A VERY SIMPLE APPROACH FOR 3-D TO 2-D MAPPING

SANDIPAN DEY [1], AJITH ABRAHAM [2], SUGATA SANYAL [3]

[1] Anshin Soft ware Pvt. Ltd.
INFINITY, Tower - II, 10th Floor,
Plot No.- 43. Block - GP, Salt Lake Electronics Complex,
Sector - V, Kolkata - 700091
email: sandipand@anshinsoft.com

[2] IITA Professorship Program, School of Computer Science,
Yonsei University,
134 Shinchon-dong, Sudaemoon-ku, Seoul 120-749, Republic of Korea
email: ajith.abraham@ieee.org

[3] School of Technology & Computer Science
Tata Institute of Fundamental Research
Homi Bhabha Road, Mumbai - 400005, INDIA
email: sanyal@tifr.res.in

Abstract.

Many times we need to plot 3-D functions e.g., in many scientific experiments. To plot this 3-D functions on 2-D screen it requires some kind of mapping. Though OpenGL, DirectX etc 3-D rendering libraries have made this job very simple, still these libraries come with many complex pre-operations that are simply not intended, also to integrate these libraries with any kind of system is often a tough trial. This article presents a very simple method of mapping from 3-D to 2-D, that is free from any complex pre-operation, also it will work with any graphics system where we have some primitive 2-D graphics function. Also we discuss the inverse transform and how to do basic computer graphics transformations using our coordinate mapping system.

1 Introduction

We have a function $f : R^2 \rightarrow R$, and our intention is to draw the function in 2-D plane. The function $z = f(x,y)$ is a 2-variable function and each tuple $(x,y, f(x,y)) \in R^3$. Let's say we want to graphically plot f onto computer screen using a primitive graphics library (like Turbo C graphics), which supports only the basic *putPixel* (to draw a pixel in 2-D screen) -like 2-D rendering function, but no 3-D rendering; i.e., our graphics library's *putPixel*'s domain is R^2 and it's not R^3.

Hence in order to draw the function f using our graphics library, we must design a coordinate conversion system, that will provide us with a function that will take as input 3-tuples $(x,y, f(x,y))$ and produce as output a 2-tuple (x', y') that can be directly passed to our graphics library to plot it onto the screen, but with 3-D look & feel. As we discussed, it's essential that we have a simple coordinate mapping system that maps R^3 to R^2 and still gives us a hypothetical feeling of drawing 3-D functions. It's very easy to find such a map, i.e., a function $h : R^3 \rightarrow R^2$ and in this paper we try to find such a simple map.

2 Proposed approach

We have a pictorial representation (Fig.1) of our 3-D to 2-D mapping system:

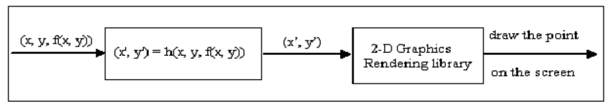

Fig. 1: Basic Model of a simple 3-D to 2-D mapping system

But, how the function f should look like after mapping and plotting? Here we simulate the 3-rd coordinate (namely Z) in our 2-D $x-y$ plane. We perform the logical to physical coordinate transform and everything by the map function h, which will basically turn out to be a 3×2 matrix. The basic mapping technique is shown in Fig. 2, which we are shortly going to explain.

If we have our Origin 0 at (x_0, y_0) screen coordinate, we have,

$$\begin{aligned} x' &= x_0 + y - x \cdot sin(\theta) \\ y' &= y_0 - z + x \cdot cos(\theta) \end{aligned} \tag{1}$$

i.e., we have our 3-D to 2-D transformation matrix:

$$M_{3\times 2} = \begin{bmatrix} -\sin(\theta) & \cos(\theta) \\ 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2}$$

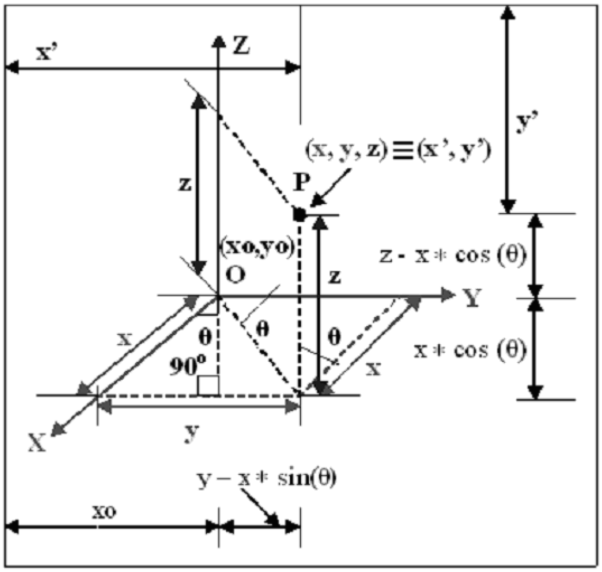

Fig. 2: The basic coordinate mapping

Again we have shifting (change of origin) by the matrix $O_{2D} = [x_0, y_0]$ so that $O_{2D} + P_{3D} \times M_{3\times 2} = P_{2D}$, here $\times$ denotes matrix multiplication and $+$ denotes matrix addition, the 3-tuple $P_{3D} = [xyz]$, the 2-tuple $P_{2D} = [x'y']$ i.e.

$$\begin{bmatrix} x_0 & y_0 \end{bmatrix} + \begin{bmatrix} x & y & z \end{bmatrix} \cdot \begin{bmatrix} -\sin(\theta) & \cos(\theta) \\ 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} x' & y' \end{bmatrix} \tag{3}$$

By default we keep the angle between $X - axis$ & $Z - axis = \theta = \frac{\pi}{4}$, that one can change if required, but with the following inequality strictly satisfied: $0^\circ < \theta < \frac{\pi}{2}$.

One can optionally use a compression factor to control the dimension along $Z - axis$ by a compression factor ρ_z and slightly modifying the equations:

$$\begin{aligned} x' &= x_0 + y - x \cdot sin(\theta) \\ y' &= y_0 - \rho_z \cdot z + x \cdot cos(\theta) \end{aligned} \tag{4}$$

Obviously, $0.0 < \rho_z \leq 1.0$

By default we take $\rho_z = 1.0$.

3 Sample output surfaces drawn using the above mapping

Following surfaces (Fig. 3 and Fig. 4) are drawn in Turbo C++ version 3.0 (BGI graphics) using the above simple 3-D to 2-D mapping.

4 Inverse Transformation - Obtaining original 3-D coordinates from the transformed 2-D coordinates

Here, our transformation function (matrix) is defined by Eqn. (1).

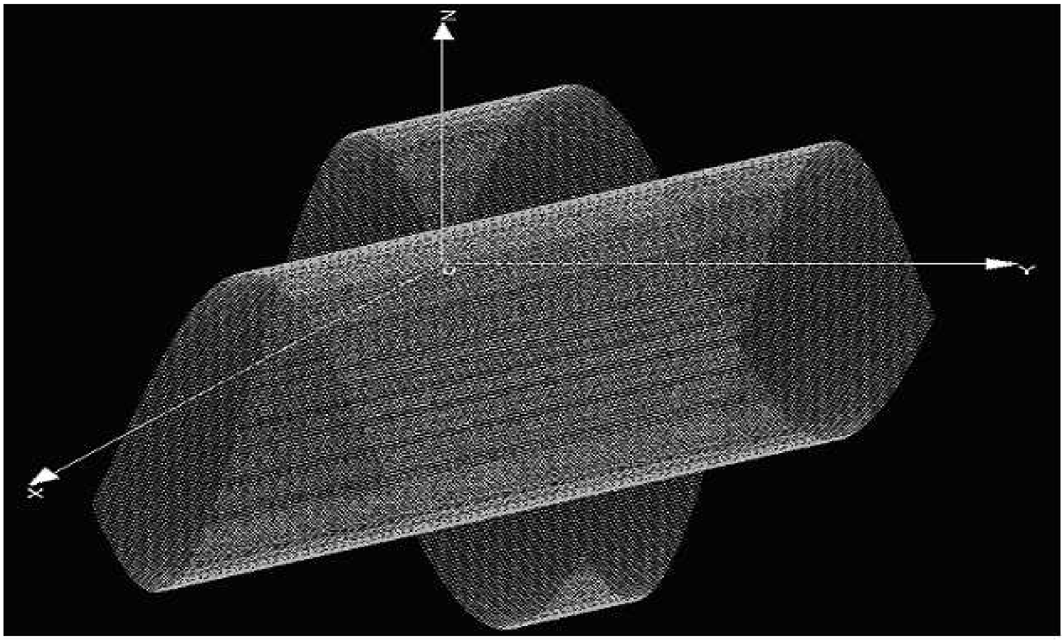

Fig. 3: Sine function drawn in TurboC++ Version 3.0 (BGI Graphics) using the 3-D to 2-D mapping

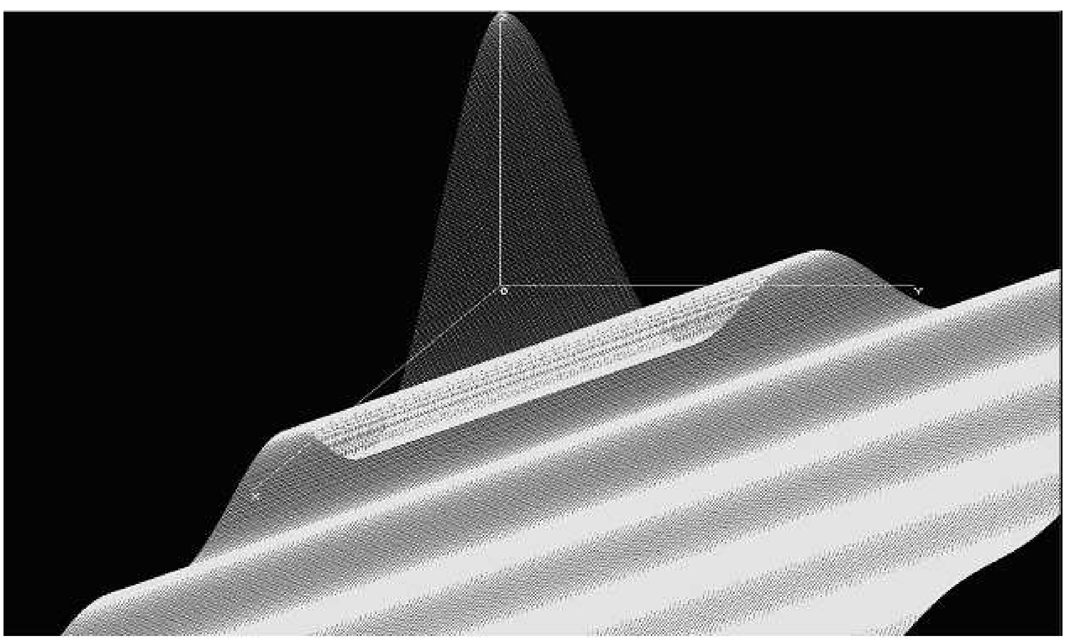

Fig. 4: Sync function drawn in TurboC++ Version 3.0 (BGI Graphics) using the 3-D to 2-D mapping

As we can see, it is impossible to re-convert and obtain the original set of coordinates, namely (x,y,z), because we have 3 unknowns and 2 equations. So, in order to be able to get the original coordinates back, we at least need to store 3 tuples as result of the transformation, for instance, $(x,y,z) \longrightarrow (x',y',z)$, the z-coordinate being stored only to get the inverse transform $(x',y',z) \rightarrow (x,y,z)$ and the (x',y') pair is used to plot the point. So, in order to get the inverse transformation, we need to solve the equations for x,y, since we already know z, we have 2-equations and 2 unknown variables:

$$\begin{aligned} y - x \cdot \sin(\theta) &= x' - x_0 \\ x \cdot cos(\theta) &= y' - y_0 + z \end{aligned} \tag{5}$$

solving the above 2 equations we get,

$$\begin{aligned} x &= (y' - y_0 + z) \cdot sec(\theta) \\ y &= x' - x_0 + (y' - y_0 + z) \cdot tan(\theta) \end{aligned} \tag{6}$$

Put it in another way, our transformation matrix is a 3×2 matrix and is done by Eqn. (2) since a non-square matrix, no question of existence of its inverse. So, in order to be able to get the inverse transform as well, we need a 3×3 invertible square matrix, e.g.,

$$M_{3\times3} = \begin{bmatrix} -\sin(\theta) & \cos(\theta) & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 1 \end{bmatrix} \tag{7}$$

with

$$Det(M_{3\times3}) = \det \begin{bmatrix} -\sin(\theta) & \cos(\theta) & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 1 \end{bmatrix} =$$

$$= 1 \cdot \begin{vmatrix} -\sin(\theta) & \cos(\theta) \\ 1 & 0 \end{vmatrix} = -\cos(\theta) \tag{8}$$

Now, $0 < \theta < \frac{\pi}{2}$, hence $cos(\theta) \neq 0$, hence $Det(M_{3\times3}) \neq 0$ and the inverse exists.

$$\begin{bmatrix} x_0 & y_0 & 0 \end{bmatrix} + \begin{bmatrix} x & y & z \end{bmatrix} \cdot \begin{bmatrix} -\sin(\theta) & \cos(\theta) & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 1 \end{bmatrix} =$$

$$= [x' \; y' \; z] \tag{9}$$

But, we have,

$$Inv(M_{3\times3}) = (M_{3\times3})^{-1} = \frac{Adj(M_{3\times3})}{Det(M_{3\times3})} \tag{10}$$

$$Det(M_{3\times3}) \neq 0$$

and,

$$Adj(M_{3\times3}) = \begin{bmatrix} 0 & -\cos(\theta) & 0 \\ -1 & -\sin(\theta) & 0 \\ -1 & -\sin(\theta) & -\cos(\theta) \end{bmatrix} \tag{11}$$

Hence,

$$Inv(M_{3\times3}) = (M_{3\times3})^{-1} = \begin{bmatrix} 0 & 1 & 0 \\ \sec(\theta) & \tan(\theta) & 0 \\ \sec(\theta) & \tan(\theta) & 1 \end{bmatrix} \quad (12)$$

here, $cos(\theta) \neq 0$.

So, the inverse transform is:

$$\begin{bmatrix} x & y & z \end{bmatrix} \cdot \begin{bmatrix} -\sin(\theta) & \cos(\theta) & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 1 \end{bmatrix} = \\ = \begin{bmatrix} x' & y' & z \end{bmatrix} - \begin{bmatrix} x_0 & y_0 & 0 \end{bmatrix} \quad (13)$$

$$\begin{bmatrix} x & y & z \end{bmatrix} = \\ = \begin{bmatrix} x' - x_0 & y' - y_0 & z \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 \\ \sec(\theta) & \tan(\theta) & 0 \\ \sec(\theta) & \tan(\theta) & 1 \end{bmatrix} \quad (14)$$

$$\begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} (y^{'} - y_0 + z) \cdot sec(\theta) & x^{'} - x_0 + (y^{'} - y_0 + z) \cdot tan(\theta) & z \end{bmatrix} \quad (15)$$

This exactly matches with our previous derivation.

5 Rotation and affine transformations

A point in 3-D, after being mapped to 2-D screen, following the above mapping procedure, may be required to be transformed using standard computer graphics transformations (translation, rotation about an axis etc). But in order to undergo such a graphics transformation and to show the point back to the screen after the transformation, it needs to go through the following steps in our previously-described coordinate mapping system:

- First obtain the inverse coordinate transformation to obtain the original 3-D coordinates from the mapped 2-D coordinates.
- Multiply the 3-D coordinate matrix by proper graphics transformation matrix in order to achieve graphical transformation.
- Use the same 3-D to 2-D map again to plot the point onto the screen.

These steps can be mathematically represented as:

- $P_{3D} = P_{2D} \times (M_{3\times3})^{-1}$
- $P'_{3D} = P_{3D} \times T_{3\times3}$
- $P'_{2D} = P'_{3D} \times M_{3\times3}$

Or, by a single line expression,

$$P'_{2D} = ((P_{2D} \times (M_{3\times3})^{-1}) \times T_{3\times3}) \times M_{3\times3})$$

Here, as before $\times$ denotes matrix multiplication, where $T_{3\times3}$ denotes the traditional graphics transformation matrix.

But, since we know the fact that matrix multiplication is associative, we have,

$$\begin{aligned} P'_{2D} &= ((P_{2D} \times (M_{3\times3})^{-1}) \times T_{3\times3}) \times M_{3\times3} \\ &= P_{2D} \times (M_{3\times3})^{-1} \times T_{3\times3} \times M_{3\times3} \\ &= P_{2D} \times (M_{3\times3})^{-1} \times T_{3\times3} \times M_{3\times3}) \end{aligned} \tag{16}$$

$$P'_{2D} = P_{2D} \times M'$$

where $M' = (M_{3\times3})^{-1} \times T_{3\times3} \times M_{3\times3}$

So, using this simple technique we can escape the 3 successive matrix multiplications every-time a point on screen needs to transformed - instead we can pre-compute the matrix $M' = (M_{3\times3})^{-1} \times T_{3\times3} \times M_{3\times3}$.

This matrix M' is needed to be computed once for a given graphics transformation (e.g., rotation about an axis) and applied to all points on the screen, so that using a single matrix multiplication thereafter any point on the screen can undergo graphics transformation, by, $P'_{2D} = P_{2D} \times M'$, where P_{2D} represents the point mapped before transformation $T_{3\times3}$ and P'_{2D} is the point re-mapped after the transformation, as obvious.
Hence, using the above tricks we are able to make the transformation more computationally efficient.

Moreover, if a transformation is needed to be applied simultaneously, we can use the property $M^{-1}_{3\times3} \times (T_{3\times3})^n \times M_{3\times3} = (M^{-1}_{3\times3} \times T_{3\times3} \times M_{3\times3})^n$, where $(T_{3\times3})^n$ denotes (n times, n is a positive integer) simultaneous matrix multiplication of $T_{3\times3}$. Let's say we have already undergone $T_{3\times3}$ a transformation, so that we have already computed $M' = (M^{-1}_{3\times3} \times T_{3\times3} \times M_{3\times3}$, and let's say that we also have frequent simultaneous $(T_{3\times3})^n$ transformation. In order to undergo a $(T_{3\times3})^n$ transformation, we first need to compute the matrix $(T_{3\times3})^n$, then we need to compute our new matrix $M'' = (M^{-1}_{3\times3} \times (T_{3\times3})^n \times M_{3\times3}$, so we need total $n+2$ matrix multiplications, every-time we want a $(T_{3\times3})^n$ transform, for each n. But if we have computed $M^{-1}_{3\times3} \times T_{3\times3} \times M_{3\times3}$ initially, here the trick is that we can reuse this it to compute our new matrix in the following manner:

$$M'' = M_{3\times3}^{-1} \times (T_{3\times3})^n \times M_{3\times3} =$$
$$= (M_{3\times3}^{-1} \times T_{3\times3} \times M_{3\times3})^n = (M')^n$$

Here we need not compute $(T_{3\times3})^n$ and M'' everytime, instead we need to compute $(M')^n$ only (that can be incremental multiplication to increase efficiency).

6 Conclusions

This article presented a very simple method of mapping from 3-D to 2-D, that is free from any complex pre-operation. The proposed technique works with any graphics system where we have some primitive 2-D graphics function. We also discussed the inverse transform and how to do basic computer graphics transformations using our coordinate mapping system.

7 References

[1] David F. Rogers, J. Alan Adams, *Mathematical Elements for Computer Graphics*, McGraw-Hill

[2] David F. Rogers, *Procedural elements for computer Graphics*, United States Naval Academy, Annapolis, MD

[3] Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, *OpenGL Programming Guide, The Official Guide to Learning OpenGL*, Version 1.4, Fourth Edition.

[4] Ken Turkowski, *The Use of Coordinate Frames in Computer Graphics*, *Graphics Gems I*, Academic Press, 1990, pp. 522-532.

[5] Ken Turkowski, *Fixed-Point Trigonometry with CORDIC Iterations*, *Graphics Gems I*, Academic Press, 1990, pp. 494-497.

[6] C. M. Ng, D. W. Bustard, A New Real Time Geometric Transformation Matrix and its Efficient VLSI Implementation, Computer Graphics Forum, Volume 13 Page 285