The Graphics Pipeline and OpenGL I: Transformations

Gordon Wetzstein
Stanford University
EE 267 Virtual Reality
Lecture 2
stanford.edu/class/ee267/
Logistics Update

• all homework submissions: https://gradescope.com/, code: MX6E5M

• office hours (instructor): Mondays, 1:50-2:50pm, Packard 236

• office hours (CAs): Tuesdays 3-4:30pm, Wednesdays 4:30-6pm, Thursdays 4:30-6pm, all in Packard 001

• change of lab times: Fridays at 9am, 12pm, or 2pm – please sign up again
• sorry, too many people couldn’t make the morning slots
Lecture Overview

• what is computer graphics?
• the graphics pipeline
• primitives: vertices, edges, triangles!
• model transforms: translations, rotations, scaling
• view transform
• perspective transform
What is Computer Graphics?

• at the most basic level: conversion from 3D scene description to 2D image

• what do you need to describe a static scene?
  • 3D geometry and transformations
  • lights
  • material properties

• most common geometry primitives in graphics:
  • vertices (3D points) and normals (unit-length vector associated with vertex)
  • triangles (set of 3 vertices, high-resolution 3D models have M or B of triangles)
The Graphics Pipeline

- geometry + transformations
- cameras and viewing
- lighting and shading
- rasterization
- texturing
Some History

- Stanford startup in 1981
- computer graphics goes hardware
- based on Jim Clark’s geometry engine
Some History

Figure 3: Geometry System: each block is a Geometry Engine.

The subsystems are:

- **Matrix Subsystem**: A stack of 4x4 floating point matrices for completely general 2D or 3D floating point coordinate transformation of graphical data.

- **Clipping Subsystem**: A windowing, or clipping, capability for clipping 2D or 3D graphical data to a window into the user's virtual drawing space. In 3D, this window is a volume of the user's virtual floating point space, corresponding to a truncated viewing pyramid with "near" and "far" clipping.

- **Scaling Subsystem**: Scaling of 2D and 3D coordinates to the coordinate system of the particular output device of the user. In 3D, this scaling phase also includes either orthographic or perspective projection onto the viewer's virtual window. Stereo coordinates are computed and optionally supplied as the output of the system.
The Graphics Pipeline

- monolithic graphics workstations of the 80s have been replaced by modular GPUs (graphics processing units); major companies: Nvidia, AMD, Intel

- early versions of these GPUs implemented fixed-function rendering pipeline in hardware

- GPUs have become programmable starting in the late 90s
  - e.g. in 2001 Nvidia GeForce 3 = first programmable shaders

- now: GPUs = programmable (e.g. OpenGL, CUDA, OpenCL) processors
The Graphics Pipeline

GPU = massively parallel processor
• OpenGL is our interface to the GPU!

• right: “old-school” OpenGL state machine

• today’s lecture: vertex transforms
WebGL

- JavaScript application programmer interface (API) for 2D and 3D graphics
- OpenGL ES 2.0 running in the browser, implemented by all modern browsers
- overview, tutorials, documentation:
three.js

- cross-browser JavaScript library/API

- higher-level library that provides a lot of useful helper functions, tools, and abstractions around WebGL – easy and convenient to use

- https://threejs.org/
- simple examples: https://threejs.org/examples/

- great introduction (in WebGL):
  http://davidscottlyons.com/threejs/presentations/frontporch14/
three.js

Scene Graph

Renderer

Scene graph objects

Camera

Mesh

Light

Geometry

Material

Texture

Image

Vertex

Face3

more in the lab on Friday ...

http://davidscottlyons.com/threejs/presentations/frontporch14/
The Graphics Pipeline

- frame buffer: dedicated memory for “screen pixels” + X
- oftentimes have multiple frame buffers (i.e. offscreen buffers)
- render into frame buffers, then send to monitor

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
The Graphics Pipeline

3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as \((x, y, z)\) position, color (RGB or RGBA), vertex-normal \((n_x, n_y, n_z)\), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
The Graphics Pipeline

2. Rasterization: Convert each primitive (connected vertices) into a set of fragments. A fragment can be treated as a pixel in 3D spaces, which is aligned with the pixel grid, with attributes such as position, color, normal and texture.
3. Fragment Processing: Process individual fragments.
4. Output Merging: Combine the fragments of all primitives (in 3D space) into 2D color-pixel for the display.

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
1. **Vertex Processing**: Process and transform individual vertices.
2. **Rasterization**: Convert each primitive (connected vertices) into a set of fragments. A fragment can be treated as a pixel in 3D spaces, which is aligned with the pixel grid, with attributes such as position, color, normal and texture.
3. **Fragment Processing**: Process individual fragments.
4. **Output Merging**: Combine the fragments of all primitives (in 3D space) into 2D color-pixel for the display.

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
The Graphics Pipeline

- **Vertex Processor (Programmable)**
  - transforms & (per-vertex) lighting

- **Rasterizer**

- **Fragment Processor (Programmable)**
  - texturing
  - (per-fragment) lighting

- **Output Merging**

- **Display**

- **Raw Vertices & Primitives**
- **Transformed Vertices & Primitives**
- **Fragments**
- **Processed Fragments**
- **2D array of color-values**

vertex shader

fragment shader

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Coordinate Systems

- right hand coordinate system

- a few different coordinate systems:
  - object coordinates
  - world coordinates
  - viewing coordinates
  - also clip, normalized device, and window coordinates
Primitives

- vertex = 3D point \( v(x,y,z) \)
- triangle = 3 vertices
- normal = 3D vector per vertex describing surface orientation \( n(n_x,n_y,n_z) \)
Pixels v Fragments

- fragments have rasterized 2D coordinates on screen but a lot of other attributes too (texture coordinates, depth value, alpha value, …)
- pixels appear on screen
- won’t discuss in more detail today

Vertex, Primitives, Fragment and Pixel

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Vertex Transforms

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Vertex Transforms

1. Arrange the objects (or models, or avatar) in the world (Model Transformation or World transformation).

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Vertex Transforms

1. Arrange the objects (or models, or avatar) in the world (Model Transformation or World transformation).
2. Position and orientation the camera (View transformation).
1. Arrange the objects (or models, or avatar) in the world (Model Transformation or World transformation).

2. Position and orientation the camera (View transformation).

3. Select a camera lens (wide angle, normal or telescopic), adjust the focus length and zoom factor to set the camera's field of view (Projection transformation).
Vertex Transforms

1. Arrange the objects (or models, or avatar) in the world (Model Transformation or World transformation).
2. Position and orientation the camera (View transformation).
3. Select a camera lens (wide angle, normal or telescopic), adjust the focus length and zoom factor to set the camera's field of view (Projection transformation).
4. Print the photo on a selected area of the paper (Viewport transformation) - in rasterization stage
Model Transform

- transform each vertex $v = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ from object coordinates to world coordinates

Objects are typically created in their **local spaces**. We need to **bring** them into the common **world space**, via a series of affine transforms (translation, rotation and scaling).

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Model Transform - Scaling

- transform each vertex \( v = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \) from object coordinates to world coordinates

1. scaling as 3x3 matrix

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

scaled vertex = matrix-vector product:

\[
Sv = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \\ s_z z \end{pmatrix}
\]
Model Transform - Rotation

- transform each vertex \( v = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \) from object coordinates to world coordinates

2. rotation as 3x3 matrix

\[
R_z(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{pmatrix} \quad R_y(\theta) = \begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix}
\]

rotated vertex = matrix-vector product, e.g.

\[
R_zv = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \\ z \end{pmatrix}
\]
Model Transform - Translation

- transform each vertex $v = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$ from object coordinates to world coordinates

3. translation cannot be represented as 3x3 matrix!

\[
\begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ z + d_z \end{pmatrix}
\]

that’s unfortunate 😞
Model Transform - Translation

- solution: use homogeneous coordinates, vertex is $v = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$

3. translation is 4x4 matrix

$$T(d) = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$Tv = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ z + d_z \\ 1 \end{pmatrix}$$

better 😊
Summary of Homogeneous Matrix Transforms

- **translation** \( T(d) = \begin{pmatrix}
1 & 0 & 0 & d_x \\
0 & 1 & 0 & d_y \\
0 & 0 & 1 & d_z \\
0 & 0 & 0 & 1
\end{pmatrix} \)

- **scale** \( S(s) = \begin{pmatrix}
s_x & 0 & 0 & 0 \\
0 & s_y & 0 & 0 \\
0 & 0 & s_z & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \)

- **rotation** \( R_z(\theta) = \begin{pmatrix}
\cos\theta & -\sin\theta & 0 & 0 \\
\sin\theta & \cos\theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix} \)

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

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

Read more: https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Summary of Homogeneous Matrix Transforms

- **translation** $T(d) = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$

- **inverse translation** $T^{-1}(d) = T(-d) = \begin{pmatrix} 1 & 0 & 0 & -d_x \\ 0 & 1 & 0 & -d_y \\ 0 & 0 & 1 & -d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$

- **scale** $S(s) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

- **inverse scale** $S^{-1}(s) = S\left(\frac{1}{s}\right) = \begin{pmatrix} 1/s_x & 0 & 0 & 0 \\ 0 & 1/s_y & 0 & 0 \\ 0 & 0 & 1/s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

- **rotation** $R_z(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

- **inverse rotation** $R_z^{-1}(\theta) = R_z(-\theta) = \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Read more: https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Summary of Homogeneous Matrix Transforms

- successive transforms: \[ v' = T \cdot S \cdot R_z \cdot R_x \cdot T \cdot v \]

- inverse successive transforms: \[
\begin{align*}
v &= \left( T \cdot S \cdot R_z \cdot R_x \cdot T \right)^{-1} \cdot v' \\
&= T^{-1} \cdot R_x^{-1} \cdot R_z^{-1} \cdot S^{-1} \cdot T^{-1} \cdot v'
\end{align*}
\]
Vector and Normal Transforms

- homogeneous representation of a vector $t$, i.e. pointing from $v_1$ to $v_2$:

$$t = \begin{pmatrix}
(v_2 - v_1)_x \\
(v_2 - v_1)_y \\
(v_2 - v_1)_z \\
(1-1)
\end{pmatrix} = \begin{pmatrix}
t_x \\
t_y \\
t_z \\
0
\end{pmatrix}$$

- successive transforms: $t' = M \cdot t = M \cdot (v_2 - v_1) = M \cdot v_2 - M \cdot v_1$

- this works!
Vector and **Normal** Transforms

- homogeneous representation of a normal (unit length, perpendicular to surface)

- successive transforms $n' = M \cdot n$

- this does **NOT** work! (non-uniform scaling is a problem)
Vector and **Normal** Transforms

- homogeneous representation of a normal (unit length, perpendicular to surface)
- need to use transpose of inverse for transformation!

\[
n = \begin{pmatrix} n_x \\ n_y \\ n_z \\ 0 \end{pmatrix}
\]

\[
n' = \left(M^{-1}\right)^T \cdot n
\]
Attention!

- rotations and translations (or transforms in general) are not commutative!
- make sure you get the correct order!
so far we discussed model transforms, e.g. going from object or model space to world space
View Transform

- so far we discussed model transforms, e.g. going from object or model space to world space
- one simple 4x4 transform matrix is sufficient to go from world space to camera or view space!
View Transform

specify camera by

- **eye position** \( \text{eye} = \begin{pmatrix} \text{eye}_x \\ \text{eye}_y \\ \text{eye}_z \end{pmatrix} \)

- **reference position** \( \text{center} = \begin{pmatrix} \text{center}_x \\ \text{center}_y \\ \text{center}_z \end{pmatrix} \)

- **up vector** \( \text{up} = \begin{pmatrix} \text{up}_x \\ \text{up}_y \\ \text{up}_z \end{pmatrix} \)
View Transform

specify camera by

- eye position: \( \text{eye} = \begin{pmatrix} \text{eye}_x \\ \text{eye}_y \\ \text{eye}_z \end{pmatrix} \)

- reference position: \( \text{center} = \begin{pmatrix} \text{center}_x \\ \text{center}_y \\ \text{center}_z \end{pmatrix} \)

- up vector: \( \text{up} = \begin{pmatrix} \text{up}_x \\ \text{up}_y \\ \text{up}_z \end{pmatrix} \)

compute 3 vectors:

\[
\begin{align*}
\text{z}^c &= \frac{\text{eye} - \text{center}}{||\text{eye} - \text{center}||} \\
\text{x}^c &= \frac{\text{up} \times \text{z}^c}{||\text{up} \times \text{z}^c||} \\
\text{y}^c &= \text{z}^c \times \text{x}^c
\end{align*}
\]
view transform $M$ is translation into eye position, followed by rotation

compute 3 vectors:

\[
\begin{align*}
z^c &= \frac{\text{eye} - \text{center}}{||\text{eye} - \text{center}||} \\
x^c &= \frac{\text{up} \times z^c}{||\text{up} \times z^c||} \\
y^c &= z^c \times x^c
\end{align*}
\]
View Transform

view transform $M$ is translation into eye position, followed by rotation

compute 3 vectors:

$$z^c = \frac{\text{eye} - \text{center}}{\|\text{eye} - \text{center}\|}$$

$$x^c = \frac{\text{up} \times z^c}{\|\text{up} \times z^c\|}$$

$$y^c = z^c \times x^c$$

$$M = R \cdot T(-e) = \begin{pmatrix}
    x^c_x & x^c_y & x^c_z & 0 \\
    y^c_x & y^c_y & y^c_z & 0 \\
    z^c_x & z^c_y & z^c_z & 0 \\
    0 & 0 & 0 & 1
\end{pmatrix} \begin{pmatrix}
    1 & 0 & 0 & -\text{eye}_x \\
    0 & 1 & 0 & -\text{eye}_y \\
    0 & 0 & 1 & -\text{eye}_z \\
    0 & 0 & 0 & 1
\end{pmatrix}$$
view transform $M$ is translation into eye position, followed by rotation

$$M = R \cdot T(-e) = \begin{pmatrix}
    x^c_x & x^c_y & x^c_z & -(x^c_{\text{eye}} x + x^c_{\text{eye}} y + x^c_{\text{eye}} z) \\
    y^c_x & y^c_y & y^c_z & -(y^c_{\text{eye}} x + y^c_{\text{eye}} y + y^c_{\text{eye}} z) \\
    z^c_x & z^c_y & z^c_z & -(z^c_{\text{eye}} x + z^c_{\text{eye}} y + z^c_{\text{eye}} z) \\
    0 & 0 & 0 & 1
\end{pmatrix}$$
View Transform

- in camera/view space, the camera is at the origin, looking into negative z
- *modelview matrix* is combined model (rotations, translations, scales) and view matrix!
View Transform

- in camera/view space, the camera is at the origin, looking into negative z
Projection Transform

- similar to choosing lens and sensor of camera – specify field of view and aspect
**Projection Transform - Perspective Projection**

- have **symmetric** view frustum
- fovy: vertical angle in degrees
- aspect: ratio of width/height
- $z_{Near}$: near clipping plane (relative from cam)
- $z_{Far}$: far clipping plane (relative from cam)

$$f = \cot(\text{fovy} / 2)$$

$$M_{proj} = \begin{pmatrix}
\frac{f}{\text{aspect}} & 0 & 0 & 0 \\
0 & f & 0 & 0 \\
0 & 0 & -\frac{z_{Far} + z_{Near}}{z_{Far} - z_{Near}} & -\frac{2 \cdot z_{Far} \cdot z_{Near}}{z_{Far} - z_{Near}} \\
0 & 0 & -1 & 0
\end{pmatrix}$$

**projection matrix**

(symmetric frustum)
Projection Transform - Perspective Projection

more general: a perspective “frustum” (truncated, possibly sheared pyramid)

- left (l), right (r), bottom (b), top (t): corner coordinates on near clipping plane (at zNear)

\[
M_{\text{proj}} = \begin{bmatrix}
\frac{2 \cdot z\text{Near}}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & \frac{2 \cdot z\text{Near}}{t-b} & \frac{t+b}{t-b} & 0 \\
0 & 0 & \frac{z\text{Far} + z\text{Near}}{z\text{Far} - z\text{Near}} & -\frac{2 \cdot z\text{Far} \cdot z\text{Near}}{z\text{Far} - z\text{Near}} \\
0 & 0 & -1 & 0
\end{bmatrix}
\]

projection matrix (asymmetric frustum)
Projection Transform

• possible source of confusion for zNear and zFar:

  • Marschner & Shirley define it as absolute z coordinates, thus zNear > zFar and both values are always negative

  • OpenGL and we define it as positive values, i.e. the distances of the near and far clipping plane from the camera (zFar > zNear)
Modelview Projection Matrix

• put it all together with 4x4 matrix multiplications!

\[ v_{\text{clip}} = M_{\text{proj}} \cdot M_{\text{view}} \cdot M_{\text{model}} \cdot v = M_{\text{proj}} \cdot M_{\text{mv}} \cdot v \]

vertex in clip space
projection matrix
modelview matrix
Clip Space

Clip-Volume Space
(2x2x1 Cuboid)

Coordinates Transform Pipeline

Model Transform | View Transform | Projection Transform | Viewport Transform | Screen Space | Display

Vertex Processing | Rasterizer

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Normalized Device Coordinates (NDC)

- not in previous illustration
- get to NDC by perspective division

$$\begin{align*}
v_{\text{clip}} &= \begin{pmatrix} x_{\text{clip}} \\ y_{\text{clip}} \\ z_{\text{clip}} \\ w_{\text{clip}} \end{pmatrix} \\
&\rightarrow \\
v_{\text{NDC}} &= \begin{pmatrix} x_{\text{clip}} / w_{\text{clip}} \\ y_{\text{clip}} / w_{\text{clip}} \\ z_{\text{clip}} / w_{\text{clip}} \\ 1 \end{pmatrix}
\end{align*}$$

vertex in clip space  vertex in NDC
Viewport Transform

- also in matrix form (let’s skip the details)

\[
\begin{bmatrix}
\frac{x_{\text{clip}}}{w_{\text{clip}}} & \frac{y_{\text{clip}}}{w_{\text{clip}}} & \frac{z_{\text{clip}}}{w_{\text{clip}}} & 1
\end{bmatrix}
\]

vertex in NDC

\[
\begin{bmatrix}
x_{\text{window}} \\
y_{\text{window}} \\
z_{\text{window}} \\
1
\end{bmatrix} \in (0, \text{win}_\text{width} - 1) \times (0, \text{win}_\text{height} - 1) \times (0, 1)
\]

vertex in window coords
The Graphics Pipeline – Another Illustration

Object Space

Modeling Transform

World Space

View Transform

Eye Space

Projection Transform

Clip Space

Perspective Divide

Normalized Device Space

Viewport and Depth Range Transform

Window Space
Note on Rotations with Quaternions

• successive rotations around each axis are most common, but sometimes problematic

• axis and angle representation is an alternative

• can be conveniently represented by quaternion (extension of complex numbers to 3 imaginary values)

• get back to that later, so stay tuned!
The Graphics Pipeline

all vertex transforms from today!

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
• assign fixed color (e.g. red) to each vertex in window coordinates (fragment)
• interpolate (i.e. rasterize) lines between vertices (as defined by user)
... and we can almost do this ...
Summary

• graphics pipeline is a series of operations that takes 3D vertices/normals/triangles as input and generates fragments and pixels

• today, we only discussed a small part of it: vertex and normal transforms

• transforms include: rotation, scale, translation, perspective projection, perspective division, and viewport transform

• most transforms are represented as 4x4 matrices in homogeneous coordinates
  ➔ know your matrices & be able to use GLM to create, manipulate, invert them!
Next Lecture: Lighting and Shading, Fragment Processing

- vertex shader
  - transforms & (per-vertex) lighting
- fragment shader
  - texturing
  - (per-fragment) lighting

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Further Reading

• good overview of OpenGL (deprecated version) and graphics pipeline (missing a few things) :
  https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html


• WebGL / three.js tutorials: https://threejs.org/