CSE 167x Computer Graphics
Ramamoorthi UC San DiegoX Unit 1

1 2D Linear Transforms

- Scale:
$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}$$
- Shear (in the x-direction; turns rectangles into parallelograms):
$$\begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + ay \\ y \end{bmatrix}$$
a is the extent of the shear. If the y-coordinate is 1, the x-coordinate moves to the right by a value of a. If the y-coordinate is -1, the x-coordinate moves to to the left by a value of a.
- Rotation:
$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$
Note that $R(X+Y) = R(X) + R(Y)$. You can translate and then rotate, or you can rotate and then translate (but in the latter case, you must be sure to rotate the translation as well).

2 3D Rotations

- The rows of a 3D rotation matrix can be interpreted as the three orthonormal axes of the destination coordinate frame, i.e. the matrix can be seen as
$$\begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}$$
where $\mathbf{u} = x_u\mathbf{x} + y_u\mathbf{y} + z_u\mathbf{z}$ and so on. Rotation matrices are orthogonal! $\mathbf{R}^T = \mathbf{R}^{-1}$.
- Accordingly, if we have the three orthonormal vectors of the target coordinate frame, we can immediately construct the corresponding rotation matrix.
- If we apply the rotation matrix to a *point* $\mathbf{p}$, we obtain
$$\begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} = \begin{bmatrix} \mathbf{u}\cdot\mathbf{p} \\ \mathbf{v}\cdot\mathbf{p} \\ \mathbf{w}\cdot\mathbf{p} \end{bmatrix}$$
or the projections of $\mathbf{p}$ onto each of the axes in the new coordinate frame ("where is the point in the new coordinate frame?").
- The rotation matrix for the $\mathbf{u}, \mathbf{v}, \mathbf{w}$ coordinate frame transforms $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ into the standard Cartesian axes ($\mathbf{u}$ becomes $\mathbf{x}$, $\mathbf{v}$ becomes $\mathbf{y}$, and $\mathbf{w}$ becomes $\mathbf{z}$). In other words, things will behave as if $\mathbf{u}, \mathbf{v}$, and $\mathbf{w}$ are now the standard basis [e.g. the rotation matrix applied to $\mathbf{u}+\mathbf{v}+\mathbf{w}$ gives $(1,1,1)$].
- The inverse (here the transpose) naturally does the opposite.

- How can we rotate a vector $\mathbf{b}$ by an angle θ about an arbitrary axis $\mathbf{a}$ (axis-angle rotation)?
  1. $\mathbf{b}$ can be written as the sum of (1) $\mathbf{b}_\perp$, a component orthogonal to $\mathbf{a}$, and (2) $\mathbf{b}_\parallel$, a component in the direction of $\mathbf{a}$. Rotating around an axis does not change the parallel component (to realize this, think about rotating a vector around the up vector $\mathbf{z}$).
     - $\mathbf{b}_\parallel = (\mathbf{a} \cdot \mathbf{b})\ \mathbf{a}$ (recall that $\mathbf{a}$ is a unit axis)
     - $\mathbf{b}_\perp = \mathbf{b} - \mathbf{b}_\parallel$
  2. Currently, $\mathbf{a}$ and $\mathbf{b}$ can be illustrated in one plane. But when $\mathbf{b}$ rotates, it will go out of the plane and have a component along the third axis ($\mathbf{c}$) in the trifecta of $(\mathbf{b}_\perp, \mathbf{c}, \mathbf{a})$. We need to define $\mathbf{c}$, which we can do using cross products.
     - $\mathbf{c} = \mathbf{a} \times \mathbf{b}$ (note: the length of $\mathbf{c}$ will be $\|\mathbf{b}\| \sin\phi$, the length of $\mathbf{b}_\perp$)
  3. By the first observation, we're really just rotating $\mathbf{b}_\perp$ by θ in the plane orthogonal to $\mathbf{a}$ (because the parallel component of $\mathbf{b}$ does not change) and then determining the projections of the rotated $\mathbf{b}_\perp$ onto $\mathbf{b}_\perp$ and $\mathbf{c}$. *Note: neither* $\mathbf{b}_\perp$ *nor* $\mathbf{c}$ *are necessarily unit vectors right now.*
     - Projection onto $\mathbf{b}_\perp$: $\|\mathbf{b}_\perp\| \cos\theta\ (\mathbf{b}_\perp / \|\mathbf{b}_\perp\|) = \mathbf{b}_\perp \cos\theta$
     - Projection onto $\mathbf{c}$: $\|\mathbf{b}_\perp\| \sin\theta\ (\mathbf{c}/\|\mathbf{c}\|) = \|\mathbf{b}_\perp\| \sin\theta\ (\mathbf{c}/\|\mathbf{b}_\perp\|) = \mathbf{c} \sin\theta$
- Overall, an axis-angle rotation about $\mathbf{a}$ by θ comes together as

$$R(\mathbf{a}, \theta) = \mathbf{I}_{3\times 3} \cos\theta + \mathbf{a}\mathbf{a}^T (1 - \cos\theta) + \mathbf{A}^* \sin\theta$$

  To arrive at this terrifying transformation, we define everything from before in terms of matrices:
  - The parallel component $(\mathbf{a} \cdot \mathbf{b})\ \mathbf{a}$ becomes $(\mathbf{a}^T\mathbf{b})\ \mathbf{a} = \mathbf{a}(\mathbf{a}^T\mathbf{b}) = \mathbf{a}\mathbf{a}^T\mathbf{b}$.
  - The projection onto $\mathbf{c}$ becomes $(\mathbf{a} \times \mathbf{b}) \sin\theta = \mathbf{A}^*\mathbf{b} \sin\theta = (\mathbf{A}^* \sin\theta)\mathbf{b}$.
    - $\mathbf{A}^*$ is the dual matrix form of $\mathbf{A}$ that we saw at the end of the previous unit:

$$\mathbf{A}^* = \begin{bmatrix} 0 & -z_a & y_a \\ z_a & 0 & -x_a \\ -y_a & x_a & 0 \end{bmatrix}$$

  - The projection onto $\mathbf{b}_\perp$ becomes $(\mathbf{b} - \mathbf{b}_\parallel) \cos\theta = (\mathbf{I}_{3\times 3}\mathbf{b} - \mathbf{a}\mathbf{a}^T\mathbf{b}) \cos\theta = (\mathbf{I}_{3\times 3} \cos\theta - \mathbf{a}\mathbf{a}^T \cos\theta)\mathbf{b}$.

  In total the transformation of $\mathbf{b}$ is the sum of these three orthogonal vectors, i.e.

$$\begin{aligned} &\mathbf{a}\mathbf{a}^T\mathbf{b} + (\mathbf{A}^* \sin\theta)\mathbf{b} + (\mathbf{I}_{3\times 3} \cos\theta - \mathbf{a}\mathbf{a}^T \cos\theta)\mathbf{b} \\ &= (\mathbf{a}\mathbf{a}^T + \mathbf{A}^* \sin\theta + \mathbf{I}_{3\times 3} \cos\theta - \mathbf{a}\mathbf{a}^T \cos\theta)\mathbf{b} \\ &= [\mathbf{I}_{3\times 3} \cos\theta + \mathbf{a}\mathbf{a}^T (1 - \cos\theta) + \mathbf{A}^* \sin\theta]\mathbf{b} \end{aligned}$$

  as written above.
- This is Rodrigues' rotation formula. It can be written explicitly (by just expanding the matrices) as

$$\cos\theta \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + (1 - \cos\theta) \begin{bmatrix} x_a^2 & x_a y_a & x_a z_a \\ x_a y_a & y_a^2 & y_a z_a \\ x_a z_a & y_a z_a & z_a^2 \end{bmatrix} + \sin\theta \begin{bmatrix} 0 & -z_a & y_a \\ z_a & 0 & -x_a \\ -y_a & x_a & 0 \end{bmatrix}$$

3 Homogeneous Coordinates

- Homogeneous coordinates allow us to do translation, as

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

  To turn homogeneous coordinates into Euclidean coordinates, we divide x, y, and z by the fourth, homogeneous coordinate w. If $w = 0$, then the point is at infinity. We also set $w = 0$ to represent vectors, because vectors shouldn't be translatable (they're just directions, and have no absolute position).

- We can represent all transformations in the graphics pipeline as homogeneous 4×4 matrices.
- *Translate, rotate* isn't the same as *rotate, translate*. Order matters. (Note: rotation and translation have received much attention because you can translate and rotate rigid bodies, but not scale them.)
  - Rotation first, then translation: $\mathbf{Rp} + \mathbf{t}$
  - Translation first, then rotation: $\mathbf{R}(\mathbf{p} + \mathbf{t}) = \mathbf{Rp} + \mathbf{Rt}$

4 Transforming Normals

- Surface normals do not transform in the same way as points on the surface (e.g. a surface normal on top of a rectangle sheared in the x-direction will continue to point upward, instead of shearing as well).
- In determining normal transformations, we should consider a point on the surface and a small plane surrounding it. The vector coming out of the point orthogonally to the surface is the *normal* $\mathbf{n}$, while vectors coming out of the point and remaining in the plane are the *tangents* $\mathbf{t}$.
  - Since tangents are actual geometric locations on the surface, they transform in the same way as points on the surface. Let's say the transformation for points on the surface is $\mathbf{M}$. Then $\mathbf{t} \to \mathbf{Mt}$.
  - The normal transforms according to a different matrix, which we'll call $\mathbf{Q}$: $\mathbf{n} \to \mathbf{Qn}$.
- We want to solve for $\mathbf{Q}$. Noting that $\mathbf{n}^T\mathbf{t} = 0$ (always, since $\mathbf{n}$ and $\mathbf{t}$ are orthogonal),

$$(\mathbf{Qn})^T(\mathbf{Mt}) = 0$$
$$\mathbf{n}^T\mathbf{Q}^T\mathbf{Mt} = 0$$

  we have the implication that $\mathbf{Q}^T\mathbf{M} = \mathbf{I}$ (because $\mathbf{n}^T\mathbf{t} = 0$ as well). Thus

$$\mathbf{Q} = (\mathbf{M}^{-1})^T$$

  This inverse transposition is *only* applied to the top-left 3×3 part of the 4×4 homogeneous matrix; it doesn't apply to the homogeneous part (e.g. translation doesn't matter since the normal is a vector).

  The formula we have derived must be applied to all surface normals.

5 Transforming Coordinate Frames

- An equivalent way of thinking about transformations: changing the coordinate frame. We can either think of the point transforming, or the coordinate system transforming.
- A coordinate frame has an origin and an orientation.
- Recall: the rows of a rotation matrix can be interpreted as the axes of the rotated coordinate frame.
- The *columns* of a rotation matrix can be seen as the transformed axes of the original coordinate frame.

6 gluLookAt

- We want to derive a 4×4 transformation matrix that can be used to position a camera in the world s.t. the eye is "looking" at an object.
- `gluLookAt(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)` positions the camera to be at the eye, looking at the center, with the specified up direction.
- Derivation:
  1. *Create a coordinate frame for the camera.*
     We can associate $\mathbf{w}$ with the negative viewing direction (i.e. eye minus center; this is the $+z$-axis in the desired case where we want the camera to look down the *negative* z-direction), $\mathbf{v}$ with the up vector, and $\mathbf{u}$ with the vector orthogonal to the first two. Since vectors in their original forms might not be orthogonal nor unit norm, we will have to ensure this in our calculations:
     - $\mathbf{w} = \mathbf{a}/\|\mathbf{a}\|$ where $\mathbf{a}$ = eye minus center
     - $\mathbf{u} = (\mathbf{b} \times \mathbf{w})/\|\mathbf{b} \times \mathbf{w}\|$, where $\mathbf{b}$ is the up vector
       - This automatically removes the non-orthogonal component from the up vector.
     - $\mathbf{v} = \mathbf{w} \times \mathbf{u}$
  2. *Define a rotation matrix corresponding to the camera coordinate frame.*
     Recall that the rows of the rotation matrix are the three unit vectors of the new coordinate frame.
     $$\mathbf{R} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}$$
  3. *Apply the appropriate translation for the camera (eye) location.*
     We need to be sure to do the translation first; *we cannot apply the translation after the rotation.* This is because the camera must be translated to the origin before the rotation is applied.

     The translation vector is the negative eye coordinates, since we're translating the eye to the origin.
- The final form of the `gluLookAt` matrix is
$$\begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

7 Orthographic Projection

- So far, our transformations have started and ended in the 3D world. However, the product of graphics is often a 2D image. So we need to be able to project our 3D points onto a 2D image.
- The most trivial projection method: orthographic projection, in which we simply drop the z-coordinate.
  - Orthographic projection preserves parallel lines (e.g. railroad tracks would not converge).
  - This is useful for technical illustrations, e.g. to show the view of an object as a sketch would.
- Setup: want to transform an arbitrary cuboid in the world to the $2 \times 2 \times 2$ cube centered at the origin.
  - The cuboid represents the region we're viewing.
- After mapping to the unit cube, we can drop the z-coordinate.
- The cuboid is parameterized by its min/max values in each of the three dimensions. To transform it, we'll translate it to the origin and then scale it appropriately.

- For viewing, OpenGL takes all of the points in a scene and then transforms them by some 4×4 matrix such as this one representing an orthographic projection.

8 Perspective Projection

- In perspective projection, further objects appear smaller and parallel lines converge. This is because we have a center of projection (the pinhole/eye location). This is the standard projection model.

$$x' = \frac{dx}{z}$$
$$y' = \frac{dy}{z}$$

Division by z ensures that further objects appear smaller. In order to *perform* the division by z, we make use of homogeneous coordinates. The transformation appears as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\frac{1}{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} -dx/z \\ -dy/z \\ -d \\ 1 \end{bmatrix}$$

Of course, this doesn't account for translation of the 2D coordinate frame or distortion or etc. (Also, why $-d$? This is because OpenGL convention has us looking down the negative z-axis, so the focal plane is at $-d$.)

9 gluPerspective

- `gluPerspective` sets up the perspective projection transformation of 3D points onto the screen.
- The transformation pipeline as seen so far: (1) a `gluLookAt` camera transformation, (2) projection.
- There's a viewing region involved with both `gluOrtho` and `gluPerspective`; in `gluOrtho` it's just a cuboid (a rectangular prism), whereas in `gluPerspective` it's a frustum.
  - The frustum consists of a near plane and a far plane, and edges connecting corresponding corners.
  - Any point that lies outside of the viewing frustum (e.g. closer than the near plane or beyond the far plane) is ignored. Note: the near and far planes are just for near/far culling.
  - The vertical field of view (`fovy`) controls the height of the image plane (and the focal length).
  - The aspect ratio $\frac{w}{h}$ controls the width and by extension the horizontal field of view.
- `gluPerspective` takes in `fovy`, `aspect`, `zNear` (> 0), and `zFar` (> 0) [latter two are distances].
- *Depth resolution*: the ability to distinguish between smaller depth differences.
  - Falls off quadratically from the near plane to the far plane in z. Set `zNear` as far as possible!
- The entire viewing pipeline:
  - model coordinates → (model transformation) → world coordinates
  - world coordinates → (camera transformation; `gluLookAt`) → eye coordinates
  - eye coordinates → (perspective transformation; `gluPerspective`) → screen coordinates
  - screen coordinates → (viewport transformation) → window coordinates
  - window coordinates → (raster transformation) → device coordinates
- Lighting, a 3D concept, is performed in eye coordinates (after eye coordinates, everything is in 2D).
- Screen coordinates are also known as normalized device coordinates (the $[-1, +1]$ cube).

10 Appendix: 4×4 Transformation Matrices

These matrices operate on (x, y, z, w) coordinates.

3D Scale

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \\ w \end{bmatrix}$$

3D Rotation

$$\begin{bmatrix} x_{\mathbf{u}} & y_{\mathbf{u}} & z_{\mathbf{u}} & 0 \\ x_{\mathbf{v}} & y_{\mathbf{v}} & z_{\mathbf{v}} & 0 \\ x_{\mathbf{w}} & y_{\mathbf{w}} & z_{\mathbf{w}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x_{\mathbf{u}} x + y_{\mathbf{u}} y + z_{\mathbf{u}} z \\ x_{\mathbf{v}} x + y_{\mathbf{v}} y + z_{\mathbf{v}} z \\ x_{\mathbf{w}} x + y_{\mathbf{w}} y + z_{\mathbf{w}} z \\ w \end{bmatrix}$$

3D Translation

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x + t_x w \\ y + t_y w \\ z + t_z w \\ w \end{bmatrix}$$

`gluPerspective`

$$\begin{bmatrix} \frac{f}{\texttt{aspect}} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{\texttt{zNear}+\texttt{zFar}}{\texttt{zNear}-\texttt{zFar}} & \frac{2*\texttt{zNear}*\texttt{zFar}}{\texttt{zNear}-\texttt{zFar}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} {}^{fx}/_{\texttt{aspect}} \\ fy \\ \frac{2*\texttt{zNear}*\texttt{zFar}}{\texttt{zNear}-\texttt{zFar}} w + \frac{\texttt{zNear}+\texttt{zFar}}{\texttt{zNear}-\texttt{zFar}} z \\ -z \end{bmatrix}$$

for $f = 1/\tan\left(\frac{\texttt{fovy}}{2}\right)$.

Notes:

- Transforms points in camera coordinates to clip coordinates.
- After this step, homogeneous clipping and perspective division are performed.
  - And thus $-$`zNear` (the near z-coordinate when looking down the negative z-axis) maps to -1.
  - And thus $-$`zFar` (the far z-coordinate when looking down the negative z-axis) maps to 1.