CS174: Computer Graphics

Professor Law

Thilan Tran

Fall 2021

Contents

CS174: Introduction to Computer Graphics

- computer graphics is the field of creating imagery by computer
  - used in the entertainment industry eg. movies and games, visualization applications, etc.
- basic elements of CG include:
  - **modeling** ie. mathematically representing objects:
    - constructing models for specific objects
    - makes use of 3D points, lines, curves, surfaces, polygons
      - volumetric vs. image-based representations
    - primitives can have attributes like colors and texture maps
  - **animation** ie. representing the motions of objects:
    - also give animators control of the motion
    - eg. keyframe animation, motion capture, procedural animation
  - **rendering** ie. simulating the real-world behavior of light and formation of images:
    - simulate light propagation
      - includes 3D scene, lighting, point-of-view, shading, projection
    - properties include reflection, absorption, scattering, emission, interference
  - **interaction** ie. enabling humans and computers to interact

Basic Graphics System

- a basic graphics system includes:
  - input devices
  - CPU and GPU:
    - CPU takes input from the user, and calculates how the world should update accordingly
    - CPU passes to GPU the world changes, and GPU generates an image that can be stored as a **framebuffer**
  - computing and rendering system
  - output devices
    - takes in framebuffer and displays it on the screen scanline by scanline

Output Devices

- in **cathode ray tube (CRT)** technology, electrons strike phosphorous coating and emit light:
  - direction of electron beam controlled by deflection plates
  - with random-scan or vector CRT, can only “burn” lines into the screen:
    - deflection plates randomly target different parts of the screen to burn in points
    - difficult to fill polygons, etc.
  - refresh rate between 60 to 85 Hz
    - burning only lasts a few milliseconds
- in **raster CRT**, the screen is broken up into pixels eg. n by m phosphorous cells:
  - a pixel becomes the smallest element we can modify on the screen
  - intrinsically has a rasterizing or aliasing problem due to finiteness of pixels
  - framebuffer depth determines color complexity:
    - 1 bit supports black and white
    - 8 bits gray scale
    - 8 bits per color (RGB) requires 24 bits and creates 16 M colors
    - 12 bits per color is HD
  - 3 different colored electron guns
    - each pixel has 3 different colors phosphors arranged in triads
  - a shadow mask helps to prevent electron beams from bleeding over into neighboring pixels
  - **interlaced** displays update odd to even scanlines:

        * rather than every scanlines, as in **non-interlaced** displays
        * human eye cannot catch the difference, while data bandwidth requirement is halved since only half of the framebuffer needs to be sent
    - potential race condition with the framebuffer:
        * as GPU is updating the framebuffer, the display may be reading it at the same time
        * instead, **double buffering** uses two framebuffers to avoid this data conflict
- typical screen resolutions:
    - TV is 640x480
    - HD is 1920x1080
    - 4K LCD is 3840x2160
    - 35mm is 3000x2000
- memory speed and space requirements:
    - given $n \times m$ resolution, refresh rate r Hz, color depth b bits per pixel
    - memory space per second is $\frac{n*m*b*r}{8}$ bytes
    - if non-interlaced, memory read time is $\frac{1}{n*m*r}$ seconds per pixel
    - if interlaced, memory read time is $\frac{2}{n*m*r}$ seconds per pixel
- flat screen displays no longer have a CRT:
    - still raster based, with an active matrix of transistors at grid points
        * vertical grid plus horizontal grid of wires allows voltages to be customized along the grid to change the lighting at each pixel
    - light sources for pixels may be:
        * **light emitting diodes (LEDs)**
        * **polarized liquid crystals (LCDs)**
        * **plasma**, where gases are energized to glow

Modeling Objects

- a sphere may be most easily modeled by its origin and radius:
    - however, this is not easiest model to render in graphics
        * complicated non-linear formulas
    - we only know how to render polygons
    - need to **discretize** the *surface* of the object into polygons:
        * essentially linearizing shapes into line segments
        * loses information of the inside of the object
    - eg. to tesselate a circle, can arrange many triangles around the origin
- polygons can be represented as collection of points connected with lines:
    - eg. for vertices v_1, v_2, v_3, v_4, the connecting edges are assumed to be $v_1v_2, v_2v_3, v_3v_4, v_4v_1$

        * to model, represent with a list of vertices with their coordinates and then a list of faces with their vertices
    - the **normal** of a polygon should face outward from the face
        * thus vertices of faces are typically ordered counter-clockwisee
    - considerations include:
        * closed / open
            - whether the last closing edge is inferred
        * wireframe / filled
            - generally only consider wireframe polygons (discretized)
        * planar / non-planar:
            - non-planar polygons span multiple planes
            - thus the normal points in different directions on the same polygon
        * convex / concave:
            - concave polygons have internal angles that are greater than 180°
            - thus the normal points in the wrong direction at some concave vertices
        * simple / non-simple
            - non-simple polygons intersect on themselves
- if we deal with only triangles, we satisfy many desired polygonal considerations:
    - simple, convex, and planar
    - modern GPUs can render 100 million triangles per second
- how can we check if a point is inside a polygon?
    - a point is *inside* a convex polygon if it lies to the *left* of all the directed edges
        * in a convex polygon, the directed edges point counter-clockwise and the normals of two consecutive edges are always consistent
    - for concave polygons, these useful properties do not hold
        * want to break up concave polygons into convex ones

Transformations

- **translations** can be written as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} T_x \\ T_y \end{bmatrix} + \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} T_x + x \\ T_y + y \end{bmatrix}$$

- **scalings** can be written as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} S_x x \\ S_y y \end{bmatrix}$$

  - note that scaling happens with respect to the origin:
    - scaling greater than one moves away from the origin, while scaling less than one moves towards the origin
    - negative scales flip across the axis
- **rotations** can be written as the following, where θ is the angle of rotation and r is the distance of the initial point from the origin:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

  - rotations also happen with respect to the origin
  - counter clockwise rotations are positive, while clockwise rotations are negative
- **shears** can be written as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + ay \\ y \end{bmatrix}$$

  - this is a horizontal shear
- transforming lines and polygons:
  - if we can *preserve* the linearity of the line, we only have to transform the endpoints of the line or polygon instead of all the contained or connected points
  - affine transformations on endpoints guarantee that straight lines will remain straight lines after the transformation!
    - ie. affine transformations preserve affine combinations eg. line segment interpolations
  - affine transformations also preserve planarity, parallelism, and relative ratios of edge lengths

- these transformations can be *stacked* by appending matrices together:
  - eg. $M_R \times M_S \times M_S \times M_R \times ... \times M_P = M_T \times M_P$
  - multiplying onto the left, AKA post-multiplication:
    - note that order matters!
    - except for special cases, such as pure transformations of a single type being commutative
      - eg. pure translations, pure scalings, pure rotation about an axis
  - can create a single transformation matrix out of the individual transformations
    - to all points, apply the single transformation matrix
  - however, translation is a matrix addition rather than a matrix multiplication:
    - cannot coalesce into a single transformation matrix!
    - however, with homogeneous representation, we can bypass this issue and treat translation as matrix multiplication
- homogeneous transformations:

$$T = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + T_x \\ y + T_y \\ 1 \end{bmatrix}$$

$$S = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} S_x x \\ S_y y \\ 1 \end{bmatrix}$$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ 1 \end{bmatrix}$$

$$Sh_x = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + ay \\ y \\ 1 \end{bmatrix}$$

- 3D transformation matrices:

$$T = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Sh_x = \begin{bmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  – note that this rotation is happening *around* the z-axis, since the z-coordinates are unchanged
  – note that this shear is a purely horizontal shear along the x-axis

- the **general rotation matrix (GRM)** is a shortcut rotation matrix, given the orthonormal unit vectors of the rotated basis, say i, j, k, to rotate them *back* onto the normal axes:

$$GRM = \begin{bmatrix} i_x & i_y & i_z & 0 \\ j_x & j_y & j_z & 0 \\ k_x & k_y & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  – looking at the 3×3 submatrix in the upper left, the first row is exactly i, the second row is exactly j, and the third row is exactly k
    * these are the θ projections used in the rotation transformation
  – to inverse, since orthornormal, we can just take the transpose, where the basis vectors are column vectors
    * this inverse is exactly how to transform normal axes *into* the rotated basis ie. change of basis transformation

- consider the 3×3 submatrix in the upper left of each of these 3D transformations:
  – each of the rows in this submatrix can be taken as a vector, and the dot product of any of these two rows in the submatrix equals 0

- an affine transformation has this property, where the upper left 3×3 submatrix is orthogonal
- translations and rotations are **rigid body transformations** since the lines, angles, and distances between points do not change:
  - the upper left 3×3 submatrix is orthonormal ie. vectors are orthogonal and unit vectors
  - note that scaling and shears are *not* rigid body transformations, since the vectors are not normalized (orthogonal, but *not* orthonormal)
  - for orthonormal matrices, $A^{-1} = A^T$

- rotations around the other axes:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- shear along the x and y-axes:

$$\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - the z-axis is locked, and shearing occurs on the other two axes

Inverses

- inverse transformations:

$$T^{-1} = \begin{bmatrix} 1 & 0 & -T_x \\ 0 & 1 & -T_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x - T_x \\ y - T_y \\ 1 \end{bmatrix}$$

$$S^{-1} = \begin{bmatrix} \frac{1}{S_x} & 0 & 0 \\ 0 & \frac{1}{S_y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{S_x} \\ \frac{y}{S_y} \\ 1 \end{bmatrix}$$

$$R^{-1}(\theta) = R(-\theta)$$

$$Sh_x^{-1}(a) = Sh_x(-a)$$

Combining Transformations

- how can we rotate by θ around an arbitrary point (x_R, x_Y), instead of just the origin?
  $$M = T(x_R, y_R)R(\theta)T(-x_R, -y_R)$$
  1. translate reference point to the origin
  2. rotate
  3. translate reference point back to its original location
- how can we scale an object by k in place?
  $$M = T(x_R, y_R)S(k)T(-x_R, -y_R)$$
  - normally, scaling only happens with respect to the origin:
    * the object will move away from its original location if it is not centered
    * we want to scale the object while retaining its position
  1. translate reference point (eg. bottom left corner of object) to the origin
  2. scale
  3. translate reference point back to its original location
- how to find a point's new coordinates after a change of basis ie. frame of reference?
  - given the basis vectors defining the new basis, simply apply the inverse transformation in the change of basis to the point to get new coordinates
  - eg. if we have a new frame of reference with origin $(6, 2)$, we can translate all points by $(-6, -2)$ to get new coordinates
    * similarly for stacked transforms eg. rotated and translated basis, have to inverse translate and then inverse rotate
- how to rotate a point by θ around a vector (with base point P_R)?
  $$M = T(P_R)R_y(-\phi_y)R_z(-\phi_z)R_x(\theta)R_z(\phi_z)R_y(\phi_y)T(-P_R)$$
  1. first, we need to align the vector to an axis
     - move vector base to origin, then rotate other axes so that the vector lies on some axis eg. x-axis
  2. rotate around the aligned axis
  3. move vector to original location
     - rotate other axes, and then translate back

Rendering Pipeline

World and Camera

- we need to define a **camera** or reference frame through which to view our world:
  - we can build a camera coordinate system that can be represented by a matrix
  - requires an **eye vector** to represent the eye direction, as well as a **top vector** to describe the tilt of the camera
  - three orthogonal vectors can be generated as follows:
    1. eye vector is the first vector
       * can also be represented by an **eye point** and **reference point**
    2. then, we need a vector orthogonal to the plane of the eye and top vectors, so we can take the cross product
    3. take the cross product of the first two bases
- camera basis vectors:

$$
\begin{aligned}
k &= \frac{P_{ref} - P_{eye}}{|P_{ref} - P_{eye}|} \\
i &= \frac{v_{up} \times k}{|v_{up} \times k|} \\
j &= k \times i
\end{aligned}
$$

- general graphics pipeline:
  1. first, we want to place various models into our **worldspace**:
     - we perform various transformations such as translations, rotations, etc.
     - these are collapsed into a single transformation matrix that is applied to all vertices
  2. then, to view this world through a camera ie. eye view:
     - we need to adjust all the coordinates into the camera's frame of reference
       * ie. moving eye view onto the origin
     - need to translate the eye point back to the origin, and rotate the camera bases
       * essentially, eye is at origin, looking down the z-axis, and head is upright

    - however, the x-axis from the eye's *point of view* has been flipped!
      - points left instead of right
      - we need to mirror across the yz-plane
    - use the following transformation matrix:

$$M = Mirror_x \times GRM(i, j, k) \times T(-P_{eye})$$

  - note that we can combine these matrices from different stages together!
    - transformation matrix (TM), eye matrix (EM), and an upcoming projection matrix (PM) can be pre-multiplied to avoid unnecessary matrix multiplies

- the mirror matrix on the yz-plane is as follows:

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

Projections

- now that everything is in eye space, we still have to perform something called a **projection**:
  - need to take a 3D object in space, and capture it *onto* a plane to display to the user
  - draw projection lines from a center of projection to the vertices of the object in space
    - the intersection of these projected vertices to the projection plane forms the projection image
  - two types of projections:
    - **parallel projection**, where the eye is placed at infinity away
      - view volume is a parallelepiped
    - **perspective projection**, where the eye's location gives a sense of depth to the projection:
      - view volume is more of a truncated pyramid, with **clipping planes** of different sizes that cut off the viewing volume
      - a viewing angle in x and y determine the slopes of this pyramid
    - having a front clipping plane prevents division by zero errors
      - during projection, usually divide by distance from eye
- in a parallel projection, all we have to do is throw away the z-coordinate:

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

- the view volume is a parallelepiped of certain dimension
  - any vertices lying outside of the box should be clipped and not rendered
- we can create a unit or normalized **parallel canonical view volume** that extends between -1 and 1 in the xy-plane and from 0 to 1 in the z-direction
- we can map an arbitrary view volume into a canonical one with the following normalized matrix:

$$Normal_M = \begin{bmatrix} \frac{2}{W} & 0 & 0 & 0 \\ 0 & \frac{2}{H} & 0 & 0 \\ 0 & 0 & \frac{1}{F-N} & \frac{-N}{F-N} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - given width, height, far distance, and near distance of view volume

- in a perspective projection, we need to take into account where the projection plane lies between the eye and object position, say a distance d from the eye:

$$Perspective_M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

  - simply use similar triangles in a ratio, we know that $\frac{x'}{d} = \frac{x}{z}$ ie. $x' = \frac{x}{z}d$
  - when applied to an arbitrary point, after normalization, the projected point is as desired:

$$Perspective_M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ \frac{z}{d} \end{bmatrix} = \begin{bmatrix} \frac{x}{z}d \\ \frac{y}{z}d \\ d \\ 1 \end{bmatrix}$$

    - z-coordinates are lost, collapsed into a constant d

- how can we handle non-square projections of other aspect ratios?
  - given parameters aspect ratio $A_R = \frac{W}{H}$ and half angle of view in the x-axis $\theta_x = \theta$
    - note that $\tan\theta = \frac{W}{2d}$, where d is the distance of the projection screen from the eye
  - we can normalize our previous x' to a canonical view volume by dividing by $\frac{W}{2}$ to get $x' = \frac{x}{z}\frac{2d}{W}$, and replace d with θ
  - similarly, we can do the same for the y-axis and use the aspect ratio to remove H and get the following:

$$x' = \frac{x}{z\tan\theta}$$
$$y' = \frac{y}{z}\frac{2d}{H} = \frac{yA_R}{z\tan\theta}$$

        * this constrains our new axes between $[-1, 1]$, as in the canonical view volume
        * note that we are indeed dividing by the distance from the eye through z
- the perspective projection matrix for any aspect ratio is as follows:

$$Perspective_{M_{AR}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & A_R & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \tan\theta & 0 \end{bmatrix}$$

    - when applied to an arbitrary point, after normalization, the projected point is as desired:

$$Perspective_{M_{AR}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ A_R y \\ z \\ z\tan\theta \end{bmatrix} = \begin{bmatrix} \frac{x}{z\tan\theta} \\ \frac{A_R y}{z\tan\theta} \\ \frac{1}{\tan\theta} \\ 1 \end{bmatrix}$$

    - however, our z values are now constant, and the depth data is lost for future calculations
- importantly, note that this perspective division step, where we normalize our points, cannot be encapsulated in a 4×4 matrix
    - transformation, eye, and projection matrices can be combined together
    - but perspective division has to be done separately, before the window-to-viewport mapping step
- to fix the loss of our depth during perspective division, we could recover our z values as follows:

$$z' = A + \frac{B}{z}$$
$$A = \frac{F}{F - N}$$
$$B = -\frac{NF}{F - N}$$

    - given the distance of the near and far planes, N and F
    - forms a system of equations where $z' = 0$ if $z = N$, and $z' = 1$ if $z = F$
    - the full normalized perspective projection matrix is as follows, which maps directly into a parallel canonical view volume that expresses height in z from 0 to 1 (rather than a flat plane with no depth):

$$Perspective_{M_{AR}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & A_R & 0 & 0 \\ 0 & 0 & A\tan\theta & B\tan\theta \\ 0 & 0 & \tan\theta & 0 \end{bmatrix}$$

    * note that after multiplying with the projection matrix, perspective divison ie. normalization must be done

Window to Viewport Mapping

- the final rendering stage is to map our viewing window to a viewport of arbitrary size:

$$M = T(V_L, V_B)S(\frac{V_R - V_L}{W_R - W_L}, \frac{V_T - V_B}{W_T - W_B})T(-W_L, -W_B)$$

  - requires viewport left, right, bottom, and top or V_L, V_R, V_B, V_T, respectively
    * as well as the same values for the window, if not normalized to the canonical window

Hidden Surface Removal

- through **hidden surface removal**, we remove parts of certain objects that are obscured by other objects:
  - saves on computations for graphics that would never be seen on the screen
  - basic operations:
    1. establish priorities among objects
    2. collect overlapping elements and resolve visibility using the priorities
  - different algorithm types:
    - **image space** algorithms perform both operations at pixel resolution
    - **list-priority** algorithms perform (1) at object resolution and (2) at pixel resolution
    - **object / world / eye space** algorithms perform both operations at object resolution
  - different surface removal algorithms may not support all object types
    - polymesh, free form, volume, CSG, implicit
  - algorithm considerations include flexibility, memory, speed

Back Face Culling

- in **back face culling**, we consider the normals of faces:
  - only works for polymesh objects
  - as soon as possible, we should throw away **back faces** in which the normal faces away from the eye
    - for any polygon, on average, half of the faces are not visible
  - to determine back faces in world space:
    - take the dot product of the normal and the vector created from any point on the face towards the eye
    - if dot product is positive, the face is a front face, otherwise it is a back face
      - when zero, the face exactly “grazes” the eyesight, and we can still ignore it
  - in eye space, we can perform the same calculation with the eye placed at the origin
    - note that the calculation is simplified since we can just take a dot product of the point and normal, and if negative, the face is a front face
  - in projection space (after perspective division), the eye is now placed at

negative infinity after projecting objects onto a flat screen:
- calculation is even simpler, front faces have a negative z-component
- all other faces are back faces

Painter's Algorithm

- the **painter's algorithm** is an object space algorithm that runs as follows:
    1. sort all polygons by z-depth
    2. scan-convert polygons in back to front order
        - ie. take polygons through the pipeline one at a time, and override the pixels on the screen accordingly
    - cannot handle cyclic or intersecting polygons!
        - need to break up such polygons into the simpler case

Z-Buffer Algorithm

- the **z-buffer algorithm** is an image space algorithm that runs as follows:
    - maintain a depth and color buffer containing information on each pixel
        - initialized to infinite distance and desired background color
    1. for each pixel covered by a polygon:
        - calculate depth z for the polygon at the pixel (x, y) using plane equations
        - if z is less than the entry in the depth buffer, update the depth and color buffers accordingly with the new z and color of the polygon
    - note that order of polygon processing doesn't matter, since we are proceeding pixel by pixel
- properties:
    - image precision algorithm
    - easy to implement in software and hardware
    - polygons scan-converted into framebuffer in random order
        - no pre-sorting of polygons is necessary
    - *pros*:
        - handles cyclic and intersecting polygons
        - extends to various faces other than polygons, all we need is a depth calculation
        - simplicity and ease of implementation
        - can be modified to reduce memory requirements
        - can be extended to A-buffer to reduce aliasing
        - theoretically can handle any number of polygons

  - *cons*:
    - buffer memory requirements, not much of an issue in modern PCs
    - aliasing issues ie. smooth edges or lines become jagged due to the pixel granularity
      - depends on screen resolution
    - complexity depends on polygons' projection area on the screen
    - difficult to handle transparency, since blending of colors must be done in a specific order
- **area weighted averaging** is one technique to perform anti-aliasing:
  - the overall color of a pixel is a weighted average combination of the polygon and background colors
    - weighted by area of the pixel covered by the polygon vs. background
  - resultant color is a blend of colors, creates a softer transition on the polygon edges
  - pixel area calculations are expensive operations
- the **scanline z-buffer algorithm** is an adaptation on the algorithm that reduces the memory usage:
  - proceed one scanline at a time
    - requires all polygons overlapping the current scanline, rather than any single polygon at a time
  1. for each scanline at height y:
  2. for each polygon which intersects scanline:
     - scan convert for specific scanline and determine span segments
  3. for each pixel in span segment at location x:
     - calculate depth z for polygon at pixel (x, y)
     - if z is less than the entry in the depth buffer, update the depth and color buffers
  - *pros*:
    - less memory requirement than z-buffer, proportional to window width instead of area
  - *cons*:
    - multiple passes through polygon database
- z-buffer speedup considerations:
  - bounding box testing:
    - associate y_{min}, y_{max} for each face (step 2)
    - calculate left and right ends ie. x-intercepts with scanlines (step 3)
  - can subdivide the entire screen into spaces to parallelize
  - can subdivide hierarchically (some screen regions more busy than others) to parallelize

– incremental depth calculation along a scanline:

$$\begin{aligned}
Ax + By + Cz + D &= 0 \\
z_{x,y} &= -\frac{Ax + By + D}{C} \\
z_{x+1,y} &= -\frac{A(x+1) + By + D}{C} \\
&= z_{x,y} - \frac{A}{C}
\end{aligned}$$

Lighting

- in global illumination, reflected light reaches the eye to create an image
  - light energy is bounced around or scattered between various surfaces of objects
    - this scattering is called **radiosity**
  - **reflection** of objects on other surfaces is a different phenomenon
  - in graphics, we instead trace rays of light from the eye onto objects in order to color pixels:
    - rather than illuminating unecessary pixels from the light source that may not eventually reach the eye
    - in addition, fully simulating radiosity is an enormously expensive calculation
      - instead, we approximate illumination through different types of lighting
  - types of lights:
    1. ambient light
    2. spot light
    3. directional light with no location eg. sun
    4. spotlight with location and direction:
       - has some spotlight angle β
       - only points hit by the spotlight within β are illuminated
       - shoot a ray from the spotlight location to the point, and illuminate if $\alpha < \beta$ ie. $L \cdot D > \cos\beta$
- lighting depends on various properties:
  - geometric:
    - position and orientation of object or eye
    - position, direction, point vs. spot vs. area of light
  - material:
    - color, reflectivity, shininess, translucency, bumpiness of object
    - color of light
    - filter or color blindness of eye
- lighting types include ambient, diffuse, and specular lighting
- **ambient** lighting is produced from a background light:
  - unrealistic, only outlines the silhouette of the object
  - good approximation of scattered light
  - does not depend on position or orientation of light, object, or eye
    - only object and light's material property

  - $L_A = k_a \cdot I_a$, ambient reflection coefficient times intensity of ambient light source

- **diffuse** lighting is produced from a point light source:
  - AKA Lambertian reflection for dull, matte surfaces
  - surfaces look equally bright from all directions, regardless of where the eye is placed
  - Lambert's Law states that the amount of light is reflected is proportional to the angle between the incident light and the normal:
    * $L_D = k_d I_p \cos\theta = k_d I_p (N \cdot L)$
    * diffuse reflection coefficient times intensity of point light source times dot product of normal and incident light
  - ignore negatives by "clamping" at zero
    * negatives indicate **self-occlusion**
  - for a light at sufficient distance, L is constant for the entire scene, and $N \cdot L$ is constant on a polygon
  - can attenuate the diffuse lighting depending on distance:

$$\begin{aligned} L_D &= f_{att} k_d I_p (N \cdot L) \\ f_{att} &= \frac{1}{d^2}, f_{att} = \frac{1}{c_1 + c_2 d + c_3 d^2} \\ f_{att} &= min(f_{att}, 1) \end{aligned}$$

    * in atmospheric attenuation, immerse object in "fog" past a certain distance by blending colors

- **specular** lighting is created off of "shiny" surfaces:
  - *does* depend on location of the eye
  - color is of the light source, not object
  - light not reflects *unequally* in different directions
  - $L_S = f_{att} k_s I_p \cos^n\theta = f_{att} k_s I_p (R \cdot V)^n$
    * attenuation factor times specular reflection coefficient times intensity of point light source times dot product raised to specular reflection exponent
  - n is the material's specular reflection exponent, higher indicates a more shiny surface
    * ie. metallic surfaces are closest to a perfect reflector which have a very bright, concentrated reflection
  - R is the reflection vector, V is the view vector from the point to the eye:
    * R is equally reflected off of N by the angle between N and L
    * $R = 2(L \cdot N)N - L$
    * is there a way we can approximate $R \cdot V$ with something else?

        · use a halfway vector H that is halfway between L and V, and use $N \cdot H$ instead
        · this is Blinn-Phong illumination, faster but less accurate
        · a faster alternative also exists for calculating t^n
  - exponentiating a term that is less than 1 draws it closer to 0:
    * higher exponent indicates a smaller region where the point light reflection is aligned with the viewer, leading to a smaller shiny spot
    * should clamp the cos to zero

- overall equation for illuminated color at a point:

$$I_\lambda = k_a I_a O_{d\lambda} + f_{att} k_d I_p (N \cdot L) O_{d\lambda} + f_{att} k_s I_p (R \cdot V)^n$$

  - I_λ is overall illuminated color (with three parts RGB), $O_{d\lambda}$ is the object's diffuse color
  - there may be an additional emmisive term k_e if the object itself creates light
  - to handle multiple point light sources, just add up the diffuse and specular terms for each point light
  - k, O, I, n are all material properties, while remaining variables are geometric properties
  - note that this illumination equation does not take into account shadows from other objects
    * requires additional energy conservation equations involving ray tracing and radiosity

Shading

- now that we know how to light certain points, how do we determine which points on our world to illuminate?
  - infinite possible points to light up
  - polygons have a color and normal
    - while vertices have position, color, normal, as well as texture coordinates
  - typically, illumination is done in world space, and interpolation is done in screen space
- in **flat / faceted shading**, each polygon is shaded and illuminated uniformly:
  1. use N of polygon
  2. find I_λ at center or any vertex of polygon
  3. apply that same color to all points inside polygon
  - means that:
    - normal is constant across polygon
    - light is at infinity so $N \cdot L$ is constant across polygon
    - viewer is at infinity so $N \cdot V$ is constant across polygon
  - no interpolation is done
  - *pros*:
    - simple
  - *cons*:
    - drastic jump in illumination at the edges
      - to the human eye, the jump is perceived as even stronger
- in **Gouraud smooth shading**, we illuminate each vertex:
  - AKA intensity interpolation or color interpolation shading
  - how to find the normal for a vertex?
    - average the normals of the neighboring polygons
    - to emphasize an edge, additionally bias the average with that polygon normal
  1. find color at each vertex and illuminate in world space
  2. in screen space, interpolate ie. smoothen vertex colors across the polygon face
  - *pros*:
    - no more jump in illumination on polygon edges
  - *cons*:
    - for large polygons, we only considered the vertices, when the faces can have important illumination consequences
      - eg. accidentally skipping specular highlights in the middle of a polygon or unnecessarily interpolating specular highlights from a vertex

  - note that the polygonal geometry is not being smoothed out, only the coloring
    - though smooth shading helps to create an illusion of a smoother object
- how to solve some of the Gouraud shading drawbacks?
  - illuminate *every* pixel inside a polygon
    - this is **Phong shading**
  - but we have to illuminate in world space (but no notion of pixels in world space), instead of screen space (no notion of lights, object colors, eye):
    - need to perform an inverse transformation while in screen space back to world space to color each pixel
    - interpolate the normals at vertices in screen space to get a normal to use for coloring

Interpolations

- given a polygon with colored vertices, how can we interpolate and smoothen out the colors across the polygon?
  - in 1D, we can perform linear interpolation using an affine combination of colors ie. $C = (1-t)C_1 + tC_2$
  - in 2D, we want to perform **bilinear interpolation**
    - an alternative interpolation technique is **barycentric interpolation**, which uses percentage areas from vertices
  - can be used to generally interpolate colors, depth, normals, texture coordinates, etc.
    - any values given at vertices can be interpolated
- bilinear interpolation algorithm for a certain point:
  1. draw a horizontal scanline through the point
  2. perform two linear interpolations for color along the left and right intersecting polygon edges of the scanline
     - gives us x_a, y_a, λ_a and x_b, y_b, λ_b
  3. perform a final linear interpolation for the point on the left and right colored intersecting points
  - but this algorithm will use different vertices to interpolate depending on how the object rotates
    - however, when using triangles, the same vertices are always utilized
  - in Gouraud shading, when trying to interpolate colors, we will also run into issues where specular lighting details are lost

Mappings

- through mappings, we adjust the shading or sometimes positions of parts of an object based on some kind of image map
- **texture / pattern mapping** is the process of mapping a digitized image onto a poly face:
  - individual elements of a texture are **texels** rather than pixels
    - usually mapped onto an s, t plane with $0 \leq s, t \leq 1$
  - given u, v coordinates for a texture, we want to use the texel color as the diffuse color to color the pixel
  - if we sample the texture colors as a low rate, we may miss out on high-frequency patterns in the texture
    - to fix this aliasing issue, we can sample at a higher resolution, or perform area averaging
  - can also apply multiple textures on the same object and blend them
- displacement and bump mappings distort the 3D appearance of the object given a map:
  - these maps are greyscale images that tell points how far to move
  - in **displacement mapping**, we displace the actual point on the surface of the object, and change the object's geometry:
    - $P' = P + BN$ ie. bump along the normal
    - points actually move, requires additional hidden surface removal
  - in **bump mapping**, we distort the normal to simulate a bump without altering the object's geometry:
    - $N' = \frac{B_u(N \times P_t) - B_v(N \times P_s)}{|N|}$
    - B_u, B_v, P_s, P_t are all partial derivatives
- with **environmental / reflection mapping**, we use polar coordinates of the reflected ray to map:
  - quick way to graphically simulate reflections
    - ray tracing allows for true reflections
  - map usually defined on faces of a box or sphere
  - add a reflection term $k_r I_r$ to the illumination equation
- with **procedural mapping**, we use some formula to procedurally adjust point displacement, color, etc.
  - eg. create impression of water waves without full liquid simulations

Appendix

Linear Algebra Review

- **points** have a location, but no size, shape, or direction
  - lie on a coordinate plane
- **vectors** have a direction and length, but no location:
  - can define a vector along two basis vectors (in 2D)
  - vectors $v_i, \dots, v_m$ are **linearly independent** if $a_1 v_1 + \dots + a_m v_m = 0$ iff. $a_i = 0$
    * ie. no projection of one vector on any of the others
  - **linear dependent** vectors are scalar multiples of each other
- a difference between two points is a vector $v = Q - P$
  - similarly, a base point plus bector offset is another point $Q = P + v$
- the **homogeneous representation** for points and vectors allows us to distinguish between the two:
  - a point is represented as $\begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}$
  - a vector is represented as $\begin{bmatrix} V_x \\ V_y \\ 0 \end{bmatrix}$
  - similarly for 3D, we have a 4th element to distinguish the two
  - we can now define vectors and points in matrix multiplication:

$$v = \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [\beta_1 \;\; \beta_2 \;\; \beta_3 \;\; 0] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

$$\begin{aligned} P &= P_0 + v = P_0 + \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 \\ &= [\alpha_1 \;\; \alpha_2 \;\; \alpha_3 \;\; 1] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} \end{aligned}$$

- more operations in homogeneous representation:

$$v+w=\begin{bmatrix}v_1\\v_2\\v_3\\0\end{bmatrix}+\begin{bmatrix}w_1\\w_2\\w_3\\0\end{bmatrix}=\begin{bmatrix}v_1+w_1\\v_2+w_2\\v_3+w_3\\0\end{bmatrix}$$

$$av+bw=a\begin{bmatrix}v_1\\v_2\\v_3\\0\end{bmatrix}+b\begin{bmatrix}w_1\\w_2\\w_3\\0\end{bmatrix}=\begin{bmatrix}av_1+bw_1\\av_2+bw_2\\av_3+bw_3\\0\end{bmatrix}$$

$$P+v=\begin{bmatrix}p_1\\p_2\\p_3\\1\end{bmatrix}+\begin{bmatrix}v_1\\v_2\\v_3\\0\end{bmatrix}=\begin{bmatrix}p_1+v_1\\p_2+v_2\\p_3+v_3\\1\end{bmatrix}$$

$$P-Q=\begin{bmatrix}p_1\\p_2\\p_3\\1\end{bmatrix}-\begin{bmatrix}q_1\\q_2\\q_3\\1\end{bmatrix}=\begin{bmatrix}p_1-q_1\\p_2-q_2\\p_3-q_3\\0\end{bmatrix}$$

- linear combination in homogeneous representation:

$$aP+bQ=a\begin{bmatrix}p_1\\p_2\\p_3\\1\end{bmatrix}+b\begin{bmatrix}q_1\\q_2\\q_3\\1\end{bmatrix}=\begin{bmatrix}ap_1+bq_1\\ap_2+bq_2\\ap_3+bq_3\\a+b\end{bmatrix}$$

  - if affine, $a+b=1$, and the combination creates a point
  - if $a+b=0$, the combination creates a vector
  - otherwise, we can normalize the result so that the last element is 1
    * combination still creates a point

- a **vector space** is a space defined with respect to certain **basis vectors**:
  - eg. in 2D, we need two bases in order to define any unique vector
  - the magnitudes in the direction of the basis vectors, added together, defines any unique vector
    * eg. $v=v_x\overrightarrow{v_1}+v_y\overrightarrow{v_2}=v_xi+v_yj$
  - basis vectors do not necessarily have to be orthogonal, or even unit magnitude:
    * however, cannot be on the same line ie. linear dependent
    * good practice to have unit basis vectors to only specify direction

- a **generator set** is a set of vectors that generate a vector space:
  - for a vector space $\mathbb{R}^n$ we need minimum n vectors to generate all vectors

  - a generator set with minimum size is called a **basis** for the specified vector space
    - basis is purely defined by vectors, and creates a vector space that only supports vectors

- a **frame** has a point of origin along with a basis, and creates an **affine space** that supports vectors and points
- the **dot product** of two vectors is defined as $v_1 \cdot v_2 = |v_1||v_2| \cos\theta$:
  - alternatively $v_{1_x} v_{2_x} + v_{1_y} v_{2_y}$
  - a scalar value
  - when the dot product is 0, the vectors are orthogonal
  - when the dot product is negative, the angle is greater than 90 degrees
  - when the dot product is positive, the angle is less than 90 degrees
  - $|u| cos\theta$ gives the projection of vector u on in the direction of v
- the **cross product** of two vectors gives a vector:

$$v = v_1 \times v_2 = det \begin{bmatrix} i & j & k \\ v_{1_x} & v_{1_y} & v_{1_z} \\ v_{2_x} & v_{2_y} & v_{2_z} \end{bmatrix}$$

  - resultant vector is perpendicular to the plane of the two vectors, pointing as defined by the right-hand rule
  - $|a \times b| = |a||b| \sin\theta$
  - not commutative
- **polygons** can be defined as a set of directed edges or connected vectors
  - the vectors can be calculated as the differences between the connected points
- **lines** can be written in parametric form as $P = P_1 + \alpha\vec{d}$:
  - where $d = P_2 - P_1$ and P_1, P_2 are the endpoints of the directed line pointing towards P_2
  - equivalently, $P = (1 - \alpha)P_1 + \alpha P_2$
  - traces ie. interpolates a line between the two endpoints that is infinite in both directions
- adding points has no meaning, but by linearly combining them with additional constraints, we can interpolate useful constructs:
  - $P = \alpha_1 P_1 + \alpha_2 P_2$ is a **linear combination**
  - with the condition $\alpha_1 + \alpha_2 = 1$, the parametric equation becomes an **affine linear combination**

    - represents a point lying on the line passing through P_1, P_2 that is infinite in both directions
  - with the *additional* condition that $\alpha_i \geq 0$, the parametric equation becomes a **convex linear combination**:
    - represents a point on the line segment between P_1, P_2
    - note convex necessitates affine
  - if we only have the constraint $\alpha_i \geq 0$ (not affine), we have a **ray** that is infinite in one direction only

- consider defining a polygon in terms of parameteric form as $P = \alpha_1 P_1 + ... + \alpha_n P_n$:
  - without any constraints, we cannot guarantee that the interpolated points are on the same plane as the polygon i.e. only a linear combination
  - with constraint $\sum \alpha_i = 1$, we have an affine combination, and the interpolated points will lie on the same plane as the polygon
  - with additional constraint $\alpha_i \geq 0$, we have a convex combination, and the interpolated points will lie within the convex hull of the polygon

- the **convex hull** can be imagined as taking a string around pegs at each corner of the polygon:
  - technically, the smallest convex polygon that contains all the points of the actual polygon
  - can be larger than the actual drawn polygon

Table 1: Summary of Scalar, Point, and Vector Operations ($*$ affine only)

Operands	Add	Subtract	Multiply
point-point	$P = a_1 * P_1 + a_2 * P_2 \ (*)$	$v = P_2 - P_1$	
vector-vector	$v = v_1 + v_2$	$v = v_1 - v_2$	
scalar-point			$P = s * P_1 \ (*)$
scalar-vector			$v = s * v_1$
point-vector	$P_2 = P_1 + v_1$	$P_2 = P_1 - v_1$	

Graphics Tips & Tricks

- to transform lines:
  1. described by 2 end points
     - if we are performing an affine transformation, we can simply transform the end points and connect the line since the points will re-

main collinear

2. described by equation $y = mx + b$:
   - find two points, transform them, and connect the line
   - for translations, we can simply adjust b
   - for rotations, we can simply adjust m

- to transform planes:
  1. described by 3 non-collinear points
     - if we are performing an affine transformation, transform the points and draw the new plane
  2. described by plane equation $Ax + By + Cz + D = 0$:
     - or $A(x - x_i) + B(y - y_i) + C(z - z_i) = 0$
     - the normal is (A, B, C)
     - if M_{point} is the matrix to transform a point, $M_{normal} = (M_{point}^T)^{-1}$
       * for rigid body transformations, $M_{normal} = M_{point}$
- point in polygon test:
  1. if convex, check if point lies to the left of every edge
  2. extend a semi-infinite ray from the point:
     - if there are an odd number of intersections with the polygon, the point is inside, else outside
     - also works for concave polygons
  3. perform angle summation from the point to each pair of vertices:
     - if the sum of subtended angles is $360°$, the point is inside, else outside
     - also works for concave polygons
- on-line test:

$$\frac{x - x_1}{y - y_1} = \frac{x_2 - x_1}{y_2 - y_1}$$
$$T(P) = (x - x_1)(y_2 - y_1) - (x_2 - x_1)(y - y_1)$$

  - if the test value is 0, the point P lies on the line connected through P_1, P_2
  - if positive, P is on the right
  - if negative, P is on the left
- edge intersection test:
  - given two edges represented by 4 points, can we determine if the edges intersect?
  - check if one endpoint of an edge is to the left of the other edge, and the other endpoint is to the right of that edge
    * and vice versa with the other edge's endpoints
  - ie. the products of the on-line tests for the endpoints against the other edge should be negative

- collinearity test:

$$\begin{aligned} t &= |P_1P| \sin\theta \\ &= \frac{|P_1P||P_1P_2| \sin\theta}{|P_1P_2|} \\ &= \frac{|P_1P \times P_1P_2|}{|P_1P_2|} \end{aligned}$$

  - t represents the shortest distance from a point P to an edge P_1, P_2
  - θ is the angle between P_1, P_2 and P_1, P
  - if t is sufficiently small, we can say the point is collinear
- calculating a normal vector:
  1. give 3 consecutive convex vertices, simply find the cross product
  2. use summation method, which works for convex and concave polygons:

$$(\sum(y_i - y_j)(z_i + z_j), \sum(z_i - z_j)(x_i + x_j), \sum(x_i - x_j)(y_i + y_j))$$

- transformation matrices:

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - the upper-left 3×3 matrix defines rotations, shears, and scalings
  - (m_{14}, m_{24}, m_{34}) defines translations
- orthogonal transformation matrices:
  - eg. translations and rotations
  - for the upper-left 3×3 matrix in a 4×4 transformation matrix:
    * each row is a unit vector, and each row is orthogonal to the others
    * can be thought of rotating these vectors to align with the xyz-axis
  - determinant is 1
  - $M^{-1} = M^T$
  - if orthonormal as well, preserves angles and lengths ie. is a rigid body transformation