhys3DFromModel3DUsingPointsRaw2D()

```
#include <ch_c3d/cmp3dpos.h>
```

```
class cf3DComputePhys3DFromModel3DUsingPointsRaw2D();
```

Global function to compute the 3D pose of an object based on a set of 3D model points that define the object in 3D physical space, 2D image points from one or more 3D calibrated cameras that correspond to the 3D model points, and the 3D camera calibration objects for the cameras from which the images containing the 2D points were acquired.

The following information is common to all the overloads of this function:

- The 3D model points must be specified in the units used to create the 3D camera calibration (defined by the grid pitch of the calibration plate).
- The 3D model points define a “3D Model space;” the pose of this model space is returned.
- The 2D image points for a given camera view together with the 3D model points corresponding to those image points are provided in a **cc3DPointSet2D3D** object. A vector of **cc3DPointSet2D3D** is provided, one element for each camera. The **cc3DPointSet2D3D** vector is indexed by camera; this vector must be in the same order as the vector of camera calibration objects.

  Within each **cc3DPointSet2D3D**, the 2D point order must correspond to the 3D model point order. Note that each **cc3DPointSet2D3D** object may provide the locations of different sets of 3D model points.
- This function computes poses for both single- and multiple-camera systems. To successfully compute the 3D pose from 2D image points from a single camera, the points must be widely spaced across the image.
- All overloads support CVL timeouts.
- If you specify 2D image points from multiple cameras, all cameras must be calibrated to a common 3D physical space.

■ cf3DComputePhys3DFromModel3DUsingPointsRaw2D()

cf3DComputePhys3DFromModel3DUsingPointsRaw2D

```
cc3DXformRigid
   cf3DComputePhys3DFromModel3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc3DPointSet2D3D>&
   pointsRaw2DAndModel3D);

void cf3DComputePhys3DFromModel3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc3DPointSet2D3D>&
   pointsRaw2DAndModel3D,
   cc3DXformRigid& phys3DFromModel3D);

void cf3DComputePhys3DFromModel3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc3DPointSet2D3D>&
   pointsRaw2DAndModel3D,
   cc3DXformRigid& phys3DFromModel3D,
   cc3DResiduals& residualsRaw2D,
   cc3DResiduals& residualsPhys3D);
```

-
```
cc3DXformRigid
   cf3DComputePhys3DFromModel3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc3DPointSet2D3D>&
   pointsRaw2DAndModel3D);
```

Returns the 3D model pose from the supplied 2D image points, 3D model points, and camera calibration objects.

Parameters

raw2DFromPhys3Ds
The 3D camera calibration objects.

pointsRaw2DAndModel3D
The 2D image points and corresponding 3D model points from each camera.

Throws

cc3DPoseDefs::BadParams
Any weight value in *pointsRaw2DAndModel3D* is less than 0; the total number of valid 2D points is less than 3 (a valid 2D point has a nonzero weight); the number of camera calibration objects does not match the number of point sets; the number of 2D image points does not match the number of 3D model points for any camera; or the size of a non-empty weights vector does not match the number of 2D image points for any camera.

cc3DPoseDefs::Singular
The supplied data produces a degenerate solution (for example fewer than three non-collinear model points are provided)

cc3DPoseDefs::BehindCameras
The computed 3D position for any image point is behind the corresponding camera.

-
```
void cf3DComputePhys3DFromModel3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc3DPointSet2D3D>&
   pointsRaw2DAndModel3D,
   cc3DXformRigid& phys3DFromModel3D);
```

Computes the 3D model pose from the supplied 2D image points, 3D model points, and camera calibration objects and places it in the supplied **cc3DXformRigid**.

Parameters

raw2DFromPhys3Ds
The 3D camera calibration objects.

pointsRaw2DAndModel3D
The 2D image points and corresponding 3D model points from each camera.

phys3DFromModel3D
The computed 3D pose of the model.

Throws

cc3DPoseDefs::BadParams
Any weight value in *pointsRaw2DAndModel3D* is less than 0; the total number of valid 2D points is less than 3 (a valid 2D point has a nonzero weight); the number of camera calibration objects does not match the number of point sets; the number of 2D image points does not match the number of 3D model points for any camera; or the size of a non-empty weights vector does not match the number of 2D image points for any camera.

cc3DPoseDefs::Singular
The supplied data produces a degenerate solution (for example fewer than three non-collinear model points are provided)

cc3DPoseDefs::BehindCameras
The computed 3D position for any image point is behind the corresponding camera.

■ cf3DComputePhys3DFromModel3DUsingPointsRaw2D()

-
```
void cf3DComputePhys3DFromModel3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc3DPointSet2D3D>&
   pointsRaw2DAndModel3D,
   cc3DXformRigid& phys3DFromModel3D,
   cc3DResiduals& residualsRaw2D,
   cc3DResiduals& residualsPhys3D);
```

Computes the 3D model pose from the supplied 2D image points, 3D model points, and camera calibration objects and places it in the supplied **cc3DXformRigid**.

Both 2D and 3D residual error is computed. 2D error is computed using the 2D distance between the input points and the corresponding computed 3D point, mapped back to 2D image space. The 3D error is computed using the 3D distance between the computed 3D point and the 3D rays.

Parameters

raw2DFromPhys3Ds
The 3D camera calibration objects.

pointsRaw2DAndModel3D
The 2D image points and corresponding 3D model points from each camera.

phys3DFromModel3D
The computed 3D pose of the model.

residualsRaw2D The 2D residual error data.

residualsPhys3D The 3D residual error data.

Throws

cc3DPoseDefs::BadParams
Any weight value in *pointsRaw2DAndModel3D* is less than 0; the total number of valid 2D points is less than 3 (a valid 2D point has a nonzero weight); the number of camera calibration objects does not match the number of point sets; the number of 2D image points does not match the number of 3D model points for any camera; or the size of a non-empty weights vector does not match the number of 2D image points for any camera.

cc3DPoseDefs::Singular
The supplied data produces a degenerate solution (for example fewer than three non-collinear model points are provided)

cc3DPoseDefs::BehindCameras
The computed 3D position for any image point is behind the corresponding camera.

■ cf3DComputePhys3DFromModel3DUsingPointsRaw2D()

cf3DFitCircle()

```
#include <ch_c3d/fit3d.h>
```

```
cf3DFitCircle();
```

Global function to fit a 3D circle to a set of 3D points.

cf3DFitCircle

```
static inline cc3DCircle cf3DFitCircle(
   const cmStd vector<cc3DVect>& pts);

void cf3DFitCircle(const cmStd vector<cc3DVect>& pts,
   const cc3DCircleFitParams& params,
   cc3DCircleFitResult& result);
```

- ```
  static inline cc3DCircle cf3DFitCircle(
     const cmStd vector<cc3DVect>& pts);
  ```

Fit a 3D circle to the supplied set of 3D points. The supplied points must include three points that are not collinear.

Parameters

pts The points to fit.

Throws

ccMathError::Singular

Not enough non-collinear points were supplied. The minimum number is three.

Notes

The start point of the fitted **cc3DCircle** is not meaningful since the input data provide no basis for selecting the start point.

- ```
  void cf3DFitCircle(const cmStd vector<cc3DVect>& pts,
     const cc3DCircleFitParams& params,
     cc3DCircleFitResult& result);
  ```

Fit a 3D circle to the supplied set of 3D points using the supplied parameters. This overload allows you to specify robust fitting parameters, and it returns information about the fit including the residual errors.

Parameters

pts The points to fit.

params A **cc3DCircleFitParams** specifying fitting parameters.

result A **cc3DCircleFitResult** into which the fitting result is placed.

cf3DFitCircle()

Throws

ccMathError::Singular

Not enough non-collinear points were supplied. The minimum number is three.

Notes

The start point of the fitted **cc3DCircle** is not meaningful since the input data provide no basis for selecting the start point.

cf3DFitLine()

```
#include <ch_c3d/fit3d.h>
```

```
cf3DFitLine();
```

Global function to fit a 3D line to a set of 3D points.

cf3DFitLine

```
static inline cc3DLine cf3DFitLine(
   const cmStd vector<cc3DVect>& pts);

void cf3DFitLine(const cmStd vector<cc3DVect>& pts,
   const cc3DLineFitParams& params,
   cc3DLineFitResult& result);
```

•

```
static inline cc3DLine cf3DFitLine(
   const cmStd vector<cc3DVect>& pts);
```

Fit a 3D line to the supplied set of 3D points. The supplied points must include at least 2 distinct points.

Parameters

pts The points to fit.

Throws

ccMathError::Singular

Not enough distinct points were supplied. The minimum number is two.

Notes

The direction of the fitted **cc3DLine** is not meaningful since the input data provide no basis for specifying the direction.

•

```
void cf3DFitLine(const cmStd vector<cc3DVect>& pts,
   const cc3DLineFitParams& params,
   cc3DLineFitResult& result);
```

Fit a 3D line to the supplied set of 3D points using the supplied parameters. This overload allows you to specify robust fitting parameters, and it returns information about the fit including the residual errors.

Parameters

pts The points to fit.

params A **cc3DLineFitParams** specifying fitting parameters.

result A **cc3DLineFitResult** into which the fitting result is placed.

■ cf3DFitLine()

Throws

ccMathError::Singular

Not enough distinct points were supplied. The minimum number is two.

Notes

The direction of the fitted **cc3DLine** is not meaningful since the input data provide no basis for specifying the direction.

cf3DFitPhysA3DFromPhysB3D()

```
#include <ch_c3d/cmp3dpos.h>
```

```
class cf3DFitPhysA3DFromPhysB3D();
```

Global function to compute the 3D rigid transformation that maps one set of 3D points to another set of 3D points. The following information is common to all overloads of this function:

- The points in the two sets must correspond to each other.
- Both sets of points must be measured in the same units.
- Both sets of points must contain at least 3 non-collinear points.
- This function supports CVL timeouts.

cf3DFitPhysA3DFromPhysB3D

```
cc3DXformRigid cf3DFitPhysA3DFromPhysB3D(
   const cmStd vector<cc3DVect>& pointsPhysA3D,
   const cmStd vector<cc3DVect>& pointsPhysB3D);

void cf3DFitPhysA3DFromPhysB3D(
   const cmStd vector<cc3DVect>& pointsPhysA3D,
   const cmStd vector<cc3DVect>& pointsPhysB3D,
   cc3DXformRigid& physA3DFromPhysB3D);

void cf3DFitPhysA3DFromPhysB3D(
   const cmStd vector<cc3DVect>& pointsPhysA3D,
   const cmStd vector<cc3DVect>& pointsPhysB3D,
   cc3DXformRigid& physA3DFromPhysB3D,
   cc3DResiduals& residualsPhysA3DAndPhysB3D);
```

- ```
  cc3DXformRigid cf3DFitPhysA3DFromPhysB3D(
     const cmStd vector<cc3DVect>& pointsPhysA3D,
     const cmStd vector<cc3DVect>& pointsPhysB3D);
  ```

Computes and returns the rigid transformation that maps between the supplied point sets that minimizes the sum squared error.

Parameters

pointsPhysA3D The first point set.

pointsPhysB3D The second point set.
  ```

Throws

cc3DPoseDefs::BadParams
The two point sets contain different numbers of points, or either point set has less than 3 points.

cc3DPoseDefs::Singular
All points in either set are collinear.

•
```
void cf3DFitPhysA3DFromPhysB3D(
   const cmStd vector<cc3DVect>& pointsPhysA3D,
   const cmStd vector<cc3DVect>& pointsPhysB3D,
   cc3DXformRigid& physA3DFromPhysB3D);
```

Computes the rigid transformation that maps between the supplied point sets that minimizes the sum squared error and places it in the supplied parameter.

Parameters

pointsPhysA3D The first point set.

pointsPhysB3D The second point set.

physA3DFromPhysB3D
The computed transformation.

Throws

cc3DPoseDefs::BadParams
The two point sets contain different numbers of points, or either point set has less than 3 points.

cc3DPoseDefs::Singular
All points in either set are collinear.

•
```
void cf3DFitPhysA3DFromPhysB3D(
   const cmStd vector<cc3DVect>& pointsPhysA3D,
   const cmStd vector<cc3DVect>& pointsPhysB3D,
   cc3DXformRigid& physA3DFromPhysB3D,
   cc3DResiduals& residualsPhysA3DAndPhysB3D);
```

Computes the rigid transformation that maps between the supplied point sets that minimizes the sum squared error and places it in the supplied parameter. This overload also computes residual error. A single set of error statistics is computed; the residual error is the same regardless of whether the points from the first space are compared with the transformed points from the second space or the points from the second space are compared with the transformed points from the first space.

Parameters

pointsPhysA3D The first point set.

pointsPhysB3D The second point set.

physA3DFromPhysB3D
The computed transformation.

residualsPhysA3DAndPhysB3D
The computed residual error.

Throws

cc3DPoseDefs::BadParams
The two point sets contain different numbers of points, or either point set has less than 3 points.

cc3DPoseDefs::Singular
All points in either set are collinear.

■ cf3DFitPhysA3DFromPhysB3D()

cf3DFitPlane()

```
#include <ch_c3d/fit3d.h>
```

```
class cc3DPlaneFitResult
```

Global function to fit a 3D plane to a set of 3D points.

cf3DFitPlane

```
static inline cc3DPlane cf3DFitPlane(
   const cmStd vector<cc3DVect>& pts);

void cf3DFitPlane(const cmStd vector<cc3DVect>& pts,
   const cc3DPlaneFitParams& params,
   cc3DPlaneFitResult& result);
```

- ```
  static inline cc3DPlane cf3DFitPlane(
     const cmStd vector<cc3DVect>& pts);
  ```

Fit a 3D plane to the supplied set of 3D points. The supplied points must include at least 3 non-collinear points.

Parameters

pts The points to fit.

Throws

ccMathError::Singular

Not enough non-collinear points were supplied. The minimum number is three.

Notes

The normal direction of the fitted **cc3DPlane** is not meaningful since the input data provide no basis for specifying that direction.

- ```
  void cf3DFitPlane(const cmStd vector<cc3DVect>& pts,
     const cc3DPlaneFitParams& params,
     cc3DPlaneFitResult& result);
  ```

Fit a 3D plane to the supplied set of 3D points using the supplied parameters. This overload allows you to specify robust fitting parameters, and it returns information about the fit including the residual errors.

Parameters

pts The points to fit.

params A **cc3DPlaneFitParams** specifying fitting parameters.

result A **cc3DPlaneFitResult** into which the fitting result is placed.

■ cf3DFitPlane()

Throwss

ccMathError::Singular

Not enough non-collinear points were supplied. The minimum number is three.

Notes

The normal direction of the fitted **cc3DPlane** is not meaningful since the input data provide no basis for specifying that direction.

cf3DFitCircle3DUsingPoints2D()

```
#include <ch_c3d/fit2d.h>
```

```
cf3DFitCircle3DUsingPoints2D();
```

Global function to fit a 3D circle to 2D image points from one or more 3D-calibrated cameras. To fit a 3D circle to multiple *3D* image points, use the function **cf3DFitCircle()** on page 268.

cf3DFitCircle3DUsingPoints2D

```
void cf3DFitCircle3DUsingPoints2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cmStd vector<cc2Vect> >& pointsRaw2D,
   const cc3DCircleFit2DParams &params,
   cc3DCircleFit2DResultSet &resultSet);

static inline void cf3DFitCircle3DUsingPoints2D(
   const cc3DCameraCalib& raw2DFromPhys3D,
   const cmStd vector<cc2Vect>& pointsRaw2D,
   const cc3DCircleFit2DParams &params,
   cc3DCircleFit2DResultSet &resultSet);
```

•

```
void cf3DFitCircle3DUsingPoints2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cmStd vector<cc2Vect> >& pointsRaw2D,
   const cc3DCircleFit2DParams &params,
   cc3DCircleFit2DResultSet &resultSet);
```

Fits a 3D circle to the supplied set of 2D points. You can specify 2D points from any number of cameras, as long as all of the cameras are 3D calibrated to the same 3D physical space. The returned circle minimizes the sum squared error in image pixels across all of the supplied cameras.

If more than one 3D circle can be fit to the supplied sets of points with similar sum squared error, then both circles are returned. Otherwise, only the best fit circle is returned. The tool never returns more than two found circles.

Parameters

raw2DFromPhys3D
A vector **cc3DCameraCalib** objects, one for each calibrated camera from which 2D image points are to be fitted.

pointsRaw2D
A doubly-indexed vector containing the points to fit. *pointsRaw2D* is indexed first by the camera index (the same index used for *raw2DFromPhys3D*). For a given camera index,

the second index is used to access the individual points for that camera. At least one of the individual vectors of points must contain at least five points.

params A **cc3DCircleFit2DParams** object giving the parameters for the fit.

result A **cc3DCircleFit2DResultSet** object into which the result of the fitting is placed.

Notes

There is no requirement that the points from different cameras correspond to each other, nor that the same number of points be provided from each camera.

You must supply points from more than one camera if *params* specifies a fitting type of *cc3DCircleFit2DDefs::eLeastSquaresComputeRadius*.

Throws

cc3DCircleFit2DDefs::BadParams
raw2DFromPhys3D (the camera calibration vector) does not contain the same number of items as the outer vector of *pointsRaw2D* or *params* specifies an expected radius of zero and a fit mode of *cc3DCircleFit2DDefs::eLeastSquaresUseSpecifiedRadius*.

ccMathError::Singular
A circle cannot be fit to the supplied points.

cc3DCircleFit2DDefs::TooFewPoints
None of the vectors of individual points contains five or more points.

cc3DCircleFit2DDefs::CannotComputeRadiusFromSingleCamera
params specifies a fit mode of *cc3DCircleFit2DDefs::eLeastSquaresComputeRadius* and *pointsRaw2D* only contains points from one camera.

•
```
static inline void cf3DFitCircle3DUsingPoints2D(
   const cc3DCameraCalib& raw2DFromPhys3D,
   const cmStd vector<cc2Vect>& pointsRaw2D,
   const cc3DCircleFit2DParams &params,
   cc3DCircleFit2DResultSet &resultSet);
```

Convenience function when using a single camera. See the previous overload for description and throws.

Parameters

raw2DFromPhys3D
A **cc3DCameraCalib** object.

pointsRaw2D A vector containing the points to fit. *pointsRaw2D* must contain at least five points.

params A **cc3DCircleFit2DParams** object giving the parameters for the fit.

result A **cc3DCircleFit2DResultSet** object into which the result of the fitting is placed.

■ cf3DFitCircle3DUsingPoints2D()

cf3DHandEyeCalibration()

```
#include <ch_c3d/handeye.h>
```

```
cf3DHandEyeCalibration();
```

Global function to perform hand-eye calibration.

Note All overloads of **cf3DHandEyeCalibration()** support the use of CVL timeouts.

cf3DHandEyeCalibration

```
void cf3DHandEyeCalibration(
   const cc3DHandEyeCalibrationInputDataVector&
   inputDataVector,
   const cc3DHandEyeCalibrationRunParams& runParams,
   const cc3DCameraCalibParams& cameraCalibParams,
   cc3DHandEyeCalibrationResult& result,
   ccDiagObject* obj=0, c_UInt32 diagFlags=0);

void cf3DHandEyeCalibration(
   const cc3DHandEyeCalibrationInputDataVector&
   inputDataVector,
   const cc3DHandEyeCalibrationRunParams& runParams,
   const ccCalib2ParamsIntrinsic& intrinsicParams,
   cc3DHandEyeCalibrationResult& result,
   ccDiagObject* obj=0, c_UInt32 diagFlags=0);

void cf3DHandEyeCalibration(
   const cc3DHandEyeCalibrationInputDataVectorXO&
   inputDataVector,
   const cc3DHandEyeCalibrationRunParams& runParams,
   cc3DHandEyeCalibrationResultXO& result,
   ccDiagObject* obj=0, c_UInt32 diagFlags=0);
```

•

```
void cf3DHandEyeCalibration(
   const cc3DHandEyeCalibrationInputDataVector&
   inputDataVector,
   const cc3DHandEyeCalibrationRunParams& runParams,
   const cc3DCameraCalibParams& cameraCalibParams,
   cc3DHandEyeCalibrationResult& result,
   ccDiagObject* obj=0, c_UInt32 diagFlags=0);
```

This overload computes the hand-eye calibration *and* the camera intrinsic parameters at the same time using the image data obtained from each robot hand station.

In general, best results are obtained by computing the camera intrinsic parameters separately using standard 3D camera calibration, then supplying those intrinsic parameters to hand-eye calibration using the second overload of this function.

Parameters

inputDataVector The input data (robot hand pose and calibration plate feature data) for each robot hand station.

runParams The run parameters (the plate sampling parameters for computing residual error).

cameraCalibParams
The camera calibration parameters (the distortion model to use).

result The result of the calibration, including residual error data.

obj A diagnostics object.

diagFlags Diagnostics flags.

Throws

cc3DHandEyeCalibrationDefs::TooFewStations
inputDataVector contains fewer than 3 elements.

cc3DHandEyeCalibrationDefs::NoRotation
The motion between two adjacent stations for any camera has no detectable rotation.

cc3DHandEyeCalibrationDefs::RotationIs180
The motion between two adjacent stations for any camera has a rotation of 180°.

cc3DHandEyeCalibrationDefs::AllRotationsParallel
All the corresponding motions for any camera have rotation axes that are (effectively) parallel.

cc3DHandEyeCalibrationDefs::MotionInconsistent
The apparent motion between two adjacent stations -- as viewed by the corresponding camera -- is inconsistent with the motion reported by the robot. This can happen, for example, if the units used by the robot are different from the units used to describe the calibration plate.

cc3DHandEyeCalibrationDefs::TooFewFeatures
At least one station from *inputDataVector* does not have enough features for camera calibration.

cc3DHandEyeCalibrationDefs::DegenerateFeatures
The input image features are degenerate so that the camera calibration can not be computed

- ```
  void cf3DHandEyeCalibration(
     const cc3DHandEyeCalibrationInputDataVector&
     inputDataVector,
     const cc3DHandEyeCalibrationRunParams& runParams,
     const ccCalib2ParamsIntrinsic& intrinsicParams,
     cc3DHandEyeCalibrationResult& result,
     ccDiagObject* obj=0, c_UInt32 diagFlags=0);
  ```

This overload computes the hand-eye calibration from the supplied robot hand poses and image data using the supplied camera intrinsic parameters.

In general, best results are obtained by computing the camera intrinsic parameters separately using standard 3D camera calibration, then supplying those intrinsic parameters to hand-eye calibration.

Parameters

inputDataVector The input data (robot hand pose and calibration plate feature data) for each robot hand station.

runParams The run parameters (the plate sampling parameters for computing residual error).

intrinsicParams

The camera intrinsic parameters, as computed using standard 3D camera calibration.

result The result of the calibration, including residual error data.

obj A diagnostics object.

diagFlags Diagnostics flags.

Throws

cc3DHandEyeCalibrationDefs::TooFewStations

inputDataVector contains fewer than 3 elements.

cc3DHandEyeCalibrationDefs::NoRotation

The motion between two adjacent stations for any camera has no detectable rotation.

cc3DHandEyeCalibrationDefs::RotationIs180

The motion between two adjacent stations for any camera has a rotation of 180°.

cc3DHandEyeCalibrationDefs::AllRotationsParallel

All the corresponding motions for any camera have rotation axes that are (effectively) parallel.
  ```

cc3DHandEyeCalibrationDefs::MotionInconsistent
The apparent motion between two adjacent stations -- as viewed by the corresponding camera -- is inconsistent with the motion reported by the robot. This can happen, for example, if the units used by the robot are different from the units used to describe the calibration plate.

cc3DHandEyeCalibrationDefs::InvalidCameraDistortionModel
The camera distortion model specified in *intrinsicParams* is neither *cc2XformCalib2Defs::e3ParamRadial* nor *cc2XformCalib2Defs::eSineTanLawProjection.*

cc3DHandEyeCalibrationDefs::TooFewFeatures
At least one station from *inputDataVector* does not have enough features for camera calibration.

cc3DHandEyeCalibrationDefs::DegenerateFeatures
The input image features are degenerate so that the camera calibration can not be computed

- ```
  void cf3DHandEyeCalibration(
     const cc3DHandEyeCalibrationInputDataVectorXO&
     inputDataVector,
     const cc3DHandEyeCalibrationRunParams& runParams,
     cc3DHandEyeCalibrationResultXO& result,
     ccDiagObject* obj=0, c_UInt32 diagFlags=0);
  ```

This overload computes the hand-eye calibration from the supplied robot hand poses and extrinsic camera parameters.

This overload is intended for use with a calibrated multi-camera head. In this case, the cameras on the head are 3D calibrated in advance, then used to determine the pose of the calibration plate with regard to the multi-camera head at each robot hand station.

Parameters

inputDataVector	The input data (robot hand pose and extrinsic camera parameters) for each robot hand station.
runParams	The run parameters (the plate sampling parameters for computing residual error).
result	The result of the calibration, including residual error data.
obj	A diagnostics object.
diagFlags	Diagnostics flags.
  ```

Throws

cc3DHandEyeCalibrationDefs::TooFewStations

inputDataVector contains fewer than 3 elements.

cc3DHandEyeCalibrationDefs::NoRotation

The motion between two adjacent stations for any camera has no detectable rotation.

cc3DHandEyeCalibrationDefs::RotationIs180

The motion between two adjacent stations for any camera has a rotation of 180°.

cc3DHandEyeCalibrationDefs::AllRotationsParallel

All the corresponding motions for any camera have rotation axes that are (effectively) parallel.

cc3DHandEyeCalibrationDefs::MotionInconsistent

The apparent motion between two adjacent stations -- as viewed by the corresponding camera -- is inconsistent with the motion reported by the robot. This can happen, for example, if the units used by the robot are different from the units used to describe the calibration plate.

■ cf3DHandEyeCalibration()

cf3DProject3DCoordinateAxesTo2DGraphicList()

```
#include <ch_c3d/shapproj.h>
```

```
cf3DProject3DCoordinateAxesTo2DGraphicList();
```

Global function to project a 3D coordinate axes graphic to 2D image space for display.

Note For more information on graphic lists and graphic properties, see the CVL documentation for **ccGraphicProps** and **ccGraphicList**.

cf3DProject3DCoordinateAxesTo2DGraphicList

```
void cf3DProject3DCoordinateAxesTo2DGraphicList(
   const cc3DVect& lenAxes3D,
   const cc3DXformRigid& phys3DFromAxes3D,
   const cc3DCameraCalib& raw2DFromPhys3D,
   const cc3DShapeProjectParams& params,
   const ccGraphicProps& props, const ccCvlString& label,
   ccGraphicList& glist, c_UInt32& clipStatus);
```

Projects a graphical representation of a set of 3D coordinate axes into the 2D raw image space associated with the supplied 3D camera calibration object. The projected axes are rendered using the specified graphics properties (line color and style) and appended to the supplied graphics list. You specify the length of the axes in 3D physical space units, and you can specify a text legend. The axes are represented as arrow-less line segments of the lengths that you specify, labeled ‘x’, ‘y’, and ‘z’.

By default, the graphics are clipped to the 2D image rectangle that corresponds to the part of the image used to calibrate the camera. You can specify a different clipping rectangle. Any portion of the graphic that lies outside the clipping rectangle or behind the camera is clipped. The clipping status is written to the *clipStatus* parameter, which you can test using the **cc3DShapeProjectDefs::ClipStatus** enumeration.

This overload includes a parameter, *phs3DFromAxes3D*, that you use to specify the pose of the 3D coordinate axes in 3D physical space.

Parameters

lenAxes3D The length of the axes in 3D physical space units.

phys3DFromAxes3D
The pose of coordinate axes in 3D physical space.

raw2DFromPhys3D
The camera calibration object for the camera associated with the 2D image space upon which you wish to project the axes.

params Shape projection parameters. These parameters let you control how the axes are clipped.

props A graphic properties object that specifies the appearance of the projected graphics.

label	A text label to display with the coordinate axes. The text is drawn using the current default font ID. You can change the default font ID using the static function **ccUIFormat::defaltFontId()**.
glist	A graphics list onto which the graphics are appended.
clipStatus	An integer into which is written a set of bit flags indicating what, if any, clipping occurred when the graphics were projected. *clipStatus* is set to a value formed by ORing together one or more of the following values: *cc3DShapeProjectDefs::eClipNone* *cc3DShapeProjectDefs::eClipZ* *cc3DShapeProjectDefs::eClipXY*

Throws

cc3DShapeProjectDefs::BadParams

lenAxes3D has a negative or zero value for any dimension or *params* specifies the use of a custom clipping rectangle, but the area of the provided rectangle is not positive.

cf3DProject3DShapeTo2DGraphicList()

```
#include <ch_c3d/shapproj.h>
```

```
cf3DProject3DShapeTo2DGraphicList();
```

Global function to project 3D shapes onto a 2D image space for display.

Note For more information on graphic lists and graphic properties, see the CVL documentation for **ccGraphicProps** and **ccGraphicList**.

cf3DProject3DShapeTo2DGraphicList

```
void cf3DProject3DShapeTo2DGraphicList(
   const cc3DShape& shapeShape3D,
   const cc3DXformRigid& phys3DFromShape3D,
   const cc3DCameraCalib& raw2DFromPhys3D,
   const cc3DShapeProjectParams& params,
   const ccGraphicProps& props,
   ccGraphicList& glist, c_UInt32& clipStatus);

void cf3DProject3DShapeTo2DGraphicList(
   const cc3DShape& shapePhys3D,
   const cc3DCameraCalib& raw2DFromPhys3D,
   const cc3DShapeProjectParams& params,
   const ccGraphicProps& props,
   ccGraphicList& glist, c_UInt32& clipStatus);
```

•

```
void cf3DProject3DShapeTo2DGraphicList(
   const cc3DShape& shapeShape3D,
   const cc3DXformRigid& phys3DFromShape3D,
   const cc3DCameraCalib& raw2DFromPhys3D,
```

■ cf3DProject3DShapeTo2DGraphicList()

```
const cc3DShapeProjectParams& params,
const ccGraphicProps& props,
ccGraphicList& glist, c_UInt32& clipStatus);
```

Projects the supplied 3D shape into the 2D raw image space associated with the supplied 3D camera calibration object. The projected shape is rendered using the specified graphics properties (line color and style) and appended to the supplied graphics list.

By default, the graphics are clipped to the 2D image rectangle that corresponds to the part of the image used to calibrate the camera. You can specify a different clipping rectangle. Any portion of the graphic that lies outside the clipping rectangle or behind the camera is clipped. The clipping status is written to the *clipStatus* parameter, which you can test using the **cc3DShapeProjectDefs::ClipStatus** enumeration.

This overload includes a parameter, *phys3DFromShape3D*, that you use to specify the pose of the 3D shape in 3D physical space.

The 3D shape that you are projecting must support either the *cc3DShapeDefs::eCurve* or *cc3DShapeDefs::eVertex* shape type. You can specify whether the projected shape includes vertex points or curves, and you can specify how vertex points are rendered. The projection is performed without hidden line removal, and no line filling is supported. Shapes that support neither *cc3DShapeDefs::eCurve* nor *cc3DShapeDefs::eVertex* cannot be projected using this function.

Note The **stateType()** value for the shape is ignored during projection.

Parameters

shapeShape3D The shape to project.

phys3DFromShape3D
The pose of *shapeShape3D* in 3D physical space.

raw2DFromPhys3D
The camera calibration object for the camera associated with the 2D image space upon which you wish to project *shapeShape3D*.

params Shape projection parameters. These parameters let you control how the shape is clipped.

props A graphic properties object that specifies the appearance of the projected shape graphics.

glist A graphics list onto which the projected shape is appended.

clipStatus An integer into which is written a set of bit flags indicating what, if any, clipping occurred when the shape was projected. *clipStatus* is set to a value formed by ORing together one or more of the following values:

cc3DShapeProjectDefs::eClipNone
cc3DShapeProjectDefs::eClipZ
cc3DShapeProjectDefs::eClipXY

Throws

cc3DShapeProjectDefs::NotImplemented

The **fill** property of *props* is true (only unfilled curve and vertex representations of 3D shapes are supported) or *shapeShape3D* does not derive from **cc3DCurve** or **cc3DVertex**.

cc3DShapeProjectDefs::UnrecognizedShape

shapeShape3D is not recognized. That is, while derived from **cc3DCurve** or **cc3DVertex**, it is not a shape that can be rendered by this function.

cc3DShapeProjectDefs::ShapeNotCompatibleWithShapeReps

params does not specify a shape representation that is supported by *shapeShape3D.*

cc3DShapeProjectDefs::BadParams

params specifies the use of a custom clipping rectangle, but the area of the provided rectangle is zero.

•
```
void cf3DProject3DShapeTo2DGraphicList(
   const cc3DShape& shapePhys3D,
   const cc3DCameraCalib& raw2DFromPhys3D,
```

■ cf3DProject3DShapeTo2DGraphicList()

```
const cc3DShapeProjectParams& params,
const ccGraphicProps& props,
ccGraphicList& glist, c_UInt32& clipStatus);
```

Projects the supplied 3D shape into the 2D raw image space associated with the supplied 3D camera calibration object. The projected shape is rendered using the specified graphics properties (line color and style) and appended to the supplied graphics list.

By default, the graphics are clipped to the 2D image rectangle that corresponds to the part of the image used to calibrate the camera. You can specify a different clipping rectangle. Any portion of the graphic that lies outside the clipping rectangle or behind the camera is clipped. The clipping status is written to the *clipStatus* parameter, which you can test using the **cc3DShapeProjectDefs::ClipStatus** enumeration.

This overload does not allow you to specify the pose of the 3D shape in 3D physical space. You must either

- Transform the shape into the 3D physical space by calling the shape's **map()** method.

  or

- Modify the **cc3DCameraCalib** object to directly map from Shape3D space to Raw2D space by calling the camera calibration's **cloneComposeWithPhys3DFromAny3D()** method.

The 3D shape that you are projecting must support either the *cc3DShapeDefs::eCurve* or *cc3DShapeDefs::eVertex* shape type. You can specify whether the projected shape includes vertex points or curves, and you can specify how vertex points are rendered. The projection is performed without hidden line removal, and no line filling is supported. Shapes that support neither *cc3DShapeDefs::eCurve* nor *cc3DShapeDefs::eVertex* cannot be projected using this function.

Parameters

shapeShape3D The shape to project.

raw2DFromPhys3D
The camera calibration object for the camera associated with the 2D image space upon which you wish to project *shapeShape3D*.

params Shape projection parameters. These parameters let you control how the shape is clipped.

props A graphic properties object that specifies the appearance of the projected shape graphics.

glist A graphics list onto which the projected shape is appended.

clipStatus An integer into which is written a set of bit flags indicating what, if any, clipping occurred when the shape was projected. *clipStatus* is set to a value formed by ORing together one or more of the following values:

cc3DShapeProjectDefs::eClipNone
cc3DShapeProjectDefs::eClipZ
cc3DShapeProjectDefs::eClipXY

Throws

cc3DShapeProjectDefs::NotImplemented
The **fill** property of *props* is true (only unfilled curve and vertex representations of 3D shapes are supported) or *shapeShape3D* does not derive from **cc3DCurve** or **cc3DVertex**.

cc3DShapeProjectDefs::UnrecognizedShape
shapeShape3D is not recognized. That is, while derived from **cc3DCurve** or **cc3DVertex**, it is not a shape that can be rendered by this function.

cc3DShapeProjectDefs::ShapeNotCompatibleWithShapeReps
params does not specify a shape representation that is supported by *shapeShape3D.*

cc3DShapeProjectDefs::BadParams
params specifies the use of a custom clipping rectangle, but the area of the provided rectangle is zero.

Notes

The **stateType()** of the shape being projected is ignored.

■ cf3DProject3DShapeTo2DGraphicList()

cf3DTriangulatePointPhys3DUsingPointsRaw2D()

```
#include <ch_c3d/cmp3dpos.h>
```

```
class cf3DTriangulatePointPhys3DUsingPointsRaw2D();
```

Global function to compute the 3D position of a point from its corresponding 2D image points as viewed from multiple 3D calibrated cameras.

The following requirements are common to all the overloads of this function:

- You must provide 2D image points and camera calibration objects for at least two cameras.
- All of the cameras must have been calibrated as part of the same 3D camera calibration; all the cameras must share a common calibrated 3D physical space.
- The order of point sets and camera calibration objects must be the same.

cf3DTriangulatePointPhys3DUsingPointsRaw2D

```
cc3DVect cf3DTriangulatePointPhys3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc2Vect>& pointsRaw2D,
   const cmStd vector<bool>& isPointValid =
   cmStd vector<bool>());

void cf3DTriangulatePointPhys3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc2Vect>& pointsRaw2D,
   cc3DVect& pointPhys3D,
   const cmStd vector<bool>& isPointValid =
   cmStd vector<bool>());

void cf3DTriangulatePointPhys3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc2Vect>& pointsRaw2D,
   cc3DVect& pointPhys3D, cc3DResiduals& residualsRaw2D,
```

```
   cc3DResiduals& residualsPhys3D,
   const cmStd vector<bool>& isPointValid =
   cmStd vector<bool>());
```

- ```
  cc3DVect cf3DTriangulatePointPhys3DUsingPointsRaw2D(
     const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
     const cmStd vector<cc2Vect>& pointsRaw2D,
     const cmStd vector<bool>& isPointValid);
  ```

Computes the 3D position of a point from the corresponding 2D image locations of the point in two or more images from 3D calibrated cameras. You may supply an optional vector of **bool** to indicate which points are valid.

The computed fit minimizes the sum squared image error between the predicted and actual 2D point locations.

Parameters

raw2DFromPhys3Ds
A vector of 3D camera calibration objects.

pointsRaw2D A vector of 2D image points.

isPointValid An vector indicating which elements of *pointsRaw2D* are valid and should be used for the fitting operation. If an empty vector is supplied, all points are assumed to be valid.

Throws

cc3DPoseDefs::BadParams
The number of supplied, valid points is less than 2 or the sizes of the supplied vectors do not mach.

cc3DPoseDefs::Singular
The supplied 2D points and camera calibration objects do not produce a 3D point (the 3D rays are parallel, for example).

cc3DPoseDefs::BehindCameras
The 3D position is behind the camera.

Notes

This function supports CVL timeouts.
  ```

•
```
void cf3DTriangulatePointPhys3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc2Vect>& pointsRaw2D,
   cc3DVect& pointPhys3D,
   const cmStd vector<bool>& isPointValid);
```

Computes the 3D position of a point from the corresponding 2D image locations of the point in two or more images from 3D calibrated cameras. You may supply an optional vector of **bool** to indicate which points are valid.

The computed fit minimizes the sum squared image error between the predicted and actual 2D point locations.

Parameters

raw2DFromPhys3Ds
A vector of 3D camera calibration objects.

pointsRaw2D A vector of 2D image points.

pointPhys3D A 3D point into which the result is placed.

isPointValid An vector indicating which elements of *pointsRaw2D* are valid and should be used for the fitting operation. If an empty vector is supplied, all points are assumed to be valid.

Throws

cc3DPoseDefs::BadParams
The number of supplied, valid points is less than 2 or the sizes of the supplied vectors do not mach.

cc3DPoseDefs::Singular
The supplied 2D points and camera calibration objects do not produce a 3D point (the 3D rays are parallel, for example).

cc3DPoseDefs::BehindCameras
The 3D position is behind the camera.

Notes

This function supports CVL timeouts.

•
```
void cf3DTriangulatePointPhys3DUsingPointsRaw2D(
   const cmStd vector<cc3DCameraCalib>& raw2DFromPhys3Ds,
   const cmStd vector<cc2Vect>& pointsRaw2D,
```

■ cf3DTriangulatePointPhys3DUsingPointsRaw2D()

```
cc3DVect& pointPhys3D, cc3DResiduals& residualsRaw2D,
cc3DResiduals& residualsPhys3D,
const cmStd vector<bool>& isPointValid);
```

Computes the 3D position of a point from the corresponding 2D image locations of the point in two or more images from 3D calibrated cameras along with both 2D and 3D residual error data. You may supply an optional vector of **bool** to indicate which points are valid.

The computed fit minimizes the sum squared image error between the predicted and actual 2D point locations.

Both 2D and 3D residual error is computed. 2D error is computed using the 2D distance between the input points and the computed 3D point mapped back to 2D image space. The 3D error is computed using the 3D distance between the computed 3D point and the 3D rays.

Parameters

raw2DFromPhys3Ds
A vector of 3D camera calibration objects.

pointsRaw2D A vector of 2D image points.

pointPhys3D A 3D point into which the result is placed.

isPointValid An vector indicating which elements of *pointsRaw2D* are valid and should be used for the fitting operation. If an empty vector is supplied, all points are assumed to be valid.

residualsRaw2D 2D residual error information.

residualsPhys3D 3D residual error information.

Throws

cc3DPoseDefs::BadParams
The number of supplied, valid points is less than 2 or the sizes of the supplied vectors do not mach.

cc3DPoseDefs::Singular
The supplied 2D points and camera calibration objects do not produce a 3D point (the 3D rays are parallel, for example).

cc3DPoseDefs::BehindCameras
The 3D position is behind the camera.

Notes

This function supports CVL timeouts.

cf3DValidateCameraCalibs()

```
#include <ch_c3d/calvalid.h>
```

```
cf3DValidateCameraCalibs();
```

Global function to validate 3D camera calibration.

To validate an existing camera calibration, follow these steps:

1. Use the calibrated camera or cameras to acquire new images of a calibration plate (preferably the same plate used to calibrate the cameras originally)
2. Construct a vector of **cc3DCameraCalibFeatures** objects, one for each plate pose.
3. Call the **cf3DValidateCameraCalibs()** function, providing the original camera calibration object or objects and the newly constructed **cc3DCameraCalibFeatures** object or objects.

The calibration validation function can validate both the intrinsic and extrinsic camera parameters. Refer to *ch_c3d/calvalid.h* and to the *Cognex 3D-Locate Developer's Guide* for more information on camera calibration validation.

The following requirements apply to both overloads of **cf3DValidateCameraCalibs()**:

- The tool may be able to validate the calibration using a single plate pose, but multiple poses provide more accurate validation.
- The *cc3DCameraCalibDefs::ePoseTilted* pose type must be used to construct the **cc3DCameraCalibFeatures** objects.
- Each **cc3DCameraCalibFeatures** object must include features from the same number of cameras.

cf3DValidateCameraCalibs

```
void cf3DValidateCameraCalibs(
   const cmStd vector<cc3DCameraCalib> &cameraCalibs,
   const cmStd vector<cc3DCameraCalibFeatures>
   &validationPlatePoseFeatures,
   const cc3DValidateCameraCalibParams &params,
   cc3DValidateCameraCalibResultSet &resultSet);

static inline void cf3DValidateCameraCalibs(
   const cc3DCameraCalibResult &cameraCalibResult,
   const cmStd vector<cc3DCameraCalibFeatures>
```

```
   &validationPlatePoseFeatures,
   const cc3DValidateCameraCalibParams &params,
   cc3DValidateCameraCalibResultSet &resultSet);
```

- ```
  void cf3DValidateCameraCalibs(
     const cmStd vector<cc3DCameraCalib> &cameraCalibs,
     const cmStd vector<cc3DCameraCalibFeatures>
     &validationPlatePoseFeatures,
     const cc3DValidateCameraCalibParams &params,
     cc3DValidateCameraCalibResultSet &resultSet);
  ```

Validate the supplied vector of 3D camera calibration objects using the supplied calibration features. In addition to the general requirements listed above, the number and order of the supplied camera calibration objects and the camera calibration features must be the same.

Parameters

cameraCalibs The camera calibration objects to validate.

validationPatePoseFeatures
The feature correspondences and plate pose characteristics used to perform the validation.

params The parameters for the validation.

resultSet A **cc3DValidateCameraCalibResultSet** into which the results are placed.

Throws

cc3DValidateCameraCalibDefs::BadParams
The elements of *validationPlatePoseFeatures* do not all contain data from the same number of cameras or there are no features in *validationPlatePoseFeatures;*

cc3DCameraCalibDefs::Singular
The input data are degenerate; validation cannot be computed.

cc3DValidateCameraCalibDefs::NotImplemented
At least one element of *validationPlatePoseFeatures* specifies a plate pose other than *cc3DCameraCalibDefs::ePoseTilted*.

- ```
  static inline void cf3DValidateCameraCalibs(
     const cc3DCameraCalibResult &cameraCalibResult,
     const cmStd vector<cc3DCameraCalibFeatures>
  ```

```
&validationPlatePoseFeatures,
const cc3DValidateCameraCalibParams &params,
cc3DValidateCameraCalibResultSet &resultSet);
```

Convenience overload that allows you to provide the camera calibrations as a **cc3DCameraCalibResult**.

cf3DValidateCameraCalibs()

Shape Functions

```
#include <ch_c3d/shapes3d.h>
```

Global utility functions related to 3D shapes.

Functions

cf3DDistance

```
double cf3DDistance(const cc3DLineSeg& lineSeg1,
   const cc3DLineSeg& lineSeg2);
```

Return the minimum distance between two line segments.

cf3DFindNearestPoints

```
void cf3DFindNearestPoints(const cc3DLine& line1,
   const cc3DLine& line2, cc3DVect& pointOnLine1,
   cc3DVect& pointOnLine2);
```

Returns the points on two lines that are closest to each other. If *line1* and *line2* are parallel or coincident, the returned point pair will be one of the pairs of nearest points.

Parameters

line1	The first line.
line2	The second line.
pointOnLine1	The closest point on the first line.
pointOnLine2	The closest point on the second line.

■ Shape Functions

cf3DIntersect

```
void cf3DIntersect(const cc3DPlane& plane,
   const cc3DLine& line, cc3DVect& result,
   cc3DShapeDefs::IntersectionStatus& status);

void cf3DIntersect(const cc3DPlane& plane,
   const cc3DRay& ray, cc3DVect& result,
   cc3DShapeDefs::IntersectionStatus& status);

void cf3DIntersect(const cc3DPlane& plane1,
   const cc3DPlane& plane2, cc3DLine& result,
   cc3DShapeDefs::IntersectionStatus& status);
```

- ```
  void cf3DIntersect(const cc3DPlane& plane,
     const cc3DLine& line, cc3DVect& result,
     cc3DShapeDefs::IntersectionStatus& status);
  ```

Returns the intersection of the supplied shapes. The supplied **cc3DShapeDefs::IntersectionStatus** is set to indicate the type of intersection.

Parameters

plane The plane.

line The line.

result The intersection point.

status The intersection status. *status* is one of the following values:

cc3DShapeDefs::eIntersect
cc3DShapeDefs::eOverlap
cc3DShapeDefs::eNone

Throws

cc3DShapeDefs::DegenerateShape
One of the supplied shapes is degenerate.

- ```
  void cf3DIntersect(const cc3DPlane& plane,
     const cc3DRay& ray, cc3DVect& result,
     cc3DShapeDefs::IntersectionStatus& status);
  ```

Returns the intersection of the supplied shapes. The supplied **cc3DShapeDefs::IntersectionStatus** is set to indicate the type of intersection.

Parameters

plane The plane.

ray The ray.

result The intersection point.

status The intersection status. *status* is one of the following values:

cc3DShapeDefs::eIntersect
cc3DShapeDefs::eOverlap
cc3DShapeDefs::eNone

Throws

cc3DShapeDefs::DegenerateShape
One of the supplied shapes is degenerate.

- ```
void cf3DIntersect(const cc3DPlane& plane1,
   const cc3DPlane& plane2, cc3DLine& result,
   cc3DShapeDefs::IntersectionStatus& status);
```

Returns the intersection of the supplied shapes. The supplied **cc3DShapeDefs::IntersectionStatus** is set to indicate the type of intersection.

**Parameters**

*plane1* The first plane.

*plane2* The second plane.

*result* The intersection line.

*status* The intersection status. *status* is one of the following values:

*cc3DShapeDefs::eIntersect*
*cc3DShapeDefs::eOverlap*
*cc3DShapeDefs::eNone*

**Throws**

*cc3DShapeDefs::DegenerateShape*
One of the supplied shapes is degenerate.
```

■ Shape Functions

cf3DProjectOntoPlane

```
void cf3DProjectOntoPlane(const cc3DPlane& plane,
   const cc3DLineSeg& lineSeg, cc3DLineSeg& result,
   cc3DShapeDefs::ProjectionStatus& resultStatus);

void cf3DProjectOntoPlane(const cc3DPlane& plane,
   const cc3DLine& line, cc3DLine& result,
   cc3DShapeDefs::ProjectionStatus& resultStatus);
```

- ```
  void cf3DProjectOntoPlane(const cc3DPlane& plane,
     const cc3DLineSeg& lineSeg, cc3DLineSeg& result,
     cc3DShapeDefs::ProjectionStatus& resultStatus);
  ```

Projects the supplied line segment onto the supplied plane, placing the resulting line segment into the supplied argument.

Parameters

plane The plane onto which to project.

lineSeg The line segment to project.

result The projected line segment.

resultStatus If the projection result is degenerate (the line segment and plane are perpendicular), *resultStatus* is set to *cc3DShapeDefs::eDegenerateProjection*. Otherwise, *resultStatus* is set to *cc3DShapeDefs::eStandardProjection*

Throws

cc3DShapeDefs::DegenerateShape
One of the supplied shapes is degenerate.

- ```
  void cf3DProjectOntoPlane(const cc3DPlane& plane,
     const cc3DLine& line, cc3DLine& result,
     cc3DShapeDefs::ProjectionStatus& resultStatus);
  ```

Projects the supplied line onto the supplied plane, placing the resulting line into the supplied argument.

Parameters

plane The plane onto which to project.

lineSeg The line to project.

result The projected line.

resultStatus If the projection result is degenerate (the line and plane are perpendicular), *resultStatus* is set to *cc3DShapeDefs::eDegenerateProjection*. Otherwise, *resultStatus* is set to *cc3DShapeDefs::eStandardProjection*

Throws

cc3DShapeDefs::DegenerateShape
One of the supplied shapes is degenerate.

cfRealAntiParallel

```
bool cfRealAntiParallel(const cc3DLine &line1,
   const cc3DLine &line2, double epsilon);

bool cfRealAntiParallel(const cc3DPlane &plane1,
   const cc3DPlane &plane2, double epsilon);
```

Returns true if the two shapes are anti-parallel within the supplied tolerance (they are parallel within the supplied tolerance and the dot product of their direction vectors are less than 0).

Shape Functions

cfRealEq

```
bool cfRealEq(const cc3DLine& r1, const cc3DLine& r2,
   double epsilon);

bool cfRealEq(const cc3DPoint& p1, const cc3DPoint& p2,
   double epsilon);

bool cfRealEq(const cc3DLineSeg& line1,
   const cc3DLineSeg& line2, double epsilon);

bool cfRealEq(const cc3DRay& ray1, const cc3DRay& ray2,
   double epsilon);

bool cfRealEq(const cc3DPlane& plane1,
   const cc3DPlane& plane2, double epsilon= 1e-15);

bool cfRealEq(const cc3DCircle& c1, const cc3DCircle& c2,
   double epsilon);

bool cfRealEq(const cc3DRect& r1, const cc3DRect& r2,
   double epsilon);

bool cfRealEq(const cc3DAlignedBox& box1,
   const cc3DAlignedBox& box2, double epsilon);

bool cfRealEq(const cc3DBox& box1, const cc3DBox& box2,
   double epsilon);

bool cfRealEq(const cc3DSphere& s1, const cc3DSphere& s2,
   double epsilon);

bool cfRealEq(const cc3DAxisAngle& v1,
   const cc3DAxisAngle& v2, double epsilon = 1e-15);

bool cfRealEq(const cc3DEulerZYX& v1,
   const cc3DEulerZYX& v2, double epsilon = 1e-15);

bool cfRealEq(const ccQuaternion& v1,
   const ccQuaternion& v2, double epsilon = 1e-15);

bool cfRealEq(const cc3DResiduals& r1,
   const cc3DResiduals& r2, double epsilon = 1e-15);

bool cfRealEq(const cc3DPositionResiduals& r1,
   const cc3DPositionResiduals& r2,
   double epsilon = 1e-15);

bool cfRealEq(const cc3DRotation& r1,
   const cc3DRotation& r2, double epsilon = 1e-15);

bool cfRealEq(const cc3DXformRigid& x1,
   const cc3DXformRigid& x2, double epsilon = 1e-15);
```

```
bool cfRealEq(const cc3DXformRigid& xRigid,
   const cc3Xform& xLinear, double epsilon = 1e-15);

bool cfRealEq(const cc3Xform& xLinear,
   const cc3DXformRigid& xRigid, double epsilon = 1e-15);
```

Returns true if the two shapes, transformations, or rotations have the same numerical values within the supplied tolerance.

Notes

The **cfRealEq()** overloads that compare two transformations should not be used to compare transformations in physical spaces, or that use physical units.

The **cfRealEq()** overloads that compare a **cc3DXformRigid** with a **cc3Xform** convert the **cc3DXformRigid()** to a **cc3Xform** constructed from the **cc3DXformRigid::rotation().matrix** and **cc3DXformRigid.trans()** vector, then compares that **cc3Xform** with the supplied one.

cfRealParallel

```
bool cfRealParallel(const cc3DLine &line1,
   const cc3DLine &line2, double epsilon);

bool cfRealParallel(const cc3DLine &line,
   const cc3DPlane &plane, double epsilon);

bool cfRealParallel(const cc3DPlane &plane1,
   const cc3DPlane &plane2, double epsilon);
```

Returns true if the two shapes are parallel within the supplied tolerance.

cfRealParallelIncludingDirection

```
bool cfRealParallelIncludingDirection(
   const cc3DLine &line1, const cc3DLine &line2,
   double epsilon);

bool cfRealParallelIncludingDirection(
   const cc3DPlane &plane1, const cc3DPlane &plane2,
   double epsilon);
```

Returns true if the two shapes are parallel within the supplied tolerance and their dot product is greater than 0.

cfRealPerpendicular

```
bool cfRealPerpendicular(const cc3DLine &line,
   const cc3DPlane &plane, double epsilon);
```

Returns true if the supplied line is perpendicular to the supplied plane within the supplied tolerance.

Index

■ Index

D

E

F

G

I

M

N

P

S

T

U

V

W

X