CS3241 Cheatsheet
by Yiyang, 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).

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.

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: They are two identical buffers, switched once writing to back buffer is done.

Geometry

Representation

Affine Space - A frame formed by a point as origin and the basis vectors.

Homogeneous Coordinates ~A 4D representation for both 3D points and 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: The Homogeneous Coordinate representation for each point is non-unique. We typically use $w = 1$ by default.
Note: All Homogeneous Coordinate elements are denoted as **Column Vectors**.

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

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

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.

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.

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.

Transformations

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

$$M_{\text{view}} = RT$$

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.

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)`:

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

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

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

Rendering

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, at least one intersection.
- `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.

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.
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.
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 orientation for convex 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 $n \times m$.

OpenGL Reference

Miscellaneous

Settings

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

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.