Computer Graphics and Three-Dimensional Modelling

Paulo Menezes

DEEC - FCTUC

14th January 2010
2D Transformations

Let

\[ P = [xy]^T \]

and

\[ P' = [x'y']^T \]

The translation

\[ T(dx, dy) = \begin{bmatrix} dx \\ dy \end{bmatrix} \]

can be applied to a point so that

\[ P + T(...) = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \end{bmatrix} = P' \]
The scaling operations is defined as

\[ S(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \]

so that

\[ S(...).P = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix} \]
Rotations

\[ R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \]

and

\[ R(...).P = \begin{bmatrix} x \cos \theta - y \sin \theta \\ z \sin \theta + y \cos \theta \end{bmatrix} \]  \hspace{1cm} (1)

Derivation of rotation

\[ x = r \cos \phi \]
\[ y = r \sin \phi \]

\[ x' = r \cos(\phi + \theta) = r \cos \phi \cos \theta - r \sin \phi \sin \theta \]
\[ y' = r \sin(\phi + \theta) = r \cos \phi \sin \theta - r \sin \phi \cos \theta \]

substituting in the above we get (1).
Shear transformation

\[ \text{SH}_x(a) = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \]

\[ \text{SH}_y(b) = \begin{bmatrix} 1 & 0 \\ b & 1 \end{bmatrix} \]

\[ P' = \text{SH}_x(a).P = \begin{bmatrix} x + ay \\ y \end{bmatrix} \]
Considering the unit vectors $i = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $j = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, applying a rotation to these vectors we get

$$R(\theta).i = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \cos \theta \\ \sin \theta \end{bmatrix} = i'$$

$$R(\theta).j = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -\sin \theta \\ \cos \theta \end{bmatrix} = j'$$

Putting together we get the usual

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = R(\theta)$$

**Definition**

The column space (row space) of a matrix is the coordinate space formed by the columns (rows) of the matrix, taken as vectors.
The column space of $R(\theta)$ above is the space formed by the vectors $i'$ and $j'$.

Now by taking the rows of $R(\theta)$ as $i''^T$ and $j''^T$ and rotating them by $R(\theta)$ itself, we get

$$R(\theta).i'' = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} \cos \theta \\ -\sin \theta \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = i$$

$$R(\theta).j'' = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \cdot \begin{bmatrix} \sin \theta \\ \cos \theta \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = j$$

Showing together

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}_{R(\theta)} \cdot \begin{bmatrix} \cos \theta \\ -\sin \theta \end{bmatrix}_{R(\theta)^T} \begin{bmatrix} \sin \theta \\ \cos \theta \end{bmatrix} = I$$

This is not surprising as the rotation matrices are (square) orthogonal with determinant equal to 1. So $R^T = R^{-1}$. 
The set of all $n \times n$ rotation matrices, together with the matrix multiplication operator forms a group, known as rotation group or special orthogonal group. Recal: Group is an algebraic structure, consisting of a set together with an operation that combines any two of its elements to form a third element, verifying closure, associativity, identity and invertibility.
Homogeneous coordinates

Goal: create a representation in which translation can be produced by a matrix multiplication.

Why?

- all the transformations (seen till now) become multiplications
- allows easy composition of transformations

The solution is to pass from 2D Cartesian coordinates to 3D homogeneous coordinates \((x, y) \rightarrow (x, y, w)\). This mapping is done just by adding a weight \(w\) as a third coordinate.

By multiplying the coordinates of \((x, y, w)\) by a scalar, i.e. \(\alpha(x, y, w) = (\alpha x, \alpha y, \alpha w)\) we obtain the same point in 2D Cartesian coordinates. This multiplication produces a line in 3D homogeneous coordinates.
To homogenize a point in 3D we divide by \( w \) and obtain \((\frac{x}{w}, \frac{y}{w}, 1)\). This projects the point onto the \( w = 1 \) plane (along the “projection line’’), \( w = 0 \) represents a “point at the inifinity” and such point cannot be homogenized (as it lies on the x-y plane). \((0,0,0)\) is not allowed.
Thus, the 2D Cartesian coordinates correspond to the points with \( w=1 \) in the 3D homogeneous space.
The translation is represented by

\[ T(dx, dy) = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \]

so applying to a point becomes

\[ T(...).P = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ 1 \end{bmatrix} \]
Scaling:

\[ S(s_x, s_y) = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

Rotation:

\[ R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

Shear:

\[ SH_x(a) = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

\[ SH_y(b) = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

The composition of these transformations becomes the chain of the multiplication of the corresponding matrices. NOTE: order matters as matrix multiplication is not commutative.
3D Cartesian coordinates are mapped in 4D homogeneous coordinates by adding a weight coordinate. Homogenization as before.

Conversion of right-hand coordinates into left-hand coordinates

\[ P_{lh} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot P_{rh} \]

Translation

\[ T(dx, dy, dz) = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Scaling

\[ S(S_x, S_y, S_z) = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

Rotation

\[ R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

\[ R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

\[ R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Inverse Transformations

Translation:

\[ T^{-1}(dx, dy, dz) = T(-dx, -dy, -dz) \]

Scaling:

\[ S^{-1}(sx, sy, sz) = S\left(\frac{1}{sz}, \frac{1}{sy}, \frac{1}{sz}\right) \]

Rotation:

\[ R^{-1}_{\{x,y,z\}}(\theta) = R_{\{x,y,z\}}(-\theta) = R^T_{\{x,y,z\}}(\theta) \]
Applying Transformations

Points: Already seen
Lines: Transform their end points
Transforming planes

Planes: A plane defined by

\[ Ax + By + Cz + D = 0 \]

has normal

\[ N = \begin{bmatrix} A & B & C & D \end{bmatrix}^T \]

A point \( P = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^T \)

belongs to the plane if

\[ N^T P = 0 \]
To maintain this after a transformation of $P$ by some matrix $M$, we need to find $Q$ such that

$$(QN)^T(MP) = 0.$$ 

By associativity we get

$$N^T(Q^TM)P = 0.$$ 

This will hold if $Q^TM$ is a multiple of $I$. For the case if $I$

$$Q^T = M^{-1}$$

or

$$Q = (M^{-1})^T$$
Now we know how to apply a rotation (or a sequence of rotations) to points, lines and planes.

How can we compute the rotation matrix given the axle of rotation and the rotation angle?

To rotate a 3D vector \( \mathbf{v} \) about a unitary axis \( \mathbf{z} \) by an angle \( \theta \) we can use the Rodrigues formula

\[
\mathbf{v}' = \mathbf{v} \cos \theta + (\mathbf{z} \times \mathbf{v}) \sin \theta + \mathbf{z}(\mathbf{z} \cdot \mathbf{v})(1 - \cos \theta)
\]
From this formula to obtain the rotation matrix we define the “cross product matrix”

\[
[z]_x \mathbf{v} = z \times \mathbf{v} = \begin{bmatrix}
0 & -z_3 & z_2 \\
z_3 & 0 & -z_1 \\
-z_2 & z_1 & 0 \\
\end{bmatrix} \mathbf{v}
\]

so the Rodrigues formula can be written as

\[
\mathbf{v}' = (I \cos \theta)\mathbf{v} + ([z]_x \sin \theta)\mathbf{v} + (1 - \cos \theta)\mathbf{z}\mathbf{z}^T \mathbf{v}
\]

\[
= (I \cos \theta + [z]_x \sin \theta + (1 - \cos \theta)\mathbf{z}\mathbf{z}^T)\mathbf{v}
\]

\[
= R\mathbf{v}
\]

so

\[
R = I + \sin \theta[z]_x + (1 - \cos \theta)(\mathbf{z}\mathbf{z}^T - I)
\]

as

\[
\mathbf{z}\mathbf{z}^T = [z]_x + I
\]

we can write it as

\[
R = I + [z]_x \sin \theta + (1 - \cos \theta)[z]_x^2
\]
Hamilton 1843: Complex numbers represent points in a plane, can't them be used to represent points in space?

Quaternion: \( q = a + bi + cj + dk \) such that the abstract symbols \( i, j, k \) satisfy the rules

\[
i^2 = j^2 = k^2 = ijk = -1
\]

noncommutative multiplication

\[
(a + bi + cj + dk)(e + fi + gj + hk) = \\
= (ae - bf - cg - dh) + (af + be + ch - dg)i + \\
+ (ag + ce + df - bh)j + (ah + de + bg - cf)k
\]
The imaginary part of a quaternion behaves like a vector \( \mathbf{v} = (b, c, d) \) in three dimensional vector space and the real part behaves like a scalar in \( \mathbb{R} \).

It is common to define them as a scalar plus a vector

\[
a + bi + cj + dk = a + \mathbf{v}
\]

we can now replace the rule \( i^2 = j^2 = k^2 = ijk = -1 \) by the vector multiplication rule

\[
\mathbf{v} \mathbf{w} = \mathbf{v} \times \mathbf{w} - \mathbf{v} \cdot \mathbf{w}
\]

From this rule we get

\[
(s + \mathbf{v})(t + \mathbf{w}) = (st - \mathbf{v} \cdot \mathbf{w}) + (s\mathbf{w} + t\mathbf{v} + \mathbf{v} \times \mathbf{w})
\]
For a quaternion \( q = w + xi + yj + zk \), its conjugate is
\[
q^* = w - xi - yj - zk
\]
and its magnitude is
\[
||q|| = \sqrt{q \times q^*} = \sqrt{w^2 + x^2 + y^2 + z^2}
\]
The multiplicative inverse of a nonzero quaternion can be computed as
\[
q^{-1} = \frac{q^*}{q \times q^*}
\]
If a quaternion has length 1, it is said a unit quaternion
\[
||q|| = 1 \equiv q^{-1} = q^*
\]
Quaternions are associative
\[
(q_1 \times q_2) \times q_3 = q_1 \times (q_2 \times q_3)
\]
Quaternions are not commutative
\[
q_1 \times q_2 \neq q_2 \times q_1
\]
Describing rotations with quaternions

The quaternion that represents the rotation about the unit vector \( \mathbf{u} \) by an angle \( \theta \) is

\[
\mathbf{q} = (\cos \frac{\theta}{2}, \mathbf{u} \sin \frac{\theta}{2})
\]

Representing a point \( \mathbf{p} \) by the quaternion \( \mathbf{P} = (0, \mathbf{p}) \) its rotation is obtained by

\[
\mathbf{P}_{\text{rotated}} = \mathbf{qPq}^{-1}
\]

Concatenation of rotations:

Suppose that \( \mathbf{q}_1 \) and \( \mathbf{q}_2 \) are unit quaternions representing two rotations.

\[
\mathbf{q}_2 \times (\mathbf{q}_1 \times \mathbf{P} \times \mathbf{q}_1^{-1}) \times \mathbf{q}_2^{-1} = (\mathbf{q}_2 \times \mathbf{q}_1) \times \mathbf{P} \times (\mathbf{q}_1^{-1} \times \mathbf{q}_2^{-1}) = (\mathbf{q}_2 \times \mathbf{q}_1) \times \mathbf{P} \times (\mathbf{q}_2 \times \mathbf{q}_1)^{-1}
\]
From quaternions to rotation matrices

Now that both rotation axis $\mathbf{v}_r$ and the rotation angle $\alpha$ are known, they can be used to construct a quaternion

$$\mathbf{q} = \cos(\alpha/2) + \sin(\alpha/2)(\mathbf{v}_x \mathbf{i} + \mathbf{v}_y \mathbf{j} + \mathbf{v}_z \mathbf{k})$$

and from its coefficients obtain the required rotation matrix, by noting $w = \cos(\alpha/2)$, $x = \sin(\alpha/2)\mathbf{v}_x$, $x = \sin(\alpha/2)\mathbf{v}_y$, $x = \sin(\alpha/2)\mathbf{v}_z$,

$$\mathbf{R} =
\begin{bmatrix}
w^2 + x^2 - y^2 - z^2 & 2xy + 2wz & 2xz - 2wy & 0 \\
2xy - 2wz & w^2 - x^2 + y^2 - z^2 & 2yz + 2wx & 0 \\
2xz - 2wy & 2yz - wx & w^2 - x^2 - y^2 + z^2 & 0 \\
0 & 0 & 0 & w^2 + x^2 + y^2 + z^2
\end{bmatrix}$$

It should be noted that the element $R_{(4,4)} = 1$, as it represents the expression of the norm of a rotation quaternion which has, by definition, unitarian norm.
Now all the rotations can be composed using quaternion multiplication and the final result can be converted into the rotation matrix.

\[
R = \begin{bmatrix}
1 - 2y^2 - 2z^2 & 2xy + 2wz & 2xz - 2wy & 0 \\
2xy - 2wz & w^2 - x^2 + y^2 - z^2 & 2yz + 2wx & 0 \\
2xz - 2wy & 2yz - wx & w^2 - x^2 - y^2 + z^2 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Use of transformations

Basic 2D Transformations

- Translation:
  \[ x' = x + tx \]
  \[ y' = y + ty \]

- Scale:
  \[ x' = x \times sx \]
  \[ y' = y \times sy \]

- Shear:
  \[ x' = x + hx \times y \]
  \[ y' = y + hy \times x \]

- Rotation:
  \[ x' = x \cos \theta - y \sin \theta \]
  \[ y' = x \sin \theta + y \cos \theta \]
Basic 2D Transformations

- Translation:
  \[ x' = x + tx \]
  \[ y' = y + ty \]

- Scale:
  \[ x' = x \times sx \]
  \[ y' = y \times sy \]

- Shear:
  \[ x' = x + hx \times y \]
  \[ y' = y + hy \times x \]

- Rotation:
  \[ x' = x \cos ! - y \sin ! \]
  \[ y' = x \sin ! + y \cos ! \]

\[ x' = (x \times sx) \cos ! + tx \]
\[ y' = (x \times sy) \sin ! + ty \]
Linear Transformations

- Scale
- Rotation
- Shear
- Mirror

Properties
- Satisfies: $T(s_1 p_1 + s_2 p_2) = T(s_1 p_1) + T(s_2 p_2)$
- Origin maps to origin
- Lines map to lines
- Parallel lines remain parallel
- Ratios are preserved
- Closed under composition
Affine Transformations

- Linear Transformations and Translations

Properties:
- Origin does not map to origin
- Lines map to lines
- Parallel lines remain parallel
- Ratios are preserved
- Closed under composition
Affine transformations and
projective warps

Properties:
- Origin does not map to origin
- Lines map to lines
- Parallel lines do not always remain parallel
- Ratios are not preserved (but cross-ratios are)
- Closed under composition
• Each level stores matrix representing transformation from parent’s coordinate system

Robot Arm

Angel Figures 8.8 & 8.9
• Well-suited for humanoid characters
Transformation hierarchies

• An object may appear in a scene multiple times

Draw same 3D data with different transformations
Transformation hierarchies

pgflastimage
Many applications use rendering of 3D polygons with direct illumination.
Each ray must be tested for intersection with some subset of all the objects in the scene. Once the nearest object has been identified, the algorithm will estimate the incoming light at the point of intersection, examine the material properties of the object, and combine this information to calculate the final color of the pixel. Certain illumination algorithms and reflective or translucent materials may require more rays to be re-cast into the scene.
Ray casting can be seen as a simplified and faster version of ray tracing, where only the closest object intersection is considered.
3D Geometric Primitives

Modeling Transformation

Lighting

Viewing Transformation

Projection Transformation

Clipping

Scan Conversion

Image

3D Rendering Pipeline

Drawing a 3D primitive onto a 3D image involves a pipelined sequence of operations.

```cpp
 glBegin(GL_POLYGON)
 glVertex3f(0.0, 0.0, 0.0);
 glVertex3f(1.0, 0.0, 0.0);
 glVertex3f(1.0, 1.0, 0.0);
 glVertex3f(0.0, 1.0, 0.0);
 glEnd();
```
3D Rendering Pipeline

3D Geometric Primitives

- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Scan Conversion
- Image

3D Geometric Primitives

```c
glBegin(GL_POLYGON)
glVertex3f(0.0, 0.0, 0.0);
glVertex3f(1.0, 0.0, 0.0);
glVertex3f(1.0, 1.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glEnd();
```

Drawing a 3D primitive onto a 3D image involves a pipelined sequence of operations.
3D Rendering Pipeline

- **Transform into 3D world coordinate system**
  - Illuminate according to lighting and reflectance
  - Transform into 3D camera coordinate system
  - Transform into 2D camera coordinate system
  - Clip primitives outside camera’s view
  - Draw pixels (including texturing, hidden surface removal, etc.)
3D Rendering Pipeline

- **Transform into 3D world coordinate system**
- **Illuminate according to lighting and reflectance**
- **Transform into 3D camera coordinate system**
- **Transform into 2D camera coordinate system**
- **Clip primitives outside camera’s view**
- **Draw pixels (including texturing, hidden surface removal, etc.)**
3D Geometric Primitives

- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Scan Conversion
- Image

- Transform into 3D world coordinate system
- Illuminate according to lighting and reflectance
- **Transform into 3D camera coordinate system**
- Transform into 2D camera coordinate system
- Clip primitives outside camera’s view
- Draw pixels (including texturing, hidden surface removal, etc.)
3D Rendering Pipeline

3D Geometric Primitives

- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Scan Conversion
- Image

- Transform into 3D world coordinate system
- Illuminate according to lighting and reflectance
- Transform into 3D camera coordinate system
- Transform into 2D camera coordinate system
- Clip primitives outside camera's view
- Draw pixels (including texturing, hidden surface removal, etc.)
3D Rendering Pipeline

3D Geometric Primitives

- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Scan Conversion
- Image

- Transform into 3D world coordinate system
- Illuminate according to lighting and reflectance
- Transform into 3D camera coordinate system
- Transform into 2D camera coordinate system
- Clip primitives outside camera’s view
- Draw pixels (including texturing, hidden surface removal, etc.)
3D Rendering Pipeline

**3D Geometric Primitives**

- **Modeling Transformation**
- **Lighting**
- **Viewing Transformation**
- **Projection Transformation**
- **Clipping**
- **Scan Conversion**

- **Image**

- Transform into 3D world coordinate system
- Illuminate according to lighting and reflectance
- Transform into 3D camera coordinate system
- Transform into 2D camera coordinate system
- Clip primitives outside camera’s view
- **Draw pixels** (including texturing, hidden surface removal, etc.)
Transformations map points from one coordinate system to another.

$p(x,y,z)$

3D Object Coordinates

Modeling Transformation

3D World Coordinates

Viewing Transformation

3D Camera Coordinates

Projection Transformation

3D Camera Coordinates

Window-to-Viewport Transformation

2D Screen Coordinates

2D Image Coordinates

$p'(x',y')$
Viewing transformations

\[ p(x,y,z) \]

3D Object Coordinates

Modeling Transformation

3D World Coordinates

Viewing Transformation

3D Camera Coordinates

Projection Transformation

2D Screen Coordinates

Window-to-Viewport Transformation

2D Image Coordinates

\[ p'(x',y') \]
Viewing transformations

- mapping from world to camera coordinates
  - origin moves to eye position
  - Up vector to Y axis
  - Right vector to X axis

- Camera
  - Back
  - Up
  - Right

- World
  - X
  - Y
  - Z

- View plane
Camera Coordinates

- Canonical coordinate system
  - By convention is right-handed (looking down -z axis)
  - Convenient for projection, clipping, etc.

Camera up vector
maps to Y axis

Camera back vector
maps to Z axis
(pointing out of page)

Camera right vector
maps to X axis
The transformation matrix maps camera basis vectors to canonical vectors in camera coordinate system.

\[
M = \begin{bmatrix}
R_x & U_x & B_x & E_x \\
R_y & U_y & B_y & E_y \\
R_z & U_z & B_z & E_z \\
R_w & U_w & B_w & E_w
\end{bmatrix}^{-1}
\]
Projection

- General definition: Transforms points in n-space to m-space (m<n)
- In computer graphics: Map 3D camera coordinates to 2D screen coordinates
Projection

- General definition: Transforms points in n-space to m-space (m<n)
- In computer graphics: Map 3D camera coordinates to 2D screen coordinates
Taxonomy of Projections

Planar geometric projections

Parallel

Orthographic
- Top (plan)
- Front elevation

Axonometric
- Side elevation

Isometric

Oblique
- Cabinet
- Cavalier

One-point
- Two-point
- Three-point

Perspective

Other
Parallel Projection

Center of projection is at infinity
Direction of projection (DOP) is the same for all points

Angel Figure 5.4
Orthographic Projection

DOP perpendicular to view plane

Angel Figure 5.5

Top Side
Front
Side

Paulo Menezes (DEEC - FCTUC)
Oblique Projections

DOP **no** perpendicular to view plane

Cavalier
(DOP at $45^\circ$)

Cabinet
(DOP at $63.4^\circ$)
Parallel Projection View Volume

H&B Figure 12.30

Parallelepiped View Volume

Back Plane

Front Plane

window

H&B Figure 12.24

• DOP not perpendicular to view plane

Cavalier (DOP at 45°)

Cabinet (DOP at 63.4°)
Parallel Projection Matrix

The parallel projection matrix is given by:

\[
\begin{bmatrix}
x_s \\
y_s \\
z_s \\
w_s
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & L_1 \cos \phi & 0 \\
0 & 1 & L_1 \sin \phi & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x_c \\
y_c \\
z_c \\
1
\end{bmatrix}
\]

Where:
- \(x_s, y_s, z_s, w_s\) are the projected coordinates.
- \(x_c, y_c, z_c\) are the original coordinates.
- \(L_1\) is the distance from the view plane to the projection plane.
- \(\phi\) is the angle between the view plane and the projection plane.

This matrix transforms the original coordinates \((x, y, z)\) to the projected coordinates \((x_p, y_p)\) on the view plane.
Taxonomy of Projections

Planar geometric
projections

Parallel

Orthographic

Top (plan)

Front elevation

Axonometric

Side elevation

Isometric

Oblique

Cabinet

Cavalier

One-point

Two-point

Three-point

Perspective

Other

Other

FVFHP Figure 6.10
Map 3D points onto “view plane” along projecting rays (projectors) emanating from the “center of projection” (COP)
Perspective Projections view Volume

H&B Figure 12.30

View Plane

Frustum View Volume

Back Plane

Front Plane

window

Projection Reference Point

$z_v$
Perspective Projections

Computing 2D coordinates from 3D coordinates with similar triangles.

What are the coordinates of the point resulting from projection of \((x,y,z)\) onto the view plane?
Computing 2D coordinates from 3D coordinates with similar triangles.
4x4 matrix representation?

\[
x_s = x_c \frac{D}{z_c}
\]

\[
y_s = y_c \frac{D}{z_c}
\]

\[
z_s = D
\]

\[
w_s = 1
\]

\[
x' = x_c
\]

\[
y' = y_c
\]

\[
z' = z_c
\]

\[
w' = z_c / D
\]

\[
\begin{bmatrix}
x_s \\
y_s \\
z_s \\
w_s
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/D & 0
\end{bmatrix}
\begin{bmatrix}
x_c \\
y_c \\
z_c \\
1
\end{bmatrix}
\]
Perspective Projections

Traditional form used in computer vision

\[
\begin{bmatrix}
U \\
V \\
S
\end{bmatrix} = \begin{bmatrix}
-f & 0 & 0 & 0 \\
0 & -f & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix} \begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\] (2)

where \( u = U/S \) and \( v = V/S \) if \( S \neq 0 \).
Perspective vs. Parallel

- **Perspective projection**
  - + Size varies inversely with distance - realistic appearance
  - - Distance and angles are not (in general) preserved
  - - Parallel lines do not (in general) remain parallel

- **Parallel projection**
  - + Good for exact measurements
  - + Parallel lines remain parallel
  - - Angles are not (in general) preserved
  - - Less realistic looking
After projecting the 3D objects onto the image plane, need to remove those that fall outside the visible area.
Is point inside window?

\[
\text{inside} = \\
(x \geq wx1) \land \land \\
(x \leq wx2) \land \land \\
(y \geq wy1) \land \land \\
(y \leq wy2); \\
\]
Is line or portion of it inside window?

2>After Clipping
Line Clipping

Is line or portion of it inside window?

2>After Clipping
Cohen Sutherland Algorithm

Bit 1  |  Bit 2  |  Bit 3  |  Bit 4
--- | --- | --- | ---
1010 | 0010 | 0110 | 0101
1000 | 0000 | 0001 | 1001

P₁  |  P₂  |  P₃  |  P₄  |  P₅  |  P₆  |  P₇  |  P₈  |  P₉  |  P₁₀
Cohen Sutherland Algorithm
Cohen Sutherland Algorithm
Cohen Sutherland Algorithm

Bit 1

Bit 2

Bit 3

Bit 4
Cohen Sutherland Algorithm
Polygon Clipping

Paulo Menezes (DEEC - FCTUC)
CGM3D
14th January 2010
Polygon Clipping
Sutherland Hodgeman Algorithm
Sutherland Hodgeman Algorithm
Sutherland Hodgeman Algorithm
Boundary Clipping

- Window Boundary
- Inside
- Outside

Points: P₁, P₂, P₃, P₄, P₅
Boundary Clipping

Window Boundary

Inside

Outside

P_1

P_2

P_3

P_4

P_5
Boundary Clipping

Window Boundary

Inside

Outside

P1, P2, P3, P4, P5
Boundary Clipping

Window Boundary

Inside
Outside

P1
P2
P3
P4
P5
Boundary Clipping

Window Boundary

Inside

Outside

P_1
P_2
P_3
P_4
P_5
P'

Boundary Clipping

Window Boundary

Inside

Outside

P₁

P₂

P₃

P₄

P₅

P'
Boundary Clipping
Boundary Clipping

Window Boundary

P′

P′′

P₁

P₂

Inside

Outside
Viewport Transformation

Screen

Image

Window

Viewport
Viewport Transformation

\[ \begin{align*}
  vx &= vx_1 + (wx - wx_1) \times (vx_2 - vx_1) / (wx_2 - wx_1); \\
  vy &= vy_1 + (wy - wy_1) \times (vy_2 - vy_1) / (wy_2 - wy_1);
\end{align*} \]
Transformation Summary

\[ p(x, y, z) \]

- **Modeling Transformation**
  - 3D Object Coordinates
  - 3D World Coordinates
  - **Viewing Transformation**
    - 3D Camera Coordinates
    - **Projection Transformation**
      - 2D Screen Coordinates
      - **Window-to-Viewport Transformation**
        - 2D Image Coordinates
        - \( p'(x', y') \)

- **Modeling transformation**
- **Viewing transformations**
- **Viewport transformation**
Scan Conversion

Render image of a geometric primitive by setting pixel colors.

```c
void SetPixel(int x, int y, Color rgba);
```

Ex: Filling a triangle

![Triangle example](image-url)
Scan Conversion

Render image of a geometric primitive by setting pixel colors.

```c
void SetPixel(int x, int y, Color rgba);
```

Ex: Filling a triangle

![Diagram of a triangle with vertices P1, P2, P3, P4]
void ScanTriangle(Triangle T, Color rgba){
    for each pixel P at (x,y){
        if (Inside(T, P))
            SetPixel(x,y,rgba);
    }
}
A point is inside a triangle if it is in the positive half space of all three boundary lines:

- triangle vertices are ordered counter-clockwise
- point must be on the left side of every boundary
Boolean Inside(Triangle T, Point P) {
for each boundary line L of T {
Scalar d=L.a*P.x + L.b*P.y + L.c;
if (d < 0.0) return False;
}
return True;
}
Previous algorithm has a big disadvantage, which is.... (?)
Arbitrary Polygons

Concave

Self-Intersecting

With Holes
Arbitrary Polygons
Shading

Angel Figure 6.34
Shading

- Flat Shading
- Gouraud Shading - bilinear interpolation of colors at vertices
- Phong Shading - bilinear interpolation of normals at vertices
Shading

Wireframe  Flat

Gouraud  Phong