The Graphics Pipeline and OpenGL I:
Transformations

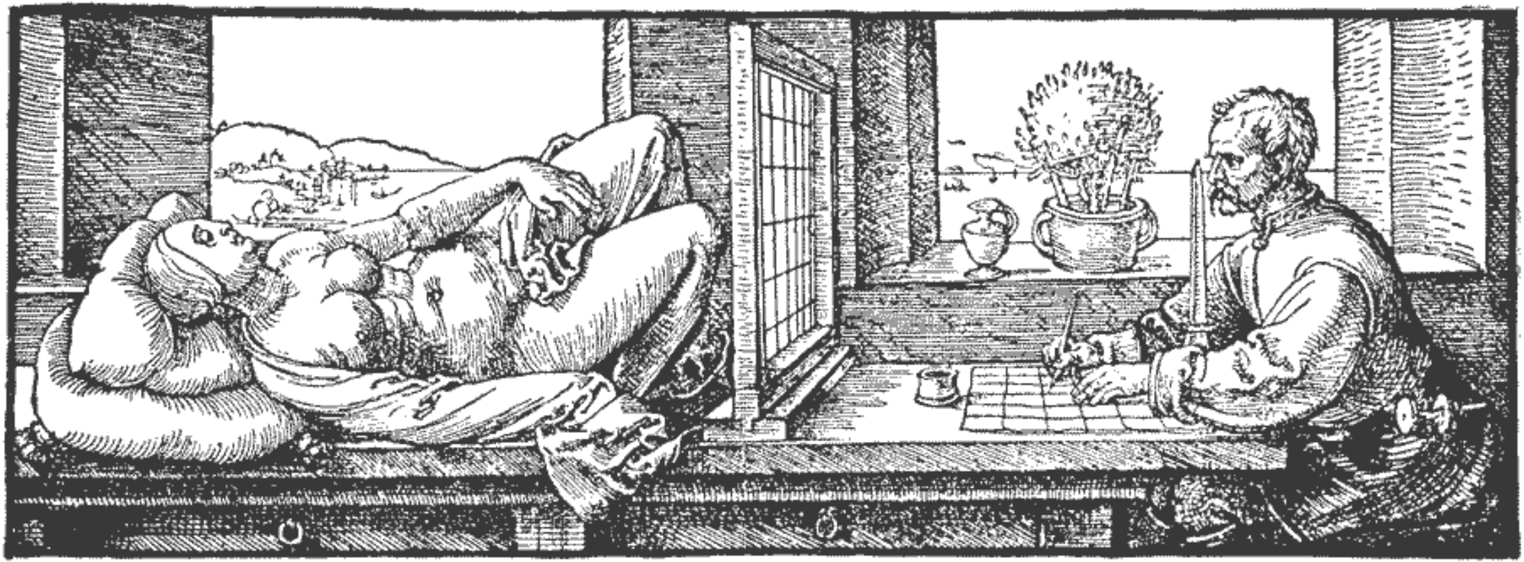
Gordon Wetzstein

Stanford University

EE 267 Virtual Reality

Lecture 2

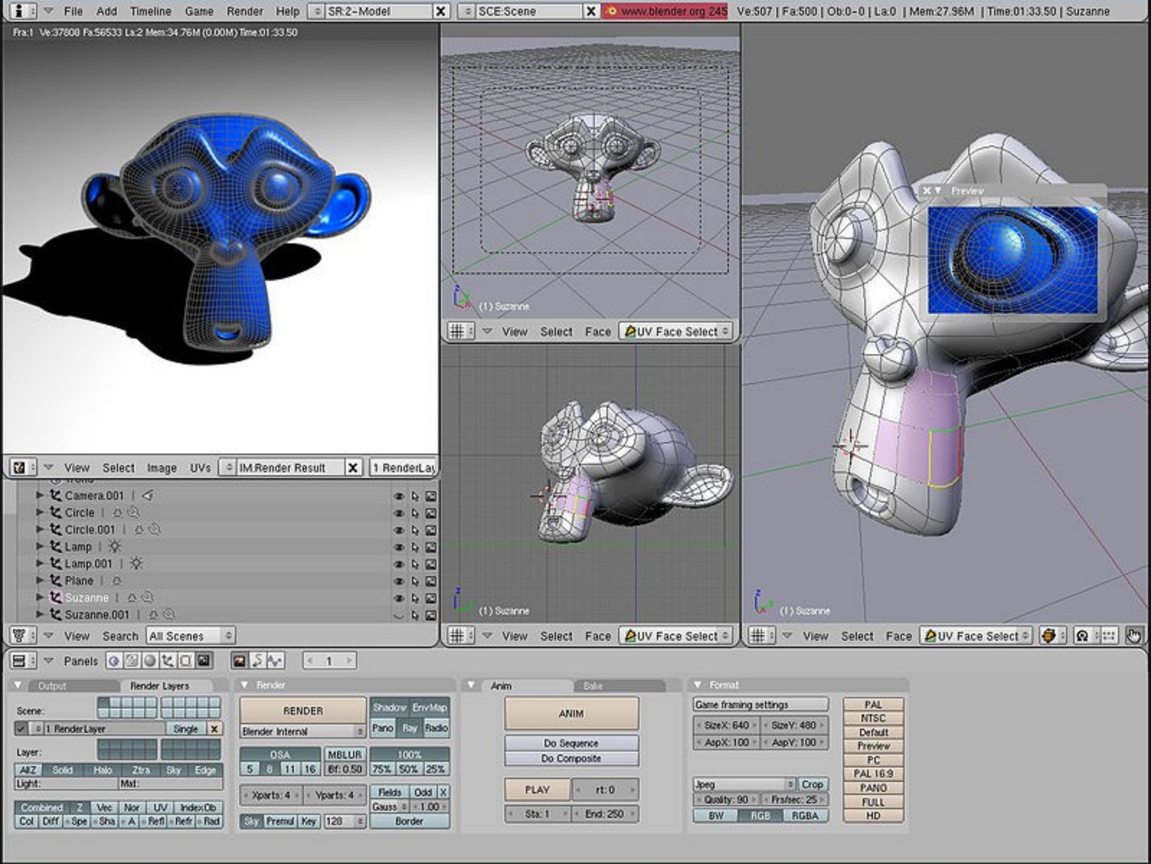
stanford.edu/class/ee267/



Albrecht Dürer, “Underweysung der Messung mit dem Zirckel und Richtscheyt”, 1525

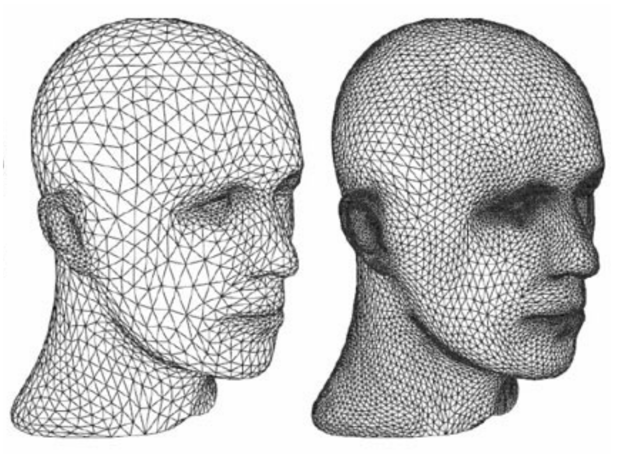
Lecture Overview

- what is computer graphics?
- the graphics pipeline
- primitives: vertices, edges, triangles!
- model transforms: translations, rotations, scaling
- view transform
- perspective transform
- window transform


File Add Timeline Game Render Help
SR:2-Model
SCE:Scene
www.blender.org 245
Ve:507 | Fa:500 | Ob:0-0 | La:0 | Mem:27.96M | Time:01:33.50 | Suzanne
View Select Image UVs IM:Render Result
Camera.001
Circle
Circle.001
Lamp
Lamp.001
Plane
Suzanne
Suzanne.001
View Search All Scenes
(1) Suzanne
View Select Face UV Face Select
Preview
Panels
Output Render Layers
Render
RENDER
Blender Internal
Anim
ANIM
Do Sequence
Do Composite
PLAY
Format
Game framing settings
SizeX: 640
SizeY: 480
AspX: 100
AspY: 100
PAL
NTSC
Default
Preview
PC
PAL 16:9
PANO
FULL
HD
Jpeg
Crop
Quality: 90
Frs/sec: 25
BW RGB RGBA

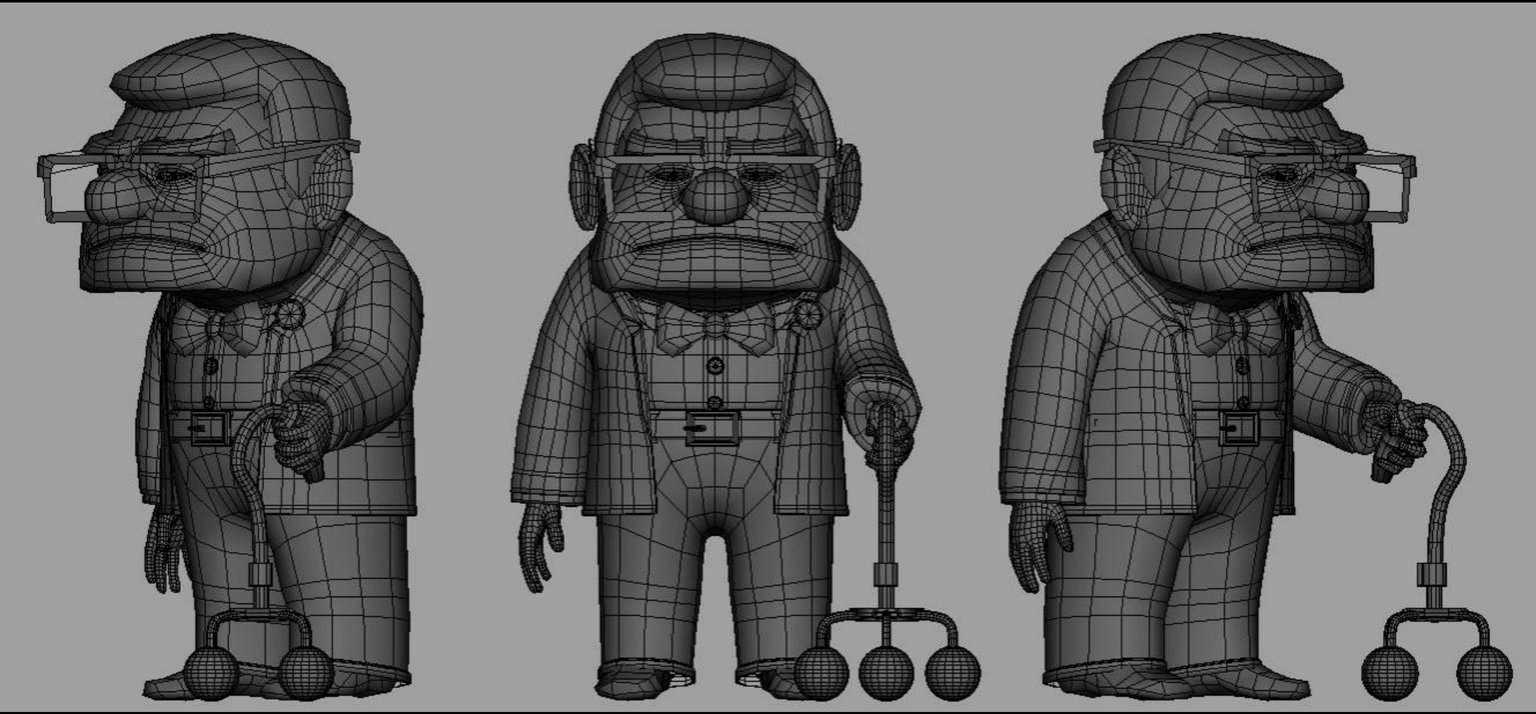
blender.org

Modeling 3D Geometry



Courtesy of H.G. Animations
https://www.youtube.com/watch?v=fewbFvA5oGk

blender.org



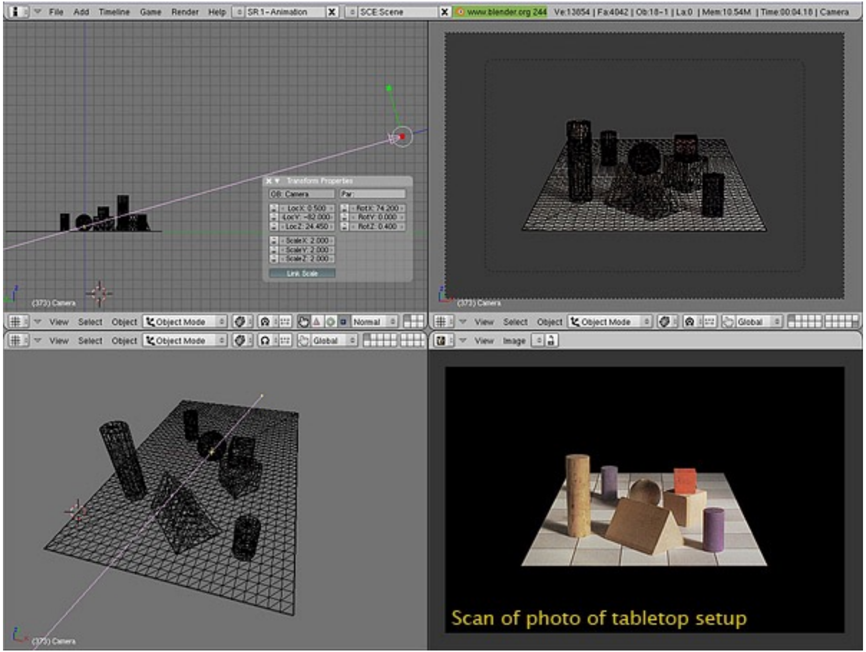
What is Computer Graphics?

- 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

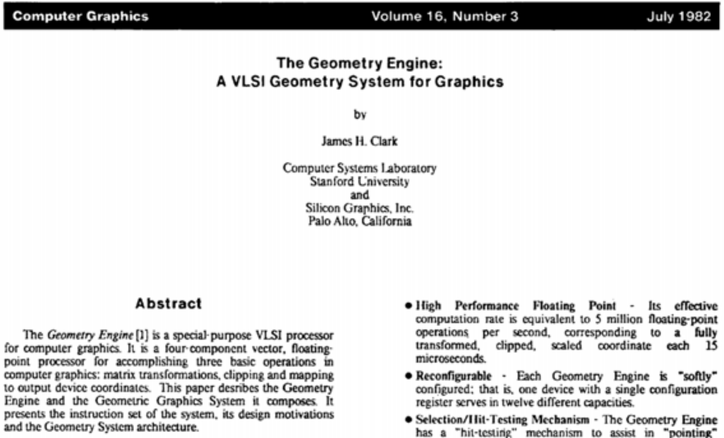
blender.org

- 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



Computer Graphics Volume 16, Number 3 July 1982

**The Geometry Engine:
A VLSI Geometry System for Graphics**

by

James H. Clark

Computer Systems Laboratory
Stanford University
and
Silicon Graphics, Inc.
Palo Alto, California

Abstract

The *Geometry Engine* [1] is a special-purpose VLSI processor for computer graphics. It is a four-component vector, floating-point processor for accomplishing three basic operations in computer graphics: matrix transformations, clipping and mapping to output device coordinates. This paper desribes the Geometry Engine and the Geometric Graphics System it composes. It presents the instruction set of the system, its design motivations and the Geometry System architecture.

- **High Performance Floating Point** - Its effective computation rate is equivalent to 5 million floating-point operations per second, corresponding to a fully transformed, clipped, scaled coordinate each 15 microseconds.
- **Reconfigurable** - Each Geometry Engine is "softly" configured; that is, one device with a single configuration register serves in twelve different capacities.
- **Selection/Hit-Testing Mechanism** - The Geometry Engine has a "hit-testing" mechanism to assist in "pointing"

SiliconGraphics

Some History

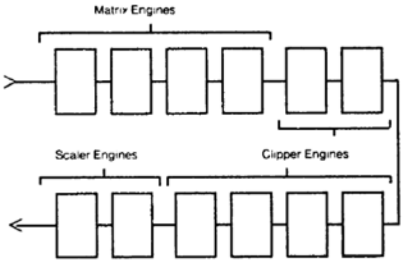
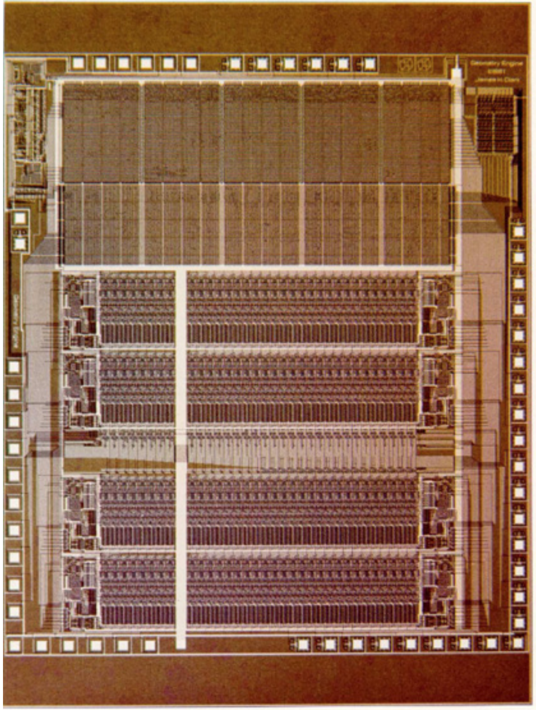


Figure 3: Geometry System; each block is a Geometry Engine.

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

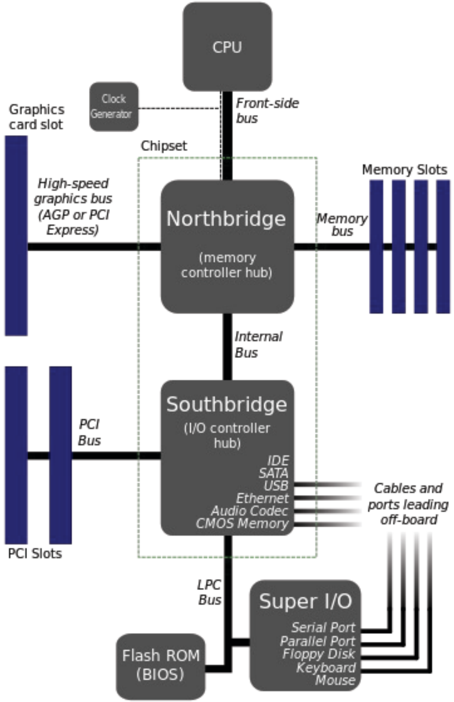
- early versions of these GPUs implemented *fixed-function* rendering pipeline in hardware

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

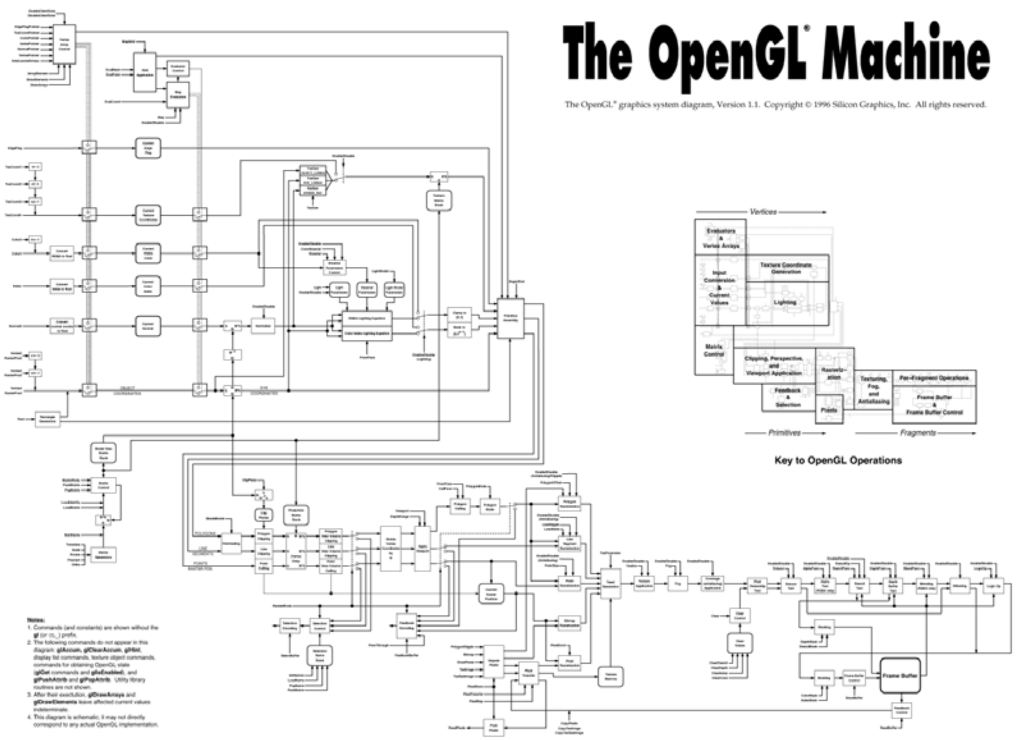
- now: GPUs = programmable (e.g. OpenGL, CUDA, OpenCL) processors

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



I had this poster hanging on my dorm wall during undergrad

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: see lab 1

three.js

- cross-browser JavaScript library/API

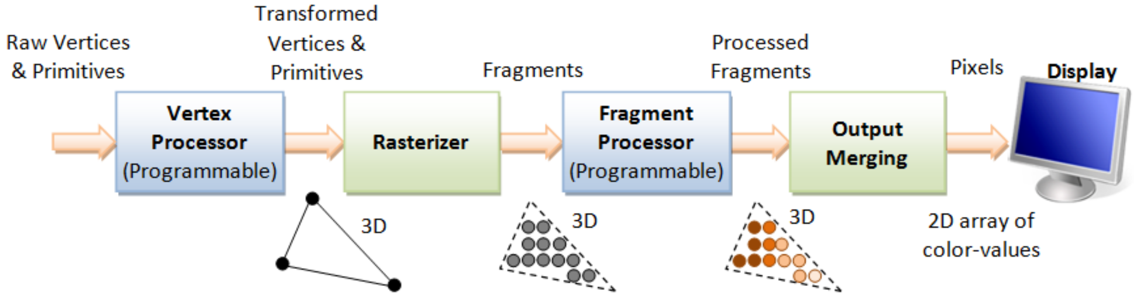
- higher-level library that provides a lot of useful helper functions, tools, and abstractions around WebGL – easy and convenient to use

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

- great introduction (in WebGL):

  http://davidscottlyons.com/threejs/presentations/frontporch14/

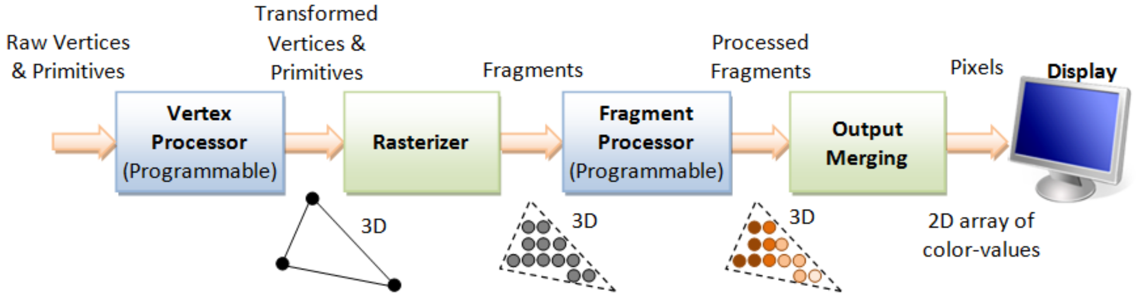
The Graphics Pipeline



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

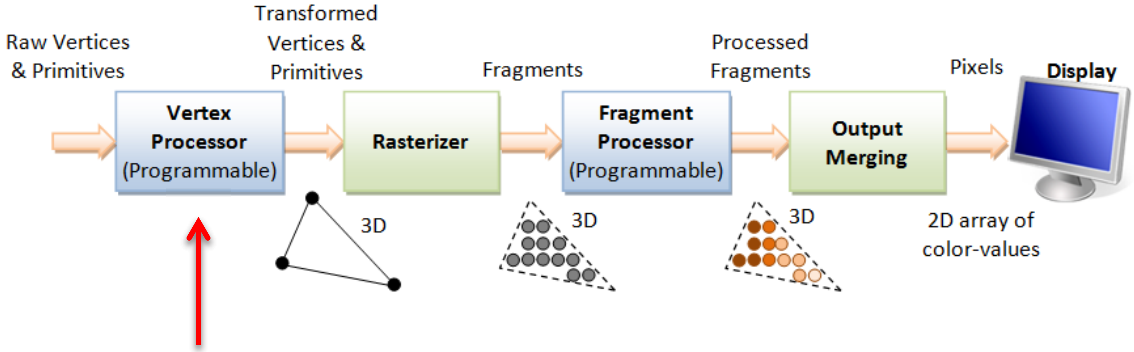
The Graphics Pipeline



1. Vertex Processing: Process and transform individual vertices & normals.

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

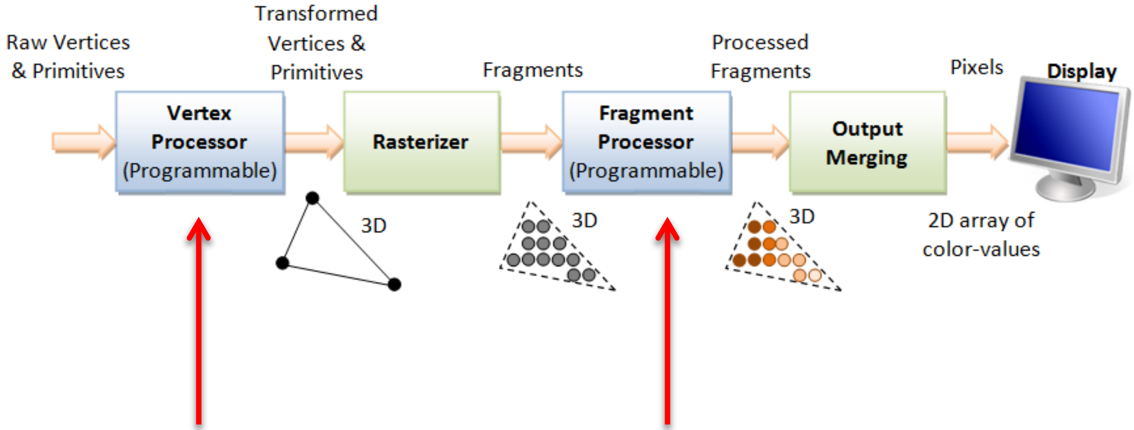
The Graphics Pipeline



1. Vertex Processing: Process and transform individual vertices & normals.
2. Rasterization: Convert each primitive (connected vertices) into a set of fragments. A fragment can be interpreted as a pixel 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 shader

- transforms & (per-vertex) lighting

fragment shader

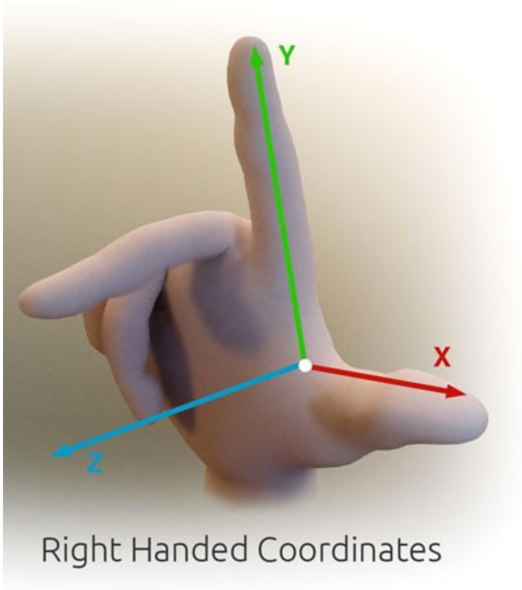
- texturing
- (per-fragment) lighting

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

Coordinate Systems

- right hand coordinate system

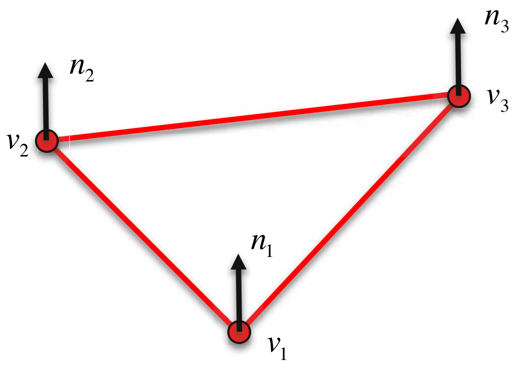
- a few different coordinate systems:
    - object coordinates
    - world coordinates
    - viewing coordinates
    - also clip, normalized device, and window coordinates



wikipedia

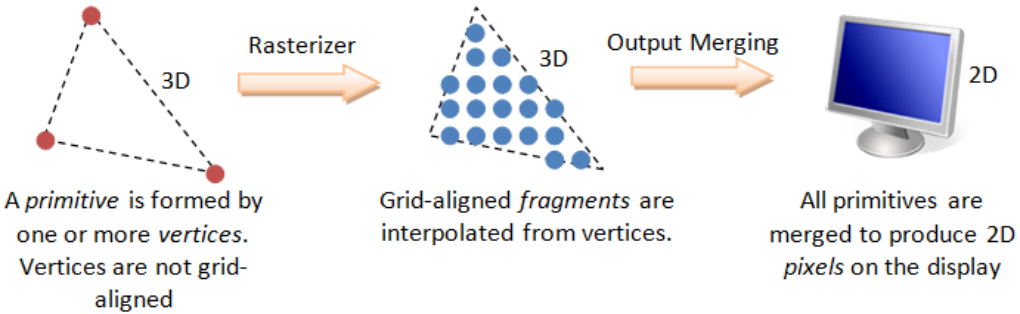
Primitives

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



Pixels v Fragments

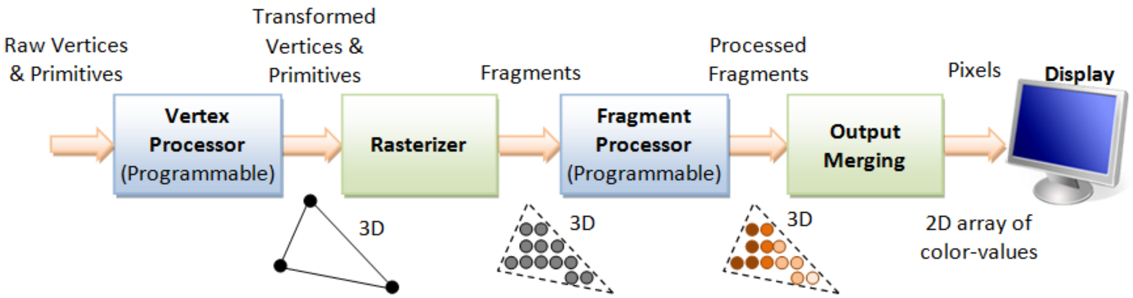
- fragments have rasterized 2D coordinates on screen but a lot of other attributes too (texture coordinates, depth value, alpha value, …)
- pixels appear on screen
- won't discuss in more detail today



Vertex, Primitives, Fragment and Pixel

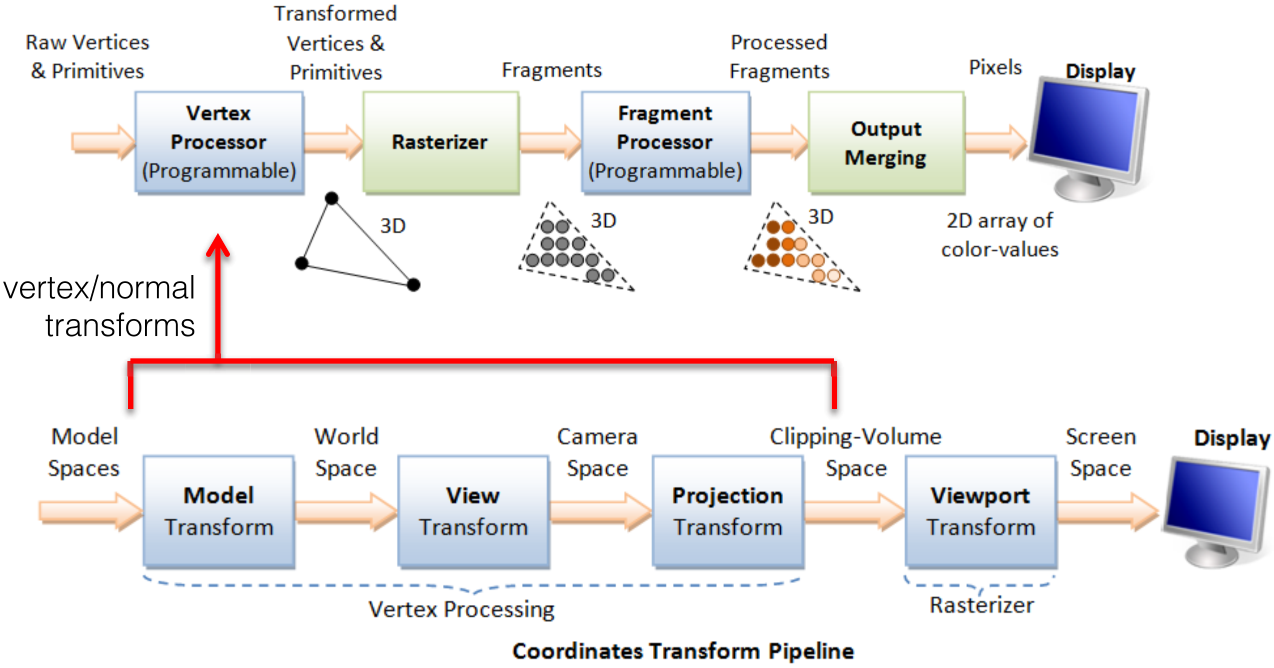
https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

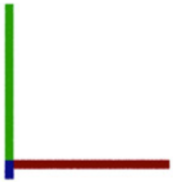


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

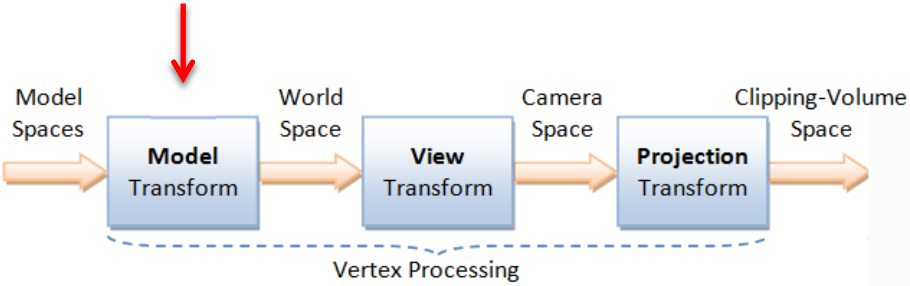


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html



Vertex Transforms

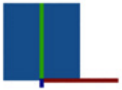
1. Arrange the objects (or models, or avatar) in the world (Model Transform).

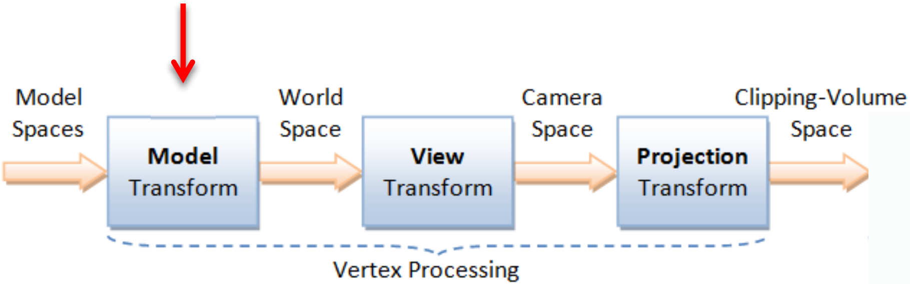


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

1. Arrange the objects (or models, or avatar) in the world (Model Transform).

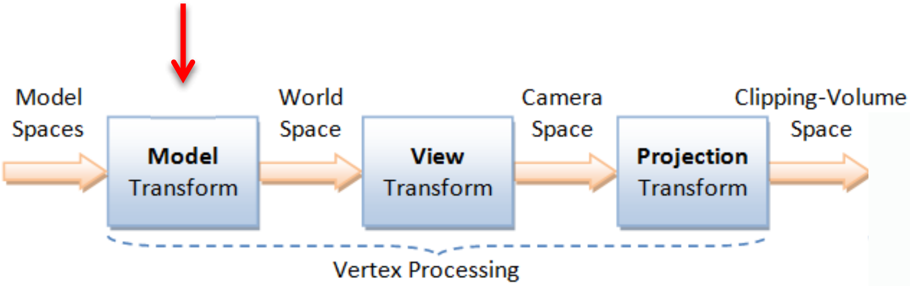




https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

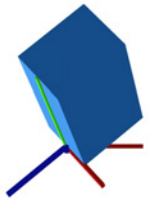
1. Arrange the objects (or models, or avatar) in the world (Model Transform).

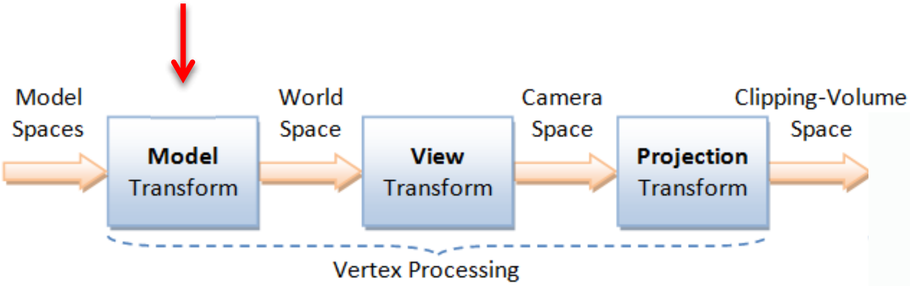


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

1. Arrange the objects (or models, or avatar) in the world (Model Transform).

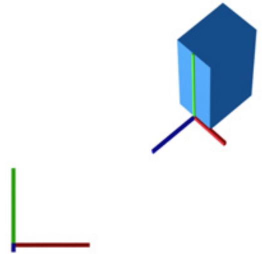


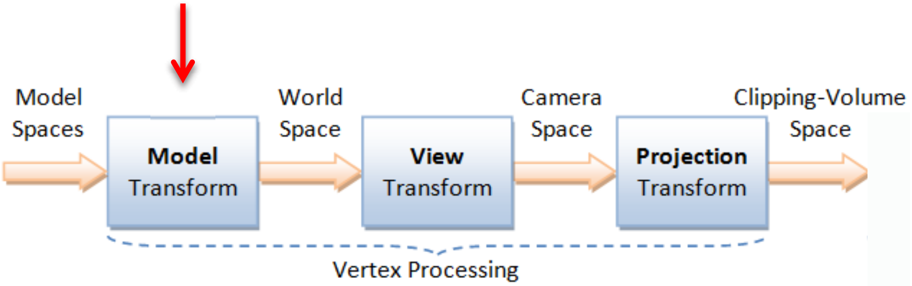


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

1. Arrange the objects (or models, or avatar) in the world (Model Transform).

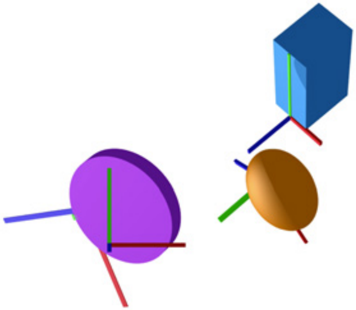


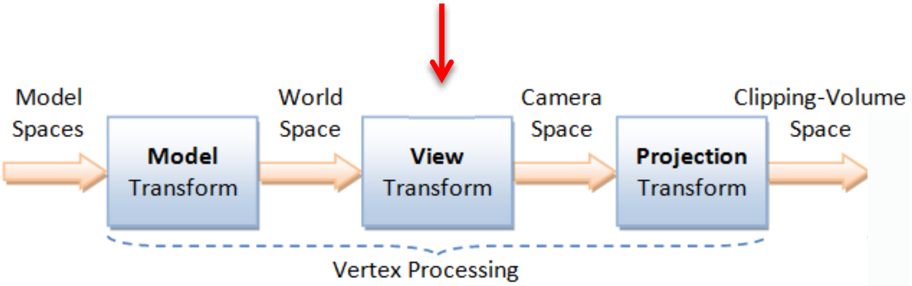


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

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

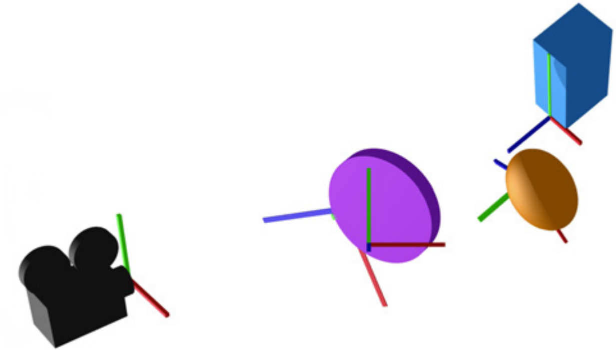


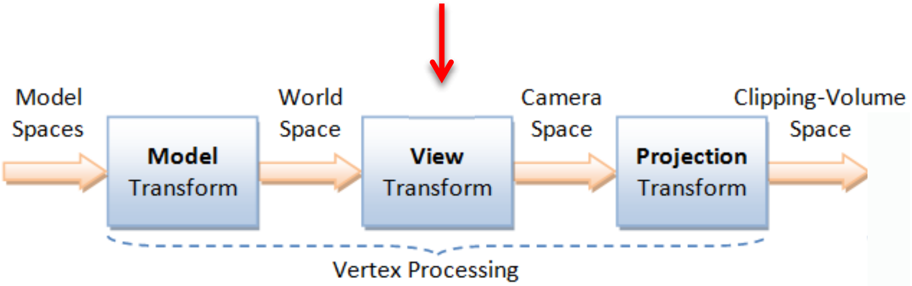


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

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

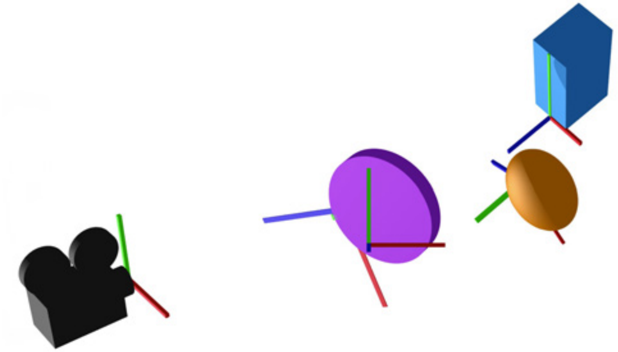


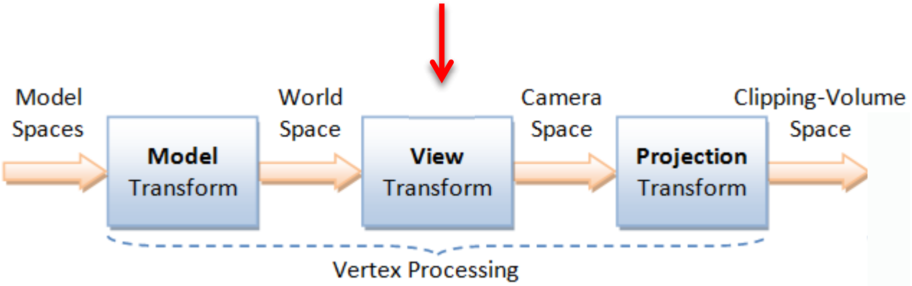


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

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

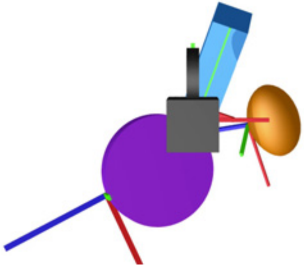


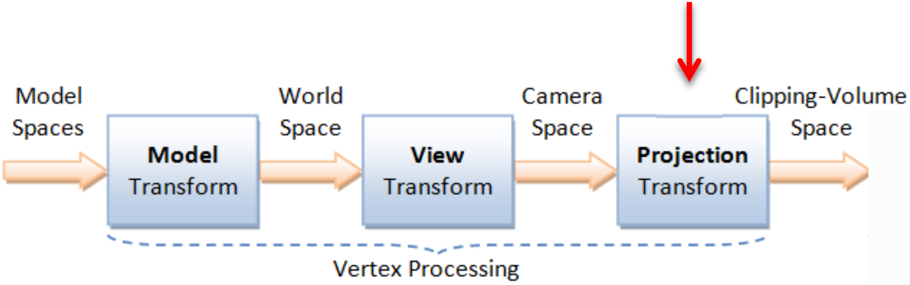


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

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

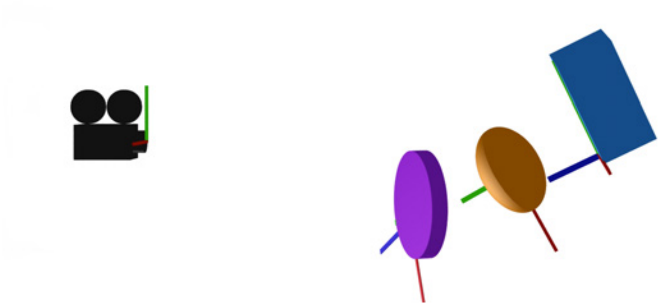


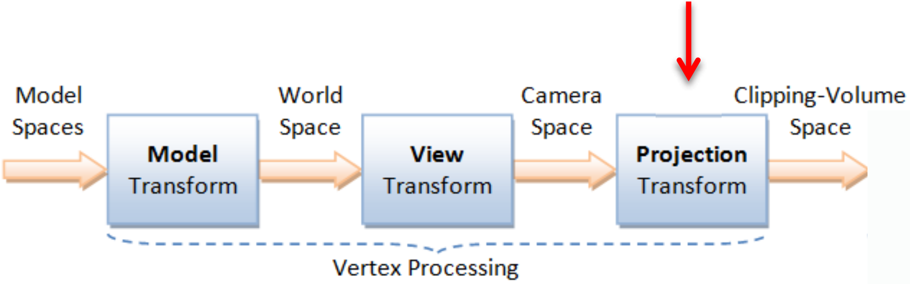


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

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

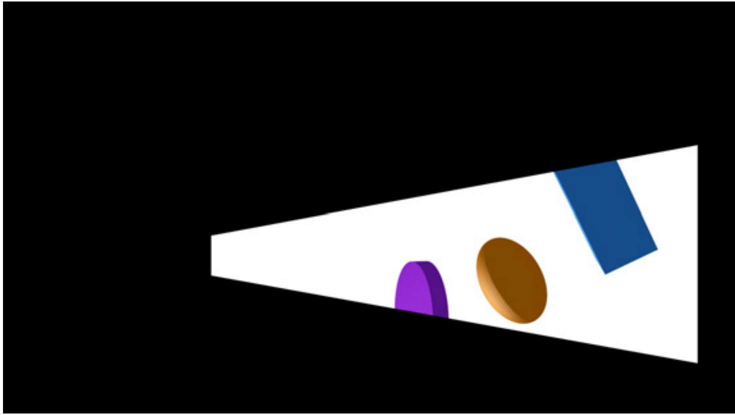


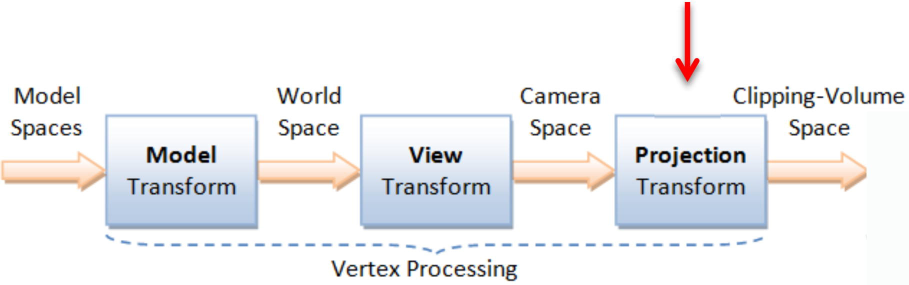


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

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

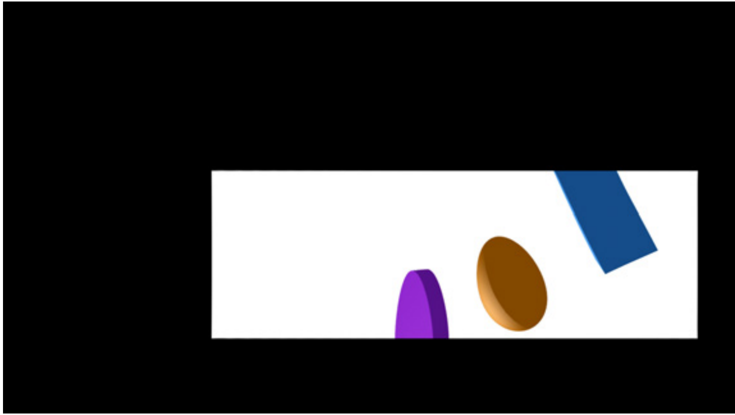


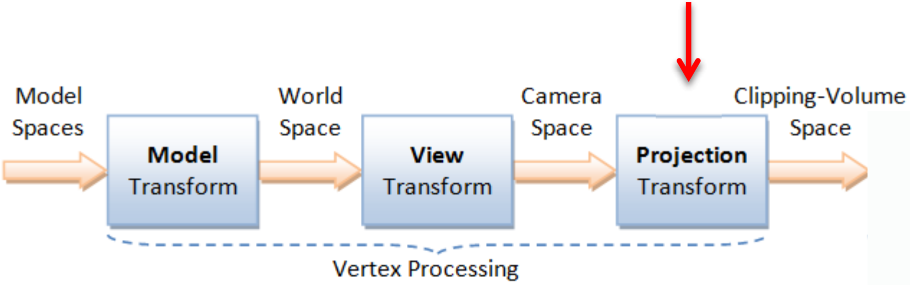


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex Transforms

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

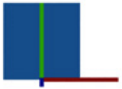


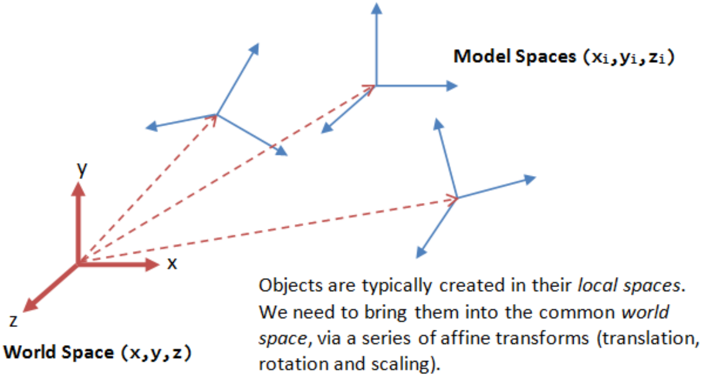


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Model Transform

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





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.

$$R_z v = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ z \end{pmatrix}$$

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:
$$\begin{aligned} v &= \left(T \cdot S \cdot R_z \cdot R_x \cdot T\right)^{-1} \cdot v' \\ &= T^{-1} \cdot R_x^{-1} \cdot R_z^{-1} \cdot S^{-1} \cdot T^{-1} \cdot v' \end{aligned}$$

Read more: https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

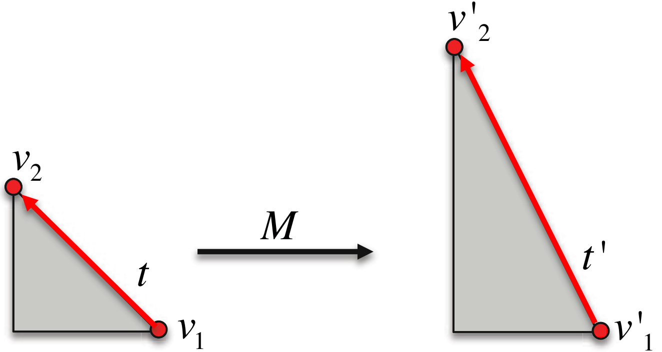
Vector and Normal Transforms

- 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!



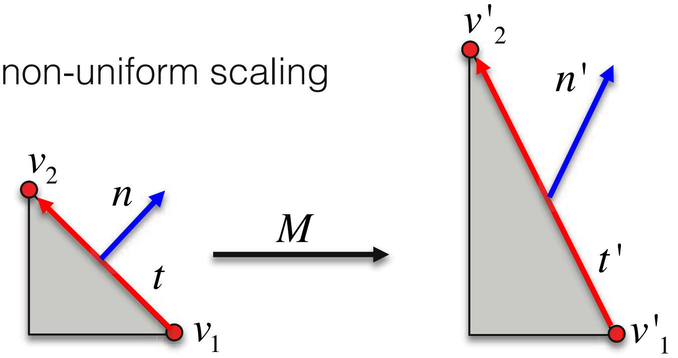
Vector and Normal Transforms

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

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

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

- this does NOT work! (non-uniform scaling is a problem)



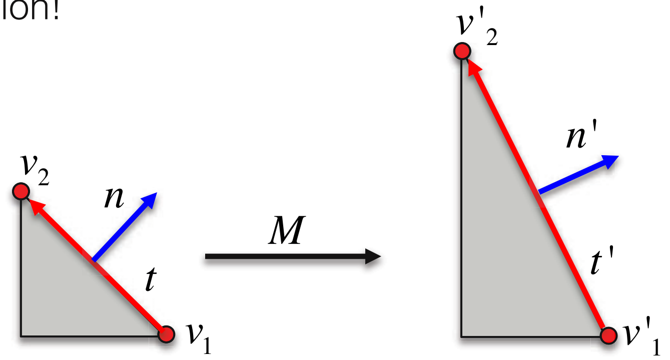
Vector and Normal Transforms

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

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

- need to use normal matrix = transpose of inverse for transformation!

$$n' = \left(M^{-1}\right)^T \cdot n$$



Vector and Normal Transforms

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

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

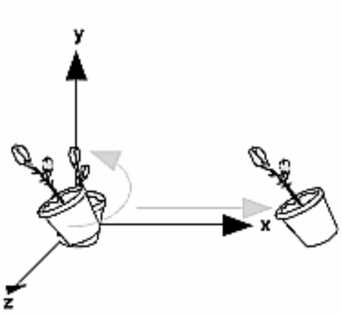
- need to use normal matrix = transpose of inverse for transformation!

$$n' = \left(M^{-1}\right)^T \cdot n$$

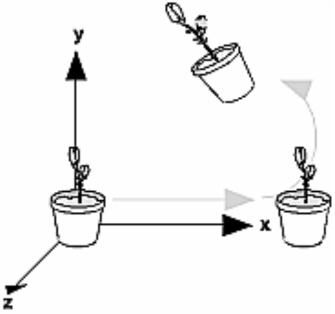
- fine print: only use upper left 3x3 part of modelview matrix for inverse transpose (no homogeneous normal representation) OR drop *w* component from *n'* after multiplying 4x4 inverse transpose (i.e. don't use *w* for normalization of *n'*!)

Attention!

- rotations and translations (or transforms in general) are not commutative!
- make sure you get the correct order!



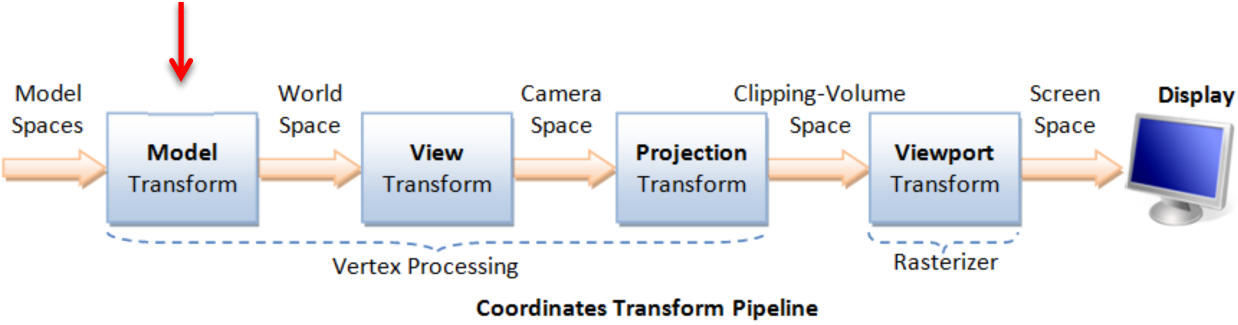
Rotate then Translate



Translate then Rotate

View Transform

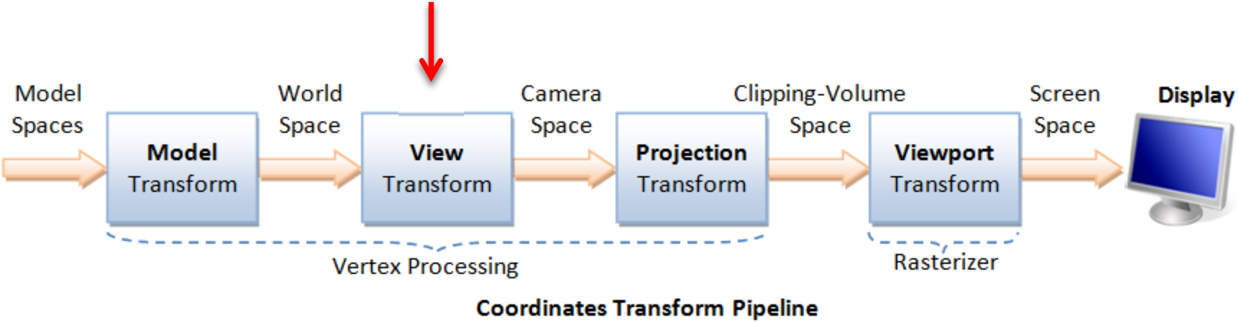
- so far we discussed model transforms, e.g. going from object or model space to world space



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

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!

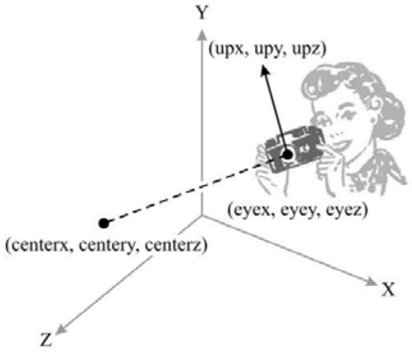


https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

View Transform

specify camera by

- eye position $eye = \begin{pmatrix} eye_x \\ eye_y \\ eye_z \end{pmatrix}$
- reference position $center = \begin{pmatrix} center_x \\ center_y \\ center_z \end{pmatrix}$
- up vector $up = \begin{pmatrix} up_x \\ up_y \\ up_z \end{pmatrix}$



View Transform

specify camera by

- eye <u>position</u> $eye = \begin{pmatrix} eye_x \\ eye_y \\ eye_z \end{pmatrix}$
- reference <u>position</u> $center = \begin{pmatrix} center_x \\ center_y \\ center_z \end{pmatrix}$
- up <u>vector</u> $up = \begin{pmatrix} up_x \\ up_y \\ up_z \end{pmatrix}$

compute 3 vectors:

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

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

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

View Transform

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

compute 3 vectors:

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

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

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

View Transform

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

compute 3 vectors:

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

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

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

$$M = R \cdot T(-e) = \begin{pmatrix} x_x^c & x_y^c & x_z^c & 0 \\ y_x^c & y_y^c & y_z^c & 0 \\ z_x^c & z_y^c & z_z^c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

View Transform

view transform M is translation into eye position,

followed by rotation

$$M = R \cdot T(-e) = \begin{pmatrix} x_x^c & x_y^c & x_z^c & -\left(x_x^c eye_x + x_y^c eye_y + x_z^c eye_z\right) \\ y_x^c & y_y^c & y_z^c & -\left(y_x^c eye_x + y_y^c eye_y + y_z^c eye_z\right) \\ z_x^c & z_y^c & z_z^c & -\left(z_x^c eye_x + z_y^c eye_y + z_z^c eye_z\right) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

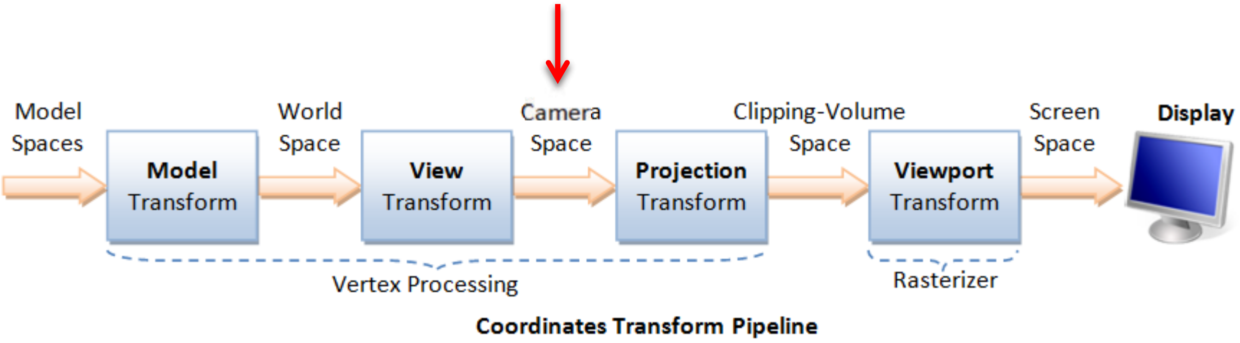
View Transform – Attention!

- many graphics APIs have a function called lookat that automatically computes the view matrix for you

- Three.js also has such a function, but that only computes the rotation, not the translation, of the view matrix. So best implement the view matrix yourself!

View Transform

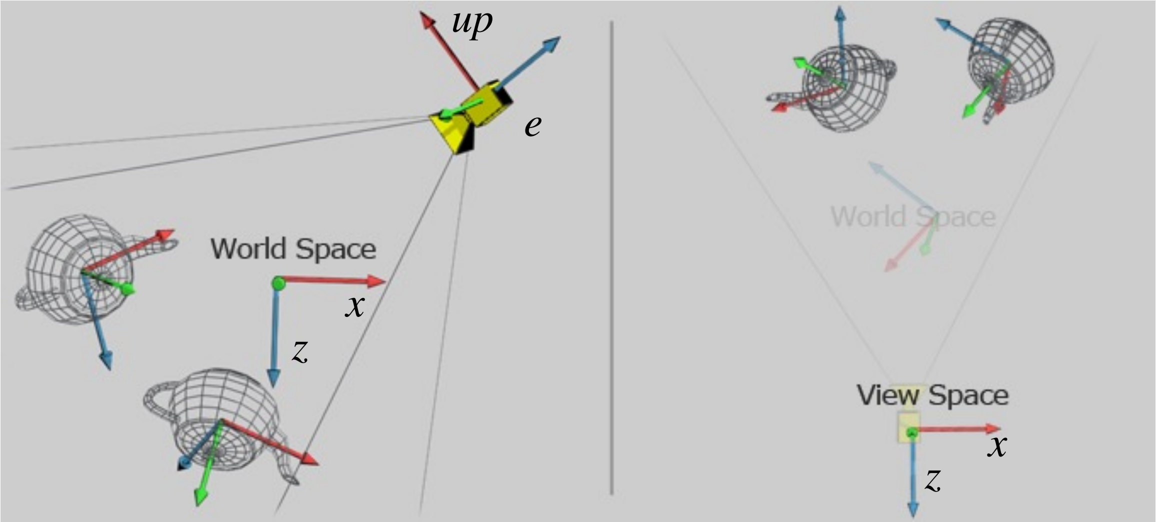
- in camera/view space, the camera is at the origin, looking into negative z
- *modelview matrix* is combined model (rotations, translations, scaling) and view matrix!



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

View Transform

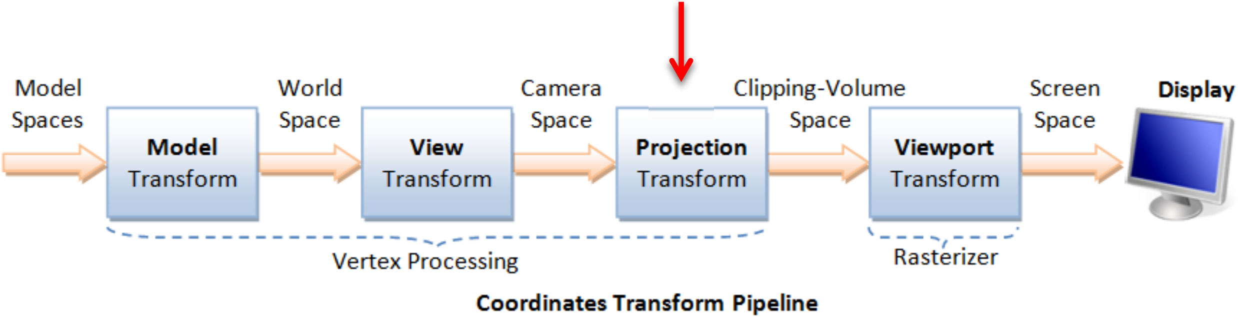
- in camera/view space, the camera is at the origin, looking into negative z



vodacek.zvb.cz

Projection Transform

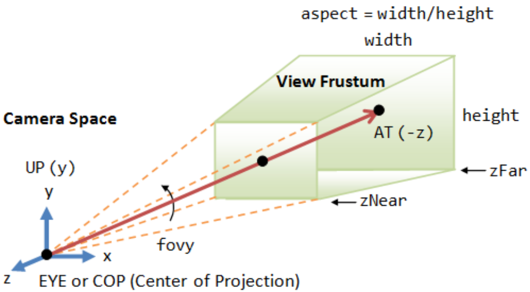
- similar to choosing lens and sensor of camera – specify field of view and aspect



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

Projection Transform - Perspective Projection

- have symmetric view frustum
- fovy: vertical angle in degrees
- aspect: ratio of width/height
- zNear: near clipping plane (relative from cam)
- zFar: far clipping plane (relative from cam)



Perspective Projection: The camera's view frustum is specified via 4 view parameters: fovy, aspect, zNear and zFar.

$$f = \cot(fovy/2)$$

$$M_{proj} = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & -\frac{zFar + zNear}{zFar - zNear} & -\frac{2 \cdot zFar \cdot zNear}{zFar - zNear} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

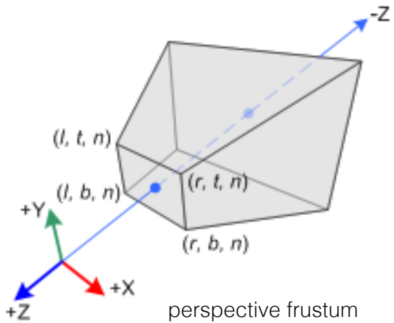
projection matrix

(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)



perspective frustum

$$M_{proj} = \begin{pmatrix} \frac{2 \cdot zNear}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2 \cdot zNear}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{zFar+zNear}{zFar-zNear} & -\frac{2 \cdot zFar \cdot zNear}{zFar-zNear} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

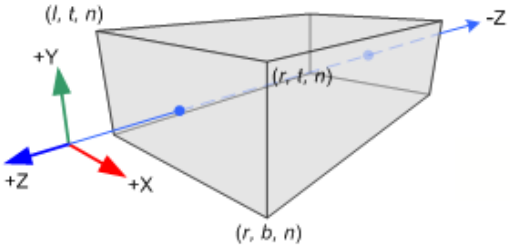
projection matrix

(asymmetric frustum)

Projection Transform - Orthographic Projection

more general: a "box frustum" (no perspective, objects don't get smaller when farther away)

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



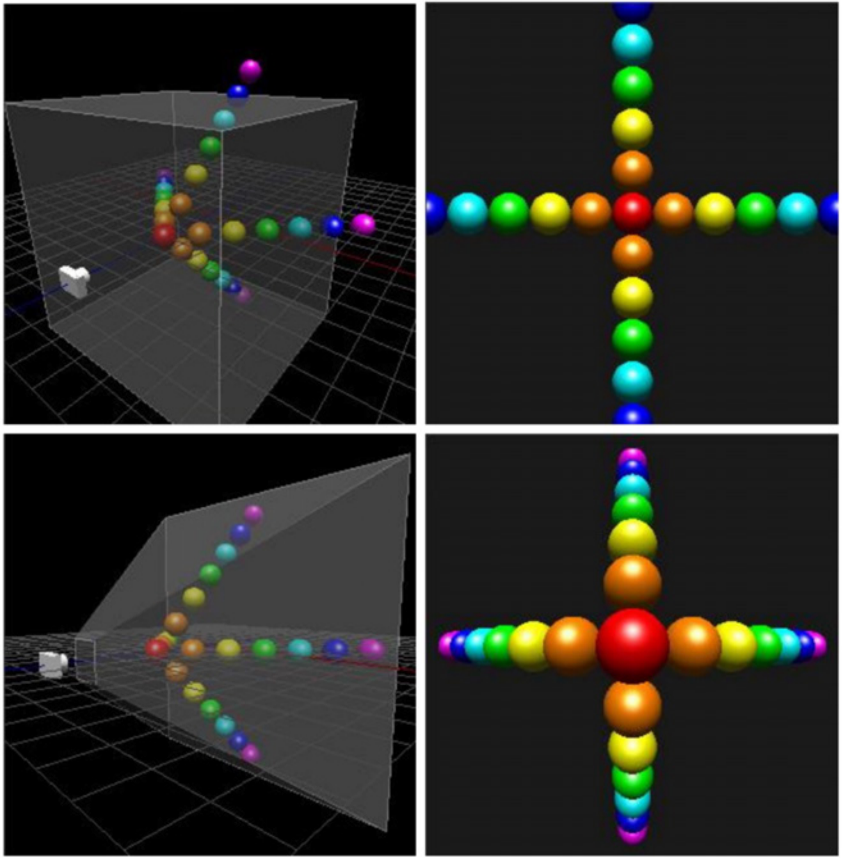
$$M_{proj} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

projection matrix

(orthographic)

Orthographic

Perspective



[Song Ho Ahn]

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!

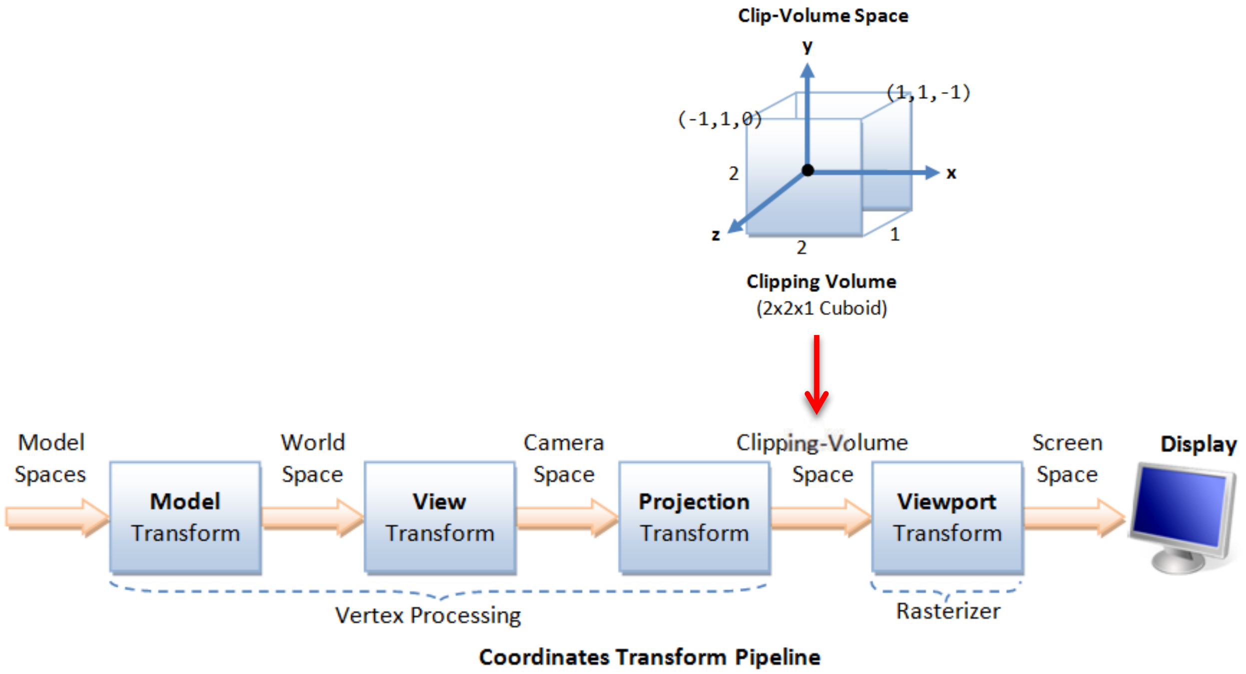
$$v_{clip} = M_{proj} \cdot M_{view} \cdot M_{model} \cdot v = M_{proj} \cdot M_{mv} \cdot v$$

vertex in clip space

projection matrix

modelview matrix

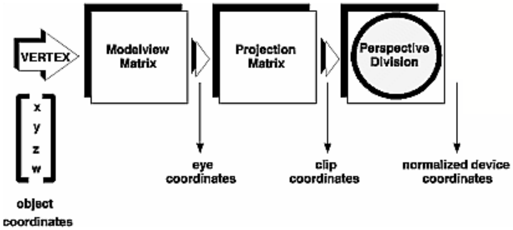
Clip Space



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

Normalized Device Coordinates (NDC)

from: OpenGL Programming Guide



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

$$v_{clip} = \begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} \longrightarrow v_{NDC} = \begin{pmatrix} x_{clip} / w_{clip} \\ y_{clip} / w_{clip} \\ z_{clip} / w_{clip} \\ 1 \end{pmatrix} \begin{matrix} \in (-1,1) \\ \in (-1,1) \\ \in (-1,1) \\ \end{matrix}$$

vertex in clip space

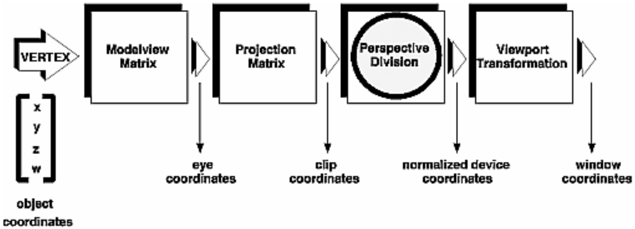
vertex in NDC

Viewport Transform

define (sub)window as viewport(x,y,width,height),

- x,y lower left corner of viewport rectangle (default is (0,0))
- width, height size of viewport rectangle in pixels

from: OpenGL Programming Guide



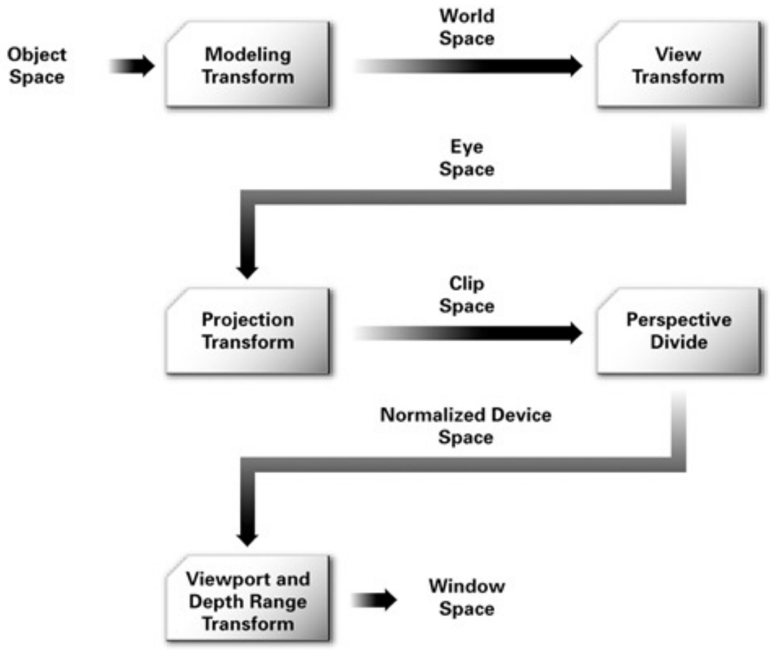
$$
\begin{aligned}
x_{window} &= \frac{width}{2}\left(x_{NDC}+1\right)+x \\
y_{window} &= \frac{height}{2}\left(y_{NDC}+1\right)+y \\
z_{window} &= \frac{1}{2}z_{NDC}+\frac{1}{2}
\end{aligned}
$$

$$
v_{NDC} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \\ 1 \end{pmatrix} \longrightarrow v_{window} = \begin{pmatrix} x_{window} \\ y_{window} \\ z_{window} \\ 1 \end{pmatrix} \begin{matrix} \in\left(0,width\right) \\ \in\left(0,height\right) \\ \in\left(0,1\right) \\ \end{matrix}
$$

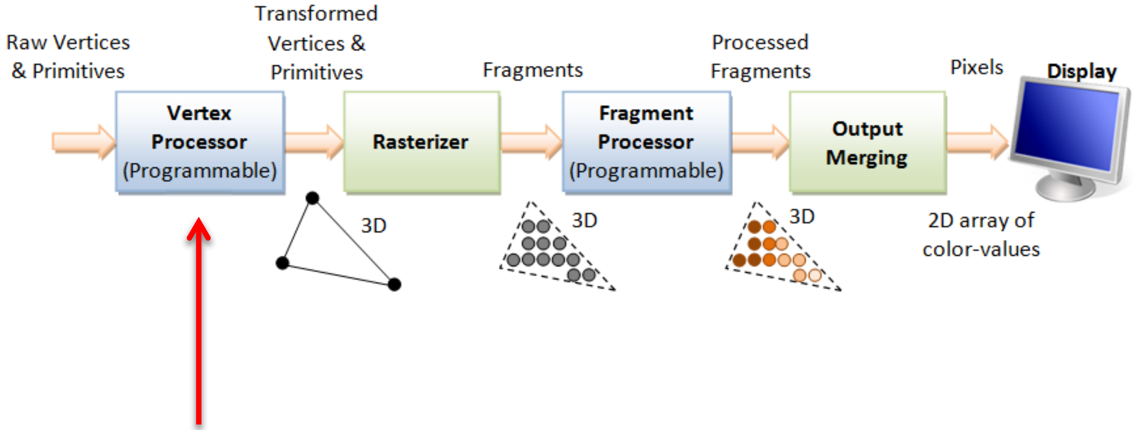
vertex in NDC

vertex in window coords

The Graphics Pipeline – Another Illustration



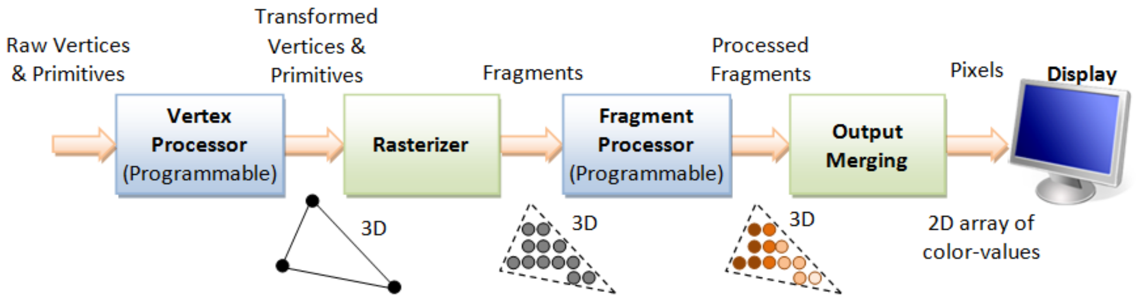
The Graphics Pipeline



all vertex transforms from today!

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

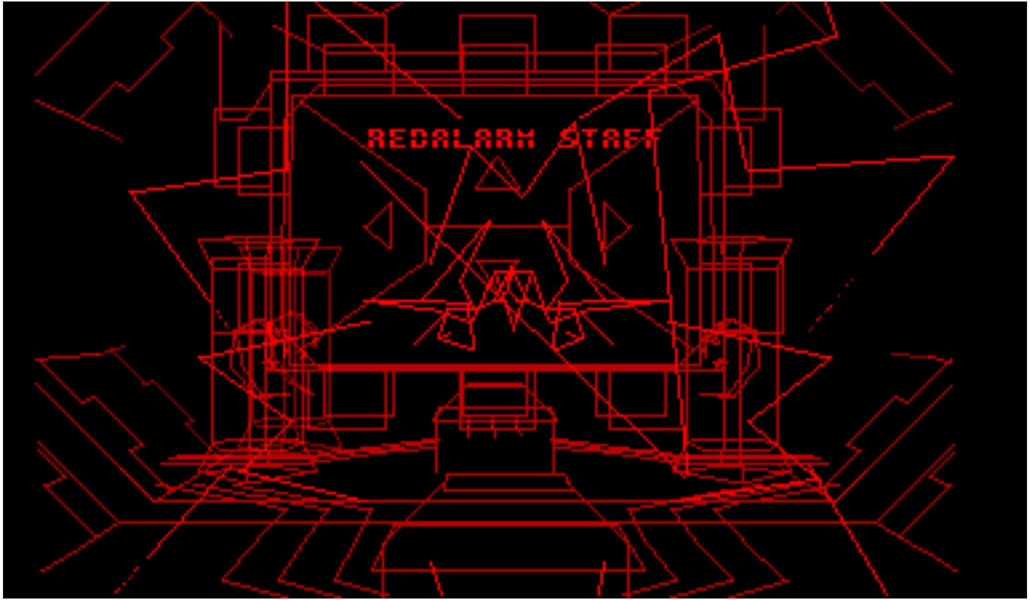
The Graphics Pipeline



- assign fixed color (e.g. red) to each vertex in window coordinates (fragment)
- interpolate (i.e. rasterize) lines between vertices (as defined by user)

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

… and we can almost do this …



Nintendo Virtual Boy Game “Red Alarm”

Summary

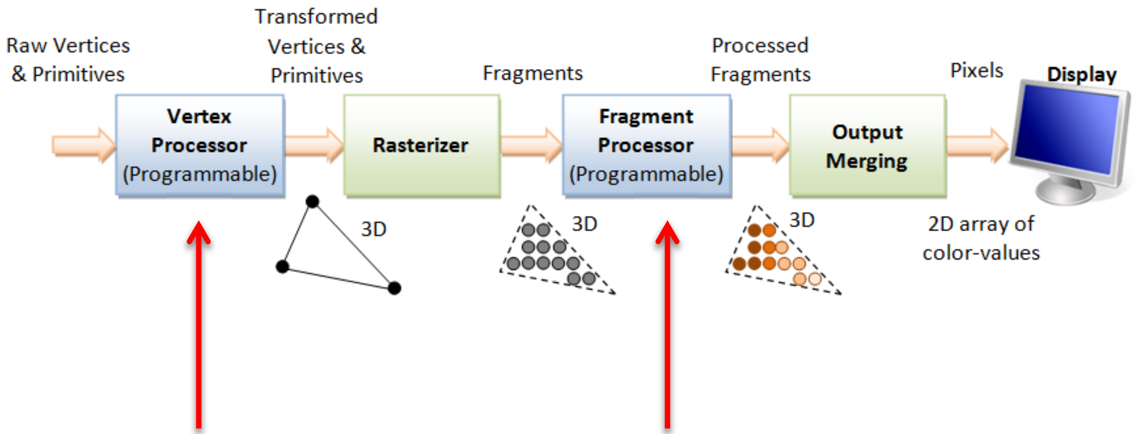
- graphics pipeline is a series of operations that takes 3D vertices/normals/triangles as input and generates fragments and pixels

- today, we only discussed a 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 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

- course notes on transforms (see course website)

- 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

- textbook: Shirley and Marschner “Fundamentals of Computer Graphics”, AK Peters, 2009

- definite reference: “OpenGL Programming Guide” aka “OpenGL Red Book”

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