University of Birmingham

Computer Science Year 2

Faisal IMH Alrajhi

Year 2 Study Guide

Contents

Chapter 1

Graphics

1.1 Surface Geometry

This section covers the basics introduced in how to represent shapes in a computer.

1.1.1 Notes

- Graphics Pipeline: It refers to the sequence of steps used to create a 2D raster representation of a 3D scene. It is the process of turning a 3D model into what the computer displays.
- Vertex: A point with three numbers representing its XYZ position in a plane
- Edge: An edge is the difference between two vertices; the segment connecting them
- Surface: A closed set of edges representing a face of a 3D object
- Polygon: A shape in space usually representing by a set of surfaces (other methods listed below)
- Polygon Table: A table containing a set of either vertices, edges and/or surfaces that is used to define the boundaries of a polygon. This is one method to define Polygons.
- Delaunay Triangulation: Given a set P of points in a plane, creates a triangular mesh DT(P) such that no point in P is inside the circumcircle of any triangle in DT(P).

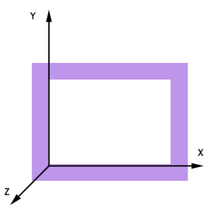

Figure 1.1: Coordinate system assumed throughout module

The default coordinate system assumed is right-handed: the positive x and y axes point right and up, and the negative z axis points forward. Positive rotation is counterclockwise about the axis of rotation.
Polygon Table consistency checks:

1. Every vertex is listed as an endpoint of at least two edges
2. Every surface is closed
3. Each surface has at least one shared edge

The order the vertices/edges are listed in a Geometric Polygon table do matter. Vertices written in clockwise order represent a surface pointing outwards. Whereas listing them counterclockwise represents an inwards pointing surface.

Meshes are a wireframe representation in which all vertices form a single set of continuous triangles, and all edges are a part of at least two triangles. Meshes can be generated by triangulation; but we covered just Delaunay Triangulation, defined above. Meshes can also be progressive. Detail in meshes is unnecessary at farther distances, so vertices can be removed and added to create less detailed or more detailed meshes, respectively. Progress meshes do this dynamically based on viewer distance.

There are a few ways to represent polygons in a space, with boundary representations being only one method.

1. Boundary Representation: Using vertices and drawing edges and surfaces from them
2. Volumetric Models: Using simple shapes and various operations to create more complex shapes
3. Implicit Models: Using implicit equations, such as that of a sphere, to generate shapes
4. Parametric Models: Uses parametric equations to plot the multiple axes of a shape

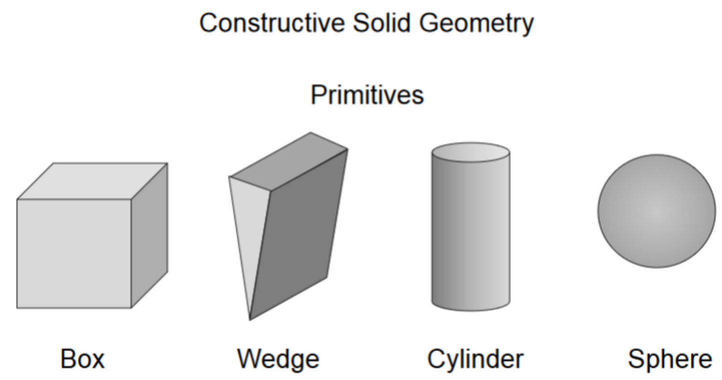

Figure 1.2: Constructive Solid Geometry (CSG) Primitives

We covered a few volumetric models in the module.

1. CSG: Uses primitve shapes and combines them uses set operations (union, difference, exclude, etc.) to generate new, more complex shapes.
2. Voxels: 3D Pixels, unit cubes
3. Octrees: Quad trees that divide in 3D space. Individual partitions are voxels
4. Sweep: Using a 2D shape, moves that shape across a path, generating a volume in position the 2D shape occupies during its path

One can also use implicit or parametric equations to generate shapes. Below is a list of equations that are common.

2D Circle:

$$\left(\frac{x}{r}\right)^2 + \left(\frac{y}{r}\right)^2 = 1 \tag{1.1}$$

2D Circle - Parametric:

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \\ -\pi \leq \theta &\leq \pi \end{aligned} \tag{1.2}$$

2D Ellipse - Parametric:

$$\begin{aligned} x &= r_x\cos\theta \\ y &= r_y\sin\theta \\ -\pi \leq \theta &\leq \pi \end{aligned} \tag{1.3}$$

3D Sphere:

$$\left(\frac{x}{r}\right)^2 + \left(\frac{y}{r}\right)^2 + \left(\frac{z}{r}\right)^2 = 1 \tag{1.4}$$

3D Ellipsoid:

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1 \tag{1.5}$$

3D Sphere - Parametric:

$$\begin{aligned} x &= r\cos\phi\cos\theta \\ y &= r\cos\phi\sin\theta \\ z &= r\sin\phi \\ -\pi &\leq \theta \leq \pi \\ -\pi/2 &\leq \phi \leq \pi/2 \end{aligned} \tag{1.6}$$

3D Ellipsoid - Parametric:

$$\begin{aligned} x &= r_x\cos\phi\cos\theta \\ y &= r_y\cos\phi\sin\theta \\ z &= r_z\sin\phi \\ -\pi &\leq \theta \leq \pi \\ -\pi/2 &\leq \phi \leq \pi/2 \end{aligned} \tag{1.7}$$

1.1.2 Examples

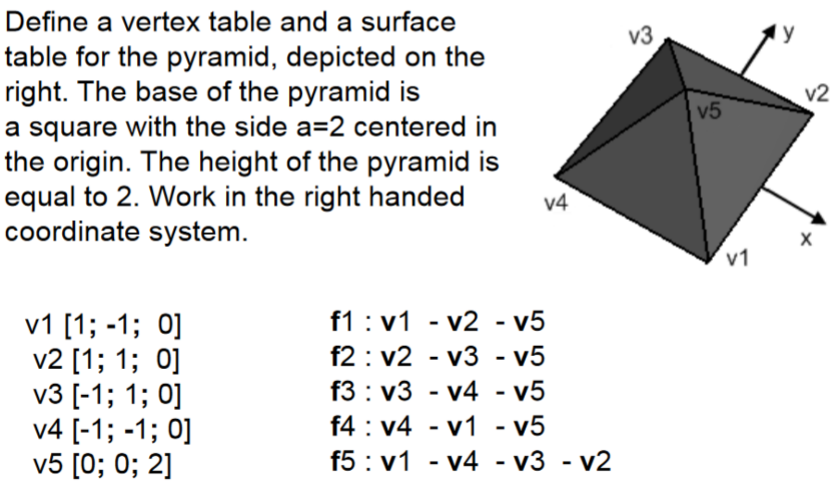

Figure 1.3: Example from Lecture

Further Examples are taken from quizzes and assignments

Consider the following vertex table and edge table for a convex 3D shape. Create the corresponding polygon (surface) table for that shape. Use vertex indices in your table.

vertex table

vertices	x	y	z
V1	2	0	-2
V2	0	1	-3
V3	1	2	-5.5
V4	2	3	-8
V5	4	3	-9
V6	5	1	-5.5
V7	4	2	4

edge table

E1	V7	V1
E2	V7	V2
E3	V7	V3
E4	V7	V4
E5	V7	V5
E6	V7	V6
E7	V1	V2
E8	V2	V3
E9	V3	V4
E10	V4	V5
E11	V5	V6
E12	V6	V1

Figure 1.4: Example from Quiz

Surfaces:
S1 = V1, V2, V3, V4, V5, V6
S2 = V1, V7, V2
S3 = V2, V7, V3
S4 = V2, V7, V3
S5 = V4, V7, V5
S6 = V5, V7, V6
S7 = V6, V7, V1

1.1.3 Normal Vectors

The normal vector of a surface points outwards from the surface. This is later used for lighting, projection and culling. Calculating normal vectors is a fairly simple task. For boundary polygons, the normal of a face is the cross product of two edges. Assuming vectors A and B, the cross product is the determinant of the following matrix;

$$\begin{bmatrix} i & j & k \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{bmatrix} \tag{1.8}$$

Which can be minimized to the following (longer) equation;

$$N = \begin{bmatrix} A_yB_z - A_zB_y \\ A_zB_x - A_xB_z \\ A_xB_y - A_yB_x \end{bmatrix} \tag{1.9}$$

To know which vectors to use for A and B, simply select an edge on a surface, and you use your right hand with your thumb pointing outwards and curl your hand around in the direction until the first vector hits your hand. Alternatively, you can piece it together by looking at the figure.

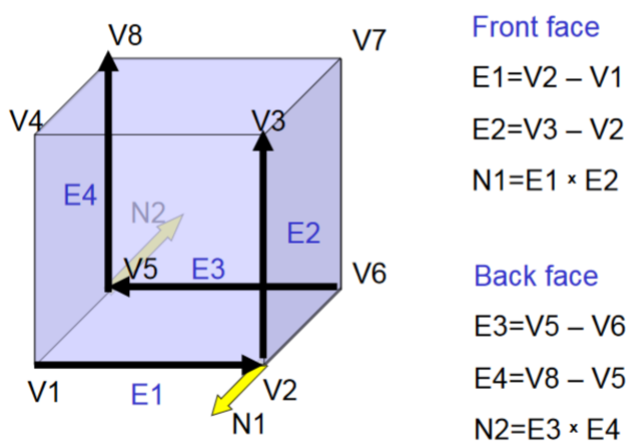

Figure 1.5: Normal Vector of cube from lecture

1.1.4 Further Sources

Surface Representations
Alternate Lecture

1.2 Transforms

This section covers simple transformation matrices.

1.2.1 Notes

- Transformation: a function that can be applied to each of the points in a geometric object to produce a new object.
- Translation: A geometric transform that adds a given translation amount to each coordinate of a point. Translation is used to move objects without changing their size or orientation.
- Rotation: A geometric transform that rotates each point by a specified angle about some point (in 2D) or axis (in 3D).
- Scaling: A geometric transform that multiplies each coordinate of a point by a number called the scaling factor. Scaling increases or decreases the size of an object, but also moves its points closer to or farther from the origin.

Transformations are applied to geometric objects to move them around. This is valuable when considering camera positions, or when laying out a world in a video game. Transformations can be applied as equations for each dimensions eg.

$$T_x = x + t_x$$

is the new x-position when applying the translation. However, there is a lack of uniformity between different transforms, some requiring x and y more than once, others being matrices. To standardize transforms, we instead use *Homogeneous Transformation Matrices*. Convert a 2D point to a 3D point by setting z = 1, and apply the transforms as matrices by replacing the variable found in each matrix template. This is an easy way of standardizing the equations, and allows for easy transforms by multiplying the transformations together before multiplying them with the point.
For example, applying a Translation T and then a Rotation R can be done by multiplying RT first and then multiplying the new transform matrix with the original points. This also makes it more efficient to move more than one point when they share the same transform, as it only need one multiplication per-point rather than one per-transform per-point.

1.2.2 Transformation Matrices

$$T_{2D} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1.10}$$

$$S_{2D} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.11}$$

$$R_{2D} = \begin{bmatrix} \cos\theta & -\sin\theta & T_x \\ \sin\theta & \cos\theta & T_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1.12}$$

$$T_{3D} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.13}$$

$$S_{3D} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.14}$$

$$R_{3D_X} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.15}$$

$$R_{3D_Y} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.16}$$

$$R_{3D_Z} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.17}$$

1.2.3 Examples

MATLAB Assignment Rotating

```
    2. Apply rotation transformation to bowl object mesh model
    %%% Define matrices for rotation of the bowl object around x, y and z ...
        axes by 20, 80 and 55 degrees respectively.
    %%% Apply these three transformations to the original ...
        (non-transformed) bowl object in the given order
    %%% and visualize the result using trisurf function.
    %%% Save the result of your visualization to "2.png" file and include ...
        this file in your submission.
    rx = [1 0 0 0; 0 cosd(20) -sind(20) 0; 0 sind(20) cosd(20) 0; 0 0 0 1];
    ry = [cosd(80) 0 sind(80) 0; 0 1 0 0; -sind(80) 0 cosd(80) 0; 0 0 0 1];
    rz = [cosd(55) -sind(55) 0 0; sind(55) cosd(55) 0 0; 0 0 1 0; 0 0 0 1];
    rotated_object_vertices = rz*ry*rx*obj_v4;
```

Scaling

```
%% 3. Apply scaling transformation to bowl object mesh model
%%% Define a matrix for scaling of bowl object with scaling factor f = ...
    [3.5, 1.5, 2] in direction of x, y and
%%% z axes. Apply this matrix to your original bowl object ...
    (non-transformed) and visualize the result
%%% using trisurf function. Save the result of your visualization to ...
    "3.png"? file and include this file in your
%%% submission.

scaling = [3.5 0 0 0; 0 1.5 0 0; 0 0 2 0; 0 0 0 1];
scaled_object_vertices = scaling*obj_v4;
```

Translating

```
%% 4. Apply translation transformation to bowl object mesh model
%%% Define a matrix for translation of bowl object by [−500, 50, −100] in ...
    direction of x, y and z axes. Apply
%%% this matrix to your original bowl object and visualize the result ...
    using trisurf function. Save the result
%%% of your visualization to "4.png"? file and include this file in your ...
    submission.

translate = [1 0 0 −500; 0 1 0 50; 0 0 1 −100; 0 0 0 1];
translated_object_vertices = translate*obj_v4;
```

3-in-1 Wombo Combo

```
%% 5. Apply all 3 transformations defined above to your original ...
    (non−transformed) bowl object one after the other in the given order.
%%% Display transformed meshes in the figure using trisurf.
%%% Save the result of your visualisation to "5.png" file and include ...
    this file to you submission folder.

%% Compute transformations (4x4 transformation matrices)

object_transformation = translate*scaling*rz*ry*rx;
```

1.2.4 References

Quick Overview

1.3 Lighting

This section covers things related to lighting and shading of objects in a scene.

1.3.1 Notes

- Diffuse: Non-shiny illumination
- Specular: Shiny reflections
- Ambient: background illumination

Ambient Light

- Global background light
- No direction
- Does not depend on anything

Diffuse Light

- Parallel Light Rays originating from a source direction
- Contributes to Diffuse and Specular Term

Spot Light

- Originates from a single source point
- Conic dispersion of light, intensity is a function of distance
- More realistic

Surface Properties

- Geometry - Position, orientation
- Colour - reflectance and Absorption spectrum
- Micro-structure - defines reflectance properties

Shading Models

- Flat shading is the simplest shading model. Each rendered polygon has a single normal vector; shading for the entire polygon is constant across the surface of the polygon. With a small polygon count, this gives curved surfaces a faceted look.
- Phong shading is the most sophisticated of the three. Each rendered polygon has one normal vector per vertex; shading is performed by interpolating the vectors across the surface and computing the color for each point of interest.
- Gouraud shading is in between the two: like Phong shading, each polygon has one normal vector per vertex, but instead of interpolating the vectors, the color of each vertex is computed and then interpolated across the surface of the polygon.

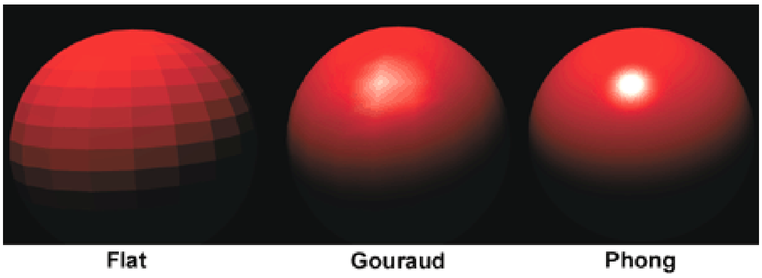

Figure 1.6: Shading Model Differences

1.3.2 Phong Shading Equation

$$\begin{aligned} Colour &= Ambient + Diffuse + Specular \\ Colour &= I_a K_a + I_d K_d \cos\theta_L + I_s K_s \cos^n \theta_S \end{aligned} \tag{1.18}$$

Ambient Term This is very easy. It is the K_a term multiplied with the Ambient intensity I_a.

$$Ambient = I_a K_a \tag{1.19}$$

Diffuse Term The diffuse term is usually straight forward.. It is the K_d term multiplied with the Light source intensity I_d. The angle θ between the light source and the normal of the surface is then computed, and the $cos(\theta)$ is multiplied to obtain the full term.

$$Diffuse = I_d K_d \cos \theta_L \tag{1.20}$$

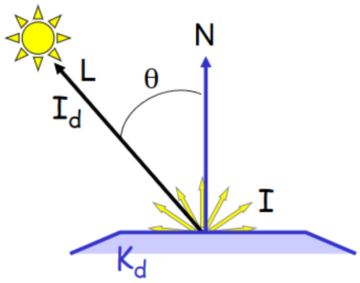

Figure 1.7: Diffuse term overview

Specular Term The specular term needs a few more steps. It is the K_s term multiplied with the reflected Light source intensity I_s. This is the ray that is bounced off of the surface, and is θ_L away from the normal of the surface. This intensity is multiplied by the cos of the angle θ_S, the angle between the reflected ray and the line of sight from the camera. The cos is raised to the n_{th} power, a factor known as a shininess factor that is usually given.

$$Diffuse = I_d K_d \cos^n \theta_S \tag{1.21}$$

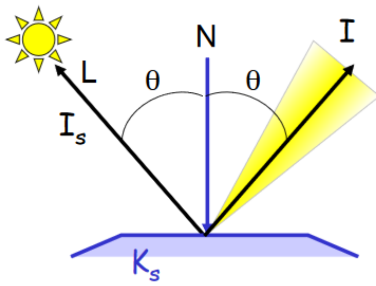

Figure 1.8: Specular term overview

1.3.3 Examples

MATLAB Assignment

Calculating Ambient Term

```matlab
function colour = calcAmbient_skeleton(pixel_colour_current, Ia,Ka)

% Colour at current pixel
colour = pixel_colour_current;

% TO DO: Compute colour of the pixel here and write values to the
% corresponding place in image ImA

ambient = Ia.* Ka;
for i = 1:size(colour, 3)
colour(:,:,i) = ambient(i,i,:);
end
end
```

Calculating Diffuse Term

```matlab
function colour = calcDiffuse_skeleton(pixel_colour_current, ...
    point_position, light_position, surface_normal, Id, Kd)

% Colour at current pixel
colour = pixel_colour_current;

%TO DO: Calculate light direction from light position and point position
light_direction = light_position - point_position;
%TO DO: Calculate normalised light direction
normalised_light = light_direction / norm(light_direction);
%TO DO: Calculate cos light direction, removing negative
%values
cos_light = dot(surface_normal, normalised_light);

%TO DO: Compute colour colour of the pixel here and write
% values to the corresponding place in image ImD
diffuse = Id.*(Kd.*cos_light);
for i = 1:size(colour, 3)
colour(:,:,i) = colour(:,:,i) + diffuse(i);
end
end
```

Calculating Specular Term

```
function colour = calcSpecular_skeleton(pixel_colour_current, ...
    point_position, light_position, surface_normal,normal_towardsViewer, ...
    shininess_factor, Is, Ks)

% Colour at current pixel
colour = pixel_colour_current;

%TO DO: Calculate light direction from light position and point position
light_direction = point_position - light_position;

%TO DO: Normalised light direction
normalised_light = light_direction / norm(light_direction);
%TO DO: Normal component
n = surface_normal / norm(surface_normal);
%TO DO: Reflected Ray (tangent + ray in one step)
R = n*2*dot(n, -1*normalised_light) + normalised_light;
%TO DO: Normalised Reflected Ray
normalised_reflection = R / norm(R);
%TO DO: Calculate cos_spec, removing negative values
cos_light = dot(normal_towardsViewer, normalised_reflection);
if (cos_light < 0)
cos_light = 0;
end
%TO DO:  compute colour colour of the pixel here and write
% values to the corresponding place in image ImS
specular = Is.*(Ks.*(cos_light^shininess_factor));
for i = 1:size(colour, 3)
colour(:,:,i) = colour(:,:,i) + specular(i);
end

end
```

1.3.4 References

WebGL Specular Term

1.4 Projection

This section deals with the virtual camera and how objects are projected from a proposed 'world' to a camera space.

1.4.1 Notes

- View Reference Point (VRP): The centre point/position where the eyes/camera is positioned
- Viewing Plane: The 2D plane in which images are rendered onto in relation to the camera
- Gaze Vector (N): The direction vector from the VRP facing towards the viewing plane
- Up Vector (U): The vector perpendicular to the normal facing upwards in relation to the Gaze Vector
- Handedness Vector (V): The vector perpendicular to the Gaze and Up Vectors facing right in relation to the gaze vector.
- Camera Coordinate System: The coordinate system formed by the N, U and V vectors acting as axes

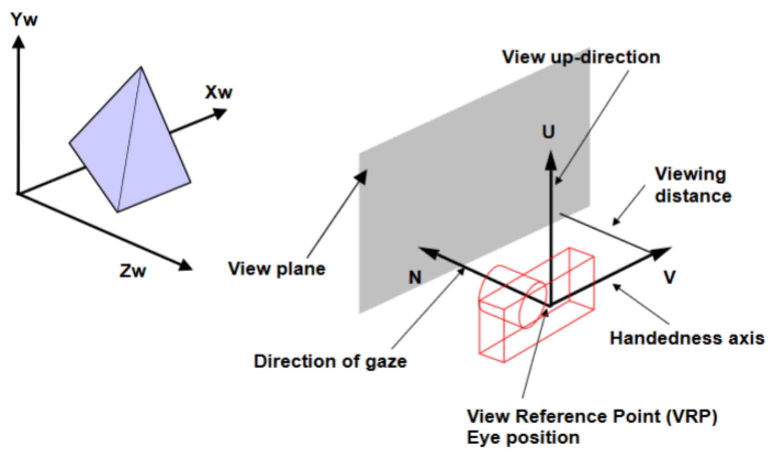

Figure 1.9: Visualised definitions

When rendering to camera, you first need to have a World Coordinate System and objects in that world. From there, you should know VRP and the Viewing Plane. (TP is the position on the Viewing Plane the camera points to.) Using this information, we can calculate the Camera Coordinate System (N V U).

$$\begin{gathered}
\mathrm{U}_{temp} = [0\ 1\ 0] \\
\mathrm{N} = \mathrm{TP} - \mathrm{VRP} \\
\mathrm{V} = \mathrm{U}_{temp} \text{ x } \mathrm{N} \\
\mathrm{U} = \mathrm{N} \text{ x } \mathrm{V}
\end{gathered}$$

We use transformation matrices to move objects form the World Coordinate System to the Camera Coordinate System. Translate to the camera position, rotate to orient with the camera, then scale the x-axis -1 to convert to the left-hand coordinate system to avoid mirroring.

$$T_{cam} = \begin{bmatrix} 1 & 0 & 0 & -x_{vrp} \\ 0 & 1 & 0 & -y_{vrp} \\ 0 & 0 & 1 & -z_{vrp} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.22}$$

$$R_{cam} = \begin{bmatrix} \frac{V_x}{|\mathbf{V}|} & \frac{V_y}{|\mathbf{V}|} & \frac{V_z}{|\mathbf{V}|} & 0 \\ \frac{U_x}{|\mathbf{U}|} & \frac{U_y}{|\mathbf{U}|} & \frac{U_z}{|\mathbf{U}|} & 0 \\ \frac{N_x}{|\mathbf{N}|} & \frac{N_y}{|\mathbf{N}|} & \frac{N_z}{|\mathbf{N}|} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.23}$$

$$S_{cam} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.24}$$

- Centre of Project (COP): The view point the perspective is drawn from, usually the camera position
- Perspective Projection: Objects further away from the COP are scaled smaller
- Orthographic Projection: All objects are on the same scale, regardless of COP

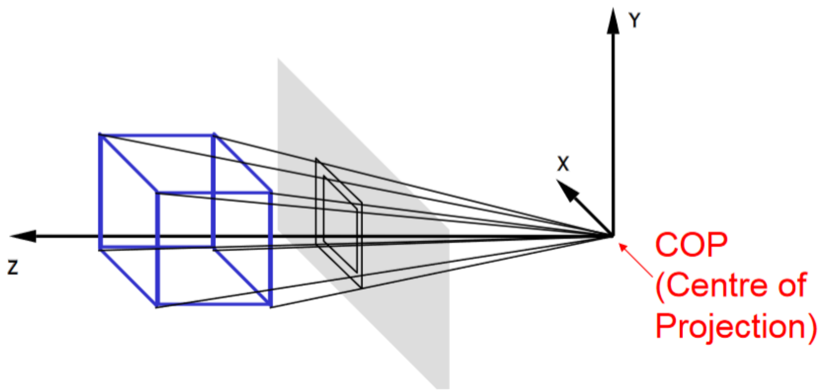

Figure 1.10: Projection Lines meet at the COP

The centre of project can be placed at either the centre/origin of the viewing coordinate system with the viewing plane on the positive z size, or the negative z side with the viewing plane placed on the centre/origin of the viewing coordinate system. Depending on which method is chosen, the computation differs slightly.

COP at Z=0

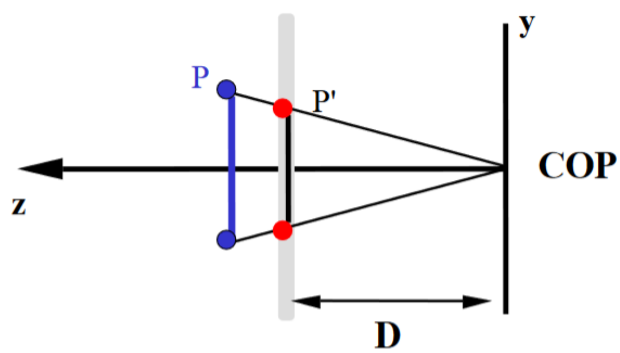

Figure 1.11: COP = 0

$$P_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/\mathbf{D} & 0 \end{bmatrix} \tag{1.25}$$

Where **D** is the distance from the COP and the viewing plane. The generated matrix after multiplication is not homogenous; to convert it to homogenous form we simply divide all terms by the 4th term, which equates to

$$z/D$$

With the perspective project matrix, we can create a general form for moving the object from the World Space to the Camera Space.

$$C = P_{per} * S_{cam} * R_{cam} * T_{cam} \tag{1.26}$$

With the new vector converted to homogenous coordinates after applying C. All new coordinates should also have their Z coordinate equal to te Z coordinate of the viewing plane.
COP at Z¡0 When the viewing plane is at the origin, P_{per} changes slightly.

$$P_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/\mathbf{D} & 1 \end{bmatrix} \tag{1.27}$$

Converting to homogenous would also be slightly different, but will still be fairly simple. All Z coordinates should end up at Z=0.

1.4.2 Examples

The questions here are from the quiz.
Consider the following vertex coordinates:

$$\begin{gathered} \text{V1} = [0,\ 10,\ 0] \\ \text{V2} = [5,\ 0,\ 5] \\ \text{V3} = [15,\ 5,\ 0] \\ \text{V4} = [5,\ 10,\ 15] \end{gathered}$$

Define the viewing (camera) coordinate system by computing its axes, i.e., direction of gaze N, handedness vector V, and vector U which is the correct up-vector. Camera view reference point is VRP = [60, 30, 100], and the target point TP is at the origin ([0, 0, 0]). The camera coordinate system should be defined in the same way as it is shown in the lecture slides: using temporary up vector of [0,1,0] to compute handedness and up vectors of the camera.

1. What is the direction of the gaze vector?
2. What is the handedness vector?
3. What is the up vector?
4. Compute the matrix C_1 to transform an object from the world coordinate system to the camera coordinate system (without project).
5. Compute the matrix C_2 to transform an object from the world coordinate system to the camera coordinate system with the COP at the origin and the viewing plane at z= +40.
6. Compute the positions of the final transformed vertices.

Answers to the previous questions

1. $N = [-0.50, -0.25, -0.83]$
2. $V = [-0.86, 0, 0.51]$
3. $U = [-0.13, 0.97, -0.21]$
4. $C_1 = \begin{bmatrix} 0.857 & 0 & -0.514 & 0 \\ -0.128 & 0.968 & -0.214 & 0 \\ -0.498 & -0.249 & -0.830 & 120 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
5. $C_2 = \begin{bmatrix} 0.857 & 0 & -0.514 & 0 \\ -0.128 & 0.968 & -0.214 & 0 \\ -0.498 & -0.249 & -0.830 & 120 \\ -0.013 & -0.006 & -0.021 & 3.010 \end{bmatrix}$
6. $V_1 = [0.00, 3.29, 40.0]$
   $V_2 = [0.60, -0.60, 40]$
   $V_3 = [4.61, 1.05, 40]$
   $V_4 = [-1.33, 2.27, 40]$

1.4.3 References

No external references were needed on this section.

1.5 Misc. Definitions

This section covers general algorithms and definitions from Rendering to Texture Mapping. Since the exam will focus more on the previous parts for the more technical questions, this part will mostly be structured as a set of definitions.

1.5.1 Rasterisation

- Rasterisation: Converting an object from vector world coordinates to a raster image to display
- Digital Differential Analyzer (DDA): Rasterisation algorithm, interpolates values in an interval by computing separate equations for (x, y). Is expensive and inefficient.
- Bresenham Algorithm: Incremental, integer only algorithm. Has many implementations.
- Antialiasing: utilizes blending techniques to blur the edges of the lines and provide the viewer with the illusion of a smoother line.

Circles can be plotted either directly with a circle equation, directly with polar coordinates, or using Bresenhams by looping across a set of values until all values on a circle are plotted. Bresenham and Polar methods abuse the symmetry of a circle's points.

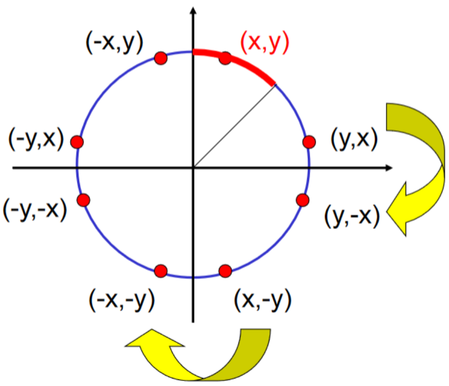

Figure 1.12: Symmetry of circle

1.5.2 Texture Mapping

Texture mapping can be defined as using an image and pasting the image onto a geometric model. There are three types of Texture Mapping.

1. Texture Image: uses iamges to fill inside polygons; inverse mapping using an intermediate surface
2. Environment/Reflection: uses a picture of the scene for texture maps
3. Bump: Emulates altering normal vectors during render process

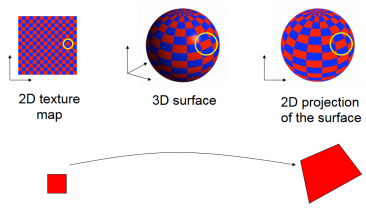

Figure 1.13: Image Mapping flow

When it comes to image mapping, there are two main methods.

1. Forward: copy pixel at source (u, v) to image destination (r, c). Its easy to compute but leaves holes.
2. Backward: for image pixels (r, c), grab texture pixel (u, v). Harder to compute but looks better.

Environment mapping uses the direction of te reflected ray to index a texture map rather than using the ray projected to its surface. This approach isn't completely accurate as it assumes all reflected rays begin from the same point and that all objects in a scene are the same distance from that point.

Bump mapping is a method used to make a surface look rough. There are two variants.

1. Displacement Mapping: Height filed is used to perturb surface point along the direction of its surface normal. Inconvenient to implement since map must perturb geometry of model.
2. Bump Mapping: A perturbation is applied to the surface normal according to the corresponding value in the map. Convenient to implement as it automatically changes the shading parameters of a surface.

There is also mip-mapping. Mapping can cause aliasing to occur; mip-mapping is an anti-aliasing technique that stores texture as a pyramid of progressive resolution images, filtered down from the original. The further away a point is to be rendered, the lower lower resolution MIP-map is used.

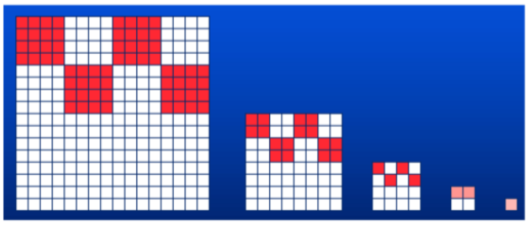

Figure 1.14: MIP-map examples

1.5.3 Hidden Surface Removal

1. Object-space method (OS): Operate on 3D Object entities.(vertices, edges, surfaces)
2. Image-sped (IS): Operate on 2D images (pixels)

There are a few algorithms to go over.

- Polygon Culling: removes all surfaces pointing away from the viewer; renders only what is visible; can be done by comparing if the z-value of the surface normal has the same sign as the z-value of the gaze vector. (If same, remove surface)
- Z-buffer Algorithm: Test visibility of surfaces one point at a time. Easy to implement, fits well with render pipeling, but some inefficiency with large distances. Standard algorithm in many packages, eg. OpenGL
- Painter's Algorithm: OS algorithm; Draw surfaces from back to front. Problems occur with overlapping polygons; as it will always render an object either above or below absolutely.
- Depth Sort: Painter's extension; sorts objects by depth like painters, but resolves overlap issues, splitting polygons if necessary. Does overlap/collision testing and splits on intersection point.

1.5.4 Splines

Splines are smooth curves generated from an input set of user-specified control points.

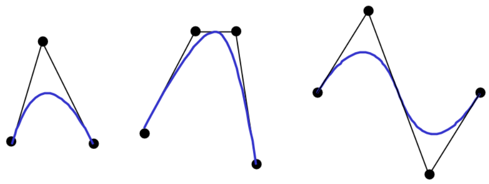

Figure 1.15: Splines with different control points

Bezier curves are generated by forming a set of polynomial functions formed from the coordinates of the control points. In parametric form, a Bezier Function $P(u)$ can be represented as

$$P(u) = \sum_{k=0}^{n} p_k B_{kn}(u)$$

There are many ways to formulate Bezier curves. One such method is the de Casteljau algorithm. This algorithm describes the curve as a recursive series of linear interpolations. We draw lines between each of our control points in order, then continuously Lerp (linearly interpolate) each of the points generated by the previous Lerp until a curve is formed.

Bezier curve shape is influenced by all of its control points. There are B-splines that are only influenced by up to four of the nearest control points. This allows for interesting shapes without the inefficient calculations from insanely high polynomial Bezier curves.

Bezier surfaces are similar to Bezier curves, but instead of just one parameter t, there are two parameters s and t, and instead of a curve, it is a surface mesh.

Chapter 2

Computational Vision

2.1 Light Capturing Devices

This section deals with the basic evolution of light capturing devices.

2.1.1 Notes

Initially, there was a single cell 1D capture of light. This had many problems, but the main one being it can only capture light from one direction, and had no understanding of the intensity of it (binary). Using multiple 1D Cells allowed for more directions to be captured. Having

Figure 2.1: 1D Cell

multiple cells curved allowed for capture of light from various directions, somewhat conic. But it had difficulty keeping track of an image, as the same image would be hitting too many cells, so it would be all over the place. The concept of a pinhole camera was the next step, but there

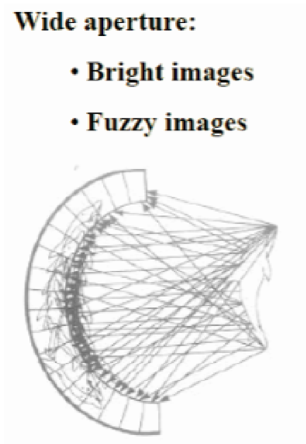

Figure 2.2: 1D Cell Curved Array

was an issue. With the wide aperture, the images were bright but they were fuzzy. With the pinhole, the images were sharp but they were too dim. How can we get the positives without the negatives? Simple, refraction from lenses. By having a lens refract light into the pinhole

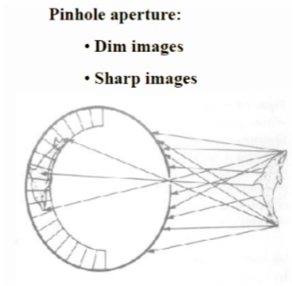

Figure 2.3: Pinhole

camera, or image point, it would allow for all the light to pass through the pinhole and hit different cells based on the initial angle the light hit the lens, maintaining the brightness from the wide aperture with the clarity of the pinhole aperture.
The virtual image created is upside down, and our eyes, based on this concept, simply allow the brain to flip it back.

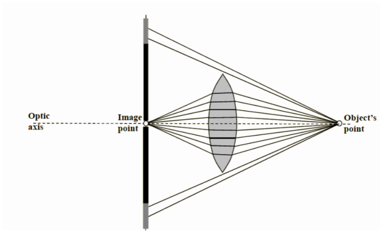

Figure 2.4: Lens in action

2.2 Human Vision

This section deals with human vision, a fairly straight forward topic (at least in our scope, this stuff can get wild pretty fast).

2.2.1 Notes

The human retina contains two types of photo receptor cells. Rods, of where there are 120 million, and Cones, of which there are 6 million. Rods are extremely sensitive to light and respond to single photons. They have poor spatial resolutions as multiple of them converge to the same neuron for data handling. However, thanks to their sensitivity, they help us see light in the dark. This is different with cones, which are active at higher light levels. Several neurons process Cone data, so they have higher spatial resolution than rods.
Receptive Field
The receptive field is the area on which light must fall for neurons to be stimulated.The size of a receptive field determines a few things. Small receptive fields are stimulated by high spatial frequencies; and large spatial fields are stimulated by low spatial frequencies. There are differences between the centre and periphery of field. We can't talk about these without talking about ganglion cells. There are two types of ganglion cells, on-centre and off-centre, as seen in the image. On-centre cells are stimulated when the center is exposed to light, and are inhibited when the surrounded area is exposed. This works opposite for off-centre cells, as the name suggests.
Ganglion cells have a higher action potential rate depending on the intensity and location of light hit. This allows us to have a grasp at contrast, as it responds differently to different

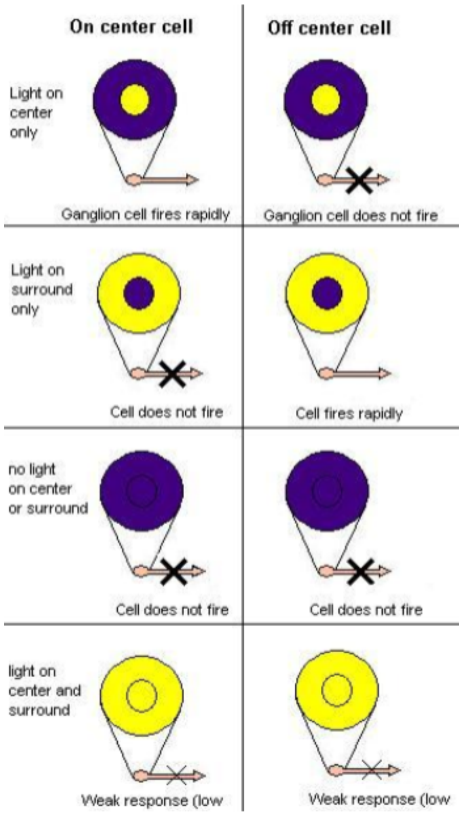

Figure 2.5: Ganglion responses

intensities of light.

Visual Pathway

1. Vision generated by photoreceptors in the eyes (as explained above)
2. The information leaves the eye by way of the optic nerve. Special note: The humans have a blind spot in the eye that does not allow them to see at one specific part of the eye; this is the optic nerve's location. Without it, we wouldn't be able to actually see.
3. There is a partial crossing of axons at the optic chiasm; this allows the brain to receive data on the same visual field from both eyes, superimposing images, creating a sense of depth, etc.
4. The axons following the chiasm, also known as the optic tract, wraps around the midbrain to get to the lateral geniculate nucleus (LGN).
5. The LGN axons fan out to the deep white matter of the brain before ultimately travelling to the primary visual cortex at the back of the brain, where the magic happens.

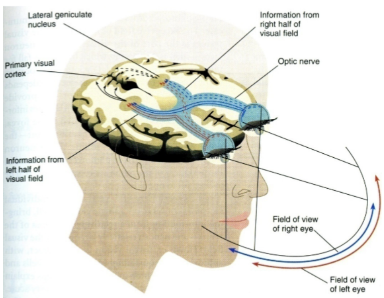

Figure 2.6: Visualising the visual pathway

2.3 Edge Detection

2.3.1 Convolving a Matrix

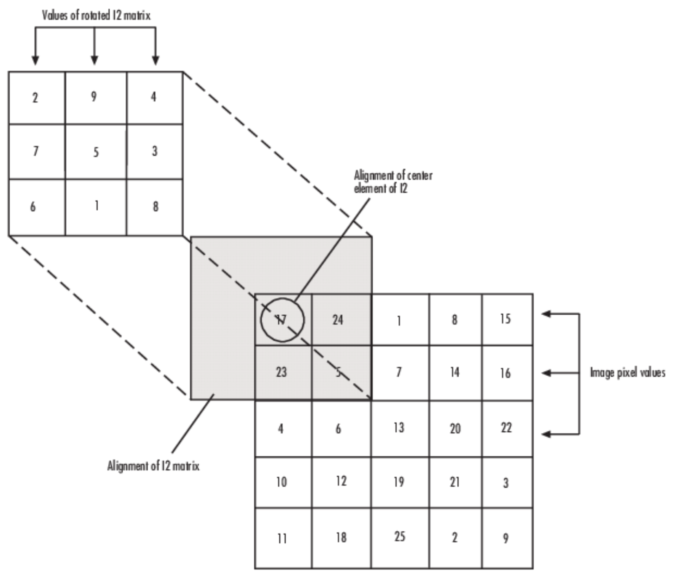

Figure 2.7: 3x3 convolution example

2D convolution of matrices is basically what the course is built off of, at least for edge detection. So its important to know how to do it. Given a 3x3 kernel and an MxN matrix, how does one convolve? Looking at the image, you will create a new matrix by overlaying the kernel with the matrix, multiplying each term on the kernel with the value it overlays and then sum them all up. The new generated value will then be placed in the same position as the centre of the overlay.

Note: for even kernels, you can select either the top-left or top-right centre value as the "centre" of the matrix.

2.3.2 First Order Operators

Edge detection operators are approximation of the first order derivative of the colours. The change in intensity of the colour from a set of points. The gradient of the intensity of colours. I could explain it in mnay different ways, but basically since edges are generally a different shade from its background, checking for a larger change in the intensity would usually return an edge point.

The approximations take the form of different kernels. Small kernels usually don't give a good enough approximation, and using too large a kernel would just blur all the values together and miss edges.

1. Sobel Operator: The sobel operator uses two kernels; one for the x-gradient and one fo the y-gradient. $G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$ and $G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$
2. Roberts Operator: The roberts operator can also use two kernels, but it instead measures the change diagonally. $G_x = \begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix}$ and $G_y = \begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix}$

Once we have applied G_x and G_y, and assuming we have a threshold h (the threshold can be found either with trial and error or setting up some Ai to test with a control set); the final edge pixels can be generated with the approximation:

$$I = (h > |G_x| + |G_y|) \tag{2.1}$$

Edges generated like this might be susceptible to noise, because derivatives are looking for changes in the intensity. Noise are not part of the source, but rather a bi product from the limitation of our cameras and files. (JPG compressed image for example) When a derivative filter is used, it will simply return the change in intensity from the noise as part of the image. Depending on the quantity of noise, it could end up being a complete mess.
We can remove noise using the most common kernel, with the power of Gaussian. It is a gaussian distribution in the form of a 2D kernel, with the highest value in the centre of the matrix. The size of the matrix alters the effect of the noise filtering. A smaller matrix won't filter much, since it's not looking at a large enough space, and using too large a matrix will just blur the image together, removing edges.

0.003	.0133	.0219	.0133	0.003
.0133	.0596	.0983	.0596	.0133
.0219	.0983	.1621	.0983	.0219
.0133	.0596	.0983	.0596	.0133
0.003	.0133	.0219	.0133	0.003

.0545	.2442	.4026	.2442	.0545

.0545
.2442
.4026
.2442
.0545

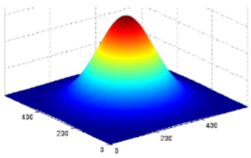

Figure 2.8: Gaussian Noise Filter overview

Rather than convolving the 2D Gaussian with the image, there is a more efficient way to go about this. You can create two 1D matrices $(1xn, nx1)$ and convolve those instead. The result will be the same, and it will be much faster than just doing the 2D matrix as there are less computations to deal with.

2.3.3 Second Order Operators

There is also the option of using Second Order Derivative operators. These function slightly different than first order, as seen in the image. Second Order Derivates look for the zero-crossings. These are the points the values of the graph change signs by crossing the 0-point of the axis. These points are the local maxima/minima in the first order operators. This method is very susceptible to noise, as noise will cause the function to cross zero many times. To alleviate these potential errors, we can apply a gaussian noise filter and a threshold for the distance needed to travel after crossing 0 to be counted as a "zero-crossing". The second order operator, Laplacian Operator, can be derived as follows:

$$
\begin{aligned}
\frac{\partial^2 f}{\partial x^2} =& \frac{\partial G_x}{\partial x} \\
=& \frac{\partial (f[i, j+1] - f[i, j])}{\partial x} \\
=& \frac{\partial f[i, j+1]}{\partial x} - \frac{\partial f[i, j]}{\partial x} \\
=& (f[i, j+2] - f[i, j+1]) - (f[i, j+1] - f[i, j]) \\
=& f[i, j+2] - 2f[i, j+1] + f[i, j]
\end{aligned} \tag{2.2}
$$

The equation above is centred on $[i, j+1]$; if we want to centre it on j, we simply do -1 on all terms, and up with the following stuff.

$$
\begin{aligned}
\frac{\partial^2 f}{\partial x^2} =& f[i, j+1] - 2f[i, j] + f[i, j-1] \\
\frac{\partial^2 f}{\partial y^2} =& f[i+1, j] - 2f[i, j] + f[i-1, j] \\
\Delta^2 \approx & \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}
\end{aligned} \tag{2.3}
$$

We can also place the equation inside of a gaussian distribution function, and end up with a Laplacian of Gaussian (LoG), a potentially efficient second order operator.

2.3.4 Canny Edge Detection

Our new friend and scientist, J. Canny, has shown that the first derivative of the Gaussian closely approximates the operator that optimizes the product of signal - to - noise ratio and localization.
Canny Edge Detection is sort of a standard (its pretty good). There are four major steps to canny edge detection.

1. Find $f_x = f * G_x$ and $f_y = f * G_y$ where $G(x, y)$ is the gaussian function and $G_{x/y}$ is the derivative with respect to the required variable. $G_a = \frac{-a}{\sigma^2}(G(x, y))$
2. Compute the gradient magnitude and direction.

$$mag(x, y) = |f_x| + |f_x y|$$

$$dir(x, y) = \arctan \frac{f_y}{f_x}$$

3. Apply non-maxima suppression. In simpler maths terms, check if the gradient magnitude at a pixel is a local maximum along the gradient direction.
4. Apply hysteresis thresholding. This is fancy for applying a threshold at a low value t_l and a high value t_h, which is usually twice t_l. First mark the edges generated from the high threshold, as these are strong edges and are generally genuine. Trace an edge with the bidirectional information and, while tracing, apply the lower threshold to trace faint sections of edges that have a start point.

2.3.5 References

Canny Edge Detection
A paper on Edge Detection
Good Lecture on Topic

2.4 Facial Recognition

This section deals with the general idea of Facial Recognition.

2.4.1 Eigenfaces

Lets start by thinking of the face as a weighted combination of different components or representations of said face. These components are referred to as Eigenfaces (because who needs real words). Luckily, we can represent all these components as vectors. Eigenfaces are used in two ways; either as a form to store faces for later reconstruction or for recognition of a new image of a familiar face. We can do the learning using Principle Component Analysis (PCA). Given a set of points in 2D, as in the image, you can use vectors in the place of the usual line of best fit. When we use vectors, we would ideally want it to be in normal form with a scaling factor applied to it. This is where Eigenvectors come in. Eigenvectors take the following form:

$$A\vec{v} = \mu\vec{v}$$

where μ is referred to as the "eigenvalue". Eigenvectors are obtained from a matrix, and follow some rules.

- Only square matrices have eigenvectors
- Different matrices have different eigenvectors
- Not all square matrices have eigenvectors
- An $n \times n$ matrix has at most n distinct eigenvectors
- All distinct eigenvectors of a matrix are orthogonal

You ask, where do we get the matrix in this instance? We do it by finding a covariance matrix. Since we have a 2D plot of points, we can generate a 2x2 covariance matrix. This can be done by the following equations, the first for the covariance matrix, and the other for the variance function.

$$C = \begin{bmatrix} Var(x_1) & Covar(x_1, x_2) \\ Covar(x_1, x_2) & Var(x_2) \end{bmatrix}$$

$$Covar(x_1, x_2) = \frac{\sum_{i=1}^{n}(x_1^i - \bar{x_1})(x_2^i - \bar{x_2})}{n-1}$$

With faces, we treat the face as a point in a high-dimensional space, and then treat the training set of faces as our set of points. We calculate the eigenvectors of our training set, and these become our eigenfaces. **TL;DR Eigenfaces are eigenvectors of covariance matrix of training set of faces**

When given an image of a face, we can transform it into face space (Facebook?) by applying the following equation:

$$w_k = x^i\vec{(v_k)}$$

where k is the number of eigenfaces, with the i^{th} face in image space, and a corresponding weight w. Image recognition becomes easy, simply find the euclidean distance of the desired face and all faces stored; the closest face is most likely our match. You can also reconstruct a face from the eigenfaces. The more eigenfaces the better the reconstruction.

2.4.2 References

Just the slides

2.5 Vision Systems

This section deals with feature detection of vision systems.

2.5.1 Invariants

Ideally, we would like some invariance in different settings for what we would like to see.

1. Illumination: Can be achieved by normalising the light levels and storing the new normal intensities. Can also used difference based metrics (sift).
2. Scaling: Store pyramids of the image, with each step half the size of the original. Find pixel values with Gaussian blur. Can also use Scale Space method, using Difference Gaussians to find repeatable points (invariant) in scale.
3. Rotation: Rotate all features to go the same way in a determined manner. Take histogram of gradient directions and rotate to the most dominant.

2.5.2 Feature Detection

In motion tracking, our goal is to see things that move between a set of images. You can begin by knowing which of the four following cases fits best:

1. Stationary Camera, Stationary Objects
2. Stationary Camera, Moving Object
3. Moving Camera, Stationary Objects
4. Moving Camera, Moving Object

When detecting change, you can compare the intensity at one point in time with its previous incarnation (or any other, if you need something specific). However, this lends to issues appearing from random noise, which will pop up everywhere. What we want is to then filter out our noise, but how do we do such a thing?

We can use the idea of 8 or 4 connectedness. Two pixels are 4-neighbours if they share an edge, and are 8-nieghbours if they share a corner. Two pixels are 8-connected if we can create a path of 8-neighbours from one to another. We can use these concepts and group pixels into clusters. We can then threshold the clusters and remove any clusters that are below said threshold, ideally leaving behind just the changes and removing noise. However, just seeing the difference in

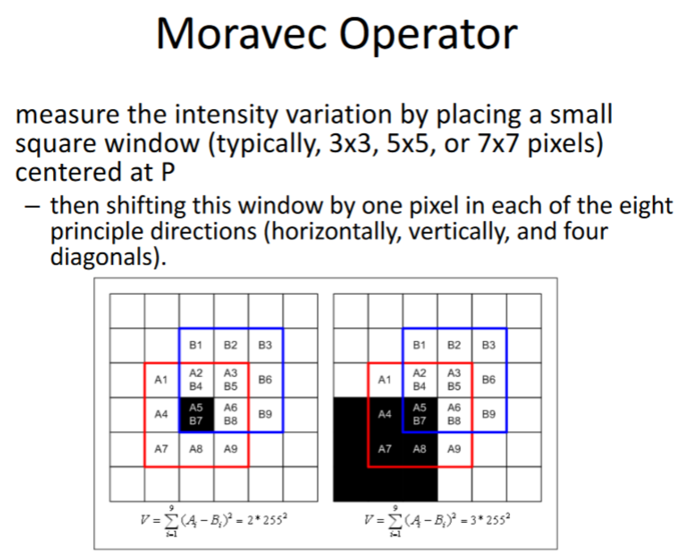

Figure 2.9: Gaussian Moravec Operator Overview

pixels might not be enough. Ideally, we would want to find our points of interest and compare those points between images. For this, we can use the Moravec operators, a corner detector. Corners are better than edges for this task because corners can measure the highest intensity change, as oppose to edges. The image explains how the Moravec operators works. Once we obtain our new values from our friend Moravec, we can just keep the local maxima.

Motion Correspondence has three main principles we need to worry about.

1. Discreteness: Measure of the distinctiveness of individual points
2. Similarity: Measure of how closely two points resemble one another
3. Consistency: Measure of how well a match conforms with nearby matches

2.5.3 Object Detection

Object recognition needs the following components:

1. Model Database: Stores all models known by the system; information stored depending on approach for recognition. Generally, these values are abstract (shape, colour, size, etc.)
2. Feature Detector: Applies operators to image and identifies location of features. How a feature detector works depends on what types of models are stored in the database.
3. Hypothesizer: Assigns likelihoods to objects present to reduce computational expense.
4. Hypothesis Verifier: Uses object models to verify hypothesis and refines the hypothesizer likelihood.

Each of these components come with issues, mostly related to what sort of thing you want to do. (2D vs 3D, model vs object, etc.)

Chapter 3

Models of Computation

3.1 Regular Expressions

3.1.1 Regular Language

Strings can be broken down as "regular expressions", a logical expression that can be used to solve matching and searching problems. A matching problem can be checking if a string is a valid password that contains at least X characters and at least Y digits. A searching problem can be, given a string, list all occurrences of an email address in it. Regexps are used to solve problems like these.
For example, the regular expression **c(bb|ca)*** will return true when matched with **cbbbbca** but return false when matched with **bca**. To solve problems using regexps, we can use Deterministic Finite Automatons (DFAs). They are deterministic because the initial state and the result of each transition are specified, and finite because there set of states is a finite set. When given a string, in the DFA seen, it will process each letter one at a time, going from state and following the transitions.

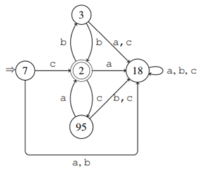

Figure 3.1: Example of a DFA

As you can see, there are a few loops/areas where the DFA will definitely return false once it reaches a certain point. A more efficient way to deal with these things is to completely remove the transitions, and have it return a failure at points we know it will definitely fail. These are called partial DFAs, denoted by the δ symbol (δDFA). These are usually faster to write and run faster.
What happens when the DFA is not deterministic? An NFA if you will. NFAs differ from DFAs in two ways.

- An NFA can have several initial states
- From a given state, when a is input, there can be several possible next states.

NFAs are a relation and not a function. They can still return acceptable words, but a program can make no use of it since its, well, non-deterministic. However, there are ways to determinize an NFA. The best, basic way to accomplish this is to combine states into a set of state transitions if they match similar inputs/outputs. Conversion can take a few steps, but it's fairly simple and systematic in this regard.

Some trouble can pop up from NFAs with an ϵ transition, or an empty transitions. These bad boys can change from one state to another for literally no reason. These NFAs are called ϵNFAs, which seems obvious. With this out of the way, let's make note of Kleene's Theorem.
For a language L, the following are equivalent:

- L is regular
- The matching problem for L can be solved by a DFA.

A systematic way to turn a regexp to an equal DFA:

1. Convert the regexp into an ϵNFA
2. Convert it down to an NFA
3. Convert it further down to a DFA.
4. Minimize DFA size by identifying equivalent states, removing unreachable states and just general refactoring.

3.1.2 Non-regular Language

3.2 Turing Machines

3.3 Decidability and Completeness

3.4 Lambda Calculus

Chapter 4

Introductory Databases

Chapter 5

Computer Systems & Architecture

Chapter 6

C/C++

Chapter 7

Mathematical Techniques for Computer Science

7.1 Vectors and Matrices

This section deals with operations on vectors and matrices that are commonly used.

7.1.1 Vector Operations

Magnitude

$$\left\|\vec{V}\right\| = \sqrt{\sum_{i=1}^{n} v_i^2} \tag{7.1}$$

Dot Product

$$\vec{V} \cdot \vec{U} = \sum_{i=1}^{n} v_i u_i \tag{7.2}$$

3x3 Cross Product

$$\vec{V} \times \vec{U} = \begin{bmatrix} v_2u_3 - v_3u_2 \\ v_3u_1 - v_1u_3 \\ v_1u_2 - v_2u_1 \end{bmatrix} \tag{7.3}$$

Equations with angles

$$\cos\theta = \frac{\vec{V} \cdot \vec{U}}{\left\|\vec{V}\right\| \left\|\vec{U}\right\|} \tag{7.4}$$

$$\sin\theta = \frac{\vec{V} \times \vec{U}}{\left\|\vec{V}\right\| \left\|\vec{U}\right\|} \tag{7.5}$$

7.1.2 Matrix Operations

Multiply

$$axb = comingsoon \tag{7.6}$$

2x2 Determinant 3x3 Determinant Row Operations Inverting a Matrix

7.2 Fields

This section deals with the little we need to know about Galois Fields, primarily GF(2).

7.2.1 Notes

7.2.2 Proofs

7.3 Lines and Planes

This section covers everything done regarding systems of equations regarding lines and planes.

7.3.1 Notes

7.3.2 Examples

7.4 Set Theory

This section covers the basics of Set Theory, with some examples.

7.4.1 Notes

7.4.2 Examples

7.5 Functions

Expanding from Set Theory, this section deals with Functions and our new understanding of them.

7.5.1 Notes

7.5.2 Examples

7.6 Probability

Continuing from Year 1 AI, probability.

7.6.1 Notes

7.6.2 Examples

7.7 Random Variables

This section deals with the handling of both discrete and continuous random variables. Don't go gambling after you cover this section.

7.7.1 Discrete

7.7.2 Continuous

7.7.3 Examples

7.8 Equations

Appending-like area to store all equations that will likely be used in the exam.

Chapter 8

Introduction to Computer Security

Chapter 9

Professional Computing

Chapter 10

Functional Programming