

Generic Sensor Model API

Lynn H. Quam
SRI International, Menlo Park, CA 94025 USA

June 2, 1997

1 Introduction

The main intent of the SRI APGD effort in developing a Generic Sensor Model API (GENSEN) is to provide the imaging research and developer communities with a uniform and efficient interface to a variety of sensor (camera) models, without requiring any knowledge of the underlying math models.

Although image sensor models are the primary focus of GENSEN, the API uses a framework suitable for arbitrary coordinate transformations. This framework has been used within the RADIUS Common Development Environment (RCDE) since 1992 and is similar to the coordinate system/transformation framework of the DARPA Image Understanding Environment (IUE).

Historically, the DARPA IU research community has utilized only idealized frame camera (pinhole) math models. Some of the computational techniques developed for pinhole models, such as the fundamental matrix and epi-polar rectification of stereo image pairs, do not have global counterparts in generic sensor models, but must be computed within spatially localized areas. GENSEN provides generic implementations for many of the common operations expected of frame camera math models.

2 Coordinate Vectors

Coordinate vectors represent positions in n-dimensional space and are implemented as vanilla C vectors of double floats. It is the responsibility of the application to make sure that the length of a coordinate vector is sufficient for the dimensionality of the coordinate space. As implemented, there is no checking of coordinate vector dimensionality.

The following C types are defined for coordinate vectors:

- **coordinate_element:** The type (double) of an element of a coordinate vector.
- **coordinate_vector:** A pointer to a vector of **coordinate_elements**.
- **cv:** An abbreviation for **coordinate_vector**.
- **cv3:** A length 3 vector of **coordinate_elements**.
- **cv2:** A length 2 vector of **coordinate_elements**.

3 Coordinate Transforms

A Coordinate Transformation implements a mathematical mapping from one coordinate system to another coordinate system.

3.1 Function `transform_vector`

```
coordinate_vector  
transform_vector(coordinate_transform transform,  
                coordinate_vector from_vector,  
                coordinate_vector to_vector)
```

transform_vector performs the coordinate mapping of the coordinates in **from_vector** storing the mapped coordinates in **to_vector** and returning **to_vector**.

All coordinate transforms must implement the following behavior:

- NULL is returned if the coordinates in **from_vector** are not appropriate for the transformation (such as a point behind the camera, or outside of the viewing cone of the camera).
- NULL is returned if **from_vector** = NULL.
- The results are undefined if **to_vector** = NULL. (It might be desirable to malloc and return an appropriate vector, or default **to_vector** to **from_vector**.)
- If **to_vector** = **from_vector**, the coordinate transform function must be careful in performing its calculations such that storing results into **from_vector** does not corrupt the calculation.

The above behavior was carefully chosen so that chaining together the results of several coordinate transform is simple. For example:

```
result = transform_vector (trans2, transform_vector(trans1, pt1, pt2), pt3);
```

If `trans1` returns NULL, then `result` will be NULL.

3.2 Function `transform_jacobian_matrix`

```
int  
transform_jacobian_matrix(coordinate_transform transform,  
                        coordinate_vector from_vector,  
                        double *jacobian_matrix)
```

transform_jacobian_matrix computes the jacobian matrix of the coordinate transformation function with respect to the components of **from_vector**, storing the resulting matrix in **partials**. If the coordinate transform operates from 3-space to 3-space, then `jacobian_matrix` is defined as follows.

$$\langle u, v, w \rangle = F \langle x, y, z \rangle$$

$$J = \begin{bmatrix} \frac{\partial F_u}{\partial x} & \frac{\partial F_u}{\partial y} & \frac{\partial F_u}{\partial z} \\ \frac{\partial F_v}{\partial x} & \frac{\partial F_v}{\partial y} & \frac{\partial F_v}{\partial z} \\ \frac{\partial F_w}{\partial x} & \frac{\partial F_w}{\partial y} & \frac{\partial F_w}{\partial z} \end{bmatrix}$$

All coordinate transforms must implement the following behavior for the function `transform_jacobian_matrix`:

- NULL is returned if the coordinates in **from_vector** are not appropriate for the transformation (such as a point behind the camera, or outside of the viewing cone of the camera).
- NULL is returned if **from_vector** = NULL.
- The results are undefined if **jacobian_matrix** = NULL.

NOTE: `jacobian_matrix` is not a pointer to vector of pointers to vectors, but a pointer to a vector of doubles. The following is an example of usage:

```
{double jacobian[3][3];
  transform_jacobian_matrix(trans, pt, (double *) jacobian);
  ...
}
```

3.3 Inverse Coordinate Transforms

Inverse Coordinate Transforms can be implemented either using a closed form computation that computes the inverse, or using a numerical iteration method. For a linear transformation, the inverse transform is another linear transformation. More complicated coordinate transforms, such as WGS-84 $\langle lat, long, elevation \rangle$ to geocentric $\langle x, y, z \rangle$ require numerical inversion.

In general, not all coordinate transforms are one-to-one mappings, and therefore may not have unique inverses. When this situation exists, the inverse coordinate transform instance must contain additional slots to control choice of inverse. Within the RCDE, no coordinate transforms or projections require such slots.

3.4 Coordinate Transform Slot Accessors

```
coordinate_transform->inverse_transform
```

This is the **inverse transform** of **coordinate_transform**. This slot is NULL if the inverse is not implemented.

```
coordinate_transform->from_cs
```

This is the **from coordinate system** of the transform corresponding to the **from_vector**. This slot may be NULL.

```
coordinate_transform->to_cs
```

This is the **to coordinate system** of the transform corresponding to the **to_vector**. This slot may be NULL.

3.5 Function `inverse_transform_vector`

`inverse_transform_vector` is a convenience function implemented as follows:

```
inverse_transform_vector(coordinate_transform transform,
                        coordinate_vector from_vector,
                        coordinate_vector to_vector)
{
    if (transform->inverse_transform)
        {return(transform_vector(transform->inverse_transform, from_vector, to_vector)}
    else return(0);
}
```

3.6 Function `read_coordinate_transform`

```
coordinate_transform read_coordinate_transform(char *filename)
```

`read_coordinate_transform` creates a coordinate transform instance from the information contained in **filename**. NULL is returned if the file does not exist, is not accessible, or is not recognized as containing a coordinate transform.

3.7 Function `make_linear_transform_4x4`

```
make_linear_transform_4x4(double M[4][4])
```

`make_linear_transform_4x4` creates an 3-space to 3-space linear transformation defined by multiplying the matrix **M** times the homogenous vector $\langle x, y, z, 1 \rangle$:

$$\langle u, v, w \rangle = M \langle x, y, z, 1 \rangle$$

```
linear_transform_4x4_matrix(coordinate_transform trans, double M[4][4])
```

Copies the contents of the transform matrix of the **trans** into **M**. The class_code of **trans** must be `LIN-EAR_TRANSFORM_4x4_CLASS_CODE`.

```
set_linear_transform_4x4_matrix(coordinate_transform trans, double M[4][4])
```

Copies the contents of **M** into the transform matrix of **trans**. The class_code of **trans** must be `LIN-EAR_TRANSFORM_4x4_CLASS_CODE`.

4 Sensor Models

Sensor Models are implemented using coordinate projections which are a special sub-class of coordinate transforms. This section describes issues peculiar to coordinate projections.

Coordinate projections in GENSEN define a 3d to 3d coordinate mapping from world coordinates $\langle x, y, z \rangle$ to sensor coordinates $\langle u, v, w \rangle$. The u, v components of a sensor coordinate vector define the position within the sensed image. The third sensor coordinate, w , is used to define a depth ordering on points in space which project to the same u, v . Usually, the w computation is the following dot-product:

$$w = \langle x, y, z, 1 \rangle \cdot \langle w_x, w_y, w_z, w_1 \rangle$$

The vector $\langle w_x, w_y, w_z, w_1 \rangle$ defines a plane in space, whose normal is usually chosen to be the camera direction vector at the center of the image (principal point). For pinhole camera models, this choice is identical to that used in 3-d graphics z-buffering calculations.

The **inverse.transform** of a coordinate projection defines the mapping from sensor coordinates $\langle u, v, w \rangle$ to world coordinates $\langle x, y, z \rangle$. For a given sensor point u, v we can define a parametric curve $C_{u,v}(w)$ where w is the position along the curve. Inverse projections can be implemented as closed form calculations or by numerical inversion techniques.

Sometimes it is desirable implement the sensor-to-world coordinate mapping in a closed form, and compute to world-to-sensor coordinate mapping with a numerical inversion technique. This is permissible in GENSEN, but can lead to a serious performance degradation if the world-to-sensor coordinate mapping is used extensively as in an interactive 3d graphics environment.

4.1 Generic Functions for Sensor Models

```
coordinate_vector  
camera_direction_vector (coordinate_transform camera,  
                        coordinate_vector world_pt,  
                        coordinate_vector dir)
```

camera_direction_vector computes the direction of the camera ray path of **camera** at point **world_pt**, storing the resulting vector components in **dir**. If **world_pt** is NULL or the transform is not defined at **world_pt**, then NULL is returned.

```
double  
coordinate_transform_gsd (coordinate_transform transform,  
                        coordinate_vector from_pt)
```

coordinate_transform_gsd computes the ground sample distance of **transform** at **from_pt**. This corresponds to the square root of the area of the projection of a unit pixel onto a plane perpendicular to the **camera_direction_vector** thru **from_point**. This projection plane was chosen rather than a horizontal plane to avoid problems with oblique imagery.

```
int
intersect_camera_ray_with_plane (coordinate_transform trans,
                                coordinate_vector image_pt,
                                double plane[4],
                                coordinate_vector intersection_pt)
```

```
int
multi_camera_triangulation (int ncams,
                            coordinate_transform cams[],
                            cv2 imgpts[],
                            cv2 covars[],
                            cv3 world_pt,
                            triangulate_image_points_control *control)
```

```
int
epipolar_rectification_matrices (coordinate_transform proj1,
                                 coordinate_transform proj2,
                                 cv3 world_pos,
                                 double *pp_normal,
                                 double mat1[2][2],
                                 double mat2[2][2],
                                 cv3 epipolar_plane_normal)
```

5 Coordinate Systems

A coordinate system corresponds to n-dimensional space. When coordinate systems are associated with coordinate transforms, a graph is defined where the arcs in the graph correspond to coordinate transforms, and the nodes correspond to coordinate systems.

In GENSEN 0.1, the use of coordinate systems is optional and incomplete. The following coordinate system attributes are supported.

- **dimensionality**: The number of dimensions of the coordinate space.
- **name**: An string naming the coordinate system.

In GENSEN 0.1, coordinate systems are implemented in a very minimal fashion. The application is responsible for associating coordinate system instances with **from_cs** and **to_cs** slots of coordinate transforms. Here is the functional interface to coordinate systems:

```
coordinate_system
make_coordinate_system(COORDINATE_SYSTEM_CLASS_CODE class_code,
                      int dimensionality,
                      char *name)
```

class_code is the type of the coordinate system. Some of the supported class_codes are:

- UNKNOWN_COORDINATE_SYSTEM_CLASS_CODE
- CARTESIAN_COORDINATE_SYSTEM_CLASS_CODE
- LAT_LONG_COORDINATE_SYSTEM_CLASS_CODE
- UTM_COORDINATE_SYSTEM_CLASS_CODE

5.1 Coordinate System Slot Accessors

```
coordinate_system->class_code
```

```
coordinate_system->name
```

```
coordinate_system->dimensionality
```

6 Implementation

GENSEN is implemented in the C language using a POSIX (I hope) interface to the operating system.

The use of dynamic shared libraries provides a “Plug and Play” aspect. Such libraries can be upgraded to include new sensor models or sensor model file formats without the need to recompile application programs.