Spatial Transformations

Computer Graphics
CMU 15-462/15-662
Assignment 1 goes out today!

Assignment 1: Rasterizer

Modern GPUs implement an abstraction called the Rasterization Pipeline. This abstraction breaks the process of converting 3D triangles into 2D pixels into several highly optimized stages for a variety of efficient hardware implementations. In this assignment, you will be implementing parts of a simplified rasterization pipeline in software. Though simplified, your pipeline will be sufficient to allow Scotty3D to create preview renders without a GPU.

Different graphics APIs may present this pipeline in different ways, but the core steps remains consistent: a GPU draws things by running code (in parallel) on a list of vertices to produce homogeneous screen positions (+ extra varying data), building triangles from that list of vertices, clipping the triangles to remove parts not visible on the screen, performing a division to compute screen positions, computing a list of "fragments" covered by those triangles, running code on each fragment, and composing the results into a framebuffer.

Transforms
Lines
Flat triangles
Depth and blending

... Interpolation
Mip-mapping
Supersampling

... Extra credit!
But let’s back up a bit
The first part of this class relates to the graphics pipeline.

Specialized processors for executing graphics pipeline computations.

- Discrete GPU card
- Integrated GPU: part of modern CPU die
- Smartphone GPU (integrated)
Goal: render very high complexity 3D scenes

- 100’s of thousands to millions to billions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution screen outputs (~10Mpixel + supersampling)
- 30-120 fps
GPU: heterogeneous, multi-core processor

Modern GPUs offer ~35 TFLOPs of performance for generic vertex/fragment programs (“compute”) still enormous amount of fixed-function compute over here

CMU 15-462/662
## OpenGL/Direct3D graphics pipeline

Our rasterization pipeline doesn’t look much different from “real” pipelines used in modern APIs / graphics hardware.

<table>
<thead>
<tr>
<th>Operations on vertices</th>
<th>Operations on primitives (triangles, lines, etc.)</th>
<th>Operations on fragments</th>
<th>Operations on screen samples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vertex Processing</strong></td>
<td><strong>Primitive Processing</strong></td>
<td><strong>Fragment Generation</strong></td>
<td><strong>Screen sample operations</strong></td>
</tr>
<tr>
<td>Vertex stream</td>
<td>Primitive stream</td>
<td>(Rasterization)</td>
<td>(depth and color)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fragment stream</td>
<td>Shaded fragment stream</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Screen sample operations</td>
</tr>
</tbody>
</table>

1. Input: vertices in 3D space
2. Vertices in positioned in 3D normalized coordinate space
3. Triangles projected to 2D screen
4. Fragments (one fragment per covered sample)

Shaded fragments

Output: image (pixels)

* Several stages of the modern OpenGL pipeline are omitted
**Rasterization Pipeline**

- Modern real time image generation based on rasterization
  - **INPUT:** 3D “primitives”—essentially all triangles!
  - possibly with additional attributes (e.g., color)
  - **OUTPUT:** bitmap image (possibly w/ depth, alpha, …)

- **Our goal:** understand the stages in between*

*In practice, usually executed by graphics processing unit (GPU)*
The Rasterization Pipeline

Rough sketch of rasterization pipeline:

- Reflects standard “real world” pipeline (OpenGL/Direct3D) — the rest is just details (e.g., API calls)
The Rasterization Pipeline

Rough sketch of rasterization pipeline:

- Reflects standard “real world” pipeline (OpenGL/Direct3D)
  - the rest is just details (e.g., API calls)
Transforms
Lines
Flat triangles
Depth and blending
... Interpolation
Mip-mapping
Supersampling
... Extra credit!
On to Spatial Transformations!
Spatial Transformation

- Basically any function that assigns each point a new location
- Today we’ll focus on common transformations of space (rotation, scaling, etc.) encoded by linear maps

\[ f : \mathbb{R}^n \rightarrow \mathbb{R}^n \]
Transformations in Computer Graphics

- Where are linear transformations used in computer graphics?
- All over the place!
  - Position/deform objects in space
  - Move the camera
  - Animate objects over time
  - Project 3D objects onto 2D images
  - Map 2D textures onto 3D objects
  - Project shadows of 3D objects onto other 3D objects
  - …
**Review: Linear Maps**

Q: What does it mean for a map $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ to be linear?

**Geometrically:** it maps lines to lines, and preserves the origin

**Algebraically:** preserves vector space operations (addition & scaling)

\[ f(x, y) = f(x) + f(y) \]
Why do we care about linear transformations?

- Cheap to apply
- Usually pretty easy to solve for (linear systems)
- Composition of linear transformations is linear
  - product of many matrices is a single matrix
  - gives uniform representation of transformations
  - simplifies graphics algorithms, systems (e.g., GPUs & APIs)

\[
\begin{bmatrix}
\cos \theta & \sin \theta & 0 \\
-\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
a & 0 & 0 \\
0 & b & 0 \\
0 & 0 & c
\end{bmatrix}
\begin{bmatrix}
\cos \theta & 0 & \sin \theta \\
0 & 1 & 0 \\
-\sin \theta & 0 & \cos \theta
\end{bmatrix}
\cdots
\begin{bmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{bmatrix}
\]

*rotation*  *scale*  *rotation*  *composite transformation*
What kinds of linear transformations can we compose?
Types of Transformations

What would you call each of these types of transformations?

- translation
- scaling
- rotation
- shear

Q: How did you know that? (Hint: you did not inspect a formula!)
# Invariants of Transformation

A transformation is determined by the **invariants** it preserves

<table>
<thead>
<tr>
<th>transformation</th>
<th>invariants</th>
<th>algebraic description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>straight lines / origin</td>
<td>( f(ax+y) = af(x) + f(y), f(0) = 0 )</td>
</tr>
<tr>
<td>translation</td>
<td>differences between pairs of points</td>
<td>( f(x-y) = x-y )</td>
</tr>
<tr>
<td>scaling</td>
<td>lines through the origin / direction of vectors</td>
<td>( f(x)/</td>
</tr>
<tr>
<td>rotation</td>
<td>origin / distances between points / orientation</td>
<td>(</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

(Essentially how your brain “knows” what kind of transformation you’re looking at…)

Rotation

Rotations defined by three basic properties:

- keeps origin fixed
- preserves distances
- preserves orientation

First two properties together imply that rotations are **linear**.

**Will have a lot more to say about rotations in a later lecture...**
2D Rotations—Matrix Representation

Rotations preserve distances and the origin—hence, a 2D rotation by an angle $\theta$ maps each point $\mathbf{x}$ to a point $f_\theta(\mathbf{x})$ on the circle of radius $|\mathbf{x}|$:

$$f_\theta(\mathbf{x})$$

Where does $\mathbf{x} = (1,0)$ go if we rotate by $\theta$ (counter-clockwise)?

How about $\mathbf{x} = (0,1)$?

What about a general vector $\mathbf{x} = (x_1, x_2)$?
2D Rotations—Matrix Representation

So, how do we represent the 2D rotation function $f_\theta(x)$ using a matrix?

$$f_\theta(x) = \begin{bmatrix} \cos \theta & -\sin(\theta) \\ \sin \theta & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$
3D Rotations

- Q: In 3D, how do we rotate around the $x_3$-axis?
- A: Just apply the same transformation of $x_1$, $x_2$; keep $x_3$ fixed

\[
\begin{align*}
\text{rotate around } x_1 & \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin(\theta) \\ 0 & \sin \theta & \cos(\theta) \end{bmatrix} \\
\text{rotate around } x_2 & \quad \begin{bmatrix} \cos \theta & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos(\theta) \end{bmatrix} \\
\text{rotate around } x_3 & \quad \begin{bmatrix} \cos \theta & -\sin(\theta) & 0 \\ \sin \theta & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}
\end{align*}
\]
Rotations—Transpose as Inverse

Rotation will map standard basis to orthonormal basis $e_1, e_2, e_3$:

$$\begin{bmatrix}
  e_1^T \\
  e_2^T \\
  e_3^T 
\end{bmatrix} \quad \begin{bmatrix}
  e_1 & e_2 & e_3 
\end{bmatrix}
= \begin{bmatrix}
  e_1^T e_1 & e_1^T e_2 & e_1^T e_3 \\
  e_2^T e_1 & e_2^T e_2 & e_2^T e_3 \\
  e_3^T e_1 & e_3^T e_2 & e_3^T e_3 
\end{bmatrix} \begin{bmatrix}
  1 & 0 & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1 
\end{bmatrix}
= \begin{bmatrix}
  1 & 0 & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1 
\end{bmatrix}$$

Hence, $R^T R = I$, or equivalently, $R^T = R^{-1}$. 
Reflections

- Q: Does every matrix $Q^T Q = I$ describe a rotation?
- Remember that rotations must preserve the origin, preserve distances, and preserve orientation.
- Consider for instance this matrix:

$$Q = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad Q^T Q = \begin{bmatrix} (-1)^2 & 0 \\ 0 & 1 \end{bmatrix} = I$$

Q: Does this matrix represent a rotation? (If not, which invariant does it fail to preserve?)

A: No! It represents a reflection across the y-axis (and hence fails to preserve orientation).
Orthogonal Transformations

- In general, transformations that preserve distances and the origin are called orthogonal transformations.

- Represented by matrices $Q^T Q = I$
  - Rotations additionally preserve orientation: $\det(Q) > 0$
  - Reflections reverse orientation: $\det(Q) < 0$
Scaling

- Each vector $\mathbf{u}$ gets mapped to a scalar multiple
  \[ f(\mathbf{u}) = a\mathbf{u}, \quad a \in \mathbb{R} \]
- Preserves the direction of all vectors*
  \[ \frac{\mathbf{u}}{|\mathbf{u}|} = \frac{a\mathbf{u}}{|a\mathbf{u}|} \]
- Q: Is scaling a linear transformation?  A: Yes!

*assuming $a \neq 0$, $\mathbf{u} \neq 0$
Scaling — Matrix Representation

Q: Suppose we want to scale a vector $\mathbf{u} = (u_1, u_2, u_3)$ by $a$. How would we represent this operation via a matrix?

A: Just build a diagonal matrix $D$, with $a$ along the diagonal:

$$
\begin{bmatrix}
  a & 0 & 0 \\
  0 & a & 0 \\
  0 & 0 & a
\end{bmatrix}
\begin{bmatrix}
  u_1 \\
  u_2 \\
  u_3
\end{bmatrix}
= 
\begin{bmatrix}
  au_1 \\
  au_2 \\
  au_3
\end{bmatrix}
$$

Q: What happens if $a$ is negative?
Negative Scaling

For $a = -1$, can think of scaling by $a$ as sequence of reflections.

E.g., in 2D:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Since each reflection reverses orientation, orientation is preserved.

What about 3D?

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Now we have three reflections, and so orientation is reversed!
Nonuniform Scaling (Axis-Aligned)

- We can also scale each axis by a different amount
  \[ f(u_1, u_2, u_3) = (au_1, bu_2, cu_3), \quad a, b, c \in \mathbb{R} \]

- Q: What's the matrix representation?

- A: Just put \( a, b, c \) on the diagonal:

\[
\begin{bmatrix}
  a & 0 & 0 \\
  0 & b & 0 \\
  0 & 0 & c \\
\end{bmatrix}
\begin{bmatrix}
  u_1 \\
  u_2 \\
  u_3 \\
\end{bmatrix}
= 
\begin{bmatrix}
  au_1 \\
  bu_2 \\
  cu_3 \\
\end{bmatrix}
\]

Ok, but what if we want to scale along some other axes?
Nonuniform Scaling

- **Idea.** We could:
  - rotate to the new axes \( (R) \)
  - apply a diagonal scaling \( (D) \)
  - rotate back* to the original axes \( (R^T) \)

- Notice that the overall transformation is represented by a **symmetric** matrix
  \[
  A := R^T DR
  \]

Q: Do all symmetric matrices represent nonuniform scaling (for some choice of axes)?

*Recall that for a rotation, the inverse equals the transpose: \( R^{-1} = R^T \)
Spectral Theorem

- A: Yes! **Spectral theorem** says a symmetric matrix \( A = A^T \) has
  - orthonormal eigenvectors \( e_1, \ldots, e_n \in \mathbb{R}^n \)
  - real eigenvalues \( \lambda_1, \ldots, \lambda_n \in \mathbb{R} \)

- Can also write this relationship as \( AR = RD \), where

\[
R = \begin{bmatrix} e_1 & \cdots & e_n \end{bmatrix} \quad D = \begin{bmatrix} \lambda_1 \\ \vdots \\ \lambda_n \end{bmatrix}
\]

- Equivalently, \( A = RDR^T \)

- **Hence, every symmetric matrix performs a non-uniform scaling along some set of orthogonal axes.**

- **If** \( A \) **is positive definite** \( (\lambda_i > 0) \), **this scaling is positive.**
Shear

- A shear displaces each point $x$ in a direction $u$ according to its distance along a fixed vector $v$:

$$f_{u,v}(x) = x + \langle v, x \rangle u$$

- **Q:** Is this transformation linear?
- **A:** Yes—for instance, can represent it via a matrix $A_{u,v} = I + uv^\top$

Example.

$u = (\cos(t), 0, 0)$
$v = (0, 1, 0)$

$$A_{u,v} = \begin{bmatrix} 1 & \cos(t) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
Composite Transformations

From these basic transformations (rotation, reflection, scaling, shear...) we can now build up composite transformations via matrix multiplication:

\[ A(t) = R_x(t)R_y(t)S(t) \]
How do we decompose a linear transformation into pieces?
(rotations, reflections, scaling, …)
Decomposition of Linear Transformations

- In general, no unique way to write a given linear transformation as a composition of basic transformations!
- However, there are many useful decompositions:
  - singular value decomposition (good for signal processing)
  - LU factorization (good for solving linear systems)
  - polar decomposition (good for spatial transformations)
  - ...
- Consider for instance this linear transformation:

\[
A = \begin{bmatrix}
  .34 & -.11 & -.89 \\
  -.65 & .52 & -.70 \\
  .25 & .23 & -.69
\end{bmatrix}
\]
Polar & Singular Value Decomposition

For example, polar decomposition decomposes any matrix $A$ into orthogonal matrix $Q$ and symmetric positive-semidefinite matrix $P$:

$$ A = QP $$

Q: What do each of the parts mean geometrically?

- rotation/reflection
- nonnegative, nonuniform scaling

Since $P$ is symmetric, can take this further via the spectral decomposition $P = VDV^T$ (V orthogonal, $D$ diagonal):

$$ A = QVDV^T = UDV^T $$

Result $UDV^T$ is called the singular value decomposition
Interpolating Transformations

- How are these decompositions useful for graphics?
- Consider interpolating between two linear transformations $A_0, A_1$ of some initial model

Goal: animate transition with some nice continuous motion
Interpolating Transformations—Linear

One idea: just take a linear combination of the two matrices, weighted by the current time $t \in [0, 1]$

$$A(t) = (1 - t)A_0 + tA_1$$

Hits the right start/endpoints... but looks awful in between!
Interpolating Transformations—Polar

Better idea: separately interpolate components of polar decomposition.

\[ A_0 = Q_0 P_0, \quad A_1 = Q_1 P_1 \]

scaling

rotation

final interpolation

\[ P(t) = (1 - t)P_0 + tP_1 \]

\[ \widetilde{Q}(t) = (1 - t)Q_0 + tQ_1 \]

\[ \widetilde{Q}(t) = Q(t)X(t) \]

\[ A(t) = Q(t)P(t) \]

…looks better!

See: Shoemake & Duff, “Matrix Animation and Polar Decomposition”
Example: Linear Blend Skinning

- Naïve linear interpolation also causes artifacts when blending between transformations on a character (“candy wrapper effect”)
- Lots of research on alternative ways to blend transformations...

LBS: candy-wrapper artifact

Jacobson, Deng, Kavan, & Lewis (2014)  
“Skinning: Real-time Shape Deformation”

Rumman & Fratarcangeli (2015)  
“Position-based Skinning for Soft Articulated Characters”

Linear Blend Skinning  
our method  
Dual Quaternion Skinning
Translations

- So far we’ve ignored a basic transformation—translations
- A translation simply adds an offset $u$ to the given point $x$:

$$f_u(x) = x + u$$

Q: Is this transformation linear? (Certainly seems to move us along a line…)

Let’s carefully check the definition…

- **Additivity**
  $$f_u(x + y) = x + y + u$$
  $$f_u(x) + f_u(y) = x + y + 2u$$

- **Homogeneity**
  $$f_u(ax) = ax + u$$
  $$af_u(x) = ax + au$$

A: No! Translation is affine, not linear!
Composition of Transformations

- Recall we can compose linear transformations via matrix multiplication:
  \[
  A_3(A_2(A_1x))) = (A_3A_2A_1)x
  \]

- It’s easy enough to compose translations—just add vectors:
  \[
  f_{u_3}(f_{u_2}(f_{u_1}(x))) = f_{u_1+u_2+u_3}(x)
  \]

- What if we want to intermingle translations and linear transformations (rotation, scale, shear, etc.)?
  \[
  A_2(A_1x + b_1) + b_2 = (A_2A_1)x + (A_2b_1 + b_2)
  \]

- Now we have to keep track of a matrix and a vector

- Moreover, we’ll see (later) that this encoding won’t work for other important cases, such as perspective transformations

  But there is a better way…
Strange idea:  
Maybe translations turn into **linear** transformations if we go into the 4th dimension...!
Homogeneous Coordinates

- Came from efforts to study perspective
- Introduced by Möbius as a natural way of assigning coordinates to lines
- Show up naturally in a surprising large number of places in computer graphics:
  - 3D transformations
  - perspective projection
  - quadric error simplification
  - premultiplied alpha
  - shadow mapping
  - projective texture mapping
  - discrete conformal geometry
  - hyperbolic geometry
  - clipping
  - directional lights
  - ...

Probably worth understanding!
Homogeneous Coordinates—Basic Idea

- Consider any 2D plane that does not pass through the origin \( O \) in 3D.

- Every line through the origin in 3D corresponds to a point in the 2D plane.
  - Just find the point \( p \) where the line \( L \) pierces the plane.

Hence, any point \( \hat{p} \) on the line \( L \) can be used to represent the point \( p \).
Homogeneous Coordinates (2D)

- More explicitly, consider a point \( p = (x, y) \), and the plane \( z = 1 \) in 3D.

- Any three numbers \( \hat{p} = (a, b, c) \) such that \( (a/c, b/c) = (x, y) \) are homogeneous coordinates for \( p \).
  - E.g., \((x, y, 1)\)
  - In general: \((cx, cy, c)\) for \( c \neq 0\)

- Hence, two points \( \hat{p}, \hat{q} \in \mathbb{R}^3 \setminus \{O\} \) describe the same point in 2D (and line in 3D) if \( \hat{p} = \lambda \hat{q} \) for some \( \lambda \neq 0 \).

Great... but how does this help us with transformations?
Translation in Homogeneous Coordinates

Let’s think about what happens to our homogeneous coordinates $\hat{p}$ if we apply a translation to our 2D coordinates $p$

Q: What kind of transformation does this look like?
Translation in Homogeneous Coordinates

- But wait a minute—shear is a linear transformation!
- Can this be right? Let’s check in coordinates…

Suppose we translate a point $p = (p_1, p_2)$ by a vector $u = (u_1, u_2)$ to get $p' = (p_1 + u_1, p_2 + u_2)$.

The homogeneous coordinates $\hat{p} = (cp_1, cp_2, c)$ then become $\hat{p}' = (cp_1 + cu_1, cp_2 + cu_2, c)$.

Notice that we’re shifting $\hat{p}$ by an amount $cu$ that’s proportional to the distance $c$ along the third axis—a shear.

**Using homogeneous coordinates, we can represent an affine transformation in 2D as a linear transformation in 3D.**
Homogeneous Translation—Matrix Representation

- To write as a matrix, recall that a shear in the direction \( \mathbf{u} = (u_1, u_2) \) according to the distance along a direction \( \mathbf{v} \) is

  \[
  f_{\mathbf{u},\mathbf{v}}(\mathbf{x}) = \mathbf{x} + \langle \mathbf{v}, \mathbf{x} \rangle \mathbf{u}
  \]

- In matrix form:

  \[
  f_{\mathbf{u},\mathbf{v}}(\mathbf{x}) = \left( I + \mathbf{uv}^\top \right) \mathbf{x}
  \]

- In our case, \( \mathbf{v} = (0,0,1) \) and so we get a matrix

\[
\begin{bmatrix}
1 & 0 & u_1 \\
0 & 1 & u_2 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
cp_1 \\
cp_2 \\
c
\end{bmatrix}
= \begin{bmatrix}
c(p_1 + u_1) \\
c(p_2 + u_2) \\
c
\end{bmatrix}
\xrightarrow{1/c}
\begin{bmatrix}
p_1 + u_1 \\
p_2 + u_2
\end{bmatrix}
\]
Other 2D Transformations in Homogeneous Coordinates

Original shape in 2D can be viewed as many copies, uniformly scaled by $x_3$

2D scale ↔ scale $x_1$ and $x_2$; preserve $x_3$
(Q: what happens to 2D shape if you scale $x_1$, $x_2$, and $x_3$ uniformly?)

2D rotation ↔ rotate around $x_3$

2D translate ↔ shear

Now easy to compose all these transformations
3D Transformations in Homogeneous Coordinates

- Not much changes in three (or more) dimensions: just append one “homogeneous coordinate” to the first three

- Matrix representations of 3D linear transformations just get an additional identity row/column; translation is again a shear

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Matrix Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotate $(x, y, z)$ around $y$ by $\theta$</td>
<td>$\begin{bmatrix} \cos \theta &amp; 0 &amp; \sin \theta &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 \ -\sin \theta &amp; 0 &amp; \cos \theta &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix}$</td>
</tr>
<tr>
<td>Shear $(x, y)$ by $z$ in $(s, t)$ direction</td>
<td>$\begin{bmatrix} 1 &amp; 0 &amp; s &amp; 0 \ 0 &amp; 1 &amp; t &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix}$</td>
</tr>
<tr>
<td>Scale $x, y, z$ by $a, b, c$</td>
<td>$\begin{bmatrix} a &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; b &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; c &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix}$</td>
</tr>
<tr>
<td>Translate $(x, y, z)$ by $(u, v, w)$</td>
<td>$\begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; u \ 0 &amp; 1 &amp; 0 &amp; v \ 0 &amp; 0 &amp; 1 &amp; w \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix}$</td>
</tr>
</tbody>
</table>
Points vs. Vectors

- Homogeneous coordinates have another useful feature: distinguish between points and vectors

- Consider for instance a triangle with:
  - vertices \( a, b, c \in \mathbb{R}^3 \)
  - normal vector \( n \in \mathbb{R}^3 \)

- Suppose we transform the triangle by appending “1” to \( a, b, c, n \) and multiplying by this matrix:

\[
\begin{bmatrix}
\cos \theta & 0 & \sin \theta & u \\
0 & 1 & 0 & v \\
-\sin \theta & 0 & \cos \theta & w \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Normal is not orthogonal to triangle! (What went wrong?)
Points vs. Vectors (continued)

Let’s think about what happens when we multiply the normal vector \( \mathbf{n} \) by our matrix:

\[
\begin{bmatrix}
\cos \theta & 0 & \sin \theta \\
0 & 1 & 0 \\
-\sin \theta & 0 & \cos \theta \\
0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
\mathbf{n} \\
\mathbf{u} \\
\mathbf{v} \\
\mathbf{w}
\end{bmatrix}
\]

*Recall that vectors just have direction and magnitude—they don’t have a “basepoint”!

But when we rotate/translate a triangle, its normal should just rotate!*

Solution? Just set homogeneous coordinate to zero!

Translation now gets ignored; normal is orthogonal to triangle

*Recall that vectors just have direction and magnitude—they don’t have a “basepoint”!
Points vs. Vectors in Homogeneous Coordinates

- In general:
  - A point has a **nonzero** homogeneous coordinate \( (c = 1) \)
  - A vector has a **zero** homogeneous coordinate \( (c = 0) \)

- But wait… what division by \( c \) mean when it’s equal to zero?

- Well consider what happens as \( c \rightarrow 0 \)...
Scene Graph

- For complex scenes (e.g., more than just a cube!) scene graph can help organize transformations

- Motivation: suppose we want to build a “cube creature” by transforming copies of the unit cube

- Difficult to specify each transformation directly

- Instead, build up transformations of “lower” parts from transformations of “upper” parts
  - E.g., first position the body
  - Then transform upper arm relative to the body
  - Then transform lower arm relative to upper arm
  - ...
Scene Graph (continued)

- Scene graph stores relative transformations in directed graph
- Each edge (+root) stores a linear transformation (e.g., a 4x4 matrix)
- Composition of transformations gets applied to nodes

E.g., \( A_1A_0 \) gets applied to left upper leg; \( A_2A_1A_0 \) to left lower leg

- Keep transformations on a stack to reduce redundant multiplication
Scene Graph—Example

Often used to build up complex “rig”:

In general, scene graph also includes other models, lights, cameras, ...
Instancing

- What if we want many copies of the same object in a scene?
- Rather than have many copies of the geometry, scene graph, etc., can just put a “pointer” node in our scene graph.
- Like any other node, can specify a different transformation on each incoming edge.

Deussen et al, “Realistic modeling and rendering of plant ecosystems”
Instancing—Example
Order matters when composing transformations!

scale by 1/2, then translate by (3,1)

translate by (3,1), then scale by 1/2
How would you perform these transformations?

Remember: always more than one way to do it!
Common task: rotate about a point \( x \)

Step 1: translate by \(-x\)

Step 2: rotate

Step 4: translate by \(x\)

Q: What happens if we just rotate without translating first?
Screen Transformation (OpenGL)

- One last transformation is needed in the rasterization pipeline: transform from viewing plane to pixel coordinates.

- E.g., suppose we want to draw all points that fall inside the square \([-1,1] \times [-1,1]\) on the \(z = 1\) plane, into a \(W \times H\) pixel image.

Q: What transformation(s) would you apply?
Screen Transformation (Vulkan, Direct3D)

- One last transformation is needed in the rasterization pipeline: transform from viewing plane to pixel coordinates

- E.g., suppose we want to draw all points that fall inside the square \([-1, 1] \times [-1, 1]\) on the \(z = 1\) plane, into a \(W \times H\) pixel image with upper-left origin.

Q: What transformation(s) would you apply? (Careful: \(y\) is now down!)
Spatial Transformations—Summary

transformation defined by its **invariants**

**basic linear transformations**
- scaling
- rotation
- reflection
- shear

**basic nonlinear transformations**
- translation
- perspective projection (next class!)

**composite transformations**
- compose basic transformations to get more interesting ones
- always reduces to a single 4x4 matrix (in homogeneous coordinates)
  - simple, unified representation, efficient implementation
- order of composition matters!
- many ways to decompose a given transformation (polar, SVD, …)
- use scene graph to organize transformations
- use instancing to eliminate redundancy

linear when represented via **homogeneous coords**

homogeneous coords also distinguish points & vectors
Drawing a Cube Creature

- Let’s put this all together: starting with our 3D cube, we want to make a 2D, perspective-correct image of a “cube creature”

- First we use our scene graph to apply 3D transformations to several copies of our cube

- Then we apply a 3D transformation to position our camera

- Then a perspective projection

- Finally we convert to image coordinates (and rasterize)

...Easy, right? :-(
Drawing a Cube Creature

- Let’s put this all together: starting with our 3D cube, we want to make a 2D, perspective-correct image of a “cube creature”

- First we use our scene graph to apply 3D transformations to several copies of our cube

- Then we apply a 3D transformation to position our camera

- Then a perspective projection

- Finally we convert to image coordinates (and rasterize)

- …Easy, right? :-)

Next class!
Next time!

- Perspective Projection and Rasterization