Principles of Computer Graphics - Summary

Andy Tràn

October 5, 2022

This will be a very personalised summary for me to use to study for the course Principles of Computer Graphics (CSCI3260). It might be complete, it might not be, it will probably not be. For questions you can refer to andtran@ethz.ch. This summary is based of the lecture notes and should be used as a supplement to the lectures.

Contents

0 Important organizational stuff

- pheng@cse.cuhk.edu.hk, office hours: Thursday 2:30 PM - 4:30 PM, SHB 929
- **Lecture hours:** Tuesday 10:30 am - 12:15 pm, Thursday 11:30 am - 12:15 PM
- **Tutorial hours**: Monday 3:30 pm - 4:15 pm, Thursday 5:30 pm - 6:15 pm
- **Reference book**: Fundamentals of Computer Graphics by Peter Shirley (not necessary), OpenGL Programming Guide
- **Course**: Consists of three parts: introduction, basics in graphics, more about graphics
- **Grading**: (2) programming assignments 0.25, course project 0.20, mid-term exam 0.25, final exam 0.30
- **Release**: Assignment 1 - release 12/9, deadline 02/10, assignment 2 - release 3/10, deadline 30/10, course project - release 31/10, deadline 27/11, mid-term exam 18/10 10:30 - 12:15

1 Lecture 1 - 06/09/2022: Introduction, Display and Colour

- Display Devices and Basic Terminologies
- Frame Buffer (Memory to Display)
- Color Space: RGB, CMY, HSV, YIQ, CIE YZ
- ALpha Channel and Double Buffering

1.1 Display Devices and Basic Terminologies

1.2 Frame Buffer (Memory to Display)

1.3 Color Space: RGB, CMY, HSV, YIQ, CIE, YZ

1.4 Alpha Channel and Double Buffering

1.4.1 Display Devices

Mechanism: shoot electrons with varying energy through vertical and horizontal deflectors to hit spot on screen, phosphors on screen jump to excited state when hit by electrons, emit monochromatic light when they drop to rest state
Random scan/Vector scan: give instruction and follow instruction
Raster scan: you go in a line and activate a line, and you turn each line on, where each spot on the screen is called a pixel. You shoot the gun, at the end of the line you turn it off and go back to the start, which is called **retrace**. There is a difference between horizontal and vertical retrace, horizontal is per line, vertical for each following line.
Interlacing: trick to get less flicker out of fixed signal bandwidth, it's like doubling the framerate, for example let's say we have 30Hz, we send two signals but each with a time difference between each other. For example to line 0 we send at 0 and 1, line 1 we send at 2 and 3. Result: doubling the perceived the framerate, without costing more bandwidth
Flat-Panel displays, two classes

- **Nonemissive displays**: LCD (optical effects to split light)
- **Emissive display**: field emission display (FED), light-emitting diode (LED), organic light-emitting diode (OLED). Which is more power efficient

On LCD: we block light instead of emitting the correct light.
FED: thousands of micro-electron guns

1.4.2 3D Displays overview

Two human visual cues used

- Stereopsis: seeing 2 slightly different images in each eye
- Motion Parallax: seeing slighty different images as you move around

Terms used:

- Stereoscopic: difference image to each eye, viewer must wear special glasses
- Autostereoscopic: different image to each eye, does not require special glasses
- Multi-view: different images depending on viewer's position

Note: Multi-view can be combined with used in combination of the others

1.4.3 Stereoscopic and Autostereoscopic displays

- Stereoscopic displays (common): two approaches
    1. using circularly polarized glasses (like in cinemas)
    2. using active shutter glasses (requires batteries, for example 3D TVs)
- Autostereoscopic displays: two approaches
    1. Lenticular lens: bright but blurry, old
    2. Parallax barriers, darker but sharper, like Nintendo 3DS

Downside of Autostereoscopic displays: usually limited to 1 or a very few viewers, and narrow sweet spot for viewing 3D.

1.4.4 Multi-view displays

Usually enabled by tracking the person's head.

1.5 Frame Buffer

Graphical storage (memory) and transformation hardware for digital images. We consider computer images as digital, we want to quantify a space into units (pixels).

1.5.1 Greyscale/Monochrome Frame Buffer

- Intensity of the raster scan beam is modulated according to the contents of a frame buffer
- Each element of the frame buffer is associated with a single pixel on the screen

Each marker corresponds to a pixel on the computer screen, remember rasterizing from the beginning of this lecture
Note: digital to analog converted (DAC)

1.5.2 Resolution

Determined by

- number of scan lines
- number of pixels per scan line
- number of bits per pixel

1.5.3 Colours

- 1 bit: B or W, 8 bit: 0 pure black to 255 pure white, with colours in between. To have true colour we need 8 bit per RGB
- To produce colours we mix intensities of each colour, to have a full spectrum we need a monitor which supports 256 voltages for each colour, the description of each colour in frame buffer memory is called **channel**. The term **truecolour (24 bits)** is for systems which the frame buffer stores the values for each channel (3 channels for RGB)
- **Color table**: for few bits per pixel, we have to map non-displayable colours to displayable ones. We can remap color table entries in software.

1.5.4 Look Up Tables (LUT)

Pseudo color: assign computed values systematically to a gray or color spectrum to indicate differences, for example height, speed, etc.

1.6 Color Space

1. RGB: additive color space, used for displays
2. CMY: subtractive color space, used for printing
3. HSV: (H circular, S distance from axis, V brightness), corresponds to artistic concepts of tint, shade, and tone
4. YIQ: (Y luminance, I orange-cyan hue, Q green-magenta hue), exploits properties of the visual system, used in TV broadcasting
5. XYZ system: defined in terms of three color matching functions

Definition 1 (Gamut) *device's range of reproducible color*

1.7 Alpha Channel and Double Buffering

1.7.1 Alpha Channel

Idea: we store one color per pixel, but we get hard edges. So we introduce an alpha channel next to the RGB channel, to blend with the lower layers to smoothen it out. Can be regarded as **1 - transparency** or **opacity**.

Example 2 (Blending) *We have a source and destination image, we can overlay them and the alpha value denotes of how much percentage we see each image when we overlay them*

1.7.2 Image Matting

What part of the image we want to keep, using a mask.

1.8 Double Buffering

Problem 3 *what happens when we write to the frame buffer while it's being displayed?*

Solution 4 (Double-buffering) *1. Render to the the back buffer and swap when rendering is done*

2. Double the memory

2 Lecture 2 - 08/09/2022: Useful 2D and 3D mathematics

2.1 Coordinate Systems

2.1.1 2D Cartesian Reference Frames

There are two ways of using this system: (a) starting at the lower-left screen corner, (b) starting at the upper-left screen corner.

2.1.2 Polar Coordinates in the XY plane

We start from a center, with a radial distance **r** and the angular displacement θ from the horizontal
We can convert it to the cartesian system: $x = rcos\theta$ and $y = rsin\theta$
To polar system: $r = \sqrt{x^2 + y^2}$ and $\theta = tan^{-1}(\frac{y}{x})$
Definition: $\theta = \frac{s}{r}$, where θ is the angle subtended by the circular arc of length s and r
We know: $P = \frac{2\pi r}{r} = 2\pi$, total distance around P

2.1.3 3D cartesian reference frames

Right-handed system: Take your right hand, palm towards you, thumb (positive x direction) to the right, index finger up (positive y direction), middle finger towards you (positive z direction). Yes, the back of you is positive z.
Left-handed system: Like RHS, but this time away from you is positive z. (Think of how to use your hand to show this)
In OpenGL: right handed (common), DirectX free to choose

2.1.4 Cylindrical-coordinate System

1. The surface of constant r is a vertical cylinder
2. The surface of constant θ is a vertical plane containing the Z-axis
3. The surface of constant z is a horizontal plane parallel to the Cartesian XY plane
4. Transformation from a cylindrical coordinate specification to a cartestian reference system

$X = rcos\theta$, $Y = rsin\theta$, $Z = z$

2.1.5 Spherical-coordinate system

Which is like polar coordinate in 3D space, we have $P(r, \theta, \phi)$
it holds that $x = rcos\theta sin\phi$, $y = rsin\theta sin\phi$, $zrcos\zeta$

Definition 5 (Angles in 3D) *We define it as* $\omega = \frac{A}{r^2}$*, total area is* $\frac{4\pi r^2}{r^2} = 4\pi$

2.2 Points and Vectors

2D vector: $V = P_2 - P_1 = (V_x, V_y)$, length is defined as $\sqrt{V_x^2 + V_y^2}$, angle is $\alpha = tan^{-1}(\frac{V_y}{V_x})$
3D vector: V is same as 2D but with one more value, length is defined equivalently with V_z. We have three direction angles

We have the following rules

1. Addition: $V_1 + V_2 = (V_{1x} + V_{2x}, V_{1y} + V_{2y}, V_{1z} + V_{2z})$
2. Scalar multiplication: $aV = (aV_x, aV_y, aV_z)$
3. Scalar product: $V_1 \cdot V_2 = |V_1||V_2| \cos\theta$, from there you can derive θ
4. Normalization: $\frac{V}{|V|}$, so its own product is 1
5. Perpendicular: $|A||B| \cos\theta = A \cdot B = \begin{cases} 0 & \text{if } \theta = 90deg \\ > 0 & \text{if } \theta < 90deg \\ < 0 & \text{otherwise} \end{cases}$
6. Cross product of two 3D vectors: $V_1 \times V_2 = u|V_1||V_2| \sin\theta$, where u is the unit vector that is perpendicular to both V_1 and V_2
7. Basis vectors: we can specify the coordinate axes in any reference frame with a set of vectors, one for each axis

Continuation of previous lecture Properties of cross product

- Anti-commutative: $V_1 \times V_2 = -(V_2 \times V_1)$
- Non-associative: $V_1 \times (V_2 \times V_3) \neq (V_1 \times V_2) \times V_3$
- Distributive: $V_1 \times (V_2 + V_3) = (V_1 \times V_2) + (V_1 \times V_3)$

2.3 Useful 2D mathematics

1. Distance from P to line AB:
   - Find line direction: $v = (x_2 - x_1, y_2 - y_1) = (d_x, d_y)$
   - Find normal of line by swapping elements in v: $n = (-d_y, d_x) = (y_1 - y_2, x_2 - x_1)$
   - Normalize n and v as $\hat{n}, \hat{v}$
   - Use dot product to find h and l: $h = |(P - A) \cdot \hat{n}$, $l = (P - A) \cdot \hat{v}$
   - Need to check l against length of AB
   - If $l < 0$ or $l > |AB|$, compute point-point distance
2. Line-Line intersection:
   - Express as parametric forms: $AB : (x_1, y_1) + t_{AB}(x_2 - x_1, y_2 - y_1)$, $CD : (x_3, y_3) + t_{AB}(x_4 - x_3, y_4 - y_3)$ and set them both equal
   - If no solution, that meas they are parallel
   - Substitute back to the parametric form
3. Which-side test: given a point and a line, Calculate
4. Area of arbitrary polygons: polygon area $\frac{1}{2} \sum det...$

5. Inside-outside test:
   - Method 1: repeat the which-side test for each edge in order (only works for convex polygons)
   - Method 2:Odd-intersection count, side = $\begin{cases} \text{outside}, & number \equiv_2 0 \\ \text{inside}, & \text{otherwise} \end{cases}$
6. Linear Interpolation
7. Barycentric coordinate: by means of area ratios
8. Spherical linear interpolation
9. Normal of a triangle
10. Approximate normal at a vertex (vertex normal vs face normal): average the face normals of neighboring faces

2.4 sth

Properties

- No fixed points under translation: all points move
- Multiple translations are order-independent, since addition is commutative

2.4.1 2D scaling

Properties

- Origin fixed: $x' = 0$ if $x = 0$
- Order independent: $x'' = x' \cdot S'_x = x \cdot S_x \cdot S_x \ = x \cdot S'_x \cdot S_x$

Single out *arbitrary fixed point scaling* at (x_0, y_0) as follows:

$$x' = x \cdot S_X + (1 - S_x)x_0$$
$$y' = y \cdot S_y + (1 - S_y)y_0$$

Rotate by θ:

$$x' = r\cos(\theta + \phi) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\sin(\theta + \phi) = r\sin\phi\cos\theta + r\cos\phi\sin\theta$$

Multiple 2D rotations are order-independent Fixed-point: origin dependent

2.4.2 Other 2D transformations

- Reflection (about X/Y - axis): not equal to a rotation, except when reflecting over x and y, then it's a rotation

2D shearing: order dependent

2.5 Matrix operations

2.5.1 Homogenous representations

1. Translation cannot be represented using $2x2$ matrices and homogenous coordinates help
2. We are left-multiplying the matrix on the vector/point

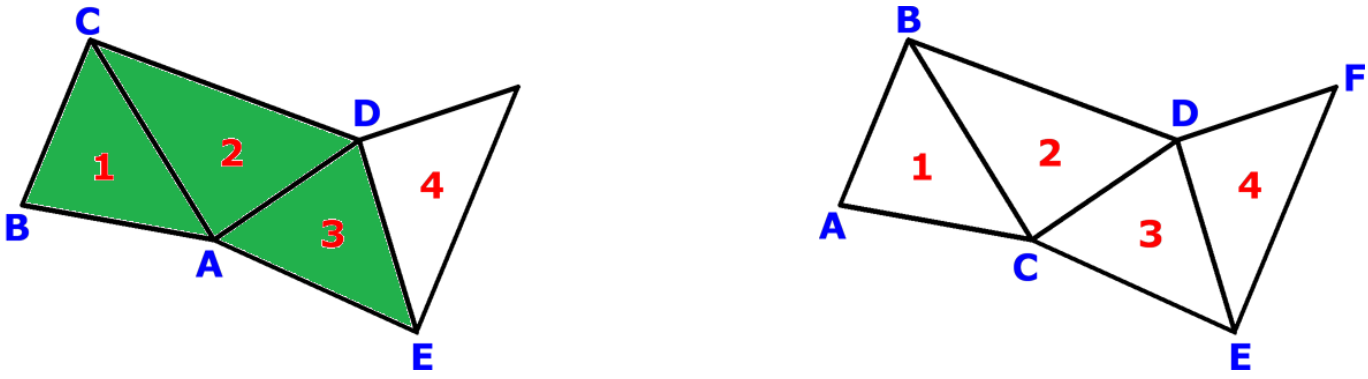
3 REAL Lecture 3: Hierarchical Model

Lecture Outline

1. Graphics Primitives: Points, Lines and Triangles
2. Data structure: vertex list and index list
3. Hierarchical Structure
4. View-world or Modelview transformations
5. Basic scenegraph concept

3.1 Graphics Primitives: Points, Lines and Triangles

- **Idea:** It's all about **coordinates** and **connectivity**
- **Meaning:** wireframe and meshes are built up by points, lines, or triangles. We call this **tessellation/triangulation**
- **Most efficient input type for rendering polygons: Triangle strip** or **triangle fan**. The differences between the two is the following, triangle strip is a set of triangles which share vertices (can share different vertices), while triangle fan has one **(!)** shared center.



(a) Triangle fan

(b) Triangle strip

Figure 1: In the triangle fan, we can get all triangles, in the strip only the coloured ones

3.2 Data structure: vertex list and index list

We have two kinds of data strucutres, namely vertex lists and index lists.

- **Vertex list:** you store all the vertex coordinates in one array
  - Benefit: sequential memory access
  - Negative: very likely to have duplicated vertices in the array, waste of storage
- **Index list/Indexed Face Set:** we have two arrays, one called the **face list**. That one contains all the polygons and the corresponding vertices. While the **vertex list** contains the coordinates for each vertex.
  - Benefit: reuse vertices and keeps a compact vertex list

– Bad: random memory accesses which lead to cache misses

Note: (1) triangle strip can be implemented on either data structure, (2) in some APIs these data structures are built in, (3) indexing can start with 0 or 1.

3.3 Hierarchical structure

A scene is composed of the multiple objects, which are composed of subjects as well. We can create a hierarchical model and break it down. Each objects has its own polygon list and associated vertex list.

3.4 View-world or Modelview transformations

- **Idea:** we model objects in a model space, which is pretty dumbed down in a cartesian matter where we start from the origin. (**Model space**) This usually doesn't reflect the reality and we want to model it to the common world (**World space**) and then to our eyes perspective (**Eye perspective**).
- **Example:** we create a moon in the object space, we put it in the sky, and then we look at it.
- **Concrete:** An object is defined as $P_{obj} = (x, y, z)^T$ to get it into the world coordinates we need to transform it, namely $M_{obj2world} \times P_obj$ and to get the eye coordinates, $M_{world2eye} \times M_{obj2world} \times P_obj$. **Note:** Since we usually don't care about the steps inbetween, we usually merge $M_{obj2world}$ and $M_{world2eye}$ into one matrix called the **modelview/viewworld** $M_{modelview}$.

Notes about the modelview Matrix $M_{modelview}$:

- OpenGL puts the eye-point at the origin looking towards the negative z-axis
- OpenGL is a state machine: it has an internal memory storage for the modelview matrix
- When calling transformation operations, the kernel constructs a matrix for the transform and right multiplies it with its internal modelview matrix.

OpenGL Modelview Transformation (1 Matrix)

Illustration:

	Memory in Graphics hardware	OGL Commands	Kernel Operations	Comment
Step 0	$M_{modelview}$			Initial value: an identity
Step 1		translate	Construct M_{tran}	Construct a matrix for T
Step 2			$M_{modelview} x M_{tran}$	Right multiply M_{tran} on $M_{modelview}$
Step 3	$M_{modelview} x M_{tran}$			Store the result

Note:

1. OGL always uses right-multiplication whereas DirectX is flexible (?)

Figure 2: Steps to construct MV matrix

3.5 Basic scenegraph concept

- Organize the whole model hiearchy as a tree structure
- Examples of language who do that: VRML, OpenSG

How does this work exactly? We have group nodes, who associate nodes into hierarchies, leaf nodes, who contain all the descriptive data of objects in the virtual world used to render them.

Avoid Modeling Glitches

1. Avoid T-Join
   - Problem: we have edges which sometimes don't touch due to computation and rounding errors of floats
   - Solution: we break a triangle into two triangles.
2. Avoid overlapping polygons in your model
   - Problem: overlapping triangles give flipping colors (**z fighting**)
   - Solution: create another triangle for the overlapping area, or turn off depth test when drawing the next triangle

3.6 Review questions

- Which two input types are the most efficient for rendering polygons and how do they differ?
- Which two data structures exists and what are their benefits and negatives compared to each other?
- What is an hierarchical model?
- What is a modelview and how does it relate to object, world and eye space?
- What are the two main modeling glitches, what causes them and how can you solve them?

4 Lecture 4: Interactive 3D Control

Lecture Outline

1. Interactive Control Concept
2. Interactive Translation relative to screen
3. Interactive rotation: rolling ball

4.1 Interactive Control Concept

- **Idea:** we use the mouse as an input device for 3D viewing control
- **Approach:** by modifying the (base) 4x4 Modelview matrix incrementally accordingly to the mouse motion, we can control our 3D viewing interactively.

Normal Modelview Transformation Equation: assume the modelview matrix is affine. In particular, it has translation, rotation and scaling only. If so we can write the Modelview transformation like this:

$$\begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & T_x \\ m_{21} & m_{22} & m_{23} & T_y \\ m_{31} & m_{32} & m_{33} & T_y \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{bmatrix}.$$

4.2 Interactive Translation relative to screen

Left-Multiply a translation matrix to the existing modelview. We can have a screen-aligned translation, since we don't need to use the fixed point rule for translations.

4.3 Interactive rotation: rolling ball

If we want to rotate our existing modelview, one of the most common mistakes being made is left-multiplying the rotation matrix to our modelview. We need to apply the **fixed point rule**

1. Undo the translation (T_x, T_y, T_z) in the modelview matrix
2. Left-multiply the rotation matrix
3. Redo the translation (T_x, T_y, T_z)

Now we need to construct the rotation matrix, to map our 2D mouse motion to a 3D rotation.

- **Method:** rolling ball
- **Idea:** imagine the object is inside a glass ball of radius r.
- To construct the rotation matrix:
  1. Put the mouse motion in an XY-eye space: (dx, dy, 0)
  2. Axis of rotation perpendicular to the motion vector: (-dy, dx, 0)
  3. Angle of rotation relative to motion vector length: $\sqrt{dx^2 + dy^2}$

5 Lecture 5: Cameras, Projections, and Clipping

Lecture Outline

1. Pin-hole camera
2. Family of Projections
3. Parallel Projection
4. Perspective Projection
5. OpenGL Projection Model: 4x4 projection matrix
6. Clipping and Perspective Division

5.1 Pin-hole camera

- How does a pin-hole camera work: we have a (small) hole to focus all light in an area. The object goes through an barrier and is displayed on film. The image is always inverted
- **Advantages:** easy to simulate, everything is in focus
- **Disadvantages:** needs a bright scene (long exposure), everything is in focus (not realistic in photography)

There is also second method, namely a camera with lens

- How does a camera with a lens work: an object goes through a lens and falls on a focal point. The distance between lens and camera is called focal length. You can compare this to an eye where the eye is the lens and it falls on cones.
- **Advantages:** no need to have a bright scene, not everything is in focus (more realistic in photography)
- **Disadvantages:** need more programming to simulate, not everything is in focus

Now, we don't care about cameras that much obviously, how does it work in programming or even OpenGL? Namely, the OpenGL viewpoint acts as a virtual camera, to define a camera we need the following

1. Viewpoint (what is in the middle of our view)
2. View direction
3. Field of view (in degrees)
4. Film size
5. Projection plane

Viewing requires three elements:

- Objects to be viewed, e.g. whatever you are taking a picture of (1), input geometry (2)
- A viewer with a projection surface, e.g. the camera (1), OpenGL camera (2)
- A projection from the objects to the viewing surface, e.g. defined by the lens, map 3D objects to the 2D surface (1), OpenGL projection matrix (2)

(1) being real life, (2) OpenGL

Planar Geometric Projections

- Standard projections are assumed to be onto a single plane (2D)
- There are two kinds of projections
    1. Perspective projection: all rays converge at a single point (center of projection)
    2. Orthographic projection: all rays are parallel

We need coordinate systems to render geometry objects, in the following way:
Object space $\overset{Lineartrans}{\rightarrow}$ World space $\overset{Lineartrans}{\rightarrow}$ Eye/Camera space $\overset{Non-lintrans}{\rightarrow}$ Screen space $\overset{raster.}{\rightarrow}$ Raster space

Object space

- Most natural coordinate space
- Local coordinate system, local to the object
- Other name: modeling coordinate system

World space

- A coordinate system that is shared by all objects in the modeled scene

Eye/Camera space

- The coordinate system with the eye or the center of projection as its origin
- A space in which the viewing volume is established
- Facilitates clipping (removing out-of-view objects)

Screen space

- Transformation to screen space is a process that describes how light rays reach our eyes
- Screen space is defined to act within a closed volume called the viewing frustum
- Viewing coordinates system, $[x_v, y_v, z_v]$ describes 3D objects with respect to a viewer
- A viewing plane is set up perpendicular to z_v and aligned with (x_v, y_v)
- To set up a viewing plane we need
  - $P = (P_x, P_y, P_z)$ the point where the camera is located
  - L, the point to look at
  - V, the view-up vector whose projection onto the view-plane is directed up

To form a viewing coordinate system we can calculate the following, $Z_v = \frac{P-L}{|P-L|}, X_v = \frac{V \times Z_v}{|V \times Z_v|}, Y_v = Z_v \times X_v$. The transformation matrix M from world-coordinate into viewing-coordinates is defined as

$$M = \begin{bmatrix} x_v^x & x_v^x & x_v^x & 0 \\ y_v^x & y_v^x & y_v^x & 0 \\ z_v^x & z_v^x & z_v^x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = R \cdot T.$$

3D Projection

- We model in 3D, but final display on the monitor is 2D
- Projection refers to the process of projecting the 3D world into 2D, the visualizing of it.
- Two major types of projection: perspective, like the eye; orthogonal projection, preserve length after projection, used in CAD systems

What is a projection: Squeezing a higher dimension into a lower dimension, where in graphics usually 3D to 2D.

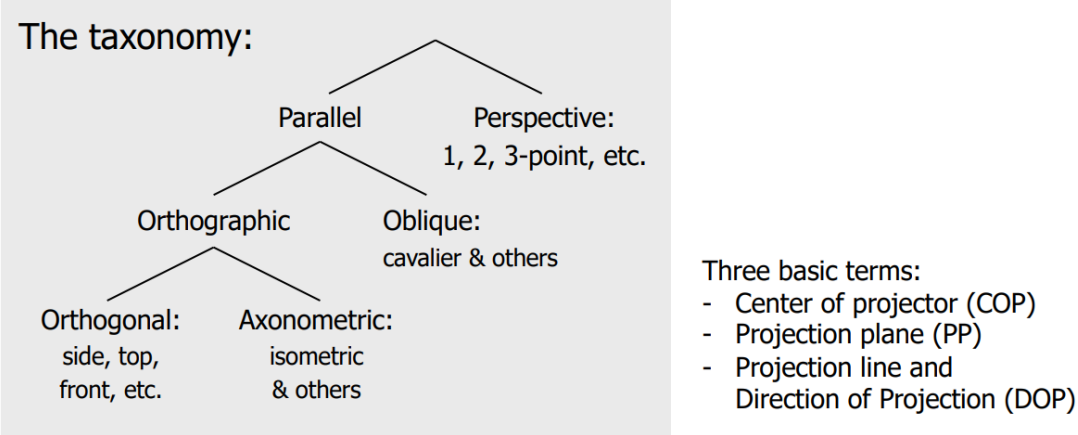


Figure 3: Family of Projections

5.2 Family of Projections

Parallel vs perspective projections

- In 3D we map points from 3-space to the projection plane (PP) along projection lines emanating from the center of projection
- The center of projection (COP) is exactly the same as the pinhole in a pinhole camera

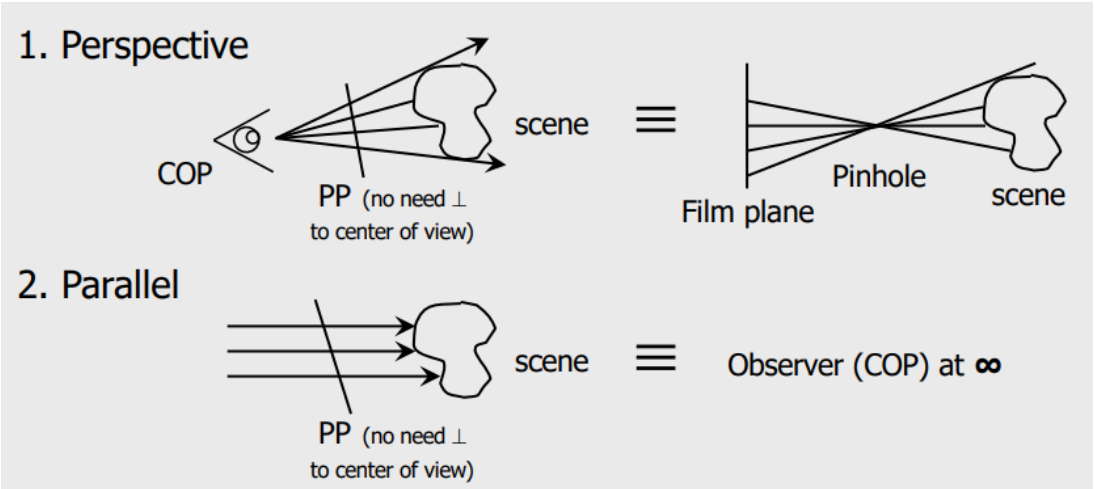


Figure 4: Difference between perspective and parallel

5.3 Parallel Projections

We can specify direction of projection (DOP) because all projections lines are parallel

1. Oblique projection (DOP not perpendicular to PP)
2. Orthographic projection (DOP perpendicular to PP)

Oblique Projection (Parallel)

Like shearing the geometry along PP, two standards of oblique projection:

1. Cavalier projection
   (a) DOP makes 45 degree angle with PP
   (b) Does not foresherten lines perpendicular to PP
   (c) Preserves length
2. Cabinet projection
   (a) DOP makes 63.4 degrees angle with PP
   (b) Foreshorten lines perpendicular to PP by one-half (called Chinese Perspective)

Orthographic Projection (parallel)

1. Orthogonal Projection
   - DOP align with object's X, Y, or Z-axis
   - implementation: just throw away one of the dimensions (set to 0)
   - side, top/plan, or front views
2. Axonometric projection
   - arbitrary DOP, not parallel to world X, Y, nor Z-axis
   - special case: isometric (exactly 120 degrees)

Properties of parallel projection:

- Not realistic looking
- Good for exact measurements, remember for CAD!
- Kind of affine transformations, parallel lines remain parallel, angles not preserved

5.4 Perspective Projection

- COP is at finite distance to the PP, so the further away an object is, the smaller it is
- How we see the nature

Vanishing points

- There exist infinitely many vanishing points
- There can be up to three principal vanishing points (on the axes)
- Perspective projections are categorized by the number of principal vanishing points, equal to the number of principal axes intersected by the viewing plane
- Most commonly used: one-point and two-points perspective

In perspective transformation, parallel lines appear to converge to a single point (easy to see if we have squares and draw parallel lines.)
Vanishing point: points at which parallel lines on PP

Summary of projections

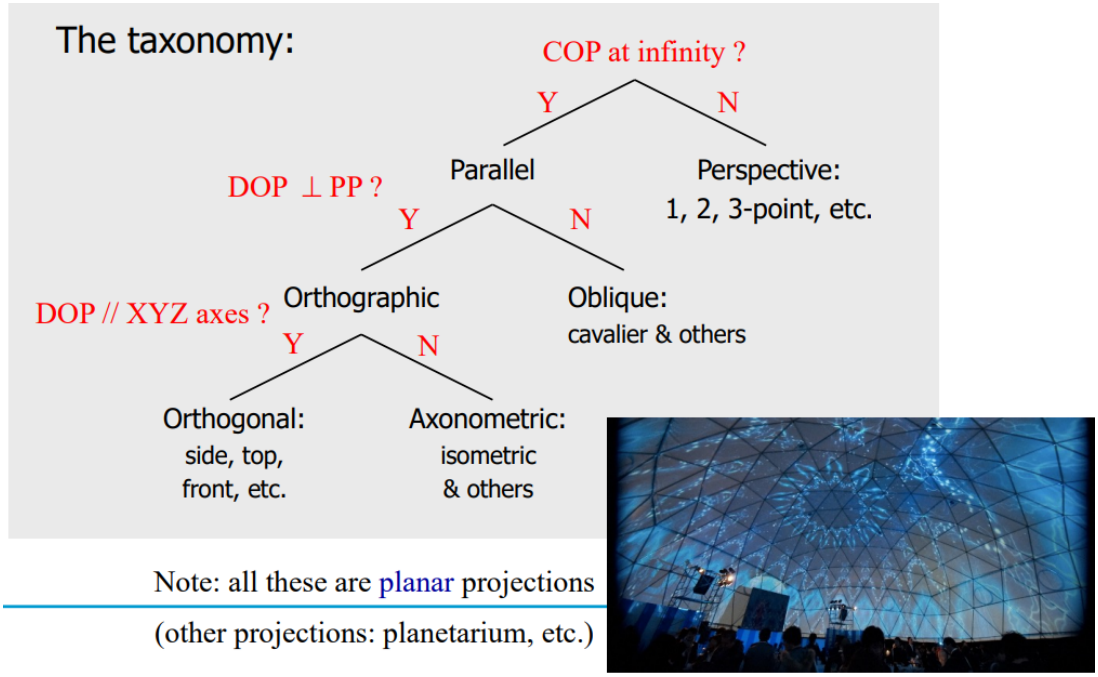


Figure 5: All kinds of projections

View Frustum: the visible "volume"

5.5 OpenGL Projection Matrix: 4x4 projection matrix

OpenGL projection settings:

1. Eye/camera position:
2. Viewing direction
3. View-up direction
4. Viewing region (viewing frustum)
5. Visible depth range

Orthographic Projection in OpenGL

We can use the method *glOrtho(l,r,b,t,n,f)* or *gluOrtho2D(l,r,b,t)*. Where they stand for left, right, bottom, top, neara, far. In gluOrtho2D, near is set to 0, far to 1.

Perspective Concept in OpenGL

Let's say a point is proejcted onto the projection plane, the x and y coordinates are scaled. We can compute it using following: $x_c = -\frac{x}{z}, y_c = -\frac{y}{z}$. For z_c, I can't find it, but a smaller z_c means a closer clip coordinate from the eye.

5.6 Clipping and Perspective Division

Perspective division: $x' = x\frac{c}{w_c}, y' = \frac{y_c}{w_c}, z' = \frac{z_c}{w_c}$

Viewport transformation

glViewport(x,y,width, height) is a command, to define the rectangle where the image is mapped in window pixel coordinates. We need to pay attention the aspect ratio!

6 Lecture 6: Clipping

Lecture contents

- Purpose: eliminate portions of objects outside the viewing Frustum
- View frustum: boundaries of the image plane projected in 3D, a near and far clipping plane

Reasons to use clipping

- Avoid degeneracies: we don't draw stuff behind the eye, and avoid division and overflow
- Efficiency: we don't care about stuff we don't see
- Prevent unnecessary rasterization (which is expensive)

What will come later is culling. The name says enough, if a part is partially in and outside we need to clip, if it's completely outside we need to cull. The final image must be no different than if we didn't do any of this!

There are different strategies

- Don't clip, lol
- Clip during rasterization
- Analytical clipping: alter input geometry

Point Clipping

Given a point (x,y,z) we have to determine whether it's in our viewing volume, which is the easiest. Namely, our view frustum consists of 6 planes, we have to test again each plane. If $H \cdot p < 0$, we reject. Where a plane is defined by $Ax + By + Cz + D = 0$, and $H = (A, B, C, D)$, we assume $D = 1$ for now and $p = (a, b, c, 1)$

Line Clipping

Idea is similar, let's say we have a segment PQ, we check the following for each plane.

- If $H \cdot P > 0$ and $H \cdot q < 0$, then clip q to plane
- If $H \cdot p < 0$ and $H \cdot q > 0$, then clip p to plane
- If $H \cdot p > 0$ and $H \cdot q > 0$, pass through
- If $H \cdot p < 0$ and $H \cdot q < 0$, clipped out

Also, to see the intersection point we can use the maths from one of the first lectures to get it.

There is also another mathematical way, namely computing the sidedness of each vertex with respect to each bounding plane and combine using logical AND.

Cohen-Sutherland Line Clipping Algorithm:

1. First, outcode of end-points pairs are checked for trivial acceptance
2. If the line cannot be trivially accepted, region checks are done trivial rejection
3. If line cannot be trivially accepted or rejected, subdivide so that one or both segments can be discarded
4. These three steps are performed iteratively until what remains can be trivially accepted or rejected

Polygon Clipping

Polygon clipping is symmetric, even when convex. Hard to do when concave. In the most naive case we make $N * m$ intersection and link all segments. We could walk along the boundary CCW and compute intersection points where they enter and leave the window.
Walking rules:

- Out-to-in pair: record clipped point and follow polygon boundary ccw
- In-to-out pair: record clipped point and follow window boundary ccw

Using this we know which part falls within the window.

7 Lecture 7: Rasterization and Anti-aliasing

Lecture outline

1. Introduction to rasterization
2. Rasterization algorithms
3. Introduction to anti-aliasing
4. Anti-aliasing concepts
5. Anti-aliasing in graphics hardware

7.1 Introduction to rasterization

Idea is, we have polygons, with information like color, depth, etc. We want to display them on our display (on the raster), so we need a method to do it, since our screen is only finite. So how does it work:

1. Select pixels on the screen, that best cover the 2D continuous geometry
2. Then interpolate the per-vertex information for each pixel
3. Geometries include: points, lines, triangles, bitmap, etc.
4. Hardware (part of graphics hardware) is called rasterizer

7.2 Rasterization algorithms

Points

There are two ways, namely

- Draw a square (non-antialiased)
- Draw a circle (antialiased)

Shortly, why we can draw circle, is to make it not look as rough, for example for lines we want to smoothen it out since we don't have an infinite amount of pixels.

Line drawing

We need to approximate the mathematical lines if we want to draw a line: $y = mx + b$, which we learned to calculate in school.

Triangle

We subdive polygons into triangles (we can always do this), and now we just have to rasterize the triangles. This can be done efficiently, because of:

1. Scan line coherence: long horizontal stretches of the same fill value
2. Edge coherence: easy to use line algorithm to computer next scan line's end points

7.3 Introduction to anti-aliasing

7.4 Anti-aliasing concepts

7.5 Anti-aliasing in grahics hardware

8 Lecture 8: Hidden Surface Removal

Lecture Outline

1. Introduction Hidden Surface Removal
2. Method 1: Ray Casting
3. Method 2: Z Buffer Method
4. Method 3: Back face culling
5. Method 4: Potentially Visible Set (PVS)
6. Hidden Surface Removal in OGL

8.1 Introduction to Hidden Surface Removal

When we have multiple 3D objects, it's possible we don't see every part, since they could be hidden somewhere. Computationally it's expensive to still render them, and just better to remove them alas *hidden surface removal/elimination*.

There are multiple methods on how to achieve it, for example object order and image order.

Object Order vs image order

- Object order: Each 3D object is considered and drawn once, we could draw same pixels multiple times if they overlap, for example Z-buffer
- Image order: Each pixel is considered once, so we draw one object and move on, we need to compute relationships between objects, for example Ray Casting

Sort first vs Sort last

- Sort first: find the depth-based ordering and draw from back to front, for example Painter's algorithm
- Sort last: sort implicitly as more information becomes available, for example Z-buffer

8.2 Ray Casting

The projection plane (PP) is partitioned into pixels to match screen resolution.

- For each pixel p_i, construct ray (projector) from COP through PP at that pixel and into scene
- Intersect the ray with every object in the scene, color the pixel according to the object with the closest intersection

Implementation

We have to parameterize the ray

$$R(t) = (1 - t)c + tp_i.$$

- If a ray intersects some object O_i, get parameter t such that first intersection with O_i occurs at $R(t_i)$
- We have to find out which object owns the pixel

More detail later in the course

8.3 Z-Buffer method

Idea is to have an additional channel in memory for pixel depth, which is called depth-buffer or Z-buffer (remember from the righthand's rule, z-axis is into or away from the screen). When drawing a pixel, we compare the depth, only draw when it's closer...

Implementation

```
# Clear depth and frame (RGB) buffer
For each pixel pi
    Z-buffer     [ pi ] = FAR
    Frame-buffer [ pi ] = BACKGROUND_COLOR
End

# Render each polygon with Z buffer
For each polygon P
    For each pixel pi in rasterized P
        compute depth z and shade s of P at pi
        if z < Z-buffer [ pi ]
            Z-buffer     [ pi ] = z
            Frame-buffer [ pi ] = s
        End
    End
End
```

Figure 6: Implementation in OGL

Positives and Negatives

- Positive:
  - Runtime is fast
  - Simple to implement
- Negative:

- Suffers from aliasing, e.g. z-fighting
- Cannot render transparency

8.4 Back face culling

- Used together with polygon-based algorithms, if we have solid objects, we don't need to draw polygons that are not in our view. So we can elimnate those (culling).
- Can be used together with ray-casting and Z-buffer algorithms
- We need a normal vector (orthogonal to the plane), we check the angle between the viewer and the normal to some polygon. If it's between -90 degrees and 90 degrees ($\cos\theta \geq 0$), then it's front facing.
- Or verify $N_p \cdot V$, where N_p is the polygon normal, V the line of sigh vector
- Hard to do for concave objects, that's why usually preprocessing step for Z-buffer or ray casting

8.5 Potentially visible set (PVS)

- Acceleration technique
- Data structure (usually pre-computed) that saves what parts of the region is always visible from within the region
- Divide into regions and precompute the objects
- Advantage: good speed up
- Disadvantage: extra memory, preprocessing time, doesn't allow dynamic scenes

8.6 Hidden Surface Removal in OGL

Three related modules:

1. View Frustum Clipping, remove objects outside view
2. Back-face culling
3. Depth-buffering
4. (Or Ray casting must be implented yourself)

Back-Face Culling in OGL

- Done before rasterization step
- OGL functions: glEnable(GL_CULL_FACE) (or disable, which is standard); glFrontFace(mode) specify which is the front face; glCullFace(mode), which side to cull.

Depth Buffering

- Request depth buffer when creating a window, e.g. glutInitDisplayMode(...— GL_DEPTH)
- Always clear depth buffer before rendering a screen again, with glClear and glClearDepth
- Enable depth test, glEnable(GL_DEPTH_TEST)
- glDepthFunc defines the test condition for pixel fragment, default and normal argument is GL_LESS. All with smaller Z-value should pass

9 Lecture 9 - Lighting, Material, and Shading

Lecture Outline

- Introduction
- Light source
- Phong Illumination Model (Lightning Model)
- Interpolative Shading
- Lighting in OGL

9.1 Introduction

Till now we have covered almost everything about how we create an object and display it, what misses is how we colour it or how to set each pixel. This is what we call **shading** (or lighting model, light reflection model, local illumination model, etc...). This is one of the hardest parts to do, since light in our real world is not perfect and gets reflected and absorbed by everything.

9.2 Light source

Firstly, a light source is any object that **emits radiant energy**, think of our own world, sun, lamps, etc. There are many kinds, directional, point, spot lights, but also extended and environment light sources.

Directional Light Source

We assume the light source is infinity away, the incoming light rays are parallel and of same intensity, which makes it all easy to compute. Think of Physics in high school and how we used the sun.

Point Light Source

The light source is at a finite distance from the surface, the light rays go in all directions and is harder to compute since the direction varies at each different surface point. Also the light dimmer with distance, which we have to consider as well (**attenuation**).

Spot Light Source

Points at particular direction, within a certain angle, think of a flashlight, we also have distance attenuation.

Extended Light Source

Extended Light Soruce, like a line or area, where every point alone the line or area is a point light source. This is realistic, but even harder to compute, and we also get shadows. In OpenGL there is no direct support, can only approximate it using ray tracing.

Environment as a Light Source

Why the environment? Everything reflects (except perfect mirrors). Consider it as a directional light source.

9.3 Phong Illumination Model (Lightning Model)

Given a light source, we need to build an approximation of the reality, which we call the **Phong illumination model**. We interaction between light and surface, but it's not a real physical model! It's a common and fast approximation.

Let's build up the model iteratively

1. $I = k_e$, where k_e is the intrinsic shade of the object (color)
2. $I = k_a I_a$, where k_a is the ambient reflection coefficient of the object, I_a is the ambient intensity of the light
3. $I(\kappa) = k_a(\kappa) I_a(\kappa)$, since the above is actually dependent on wavelength, and we do it over all RGB values
4. $I = I_l \cos\theta$, where this is Lambert's cosine law, a surface reflect light equally in all directions (or $k_d I_l \mathbf{N} \cdot \mathbf{L}_+$, where N is the normal at the surface point, L the direction to the light source, which we can rise to n_s for how reflective a material is).
5. Point 4 is intensity, but physics say intensity of a point light source drops off with its distance squared. Normally multiply with $\frac{1}{d^2}$, but that's too harsh so we multiply with $f(d) = min(1, \frac{1}{a+bd+cd^2})$, a, b, c chosen by the user
6. For spots, where the light is perfectly reflected we have a **specular reflection**, so $I = \begin{cases} I_l & \text{if V = R} \\ 0 & \text{otherwise} \end{cases}$, where V is the viewing angle, R the reflection angle
7. Since we have multiple light, we can take the sum of each light and we can put them all together for the Phong Illumination Model

$$I = k_e + k_a I_a + \sum_i f(d_i) I_{li} [k_d (\mathbf{N} \cdot \mathbf{L_i}) + k_s (\mathbf{V} \cdot \mathbf{R})_+^{n_s}].$$

Blinn-phong shading

Computation of $\mathbf{R}$ is expensive and time consuming, so there is the following approximation

$$\mathbf{V} \cdot \mathbf{R} \approxeq \mathbf{N} \cdot \mathbf{H}.$$

, where $\mathbf{H} = \frac{\mathbf{L}+\mathbf{V}}{|\mathbf{L}+\mathbf{V}|}$

Choosing the parameters

- N_s in the range [0,100]
- Try $K_a + K_d + K_s <= 1$
- Use a small K_a (~0.1)

	Ns	Kd	Ks
Metal	Large	Small, color of metal	Large, color of metal
Plastic	Medium	Medium, color of plastic	Medium, white
Planet	0	Varying	0

Figure 7: Choosing parameters

Summary of Phong Illumination Model

	Emission (0th iteration)	Ambient (1st iteration)	Diffuse (2nd iteration)	Specular (3rd iteration)
Depends on surface property	Y	Y	Y	Y
Depends on light intensity	N	Y	Y	Y
Depends on light direction	N	N	Y	Y
Depends on viewing direction	N	N	N	Y

Figure 8: Summary of Phong Illumination Model

9.4 Interpolative Shading

Smooth surface are approximated by polygonal facets, since it gives us better photorealism. For shading, we use a term called **interpolative shading**. Since we are having polygons and triangles, we need to have the visual appeareance that's curved.

Assumptions:

- Approximate normal can be computed for each vertex by averaging the normal of the polygons sharing that vertex
- The shading of a pixel can be obtained by a *bilinear interpolation* or *barycentric-coordinate-based interpolation* of the values of the adjacent vertices

Gouraud Shading

- Calculate the intensity at each vertex using a local reflection model
- Intensities of interior pixels are determined by bilinearly interpolating the vertices'intesities
- There are some equations but I dont understand them

There are some flaws:

- Highlight anomalies, if the highlight is inside a polygon, the shading may fail because are no intensities recorded at the vertices. Solution: smaller and more polygons
- Mach banding: Intensity changed occuring at the boundaries of polygons (of high curvature). Solution: more polygons

Phong Shading

- Instead of interpolating the vertex intensities, we interpolate the vertex normal vector
- No interior highlight problem
- More expensive Gouraud shading, since we do it at each interior surface point

9.5 Lighting in OGL

Lighting in OGL is typically done by:

- Light Source:
  - Ambient light source: glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ...)
  - Directional light: see glLightfv, w is set to 0
  - Point light: see glLightfv, w is nonzero
  - Spot light: CUTOFF and EXPONENT terms
- Surface Property: Material
  - Use glMaterial(arg1, arg2, input values)
  - arg1: GL_FRONT and GL_BACK, etc.
  - arg2: GL_EMISSION, GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_SHINNES, etc. (terms in Phong model)
  - Input values: corresponding coefficient for RGBA

Important notes:

- Check the OGL Red book, simmilar to Phong Illumination Model
- glEnable (GL_LIGHTING), uses its own illumination model, only use it when you need shading, because glColor is much faster

- OGL needs normals, call glNormal befre glVertex (one per face for face normal, one per vertex for vertex normal)
- Try to provide unit vector to OGL when calling glNormal, else if it's not a unit vector or if you have scaling/shearng then you need to call glEnable(GL_NORMALIZE)
- OGL shader program GLSL and CG, or implement yourself