Easy Transformations between Cartesian, Cylindrical and Spherical Coordinates

Carole Blanc Christophe Schlick

LaBRI [1], 351 cours de la libération, 33405 Talence (FRANCE)
[blanc|schlick]@labri.u-bordeaux.fr

We propose a set of functions that realize all possible transformations of a point in a threedimensional euclidian space between global and local frames using either cartesian, cylindrical or spherical coordinates.

1 Introduction

A widely used technique to create scenes for computer imagery is to define each object in a given *local* frame and then to express these frames in a given global (or *world*) frame. The main advantage of such a process is that it enables to move (translate, rotate, scale) the object by simply changing the coordinates of the local frame in the world frame. Another technique (much less usual) is to choose the local frame so as the object is totally enclosed in the unit cube[2] (i.e. local coordinates are within $[-1,1]^3$). The idea here is to associate the same canonic threedimensional domain with each object, whatever its position, orientation or scale. This canonic domain will greatly simplify the application of procedural deformations or procedural solid textures (see also [BLAN95a] and [BLAN95b] for an illustration of this idea). Moreover, such a constant domain insures the deformation or the texture applied on a given object will not change while it is moved.

2 Cartesian Coordinates

Converting the cartesian coordinates of a point P from the world frame to the local one (and reciprocally) may be done in an elegant way with homogeneous coordinates [FOLE90]. In the case where transformations do not have to be composed, using the affine formulation saves a few multiplications for the frame conversion routines.

$$\text{In the world frame :} \quad \Omega P = xX + yY + zZ$$

where (Ω, X, Y, Z) is the world frame and (x, y, z) the world cartesian coordinates of P.

$$\text{In the local frame :} \quad OP = uU + vV + wW$$

where (O, U, V, W) is the local frame and (u, v, w) the local cartesian coordinates of P.

$$\text{World} \rightarrow \text{Local :} \quad (u,v,w) = \frac{(x-x_0,\, y-y_0,\, z-z_0)}{\text{determinant}(U,V,W)} \; [V\times W \quad W\times U \quad U\times V]$$

$$\text{Local} \rightarrow \text{World :} \quad (x,y,z)^\top = [U \quad V \quad W]\,(u,v,w)^\top + (x_0,y_0,z_0)^\top$$

where (x_0, y_0, z_0) are the world cartesian coordinates of point O. In the source file below, these two transformations are respectively called `world_to_car` and `car_to_world`.

3 Cylindrical and Spherical Coordinates

When (O, U, V, W) is the canonic frame defined above, the local cartesian coordinates (u, v, w) belong to the unit cube : $(u, v, w) \in [-1, 1]^3$. As we have shown elsewhere [GUIT93] [BLAN94], many interesting

[1] Laboratoire Bordelais de Recherche en Informatique (*Université Bordeaux I* and *Centre National de la Recherche Scientifique*). The present work is also granted by the *Conseil Régional d'Aquitaine*.

[2] A simple way to create such a frame is to compute the bounding box of the object, put the origin at the center of the box and scale the unit frame vectors by half the size of the box in each direction.

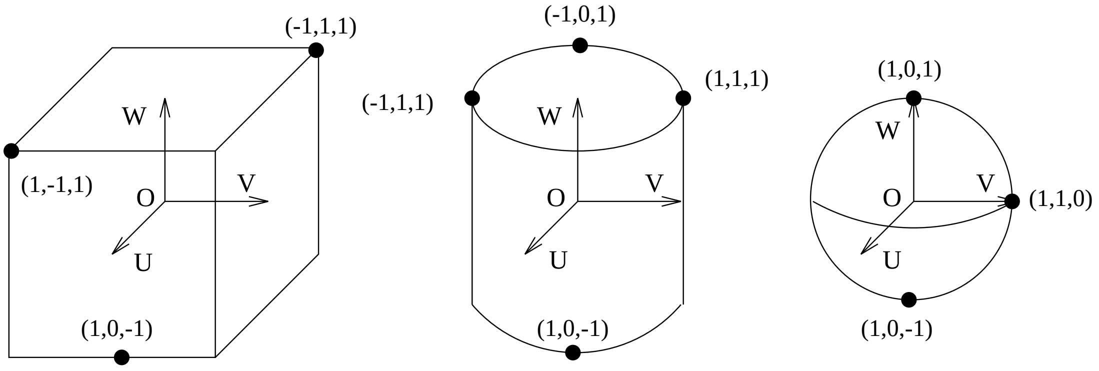

Figure 1: (a) Unit cube (b) Unit cylinder (c) Unit sphere

procedural deformation or texture effects may be obtained when using the cylindrical coordinates (r, θ, z) or the spherical coordinates (r, θ, ϕ) instead of the cartesian ones. Transforming cartesian coordinates into either cylindrical or spherical coordinates can be done quite easily. The problem is that the resulting coordinates do not belong to the unit cube any longer ($r \in [0, 1], \theta \in [0, 2\pi]$ and $\phi \in [-\pi/2, \pi/2]$). Moreover the topology of volume induced by these coordinates has changed (for instance, $\theta = 0$ and $\theta = 2\pi$ yields the same point for a given value of r and ϕ) which may introduce side effects in the deformation or texturing scheme. To correct these drawbacks, we propose a new parametrization $(u, v, w) \in [-1, 1]^3$ of the unit cylinder and the unit sphere which provides a topological equivalence to the unit cube :

$$\text{Cylindrical coordinates :} \quad OP = (u \cos \frac{\pi v}{2})U + (u \sin \frac{\pi v}{2})V + wW$$

$$\text{Spherical coordinates :} \quad OP = (u \cos \frac{\pi v}{2} \cos \frac{\pi w}{2})U + (u \sin \frac{\pi v}{2} \cos \frac{\pi w}{2})V + u \sin \frac{\pi w}{2} W$$

Figure 1 shows the coordinates of some particular points on the unit cylinder/sphere to illustrate how the new parametrization acts. In addition to manipulating only normalized coordinates, the new parametrization has another convenient property : if two different points have got the same signs for their coordinates (pair to pair) they lay in the same octant. In other words, the octant in which belongs a point can be found simply by looking at its coordinate signs (whether these coordinates are cartesian, cylindrical or spherical).

$$\begin{aligned}
\text{Car}\rightarrow\text{Cyl} : (u', v', w') &= (\overline{u}\sqrt{u^2 + v^2}, \frac{2}{\pi} \arcsin \frac{v}{\sqrt{u^2 + v^2}}, w) \\
\text{Cyl}\rightarrow\text{Car} : (u', v', w') &= (u \cos \frac{\pi v}{2}, u \sin \frac{\pi v}{2}, w) \\
\text{Car}\rightarrow\text{Sph} : (u', v', w') &= (\overline{u}\sqrt{u^2 + v^2 + w^2}, \frac{2}{\pi} \arcsin \frac{v}{\sqrt{u^2 + v^2}}, \frac{2}{\pi} \arcsin \frac{w}{\sqrt{u^2 + v^2 + w^2}}) \\
\text{Sph}\rightarrow\text{Car} : (u', v', w') &= (u \cos \frac{\pi v}{2} \cos \frac{\pi w}{2}, u \sin \frac{\pi v}{2} \cos \frac{\pi w}{2}, u \sin \frac{\pi w}{2})
\end{aligned}$$

where $\overline{u}$ is the sign value (± 1) of u. In the source file, these four transformations are respectively called `car_to_cyl`, `cyl_to_car`, `car_to_sph` and `sph_to_car`. Note that four other routines are provided, which to convert directly world cartesian coordinates to and from local cylindrical and spherical coordinates (`world_to_cyl`, `cyl_to_world`, `world_to_sph`, `sph_to_world`).

4 Optimization

The conversion routines make heavy use of expensive trigonometric functions. Sometimes (during the creation of a rendering scene, for instance) it may be interesting to get a low precision but high speed approximation. We propose here such an optimization which uses the halved tangent. Indeed, as said in [PAET92] :

$$t \simeq \tan \frac{\pi t}{4} \quad \text{when} \quad t \in [-1, 1] \qquad \text{(see Figure 2a)}$$

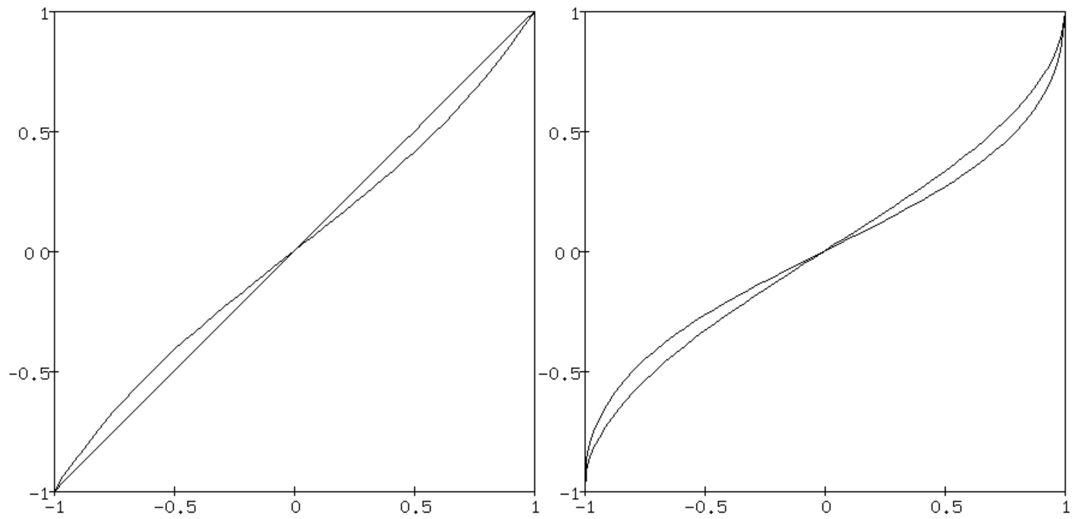

Figure 2: (a) Approximation of tan   (b) Approxmation of arcsin

Therefore, we may use the following approximation :

$$\cos\frac{\pi t}{2} \simeq \frac{1-t^2}{1+t^2} \quad \text{and} \quad \sin\frac{\pi t}{2} \simeq \frac{2t}{1+t^2} \quad \text{where} \quad t \in [-1,1]$$

Taking the reciprocal function provides approximations for arctan, arcsin and arccos. We are only interested in the arcsin function (which is the only one used in the conversion routines). Here again, the approximation is quite good :

$$\frac{2}{\pi}\arcsin t \simeq \frac{1-\sqrt{1-t^2}}{t} \quad \text{where} \quad t \in [-1,1] \qquad \text{(see Figure 2b)}$$

These optimized conversion routines can be found in the source file distinguished from the other ones by the prefix `fast_xxx`.

5 Source Files

```
/* ------------------------------------------------------------------------- *\
   FRAME.H : Carole Blanc & Christophe Schlick (3 June 1994)

   This package provides easy transformations routines between global and
   local frames using cartesian, cylindrical and spherical 3D coordinates.

\* ------------------------------------------------------------------------- */

#ifndef _FRAME_
#define  _FRAME_

/*
** This package uses the "Toolbox of Macros Functions for Computer Graphics"
** which provides files : tool.h, real.h, uint.h, sint.h, vec?.h and mat?.h
*/
#include "real.h"

/*
** Type definition of a 3D frame = 3 vectors + 1 point
*/
typedef realvec3 frame[4];

/*
** Get from file and put in file
*/
#define GET_FRAME(File,Var)\
        (GET_REALVEC3(File,(Var)[0]),\
         GET_REALVEC3(File,(Var)[1]),\
         GET_REALVEC3(File,(Var)[2]),\
         GET_REALVEC3(File,(Var)[3]))
#define PUT_FRAME(File,Var)\
        (PUT_REALVEC3(File,(Var)[0]),\
         PUT_REALVEC3(File,(Var)[1]),\
         PUT_REALVEC3(File,(Var)[2]),\
         PUT_REALVEC3(File,(Var)[3]))

/*
** External declarations
*/
extern void car_to_cyl (realvec3 *Point);
extern void fast_car_to_cyl (realvec3 *Point);
extern void cyl_to_car (realvec3 *Point);
extern void fast_cyl_to_car (realvec3 *Point);
extern void car_to_sph (realvec3 *Point);
extern void fast_car_to_sph (realvec3 *Point);
extern void sph_to_car (realvec3 *Point);
extern void fast_sph_to_car (realvec3 *Point);
extern void world_to_car (realvec3 *Point, frame Frame);
extern void car_to_world (realvec3 *Point, frame Frame);
extern void world_to_cyl (realvec3 *Point, frame Frame);
extern void fast_world_to_cyl (realvec3 *Point, frame Frame);
extern void cyl_to_world (realvec3 *Point, frame Frame);
extern void fast_cyl_to_world (realvec3 *Point, frame Frame);
extern void world_to_sph (realvec3 *Point, frame Frame);
```

```
extern void fast_world_to_sph (realvec3 *Point, frame Frame);
extern void sph_to_world (realvec3 *Point, frame Frame);
extern void fast_sph_to_world (realvec3 *Point, frame Frame);

#endif

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/* ------------------------------------------------------------------------- *\
   FRAME.C : Carole Blanc & Christophe Schlick (3 June 1994)

   This package provides easy transformations routines between global and
   local frames using cartesian, cylindrical and spherical 3D coordinates.

\* ------------------------------------------------------------------------- */

#include "frame.h"
#include "vec2.h"
#include "vec3.h"
#include "mat2.h"
#include "mat3.h"

/*
** car_to_cyl :
**
** Input  : Point = unit cartesian coordinates (in [-1,1]) in a local frame
** Output : Point = unit cylindrical coordinates (in [-1,1]) in the same frame
**                  (Point.x=radius, Point.y=2*theta/PI, Point.z=z)
*/
void car_to_cyl (realvec3 *Point)
{
  real u, v;

  if (ZERO (Point->x) && ZERO (Point->y)) {
    u = 0.0; v = 0.0;
  }
  else {
    u = (Point->x < 0.0) ? -LEN_VEC2 (*Point) : LEN_VEC2 (*Point);
    v = asin (Point->y / u) / HALF_PI;
  }
  MAKE_VEC2 (*Point, u, v);
}

/*
** cyl_to_car :
**
** Input  : Point = unit cylindrical coordinates (in [-1,1]) in a local frame
**                  (Point.x=radius, Point.y=2*theta/PI, Point.z=z)
** Output : Point = unit cartesian coordinates (in [-1,1]) in the same frame
*/
void cyl_to_car (realvec3 *Point)
{
  real u, v;

  u = Point->x; v = Point->y * HALF_PI;
  MAKE_VEC2 (*Point, u*cos (v), u*sin (v));
}
```

```
/*
** car_to_sph :
**
** Input  : Point = unit cartesian coordinates (in [-1,1]) in a local frame
** Output : Point = unit spherical coordinates (in [-1,1]) in the same frame
**                  (Point.x=radius, Point.y=2/PI*theta, Point.z=2/PI*phi)
*/
void car_to_sph (realvec3 *Point)
{
  real u, v, w;

  if (ZERO (Point->x) && ZERO (Point->y)) {
    u = ABS (Point->z); v = 0.0; w = SGN (Point->z);
  }
  else {
    u = (Point->x < 0.0) ? -LEN_VEC2 (*Point) : LEN_VEC2 (*Point);
    v = asin (Point->y / u) / HALF_PI;
    u = (Point->x < 0.0) ? -LEN_VEC3 (*Point) : LEN_VEC3 (*Point);
    w = asin (Point->z / u) / HALF_PI;
  }
  MAKE_VEC3 (*Point, u, v, w);
}

/*
** sph_to_car :
**
** Input  : Point = unit spherical coordinates (in [-1,1]) in a local frame
**                  (Point.x=radius, Point.y=2/PI*theta, Point.z=2/PI*phi)
** Output : Point = unit cartesian coordinates (in [-1,1]) in the same frame
*/
void sph_to_car (realvec3 *Point)
{
  real u, v, w, a, b;

  u = Point->x; v = Point->y * HALF_PI; w = Point->z * HALF_PI;
  a = sin (v); v = cos (v); b = sin (w); w = cos (w);
  MAKE_VEC3 (*Point, u*v*w, u*a*w, u*b);
}

/*
** In order to speed-up the computation compared to the previous
** versions, the four following "fast_*" routines use the fact that
** "v" is a good approximation of "tan (v*PI/4)" in the range [-1,1]
** (see "The Joys of the Halved Tangent" by A. Peath in Graphics Gems II)
** If precision is not imperative for you, these are the routines to use.
*/

/*
** fast_car_to_cyl :
**
** Input  : Point = unit cartesian coordinates (in [-1,1]) in a local frame
** Output : Point = unit cylindrical coordinates (in [-1,1]) in the same frame
**                  (Point.x=radius, Point.y=2*theta/PI, Point.z=z)
*/
void fast_car_to_cyl (realvec3 *Point)
{
```

```
  real u, v;

  u = (Point->x < 0.0) ? -LEN_VEC2 (*Point) : LEN_VEC2 (*Point);
  v = ZERO (Point->y) ? 0.0 : (u - Point->x) / Point->y;
  MAKE_VEC2 (*Point, u, v);
}

/*
** fast_cyl_to_car :
**
** Input  : Point = unit cylindrical coordinates (in [-1,1]) in a local frame
**                  (Point.x=radius, Point.y=2*theta/PI, Point.z=z)
** Output : Point = unit cartesian coordinates (in [-1,1]) in the same frame
*/
void fast_cyl_to_car (realvec3 *Point)
{
  real u, v, s;

  u = Point->x; v = Point->y;
  s = u / (1.0 + v*v);
  MAKE_VEC2 (*Point, s*(1.0-v*v), s*(v+v));
}

/*
** fast_car_to_sph :
**
** Input  : Point = unit cartesian coordinates (in [-1,1]) in a local frame
** Output : Point = unit spherical coordinates (in [-1,1]) in the same frame
**                  (Point.x=radius, Point.y=2/PI*theta, Point.z=2/PI*phi)
*/
void fast_car_to_sph (realvec3 *Point)
{
  real u, v, w;

  if (ZERO (Point->x) && ZERO (Point->y)) {
    u = ABS (Point->z); v = 0.0; w = SGN (Point->z);
  }
  else {
    w = (Point->x < 0.0) ? -LEN_VEC2 (*Point) : LEN_VEC2 (*Point);
    v = (ABS (Point->y) < TOL) ? 0.0 : (w - Point->x) / Point->y;
    u = (Point->x < 0.0) ? -LEN_VEC3 (*Point) : LEN_VEC3 (*Point);
    w = (ABS (Point->z) < TOL) ? 0.0 : (u - w) / Point->z;
  }
  MAKE_VEC3 (*Point, u, v, w);
}

/*
** fast_sph_to_car :
**
** Input  : Point = unit spherical coordinates (in [-1,1]) in a local frame
**                  (Point.x=radius, Point.y=2/PI*theta, Point.z=2/PI*phi)
** Output : Point = unit cartesian coordinates (in [-1,1]) in the same frame
*/
void fast_sph_to_car (realvec3 *Point)
{
  real u, v, w, a, b;
```

```
  u = Point->x; v = Point->y; w = Point->z;
  a = 1.0 / (1.0 + v*v); b = u / (1.0 + w*w);
  MAKE_VEC3 (*Point, a*b*(1.0-v*v)*(1.0-w*w), a*b*(v+v)*(1.0-w*w), b*(w+w));
}

/*
** world_to_car :
**
** Input  : Frame = 3 vectors and 1 point defining a local frame
**          Point = cartesian coordinates in the world frame
** Output : Point = unit cartesian coordinates (in [-1,1]) in the local frame
*/
void world_to_car (realvec3 *Point, frame Frame)
{
  frame NewFrame;
  real  Delta;

  CROSS_VEC3 (NewFrame[0], Frame[1], Frame[2]);
  CROSS_VEC3 (NewFrame[1], Frame[2], Frame[0]);
  CROSS_VEC3 (NewFrame[2], Frame[0], Frame[1]);
  Delta = DOT_VEC3 (Frame[0], NewFrame[0]);
  SUB_VEC3 (NewFrame[3], *Point, Frame[3]);
  DIVS_VEC3 (NewFrame[3], NewFrame[3], Delta);
  LMAT_VEC3 (*Point, NewFrame[3], NewFrame);
}

/*
** car_to_world :
**
** Input  : Frame = 3 vectors and 1 point defining a local frame
**          Point = unit cartesian coordinates (in [-1,1]) in the local frame
** Output : Point = cartesian coordinates in the world frame
*/
void car_to_world (realvec3 *Point, frame Frame)
{
  realvec3 NewPoint;

  RMAT_VEC3 (NewPoint, Frame, *Point);
  ADD_VEC3 (*Point, NewPoint, Frame[3]);
}

/*
** world_to_cyl and fast_world_to_cyl :
**
** Input  : Frame = 3 vectors and 1 point defining a local frame
**          Point = cartesian coordinates in the world frame
** Output : Point = unit cylindrical coordinates (in [-1,1]) in the local frame
**                  (Point.x=radius, Point.y=2*theta/PI, Point.z=z)
*/
void world_to_cyl (realvec3 *Point, frame Frame)
{
  world_to_car (Point, Frame); car_to_cyl (Point);
}

void fast_world_to_cyl (realvec3 *Point, frame Frame)
{
  world_to_car (Point, Frame); fast_car_to_cyl (Point);
```

```
}

/*
** cyl_to_world and fast_cyl_to_world :
**
** Input  : Frame = 3 vectors and 1 point defining a local frame
**          Point = unit cylindrical coordinates (in [-1,1]) in the local frame
**                  (Point.x=radius, Point.y=2*theta/PI, Point.z=z)
** Output : Point = cartesian coordinates in the world frame
*/
void cyl_to_world (realvec3 *Point, frame Frame)
{
  cyl_to_car (Point); car_to_world (Point, Frame);
}

void fast_cyl_to_world (realvec3 *Point, frame Frame)
{
  fast_cyl_to_car (Point); car_to_world (Point, Frame);
}

/*
** world_to_sph and fast_world_to_sph :
**
** Input  : Frame = 3 vectors and 1 point defining a local frame
**          Point = cartesian coordinates in the world frame
** Output : Point = unit spherical coordinates (in [-1,1]) in the local frame
**                  (Point.x=radius, Point.y=2/PI*theta, Point.z=2/PI*phi)
*/
void world_to_sph (realvec3 *Point, frame Frame)
{
  world_to_car (Point, Frame); car_to_sph (Point);
}

void fast_world_to_sph (realvec3 *Point, frame Frame)
{
  world_to_car (Point, Frame); fast_car_to_sph (Point);
}

/*
** sph_to_world and fast_sph_to_world :
**
** Input  : Frame = 3 vectors and 1 point defining a local frame
**          Point = unit spherical coordinates (in [-1,1]) in a local frame
**                  (Point.x=radius, Point.y=2/PI*theta, Point.z=2/PI*phi)
** Output : Point = cartesian coordinates in the world frame
*/
void sph_to_world (realvec3 *Point, frame Frame)
{
  sph_to_car (Point); car_to_world (Point, Frame);
}

void fast_sph_to_world (realvec3 *Point, frame Frame)
{
  fast_sph_to_car (Point); car_to_world (Point, Frame);
}
```

6 References

[BLAN95a] C. Blanc, *A Generic Implementation of Axial Deformations*, Graphics Gems V, Academic Press, 1995,

.

[BLAN95b] C.Blanc, *A Generic Implementation of Free Form Deformations*, Technical Report, 1995,

.

[GUIT93] P. Guitton, and C. Schlick , *A Methodology for Description of Texturing Methods*, Proc. of Fifth Eurographics Workshop on Rendering (Paris, France), pp 267-280, 1993,

.

[BLAN94] C. Blanc, P. Guitton, and C. Schlick , *A Methodology for Description of Geometrical Deformations*, Proc. of Pacific Graphics 94 (Beijing, China), 1994,

.

[FOLE90] J. Foley, and A. VanDam, and S. Feiner, and J. Hughes, *Computer graphics : principles and practice*, Addison-Wesley, 1990,

.

[PAET92] A. Paeth , *A Half-Angle Identity for Digital Computation : The Joys of the Halved Tangent*, Graphics Gems II, Academic Press, pp 381-386, 1992 ,

.