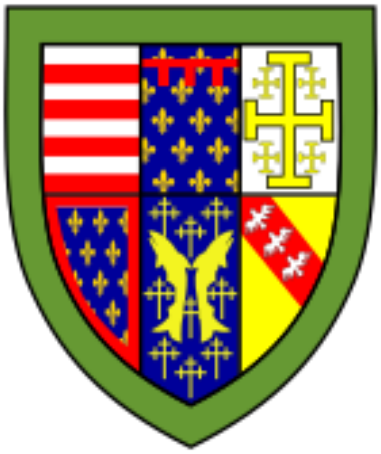
Queens' College Cambridge

# Introduction to Graphics



Alistair O'Brien

Department of Computer Science

October 11, 2020

# Contents

# 1 Image, Pixels and Sampling

## 1.1 Images and Pixels

**Definition 1.1.1.** (**Pixel**) A pixel (picture element) is a point in a raster image.

**Definition 1.1.2.** (**Raster Image**) A raster image is a discrete 2D array containing pixel values for each pixel (usually a set of tristimulus values).

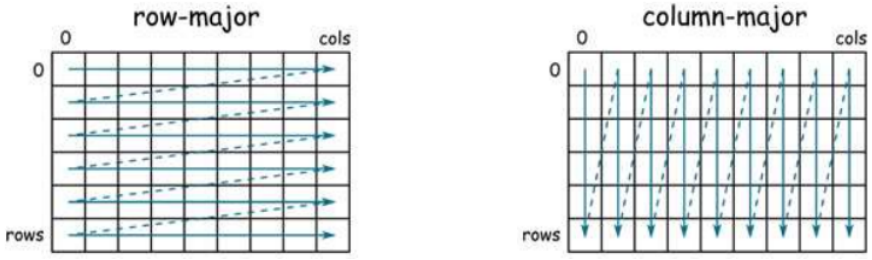
- Raster images are convenient for raster devices, however, a device independent representation is required.

**Definition 1.1.3.** (**Image**) An image is the function

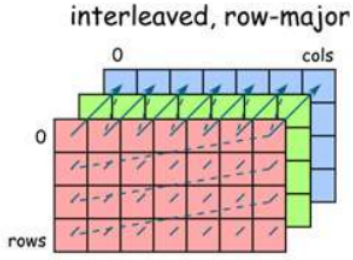
$$I(x, y) : X \times Y \to V,$$

where $X \times Y \subset \mathbb{R}^2$ and $V$ is the set of possible pixel values.

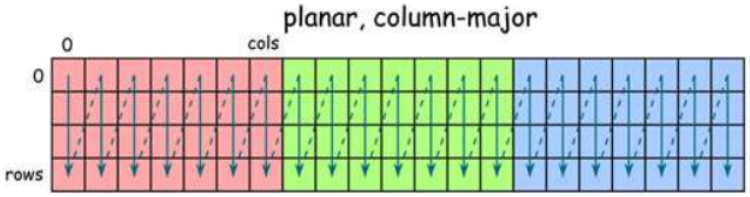
- For a raster image, we define $X \times Y = [0, n_x - 1] \times [0, n_y - 1] \subset \mathbb{R}$ where $n_x$ is the number of columns and $n_y$ is the number of rows.
- Two main representations:
  - Row-major:
    $$I(x, y) = x + y \cdot n_x.$$
  - Column-major:
    $$I(x, y) = x \cdot n_y + y.$$




- Introduction of color provides two more representations:
  - Interleaved:
    $$I(x, y, c) = x \cdot s_x + y \cdot s_y + c,$$
    where $c$ specifies the color dimension.




  - Planar:
    $$I(x, y, c) = x \cdot s_x + y \cdot s_y + c \cdot n_x.$$




  - General case:
    $$I(x, y, c) = x \cdot s_x + y \cdot s_y + c \cdot s_c,$$
    where $s_x, s_y$ and $s_c$ are the "strides".

- Monocular depth cues in raster images:
  - Perspective-based cues:
    - Occlusion
    - Relative size
    - Shadow
    - Distance to horizon
  - Other cues:
    - Shading
    - Color
    - Relative Brightness
    - Atmosphere
    - Focus (see distributed ray tracing)
  - Partially Perspective-based cues:
    - Similar size
    - Texture Gradient
- Binocular depth cues required special 3D displays and 2 raster images. This is due to *Steropsis*, the process which angular disparity between the left and right images in the eye is used to compute the depth points within the image. IMAGE HERE

## 1.2 Pixel Values

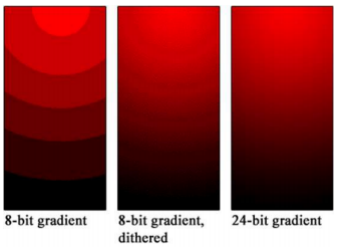
- We may describe pixel values using color spaces, with each tristimulus value represented as a floating-point number.
- Since tristimulus values exist in the range $[0, 1]$. Images that store floating-point numbers are *high dynamic range* (HDR) images. Images that store 8-bit integers, to represent $0, 1/255, \ldots, 254/255, 1$ are *low dynamic range* (LDR).
- Typical representations:
  - 1-byte grayscale.

  - 8-bit RGB (24 bits) referred to as true-color.
  - 8- to 10-bit RGB (24-30 bits) ITU-R 709 color space.
  - 12- to 14-bit RGB (36-42 bits) ITU-R 2020 color space.
  - 16-bit floating-point RBG (48 bits) HDR images.
  - 32-bit floating-point RBG (48 bits) HDR images.

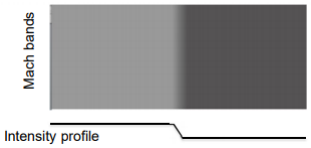
- Reducing the number of bits introduces artifacts.

**Definition 1.2.1.** (**Artifacts**) An artifact is any anomaly in the representation of images.

- Encoding images with limited precision leads to *quantization* artifacts, or *banding*.




(c) Banding




(d) Mach Band Illusion

- This type of artifact is particularly noticeable due to the Mach band illusion
- Dithering (added noise) can reduce banding

## 1.3 Sampling

**Definition 1.3.1.** (**Sampling**) Sampling is the process of mapping a continue function $x(n)$ to a discrete function $x[n]$, more formally, it is the process of mapping a continuous domain $\mathcal{D} \subseteq \mathbb{R}$ to a discrete domain $[\mathcal{D}] \subset \mathbb{N}$.

- Samples are equally spaced along the continuous function (or signal)

- $T$ denotes the "time interval" between samples, so

$$x[n] = x(nT),$$

  where $n \in \mathbb{N}$ is the sample number.

**Theorem 1.3.1.** (**Nyquist's Theorem**) A continuous function $x(n)$ with maximum frequency $f$ can be perfectly reconstructed from $x[n]$ if the sampling frequency $f_S = 1/T > 2f$.

**Definition 1.3.2.** (**Quantization**) Quantization is the process of mapping a continue range $\mathcal{R}$ to a discrete range $[\mathcal{R}] \subset \mathcal{R}$

- Unlike sampling, quantization is generally irreversible (we may approximately reconstruct $x(n)$ from $x[n]$ using nearest-neighbor).
- With $L$ levels, we require $N = \log_2 L$ bits to representation the different levels. Quantization maps some continuous variable to a discrete set of quantization levels $\{r_1, \ldots, r_L\}$.
- Mapping is based on thresholding / comparison with transition levels $t_k$.

$$t_k \leq x < t_{k+1} \implies x \mapsto r_k.$$

# 2 Ray Tracing

## 2.1 The Ray Tracing Algorithm

- A ray tracer has three main parts:
  - *ray generation*, computes a ray for each pixel based on the **eye**.
  - *ray intersection*, calculates the intersections of the ray and the objects, finds the closest object intersecting the ray.
  - *shading*, using the intersection point, surface normal, etc to determine color of the pixel
- So the structure of a ray tracing algorithm is:

```
for (pixel ∈ screen) {
    compute viewing ray
    find first object hit by ray and it's surface normal n̂
    compute pixel color using intersections, light and n̂
}
```

## 2.2 Ray Generation and Projections

- Represent a ray using a vector line

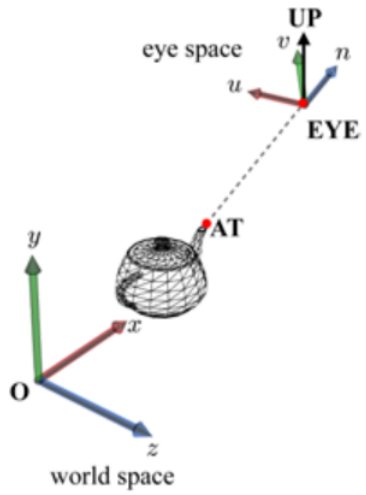
$$\mathbf{r} = \mathbf{a} + \lambda \mathbf{b},$$

  where $\mathbf{a}$ is the *origin* and $\mathbf{b}$ is the ray's *direction*.
- We define the viewer using the following parameters:
  - **eye**, the position of the eye.
  - **at**, the reference point which the eye looks at,

– **up**, the view up vector describes where the top of the eye points.

We define the **view space** $(\mathbf{eye}, \hat{\mathbf{n}}, \hat{\mathbf{u}}, \hat{\mathbf{v}})$ using

$$\hat{\mathbf{n}} = \frac{\mathbf{eye} - \mathbf{at}}{\|\mathbf{eye} - \mathbf{at}\|}$$
$$\hat{\mathbf{u}} = \frac{\mathbf{up} \times \hat{\mathbf{n}}}{\|\mathbf{up} \times \mathbf{n}\|}$$
$$\hat{\mathbf{v}} = \mathbf{n} \times \mathbf{u}$$




- The viewing plane is a plane perpendicular to $\hat{\mathbf{n}}$, that is

$$\mathbf{r} \cdot \hat{\mathbf{n}} = -d,$$

where $d$ is the distance from **eye** to the plane.

- The region of interest (ROI) is defined using:
  - Let $l$ and $r$ be the positions of the left and right edges of the image along the $\hat{\mathbf{u}}$ axis. $l < 0 < r$.
  - Let $t$ and $b$ be the positions of the top and bottom edges of the image along the $\hat{\mathbf{v}}$ direction. $b < 0 < t$

- From our ROI, pixels are spaced $(r-l)/n_x$ apart horizontally and $(t-b)/n_y$ vertically.
- We define our rays to intersect each pixel in the center, so the position $(x, y)$ in the raster image has

$$u = l + (r-l)\frac{x+1/2}{n_x}$$
$$v = b + (t-b)\frac{y+1/2}{n_y}$$

  where $1/2$ ensures the ray intersects the pixel at it's center and $(u, v)$ are the coordinates of the pixel's position on the view plane.

### 2.2.1 Parallel Projection

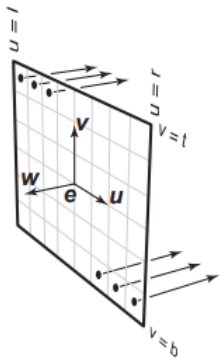
- For parallel projection all rays have direction $-\hat{\mathbf{n}}$, and the view plane has distance $d = 0$ from **eye**.
- So the ray is defined by

$$\mathbf{r} = (\mathbf{eye} + u\hat{\mathbf{u}} + v\hat{\mathbf{v}}) + \lambda(-\hat{\mathbf{n}}).$$

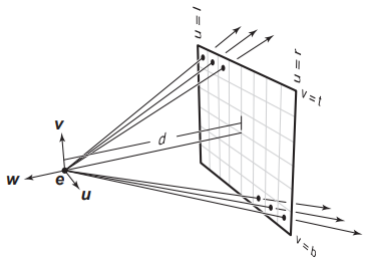
### 2.2.2 Perspective Projection

- All rays have the same origin (**eye**).
- The view plane is distanced $d$ from **eye**, known as the *focal length*.
- So the ray is defined by

$$\mathbf{r} = \mathbf{eye} + \lambda\left[d(-\hat{\mathbf{n}}) + u\hat{\mathbf{u}} + v\hat{\mathbf{v}}\right].$$




(a) Parallel Projection




(b) Perspective Projection

## 2.3 Ray Intersections

- Given a ray $\mathbf{r} = \mathbf{a} + \lambda\mathbf{b}$ and a implicit surface $f(\mathbf{r}) = 0$, find their intersection where $\lambda \in [0, \infty)$.

### 2.3.1 Plane

- Suppose we have a ray $\mathbf{r} = \mathbf{a} + \lambda\mathbf{b}$ and the plane $\mathbf{r} \cdot \mathbf{n} = p$. For our intersections, we have

$$\begin{aligned} & (\mathbf{a} + \lambda\mathbf{b}) \cdot \mathbf{n} = p \\ \iff & \mathbf{a} \cdot \mathbf{n} + \lambda(\mathbf{b} \cdot \mathbf{n}) = p \\ \iff & \lambda = \frac{p - \mathbf{a} \cdot \mathbf{n}}{\mathbf{b} \cdot \mathbf{n}} \end{aligned}$$

- We may use this result to determine whether a ray $\mathbf{r} = \mathbf{a} + \lambda\mathbf{b}$ intersects with a planar polygon with $m$ vertices $\mathbf{p}_1, \ldots, \mathbf{p}_2$ and a surface normal $\mathbf{n}$.

  Project 3D intersection $\mathbf{p}$ onto a 2D intersection $\mathbf{p}'$ by finding the 2D orthonormal basis $\{\hat{\mathbf{u}}, \hat{\mathbf{v}}\}$ of the plane. Determine number of intersections with

$$\mathbf{r} = \mathbf{p}' + \mu\hat{\mathbf{u}},$$

  and the edges of the polygon. Odd number of intersections $\implies$ $\mathbf{r}$ does intersect the polygon.

### 2.3.2 Sphere

- Suppose we have a ray $\mathbf{r} = \mathbf{a} + \lambda\mathbf{b}$ and the sphere $\|\mathbf{r} - \mathbf{c}\| = R$. We have

$$\begin{aligned} \|(\mathbf{a} + \lambda\mathbf{b}) - \mathbf{c}\| &= R \\ \iff ((\mathbf{a} + \lambda\mathbf{b}) - \mathbf{c}) \cdot ((\mathbf{a} + \lambda\mathbf{b}) - \mathbf{c}) &= R^2 \\ \iff (\mathbf{b} \cdot \mathbf{b})\lambda^2 + 2\mathbf{b} \cdot (\mathbf{a} - \mathbf{c})\,\lambda + (\mathbf{a} - \mathbf{c}) \cdot (\mathbf{a} - \mathbf{c}) - R^2 &= 0 \end{aligned}$$

  So we have a quadratic equation, with coefficients

$$\begin{aligned} A &= \mathbf{b} \cdot \mathbf{b} \\ B &= 2\mathbf{b} \cdot (\mathbf{a} - \mathbf{c}) \\ C &= (\mathbf{a} - \mathbf{c}) \cdot (\mathbf{a} - \mathbf{c}) - R^2 \end{aligned}$$

  Hence

$$\lambda_{1,2} = \frac{-B \pm \sqrt{\Delta}}{2A},$$

  where the discriminant $\Delta = B^2 - 4AC$ determines the number of intersection points.

## 2.4 Shading

- In the Phong shading model, we first have the following assumptions:
  - There is only Lambertian reflection, also known referred to as diffuse reflection.
  - All light falling on a given surface comes directly from a point light, there is no interaction between objects
  - No object casts shadows on each other, so we may consider each object individually.
- Each material / object in the scene, the following parameters are defined
  - $k_s$, the specular reflection constant. The ratio of reflection of the specular term of incoming light.

  - $k_d$, the diffuse reflection constant. The ratio of reflection of the diffuse term of incoming light.
  - $k_\alpha$, the ambient reflection constant, the ratio of reflection of the ambient term present in all points of the scene rendered.
  - $n$, the shininess constant of a material.

- Notation
  - lights, the set of light sources.
  - $\hat{\mathbf{L}}_i$, the direction vector from the point of the surface towards the $i$th light source.
  - $\hat{\mathbf{n}}$, the normal at the point on the surface.
  - $\hat{\mathbf{R}}_i$, the direction that a reflected ray of light would take from the point on a surface.
  - $\hat{\mathbf{v}}$, the direction pointing towards the viewer (or camera).

### 2.4.1 Lambertian Shading

- Consider some light ray from the $i$th with cross-sectional vector area $\mathbf{S}_i$.
- The area projected onto some surface with normal $\mathbf{n}$ is $\mathbf{S}'_A = (\mathbf{S}_A \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} = \|\mathbf{S}_A\| \cos\theta \hat{\mathbf{n}}$ where $\theta$ is the angle between the normal and $\hat{\mathbf{L}}_i$.
- **Energy conservation**: In a closed system, energy is conserved, hence the rate of energy transferred by the ray (power in) is equal to the rate of energy transferred by the ray onto the surface.
- The power of a electromagnetic wave is $P = IA$ where $I$ is the intensity and $A$ is the area.

  So

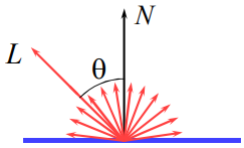
$$\begin{aligned} & I\|\mathbf{S}_A\| = I_i\|\mathbf{S}_A\| \cos\theta \\ \iff & I = I_i \cos\theta \end{aligned}$$

  where $I$ is the intensity of the reflected wave and $I_i$ is the intensity of the $i$th light source.

- $I_d \propto k_d$ and $I_d \propto I_i$. So the diffuse component is given by

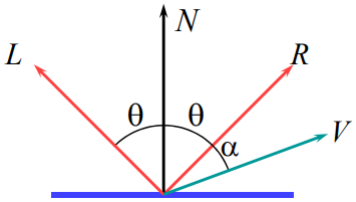
$$I_d = k_d I_i \cos\theta = k_d I_i (\hat{\mathbf{n}} \cdot \hat{\mathbf{L}}_i).$$

where $I_d$ is the intensity of the diffuse component.




### 2.4.2 Specular Reflection

- Approximation of imperfect specular reflection
- Consider




where $\alpha$ is the angle between $\hat{\mathbf{R}}_i$ and $\hat{\mathbf{v}}$.

- Maximum specular reflection is achieved when $\alpha = 0$ and decreases to zero as $a \to \pi/2$.
- Hence $I_s \propto \cos^n \alpha$, for sufficiently large $n$. Since $I_s \propto I_i, k_s, \cos^n \alpha$, then

$$I_s \propto I_i k_s \cos^n \alpha = I_i k_s (\hat{\mathbf{R}}_i \cdot \hat{\mathbf{v}})^n.$$

- $n$ may be considered as a "roughness" coefficient.

### 2.4.3 Ambient Illumination

- Since no interactions between other objects $\implies$ light reflected off all other surfaces onto a given surface can be approximated by a constant *ambient illumination* term.
- We define
$$I_a = I_\alpha k_\alpha,$$
where $I_\alpha$ is a ambient light intensity and $k_\alpha$ is the ambient reflection constant.

### 2.4.4 The Phong Shading Equation

So we have the following shading equation

$$I = I_\alpha k_a + \sum_{i \in \text{lights}} I_i (k_d (\hat{\mathbf{L}}_i \cdot \hat{\mathbf{n}}) + k_s (\hat{\mathbf{R}}_i \cdot \hat{\mathbf{v}})^n).$$

### 2.4.5 Shadows

- For shadows, we consider a *shadow ray* (a ray $\mathbf{p} + \lambda \hat{\mathbf{L}}_i$) If the shadow ray intersects with an object that is closer than the $i$th light, then $\mathbf{p}$ is in shadow.

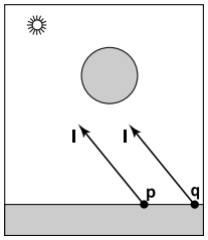



Figure 2.1: $\mathbf{q}$ is in shadow, $\mathbf{p}$ isn't

- **Problem**: Self intersection between the original surface and the shadow ray, due to imprecision of floating point arithmetic.
- **Solution**: Add a *shadow bias* $\epsilon$ to the origin of the shadow ray. So the origin of the shadow ray is now $\mathbf{p} + \epsilon \hat{\mathbf{L}}_i$.

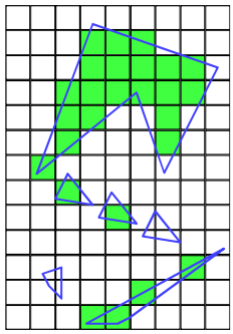
- We have

```
p is ray-intersection point
for (i ∈ lights) {
    d ← distance to i
    p' ← scene.closest_intersection((p + ε L̂_i) + λ L̂_i)
    if (|p'| < d) {
        // cast shadow
        continue
    }
    // shading model
}
```

## 2.5 Distribution Ray Tracing

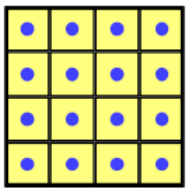
### 2.5.1 Anti-Aliasing

- In ray tracing, we assume that the ray passes through the center of the pixel.
- This can lead to jagged objects, small objects being missed completely or thin objects split into smaller pieces.



- (*Spatial*) Anti-aliasing is a technique for minimizing the distortion artifacts referred as aliasing.

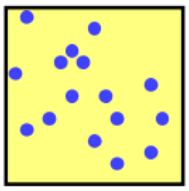
- Uses *super sampling*; color samples are taken at serval instances inside the pixel and an average color value is calculated.
- Assume bounds for the pixel $(x, y)$ in raster image is $[x, x+1] \times [y, y+1]$ (see section ??).
- Super sampling algorithms:
  - **Grid**: Simplest super sampling algorithm. Pixel is split into $n \times n$ sub-pixels. A sample is taken from the center of each.



$c \leftarrow 0$
`for` $(i \leftarrow 0;\ i < n;\ i$`++`$)$ `{`
    `for` $(j \leftarrow 0;\ j < n;\ j$`++`$)$ `{`
        $c \leftarrow c +$ `color`$\left(x + \frac{i+1/2}{n}, y + \frac{j+1/2}{n}\right)$
    `}`
`}`
$c_{x,y} \leftarrow c/n^2$

**Problem**: leads to noticeable aliasing unless $n$ is large.

  - **Random**: (Stochastic sampling). Pixel is sampled at $n$ random points. I

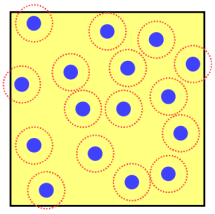


$c \leftarrow 0$
`for` $(i \leftarrow 0;\ i < n;\ i$`++`$)$ `{`

$c \leftarrow c + \texttt{color}(x + \text{random}([0,1)), y + \text{random}([0,1)))$
}
$c_{x,y} \leftarrow c/n$

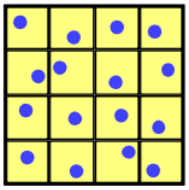
**Problem**: irregularity of the pattern, can introduce noise artefacts (although eye is less sensitive to noise than aliasing). Also can produce clusters $\implies$ useless rays.

– **Poisson disc**: Similar to random sampling, however, algorithm checks whether the randomly selected point is at least $\varepsilon$ from any other point.



**Problem**: very difficult to implement.

– **Jittered**: Modified grid implementation that approximates Poisson disc. A pixel is split into $n \times n$ sub-pixels. A sample is taken from a random point within the sub-pixel.

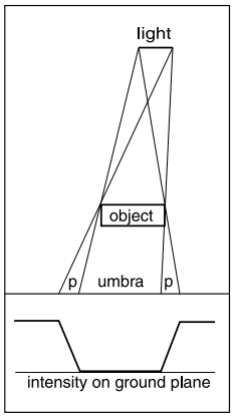


$c \leftarrow 0$
for $(i \leftarrow 0;\ i < n;\ i\texttt{++})$ {
    for $(j \leftarrow 0;\ j < n;\ j\texttt{++})$ {
        $c \leftarrow c + \texttt{color}\left(x + \frac{i+\xi_1}{n}, y + \frac{j+\xi_2}{n}\right)$
    }
}
$c_{x,y} \leftarrow c/n^2$

where $\xi_1, \xi_2$ are random$([0,1))$.

### 2.5.2 Soft Shadows

- We often model lights as points, however lights have non-zero area (irl).




- Region where light is entirely invisible is the *umbra*, the partially visible region (denoted by $p$) is the *penumbra*.
- Suppose a light surface is modelled by the implicit equation $f(\mathbf{r}) = 0$.
- We then select a random point on the light surface. e.g. for a plane

$$\mathbf{r} = \mathbf{a} + \xi_1 \mathbf{b} + \xi_2 \mathbf{b},$$

  where $\xi_1$ and $\xi_2$ are uniform random numbers.

- We have

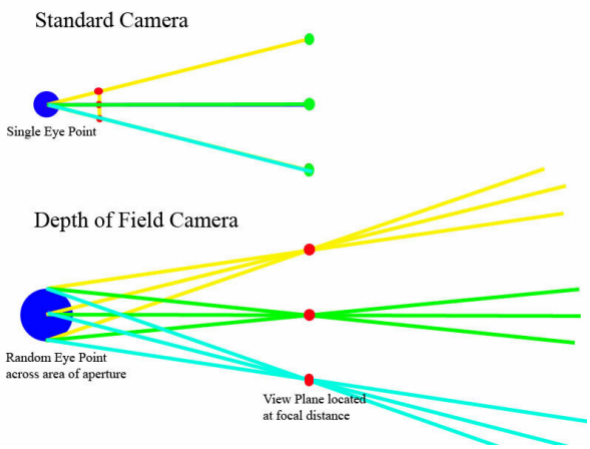
```
p is ray-intersection point
for (i ∈ lights) {
    occluded_count ← 0
    for (j ← 0; j < SHADOW_RAY_COUNT; j++) {
        l ← randomly selected point from f_i(r) = 0
        d ← ||l − p||
        L̂_s ← (l − p) /d
        p′ ← scene.closest_intersection((p + εL̂_s) + λL̂_s)
        if (||p′|| < d) {
            occluded_count++
```

```
            }
        }
        k ← 1 − occluded_count/SHADOW_RAY_COUNT
        // shading model
    }
```

where $k$ is the **proportion of light**. In the umbra, $k = 0$ and in the penumbra $k \in [0, 1)$. e.g. In the Phong shading model, $I_d$ and $I_s$ are both proportional to $k$.

### 2.5.3 Depth of Field

- Depth of field is simulated by using a camera with a *finite aperture*.
- We place a artificial lens (focal plane) at a distance $d_f$ from the finite aperture (the focal distance).




- We have

```
for (pixel (x, y) ∈ screen) {
    r is projected ray (see section ??)
    p is intersection between r and focal plane Π

    c ← 0
    for (i ← 0; i < DOF_RAY_COUNT; i++) {
```

```
            e′ random point on finite aperture
            d ← (p − e′)/‖p − e′‖
            c ← c + trace_ray(e′ + λd)
        }
        c_{x,y} ← c/DOF_RAY_COUNT
    }
```

# 3 Transformations

## 3.1 2D Linear Transformations

- Scale: Scale $x$ and $y$ by $\lambda_x$ and $\lambda_y$ respectively

$$\text{scale}(\lambda_x, \lambda_y) = \begin{bmatrix} \lambda_x & 0 \\ 0 & \lambda_y \end{bmatrix}.$$

- Shear: Has the effect of pushing things sideways by a factor of $\lambda$:

$$\text{shear-}x(\lambda) = \begin{bmatrix} 1 & \lambda \\ 0 & 1 \end{bmatrix}, \qquad \text{shear-}y(\lambda) = \begin{bmatrix} 1 & 0 \\ \lambda & 1 \end{bmatrix}.$$

  Alternatively, thought as a clockwise (or anticlockwise) rotation of the vertical (or horizontal) axis by $\phi$

$$\text{shear-}x(\phi) = \begin{bmatrix} 1 & \tan\phi \\ 0 & 1 \end{bmatrix}, \qquad \text{shear-}y(\phi) = \begin{bmatrix} 1 & 0 \\ \tan\phi & 1 \end{bmatrix}.$$

- Rotation: (anticlockwise)

$$\text{rotation}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}.$$

## 3.2 Composition and Decomposition of Transformations

- Compositional transformation of $\mathbf{S}$ followed by $\mathbf{R}$ is $\mathbf{M} = \mathbf{RS}$.
- Hermitian (symmetric real matrices) decomposition:

$$\mathbf{M} = \mathbf{SDS}^T,$$

where $\mathbf{S}$ is an orthogonal matrix, $\mathbf{D}$ is a diagonal matrix. Suppose $\mathbf{M}$ has eigenvalues and eigenvector pairs $(\lambda_1, \mathbf{x}_1), \ldots$

$$\mathbf{S} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \\ \downarrow & \downarrow & & \downarrow \end{bmatrix}.$$

$$\mathbf{D} = \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_n \end{bmatrix}.$$

In geometric terms $\mathbf{S}$ is a rotation and $\mathbf{D}$ is a scale by $\lambda_1, \ldots$.

- For non-symmetric square matrices, we have

$$\mathbf{M} = \mathbf{S}\mathbf{D}\mathbf{S}^{-1}.$$

## 3.3 3D Linear Transformations

- Scale:

$$\text{scale}(\lambda_x, \lambda_y, \lambda_z) = \begin{bmatrix} \lambda_x & 0 & 0 \\ 0 & \lambda_y & 0 \\ 0 & 0 & \lambda_z \end{bmatrix}.$$

- Shear:

$$\text{shear-}x(\lambda_y, \lambda_z) = \begin{bmatrix} 1 & \lambda_x & \lambda_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{shear-}y(\lambda_x, \lambda_z) = \begin{bmatrix} 1 & 0 & 0 \\ \lambda_x & 1 & \lambda_z \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{shear-}z(\lambda_x, \lambda_y) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \lambda_x & \lambda_z & 1 \end{bmatrix}$$

- Rotation:

$$\text{rotation-}x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \end{bmatrix}$$

$$\text{rotation-}y(\phi) = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix}$$

$$\text{rotation-}z(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Rotations about an arbitrary axis $\mathbf{a}$. Compute the orthonormal basis containing $\mathbf{a}$, $(\mathbf{u}, \mathbf{v}, \mathbf{w})$

$$\mathbf{w} = \frac{\mathbf{a}}{\|\mathbf{a}\|}.$$

Find $\mathbf{t}$ s.t $\mathbf{t}$ isn't co-linear to $\mathbf{w}$

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{\|\mathbf{t} \times \mathbf{w}\|}.$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}.$$

Then a rotation that maps $\mathbf{u}$ to $\hat{\mathbf{i}}$, etc is

$$\mathbf{R}_{uvw} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}.$$

Rotation of $\phi$ about $\mathbf{a}$ is

$$\mathbf{M} = \mathbf{R}_{uvw}^T \cdot \text{rotate-}z(\phi) \cdot \mathbf{R}_{uvw}.$$

## 3.4 Transforming Normal Vectors

- Transformation by non-orthogonal matrices doesn't preserve angles.

- Let $\mathbf{M}$ be our transformation matrix, $\mathbf{n}$ be the normal prior to transformation. Let $\mathbf{N}$ be the normal transformation matrix and $\mathbf{n}_N = \mathbf{N}\mathbf{n}$. Let $\mathbf{t}$ be the surface tangent s.t

$$\mathbf{n} \cdot \mathbf{t} = \mathbf{n}^T\mathbf{t} = \mathbf{0}.$$

So

$$\begin{aligned} \mathbf{n}^T\mathbf{t} = \mathbf{n}^T\mathbf{M}^{-1}\mathbf{M}\mathbf{t} &= \mathbf{0} \\ = (\mathbf{n}^T\mathbf{M}^{-1})\mathbf{t}_M &= \mathbf{0} \end{aligned}$$

Hence

$$\begin{aligned} \mathbf{n}_N^T &= \mathbf{n}^T\mathbf{M}^{-1} \\ \iff \mathbf{n}_N &= (\mathbf{n}^T\mathbf{M}^{-1})^T \\ &= (\mathbf{M}^{-1})^T\mathbf{n} \end{aligned}$$

So $\mathbf{N} = (\mathbf{M}^{-1})^T$

## 3.5 Homogenous Coordinates

- Translations cannot be represented using cartesian coordinates.
- Use *homogenous coordinates*:

$$\begin{aligned} (x, y, w) &\equiv \left(\frac{x}{w}, \frac{y}{w}\right) \\ (x, y, z, w) &\equiv \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right) \end{aligned}$$

- So for 2D position vectors $(x, y)$ use their homogenous form $(x, y, 1)$.

$$\text{translate}(x_t, y_t) = \begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix}.$$

- Note that for displacement vectors $(x, y)$ use their homogenous form $(x, y, 0)$. Since displacement vectors aren't affected by translations.

## 3.6 Viewing Transformations

- We define transformations that project 3D points in the world space to 2D points in the image space.
- Most graphics systems split this up into:
  - A *eye transformation* which places the eye at the origin, the **view space**
  - A *projection transformation* which projects points from the view space so that all visible points fall in the ROI $[-1,1] \times [-1,1]$.
  - A *viewport transformation* maps the unit image rectangle $[-1,1]^2$ to the raster image domain $[0, n_x - 1] \times [0, n_y - 1]$ (or $[-1/2, n_x - 1/2] \times [-1/2, n_y - 1/2]$ due to centered pixel coordinates).

### 3.6.1 The Eye Transformation

- Define the viewer using the following parameters:
  - **eye**, the position of the eye.
  - **at**, the reference point which the eye looks at,
  - **up**, the view up vector describes where the top of the eye points.
- Define the **view space** (**eye**, $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, $\hat{\mathbf{w}}$) as

$$\hat{\mathbf{w}} = \frac{\mathbf{eye} - \mathbf{at}}{\|\mathbf{eye} - \mathbf{at}\|}$$
$$\hat{\mathbf{u}} = \frac{\mathbf{up} \times \hat{\mathbf{w}}}{\|\mathbf{up} \times \hat{\mathbf{w}}\|}$$
$$\hat{\mathbf{v}} = \hat{\mathbf{w}} \times \hat{\mathbf{u}}$$

- Hence the eye is looking in the $-\hat{\mathbf{k}}$ direction (after the change of space).
- So we have

$$\mathbf{M}_{eye} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_{eye} \\ 0 & 1 & 0 & -y_{eye} \\ 0 & 0 & 1 & -z_{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### 3.6.2 The Projection Transformation

- **Parallel Projection**: the identity matrix $\mathbf{M}_{orth}$ is used for $\mathbf{M}_{per}$.
- **Perspective Projection**: we define a view frustum with distance $n$ from the origin (**eye**).
- Use a non-unit homogenous coordinates $\begin{bmatrix} x & y & z & w \end{bmatrix}^T$. This allows for two homogenous vectors to specify the same point:

  $$\forall \lambda \neq 0 . \mathbf{x} \sim \lambda \mathbf{x},$$

  where $\sim$ denotes an equivalence relation between two homogenous vectors iff they represent the same cartesian point.

  IMAGE HERE

  So we have

  $$\begin{aligned} x' &= x\frac{d}{z} \\ y' &= y\frac{d}{z} \\ z' &= d \end{aligned}$$

  Note that

  $$\begin{bmatrix} x' \\ 1 \end{bmatrix} \sim \begin{bmatrix} d & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ z \\ 1 \end{bmatrix}$$

- Hence

  $$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

  This maps the view frustum to a orthographic projection space, a $[l, r] \times [b, t] \times [f, n]$ box. So

  $$\mathbf{M}_{per} = M_{orth}\mathbf{P}.$$

- We may define $(l, r, b, t)$ and $n$ using a "field-of-view" angle $\phi$. and a aspect ratio *aspect*. In this context, the camera is aligned to the center of the near plane:

$$l = -r$$
$$b = -t$$

- We define the aspect ratio as

$$aspect = \frac{n_x}{n_y} = \frac{2r}{2t} = \frac{r}{t}.$$

- The fov angle must satisfy

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

- For a more simple view (or focal) plane with distance $d$, we have

$$\mathbf{M}_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/d \\ 0 & 0 & 1/d & 0 \end{bmatrix}.$$

### 3.6.3 The Viewport Transformation

- For an image in canonical projection space i.e. $[-1, 1]^3$, we have a camera looking in the $-\hat{\mathbf{k}}$ direction.
- More generally: translate the rectangle $[x_l, x_h] \times [y_l, y_h]$ to $[x'_l, x'_h] \times [y'_l, y'_h]$. We split this up into the following sequence of three operations:
  1. Move $(x_l, y_l)$ to $(0, 0)$
  2. Scale the rectangle
  3. Move the origin to $(x'_l, y'_l)$.

$$\begin{aligned}\mathbf{M} &= \text{translate}(x'_l, y'_l) \cdot \text{scale}\left(\frac{x'_h - x'_l}{x_h - x_l}, \frac{y'_h - y'_l}{y_h - y_l}\right) \cdot \text{translate}(-x_l, -y_l) \\ &= \begin{bmatrix} 1 & 0 & x'_l \\ 0 & 1 & y'_l \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & 0 \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_l \\ 0 & 1 & -y_l \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & \frac{x'_h x_h - x'_l x_l}{x_h - x_l} \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & \frac{y'_h y_h - y'_l y_l}{y_h - y_l} \\ 0 & 0 & 1 \end{bmatrix}\end{aligned}$$

- Hence

$$\mathbf{M}_{vp} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x - 1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y - 1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

**Orthographic Projection Transformation**

- The orthographic projection space is $[l, r] \times [b, t] \times [f, n]$ with a camera looking along $-\hat{\mathbf{k}}$. where

$$\begin{aligned} x = l &\equiv \text{left plane} & x = r &\equiv \text{right plane} \\ y = b &\equiv \text{bottom plane} & y = t &\equiv \text{top plane} \\ z = n &\equiv \text{near plane} & z = f &\equiv \text{far plane} \end{aligned}$$

So

$$\begin{aligned}\mathbf{M}_{orth} &= \text{translate}(-1, -1, -1) \cdot \text{scale}\left(\frac{2}{r - l}, \frac{2}{t - b}, \frac{2}{n - f}\right) \cdot \text{translate}(-l, -b, -n) \\ &= \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-n} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}\end{aligned}$$

# 4 The Graphics Pipeline

- Ray tracing is computationally expensive. Real time applications use *rasterization*, an *object-order rendering* technique that considers pixels influenced by an object, instead of objects that influence a pixel.
- The sequence of operations required to process objects to images is known as a *graphics pipeline*. IMAGE

## 4.1 Vertex Shading

- The vertex shading or *vertex-processing* stage is one of the programmable stages of the graphics pipeline.
- It operates on every vertex vertex and performs various operations on the vertex, such as applying transformations:
  - **World Transformation**. The transformation where an object is translated from object (or local) space to world space.
  - **Viewing Transformations**. See section ??

  The vertex shader is also responsible for calculating transformed vertex normals and *uv* coordinates.

## 4.2 Primitive Assembly

- The vertex shader passes the view-space vertices to the next stage, the primitive assembler.
- The primitive assembler combines the vertices into a sequence of *primitivies*.
- The primitives are classified into points, lines and triangles.

## 4.3 Clipping and Culling

### 4.3.1 Clipping

- Before rasterizing, we need to remove primitives that are outside the view volume (i.e. primitives behind the eye)
- **Solution**: Using six planes $S_{planes}$ that form the view frustum, do

```
for (Π ∈ S_planes) {
    if (triangle entirely outside Π) {
        break (triangle not visible)
    } else if (triangle intersects Π) {
        clip
        split remainder into triangles
    }
}
```

- We determine the equations of the planes using $l, r, t, b, n, f$ (or $n, f, \phi$).

$$\begin{aligned}
\Pi_n &: \mathbf{r} \cdot \hat{\mathbf{k}} = n \\
\Pi_f &: \mathbf{r} \cdot \hat{\mathbf{k}} = f \\
\Pi_l &: \mathbf{r} \cdot \left[\hat{\mathbf{j}} \times (l, b, n)\right] = 0 \\
\Pi_r &: \mathbf{r} \cdot \left[(r, b, n) \times \hat{\mathbf{j}}\right] = 0 \\
\Pi_t &: \mathbf{r} \cdot \left[\hat{\mathbf{i}} \times (l, t, n)\right] = 0 \\
\Pi_b &: \mathbf{r} \cdot \left[(l, b, n) \times \hat{\mathbf{i}}\right] = 0
\end{aligned}$$

  (in view space, multiply by $\mathbf{M}_{eye}^{-1}$ to get world space planes).

- Recall that the implicit equation of a plane through $\mathbf{a}$ w/ normal $\mathbf{n}$ is

$$f(\mathbf{r}) = (\mathbf{r} - \mathbf{a}) \cdot \mathbf{n} = 0.$$

- A triangle $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ intersects the plane iff one of the edges $\mathbf{v}_0\mathbf{v}_1, \mathbf{v}_1\mathbf{v}_2, \mathbf{v}_2\mathbf{v}_0$ intersects the plane $\Pi$. Note that an edge $\mathbf{uv}$ intersects $\Pi$ iff $f(\mathbf{u})$ and $f(\mathbf{v})$ have different signs i.e. $f(\mathbf{u})f(\mathbf{v}) < 0$.

- If the edge intersects the plane, then the intersection point $\mathbf{p}$ is

$$\begin{aligned} \mathbf{p} &= \mathbf{u} + \lambda(\mathbf{v} - \mathbf{u}) \\ f(\mathbf{p}) = [\mathbf{u} + \lambda(\mathbf{v} - \mathbf{u}) - \mathbf{a}] \cdot \mathbf{n} &= 0 \\ \lambda &= \frac{(\mathbf{a} - \mathbf{u}) \cdot \mathbf{n}}{(\mathbf{v} - \mathbf{u}) \cdot \mathbf{n}} \end{aligned}$$

### 4.3.2 Culling

- Culling is the process of removing hidden primitives from the scene.
- *Back face culling* is the process of removing primitives facing away from the eye. Recall that triangles are counter clockwise wound, that is

$$\text{Orient}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2) \geq 0.$$

  However, on the back face, triangles are clockwise wound, hence

$$\text{Orient}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2) \leq 0.$$

  So if $\text{Orient}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2) \leq 0$, we can remove the primitive

## 4.4 Rasterization

- A rasterizer takes a collection of *primitives* and enumerates the pixels covered by the primitive and *interpolates* values (attributes) across the primitive (e.g color).
- A set of *fragments* (one each pixel covered by the primitive) is outputted.
- Most rasterizers use line segments and triangles as *primitives* since surfaces can be modelled as *polyhedra* (meshes of triangles) since a triangle is planar (and has numerous helpful properties).

### 4.4.1 Triangles

**Definition 4.4.1.** (**Half Space**) An edge $v_0v_1$ divides the plane into two halves, referred to a *half spaces* IMAGE

IMAGE

- Note that
  - The green is the positive half space of $v_1v_2$, $v_2v_0$ and the negative half space of $v_0v_1$.
  - The cyan is the positive half space of $v_2v_0$ and the negative half space of $v_0v_1, v_1v_2$.
  - The blue is the positive half space of $v_0v_1, v_2v_0$ and the negative half space of $v_1v_2$.
  - The magenta is the positive half space of $v_0v_1$ and the negative half space of $v_1v_2, v_2v_0$.
  - The red is the positive half space of $v_0v_1, v_1v_2$ and the negative half space of $v_2v_0$.
  - The yellow is the positive half space of $v_1v_2$ and the negative half space of $v_0v_1, v_2v_0$.
  - The grey is the positive half space of all three edges.
- So a point $(x, y)$ lies in triangle iff point lies in all three positive half spaces of the edges.
- Let $\Delta(v_0, v_1, v_2)$ denote the area of the triangle $v_0v_1v_2$, so

$$\Delta(v_0, v_1, v_2) = \frac{1}{2}\begin{vmatrix} v_0^x & v_1^x & v_2^x \\ v_0^y & v_1^y & v_2^y \\ 1 & 1 & 1 \end{vmatrix} = \frac{1}{2}\begin{vmatrix} v_1^x - v_0^x & v_2^x - v_0^x \\ v_1^y - v_0^y & v_2^y - v_0^y \end{vmatrix}.$$

  Note the use of homogenous coordinates. Follows from the fact that a 2D determinant is the signed area of a parallelogram.
- If $\Delta(v_0, v_1, v_2) > 0 \implies v_2$ lies in the positive half space of $v_0v_1$.
- Any non-cyclic permutation of $v_0, v_1, v_2$ yields a negative $\Delta(v_0, v_1, v_2)$.

**Definition 4.4.2.** (**Orientation**) The orientation of a triangle $v_0v_1v_2$ is given by

$$\text{Orient}(v_0, v_1, v_2) = \begin{vmatrix} v_0^x & v_1^x & v_2^x \\ v_0^y & v_1^y & v_2^y \\ 1 & 1 & 1 \end{vmatrix} = 2\Delta(v_0, v_1, v_2).$$

**Idea**: orientation encodes clockwise or anticlockwise direction of edges $v_0v_1v_2$.

- To determine if $p$ lies in the triangle, consider the point in relation to some edge. e.g. $v_0v_1$.

  $$\text{Orient}(v_0, v_1, p) = \begin{vmatrix} v_1^x - v_0^x & p_x - v_0^x \\ v_1^y - v_0^y & p_y - v_0^y \end{vmatrix} = (v_0^y - v_1^y)p_x + (v_1^x - v_0^x)p_y + (v_0^x v_1^y - v_0^y v_1^x).$$

  If $\text{Orient}(v_0, v_1, p) > 0 \implies p$ is in positive half space of $v_0v_1$.

**Definition 4.4.3.** (**Edge Functions**) For the triangle $v_0v_1v_2$, we define the edge functions for $v_0v_1, v_1v_2$ and $v_2v_0$ as

$$\begin{aligned} F_{01}(p) &= (v_0^y - v_1^y)p_x + (v_1^x - v_0^x)p_y + (v_0^x v_1^y - v_0^y v_1^x) \\ F_{12}(p) &= (v_1^y - v_2^y)p_x + (v_2^x - v_1^x)p_y + (v_1^x v_2^y - v_1^y v_2^x) \\ F_{20}(p) &= (v_2^y - v_0^y)p_x + (v_0^x - v_2^x)p_y + (v_2^x v_0^y - v_2^y v_0^x) \end{aligned}$$

respectively.

- If $F_{01}(p), F_{12}(p), F_{20}(p) > 0 \implies p$ is inside the triangle since the edge functions relate to the half spaces (noted above).

- Also note that

  $$F_{01}(p) + F_{12}(p) + F_{20}(p) = 2\Delta(v_0, v_1, v_2).$$

  since $p$ partitions the triangle into 3 smaller triangles. The edge functions are nothing but $2\times$ the signed area of these triangles $\implies$ sum of edges functions is $2\times$ area of the triangle $v_0v_1v_2$. IMAGE

**Definition 4.4.4.** (**Barycentric Coodinates**) The Barycentric coordinates of a point $\mathbf{p}$ are $(\alpha, \beta, \gamma)$ such that

$$\begin{aligned} \mathbf{p} &= \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) = (1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \\ &= \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \end{aligned}$$

where $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are the edges of the triangle.

- So Barycentric coordinates are a non-orthogonal coordinate system. with

  $$\alpha + \beta + \gamma = 1.$$

  where $(\alpha, \beta, \gamma)$ are the signed scale distances from $v_1v_2, v_2v_0$ and $v_0v_1$ to $p$ respectively.

- So $\mathbf{p}(1,0,0) = \mathbf{a}, \mathbf{p}(0,1,0) = \mathbf{b}$ and $\mathbf{p}(0,0,1) = \mathbf{c}$.
- To calculate $(\alpha, \beta, \gamma)$, note that for $v_0, v_1$ on $v_0v_1$, the edge function is zero
$$F_{01}(v_0) = F_{01}(v_2) = 0.$$
and for $v_2$
$$F_{01}(v_2) = 2\Delta(v_0, v_1, v_2).$$
We deduce that
$$\alpha(p) = \frac{F_{12}(p)}{2\Delta(v_0, v_1, v_2)}$$
$$\beta(p) = \frac{F_{20}(p)}{2\Delta(v_0, v_1, v_2)}$$
$$\gamma(p) = \frac{F_{01}(p)}{2\Delta(v_0, v_1, v_2)}$$

### 4.4.2 Line Rasterization

- Consider drawing line between coordinates $(x_0, y_0)$ and $(x_1, y_1)$, where $x_0 \leq x_1$.
- Implicit equation of the line
$$f(x,y) = (y_1 - y_0)x - (x_1 - x_0)y + (x_0y_1 - x_1y_0),$$
and slope
$$m = \frac{y_1 - y_0}{x_1 - x_0}.$$
- Consider case $0 < m \leq 1$.
- Start at $(x, y) = (x_0, y_0)$, so $f(x_0, y_0) = 0$.
- Increment $(x, y)$ to either $(x + 1, y)$ or $(x + 1, y + 1)$.
- Increment dependent on whether $err = f(x + 1, y + 1/2)$. If $err > 0 \implies$ midpoint is in positive half space, hence $(x + 1, y)$ is optimal (vice versa).

```
y = y_0
for (x ← x_0; x ≤ x_1; x++) {
    draw(x, y)
    if (f(x + 1, y + 1/2) < 0) {
        y++
    }
}
```

- Note that

$$f(x+1,y) = f(x,y) + (y_0 - y_1)$$
$$f(x+1,y+1) = f(x,y) + (y_0 - y_1) + (x_1 - x_0)$$

So faster implementation:

```
y = y_0
err = f(x_0 + 1, y_0 + 1/2)
for (x ← x_0; x ≤ x_1; x++) {
    draw(x, y)
    if (d < 0) {
        y++
        d ← d + (x_1 − x_0) + (y_0 − y_1)
    } else {
        d ← d + (y_0 − y_1)
    }
}
```

- Other cases are essentially identical.

### 4.4.3 Triangle Rasterization

- Suppose we have a triangle with vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$.
- We can edge functions (or Barycentric Coordinates) to determine whether a given point $\mathbf{p}$ lies in the triangle.

```
for (pixel (x, y) ∈ screen) {
    compute Barycentric Coordinates (α, β, γ) for p = (x, y)
    if (α, β, γ ∈ [0, 1]) {
```

```
            // optional c ← α c_0 + β c_1 + γ c_2
            draw(x, y) // or draw(x, y, c)
        }
    }
```

where $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2$ are the colors of the triangle vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$ respectively. This technique of interpolating color is known as "Barycentric Interpolation".

- **Improvement**: By finding the bounding rectangle of the triangle; and looping over each pixel in the bbox.

```
x_min ← floor(min(v_i^x)); x_max ← ceil(max(v_i^x))
y_min ← floor(min(v_i^y)); y_max ← ceil(max(v_i^y))

for ((x, y) ∈ [x_min, x_max] × [y_min, y_max]) {
    compute Barycentric Coordinates (α, β, γ) for p = (x, y)
    if (α, β, γ ∈ [0, 1]) {
        c ← α c_0 + β c_1 + γ c_2
        draw(x, y, c)
    }
}
```

- **Problem**: Usually we are rasterizing triangles that share vertices and edges. We could draw edges using line rasterization to prevent holes, but if two adjacent triangles have different colors, the image will depends on the order of rasterization since each triangle will be drawing over the same pixels (the shared edges).

- **Solution**: Use the convention that pixels are drawn iff the pixel center lies inside the triangle $\implies$ barycentric coordinates $(\alpha, \beta, \gamma)$ satisfies: $\alpha, \beta, \gamma \in (0, 1)$. Since $\alpha+\beta+\gamma = 1 \implies \alpha, \beta, \gamma \in (0, 1) \iff \alpha, \beta, \gamma > 0$.

**Triangle Edge Rasterization**

- **Problem**: Previous solution doesn't deal w/ triangle edges, just solves the order problem.

- **Solution**: Observe that an arbitrary off-screen point $\mathbf{x}$ lies on one half space of the shared edge (provided the shared edge doesn't pass through $\mathbf{x}$). Draw the edge of the triangle that lies in the same half space of $\mathbf{x}$.
- Note that two non-overlapping triangles, the vertices $V = \{\mathbf{y}_1, \mathbf{y}_2\}$ not on the shared edge are on opposite sides. Hence only one vertex in $V$ is on the same side as $\mathbf{x}$.
- Recall that $\text{Orient}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{p}) > 0 \implies \mathbf{p}$ is in positive half space of $\mathbf{v}_0\mathbf{v}_1$. Hence $\mathbf{x}$ and $\mathbf{y}$ lie in the same half space iff $\text{Orient}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{x})\text{Orient}(\mathbf{v}_0, \mathbf{v}_1, \mathbf{y}) > 0$

$x_{min} \leftarrow \text{floor}(\min(\mathbf{v}_i^x))$; $x_{max} \leftarrow \text{ceil}(\max(\mathbf{v}_i^x))$
$y_{min} \leftarrow \text{floor}(\min(\mathbf{v}_i^y))$; $y_{max} \leftarrow \text{ceil}(\max(\mathbf{v}_i^y))$
`select arbitrary off-screen point x`
$f_\alpha \leftarrow F_{12}(\mathbf{v}_0)$; $f_\beta \leftarrow F_{20}(\mathbf{v}_1)$; $f_\gamma \leftarrow F_{01}(\mathbf{v}_2)$

```
for ((x, y) ∈ [x_min, x_max] × [y_min, y_max]) {
    compute (α, β, γ) for p = (x, y)
    if (α, β, γ ≥ 0) {
        if ([(α > 0 ∨ f_α F_12(x) > 0) ∧ (β > 0 ∨ f_β F_20(x) > 0)
            ∧ (γ > 0 ∨ f_γ F_01(x) > 0)]) {
            c ← αc_0 + βc_1 + γc_2
            draw(x, y, c)
        }
    }
}
```

### 4.4.4 The Z-Buffer Algorithm

- **Problem**: Occlusions
- **Solution**: Draw primitives from back-to-front (the painters algorithm).
- **Problem**: Primitives can form *occlusion cycles*, so a back-to-front ordering doesn't exist. And sorting primitives by depth is slow. IMAGE
- **Solution**: We use the *z-buffer* algorithm. At each pixel we store the closest distance (or depth $z$) in a depth buffer (or $z$-buffer).

```
for ((x, y) ∈ screen) z-buffer(x, y) ← f

for (triangle t ∈ scene) {
    for (fragment (x, y) ∈ t) {
        calculate depth z
        if (z < z-buffer(x, y)) {
            z-buffer(x, y) ← z
            draw(x, y, f.c)
        }
    }
}
```

- After rasterization, we have a set of fragments. We interpolate the per-vertex attributes such as colors, normals and texture $uv$ coordinates to determine the per-fragment attributes. (Barycentric interpolation).
- These per-fragment attributes are then passed to the fragment shader.

## 4.5 Fragment Shading

- The fragment shader (or *fragment-processing stage*) is the 2nd programmable stage of the graphics pipeline.
- It operations on every fragment produced by the rasterizer and deals with texturing and lighting. The inputs of the fragment shader consist of:
  - fragment variable attributes, such as normals, colors, texture $uv$ coordinates.
  - uniforms.
  - samplers (for textures).
- For lighting, we use the Phong shading model.

# 5 OpenGL

- OpenGL is a graphics API:
  - Platform-independent
  - Open source
  - Focuses on general 3D applications
  - OpenGL driver manages graphics hardware and resources.
- DirectX (Microsoft) graphics API:
  - Windows based
  - Proprietary
  - Focuses on games.
  - Application manages resources.
- Vulkun
  - Platform-independent
  - Open source
  - Reduces CPU usage and better support for multi-core CPUs
  - Lower level control of GPU

## 5.1 OpenGL Data Structures

### 5.1.1 Shapes

- A vertex is a coordinate or a point.
- OpenGL has three *modes* indicating how vertices:

  - Each vertex is an independent point `GL_POINTS`,
  - the vertices represent a line `GL_LINES`,
  - the vertices represent a triangle `GL_TRIANGLES`

- OpenGL triangles $v_1v_2v_3$ are specified in a *counter-clockwise* order.

```
float triangle[] = {
    1.0f, 1.0f, // Top
    0.0f, 0.0f, // Left
    1.0f, 0.0f, // Right
};
```

### 5.1.2 Buffers

- A buffer is an OpenGL object that manages a address space in GPU memory in an unformatted way.
- A buffer is interpreted based on the bound *buffer target*; `GL_ARRAY_BUFFER` and `GL_ELEMENT_ARRAY_BUFFER`. This provides a specification for reconstructing the formatted data.
- **Vertex Array**. A vertex array is an OpenGL object which contains one or more buffers, designed to store information for a rendered object. e.g. contains vertex buffer, normal buffer, color buffer and an index buffer.
- **Vertex Buffer**. A vertex buffer stores a list of vertex coordinates. The vertex buffer is an `GL_ARRAY_BUFFER`. e.g. `triangle[]`.
- **Index Buffer**. An index buffer stores a list of indices, which reference various array buffers (e.g. vertex, color, normal). It is an instance of `GL_ELEMENT_ARRAY_BUFFER`.

  Suppose we have $n$ triangles (`GL_TRIANGLES`). Note that for two adjacent triangles, an edge $v_1v_2$ is shared. Hence for $n$ triangles, we have $n-1$ shared edges.

  Without using index buffers (i.e. using an vertex array), we'd have $3n$ vertices stored. However, since triangles share vertices, we can use a

index buffer to store $3n - 2(n-1) = n + 2$ vertices, and use $3n$ indices stored in `index_buffer` to construct the triangle mesh, where each element $i \in$ `index_buffer` relates to the vertex `vertices`$[i]$.

DIAGRAM HERE

- To create and bind buffers:

```
uint64_t buffers[2];
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, &buffer[0]); // vertex buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
    vertices, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, &buffer[1]); // index buffer
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices),
    indices, GL_STATIC_DRAW);
```

## 5.2 Shaders

- Shaders are programmable stages, written in GLSL (OpenGL Shading Language) in the OpenGL graphics pipeline, that are compiled into GPU-native instructions.
- See section ?? for theory.

### 5.2.1 Shader Variables

- **In**. Variables declared with the `in` qualifer are shader input variables. These variables are given values by the previous stage in the graphics pipeline. However, they may be set by the OpenGL API.
- **Out**. Variables declared with the `out` qualifier are shader output variables. These variables are passed onto the next stage of the pipeline.

- **Uniform**. A uniform is a global shader variable declared with the `uniform` qualifier. They do not change from one shader invocation to the next within a rendering call. They are set by the OpenGL API.

```
#version 140
in vec3 oc_position;
in vec3 oc_normal;

output vec3 wc_frag_normal;
output vec3 wc_frag_position;

uniform mat4 model_matrix;
uniform normal_matrix;
uniform mat4 mvp_matrix;

void main() {
    wc_frag_position = vec3(model_matrix * vec4(oc_position, 1.0));
    wc_frag_normal = normalize(normal_matrix * oc_normal);

    gl_Position = mvp_matrix * vec4(oc_position, 1.0);
}
```

DIAGRAM

### 5.2.2 Vertex Shaders

- The vertex shader is executed once per vertex in a triangle mesh.
- The shader operates on the built-in `gl_Position` variable, which defines the `vec4` homogenous coordinate for the vertex.

```
#version 140

in vec4 pos;

void main() {
    gl_Position = pos;
}
```

- The shader applies various transforms on the vertex and it's attributes e.g. transforming normals, calculating $uv$ texture coordinates.

### 5.2.3 Fragment Shaders

- The fragment shader is executed once per a fragment (a potential pixel).
- The shader has two important tasks: texturing and lighting. The shader operates on the built-in `gl_FragColor` variable, defined in RGBA (RGB and $\alpha$, which controls transparency).

```
#version 140

uniform sampler2D tex;
in vec2 uv_texCoord;

void main() {
    gl_FragColor = texture2D(tex, uv_texCoord);
}
```

- Common design method: Use rasterization to determine fragments, and then ray-tracing based illumination for lighting e.g. phong shading model.

## 5.3 Textures

**Definition 5.3.1.** (**Texture Mapping**) Texture mapping is the process of projecting an function / image onto the surface of a shape or polygon.

- The function / image is referred to as a *texture map*.
- A texture map may be a raster image or function (for procedural textures).

### 5.3.1 2D Textures

- In 2D textures, we use $uv \in [0,1]^2$ texture coordinates.
- $uv$ coordinates are assigned to vertices of the polygon mesh via a process known as **surface parameterization**.
- Once parameterized, we can sample the texture map using these $uv$ coordinates.

#### Parameterization of a Sphere

To map a $(u,v) \in [0,1]^2$ texture map onto a sphere, we compute the polar coordinates. So for a sphere with center $\mathbf{c}$ and radius $r$,

$$\begin{aligned} x &= \mathbf{c}_x + r\sin\theta\cos\phi \\ y &= \mathbf{c}_y + r\sin\theta\sin\phi \\ z &= \mathbf{c}_z + r\cos\theta \end{aligned}$$

Hence $(\theta, \phi)$ are given by

$$\begin{aligned} \theta &= \arccos\left(\frac{z-\mathbf{c}_z}{r}\right) \\ \phi &= \arctan\left(\frac{y-\mathbf{c}_y}{x-\mathbf{c}_x}\right) \end{aligned}$$

Since $(\theta,\phi) \in [0,\pi] \times [0,2\pi]$, we convert to $(u,v)$ as follows:

$$\begin{aligned} u &= \frac{\phi}{2\pi} \\ v &= \frac{\pi-\theta}{\pi} \end{aligned}$$

#### Parameterization of Rasterized Triangles

- For surfaces represented by triangle meshes, texture coordinates are defined by storing $(u,v)$ coordinates at each vertex of the mesh (in `GL_ARRAY_BUFFER`).

- $uv$ coordinates are then interpolated using Barycentric interpolation. Recall that for a point $\mathbf{p}$ with Barycentric coordinates $(\alpha, \beta, \gamma)$,

$$\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}),$$

  A similar equation is applied to $(u, v)$:

$$u = u_a + \beta(u_b - u_a) + \gamma(u_c - u_a)$$
$$v = v_a + \beta(v_b - v_a) + \gamma(v_c - v_a)$$

- **Problem**: Barycentric interpolation in screen space (not world space coordinates) yields incorrect results.

- **Solution**: For each screen space coordinate $\mathbf{s}$, compute the world space coordinate $\mathbf{p}$ then interpolate using $\mathbf{p}$ (not $\mathbf{s}$). This can be done by computing $\mathbf{M} = (\mathbf{M}_{vp}\mathbf{M}_{per}\mathbf{M}_{eye})^{-1}$ and applying it to $\mathbf{s}$ to yield $\mathbf{p}$.

- **Texture Tiling**: A common parameterization for texture tiling (repetitive patterns) is to parameterize the surface $S$ and then use the decimal/fractional part of the $uv$ coordinates as the $uv$ coordinates e.g. $\mathbf{c}(0.0, 0.0) = \mathbf{c}(1.0, 0.0)$

### 5.3.2 3D Textures

- In 3D textures, we simply use the vertex position/fragment position $\mathbf{p}$ to determine the color.

#### Procedural Textures

- 3D procedural textures are functions $f : \mathbb{R}^3 \to C$, where $C$ is some color space.

- A common example is the stripe texture:

```
vec3 stripe(vec3 p, float w) {
    if (sin πp_x/w > 0) {
        return c_0
    } else {
        return c_1
    }
```

```
}

// Linear interpolation variant
vec3 stripe(vec3 p, float w) {
    float t ← (1+sin(π p_x / w)) / 2
    return (1 − t)c_0 + t c_1
}
```

where $w$ is the width of the stripes.

**Texture Arrays**

- A texture array is an array of 2D raster image textures.
- Suppose we have $n_z$ 2D raster image textures, each having $n_x \times n_y$ texels. The texel position for the point $\mathbf{p} = (u, v, w) \in [0, 1]^3$ is given by
$$(i, j, k) = (\lfloor un_x \rfloor, \lfloor vn_y \rfloor, \lfloor wn_z \rfloor).$$
IMAGE
- Trilinear interpolation is used to interpolate colors.

**Cube Maps**

- A cube map is a texture consisting of 6 individual 2D raster image textures, each form one side of a cube.
- The texture is sampled using a direction vector $\mathbf{p}$.

IMAGE

### 5.3.3 Sampling and Mipmaps

**Definition 5.3.2.** (**Up Sampling**) Up sampling is the technique for sampling a texture where there are more fragments than texels in the texture.

IMAGE

- When up sampling, color/texel values must be interpolated using *nearest neighbor* or bilinear interpolation.

- **Nearest Neighbor**: Selects the nearest texel to the fragment's $uv$ position. Doesn't consider the values of neighboring texels. Produces block artifacts due to piecewise nature.

- **Bilinear Interpolation**: An extension of linear interpolation in which the two axes $u$ and $v$ are interpolated independently. Consider the color $\mathbf{c}(u, v)$. Assume we have the 4 neighbors $\mathbf{x}_1 = (u_1, v_1), \mathbf{x}_2 = (u_1, v_2), \mathbf{x}_3 = (u_2, v_1), \mathbf{x}_4 = (u_2, v_2)$. Interpolating in the $u$ direction yields

$$\mathbf{c}(u, v_1) = \frac{1}{u_2 - u_1} \left[(u_2 - u)\mathbf{c}(\mathbf{x}_1) + (u - u_1)\mathbf{c}(\mathbf{x}_3)\right]$$
$$\mathbf{c}(u, v_2) = \frac{1}{u_2 - u_1} \left[(u_2 - u)\mathbf{c}(\mathbf{x}_2) + (u - u_1)\mathbf{c}(\mathbf{x}_4)\right]$$

  Now interpolating in the $v$ direction gives us

$$\begin{aligned}
\mathbf{c}(u, v) &\approx \frac{1}{v_2 - v_1} \left[(v_2 - v)\mathbf{c}(u, v_1) + (v - v_1)\mathbf{c}(u, v_2)\right] \\
&= \frac{1}{(v_2 - v_1)(u_2 - u_1)} \left[\mathbf{c}(\mathbf{x}_1)(v_2 - v)(u_2 - u) + \mathbf{c}(\mathbf{x}_3)(v_2 - v)(u - u_1)\right. \\
&\qquad \left. + \mathbf{c}(\mathbf{x}_2)(v - v_1)(u_2 - u) + \mathbf{c}(\mathbf{x}_4)(v - v_1)(u - u_1)\right] \\
&= \frac{1}{(v_2 - v_1)(u_2 - u_1)} \begin{bmatrix} u_2 - u & u - u_1 \end{bmatrix} \begin{bmatrix} \mathbf{c}(\mathbf{x}_1) & \mathbf{c}(\mathbf{x}_2) \\ \mathbf{c}(\mathbf{x}_3) & \mathbf{c}(\mathbf{x}_4) \end{bmatrix} \begin{bmatrix} v_2 - v \\ v - v_1 \end{bmatrix}
\end{aligned}$$

  Produces blurry artifacts, since a single texel in the texture covers several fragments.

- To prevent / reduce artifacts produced by up sampling, ensure the texture map is of sufficiently high resolution.

**Definition 5.3.3.** (**Down Sampling**) Down sampling is the technique for sampling a texture where there are fewer fragments than texels.

IMAGE

- When down sampling, color values are averaged across the texels that the fragment covers.

**Definition 5.3.4.** (**Mipmaps**) A **mipmap** is a pre-calculated optimized sequence of raster image textures, each of which is a lower resolution representation of the same image. *Typically*, the height and width of each image is 1/2 smaller than the previous level.

IMAGE

- Mipmaps can be used for down sampling. Consider a mipmap with $n$ levels and a fragment that covers $m \times m$ texels (in the level-0 texture). Let $\lambda = \log_2 m$ be the level we select, note that $\exists \lambda \notin \mathbb{N}$, so consider $\lfloor \lambda \rfloor$ and $\lceil \lambda \rceil$. Options:
  1. Use level $\lfloor \lambda \rfloor$ and apply nearest neighbor / bilinear interpolation.
  2. Use level $\lfloor \lambda \rfloor$ and $\lceil \lambda \rceil$ and apply bilinear interpolation between levels.
- Mipmaps are also used to increase rendering speeds and reduce aliasing artifacts.

### 5.3.4 Normal Mapping

- Technique for rendering bumps/dents in a surface without using more polygons. Sometimes referred to as *bump mapping*
- Modified surface normals of an object are stored as $(r, g, b)$ components in a 2D raster image texture, referred to as a "normal map".
- The surface normals must be represented in a basis relative to the surface, referred to as "normal space". Which is defined using the tangent space of the surface:

  IMAGE

  The tangent space has the basis vectors $(\hat{\mathbf{t}}, \hat{\mathbf{b}}, \hat{\mathbf{n}})$, the tangent, bitangent and normal vectors. Note that $\hat{\mathbf{n}} = \hat{\mathbf{t}} \times \hat{\mathbf{b}}$. To calculate $\hat{\mathbf{t}}, \hat{\mathbf{b}}$, consider 3 points $\mathbf{p}_1, \mathbf{p_2}, \mathbf{p_3}$ on the normal map with $uv$ coordinates, $(u_1, v_1), (u_2, v_2)$, and $(u_3, v_3)$. Consider the vectors $\mathbf{p}_1 - \mathbf{p}_2$ and $\mathbf{p}_3 - \mathbf{p}_2$.
  IMAGE

  These can be expressed as a linear combination of $\hat{\mathbf{t}}, \hat{\mathbf{b}}$, yielding

$$\mathbf{e}_1 = \mathbf{p}_1 - \mathbf{p}_2 = (u_1 - u_2)\hat{\mathbf{t}} + (v_1 - v_2)\hat{\mathbf{b}}$$
$$\mathbf{e}_2 = \mathbf{p}_3 - \mathbf{p}_2 = (u_3 - u_2)\hat{\mathbf{t}} + (v_3 - v_2)\hat{\mathbf{b}}$$

Hence

$$\begin{bmatrix} \mathbf{e}_1 \longrightarrow \\ \mathbf{e}_2 \longrightarrow \end{bmatrix} = \begin{bmatrix} u_1 - u_2 & v_1 - v_2 \\ u_3 - v_2 & v_3 - v_2 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{t}} \longrightarrow \\ \hat{\mathbf{b}} \longrightarrow \end{bmatrix}$$

$$\iff \begin{bmatrix} \hat{\mathbf{t}} \longrightarrow \\ \hat{\mathbf{b}} \longrightarrow \end{bmatrix} = \begin{bmatrix} u_1 - u_2 & v_1 - v_2 \\ u_3 - v_2 & v_3 - v_2 \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{e}_1 \longrightarrow \\ \mathbf{e}_2 \longrightarrow \end{bmatrix}$$

With $(\hat{\mathbf{t}}, \hat{\mathbf{b}}, \hat{\mathbf{n}})$ known, we can calculate the matrix $TBN$, which changes basis from tangent space to object space

$$TBN = \begin{bmatrix} \hat{\mathbf{t}} & \hat{\mathbf{b}} & \hat{\mathbf{n}} \\ \downarrow & \downarrow & \downarrow \end{bmatrix}.$$

Alternatively, we may use the $TBN^{-1}$ matrix to transform other vectors to tangent space, and then perform lighting calculations in tangent space.

- The modified normals are then used in lighting calculations.

### 5.3.5 Displacement Mapping

- Note that normal / bump mapping cannot cast shadows (e.g. no self occulusion / silhouette changes)
- Displacement mapping is a more realistic technique of rendering bumps / dents. However is computationally more costly.
- Displacement mapping uses a "height map", which stores the modified height $h$ of the object in a texture. The new surface is then calculated using

$$\mathbf{p}' = \mathbf{p} + h\hat{\mathbf{n}}.$$

## 5.4 Raster Buffers

- A raster buffer is a OpenGL framebuffer object that stores the raster image that represents the current screen contents.
- There are several OpenGL buffers:

  - A Color buffer (consists 4 buffers, `GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT`). The left and right buffers are used for stereoscopic rendering. The front and back buffers are used for **double buffering** .
  - A Depth buffer. Used by the Z-buffer algorithm (`GL_DEPTH`)
  - A stencil buffer. Used to block rendering of select pixels. (`GL_STENCIL`)

### 5.4.1 Double Buffering

- To avoid tearing artifacts, we use double buffering, which consists of using two OpenGL color (raster) buffers, a front buffer, which stores what is displayed on the screen and a back buffer, which stores what the GPU is rendering.
- When the GPU finishes rendering, we swap the front and back buffers. IMAGE
- **Problem**: There is some delay while the GPU switches buffers. Hence rendering the next frame cannot start until the buffers are swaped.
- **Solution**: Introduce a 3rd buffer, which is used once the back buffer is written too. IMAGE
- Disadvantages:
  - Requires more memory
  - Increases delay between rendering the frame and displaying the frame.

### 5.4.2 Vertical Synchronization (V-Sync)

- The screen copies pixels from the front color buffer in a row-major order.
- If the front and back buffers are swapped during this process:
  - The upper part of the screen contains the previous frame.
  - The lower part of the screen contains the current frame.

This produces a tearing artifact.

- **Solution**: `glwfSwapInterval(1)`, causes `glSwapBuffers()` to wait until the last row is copied to the screen.
- **Problem**: Introduces lag. IMAGE
- **Solution**: Adaptive synchronization. The GPU controls the timing of swaps based on whether the display timings, etc. IMAGE