CS3241 Cheatsheet
by Yiyang & Zihao, AY22/23

Overview

Graphics Basics

Addictive Colour - Form a colour by adding amounts of three primaries (RGB).
Subtractive Colour - Form Form a colour by filtering white with Cyan (C), Magenta (M), and Yellow (Y). Note: For subtractive colours, M absorbs G and allows R & B to pass through.

OpenGL Basics

Polygons in OpenGL need to be **Simple** (edges cannot cross), **Convex**, and **Flat** (all vertices in the same plane), or they will not be displayed correctly.

Colour info is stored in each vertex of a polygon, and the shading mode determines how the polygon is coloured:
Smooth ~ - Interpolation of vertex colours across polygon. The default setting. `glShadeModel(GL_SMOOTH)`.
Flat ~ - Fill colour is colour of first vertex. `glShadeModel(GL_FLAT)`.

Hidden Surface Removal - It deals with 3D objects overlapping over one another from our perspective, by using an extra **z-buffer** that saves depth information.

3. Interaction

Overview

Input devices generate **Triggers**, namely signals, and return **Measures**, which are trigger with other meta-info from the input devices, to the OS. Each trigger generates an **Event** whose measure is put in an **Event Queue** to be examined by the user program.
The user program defines a **Callback** for each type of event to GLUT and it is executed when the event occurs.

Positioning in Callbacks

Window systems (such as mouse & motion callbacks) measure positions with origin at **top-left corner**. OpenGL measures positions with origin at **bottom-left corner**. Conversion:

$$x_{\text{opengl}} = x_{\text{win}}, \quad y_{\text{opengl}} = h - 1 - y_{\text{win}}$$

Animation

Double Buffering - The usage of two **colour buffers** for display where the **Front Buffer** displays and the **Back Buffer** is written to. It minimises flickering in animation.
Note: Two identical buffers, switched once writing to back is done.

4. Geometry

Representation

Affine Space - A frame formed by an origin point and basis vectors.

Homogeneous Coordinates ~A 4D system for 3D points & vectors.

- Vector $v = [\alpha_1, \alpha_2, \alpha_3, 0]^T = (\alpha_1, \alpha_2, \alpha_3)$
- Point $p = [\beta_1, \beta_2, \beta_3, w]^T = (\frac{\beta_1}{w}, \frac{\beta_2}{w}, \frac{\beta_3}{w})$

Note: [1] All Homogeneous Coordinate elements are denoted as **Column Vectors**. [2] Point + Vector = Point. [3] Representation for each point is non-unique. We typically use $w = 1$. **Perspective Division** is the process of dividing x, y, z by w.

Transformation

All transformations can be expressed as a 4×4 matrix, **pre-multiplied** to the Homogeneous Coordinates.
Note: $p' = ABp$ is to apply transformation B then A to element p.

Translation

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

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

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

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

Scaling

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = diag(s_x, s_y, s_z, 1)$$

Shear

Shearing along x-axis maps (x, y, z) to $(x + y\cot\theta, y, z)$:

$$H(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note: There is no direct matrix for shearing in OpenGL.

Inverse of Transformation Matrices

- $T^{-1}(d_x, d_y, d_z) = T(-d_x, -d_y, -d_z)$
- $R^{-1} = R^T$, as all R orthogonal
- $S^{-1}(s_x, s_y, s_z) = S(1/s_x, 1/s_y, 1/s_z)$

Combinations of Transformation

General Rotation about Origin: $R(\theta) = R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$, where $\theta_x, \theta_y, \theta_z$ are the Euler angles.
Rotation about Arbitrary Point: $M = T(p_f)R(\theta)T(-p_f)$, where p_f is the point and θ rotational angle.
Note: Matrix ordering is the reverse of the actual transformation's.

OpenGL Transformation

For each matrix mode, **Current Transformation Matrix** (CTM) stores the 4×4 homogeneous coordinate matrix, as part of the state and it is applied to all vertices in the pipeline. Some matrix modes include Model-View (`GL_MODEVIEW`) and Projection (`GL_PROJECTION`). OpenGL also maintains a stack for each matrix mode.

5. Camera & Viewing

OpenGL Spaces: **Object Space**, for modelling each object where the object is centred in the frame. **World Space**, a common frame for all objects, and for defining lightning and camera pose. **Camera Space**, the local frame for camera where it is at the origin and looking into the **Negative z-direction**, initially the same as the world frame.

An OpenGL **Viewport** defines a rectangular region of the window in which OpenGL can draw.
Note: [1] A window can have multiple viewports. [2] A viewport can be larger than the window, and whatever inside the viewport but outside the window will not be displayed.

Transformations

View Transformation - Transform points from World Frame to Camera Frame. The transformation matrix

$$M_{\text{view}} = RT$$
$$= \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where T moves camera position back to world origin, and R rotates axes of camera to coincide with world frame's.
View transformation for normal vectors:

$$M_n = (M_t^{-1})^T$$

, where M_t is the upper left 3×3 sub-matrix of M_{view}.

Projection Transformation - Specify a **View / Clipping Volume** in the camera frame, and project the volume to a $2 \times 2 \times 2$ cube centred at origin, known as the **Normalised Device Coordinate** (NDC) / **Canonical View Volume**.
Note: It preserves depth order and lines.

Viewport transformation The canonical view volume is then mapped to the viewport (from NDC to window coordinates)

$$\frac{x_{NDC}-(-1)}{2} = \frac{x_{win}-x_{vp}}{w} \Rightarrow x_{win} = x_{vp} + \frac{w(x_{NDC}+1)}{2}$$
$$\frac{y_{NDC}-(-1)}{2} = \frac{y_{win}-y_{vp}}{h} \Rightarrow y_{win} = y_{vp} + \frac{h(y_{NDC}+1)}{2}$$
$$z_{win} = \frac{z_{NDC}+1}{2}$$

Projections

Types of projection

- **Perspective projection** - Projectors converge at centre of projection. Objects further from viewer are projected smaller.
- **Parallel Projection** - Projectors are parallel. One special case **Orthographic Projection** is where projectors are orthogonal to projection surface.

Perspective Transformation for Homogeneous Coordinates Any point $q = [x,y,z,1]^T$ projected perspectively onto vertical plane at $z = d < 0$ will be at $(xd/z, yd/z, d)$, equivalent to $p = [x,y,z,z/d]^T = Mq$ for

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}$$

OpenGL Projection Matrix Equivalences

`glOrtho(L, R, B, T, N, F)`:

$$M_{\text{ortho}} = \begin{bmatrix} \frac{2}{R-L} & 0 & 0 & -\frac{R+L}{R-L} \\ 0 & \frac{2}{T-B} & 0 & -\frac{T+B}{T-B} \\ 0 & 0 & \frac{-2}{F-N} & -\frac{F+N}{F-N} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$$= S\left(\frac{2}{R-L}, \frac{2}{T-B}, \frac{2}{N-F}\right) T\left(\frac{-(R+L)}{2}, \frac{-(T+B)}{2}, \frac{F+N}{2}\right)$$

`glFrustum(L, R, B, T, N, F)`:

$$M_{\text{persp}} = \begin{bmatrix} \frac{2N}{R-L} & 0 & \frac{R+L}{R-L} & 0 \\ 0 & \frac{2N}{T-B} & \frac{T+B}{T-B} & 0 \\ 0 & 0 & -\frac{F+N}{F-N} & -\frac{2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Note: The default projection matrix in OpenGL is an identity and orthographic, equivalent to `glOrtho(-1, 1, -1, 1, -1, 1)`.

6. Rendering

Overview

Pipeline Stages

Vertex Process - The stage where Model-View and Projection transformations, and lighting and textures are applied. The result is in Clip Space.
Primitive Assembly - The stage where primitive types are recalled and their processed vertices are collected for polygons, so that we can redefine the primitive for clipping and rasterisation.
Perspective Division - The stage where points with $w \neq 1$ are converted into $w = 1$, from Clip Space into NDC Space.
Viewport Transformation - Transformation into Window Space, with depth range linearly scaled from $[-1,1]$ in NDC to $[0,1]$ in Window Space.
Back-Face Culling - Eliminate polygon if it is **back-facing** and invisible, i.e. when $N_p \cdot N < 0$. Note: Need to specify polygon vertices in Anti-Clockwise order to ensure correct front-face (RHGR).

Visibility & Occlusion Tests

Visibility Tests: Back-Face Culling, Screen Clipping.
Occlusion Tests: Painter's Algorithm, z-Buffer Algorithm.
Note: Perform Visibility before Occlusion Tests.

Clipping

Brute Force Approach

Compute intersections with all sides of the clipping window.
Analysis: Inefficient, require one division per intersection.

2D Cohen-Sutherland Algorithm

Define **Outcode** $b_0b_1b_2b_3$ where bit is 1 iff it is outside $y_{\max}$, $y_{\min}$, $x_{\max}$, $x_{\min}$ respectively (top, btn, right, left) for each endpoint.
For a line with 2 endpoints A and B, four cases to consider:

- `Out(A) == Out(B) == 0` - Both points inside all 4 boundaries. Accept the entire line.
- `Out(A) == 0 && Out(B) != 0` - One point inside all and one outside some. Num of intersections == num of 1 bit in B.
- `Out(A) & Out(B) != 0` - Both points outside the same boundary. Discard the entire line.
- `Out(A) & Out(B) == 0 && Out(A) | Out(B) != 0` - May or may not be intersection. Most complicated. Need to compute intersections for all 4 sides.

Analysis: [1] Efficient, as it filters out simple cases. [2] Yet, still need computation for complicated cases. [3] Can extend to 3D using 6-digit outcodes.

Polygon Clipping

Clipping a polygon may yield multiple polygons, but clipping a **convex polygon** can yield at most one other.
Tessellation - replace concave polygons with a set of smaller simple polygons, typically triangles. Tessellation enables simpler clipping.

AABB Clipping Algorithm

Axis-Aligned Bounding Box (AABB) - 1) Draw a bounding box for the polygon. 2) Consider whether and whereh we need to perform detailed clipping.
Analysis: Similar idea of "easy filtering first".

Rasterisation

Rasterisation - The process of determining which pixels are inside primitive specified by a set of vertices. It produces a set of fragments.
Fragments ~"Potential pixels", with a pixel location and attributes such as colour and depth.

DDA Algorithm

Simple iterative method that moves along one axis one pixel at a time. To draw a line $y = mx + b$ from (x_0, y_0) to (x_e, y_e):

```
for (x=x0, y=y0; x <= xe; x++) {
    write_pixel(x, round(y), color);
    y += m;
}
```

Analysis: [1] Simple, and applicable to curves as well. [2] Inefficient due to many floating point calculations. [3] Lines rasterised are discontinuous when $|m| > 1$. Solution is to flip roles of x and y.

Bresenham's Algorithm

Use a **Decision Variable**, p_k to determine which y-value to rasterise for every x.

1. Input line and store left end in (x_0, y_0). Assume $|m| <= 1$.
2. Calculate constants $\triangle x, \triangle y, 2\triangle y, 2\triangle y - 2\triangle x$.
3. Fill (x_0, y_0), and obtain $p_0 = 2\triangle y - \triangle x$.
4. At each x_k along the line, starting at $k = 0$, perform the following test: [1] If $p_k < 0$, then next point is $(x_k + 1, y_k)$, and $p_{k+1} = p_k + 2\triangle y$. [2] Otherwise, next point $(x_k + 1, y_k + 1)$ and $p_{k+1} = p_k + 2\triangle y - 2\triangle x$.
5. Repeat Step 4 for $\triangle x - 1$ more times.

Analysis: [1] Only applicable to line (and circle) rasterisation. [2] Efficient, due to no floating points.

Polygon Rasterisation Algorithms

Scan-Line Fill Algorithm - [1] First tessellate if concave. [2] Compute colours & depths for vertices. [3] For each horizontal line, interpolate colours and z-values.
Floor Fill Algorithm - Start with one point inside the polygon, expand to 4 directions using BFS recursively.

Hidden Surface Removal

Painter's Algorithm - Render polygons in back-to-front order so that polygons behind others are simply painted over.
Analysis: It involves **Depth Sorting**, with time complexity $O(n \log n)$ for n polygons.

Image-Space Approaches - Look at each pixel / projector and find closest of all polygons along that pixel.
Example: Ray Tracing, z-Buffering (using a z-buffer / depth buffer).
Analysis: Complexity $O(nmk)$ for k polygons and frame buffer sized $n \times m$.

7. Illumination & Shading

Illumination ~To compute colour of a point on the surface, given the light source, and the viewpoint. It can be **Local** (between a light source, a surface point and a viewpoint) or **Global** (of all light sources, surfaces and their reflections & shadows).

Phong Illumination Equation (PIE)

PIE model calculates colour at a point, using **Ambient**, **Diffuse**, and **Specular** terms.

$$I_{\text{Phong}} = I_a k_a + f_{att} I_p k_d (N \cdot L) + f_{att} I_p k_s (R \cdot V)^n$$

Note: [1] All I's are **Luminance** column vectors for RGB values. [2] All **Material Property** constants k's are column vectors $k = (k_r, k_g, k_b)^T$. [3] N, L, R, V are directional (unit) vectors. [4] n is Shininess Coefficient, and $n = 1, \dots, 128$ in OpenGL.

Lamber's Cosine Law - The diffusion intensity is proportionate to the cosine of the angle between the reflection angle and the normal:

$$\text{diffusion reflection} \propto \cos\theta = N \cdot L$$

PIE For Multiple Light Sources

Simply apply each point light sources to each surface for diffuse and specular term

$$I_{\text{Phong}} = I_a k_a + \sum_i f_{\text{att},i} I_{p,i} [k_d (N \cdot L_i) + k_s (R_i \cdot V)^n]$$

Point Source & Attenuation

Attenuation captures the effect that When distance increases, light received decreases, up to the inverse of quadratic terms.

$$f_{\text{att}} I_p = \frac{1}{a + bd + cd^2} I_p$$

Note: a, b, c are customisable parameters in OpenGL, but they are usually disabled, with $a = 1, b = c = 0$.

OpenGL Illumination

Illumination is calculated in **Vertex Processing States** in OpenGL pipeline, after **Model View** transformation and before **Projection Transformation**, i.e. in **Eye Space**.

Shading

Shading How we assembly vertices' information to colour the polygons.

Flat Shading

Flat Shading - Colour the whole polygon with colour of one point on each polygon (in OpenGL the first vertex).
Implementation: `glShadeModel(GL_FLAT)`
Analysis: Distinctive colour differences between neighbouring polygons. Not photo-realistic.

Gouraud Shading

Gouraud Shading - 1) For each vertex, compute average normal vector of polygons sharing the vertex. 2) Apply PIE at the vertex using the average normal vector. 3) For each polygon, interpolate computed colours at the vertices to the interior of the polygon.
Implementation: `glShadeModel(GL_SMOOTH)`
Analysis: [1] **Per-Vertex Lighting** [2] Require polygon connectivity information. [3] No colour difference between neighbouring polygons. [4] Linear interpolation of colour, cannot show specular.

Phong Shading

Phong Shading - 1) For each vertex, compute average normal vector of polygons sharing the vertex. 2) For each fragment in a polygon, interpolate the normal vectors from the vertices. 3) At each fragment, apply PIE on the interpolated normal to compute a colour.
Implementation: Not available in OpenGL unless custom shader.
Analysis: [1] **Per-Pixel Lighting**, so computationally expensive. [2] Require polygon connectivity information. [3] Able to render faithful results, especially highlights.

8. Texture Mapping

Forward Mapping ~Start with a 2D texture space, wrap it around a 3D object to be mapped.
Inverse Mapping ~For each pixel on 3D object surface, draw from the texture space, known as **Pre-image**, during rasterisation.

Inverse Mapping

Surface Parameterisation ~A function mapping between 3D surface point $(x_w, y_w, z_w) \in \mathbb{R}^3$ and 2D texture map $(s, t) \in [0,1]^2$. It usually consists of two steps:

- **S Mapping** (Shape Mapping) - Project 2D texture map onto an "easy" intermediate surface (of a simple 3D geometry)
- **O Mapping** (Object Mapping) - Map from 3D intermediate surface onto object surface.

Texture Filtering - Performance of bilinear interpolation of adjacent **4 texels** at each fragment.

Texture Coordinates Wrapping - OpenGL's handling of vertex texture coordinates that $(s, t) \notin [0,1]^2$, an implementation feature. Two wrapping modes are supported:

- Clamp - Clamp to $[0, 1]$
- Repeat - Ignore integer part of texture coordinates

Anti-Aliasing

Aliasing happens when [1] texture map is point-sampled at each fragment, or [2] during **Texture minification**, where a larger region of texture image is supposed to be mapped to one fragment.

Area-Sampling addresses aliasing - a fragment is mapped to a **quadrilateral** area of pre-image in the texture space.
Mipmapping is one implementation of Area Sampling Solution: 1) A **Mipmap** is created by averaging down the original image successively by halving the resolution. 2) During texture mapping of a fragment, find the appropriate mipmap level according to the amount of texture minification.

Texture Mapping Application

Environment / Reflection Mapping ~a shortcut to render shiny objects with reflection: 1) Capture the image of the surroundings from the position where the object is placed and store in a texture map. 2) During rendering of the object, the reflected eye ray is used to reference the texture map.
Analysis: [1] There is no **self-reflection**. [2] When the object is close, reflection does not look real (distortion arises).

Bump Mapping - Simulation of small complex geometric features on the surface by using textures instead of modelling real geometry.
Analysis: Only applicable to small bumps, may not look normal when viewing from a close distance.

Billboarding - Rendering of simple objects as images that are always rotating and facing the viewer. A type of image-based rendering, used in old 2D games.

9. Ray Tracing

Ray Casting For each pixel, construct a ray from the eye through the pixel, and find the first object it intersects with, if any, and compute shading.
Pros: [1] Ray tracing achieves hidden surface removal implicitly.
Cons: [1] (Whitted) Ray tracing produces hard shadow, and there is inconsistency between highlights and reflections, and there is aliasing. [2] Ray tracing requires entire scene data to be available when computing each pixel.

Whitted Ray Tracing

Whitted Ray Tracing / **Recursive Ray Tracing** - One ray casting algorithm using PIE model for shading, with secondary rays from closest points of intersection, in terms of 1) Reflection Rays, 2) Refraction Rays, and 3) Shadow Rays:

$$I = I_{\text{local}} + k_{\text{rg}} I_{\text{reflected}} + k_{\text{tg}} I_{\text{transmitted}}$$
$$I_{\text{local}} = I_a k_a + I_{\text{source}} [k_d (N \cdot L) + k_r (R \cdot V)^n + k_t (T \cdot V)^m]$$

Here the unit vectors are L: Incident, N: Normal, R: Reflected, V: Observer, T: Refracted.

Reflection & Refraction

Reflection follows that L and R are symmetric about N:

$$R = 2N\cos\theta - L = 2(N \cdot L)N - L$$

Refraction follows **Snell's Law**: $\mu_1 \sin\theta = \mu_2 \sin\phi$. Define $\mu = \mu_1 / \mu_2$ for the two materials:

$$\begin{aligned} T &= -\mu L + \left(\mu\cos\theta - \sqrt{1 - \mu^2(1 - \cos^2\theta)}\right)N \\ &= -\mu L + \left(\mu(N \cdot L) - \sqrt{1 - \mu^2(1 - (N \cdot L)^2)}\right)N \end{aligned}$$

Shadow Rays

At each surface intersection point, a **Shadow Ray** (aka. light ray, shadow feeler) is shot towards each light source to determine any occlusion between light source and surface point.
The local equation with shadow rays:

$$I_{\text{local}} = I_a k_a + k_{\text{shadow}} I_{\text{source}} [k_d (N \cdot L) + k_r (R \cdot V)^n + k_t (T \cdot V)^m]$$

Here, k_{shadow} is the shadow coefficient

- For opaque objects, it is 0 when occluded or 1 when unoccluded
- For translucent occluders, it is attenuated by t_{tg} of the occluder.

Recursive Ray Tracing

Recursive ray tracing can be represented as a **Ray Tree**, and it has the equation

$$\begin{aligned} I(\mathbf{P}) &= I_{\text{local}}(\mathbf{P}) + I_{\text{global}}(\mathbf{P}) \\ &= I_{\text{local}}(\mathbf{P}) + k_{\text{rg}} I(\mathbf{P}_r) + k_{\text{tg}} I(\mathbf{P}_t) \end{aligned}$$

Note: $\mathbf{P}, \mathbf{P}_r, \mathbf{P}_t$ are hit point, hit point by tracing the reflected, and by tracing the transmitted from $\mathbf{P}$.

Cases for recursion termination: [1] When surface is totally diffuse (no reflection) and opaque (no refraction). [2] When rays hit nothing. [3] When a preset maximum recursion depth is reached. [4] When contribution of rays to top level colour negligible, i.e. for some preset small value $\delta > 0$:

$$(k_{\text{rg1}}|k_{\text{tg1}}) \times \ldots \times (k_{\text{rg(n-1)}}|k_{\text{tg(n-1)}}) < \delta$$

Ray-Object Intersection

Let $\mathbf{R}(t) = \mathbf{R}_o + t \times \mathbf{R}_d,\ t \geq 0$ be the ray, for $\mathbf{R}_o, \mathbf{R}_d$ ray origin and ray direction vector, with $|\mathbf{R}_d| = 1$.

Ray-Plane

For plane $\pi(\mathbf{P}) : \mathbf{N} \cdot \mathbf{P} + D = 0$,
Intersection: Solve $\mathbf{N} \cdot \mathbf{R}(t_0) + D = 0$ for $t_0 > 0$ to get t_0 - a) If $t_0 = \infty$ no solution, otherwise b) intersection $\mathbf{R}(t_0)$.
Normal: $\pm\mathbf{N}$.

Ray-Sphere

For sphere centered at origin $\mathrm{T}(\mathbf{P}) : \mathbf{P} \cdot \mathbf{P} - r^2 = 0$ (apply translation if not origin),
Intersection: Solve quadratic equation

$$\mathbf{R}(t) \cdot \mathbf{R}(t) = \mathbf{R}_d \cdot \mathbf{R}_d t^2 + 2\mathbf{R}_d \cdot \mathbf{R}_o t + \mathbf{R}_o \cdot \mathbf{R}_o - r^2 = 0$$

Choose t_0 as the closest positive t value if exists.
Normal: $\mathbf{R}(t_0)/|\mathbf{R}(t_0)|$.

Ray-Box

For a **axis-aligned** 3D box specified by coordinates of two diagonally opposite corners,
Intersection: 1) For each parallel plane pair, find distance to first and second plane from ray origin, $0 < t_{\text{near}} \leq t_{\text{far}}$ 2) Repeat for all 3 pairs, and keep $\max t_{\text{near}}$ and $\min t_{\text{far}}$. 3a) if $t_{\text{near, max}} > t_{\text{far, min}}$, no intersection, otherwise 3b) intersect at $t_{\text{near, max}}$.

Ray-Triangle

For a triangle with vertices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, the **Barycentric Coordinates** of an interior point $\mathbf{P}$ is $(\alpha, \beta, \gamma) \in [0, 1]^3$ that

$$\mathbf{P} = \alpha\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C},\ \alpha + \beta + \gamma = 1$$

Intersection: Solve 4 linear equations (sum to 1, and each coordinate value of $\mathbf{P}$) involving α, β, γ, t.
Normal: $\mathbf{N}_P // \alpha\mathbf{N}_A + \beta\mathbf{N}_B + \gamma\mathbf{N}_C$, need normalisation.

The Epsilon Problem

The Epsilon Problem - Ray traced image with noisy shadows due to false intersection for very small positive t.
Solution: [1] Pre-determine a small value $\epsilon > 0$, and accept intersection for $t > \epsilon$ instead of $t > 0$. [2] Advance ray origin by $\epsilon\mathbf{R}_t$ when a new ray is spawned.

10. Curves

Parametric Polynomial Representations

A 3D parametric polynomial curve of degree n is

$$\mathbf{p}(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix} = \sum_{k=0}^{n} u^k \mathbf{c}_k, \text{ where } \mathbf{c}_k = \begin{bmatrix} c_{xk} \\ c_{yk} \\ c_{zk} \end{bmatrix}$$

Note: [1] There are $3(n+1)$ coefficients needed. [2] A **curve segment** is defined $u_{min} \leq u \leq u_{max}$, and we typically choose $0 \leq u \leq 1$.

A 3D parametric polynomial surface is

$$\mathbf{p}(u, v) = \begin{bmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{bmatrix} = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{c}_{ij} u^i v^j$$

Note: [1] There are $3(n+1)(m+1)$ coefficients in $\{c_{ij}\}$. [2] Normally we choose $n = m$. [3] A **Surface Patch** is defined for $0 \leq u, v, \leq 1$.

Joint Curves

Long curves can be drawn by joining multiple curve segments of lower degree. A **Joint Point** is where the endpoints of two curve segments meet.
We typically use **Cubic** parametric polynomials. Rationale: [1] Local control of shape. [2] Stability, as higher terms are more sensitive to small input changes.

Geometric Properties of Joining Curves

Gⁿ Parametric Continuity ~Whether two curve segments have n-th order gradient **in the same direction**, for $n = 1, 2, \ldots$
Cⁿ Parametric Continuity ~Whether two curves segments have identical n-th order gradient, for $n = 0, 1, \ldots$.
Analysis: C^n is stronger than G^n, for all $n = 1, 2, \ldots$.
Examples: Assuming two segments have joining point $\mathbf{p}(0) = \mathbf{q}(1)$, then $C^0 : \mathbf{p}(1) = \mathbf{q}(0)$, $C^1 : \mathbf{p}'(1) = \mathbf{q}'(0)$, and $G^1 : \mathbf{p}'(1) = \alpha\mathbf{q}'(0)$ for some $a0$.

Cubic Parametric Curves & Surfaces

$$\mathbf{p}(u) = \mathbf{c}_0 + \mathbf{c}_1 u + \mathbf{c}_2 u^2 + \mathbf{c}_3 u^3 = \sum_{k=0}^{3} u^k \mathbf{c}_k = \mathbf{u}^T \mathbf{C}$$

Here $\mathbf{C} = [\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3]^T \in \mathbb{R}_{4\times}3$, $\mathbf{u} = [1, u, u^2, u^3]^T \in \mathbb{R}_{4\times 1}$.

Supposed the curve interpolates $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ corresponding to values $u = 0, 1/3, 2/3, 1$, to determine coefficients

$$\mathbf{P} = \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} = \mathbf{AC} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & \frac{1}{3} & \frac{1}{9} & \frac{1}{27} \\ 1 & \frac{2}{3} & \frac{4}{9} & \frac{8}{27} \\ 1 & 1 & 1 & 1 \end{bmatrix} \mathbf{C}$$

Then we have $\mathbf{C} = \mathbf{M}_I\mathbf{P}$, for $\mathbf{M}_I$ the **Geometry Matrix**,

$$\mathbf{M}_I = \mathbf{A}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -5.5 & 9 & -4.5 & 1 \\ 9 & -22.5 & 18 & -4.5 \\ -4.5 & 13.5 & -13.5 & 4.5 \end{bmatrix}$$

To get a new point on curve, $\mathbf{p}(u) = \mathbf{b}(u)^T\mathbf{P}$ where $\mathbf{b}(u)^T = \mathbf{M}_I^T\mathbf{u}$ is the **Blending Function**, consisting of 4 **Blending Polynomials**.
Note: $\mathbf{M}_I$ and $\mathbf{b}(u)$ are the same for all curves specified this way.

Cubic Bezier Curves & Surfaces

Bezier Curve - Given control points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$, we want a curve that satisfies

$$\begin{aligned} \mathbf{p}(0) &= \mathbf{p}_0 \\ \mathbf{p}(1) &= \mathbf{p}_3 \\ \mathbf{p}'(0) &= 3(\mathbf{p}_1 - \mathbf{p}_0) \\ \mathbf{p}'(1) &= 3(\mathbf{p}_3 - \mathbf{p}_2) \end{aligned}$$

The **Bezier Geometry Matrix** and **Blending Functions** are

$$\mathbf{M}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix},\ \mathbf{b}(u) = \begin{bmatrix} (1-u)^3 \\ 3u(1-u)^2 \\ 3u^2(1-u) \\ u^3 \end{bmatrix}$$

Others

From Tutorials & Assignments

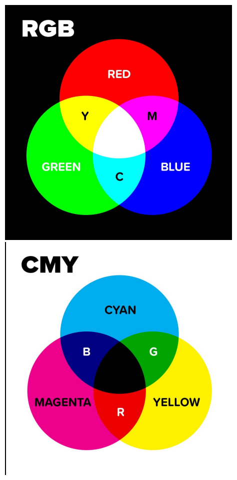
(Tut1 Qn2) Human eyes' sensitivity to change in colour: G > R > B.
(Tut5 Qn4) **z-Fighting** ~when different fragments at the same pixel have the same or very similar z-buffer values. Solution: [1] Minimise distance `far-near`. [2] Use z-buffers with higher precision.

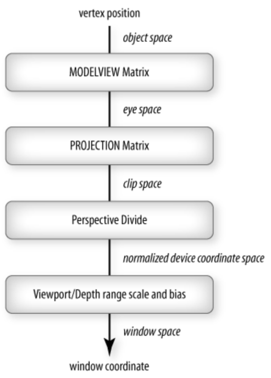
From Past Year Papers

(AY19/20 Sem1 Midterm Qn7&8) Vertices for Primitive Assembly are in Clip Space. Vertices for Rasterisation are in Window Space.
(AY20/21 Sem1 Midterm Qn22) z-fighting is more serious for polygons further away in a **perspective** projection; z-fighting is unaffected by how far a polygon is in a **orthographic** projection.
(AY20/21 Sem1 Midterm Qn24) Back-Face Culling can be performed in Window Sapce or Camera Space, but cannot be performed in the Vertex Processing stage.

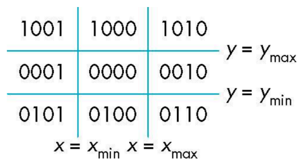
Diagrams

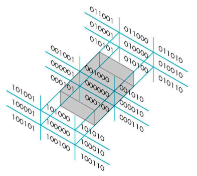
Colour Theory, Transformation Pipeline



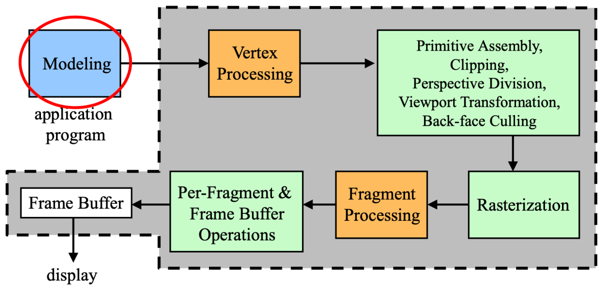


Outcode



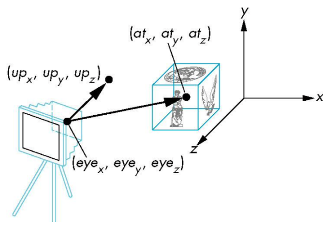


Rendering Pipeline


Modeling
application program
Vertex Processing
Primitive Assembly, Clipping, Perspective Division, Viewport Transformation, Back-face Culling
Rasterization
Fragment Processing
Per-Fragment & Frame Buffer Operations
Frame Buffer
display

gluLookAt Coordinate Specification
$(u, v, n$ for x, y, z of the Camera$)$

- $\mathbf{n} = \text{normalize}(\,\mathbf{eye} - \mathbf{at}\,)$
- $\mathbf{u} = \text{normalize}(\,\mathbf{up}\,) \times \mathbf{n}$
- $\mathbf{v} = \mathbf{n} \times \mathbf{u}$



OpenGL Reference

Miscellaneous

Settings

`glutInitDisplayMode(...|GLUT_DOUBLE)` uses Double Buffering, as compared to `GLUT_SINGLE` for Single Buffering.

`glViewport(u, v, w, h)` - Define the rectangular viewport.

Callback-Related

`glutDisplayFunc(void (*func)(void))` - when GLUT determines the window to be refreshed (e.g. opened, reshaped, exposed etc.)
`glutIdleFunc` - when there is no trigger (e.g. for animation).
`glutMotionFunc` - when mouse on hold and moving.
`glutPassiveMotionFunc` - when mouse moving but not on hold.
`glutTimerFunc(unsigned int msecs, void (*func)(int value), value)` - Set a timer and trigger the callback when timer elapses.
Other common trigger callbacks: `glutMouseFunc`, `glutReshapeFunc`, `glutKeyboardFunc`.

`glutPostRedisplay()` - Set a flag for posting redisplay, which avoids frequent repeated execution.

OpenGL Matrices & Transformation

`glMatrixMode(GLenum mode)` - Specify the current matrix mode. Modes are `GL_MODELVIEW`, `GL_PROJECTION`, `GL_TEXTURE` or `GL_COLOR`.

Transformations

`glLoadIdentity()` - Override CTM with I_4.
`glTranslatef(dx, dy, dz)` - Post-mul. translation matrix to CTM.
`glRotatef(theta, vx, vy, yz)` - Post-mul. rotation matrix to CTM.
`glScalef(sx, sy, sz)` - Post-mul. scaling matrix to CTM.
`glLoadMatrixf(m)` - Override CTM with matrix $m_{4\times4}$.
`glMultMatrixf(m)` - Post-mul. $m_{4\times4}$ to CTM.
Note: [1] Each has a Double(`gl...d`) and a Float (`gl...f`) format. [2] m is a 1D array of length 16, ordered by column.

Matrix Stacks

`glPushMatrix()` - Push CTM to the stack, and duplicate the CTM.
`glPopMatrix()` - Pop top matrix from the stack and replace CTM.
Note: Typically an indentation is applied for every Push-Pop pair.

Camera & Viewing

`gluLookAt(eyex, eyey,eyez, atx, aty, atz, upx, upy, upz)`
- Create a view transformation matrix for the camera specified. `eye*` specify the camera location, `at*` specify where to look at, and `up*` is **Up-Vector** for orientation of camera.
Note: [1] The up vector does not need to be perpendicular to Eye-At. [2] The up is usually $(0, 1, 0)$.

`glOrtho(left, right, bottom, top, near, far)` - Post-mul. to CTM the orthographic projection matrix.
Note: It will map $(-\text{near}, -\text{far})$ to $(-1, 1)$.

`glFrustum(left, right, bottom, top, near, far)` - Post-mul. to CTM the perspective projection matrix.
Note: [1] LRBT all refer to the locations on the near plane. [2] It will map $(-\text{near}, -\text{far})$ to $(-1, 1)$.

`gluPerspective(fovy, aspect, near, far)` - Post-mul. to CTM the ”central & symmetric” perspective projection matrix.
Note: `fovy` vertical field of view angle in degrees. `aspect` w/h ratio.

`glViewport(x, y, width, height` - Specify transformation matrix for xy from NDC to window coordinates.