Contents

1 Introduction

2 Misc Math

1. (trig identities)
2. **(dot product)**
$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\phi \qquad proj_{\mathbf{a}}(\mathbf{b}) = \|\mathbf{a}\| \cos(\phi) = \frac{a \cdot b}{\|\mathbf{b}\|}$$
3. **(cross product)**
$$\mathbf{a} \times \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \sin(\phi)\mathbf{n} = \det \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix}$$
where $\mathbf{n}$ is unit vector perpendicular to plane formed by $\mathbf{a}, \mathbf{b}$, determined by the right hand rule
4. **(coordinate frames)**
   - the **global coordinate system** is one formed by cartesian canonical orthonormal basis and the canonical origin that is not stored explicitly.
   - Other coordinate system is called a **frame of reference** or **coordinate frame** are stored explicitly (the origin, and 3 orthonormal vectors as a function of the canonical basis) and is called a **local coordinate system**
   - If $\mathbf{b}$ is in canonical $\mathbf{x} - \mathbf{y} - \mathbf{z}$ coordinate and want to transform to a local $\mathbf{u} - \mathbf{v} - \mathbf{w}$ coordinate. then the transformed vecter can be written as
$$\sum_{\mathbf{q} \in \{\mathbf{u},\mathbf{v},\mathbf{w}\}} \langle \mathbf{q}, \mathbf{b} \rangle \, \mathbf{q}$$
5. **(construct basis)** by GramSchmidt process
6. (linear interpolation) over 2 points or a set of points where the function is piece-wise linear
7. (triangles)
   - barycentric coordinates is a nonorthogonal makes interpolation straight-forward over a triangle. Say $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ forms a triangle, then the vertices $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ act as basis vectors and form the barycentric coordinates. Any point $\mathbf{p}$ is
$$\mathbf{p}(\alpha, \beta, \gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \qquad \alpha + \beta + \gamma = 1$$
   (α, β, γ) is the barycentric coordinate of $\mathbf{p}$ w.r.t. $\mathbf{a}, \mathbf{b}, \mathbf{c}$. Given $\mathbf{p}$ in cartesian coordinate, we can find its baarycentric coordinate by solving
$$\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \quad \iff \quad \begin{pmatrix} x_b - x_a & x_c - x_a \\ y_b - y_a & y_c - y_a \end{pmatrix} \begin{pmatrix} \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} x_p - x_a \\ y_p - y_a \end{pmatrix}$$
   Geometrically, α, β, γ can be computed using triangle area
$$\alpha = A_\alpha / A \quad \beta = A_\beta / A \quad \gamma = A_\gamma / A$$
   Properties
   - can detect pointson edge/verties easily
   - mixes coordinates of vertices smoothly
   - can detect insidedness of points easily, i.e. $\alpha, \beta, \gamma > 0$

3 Raster Images

1. **(raster display)** show images as rectangular arrays of pixels.
2. **(raster images)** a 2D array that stores pixel value for each pixel that is device-independent
3. **(vector images)** description of shapes that is resolution independent but needs to be rasterized before display
4. **(point sample)** the reflectance, or fraction of light reflected as a function of position on a piece of paper

$$I(x, y) : R \subset \mathbb{R}^2 \to V$$

where V is the set of possible pixel values, i.e. $V = \mathbb{R}^+$ for grayscale images. Pixel in raster images are local average of the color of the image, called a point sample of the image

5. **(pixel values)** as tradeoff between memory and resolution. Lowered bits introduce artifacts such as clipping and banding
6. **(RGB color)**
   (a) (pixel coverage) α is the fraction of pixerl covered by the foreground layer.
   (b) (composite pixel) Let $\mathbf{c}_f, \mathbf{c}_b$ be foreground and background colors respectively, then $\mathbf{c} = \alpha\mathbf{c}_f + (1 - \alpha)\mathbf{c}_b$
   (c) **(alpha/transparency mask)** α values for all pixels, stored as a separate grayscale image; or stored as a 4th channel, called the alpha channel

4 Ray Tracing

1. **(rendering)** take a scene, a model, composed of many geometric objects arranged in 3D space and produce a 2D image that shows the objects as viewed from a particular viewpoint
   - **(object-order rendering)** for each object, update pixels that the object influences (gaphics pipeline)
   - **(image-order rendering)** for each pixel, set pixels based on the objects that influence it (ray tracing)
2. **(ray-tracing)** image-order algorithm for making renderings of 3D scenes
   (a) **(ray generation)** compute origin and direction of each pixel's viewing ray based on the camera geometry
   (b) **(ray intersection)** finds closest object intersecting the viewing ray
   (c) **(shading)** computes the pixel color based on the reseults of ray intersection
3. **(linear perspective)** 3D objects projected to image plane in such a way that stright lines in the scene become straight lines in the image
4. **(parallel projection)** 3D points are mapped to 2D by moving them along a projection direction until they hit the image plane
   - **(orthographic/oblique projection)** if image plane is perpendicular to view direction, the projection is orthogrpahic; otherwise, it is called oblique
5. **(camera frame)** let $\mathbf{e}$ be the viewpoint, and $\mathbf{u}, \mathbf{v}, \mathbf{w}$ be the three basis vectors. $\mathbf{u}$ points upward (from camera's view), $\mathbf{v}$ points upward, and $\mathbf{w}$ points backward. $-\mathbf{w}$ is called the view direction.
6. **(computing viewing ray)** represent ray with 3D parametric line

$$\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$$

idea is to find $\mathbf{e}, \mathbf{d}$

(a) **(orthographic view)** where **e** the viewpoint is placed on the image plane

- compute coordinate (u, v) of each pixel (i, j) on the image plane of size $(r - l) \times (t - b)$

$$u = l + (i + 0.5)\frac{r - l}{n_x}$$
$$v = b + (j + 0.5)\frac{t - b}{n_y}$$

- set ray's origin to be $\mathbf{e} + u\mathbf{u} + v\mathbf{v}$ and direction to be $-\mathbf{w}$

(b) **(perspective views)** project along lines that pass through a single point, the viewpoint, rather than along parallel lines. **e** the viewpoint is placed some distance d in front of **e**, call this distance the **image plane distance / focal length**

- compute coordinate (u, v) of each pixel (i, j) on the image plane using previous formula
- set ray's origin to be **e** and direction to be $-d\mathbf{w} + u\mathbf{u} + v\mathbf{v}$

from eye **e** to a point **s** on the image plane.

7. **(ray-sphere intersection)** Solve a quadratic formula satisfying the 2 condition, that the intersecting point is both on the ray and on the sphere.

8. **(ray-triangle intersection)** Want to find the first intersection between the ray $\mathbf{e} + t\mathbf{d}$ and a surface that occurs at a $t \in [t_0, t_1]$. Given parametric surfaces , we can solve for

$$\mathbf{e} + t\mathbf{d} = \mathbf{f}(u, v)$$

for 3 unknowns, u, u, v. If the surface is a plane in barycentric coordinate, then solve for

$$\mathbf{e} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

for some t, β, γ. The intersection is inside the triangle if and only if $\beta > 0$,$\gamma > 0$, and $\beta + \gamma < 1$. There is no solution if either the triangle is degenerate or the ray is parallel to the plane containing the triangle. This equation can be solved analytically with Cramer's rule

9. **(ray-polygon intersection)** Given a planar polygon with vertices $\{\mathbf{p}_1, \cdots, \mathbf{p}_m\}$ and surface normal **n**, we can compute intersection point between ray and plane containing polygon **p** with

$$(\mathbf{p} - \mathbf{p}_1) \cdot n = 0 \qquad t = \frac{(\mathbf{p}_1 - \mathbf{e}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

then we check if **p** is inside the polygon by sending 2D ray out from **p** and count the number of intersections between the ray and boundary of the polygon: if the number of intersections is odd, then the point is inside the polygon.

10. **(ray-scene-intersection)** To intersect a ray with a group of objects, i.e. the scene, simply intersect ray with the objects in the group and return the intersection with the smallest t value.

11. **(shading model)** is designed to capture the process of light reflection, whereby surfaces are illuminated by light sources and reflect part of the light to the camera

- (light direction) **l** is unit vector pointing toward light source
- (view direction) **v** is the unit vector pointing toward camera
- (surface normal) **n** is unit surface normal at point of reflection
- (properties of the surface), i.e. color, shininess

12. **(Lambertian Shading)** For each color channel, compute pixel color L with

$$L = k_d I \max(0, \mathbf{n} \cdot \mathbf{l})$$

where k_d is the diffuse coefficient or the surface color, and I is intensity of light source, and $n \cdot l = \cos(\theta)$ where θ is angle between surface normal and light source. Lambertian shading is **view independent**, i.e. color of the surface does not depend on the viewing direction **v**, leading to matte, chalky appearance. Higher k_d, more contrasty

13. **(Blinn-Phong Shading)** To account for shininess, or **specular reflection**, a **specular component** is added to the **diffuse component** to account for highlights. Blinn-Phong accounts for specular reflection by generating brightest reflection when $\mathbf{v}$ and $\mathbf{l}$ are symemtrically positioned across the surface normal, i.e. mirror reflection; the reflection decreases smoothly as the vectors move away from the mirror configuration. Idea is to compare bisector of $\mathbf{v}$ and $\mathbf{l}$ to the surface normal $\mathbf{n}$

$$L = k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p \quad \mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}$$

where k_s is the specular coefficient of the surface and p is Phong exponent where larger value indicate shinier/glossier surface

14. **(Ambient Shading)** To avoid rendering completely black pixels for surfaces that receive no illumination, add a constant illumination to surfaces, with no dependence on surface geometry

$$L = k_a I_a + k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

where k_a is surface specific ambient coefficient and I_a is the ambient light intensity.

15. **(multiple point lights)** the effect by more than one light source is simply the sum of the effects of the light sources individually by superposition. Hence

$$L = k_a I_a + \sum_{i=1}^{N} (k_d I_i \max(0, \mathbf{n} \cdot \mathbf{l}_i) + k_s I_i \max(0, \mathbf{n} \cdot \mathbf{h}_i)^p)$$

where $I_i, \mathbf{l}_i, \mathbf{h}_i$ are intensity, direction, and half vector of the i-th light source

16. **(shadows)** idea is surface is illuminated if nothing blocks the view of the light, i.e. the shadow ray $\mathbf{p} + t\mathbf{l}$ does not intersect any object in the scene.

17. **(ideal specular reflection)** or mirror reflection of a surface. the viewer sees the reflection of other objects instead of the highlights. The mirrored object's color is the color of ray at the surface in the reflection direction

5 Linear Algebra

1. **(eigenvalues and diagonalization)** If a matrix has eigenvectors, then we can find them by solving a quadratic equation

$$(\mathbf{A} - \lambda \mathbf{I})\mathbf{v} = 0$$

2. **(eigenvalue decomposition)** If $\mathbf{A}$ is symmetric, i.e. $\mathbf{A} = \mathbf{A}^T$, then we can decompose

$$\mathbf{A} = \mathbf{Q}\mathbf{D}\mathbf{Q}^T$$

where $\mathbf{Q}$ is an orthogonal matrix whose column are eigenvectors of $\mathbf{A}$ and $\mathbf{D}$ is a diagonal matrix whose diagonals are eigenvalues of $\mathbf{A}$

3. **(single value decomposition)** Any A can be decomposed to

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

where $\mathbf{U}, \mathbf{V}$ are orthogonal matrices, whose columns are left,right singular vectors of $\mathbf{A}$ and $\mathbf{S}$ is a diagonal matrix containing singular values of $\mathbf{A}$. Let $\mathbf{M} = \mathbf{A}\mathbf{A}^T$,

$$\mathbf{M} = \mathbf{A}\mathbf{A}^T = (\mathbf{U}\mathbf{S}\mathbf{V}^T)(\mathbf{U}\mathbf{S}\mathbf{V}^T)^T = \mathbf{U}\mathbf{S}^2\mathbf{U}^T$$

since $\mathbf{M}$ is symmetric so svd reduces to eigenvalue decomposition. Therefore singlular values for $\mathbf{A}$ are square roots of eigenvalues of $\mathbf{M}$ and singular vector for $\mathbf{A}$ are eigenvectors for $\mathbf{M}$

6 Transformation Matrices

1. **(2D linear transformations)** scaling, shearing, reflection, rotation, etc. talks bout how symmetric eigenvalue decomposition can decompose a matrix into rotation, scaling, and rotation back to previous direction. and how singular value decomposition also has a geometric meaning, except the two rotation are not the same

2. **(3D linear transforations)**

   - **(scaling)**
   $$scale(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix}$$
   - **(rotation)** about one of x,y,z axis.
   $$rotate\_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
   - **(arbitrary rotations)**
   - **(transforming normal vectors)** surface normals are perpendicular to the tangent plane of a surface. Let $\mathbf{t}$ be any tangent vector and $\mathbf{n}$ be surface normal. $\mathbf{t}_M = \mathbf{M}\mathbf{t}$ will still be tangent to the transformed surface. However $\mathbf{M}\mathbf{n}$ may not be perpendicular to the transformed surface. However we can find some matrix $\mathbf{N}$ that transform surface normal correctly
   $$\mathbf{n}^T\mathbf{t} = 0 \quad \mathbf{t}_M = \mathbf{M}\mathbf{t} \quad \mathbf{n}_N = \mathbf{N}\mathbf{n} \quad \mathbf{n}_N^T\mathbf{t}_M = 0$$
   $$\mathbf{n}^T\mathbf{t} = 0 \quad \rightarrow \quad (\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}\mathbf{t}) = (\mathbf{n}^T\mathbf{M}^{-1})\mathbf{t}_M = 0 \quad \rightarrow \quad \mathbf{n}_N = \left(\mathbf{n}^T\mathbf{M}^{-1}\right)^T = (\mathbf{M}^{-1})^T\mathbf{n}$$
   therefore $N = (M^{-1})^T$

3. **(Translation and Affine Transformation)** affine transformation is linear transformation followed by a translation. We an represent a 2D affine transformation on position vector with a 3×3 matrix by sing homomeneous coordinate
   $$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & x_t \\ m_{21} & m_{22} & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$
   Matrix multiplication can be used to compose affine transformations. For vector that represent directions or offsets, vectors should not change when we translate the object, we set the third coordinate to zero to accomodate this
   $$\begin{pmatrix} m_{11} & m_{12} & x_t \\ m_{21} & m_{22} & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$$
   Another way uses shearing in a higher dimension to represent translation. More detail in book

4. **(windowing transformation)** Create transformation matrices that takes point in rectangle $[x_l, x_h] \times [y_l, y_h]$ to rectangle $[x'_l \times x'_h, y'_l \times y'_h]$, which can be accomplished by translation to origin, scale, then translate back
   $$window = translate(x'_l, y'_l) \cdot scale(\frac{x'_h - x'_l}{x_h - x_l}, \frac{y'_h - y'_l}{y_h - y_l}) \cdot translate(-x_l, -y_l)$$
   For 3D windowing transformation from $[x_l, x_h] \times [y_l, y_h] \times [z_l, z_h]$ to $[x'_l, x'_h] \times [y'_l, y'_h] \times [z'_l, z'_h]$ is given by
   $$\begin{pmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & 0 & \frac{x'_l x_h - x'_h x_l}{x_h - x_l} \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & 0 & \frac{y'_l y_h - y'_h y_l}{y_h - y_l} \\ 0 & 0 & \frac{z'_l z_h - z'_h z_l}{z_h - z_l} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

5. **(Inverses of Transformation Matrices)** certain types of transformation matrices are easy to invert. scaling are diagonal matrices. rotation are orthogonal matricies so inverse is simply the transpose. For arbitrary matrix, we can apply SVD and decompose into rotation, scaling, and another rotation, take inverses of each and then compose them back.

6. **(coordinate transformations)** Let $\mathbf{o}, (\mathbf{x}, \mathbf{y})$ be canonical coordinate system in 2D and let $\mathbf{e}, (\mathbf{u}, \mathbf{v})$ be another coordinate frame. We can specify a **frame-to-canonical matrix** for the (u, v) frame, taking points expressed in (u, v) frame and converts them to same points expressed in the canonical frame.

$$\mathbf{p}_{xy} = \begin{pmatrix} \mathbf{u} & \mathbf{v} & \mathbf{e} \\ 0 & 0 & 1 \end{pmatrix} \mathbf{p}_{uv}$$

The **canonical-to-frame matrix** is the inverse

$$\mathbf{p}_{uv} = \begin{pmatrix} \mathbf{u} & \mathbf{v} & \mathbf{e} \\ 0 & 0 & 1 \end{pmatrix}^{-1} \mathbf{p}_{xy}$$

Similarly for 3D...

7 Viewing

1. **(viewing transformations)** Moving object between their 3D positions (world space) and their positions in a 2D view of the 3D world (image space). Important to object-order rendering. Idea is we can find matrices that can project any point on a given pixel's viewing ray back to that pixel's position in the image space

2. **(viewing transformations)** map 3D location w.r.t. (x, y, z) coordinate in the canonical coordinate system to coordinates in the image w.r.t. pixels

   (a) (camera/eye transformation) is a rigid body transformation that places the camera at the origin in a convenient orientation, depending on position and orientation, or pose, of the camera

   (b) (projection transformation) projects points from camera space so that all visible points fall in the unit cube $x \in [-1, 1]^3$, only depending on the type of projection desired

   (c) (viewport/windowing transformation) maps unit image rectangle to desired rectangle in pixel coordinate, depending on the size and position of the output image

The camera transformation converts points in world space to camera space. The projection transformation converts points in camera space to canonial view volume. The viewport transformation maps canonical view volume to screen space Assume we wish to view a scene with an orthographic camera

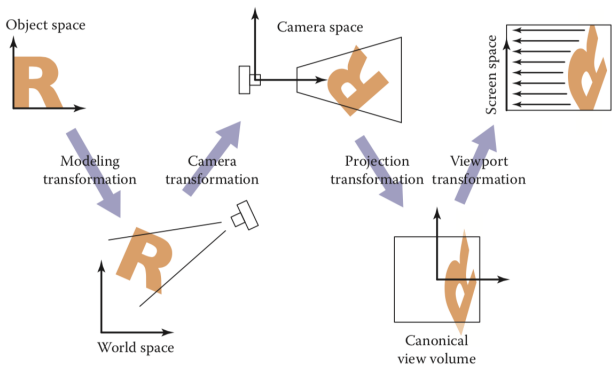

Figure 7.2. The sequence of spaces and transformations that gets objects from their original coordinates into screen space.

Figure 1: viewing_transformations

looking along $-z$ direction with $+y$ up

3. **(viewport transformation)** maps axis-aligned rectangle to another, i.e. $[-1,1]^2 \to [-0.5, n_x - 0.5] \times [-0.5, n_y - 0.5]$ where n_x, n_y are number of pixels for the image. Recall windowing transformation, we have a matrix that ignores z-coordinate of points in the canonical view volume, since a point's distance along the projection direction doesn't affect where the point projects in the image.

$$M_{vp} = \begin{pmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

the matrix is called the viewport matrix M_{vp}, where it leaves z-coordinates of points unmodifieds

4. **(orthographic projection transformation)** Instead of the canonical view volume, we want to accomodate arbitrary n-orthotope for orthographic projection. Given coordinates of the side of an axis-aligned volume, called the orthographic view volume, of size $[l,r] \times [b,t] \times [f,n]$, we want a map to the canonical view volume. This is again a windowing transformation

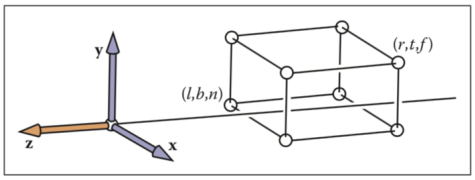

Figure 2: orthographic view volume: on the -z axis with $0 > n > f$

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

5. **(the camera transformation)** convention is to use **e** or eye position, **g** for gaze direction and **t** for view-up vector. We can construct a righ-handed basis with

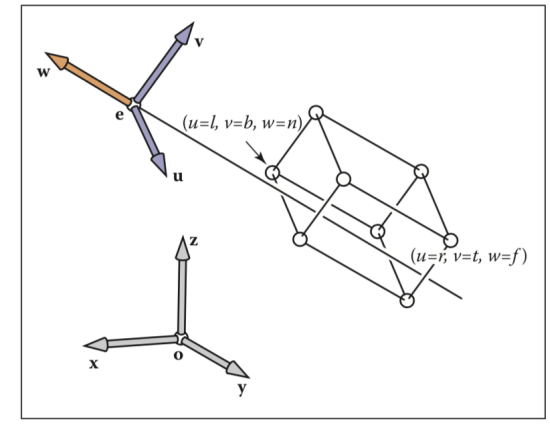

Figure 3: the camera's coordinate system

$$\mathbf{w} = -\frac{\mathbf{g}}{\|\mathbf{g}\|} \quad \mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|} \quad \mathbf{v} = \mathbf{w} \times \mathbf{u}$$

idea is $\mathbf{w}$ acts similarly to $-\mathbf{z}$. Now we have a camera coordinate given by $\mathbf{e}, (\mathbf{u}, \mathbf{v}, \mathbf{w})$. We want point's coordinate be with respect to the camera coordinate system, however the points coordinate is in world space. The camera transformation is simply the world-to-frame transformation

$$M_{cam} = \begin{pmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1}$$

To sum up,

$$\begin{pmatrix} x_{pixel} \\ y_{pixel} \\ z_{canonical} \\ 1 \end{pmatrix} = (\mathbf{M}_{vp}\mathbf{M}_{orth}\mathbf{M}_{cam}) \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

6. **(projective transformation)** Key property is that the size of an object on the screen (view plane) is proportional to $1/z$ for an eye at the origin looking up the negative z-axis

$$y_s = \frac{d}{z}y$$

where y is distance of point along y-axis and y_s is where the point should be drawn on the screen Two

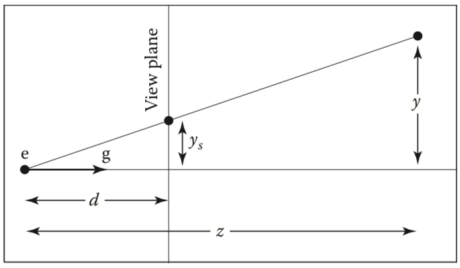

Figure 4: Eye at **e** gaze along **g** (i.e. $-\mathbf{z}$ axis). The view plane is distance d from the eye.

method of representing projection transformation with matrices

(a) (3D projective transformation) idea is to define $(x\ y\ z\ w)^T$ to represent the point $(x/w, y/w, z/w)$ since $w = 1$. In this way the denominator w can be an affine function of the original coordinates,

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \\ \tilde{w} \end{pmatrix} = \begin{pmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ e & f & g & h \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \Rightarrow \quad (x', y', z') = (\tilde{x}/\tilde{w}, \tilde{x}/\tilde{w}, \tilde{z}/\tilde{w})$$

(b) (4D linear transformation) Idea is to view 3D projective transformation as a 4D linear transformation with the constraint that $\mathbf{x} \sim \alpha\mathbf{x}$ for all $\alpha \neq 0$, where $\mathbf{a} \sim \mathbf{b}$ means $\mathbf{a}$ and $\mathbf{b}$ are two homogeneous vectors representing the same point in cartesian coordinate (i.e. last coordinate is 1). Intuitively, 3D points in cartesian coordinate is a linear vector in 4D.

7. **(perspective projection)** In 1D, use projection transformation to implement perpsective projection

$$\begin{pmatrix} y_s \\ 1 \end{pmatrix} \sim \begin{pmatrix} d & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} dy \\ z \end{pmatrix} \sim \begin{pmatrix} \frac{dy}{z} \\ 1 \end{pmatrix}$$

In 3D, the image plane distance is $-n$ and the distance of point (x, y, z) is $-z$. The perspective matrix maps the persepctive view volume (a slice of a pyrmid) to the orthographic view volume

$$\mathbf{P} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 1 \end{pmatrix} \qquad \mathbf{P}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ (n+f)z - fn \\ z \end{pmatrix} \sim \begin{pmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n+f-\frac{nf}{z} \\ 1 \end{pmatrix}$$

where row 1,2,4 implements perspectiv projection and row 3 leaves points on $z = n$ plane alone and points on $z = f$ plane alone while squishing points in between by the appropriate amount and preserves relative order of z values. To convert screen coordinate plus z back to the original space, we can apply the inverse of projective projection

$$\mathbf{P}^{-1} = \begin{pmatrix} \frac{1}{n} & 0 & 0 & 0 \\ 0 & \frac{1}{n} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{1}{fn} & \frac{n+f}{fn} \end{pmatrix} \sim \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 0 & fn \\ 0 & 0 & -1 & n+f \end{pmatrix}$$

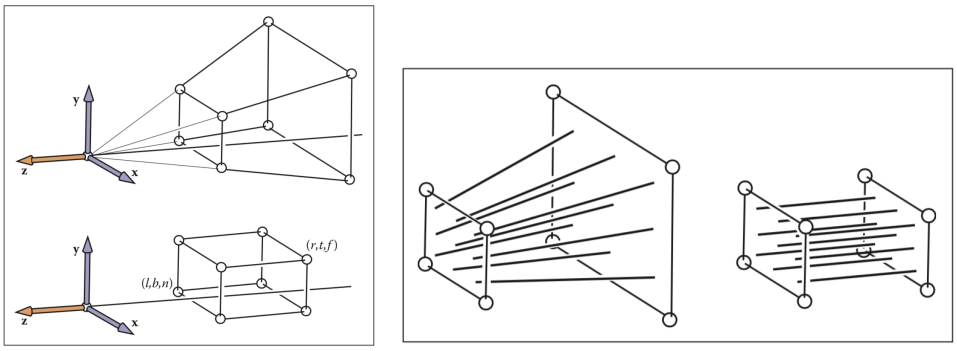

Figure 5: perspective projection maps line thorugh origin/eye to a line parallel to z-axis

The perspective projection matrix maps perspective view volume to canonical view volume

$$\mathbf{M}_{per} = \mathbf{M}_{orth}\mathbf{P} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

To sum up,

$$\begin{pmatrix} x_{pixel}/w_{pixel} \\ y_{pixel}/w_{pixel} \\ z_{ordered}/w_{pixel} \\ 1 \end{pmatrix} \sim \begin{pmatrix} x_{pixel} \\ y_{pixel} \\ z_{ordered} \\ w_{pixel} \end{pmatrix} = (\mathbf{M}_{vp}\mathbf{M}_{orth}\mathbf{P}\mathbf{M}_{cam}) \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

note we need to divide by the homogeneous coordinate w if perspective projection $\mathbf{M}_{per}$ is included.

8. **(properties of perspective transform)** perspective transform preserves straight lines, planes, and triangles, and preserves the ordering of the points.

9. **(field of view)** although we can specify any window (l, r, b, t) and distance to plane n, we can have a simpler system where we

   (a) look through the center of the window ($l = -r$ and $b = -t$)

   (b) pixels are square and we want no distortion in the image ($nx/ny = r/t$)

   So pixel count n_x, n_y, (verticle) field-of-view θ, and distance to plane n defines the viewing window.

$$\tan\frac{\theta}{2} = \frac{t}{|n|}$$

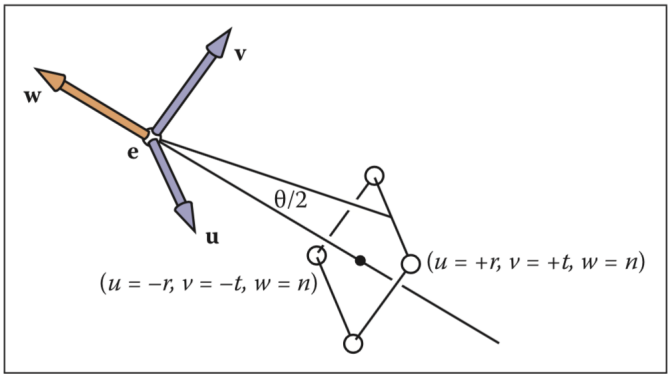

Figure 6: field-of-view θ is the angle from bottom of the screen to the top of the screen

8 The Graphics Pipeline

1. **(the graphics pipeline)** object-order rendering cosiders each object and find pixels that it could have an effect on. The proces of finding all pixels in an image occupied by an object is called rasterization. object-order rendering can also be called rendering by rasterization. This sequence of operations in called the graphics pipeline. It is more effient.

   (a) interactive rendering via hardware for speed (OpenGL, Direct3D)

   (b) film production rendering for quality (RenderMan)

   Steps are

   (a) (vertex processing) incoming vertices transformed by modeling, viewing, projection transformation which maps them from original coordinate into screen space. Color, surface normal, texture are also transformed as needed.

   (b) (rasterization) converts continuous representation of object to discrete pixels

   (c) (fragment processing) compute a color and depth for each fragment. shading operations

   (d) (fragment blending) compute final cover by combining fragments that overlapped each pixel, i.e. pick colo of fragment with smallest depth

2. **(rasterization)** In essence, rasterization converts continuous representation of objects to a discrete representation with pixels. The rasterizer enumerates pixels covered by the primitives and it interpolates values, called attributes, across the primitive. The output of the rasterizer is a set of fragments, one for each pixel covered by the primitive. Each fragment (lives) at a particular pixel and carries its own set of attribute values

3. **(line drawing)** given implicit function

$$f(x,y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0$$

   assuming $x_0 < x_1$ and the slope $m = (y_1 - y_0)/(x_1 - x_0) \in (0, 1]$. The midpoint algorithm keep drawing pixels from left to right and move upward in the y-direction if $f(x + 1, y + 0.5) < 0$, i.e. the line passes over the midpoint of $(x + 1, y)$ and $(x + 1, y + 1)$

4. **(triangle rasterization)** In 2D, given $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ in screen coordinates, we can draw triangle and fillin color with barycentric coordinates

$$\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$$

   where $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2$ are color of triangle vertices. This is called Gouraud interpolation

5. **(A simple 2D pipeline)** rasterization followed by blending stage where each fragment simply overwrites value of the previous one. This is the typical API for interfaces, graphing libraries

6. **(A minimal 3D pipeline)** To draw 2D objects in 3D, simply follow the 2D pipeline with one addition: in vertex processing stage the incoming vertex positions are multiplied by the modeling, camera, projection, and viewport mtraices, resulting in screen-space triangles drawn in the same way as if they'd been specified directly in 3D. Use **painter's algorithm** to get the correct occlusion relationship, i.e. primitives are draw in back-to-front order. However, the drawbacks are

   (a) cannot handle triangles that intersect one another

   (b) several triangles can be arranged in an occlusion cycle

   (c) sorting primitives by depth is slow

7. **(use z-buffer for hidden surfaces)** for each pixel, allocate a **depth buffer**, or z-buffer to store the closest distance of fragments that have been drawn so far. In the fragment blending stage, fragments with distance larger than the depth buffer is thrown away. This requires each fragment to carry a depth, which is computed by interpolation of z-coordinates. Thereare precision issues with z-buffer.z buffer are nonnegative integers instead of float for memory speed reasons.

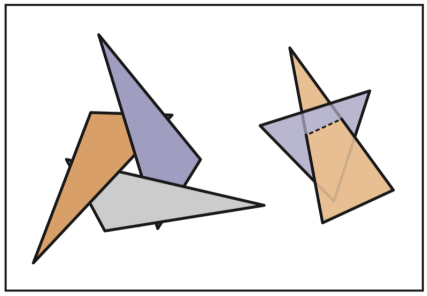

Figure 7: Drawback's of painter's algorithm

8. **(per-vertex shading)** idea is we want 3D objects to be drawn with shading, using the same illumination euqation used for image-order rendering. These equation requries a light direction, eye direction and surface normal to compute the color of a surface. Idea is

   (a) application provides normal vectors at each vertex

   (b) positions and color of lights are provided separately

   (c) at each vertex, the direction to the viewer and direction to each lights are computed based on positions of the camera, lights, and vertex.

   the shading equation is evaluated to compute a color, which is passed onto the rasterizer as the **vertex color**. Per-vertex shading is called **Gouraud shading**. Usually, computation of vertex color is in a coordinate system orthonormal when viewed in the world space, e.g. the eye space. per-vertex shading has the disadvantage of not producing details in shading that are smaller than the primitives used to draw the surfaces, since it computes shading once for each vertex and never in between vertices.

9. **(per-fragment shading)** To avoid interpolation artifacts associated with per-vertex shading, we can avoid interpolating colors by performing shading computations after the interpolation in the fragment stage. Geometric information needed for shading is passed through the rasterizer as attributes, so the vertex stage must coordinate with fragment stage to prepare the data appropriately. One approach is to interpolate eye-space surface normal and eye-space vertex positions.

10. **(texture mapping)** textures are images that are used to add extra detail to the shading of surfaces that would otherwise look too homogeneous and artificial. Idea is each time shading is computed, read one values used in the shading computation, e.g. diffuse color, from the texture instead of attribute values attached to the geometry. This is called **texture lookup** the shading code specifies a **texture coordinate**

11. **(shading frequency)**

    (a) (low shading frequency) shading with large-scale features can be evalauted fairly infrequently and interpolated

    (b) (high shading frequency) shading that produces small-scale features, such as sharp highlights or detailed textures. Most likely the shading frequency needs to be at least one shading sample per pixel

9 Signal Processing

10 Surface Shading

11 Texture Mapping

1. **(texture mapping)** are spatially varying surface properties, i.e. surface detail, make shadows/reflections, provide illumination, define surface shape

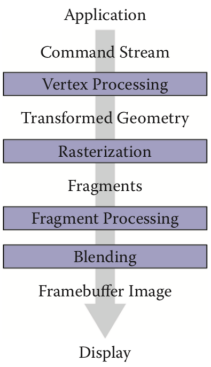

Figure 8: graphics pipeline

1. **(graphics hardware)** hardware necessary to quickly render 3D objects using specialized rasterization-based hardware architectures. CPU can be programmed with OpenCL and CUDA

2. **(heterogeneous multiprocessing)**

   (a) **(context)** the mapping between the OS, hardware driver, the hardware and the windowing system is known as the graphics context. Context are established via API calls to the windowing system

   (b) **(OpenGL)** is cross-platform graphics API. Need to write code for both CPU and GPU. Currently data is sent to graphis card before it is needed and instacing it at render time. Matrix stack is deprecated, need to use third-party libraaries like GLM. The shader langauge (GLSL) takes a larger role: perform matrix transformations, along with lighting and shading within the shaders.

3. **(buffer)** linear allocation of memory on device on which GPU can operate. Programmatically, need to initialize buffer needed for application beforehand, i.e. a copy from host to device. At end of various stages of execution, can do device to host copies or pull data from GPU to CPU memory.

   (a) **(display buffer)** 2d array of color values for display eventually mapped to the window (implemented as 1d linear buffer). Fragment processing and blending stages write data to the output display buffer memory. The windowing system reads the contents of the display buffer to produce the raster images on the monitor's window

   (b) **(cycle of refresh)** double-buffered display state, i.e. the front buffer (drive pixel color) and the back buffer (holds current changes) are associated with a window. At end of rendering loop, buffers are swapped through a pointer texchange, the rendering will appear seamless if buffer pointer swap is synchronized with windowing system's refresh.

   (c) **(clearing memory to default state)** we can clear background color (to black) with

   ```
   glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
   glClear(GL_COLOR_BUFFER_BIT);
   ```

   we can clear depth buffer, representing distance of fragments relative to camera, with

   ```
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   ```

   These clearing needed before any geometry/fragments are processed

4. **(state)** graphics card maintain a computational state, which determines how (subsequent) computations associated with scene data and shader will occur on the graphics hardware. efficient OpenGL programs attempt to minimize state changes, enabling states needed, while disabling states that are not requried for rendering.

(a) **(color states)** `glClearColor` function sets default color values that are written to all pixels within the color buffer when `glClear` is called. Each time `glClear` is called it uses the previously set state of the clear color (black in this case)

(b) **(z-buffer algorithm state)** in depth test, a fragment's depth values will be compared to the depth value currently stored in the depth buffer prior to writing any fragment colors to the color buffer.

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
```

Sometimes depth test is not necessary and we can prevent z-buffer computation

```
glDisable(GL_DEPTH_TEST);
```

5. **(basic OpenGL application layout)** A simple OpenGL application is a display loop that is either as fast as possible, or at a rate that coincides with the refresh rate of the monitor of display device. Below uses `GLFW` as for cross-platform windowing

```
while (!glfwWindowShouldClose(window)) {
    // OpenGL code called each time loop executed
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // swap front and back buffers
    // synchronize graphics context with display refresh
    glfwSwapBuffers(window);

    // poll for evernts
    glfwPollEvents();

    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, 1);
}
```

framebuffer is collection of depth and color buffers. framebuffer is directly related to size of the window that has been opened to contain the graphics context. The window, or viewport dimension is needed to contruct M_{vp} matrix. This is accomplished with

```
int nx, ny;
glfwGetFrameBufferSize(window,&nx, &ny);
glViewport(0, 0, nx, ny);
```

6. **(geometry)** specified using arrays to store vertex data and other vertex attributes, i.e. vertex color, normal, texture coordinates.

   (a) **(hardware representation)** The primitive types are limited to
   - **(points)** to represent points or particle system
   - **(lines)** pair of vertices to represent lines, sihouettes, edge highlighting
   - **(triangles)** triangles, triangles strips, indexed triangles, indexed triangle strips, quadrilaterals, or triangle meshes approximating the geometric surfaces

7. **(shaders)** mechanism by which computation occurs on GPU related to per-vertex or per-fragment processing. No primitives can be rendered without at least one vertex shader to process incoming primitive vertices and nother shader to process the rasterized fragments. *geometry shader* designed to process primitives, and potentially creating additional primitives. *compute shader* designed for performing general computation on GPU, and can be linked into the set of shaders necessary for a specific application

   (a) **(vertex shader)** control over how vertices are transformed, stage intermediate per-vertex values that will be interpolated in the fragment shader. Additionally, it could be used to perform general computations on GPU. (e.g. model particle motion of particles) Advanced general computation maybe coded with compute shaders. Data flows out of vertex shader, variable with `out` specifier, are inputs `in` variable to fragment shader if the name match up!

(b) **(passthrough vertex shader)** passes incoming vertex position out as `gl_Position`, a built-in reserved variable that OpenGL uses to rasterize fragments, without accoutning for projection, viewing, or model transformations

```
#version 330 core
layout(location=0) in vec3 in_Position;
void main(void)  gl_Position = vec4(in_Position, 1.0);
```

Operations in vertex/fragment shaders are SIMD operations operating on all vertices/fragments in parallel. Additional data can be communicastedf from host to shaders by using input, output, or *uniform variables*. `location=0` indicate that the source data is the attribute index 0 that is associated with the geometry.

(c) **(fragment shader)** outgoing value is written to the color buffer. Note `out` varaib lvariables passed from vertex to fragment shader are interpolated across the fragmnts (i.e. face of a triangle) using barycentric interpolation.

```
#version 330 core
layout(location=0) out vec4 out_FragmentColor;
void main(void)  out_FragmentColor = vec4(1,1,1,1);
```

in this fragment shader, all fragment set to white. `location=0` means location of output is color buffer index 0.

8. **(loading, compiling, using shaders)** shader are transferred to graphics hardware in form of character strings, which must be compiled and linked (to a shader program). At end of compilation, compilation status and errors can be queried. A shader program is what is used to affect rendering of geometry.

9. **(object)** such as vertex buffer objects, framebuffer objects, texture objects, shader programs, are primary target for computation and processing, and should be bound to a known OpenGL state when used and unbounded when not in use.

10. **(vertex buffer objects (VBO))** are storage containers for vertices, as well as vertex attributes, i.e. colors, normal vectors, texture coordinates. Steps

(a) allocate vertices for primitives in host memory

`GLfloat vertices[] =` $\{-0.5f, -0.5f, -0.5f\}$;

(b) vertex buffer object created on device (actual allocation not yet performed)

```
GLuint triangleVBO[1];
glGenBuffers(1, triangleVBO);
```

(c) bind state of OpenGL to the VBO handle

```
glBindBuffer(GL_ARRAY_BUFFER, triangleVBO[0]);
```

Any operations that affect vertex buffers that follow `glBindBuffer` will use the triangle data stored in VBO by reading/writing to it.

(d) vertex on host copied to device before display loop

```
glBufferData(GL_ARRAY_BUFFER, 3*sizeof(GLfloat), verticecs, GL_STATIC_DRAW);
```

`GL_STATIC_DRAW` indicate that vertices will not change over course of rendering

(e) host memory can be released after vertex data is transferred

(f) when VBO no longer needs to be active for reading/writing, unbound with

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

11. **(vertex array object (VAO))** represent OpenGL's mechanism to bundle vertex buffers together into a consistent vertex state that can be communicated asnd linked with shadrs in the graphics hardware. In short, VAO makes connection between data (VBO) and shader input variable indices. The following steps allows for `layout(location=0) in vec3 in_Position` to access bound vertex buffer in shader code.

    (a) create and bind vertex array buffer

    ```
    GLuint VAO;
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);
    ```

    (b) enables vertex attribute index (in this case, 0). Without this, data in VBO will not be visible to shader

    ```
    glEnableVertexAttribArray(0);
    ```

    (c) activate the VBO storing vertex data to be bound to vertex attribute

    ```
    glBindBuffer(GL_ARRAY_BUFFER, triangleVBO[0]);
    ```

    (d) tells how OpenGL interprets data stored in VBO. Specifically, set attribute index 0 to hold 3 component (x, y, z) of `GLfloats` that are not normalized. `3 * sizeof(GLfloat)` means vertiecs are tightly packed in memory

    ```
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
    ```

    (e) Now inside display loop, the following trigger processing of vertex array objects.

    ```
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glBindVertexArray(0);
    ```

    where `glDrawArrays` initiates pipleline for this geometry, interpreted as a series of triangle primitives starting at offset 0 and only rendering 3 of the indices.

12. **(transformation matrices)** GLM, OpenGL Mathematics, extends C++'s math capabilities, like Eigen

13. **(uniform data)** static data that is invariant across the execution of a shader program. It is same for all elements and is static. Usually uniform variable represent graphics state, i.e. projection, view, model matrices, light sources.

    ```
    #version 330 core
    layout(location=0) in vec3 in_Position;
    uniform mat4 projMatrix;
    void main(void) {
      gl_Position = projMatrix * vec4(in_Position, 1.0);
    }
    ```

    To make `projMatrix` be available in shader, need to

    (a) get handle to a uniform variable

    ```
    GLint pMatID = glGetUniformLocation(shaderProgram, "projMatrix");
    ```

    (b) bind shader program

    ```
    glUseProgram(shaderID);
    ```

(c) transfer memory from host to device with `glUniform` functions. `4fv` means its 4×4 matrix of floats and that array contains data, rather than passing value of pointer.

```
glUniformMatrix4fv(pMatID, 1, GL_FALSE, glm::value_ptr(projMatrix));
```

14. **(shading with per-vertex attributes)** can specify additional data, i.e. normal, texture coordinates, colors in VBO and bind to vertex attributes and access with

```
layout(location=0) in vec3 in_Position;
layout(location=1) in vec3 in_Color;
```

15. **(structure for vertex data)** can bundle vertex position and vertex attributes into structures

```
struct vertexData { glm::vec3 pos; glm::vec3 color };
vector<vertexData> modelData;
```

16. **(shading in fragment processor)** gives better visual results and more accurate approximation of lighting than in vertex shader; in latter case, lighting only calculated per-vertex and fails to approximates lighting across surface. Goal of fragment shader is to output a value to framebuffer, i.e. color of a pixel, and depth if depth test enabled

17. **(Blinn-Phone Shader Program)** where VBO contains vertex position and normal vectors. Shading performed in fragment shader in camera space

```
// vertex shader
#version 330 core
layout(location=0) in vec3 in_Position;
layout(location=1) in vec3 in_Normal;
out vec4 normal;
out vec4 half;
out vec3 lightdir;
struct LightData {
    vec3 position;
    vec3 intensity;
};
uniform LightData light;
uniform mat4 projMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;
uniform mat4 normalMatrix;

void main(void) {
    // calculate lighting in camera space
    vec4 pos = viewMatrix * mdoelMatrix * vec4(in_Positino, 1);
    vec4 lightPos = viewMatrix * vec4(light.position, 1);

    normal = normalMatrix * vec4(in_Normal, 0);

    vec3 v = normalize(-pos.xyz);    // view direction
    lightdir = normalize(lightPos.xyz - pos.xyz)
    half = normalize(v + lightdir);

    gl_Position = projMatrix * pos;
}
```

```
// fragment shader
#version 330 core

in vec4 normal;
in vec3 half;
in vec3 halfdir;

layout(location=0) out vec4 fragmentColor;

struct LightData {
    vec3 position;
    vec3 intensity;
};
uniform LightData light;

uniform vec3 Ia;
uniform vec3 ka, kd, ks;
uniform float phongExp;

void main(void) {
    vec3 n = normalize(normal.xyz);
    vec3 h = normalize(half);
    vec3 l = normalize(lightdir);

    vec3 intensity = ka*Ia +
            kd*light.intensity*max(0, dot(n, l)) +
            ks*light.intensity*pow(max(0, dot(n, h)), phongExp);
    fragmentColor = vec4(intensity, 1);
}

// a normal shading fragment shader
#version 330 core
in vec4 normal;
layout(location=0) out vec4 fragmentColor;
void main(void) {
    vec3 intensity  = normalize(normal.xyz) * 0.5 + 0.5;
    fragmentColor = vec4(intensity, 1);
}
```

18. **(meshes and instancing)** for meshes, it is better to use `glDrawElements` over `glDrawArrays` since there are substantial vertex reuse. Also, OpenGL acn render many object from a single copy of VAO and associated VBO

19. **(texture objects)** need to be allocated and initialized by copying data on the host to device and setting the OpenGL state. Texture coordinates are often integrated into vertex buffer object and passed as vertex attributes to shader programs. Fragment shader usually performs texture lookup functions

```
float* imgData = new float[imgHeight * imgWidth * 3];
GLuint texID;
glGenTextures(1, &textID)
glBindTexture(GL_TEXTURE_2D, textID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
// host -> device copy
glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, imgWidth, imgHeight, 0, GL_RGB, GL_FLOAT, imgData);
glBindTexture(GL_TEXTURE_2D, 0);
```

We can then access textures by storing texture coordinates in vertex buffer object

```
struct vertexData{
 glm::vec3 pos;
 glm::vec3 normal;
 glm::vec2 texCoord;
}
```

Also need to bind texture to *texture units*, which will be made available as a uniform variable in shader

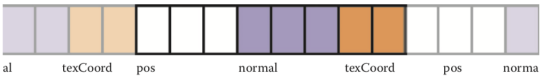

Figure 9: Data layout of vertex buffer

programs

20. **(texture lookup in shaders)** texture lookup mostly done in fragment shader but vertex shader often stages fragment computation by passing texture coordinate out to the fragment shader, so that texture coordinates will be interpolated and afford per-fragment lookup of texture data

```
// vertex shader
layout(location=2) in vec2 in_TexCoord;
out vec2 tCoord;
tCoord = in_TexCoord;
```

In fragment shader, a uniform varaible store the texture unit to which the texture is bound. This uniform variable has a *sampler* type, i.e `sampler2D`. Texture lookup with `texture`, which replaces diffuse coefficient of the shading model.

```
in vec2 tCoord;
uniform sampler2D textureUnit;
void main(void){
  vec3 kdTexel = texture(textureUnit, tCoord).rgb;
  vec3 intensity = ka * Ia +
    kdTexel * light.intensity * max(0, dot(n, l)) +
    ks * light.intensity * pow(max(0, dot(n, h)), phongExp); }
```

21. **(object-oriented design)**