Computer Graphics

Camera & Projective Transformations

Philipp Slusallek
Motivation

- Rasterization works on 2D primitives (+ depth)
- Need to project 3D world onto 2D screen
- Based on
  - Positioning of objects in 3D space
  - Positioning and parameters of the virtual camera
Coordinate Systems

- **Local (object) coordinate system (3D)**
  - Object vertex positions
  - Can be hierarchically nested in each other (scene graph, transf. stack)

- **World (global) coordinate system (3D)**
  - Scene composition and object placement
    - Mostly rigid objects: translation, rotation per object, (scaling)
    - Animated objects: time-varying transformation in world or local space
  - Illumination can be computed in this space

- **Camera/view/eye coordinate system (3D)**
  - Coordinates relative to camera pose (position & orientation)
    - Camera itself specified relative to world space
  - Illumination can also be done in this space

- **Normalized device coordinate system (2.5D)**
  - After *perspective transformation*, rectilinear, in $[0, 1]^3$
  - Normalization to view frustum (for rasterization and depth buffer)
  - Rasterization & shading done here (e.g., interpolation across triangle)

- **Window/screen (raster) coordinate system (2D)**
  - 2D transformation to place image in window on the screen
Hierarchical Coordinate Systems

- **Used in Scene Graphs**
  - Group objects hierarchically
  - Local coordinate system is relative to parent coordinate system
  - Apply transformation to the parent to change the whole sub-tree (or sub-graph)
Hierarchical Coordinate Systems

- **Hierarchy of transformations**

  - $T_{\text{root}}$ Positions the character in the world
  - $T_{\text{ShoulderR}}$ Moves to the right shoulder
  - $T_{\text{ShoulderRJoint}}$ Rotates in the shoulder (3 DOF) [$\leftarrow$ User]
  - $T_{\text{UpperArmR}}$ Moves to the Elbow
  - $T_{\text{ElbowRJoint}}$ Rotates in the Elbow (1 DOF) [$\leftarrow$ User]
  - $T_{\text{LowerArmR}}$ Moves to the wrist
  - $T_{\text{WristRJoint}}$ Rotates in the wrist (1 DOF) [$\leftarrow$ User]
  - Further for the right hand and the fingers

  - $T_{\text{ShoulderL}}$ Moves to the left shoulder
  - $T_{\text{ShoulderLJoint}}$ Rotates in the shoulder (3 DOF) [$\leftarrow$ User]
  - $T_{\text{UpperArmL}}$ Moves to the Elbow
  - $T_{\text{ElbowLJoint}}$ Rotates in the Elbow (1 DOF) [$\leftarrow$ User]
  - $T_{\text{LowerArmL}}$ Moves to the wrist
  - Further for the left hand and the fingers

  - Each transformation is relative to its parent
    - Concatenated by multiplying (from right) and pushing onto a stack
    - Going back by popping from the stack
  - This transformation stack was so common, it was built into OpenGL
Coordinate Transformations

• **Model transformation**
  - Object space to world space
  - Can be hierarchically nested
  - Typically an affine transformation
  - As just discussed

• **View transformation**
  - World space to eye space
  - Typically an affine transformation

• **Combination of both: Modelview transformation**
  - Used by *traditional* OpenGL (although world space is conceptually intuitive, it was not explicitly exposed in OpenGL)
Coordinate Transformations

- **Projective transformation**
  - Eye space to normalized device space
  - Parallel or perspective projection (defined by view frustum)
  - 3D to 2D: With preservation of depth (2.5 D)

- **Viewport transformation**
  - Normalized device space to window (raster) coordinates
Camera Parameters: Rend.Man

- **RenderMan camera specification**
  - Distance of Screen Window from origin given by “field of view” (fov)
    - fov: Full angle of segment (-1,0) to (1,0), when seen from origin
  - CW given implicitly
  - No offset on screen

  - Note: Left-handed coordinate system!

  - All geometry is assumed to be in camera coordinates!
    - Or needs to be transformed into it
Simple Camera Parameters

- **Camera definition** (typically used in ray tracers)
  - \( o \in \mathbb{R}^3 \): center of projection, point of view (PRP)
  - \( CW \in \mathbb{R}^3 \): vector to center of window
    - “Focal length”: projection of vector to \( CW \) onto VPN
      - \( focal = |(CW - o) \cdot VPN| \)
  - \( x, y \in \mathbb{R}^3 \): span of half viewing window
    - \( VPN = (y \times x)/|(y \times x)| \)
    - \( VUP = -y \)
    - \( width = 2|x| \)
    - \( height = 2|y| \)
    - Aspect ratio: \( \text{camera}_{ratio} = |x|/|y| \)

PRP: Projection reference point
VPN: View plane normal
VUP: View up vector
CW: Center of window
Full Camera Transformation

• **Goal**
  – Compute the transformation between points in 3D and pixels on the screen
  – Required for rasterization algorithms (e.g., OpenGL)
    • They project all primitives from 3D to 2D
    • Rasterization happens in 2D (actually 2.5D, XY plus Z attribute)

• **Given**
  – Camera *pose* (pos. & orient.)
    • *Extrinsic* parameters
  – Camera *configuration*
    • *Intrinsic* parameters
  – Pixel raster description
    • Resolution and placement on screen

• **In the following: Stepwise Approach**
  – Express each transformation step in homogeneous coordinates
  – Multiply all 4x4 matrices to combine transformations
Camera Transformation

- **Need camera position and orientation in world space**
  - External (extrinsic) camera parameters
    - Center of projection: projection reference point (PRP)
    - Optical axis: view-plane normal (VPN)
    - View up vector (VUP)
      - Not necessarily orthogonal to VPN, but not co-linear

- **Needed Transformations**
  1) Translation of PRP to the origin (-PRP)
  2) Rotation such that viewing direction is along negative Z axis
  2a) Rotate such that VUP is pointing up on screen
Camera Transformation

• **Goal:** Camera: at origin, view along –Z, Y upwards
  – Assume right-handed coordinate system!
  – Translation of PRP to the origin
  – Rotation of VPN to Z-axis
  – Rotation of projection of VUP to Y-axis

• **Rotations**
  – Build orthonormal basis for the camera and form inverse
    • \( Z' = VPN, X' = \text{normalize}(VUP \times VPN), Y' = Z' \times X' \)

• **Viewing transformation** \( V \)
  – Translation \( T \) followed by rotation \( R \)

\[
V = RT = \begin{pmatrix}
X'_x & Y'_x & Z'_x & 0 \\
X'_y & Y'_y & Z'_y & 0 \\
X'_z & Y'_z & Z'_z & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}^T \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}
\]
Viewing Transformation

- **Define projection (perspective or orthographic)**
  - Needs internal (intrinsic) camera parameters
  - Screen window (Center Window (CW), width, height)
    - Window size/position on image plane (relative to VPN intersection)
    - Window center relative to PRP determines viewing direction (≠ VPN)
  - Focal length (f)
    - Distance of projection plane from camera along VPN
    - Smaller focal length means larger field of view
  - Alternative: Field of view (fov) (defines width of view frustum)
    - Often used instead of screen window and focal length
      - Only valid when screen window is centered around VPN (often the case)
    - Vertical (or horizontal) angle plus aspect ratio (width/height)
      - Or two angles (both angles may be half or full angles, beware!)
  - Near and far clipping planes
    - Given as distances from the PRP along VPN
    - Near clipping plane avoids singularity at origin (division by zero)
    - Far clipping plane restricts the depth for fixed-point representation in HW
Shearing Transformation

- **Step 1: VPN may not go through center of window**
  - Possible oblique viewing configuration
- **Shear**
  - Shear space such that window center is along Z-axis
  - Window center CW (in 3D view coordinates)
    - RenderMan: \( CW = ((\text{right}+\text{left})/2, (\text{top}+\text{bottom})/2, -\text{focal})^T \)

- **Shear matrix**

\[
H = \begin{pmatrix}
1 & 0 & -\frac{CW_x}{CW_z} & 0 \\
0 & 1 & -\frac{CW_y}{CW_z} & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
Normalizing

- **Step 2: Scaling to canonical viewing frustum**
  - Goal: Scale in X and Y such that screen window boundaries open at 45-degree angles (at focal plane)
  - Scale in Z such that far clipping plane is at $Z = -1$

- **Scaling matrix**

\[
S = S_{far} S_{xy} = \begin{pmatrix}
\frac{1}{far} & 0 & 0 & 0 \\
0 & \frac{1}{far} & 0 & 0 \\
0 & 0 & \frac{1}{far} & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
\frac{2\text{focal width}}{} & 0 & 0 & 0 \\
0 & \frac{2\text{focal height}}{} & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
Perspective Transformation

• **Step 3: Perspective transformation**
  – From canonical perspective viewing frustum (= cone at origin around -Z-axis, 45° opening) to regular box [-1 .. 1]^2 x [0 .. 1]

• **Mapping of X and Y**
  – Lines through the origin are mapped to lines parallel to the Z-axis
    • \(x' = x/z\) and \(y' = y/z\) (coordinate given by slope with respect to -z!)
  – Do not change X and Y additively (first two rows stay the same)
  – Set \(W\) to \(-z\) so we divide by it when converting back to 3D
    • Determines last row

• **Perspective transformation**
  – \(P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ A & B & C & D \end{pmatrix}\)
    
    Still unknown

  – **Note:** Perspective projection = perspective transformation + parallel projection
Perspective Transformation

- **Computation of the coefficients A, B, C, D**
  - No shear of Z with respect to X and Y
    - \( A = B = 0 \)
  - Mapping of two known points
    - Computation of the two remaining parameters C and D
      - \( n = \text{near} / \text{far} \) (due to previous scaling by \( 1/\text{far} \))
    - Following mapping must hold
      - \((0,0,-1,1)^T = P(0,0,-1,1)^T\) and \((0,0,0,1)^T = P(0,0,-n,1)^T\)

- **Resulting Projective transformation**
  - \( P = \begin{pmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 1/n & n \\
  0 & 0 & 1-n & 1-n
  \end{pmatrix} \)
  - Transforms Z non-linearly (in 3D)
  - \( z' = -\frac{z+n}{z(1-n)} \)
Parallel Projection to 2D

• **Parallel projection** $P_{\text{parallel}}$ to $[-1 .. 1]^2$
  – Formally scaling in Z with factor 0
  – Typically still maintains Z in $[0,1]$ for depth buffering
    • As a vertex attribute (see OpenGL later)

• **Normalizing Transform** $N$
  – From $[-1 .. 1]^2$ to NDC ($[0 .. 1]^2$)
  – Scaling (by 1/2 in X and Y) and translation (by (1/2,1/2))

\[
P_{\text{parallel}} = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 \text{ or } 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \quad N = \begin{pmatrix}
\frac{1}{2} & 0 & 0 & \frac{1}{2} \\
0 & \frac{1}{2} & 0 & \frac{1}{2} \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
Viewport Transformation

- **Normalized Device Coordinates (NDC)**
  - Intrinsic camera parameters transform to NDC
    - \([0,1]^2\) for \(x, y\) across the screen window
    - \([0,1]\) for \(z\) (depth)

- **Mapping NDC to raster coordinates on the screen**
  - \(x_{res}, y_{res}\): Size of window in pixels
    - Should have same aspect ratios to avoid distortion
      - \(\text{camera}_{ratio} = \frac{x_{res} \, \text{pixelspacing}_x}{y_{res} \, \text{pixelspacing}_y}\)
    - Horizontal and vertical pixel spacing (distance between pixel centers)
      - Today, typically the same but can be different e.g. for some video formats
  - Position of window on the screen
    - Offset of window from origin of screen
      - \(posx\) and \(posy\) given in pixels
    - Depends on where the origin is on the screen (top left, bottom left)
  - “Scissor box” or “crop window” (region of interest)
    - No change in mapping but limits which pixels are rendered
Viewport Transformation

• **Scaling and translation in 2D**
  – Scaling matrix to map to entire window on screen
    • \( S_{\text{raster}}(x_{\text{res}}, y_{\text{res}}) \)
    • No distortion if aspect ratios have been handled correctly earlier
      – I.e. aspect ratio of window in world space == aspect ratio of raster window
    • In some cases, one needs to reverse direction of y
      – Some formats have screen origin at bottom left, some at top left
      – Needs additional translation/scaling
  – Positioning on the screen
    • Translation \( T_{\text{raster}}(x_{\text{pos}}, y_{\text{pos}}) \)
    • May be different depending on raster coordinate system
      – Origin at upper left or lower left
Orthographic Projection

- **Step 2a: Translation (orthographic)**
  - Bring near clipping plane into the origin

- **Step 2b: Scaling to regular box** \([-1 \ldots 1]^2 \times [0 \ldots -1]\)

- **Mapping of X and Y**

  \[
  P_o = S_{xyz}T_{near} =
  \begin{pmatrix}
    \frac{2}{\text{width}} & 0 & 0 & 0 \\
    0 & \frac{2}{\text{height}} & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & 0 & 1
  \end{pmatrix}
  \begin{pmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & \text{near} \\
    0 & 0 & 0 & 1
  \end{pmatrix}
  \]
Full Camera Transformation

- **Complete transformation (combination of matrices)**
  - Perspective Projection
    - \( T_{\text{camera}} = T_{raster} S_{raster} N P_{\text{parallel}} P_{\text{persp}} S_{\text{far}} S_{xy} H R T \)
  - Orthographic Projection
    - \( T_{\text{camera}} = T_{raster} S_{raster} N P_{\text{parallel}} S_{xyz} T_{\text{near}} \cdot H R T \)

- **Other representations**
  - Other literature uses different conventions
    - Different camera parameters as input
    - Different canonical viewing frustum
    - Different normalized coordinates
      - \([-1 .. 1]^3\) versus \([0 .. 1]^3\) versus ...
  - ...

→ *Results in different transformation matrices – so be careful !!!*
Per-Vertex Transformations

- **Traditional OpenGL pipeline**
  - Hierarchical modeling
    - Modelview matrix stack
    - Projection matrix stack
  - Each stack can be independently pushed/popped
  - Matrices can be applied/multiplied to top stack element

- **Today**
  - Arbitrary matrices as attributes to vertex shaders that apply them as they wish (later)
  - All matrix stack handling must now be done by application
OpenGL

• **Modern OpenGL**
  – Transformation provided by app, applied by vertex shader
  – Vertex or Geometry shader must output clip space vertices
    • Clip space: Just before perspective divide (by w)

• **Viewport transformation**
  – `glViewport(x, y, width, height)`
  – Now can even have multiple viewports
    • `glViewportIndexed(idx, x, y, width, height)`
  – Controlling the depth range (after Perspective transformation)
    • `glDepthRangeIndexed(idx, near, far)`
Discussion

• **Pinhole camera model**
  – Linear in homogeneous coordinates

• **A lot of things that we ignored**
  – Complex lenses distortion, aberrations
  – Flare
  – Depth-of-field
  – Vignetting