**CH6 GPU & Shaders**

Real-time without overworking CPU: phong, bump mapping, particle systems(EX: 火焰), animation

Real-time graphics: GP-GPU(CUDA, OpenCL): scientific data processing, computer vision, deep learning

Shader: replacing fixed-function vertex and fragment processing

Vertex shader: dealing with per-vertex functions and we can control lighting and position of each vertex

Fragment shader: dealing with per-pixel functions and we can control color of each pixel by user-defined programs (phong shading, bump mapping, smooth shading)

Vertex shader application: control movement with uniform variables and vertex attributes (time, velocity, gravity), moving vertices (morphing, wave motion), lighting (realistic models, cartoon shaders)

Fragment shader: fragment(potential pixel: 可能很多點黏在一起但最終只會在螢幕上顯示一個點)，operations on interpolated values, texture access and application, other function(fog, color lookup)

D3D 10 pipeline:

Input assembler: supplies data (triangles, lines, points) to the pipeline

Vertex shader: processes vertices (transformations, skinning, lighting)

Geometry shader: processes entire primitives: supports limited geometry amplification and de-amplification (discard the primitive, emit one or more new primitives)

EX: subdivision, point->billboard, silhouette edge->fur

Stream-output stage: data can be streamed out and/or passed into the rasterizer. Data streamed out to memory can be recirculated back into the pipeline as input data or read-back from the CPU.

Output-merger stage: combines various types of output data (pixel shader values, depth and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.(三角形做混合或把不用的東西砍掉)

D3D 11 pipeline: improve the tessellation ability by three new stages: hull shader, tessellator (把三角形割碎細化), domain shader. The tessellated patches can still be applied to geometry shaders.

**Ch7 Rasterization:** clips primitives, prepares primitives for the pixel shader and determines how to invoke pixel shaders

Pixel shader: receives interpolated data for a primitive and generates per-pixel data, such as color

Discrete video screen: assigning pixel values by functions: SetPixel(x, y, color)=>離散地放pixel在螢幕上、每次都要syscall=>速度慢、可能會interrupt;

Buffer or array: FrameBuf[x][y] = color=>這個方法比較好、array化比較有效率

Line-Drawing algorithms: start with line segment in window coordinated with integer values for endpoints(離散化畫在畫面上)

DDA algorithm: digital differential analyzer = for each x plot pixel at closest y

Bresenham’s Algorithm(純整數運算法):

DDA requires one floating point addition per step vs. Bresenham eliminates all fp

Consider only 0<=m<=1(handing other cases by symmetry)

Assume pixel centers are at half integers

Characteristics: if we start at a pixel that has been written, there are only two candidates for the next pixel

General polygons: 畫出點之後會自動幫你連線，可能還會幫你補點(圖形可能會歪掉)打斷convex(凸多邊形)法: 三角形太細長不好; 中間(扇形)打斷法: 三角形數量會太多

Inside or Outside: 一條線經過，會經過幾個邊界，適用中間中空的圖形

Odd-even rule: draw a ray to infinity and count the number of edges that cross it=>even: outside; odd: inside=>usually used for polygon rasterization

Non-zero winding rule: 判斷某個點是否在圖形內 : 外面有沒有點繞過他 (這種計算方式會有太多重複運算的東西): (+1 for clockwise; -1 for counter clockwise) =>non-zero winding counts = inside

Polygon Filling by Scan Lines: fill maintaining a data structure of all intersections of polygons with scan lines: sort the scan lines=>fill each span

Data Structure for General Cases: 先準備好點=>sorting=>畫線打點=>同一個高度上會有哪些點

**Ch8 Hidden Surface Removal (HSR) & Culling:** 三角形來就打點上去，但是圖形會有互相遮蔽(occlusion)問題(圖形擺放先後)

Back-face Removal(Culling): 視角內就畫，視角外就不要畫=>仍無法處理所有的case(還是有一部份不用畫的無法被這個方法清掉)

Hidden Surface Removal: 直接檢查前後只畫在前的，但是遇到部分重疊會壞掉: object-space approach: use pairwise testing between polygons(objects) =>worst case complexity O(n^2) for n polygons

Painter’s Algorithm: 看圖形順序性的關係: 後面東西先畫才畫前面，但是有排序的成本(只適合某些case)

Depth Sort: requires ordering polygons first: O(n log n) calculation for ordering; not every polygon is either in front or behind all other polygons=>Order polygons and deal with easy cases first, harder later

Easy cases: a polygon lies behind all other polygons=>render; polygons overlap in z but not in either x or y=>分別render

Difficult cases: 畫家演算法不能處理的事情: 物件一多就sort不完: overlap in all directions but can one is fully on one side of the other; cyclic overlap(z互相交錯); penetration(滲透: 斷開三角形，但可能會產生新的平面去切別人=>切到別人就把別人切開=>三角形指數增加)

Image Space Approach: rasterization畫成打點: 每次只要畫(set pixel)射線最前面的、紀錄最靠近螢幕的點在z buffer

z-buffer Algorithm:存目前最靠近螢幕的polygon；清空=>設無窮遠；as we render each polygon, compare the depth of each pixel to depth in z buffer(z的order不用管，因為在螢幕上是離散的), 靠近的點就蓋掉(上色)螢幕上的pixel。

螢幕解析度和z buffer是一樣的(buffer解析度，EX: 8 bits, 24 bits)，解析度越高、可以切分的數量越細

Scan-Line Algorithm: 要有深度資料才能正確判斷

Interpolation of Z values: rasterize後可以內插出每個點的z值 (color, normal, depth都可以做內插)

Screen Space vs. 3D Space: 與近大遠小有關: 畫面上1:1的東西，實際上可能其實是有交錯的，螢幕顯示的三角形邊界會有問題

Simple Screen Interpolation: 被深度值和螢幕比例關係干擾，找到錯誤的點

Space partitioning: avoid rendering an object when it’s unnecessary=>reduce burden on pipeline, reduce traffic on bus=>Octree, BSP tree

Octree: 用好幾個類似bounding box的東西去包它，每個box都細分很多boxes: 如果view有牽扯到boxes，就去檢測intersection，如果一半有相交、一半沒有=>box才再細分(階層式)，並且不需要走的tree可以先槓掉，只走需要的branch

BSP tree: 原因: hidden surface removal (painter: 一定會照著同一個方向sort(由後往前)，但是每次都要做z sort)、partition space with Binary Spatial Partition (BSP) tree，一般從最中間的圖案當作root(否則tree unbalance=>效率會變差)=>也可以用做visibility and occlusion testing

Back-to-Front Render: painter's algorithm變形: 先畫遠的再畫近的，與view不一定有關

BSP-based Culling: pervasively used in first person shooting games (doom, quake..)、visibility test(檢測view和門的關係)、skip objects that are occluded

Other Culling Methods: Portal Culling: walking through architectures、dividing space into cells、cells only see other cells through portals

**Ch9 Buffer and Mapping technique**

Define a buffer by its spatial resolution(n\*m) and its depth (or precision) k, the number of bits/pixel.

OpenGL frame buffer: front, back (canvas), depth, stencil(標記位置做後續處理) buffer，東西先寫到back，寫完再前(front)後互換，換到前面顯示出來(display :直接map到螢幕上，cost高)

Bump mapping(凹凸映射): emulates altering normal vectors during the rendering process: 局部處理貼圖打光資訊(因為原貼圖在光改變時無法跟著動) 改lighting algorithm: phong改normal，lighting讓表面看起來有凹凸

Coordinate systems:

Parametric: may be used to model curves and surfaces

Texture: used to identify points in the image to be mapped

Object or World: conceptually where the mapping takes place

Window: where the final image is really produced (projection coordinate)

forward mapping:建立每一點的s, t參數式對應到曲面上的位置x, y, z

backward mapping: 光柵化後回去x, y, z拿點較順手: s = s(x, y, z), t = t(x, y, z)

two-part mapping: one solution to the mapping problem is to first map texture to a simple intermediate surface (EX: map to cylinder)OpenGL在做，實用性不高；不同貼法效果不同: 會有錯覺(視覺效果)；其他方法: 人為直接對應點和圖比較精美

box mapping: 邊界可能會斷裂有縫

easy to use with simple orthogonal projection, also used in environment maps (cube mapping)

second mapping: map from an intermediate object to an actual object: normal from the intermediate to the actual; normal from the actual to the intermediate; vectors from the center of the intermediate

texture mapping for polygons: based on parametric texture coordinate

Interpolation: OpenGL uses interpolation to find proper texels from specified texture coordinates；內插只保證顏色連續，但有可能因為扭曲而有破綻

Reduction of the flaws: 打斷一點三角形(增加小三角形數量)，縮小畫面扭曲問題

Minification: more than one pixel can cover a pixel；Magnification: more than one pixel can cover a texel

Aliasing: 圖片變小之後pixel跳著拿，但因為線是連續的、pixel是離散的，所以顏色不連續、鋸齒狀

Resampling: resample the checkerboard by taking one sample at each circle

Area averaging: a better but slower option is to use area averaging vs. 高斯: 比較符合透鏡效果，但人眼效果不明顯

Box filter: 相鄰顏色會交互干擾=>失焦、模糊化=>再放大圖片就會降低鋸齒感

Mipmapped Textures: 階層緩衝, allows for pre-filtered(想減少run time) texture maps of decreasing resolutions, lessens interpolation errors for smaller textured objects

Pre-filter: 先算出幾個不同的scale (solution)(減少做錯的次數和時間)，

一個三角形進來後直接去找適合哪個resolution、才去做sampling、顏色取平均

Highly specular surfaces: shader允許把frame buffer變成texture，再放到polygon上，每產生一個平面鏡就要產生一個virtual camera、會有折角感

Render a flat mirror: 1. 要把view考慮進來，但是每次都要重新計算射線(retrace)=>x 2. 把攝影機移到鏡子對面(virtual view : 虛擬視角)，再做原本的render，不過要做兩次render(second pass)(因為要合併場景)不然就是直接複製所有物體到鏡子後，不過前提是場景能擋得住

Environment(reflection) mapping: 假設物體非鏡平面，讓最外面有套一個很大的texture(可以用render的，表示物體都離camera很遠)，看射線交到map的哪裡就拿來，但是視角轉換時會有差距

box map: 假設要產生一個鏡反射，想像物體被一個大方塊包住，物體往六個方向各拍一個view產生texture，計算射線和box的相交點(利用normal和反射角)，拿到的顏色再和鏡反射物體材質的顏色混合

Sphere mapping: the image is taken from a perfectly reflective sphere, assume the size of the sphere->0, map the rays to the environment, using orthogonal projection

Bump and normal mapping: represent surface details and avoid heavy geometric computation, 同一顆球 : geometry相同=>加入不同材質、改變normal，原本用很多小三角形畫的話要經過很多種transformation，但mapping可以用大三角形畫，但邊緣平滑(理想 : 凹凹凸凸)

Opacity and transparency: 因為不想要切太多太細的三角形，所以會用半透明片處理, opaque surfaces permit no light to pass through(alpha=1); transparent surfaces permit all light to pass; translucent surfaces pass some light (translucency=1-opacity(alpha))

Physical models: dealing with translucency in a physically correct manner is difficult due to the complexity of the internal interactions of light and matter; using a pipeline renderer

Order dependency: 疊顏色片的順序會影響最終呈現, 用不透明片擋住透明片，透明片的顏色會完全被擋掉；若透明片在不透明片上=>blend color

Translucent polygons should not affect depth buffer=>render with glDepthMask(GL\_FALSE) which makes depth buffer read-only(不透明片要去跟z buffer去看有沒有遮蔽問題，要用透明片的時候先關掉depth test、要排序)

Motion blur: 揮劍會有殘影、隨著時間decay、將連續兩三個frame疊在一起製作軌跡效果; 曝光大，畫面清晰；場景暗、動作快速畫面模糊=>殘影

full screen anti-aliasing(FSAA) : 把圖render大一點(拉高解析度)，cover一個位置的pixel數增加，把顏色疊在一起=>cost高

multisampling anti-aliasing(MSAA): the fragment shader still run once per pixel for each primitive; MSAA then uses a larger depth/stencil buffer to determine subsample coverage(邊緣和背景結合；sampling在圖案斜角上做，視覺效果較佳)

deep learning super sampling(DLSS) 2.0: 先render鋸齒狀的圖=>DL處理雜訊(PLS) : 提高解析度 + 反鋸齒=>美美的圖(4K)

motion vector : 上一張圖到下一張圖差距很大的時候，因為深度不同，遠近動的比例不同，anti-aliased較容易使用，

motion vector可以提供顏色資訊，但不需要每個物件都train

autoencoder: noise=>clear : 操作時會先把畫值下壓(subspace)，踢掉不必要的東西之後再放大；圖像自由度不高、圖片有相依性

**Ch10 Advanced Rendering**

Forward ray tracing: rays emanate from light sources and bounce around in the scene; rays that pass through the projection plane and contribute to the final image

打的層數越多、畫面越擬真、但cost會越高

Whitted ray-tracing: for each pixel, trace primary ray(eye ray) to the first visible surface; for each intersection trace secondary rays: shadow, reflected考慮物體的交互關係=>recursive call: 會生成新的、亮度較低的ray ), refracted (transmitted), Shaded color of Vi = Valid(Li) \* PhongModel + ReflectedRay + TransmittedRay

A simple ray tracer: 每道光每次三個分支(樹狀圖)，到phong的時候就停止，或設置最多打到幾層(會傳遞層數的參數)就停止

Supersampling: 一個pixel多打一些點(拿多一點資料)但是蠻多都只有打到空洞

Efficiency of Ray Tracing: efficient representation of an object(使用有效率的表達方式，EX : 球體以方程式來表達比較好retrace)、bounding box(要畫很多小兵的話，就看ray有沒有打到包住全部小兵的bounding box)、space partitioning(Octree, BSPtree)、distributed ray tracing(non-uniform ray distribution)

Ray tracing: an image space algorithm (畫面一旦有更動就要重算), view-dependent, rendering scenes with perfect specular reflection and refraction, point light sources, ideas from the path of light flow

Radiosity: an object space algorithm, view-independent(pre-computed), rendering perfect diffuse scenes, light sources are polygonal patches, ideas from the conservation of energy

One way to simplify the rendering equation: all surfaces are perfectly diffuse reflectors; dealing with diffuse-diffuse interactions、a scene is divided into patches

Hemicube algorithm: a hemicube is constructed around the center of each patch; faces of the hemicube are divided into pixels; each patch is projected(rasterized) onto the faces of the hemicube; each pixel stores its pre-computed form factor; the form factor for a particular patch is just the sum of the pixels it overlaps; patch occlusions are handled similar to z-buffer rasterization

Monte-carlo ray tracing: cast a ray from the eye through each pixel; cast random rays from the visible point

Monte-carlo path tracing: trace only secondary ray per recursion; but send many primary rays per pixel(直接從起點控制ray的數量，出去之後random往外彈，recursive比較難控制ray的數量)

Photon mapping: bi-directional paths(construct paths not only from the eye, but also from the light sources); caching(cache photons distributed along paths from the light sources); interpolation (interpolate radiance from cached photons); 可以設定specular、catch、diffuse的比例

Radiance estimation: 要考慮周圍的光(area filtering)周圍顏色取平均

光點多area可以取小一點(可以做的較精細)；光點少area就要取大一點

Real-time ray tracing: 運算最複雜，ray到三角形的距離、polygons之間的intersection=>用HW來做 : 用bounding box、bounding volume hierarchy(BVH) traversal(like Octree)

每秒能處理的ray還是有限 : 每個畫面只射一個ray，可以用train過的neural network、CNN(denoise)來補光線不夠的地方