2024 圖學重點整理

- **Three Pillars of Computer Graphics(電腦繪圖的三大支柱)**

1. Modeling 2. Rendering 3. Animation

- **Rendering的兩種方式**

1. Ray tracing：從相機發射一條射線穿透pixel到物體，該點就是該pixel顏色。

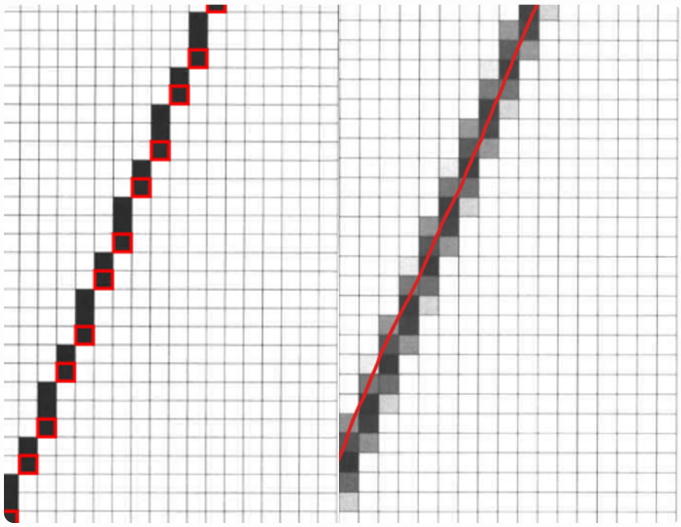
   > 優點：較通用
   > 缺點。計算複雜較慢(需要先比較三角形遠近)

2. Rasterization：從3D物體投影回2D平面，會佔據哪些pixel。**interactive computer graphics搭配pinhole camera一起使用**(因pinhole中光線是直線的，不會像透鏡一樣多次折射，比較好計算)。

   > 優點：可以平行運算(使用GPU)
   > 缺點：平行運算後需要判斷覆蓋的fragments

- **如何避免Anti-aliasing(反鋸齒)**

可以將兩端點連成一條線，查看線條在該pixel的比例，若占比高則填黑色，若占比低則填灰色。



- **vertex buffer和index buffer的儲存方式**

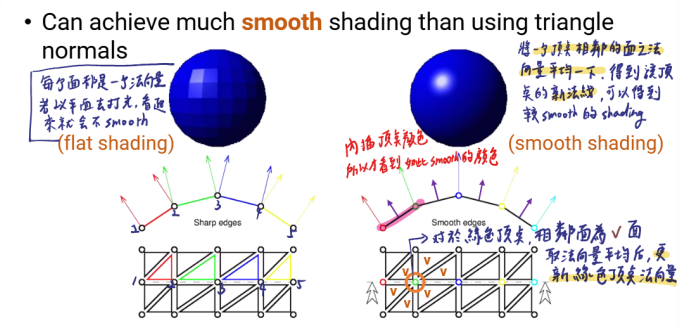
- vertex buffer

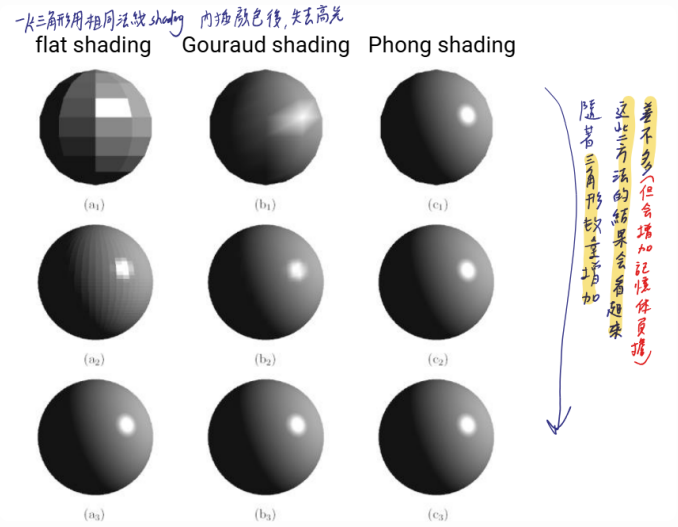
P1	T1	N1	P2	T2	N2	...
(x,y,z)	(x,y)	(x,y,z)	(x,y,z)	(x,y)	(x,y,z)	...

- index buffer：儲存要連線成三角形的頂點index(三個一組)

> - **如何lighting才會使物體表面變光滑**

- flat shading：一片三角形用相同的法線shading(看起來很不smooth)
- Gouraud shading：內插方法對物體打光，但缺點是會失去高光顏色(因為內插兩端fragments可能都沒有高光，這樣內插出來中間的fragments就不會有高光)
- Phong shading：使用內插法算頂點特徵(法向量)，再用法向量去打光
- 若物體被分為超級多個三角形後，**這三個方法打光的結果會看起來相同**。





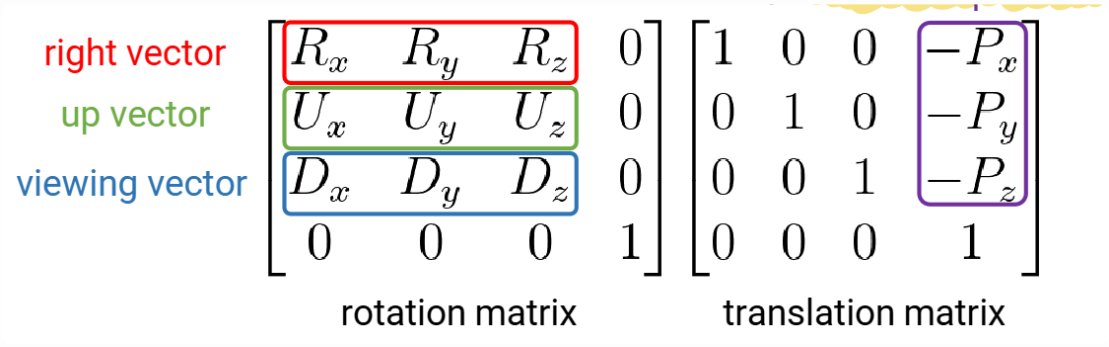
> - **各座標系的轉換以及Homogeneous Coordinate**

$Object\ space > World\ space > Camera\ space > Clip\ space > Screen\ space$

- $Object\ space > World\ space$的轉換

- 首先物件(object)在object space中被定義，當建立場景時，會透過**world transform**把物件們轉換到相同的座標。
- 這樣做的好處是**1)同樣的物件可以被重複使用(Reuse model)，不用多次定義，也可以2)節省記憶體(Memory saving)**
- Transform包含位移(Translation)、縮放(Scaling)和旋轉(Rotation)
- **Homogeneous Coordinate(齊次座標)**：點座標被表示為$[x, y, z, 1]$(為了做位移，會多一維)，透過齊次座標，我們可以定義一個world transform matrix，將位移縮放和旋轉矩陣合併到裡面。

- $World\ space > Camera\ space$的轉換
  - 為了讓計算方便，所以我們訂出一個 $Camera\ space$
  - 我們transform物體到$Camera\ space$需要進行兩個步驟：
    - 做**inverse translation**，移動所有物體和相機，將相機移動到原點，並且所有物體和相機的相對位置保持不變
    - 做**rotation**讓相機的的viewing direction對齊$-Z$軸
    - 在實作中(api)中，我們只需要定義 $CameraPos$、$TargetPos$、$Temporal_upVector$即可。
  - 將剛剛提到的**inverse translation**和**rotation**相乘，我們得到了camera matrix(4*4)



- $Camera\ space > Clip\ space$的轉換：
  - 投影方式有兩種：
    - Orthographic Projection(正交投影)：建築學常用，可以維持物體x,y的相對距離。
    - Perspective Projection(透視投影)：模擬人類視覺的投影法，會因為東西遠進而改變。
      - 推導Perspective Projection矩陣時，需要以下參數：
        - aspect ratio(螢幕長寬比)
        - vertical field of view(fov, 可視範圍角度)
        - near Z plane
        - far Z plane

    - 推導完以後，經過透視除法，才會轉換所有座標到$[-1,1]$範圍，成為NDC空間的座標。
  - Projection matrix：

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{ar \cdot \tan(\frac{\alpha}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\alpha}{2})} & 0 & 0 \\ 0 & 0 & \frac{-nearZ-farZ}{nearZ-farZ} & \frac{2 \cdot farZ \cdot nearZ}{nearZ-farZ} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

  - NDC matrix(在除以W前稱為$Clip\ space$，除完後變成$NDC\ space$)：

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

將物體從$Object\ space > Clip\ space$總共有三個矩陣：

- model matrix($Object > World$)
- view matrix($World > Camera$)
- projection matrix($Camera > NDC$)

> - **GPU pipeline重點整理**
>
> 原本在CPU上處理的話，對於每個頂點，我們用for-loop去跑，沒辦法像GPU一樣做平行處理，所以後來用GPU處理頂點資料，將頂點資料轉到NDC座標系上，並輸入到openGL上畫出

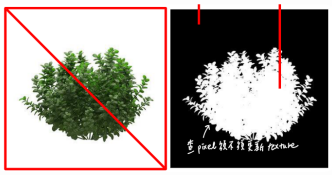
- OpenGL 1.0 pipeline架構
  - **Vertex Data**：將頂點資料讀入，把資料傳到**GPU**上處理
  - **Primitive Processing**：決定頂點的連接方式，使其成為三角形或多邊形(openGL 1.1可以為多邊形)
  - **Transform and Lighting**：將頂點轉到$Camera\ space$，並對齊打光
    - Gouraud shading：使用內插在頂點上計算lighting的方法，這會導致高光可能無法被算出。
  - **Primitive Assembly**：將三角形接起來，轉到$NDC\ space$，並對溢出的螢幕外的三角形做clipped。視情況決定要不要做back-face culling，最後把NDC座標map到螢幕上。
    - clipped：當三角形溢出螢幕時，切除多餘的部分，剩下的部分轉換為多個三角形。

- back-face culling：省略視角中看不到的三角形不用畫，計算viewing direction和face normal的夾角，若小於90度，則代表看不到該面。
- NDC map到螢幕：

$$x_s = w(x_{ndc}+1)/2 + \text{screen location}$$
$$y_s = h(y_{ndc}+1)/2$$
$$z_s = (z_{ndc}+1)/2 \quad 0 \le z_s \le 1$$
$$w_s = w_{ndc}$$

（有時候會加上視窗位置）

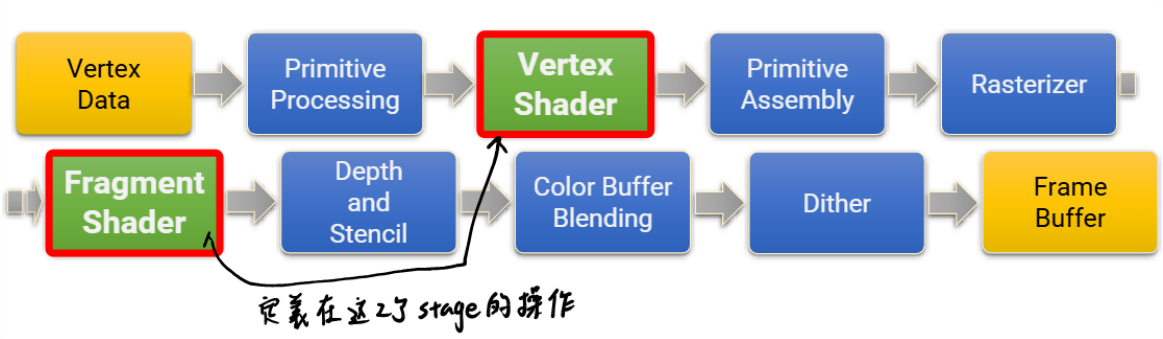
- **Rasterizer**：將螢幕上的三角形用演算法找邊，並對三角形中的fragments進行插值計算顏色或lighting。
  - Digital Differential Analyzer (DDA)：利用兩點求直線方程式，之後遞增帶入x座標，求出y以後四捨五入，並將這些點連線形成邊。
  - Bresenham Algorithm：將兩點連線，查看該線經過的格子(假設有A、B格子經過)，計算兩個格子中心到邊的距離，選距離短的格子為邊。
  - Scanline Rasterization：使用DDA或者Bresenham Algorithm方法查出三角形的邊，並用頂點資料插值計算fragments顏色。
  - Barycentric Coordinates(更有效率決定邊的方式)：使用$p = p_0 + \alpha p_0 + \beta p_1 + \gamma p_2$，$p$點由此公式表示，若$p$點位於三角形內部，則$\alpha, \beta, \gamma$範圍會介於$[0, 1]$之間。之後內插也可以使用$\alpha, \beta, \gamma$這三個權重來內插。
- **Texture Environment**：處理貼圖，頂點資料會紀錄貼圖座標(x, y)，在Rasterizer後，可以計算每個fragments貼圖位置屬性的內插，並到texture上去找。
- **Color sum**：openGL 1.0同時計算物體打光和貼圖的方式是$texture * lighting$，但lighting會被貼圖限制，假設texture為黑色，則不管lighting再怎麼強，相乘結果都還會是黑色，所以補償做法是後面會接一個color sum。
- **Fog**：定義霧的顏色，和當前物體顏色做linear combination來模擬霧的效果(減少龐大計算)。
- **Alpha Test**：使用貼圖來畫外型複雜且不重要的物體(ex：樹木上的葉子)。
  - 畫兩個三角形，上面貼貼圖
  - 對於非樹葉的部分，我們要使其透明，才不會遮住後面的背景
  - 定義alpha mask，如果為白色，可以將該位置畫在screen上(有物體)，若為黑色則丟棄該pixel。
  - 事實上texture可以跟alpha mask合併為RBGA使用



- **Depth and Stencil**：Z-buffer判斷pixel深度，並用stencil buffer處理Z-buffer中可畫但不想畫出來的區域。

        - painter's algorithm：從遠畫到近，但須要對scene中的物體進行排序，且無法畫出彼此互相交疊的物體。
        - Z-buffer：產生一個和原圖長寬大小相同的buffer，紀錄fragments離camera的深度，並更新畫出(早期由於記憶體空間有限，所以這個做法以前不盛行，但現在成為主流)。
        - Stencil buffer：處理Z-buffer中可畫但不想畫出來的區域，ex：窗戶，類似mask的概念。
    - **Color Buffer Blending**：半透明處理。會改變畫圖順序，先畫遠物再畫近物(黃色窗戶)，並看alpha值，若越高則保留越多遠物顏色。
    - **Dither**：替顏色編碼，用比較少的bit表示該pixel。(ex：透過密集排列藍色和紅色，我們可以創造出紫色，就不用保存紫色的編碼)，但因為現代記憶體發達，所以不會用這個。
    - **Frame Buffer**

- OpenGL 2.0 pipeline架構



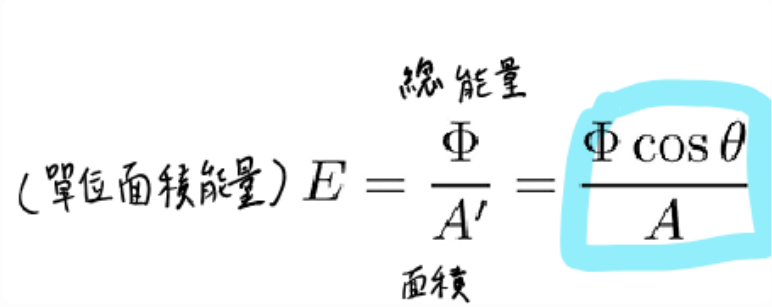
    - Vertex shader：裡面定義的operations(如transform)會對所有頂點執行一次，最後要輸出的頂點需在$Clip\ space$
    - Fragment shader：裡面定義的operations(如lighting)會對所有fragments執行一次，最後要輸出頂點的顏色。
        - Phong shading：利用rasterizatin內插頂點法向量資訊，再用此資訊去算顏色(打光)

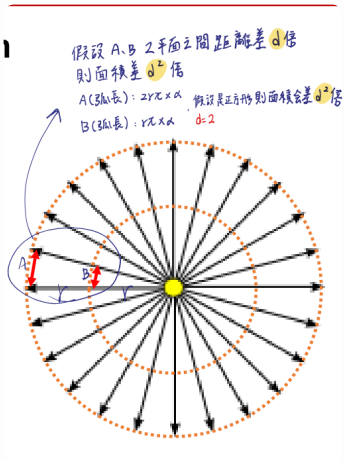
> - **Lighting and shading**

在Graphics中，shading指決定一個點的顏色。顏色和以下屬性有關：

- Surface orientation (normal)-法向量
- Lighting direction vL (and Θi)-光入射方向(入射角)
- Viewing direction vE (and Θo)-眼睛所在位置(反射角)
- Material properties-物體材質
- Participating media-中間介質(ex:霧)

- **Lambertian Cosine Law**：光源對於顏色的影響，物體上點的亮度取決於單位面積受到多少能量(Energy)



- 對於各式各樣的光，我們分為兩大類：
  - Local lights：光在scene的內部
    - Point light：點光，很小的光源，不考慮面積(**往四周發射出去的能量都相同**)，需要定義光的位置和強度。
    - Spot light：也是一個點，會有一個入射方向，在某個範圍內光強度會很強，在falloffstart範圍內強度很強且皆相同(會被包在totalWidth中)，totalWidth是光可作用的整體範圍，到了falloffstart外，能量會遞減，逐漸衰落。需要定義光的位置、強度、主要照射方向、totalWidth和falloffstart。
    - Area light：面積光，使用其著色scene會比較柔和，計算量大，通常會對area light進行sampling(採樣)，再從sampling的結果回推。
    - 
  - Distant lights：光在scene外部(ex:太陽光)，光的入射方向固定，且不會考慮距離對光產生的衰減。
    - Directional Light：入射光。需要定義光的入射方向，和沿著方向會遞減的輻射量(Light radiance)
    - Environment Light：很多方向的入射光，當太陽光經過雲層會產生散射
      - image-based lighting (IBL)：在一個2D的texture上上使用經緯度當成x,y座標，這可以使我們mapping到地球表面位置，之後該位置和地心連線就會產生入射方向，該texture上的pixel紀錄該入射方向的能量。

>

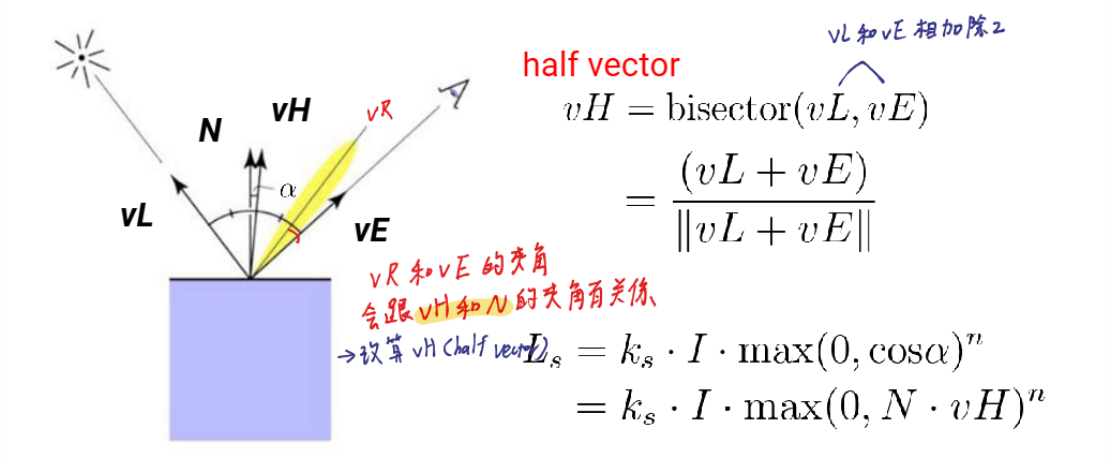
- **比較Local, Direct, and Global Illumination**

- Local illumination：只考慮光直射，不考慮物體遮擋和光的反射。
- Direct illumination：只考慮光直射和物體遮擋，不考慮光的反射。
- Global illumination：全域照明，同時考慮物體遮擋和光的反射，計算量龐大。

- **Material**

Phong Lighting Model提出如何對不同材質打光的方法，分為：

- Diffuse reflection(漫反射)：和人眼的位置無關，每個方向有一樣的反射。
  $L_a = k_a * I * max(0, N * vL)$
  - K_d：物體對於RGB三個channel不同的反射程度
  - I：光的總強度
  - $max(0, N * vL)$：入射角和法向量的夾角(我們通常會使入射角指向光源，因角度小會比較好計算)

- Specular reflection(鏡面反射)：會和人眼的位置**有關**，
  - VR(先找出入射角=反射角的反射角)：
    $vR = vL * 2((N * vL)N - vL) = 2(N * vL)N - vL$
  - L_s(光反射)：$L_s = k_s * I * max(0, vE * vR)$，其中$vE * vR$為人眼方向和反射向量的夾角，隨著此夾角越大，能量會開始衰減。
  - **Blinn-Phong**：他把Phong Specular reflection做一些修正，他發現人眼方向和反射向量的夾角可以用$N * vH$表示，vH則可以用vL和vE計算得出。
  - 

- Ambient reflection(環境反射)：模擬全域照明，和人眼的位置無關。$L_a = k_a * I_a$
  - K_a：物體對於RGB三個channel不同的反射程度

    - I_a：場景中，ambient light的強度。

- 最後，若一個scene中有很多盞燈，我們只要將這些結果相加即可。