3D Graphics Rendering Pipeline

Computer Graphics

Terms

Right hand Coordinate System

Fragment
- 3D, (x,y,z)
- z denotes depth
- produced by interpolation of vertices

vs

Pixel
- 2D dots on display
- (x,y) + color

Primitives → Input to GRP

Formed by one or more vertices

Basic geometrics (point, line, etc)

Vertices → attributes
- position/coordinates
- color (RGB)
- Texture T=(s,t)
- Vertex-normal

Hardware

Inputs → CPU ⟷ External / Main Memory (RAM)

API → GPU ⟷ Graphics Memory VRAM + Frame Buffer → Display

Vertex Transform (stage of vertex processor)

Model Spaces
↓ Model Transform
↓ World Space
↓ View Transform
↓ Camera space
↓ Projection Transform
↓ Clipping-Volume Space
↓ Viewport Transform
↓ Screen Space
Display

Rasterizer ←

Visibility

1) store current min. z for each sample
2) z-buffer store depth + frame buffer store color

z-Buffer
- front to back is the best

Painter's Algo
- back to front, overwrite
- z as front
- counter clockwise

↑ time
dis
↓ unsortable

z-value: [0,1] distance from camera

Vertex Shader

compute position + evaluate normal, texture

Raw Vertices/ Primitives → Vertex Processor (Programmable)

Interpolates vertices → Rasterizer 光栅化

Fragments → Fragment Shader → Fragment Processor (Programmable)

Processed Fragments → Output Merging
- combine fragments into pixels

Pixels → Display

3D Rotation

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation by angle α around axis $\wedge$

$$R(\wedge,\theta) = I + (\sin\theta)K + (1-\cos\theta)K^2$$

$$K = \begin{pmatrix} 0 & -\wedge_z & \wedge_y \\ \wedge_z & 0 & -\wedge_x \\ -\wedge_y & \wedge_x & 0 \end{pmatrix}$$

$$R_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} \rightarrow \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix}\cdot\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

point vector

$$\begin{pmatrix} x' \\ y' \\ w \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}\cdot\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

valid operation if $w = 1/0$

- add w-coordinate

Homogenous Coordinates 齐次坐标

$$\begin{pmatrix} x \\ y \\ w \end{pmatrix} = \begin{pmatrix} x/w \\ y/w \\ 1 \end{pmatrix}, w \neq 0$$

$$\begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$$

- matrix $B \cdot A$
- A first, B last

* ordering matters

Composite transform

linear map + translation

=

Affine map

no matrix form

Transformation

3D ←

- point $= (x, y, z, 1)^T$
- vector $= (x, y, z, 0)^T$

$(x, y, z, w)(w \neq 0)$

$\downarrow$

$(x/w, y/w, z/w)$

parallel / perspective projections from 3D to 2D

Why?

Viewing

world coordinates ↕ camera coordinates

Modeling → define shapes in convenient coordinates

efficiently represent hierarchical scenes

enable multiple copies of same object

2D

translation

default: (0,0) as reference

rotation

matrix $R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$

$$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

shear

- matrix

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & a \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$$

Scale 放大/缩

matrix

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$$

s can be negative

negative s → reflection

reflection ⊂ scale

default

- uniform scale
- (0,0) as reference

$$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- vector + vector = vector
- point - point = vector
- point + vector = point
- point + point $w = 2$ invalid

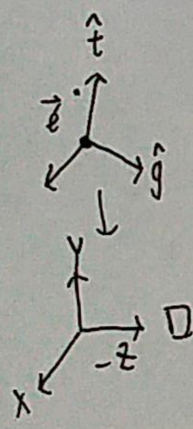
$$M_{viewport} = \begin{pmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & \frac{h}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- $(-1,1)^2$ to $(0,w)\times(0,h)$
- Transform in xy plane

$M_{view} = R_{view} T_{view}$

$$T_{view} = \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_{view} = \begin{pmatrix} x_{\hat{g}\times\hat{t}} & y_{\hat{g}\times\hat{t}} & z_{\hat{g}\times\hat{t}} & 0 \\ x_t & y_t & z_t & 0 \\ x_{-g} & y_{-g} & z_{-g} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



transform camera by M_{view} ← View transformation

1) translate $\hat{e} \to$ origin
2) rotate g to $-z$
3) rotate t to Y
4) rotate $(g\times t)$ to X

camera space
- camera at origin
- camera facing negative z
- y = vertical

Canonical Cube to Screen

Screen
- an array of pixels
- resolution = size of array

Screen Space
- pixels locate at (x,y)
- range : (0,0) ~ (width-1, height-1)
- (x,y) centered at (x+0.5, y+0.5)

Viewing and Perspective
- procedure of taking photo

1) find a spot and arrange position (model transformation)
2) find a good angle for camera (view transformation)
3) take it (projection transformation)

Projection Transformation

Steps
1) squish frustum → cuboid
2) do orthographic projection

$$M_{persp\to ortho} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

* any point on near plane will not change
* any point's z on the far plane will not change
* middle point's z become nearer to far plane

Orthographic Projection
正交投影
- camera at $z=\infty$
- same size regardless of distance

Steps
1) translate cuboid to origin
2) scale cuboid into canonical cube $[-1,1]^3$

Perspective Projection
透視投影
- 近小遠大

View frustum (truncated pyramid)
- total vertical angle of view fovy
- ratio of width/height aspect
- z Near - near clipping plane
- z Far - far clipping plane

sampling multiple locations within pixel and average

MSAA
Multisampling Antialiasing / Supersampling

-averaging values in pixel area
-sample at pixel's center

increase sampling rate
- high resolution
- expensive

how to solve?

Spatial Domain

inverse transform ↑ | ↓ Fourier Transform

Frequency Domain

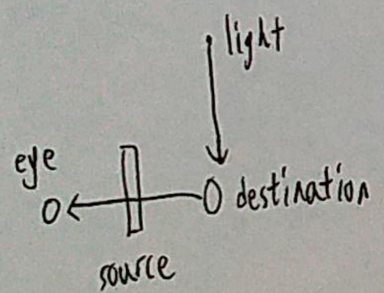
*to solve aliasing problem

Antialiasing

*filter out high frequency before sampling

Convolution Theorem

Filter by convolution in spatial domain



1) transform to frequency domain
2) multiply by Fourier transform of convolution kernel
3) transform back to spatial domain

Transparency

$C = \alpha C_S + (1-\alpha) C_D$

$\alpha \in [0,1]$

color of soure

color of destination

Shading Basics

Shading Models
- describe how object's color vary based on (surface, orientation, view direction, light)

example

Gooch Shading Model
- stylized
- compare surface normal to the light's location

normal point to light → warmer tone

normal point away → cooler tone

Aliasing
- undersampling causes frequency aliases

well-defined method for interpolating vertices

well-defined interior

Triangles as fundamental Area Primitive
- most basic polygon (can form other polygons)

三点共面 (guaranteed to be planar)

Light Sources
- parameters: size, shape, color, intensity

Directional lights
- no location
- only direction + intensity
- 无穷远发出

Shadows

soft: no boundary

hard
- only 1 point light

- Punctual Lights 点光源
- has location
- no shape/size

point/omni lights
- emit light uniformly in all directions
- intensity varies based on distance

spotlight
- project light in a circular cone
- para: angle

$L_a = k_a I_a$

- fake
- independent of everything

Ambient Shading

$L = L_a + L_d + L_s$

Blinn-Phong Reflection Model

Shading

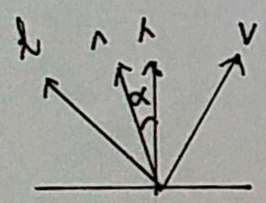
Shading Type

Blinn-Phong / Specular Shading

- depends on view direction

$L_s = k_s (I/r^2) \max(0, n \cdot h)^p$

$= \cos^p \alpha$



$h = \text{bisector}(v, l) = \frac{v+l}{|v+l|}$

↑ p, narrow reflection lobe

↓

光更集中

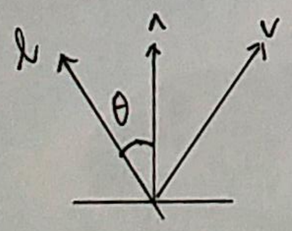
Lambertian / Diffuse Shading

- independent of view direction, uniform in all direction

$L_d = k_d (I/r^2) \max(0, n \cdot l)$

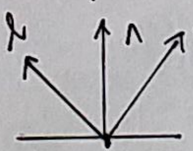
$= \cos\theta$

$=$ energy received by shading point



Phong Reflectance Model

- compute light reflected toward camera at a specific shading point



v : viewer direction

n : surface normal

l : light direction

Shading Frequencies

Flat shading

- shade each face
- face = 1 normal vector

Phong Shading

- shade each pixel
- interpolate normal vectors across each triangle

cost ↑ but if no. of vertices ↑

effect ↑ differences ↓

Gouraud Shading

- shade each vertex
- vertex = 1 normal vector
- interpolate colors from vertices

Shading normal vs geometric normal

- made up
- stored in vertex

- true in physics

change height with change in viewing direction — Parallex

guess appearance of pixel by examining height

position of object relative to each other

Parallax Mapping 视差贴图

bumps stored in heightfield texture

disadvantages
- never shift location with view angle, don't block each other

凹凸贴图

Blinn's Method
- use a heightfield to modify surface normal's direction

directly store normal map

Bump Mapping
- mesoscale modeling (adjust at pixel level)
- access texture to modify surface normal

solution

mipmapping (GPU)
- adjust resolution of image
- interpolate between 2 samples of texture
- mipmap chain (multiple sizes of texture sample) * must be power of 2

magnification vs minification
- $\frac{\text{texels}}{\text{pixels}} < 1$
- $\frac{\text{texels}}{\text{pixels}} > 1$

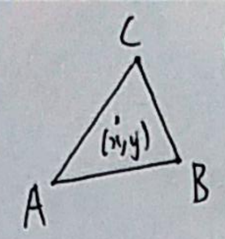
<1 | 1 | >1

resolution high ⟶ low

* limitation: blurring horizon

solution ⟶ 各向异性过滤

Anisotrophic sampling
- take extra sample along axis and blend them



(α, β, γ)

$(x, y) = \alpha A + \beta B + \gamma C$

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

* if $\alpha, \beta, \gamma \geq 0$, inside triangle

- used to linearly interpolate values at vertices

Interpolation across triangles
- Barycentric coordinates 重心坐标系

Texture Mapping / Texturing 纹理映射

- takes a surface and modifies its appearance at each location using image/function/other data source

↓

Texture Pipeline

↓ object space location

① projector function 投影函数

- take surface location (x, y, z) pixels → project → texels texture coor space (u, v)

* result stored at vertices

texture coor ⟶ ② Corresponder function 映射函数 (u, v) * no. of texels, drop fractions

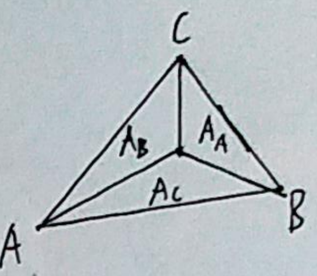
- texture coor ⟶ texture space locations
- control the way an image is applied
- determine behavior when $(u, v) \notin [0, 1]$

a) wrap
- repeat itself

b) mirror
- repeat itself but flipped

c) clamp
- clamped inside $[0, 1]$, edge

d) border
- defined color



$$\alpha = \frac{A_A}{A_A + A_B + A_C} \qquad \beta = \frac{A_B}{A_A + A_B + A_C}$$

$$\gamma = \frac{A_C}{A_A + A_B + A_C}$$

* center point

$(x, \beta, \gamma) = (\frac{1}{3} + \frac{1}{3} + \frac{1}{3})$

$$\alpha = \frac{[-(x - x_B)(y_C - y_B) + (y - y_B)(x_C - x_B)]}{[-(x_A - x_B)(y_C - y_B) + (y_A - y_B)(x_C - x_B)]}$$

transformed texture value
- used to modify property of surface

④ Value transform function

texture value

- access texture to obtain info (color)

③ Obtain value

texture space locations

Geometry

Bézier Curve

- Properties
  - tangent to end segments
    - tangent to end pts mid control points
  - interpolates endpoints
    - line pass p_0, p_n
  - convex hull
    - curve is within convex hull of control points
  - transform curve by transforming control points
    - 仿射 affine transformation
- Piecewise Bézier Curves
  - chain low-order Bezier curve
  - ($\because a_n = b_0 = \frac{1}{2}(a_{n-1} + b_1)$) continuity
- de Casteljau Algo
  - recursively insert point using linear interpolation
- Bernstein Form

$$b^n(t) = b_0^n(t) = \sum_{i=0}^{n} b_i B_i^n(t)$$

  - $b^n(t)$: Bézier curve order n
  - b_i: control points
  - $B_i^n(t)$: Bernstein Polynomial

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

Pros
- sampling is easy (plug in (u,v))

Cons
- hard to do inside test

Explicit 显式

- all points given directly or via parameter mapping

Polygon Mesh

- store vertices + polygons
- easy to do processing / simulation / adaptive sampling
- more complex data structures
- most common

Wavefront Object File Format

- common in Graphics research
- text file: specify vertices, normals, texture coordinates and their connectivities

Point Cloud

- list of points (x, y, z)
- easily represent any geometry
- useful for large datasets ($>>1 \frac{\text{point}}{\text{pixel}}$)
- often converted into polygon mesh
- difficult to draw in undersampled regions

Pros
- compact description
- easy for certain queries (inside object, distance to surface)
- good for ray-to surface intersection
- no sampling error / exact description (for simple shapes)
- easy to handle changes in topology

Cons
- difficult to model complex shapes

Implicit 隐式

- based on classfying points
- points satisfy some specified relationship

Algebraic Surfaces

- surface is zero set of polynomial in x, y, z

Constructive Solid Geometry

- combine implicit geometry via Boolean operations ($\cup / \cap / \setminus$, etc)

Distance Functions 距离场函数

- give min distance from anywhere to object
- blend surfaces together using DF

Level Set Methods 水平集

- store a grid of values approximating function
- when $f(x) = 0$, surface is found, ↑ explicit control over shape

Bezier Surfaces

- exhibit self-similarity
- natural phenomena
- hard to control shape

Fractals

1) use algo to evaluate point u on 4 Bezier curves in u → get 4 control points → moving curve
2) use Algo to evaluate point v on the moving curve

(u,v) - separable 1D de Casteljau Algo

- break polygons into triangles
Triangulation

T[0]: $(x_0,y_0,z_0),(x_1,y_1,z_1),(x_2,y_2,z_2)$

P[0]: (x_0,y_0,z_0)

special case

List of triangles

list of points
+
indexed triangles
T[0]: (2, 1, 0)

How to store triangles

- covering a plane using geometric shapes
Tessellation

the way triangles linked

the shape

Topology vs geometry

- pointer reassignments
- no elements created / destroyed

Edge Flip

- pointer reassignments
- add elements

Geometry Mesh

centering vertices
- to get more regular triangles

mesh regularization
- modify sample distribution to improve quality

Local Mesh Operations
- linked list

Edge Split

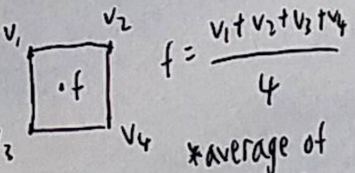
- delete elements
- pointer reassign

Edge Collapse
- replace edge with vertex

flip
- improve degree

Mesh simplification (downsampling)
- decrease resolution while preserve shape

Global mesh operations



$f = \frac{v_1+v_2+v_3+v_4}{4}$

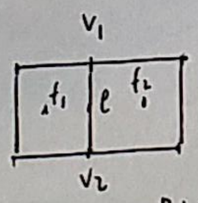
*average of surrounding vertices

mesh subdivision (upsampling)
- increase resolution via interpolation

Face point

edge collapsing

choose point with min quadric error



Edge Point

$e = \frac{v_1+v_2+f_1+f_2}{4}$

average of adjacent face points

old vertex

$v = \frac{\bar{f}}{n} + \frac{2\bar{m}}{n} + \frac{p(n-3)}{n}$

average of adjacent midpoints

valence of vertex

vertex point

Catmull-Clark subdivision
1) add vertex in each face
2) add midpoint on each edge
3) connect all new vertices

n = vertex degree
u = 3/16 if n=3
or
3/(8n)

Loop subdivision
1) create more vertices
2) tune their positions

update old vertices
- $(1-n*u)$ * original
+
u * neighbour sum

update new vertices
- neighbour vertices weighted sum

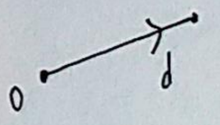
Ray Tracing

- accurate but slow

→ ray equation

$r(t) = o + td$

- point along ray
- time
- origin
- unit direction $\|d\| = 1$



(naive) intersection *

$0 \le t < \infty$ $t = \frac{(p'-o)\cdot N}{d \cdot N}$

ray-plane intersection

1) $p : (p - p') \cdot N = 0$
   - p': any point
   - N: normal vector
2) sub $p = r(t)$
3) solve

triangle mesh

Implicit surface

1) $p : f(p) = 0$
2) substitute $p = r(t)$
3) solve $f(r(t)) = 0$ for real, positive roots

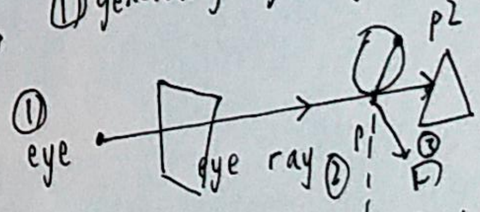
Light rays

- travels in straight line
- do not collide with each other
- travel from light source to eye

Ray Casting

- generate an image by casting one ray per pixel
- send shadow ray to light

① generating eye rays ② shading pixels ③ (recursive) calculate secondary ray



secondary ray
- shadow ray
- refracted ray
- reflected ray

Accelerating Ray-Surface Intersection

Bounding Volumes
- bound complex objects with simple volume

test Bvol → test obj

if ray x hit vol, ↓ x hit obj

Axis-Aligned Bounding Box AABB

key idea
- ray enters box only when enters all pairs
- ray exits if exits any pair

steps
1) calculte t_{min}, t_{max} for each pair
2) $t_{enter} = \max t_{min}$
   $t_{exit} = \min t_{max}$
3) if $t_{enter} < t_{exit}$ + $t_{exit} \ge 0$ ∴ intersect

Uniform Spatial Partitions (Grid)

- build acceleration grid
  a) Preprocess
  1) find bounding box
  2) create grid
  3) store each object in overlapping cells
- b) step through grid in ray traversal order
- if object test intersection

* grid resolution
  ≈ best
  cells = C * objs
  - 5
  - 27

Object Partitions and BVH (Bounding Volume Hierarchy)
- find bounding box
- recursively split objects → 2 subsets
- recompute BB of subset
- objects stored in leaf
- stop when no. of obj reach √

- leaf node
  - BB
  - list of objs
- internal node
  - bounding box
  - children

Spatial Partition
- BSP-Tree
- Oct-Tree
- KD-Tree
  - Leaf nodes
    - list of objects
  - Internal node
    - split axis
    - split position
    - children

    * no object