

UNIVERSITY OF BIRMINGHAM

COMPUTER SCIENCE YEAR 2

FAISAL IMH ALRAJHI

---

## Year 2 Study Guide

---



UNIVERSITY OF  
BIRMINGHAM

# Contents

<b>1</b>	<b>Graphics</b>	<b>1</b>
1.1	Surface Geometry . . . . .	1
1.1.1	Notes . . . . .	1
1.1.2	Examples . . . . .	3
1.1.3	Normal Vectors . . . . .	5
1.1.4	Further Sources . . . . .	5
1.2	Transforms . . . . .	6
1.2.1	Notes . . . . .	6
1.2.2	Transformation Matrices . . . . .	6
1.2.3	Examples . . . . .	7
1.2.4	References . . . . .	8
1.3	Lighting . . . . .	9
1.3.1	Notes . . . . .	9
1.3.2	Phong Shading Equation . . . . .	9
1.3.3	Examples . . . . .	11
1.3.4	References . . . . .	12
1.4	Projection . . . . .	13
1.4.1	Notes . . . . .	13
1.4.2	Examples . . . . .	15
1.4.3	References . . . . .	16
1.5	Misc. Definitions . . . . .	17
1.5.1	Rasterisation . . . . .	17
1.5.2	Texture Mapping . . . . .	17
1.5.3	Hidden Surface Removal . . . . .	18
1.5.4	Splines . . . . .	19
<b>2</b>	<b>Computational Vision</b>	<b>20</b>
<b>3</b>	<b>Models of Computation</b>	<b>21</b>
<b>4</b>	<b>Introductory Databases</b>	<b>22</b>
<b>5</b>	<b>Computer Systems &amp; Architecture</b>	<b>23</b>
<b>6</b>	<b>C/C++</b>	<b>24</b>
<b>7</b>	<b>Mathematical Techniques for Computer Science</b>	<b>25</b>
<b>8</b>	<b>Introduction to Computer Security</b>	<b>26</b>
<b>9</b>	<b>Professional Computing</b>	<b>27</b>



# 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  $DT(P)$ .

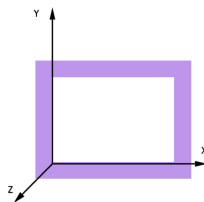


Figure 1.1: Coordinate system assumed throughout module

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

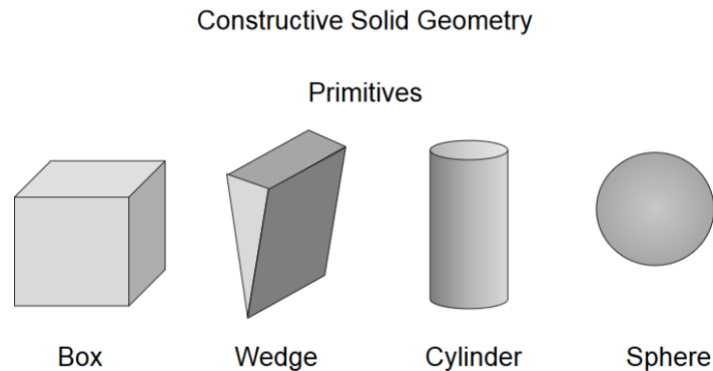


Figure 1.2: Constructive Solid Geometry (CSG) Primitives

We covered a few volumetric models in the module.

1. CSG: Uses primitive 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:

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

2D Circle - Parametric:

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

2D Ellipse - Parametric:

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

3D Sphere:

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

3D Ellipsoid:

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

3D Sphere - Parametric:

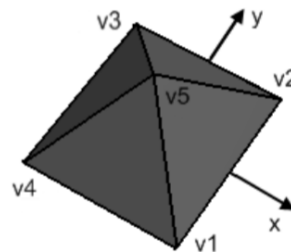
$$\begin{aligned} x &= r \cos \phi \cos \theta \\ y &= r \cos \phi \sin \theta \\ z &= r \sin \phi \\ -\pi &\leq \theta \leq \pi \\ -\pi/2 &\leq \phi \leq \pi/2 \end{aligned} \quad (1.6)$$

3D Ellipsoid - Parametric:

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

### 1.1.2 Examples

Define a vertex table and a surface table for the pyramid, depicted on the right. The base of the pyramid is a square with the side  $a=2$  centered in the origin. The height of the pyramid is equal to 2. Work in the right handed coordinate system.



<b>v1</b> [1; -1; 0]	<b>f1</b> : <b>v1</b> - <b>v2</b> - <b>v5</b>
<b>v2</b> [1; 1; 0]	<b>f2</b> : <b>v2</b> - <b>v3</b> - <b>v5</b>
<b>v3</b> [-1; 1; 0]	<b>f3</b> : <b>v3</b> - <b>v4</b> - <b>v5</b>
<b>v4</b> [-1; -1; 0]	<b>f4</b> : <b>v4</b> - <b>v1</b> - <b>v5</b>
<b>v5</b> [0; 0; 2]	<b>f5</b> : <b>v1</b> - <b>v4</b> - <b>v3</b> - <b>v2</b>

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;

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

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

$$N = \begin{bmatrix} A_y B_z - A_z B_y \\ A_z B_x - A_x B_z \\ A_x B_y - A_y B_x \end{bmatrix} \quad (1.9)$$

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.

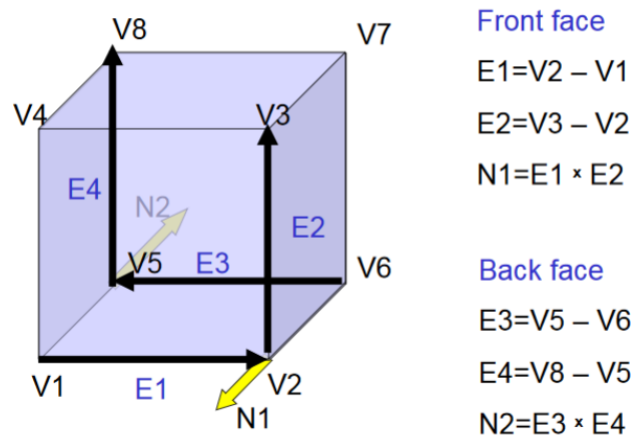


Figure 1.5: Normal Vector of cube from lecture

### 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

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

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

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

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

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

$$R_{3D_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.15)$$

$$R_{3D_y} = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.16)$$

$$R_{3D_z} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.17)$$

### 1.2.3 Examples

#### MATLAB Assignment Rotating

```

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

#### Scaling

```

1      %% 3. Apply scaling transformation to bowl object mesh model
2      %%% Define a matrix for scaling of bowl object with scaling factor f = ...
        [3.5, 1.5, 2] in direction of x, y and
3      %%% z axes. Apply this matrix to your original bowl object ...
        (non-transformed) and visualize the result
4      %%% using trisurf function. Save the result of your visualization to ...
        "3.png"? file and include this file in your
5      %%% submission.
6
7      scaling = [3.5 0 0 0; 0 1.5 0 0; 0 0 2 0; 0 0 0 1];
8      scaled_object_vertices = scaling*obj_v4;
```

## Translating

```
1 %% 4. Apply translation transformation to bowl object mesh model
2 %%% Define a matrix for translation of bowl object by [-500, 50, -100] in ...
   direction of x, y and z axes. Apply
3 %%% this matrix to your original bowl object and visualize the result ...
   using trisurf function. Save the result
4 %%% of your visualization to "4.png"? file and include this file in your ...
   submission.
5
6 translate = [1 0 0 -500; 0 1 0 50; 0 0 1 -100; 0 0 0 1];
7 translated_object_vertices = translate*obj_v4;
```

## 3-in-1 Wombo Combo

```
1 %% 5. Apply all 3 transformations defined above to your original ...
   (non-transformed) bowl object one after the other in the given order.
2 %%% Display transformed meshes in the figure using trisurf.
3 %%% Save the result of your visualisation to "5.png" file and include ...
   this file to you submission folder.
4
5 %% Compute transformations (4x4 transformation matrices)
6
7 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.

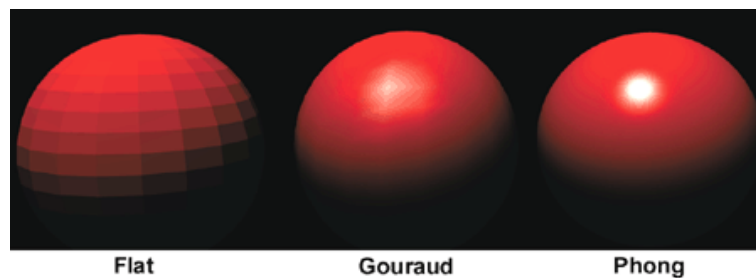


Figure 1.6: Shading Model Differences

### 1.3.2 Phong Shading Equation

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

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

$$Ambient = I_a K_a \quad (1.19)$$

**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  $\theta$  between the light source and the normal of the surface is then computed, and the  $\cos(\theta)$  is multiplied to obtain the full term.

$$Diffuse = I_d K_d \cos \theta_L \quad (1.20)$$

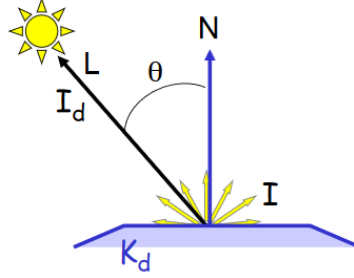


Figure 1.7: Diffuse term overview

**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  $\theta_L$  away from the normal of the surface. This intensity is multiplied by the cos of the angle  $\theta_S$ , 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.

$$Diffuse = I_d K_d \cos^n \theta_S \quad (1.21)$$

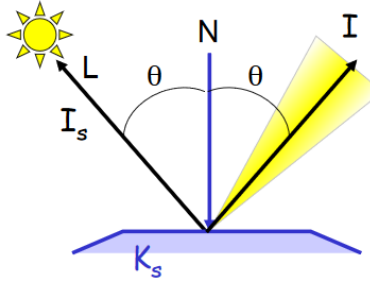


Figure 1.8: Specular term overview

### 1.3.3 Examples

#### MATLAB Assignment

##### Calculating Ambient Term

```
1 function colour = calcAmbient_skeleton(pixel.colour.current, Ia,Ka)
2
3 % Colour at current pixel
4 colour = pixel.colour.current;
5
6 % TO DO: Compute colour of the pixel here and write values to the
7 % corresponding place in image ImA
8
9 ambient = Ia.* Ka;
10 for i = 1:size(colour, 3)
11 colour(:, :, i) = ambient(i,i,:);
12 end
13 end
```

##### Calculating Diffuse Term

```
1 function colour = calcDiffuse_skeleton(pixel.colour.current, ...
    point.position, light.position, surface.normal, Id, Kd)
2
3 % Colour at current pixel
4 colour = pixel.colour.current;
5
6 %TO DO: Calculate light direction from light position and point position
7 light_direction = light.position - point.position;
8 %TO DO: Calculate normalised light direction
9 normalised_light = light_direction / norm(light_direction);
10 %TO DO: Calculate cos light direction, removing negative
11 %values
12 cos_light = dot(surface.normal, normalised_light);
13
14 %TO DO: Compute colour colour of the pixel here and write
15 % values to the corresponding place in image ImD
16 diffuse = Id.*(Kd.*cos_light);
17 for i = 1:size(colour, 3)
18 colour(:, :, i) = colour(:, :, i) + diffuse(i);
19 end
20 end
```

## Calculating Specular Term

```
1 function colour = calcSpecular_skeleton(pixel_colour_current, ...
    point_position, light_position, surface_normal, normal_towardsViewer, ...
    shininess_factor, Is, Ks)
2
3 % Colour at current pixel
4 colour = pixel_colour_current;
5
6 %TO DO: Calculate light direction from light position and point position
7 light_direction = point_position - light_position;
8
9 %TO DO: Normalised light direction
10 normalised_light = light_direction / norm(light_direction);
11 %TO DO: Normal component
12 n = surface_normal / norm(surface_normal);
13 %TO DO: Reflected Ray (tangent + ray in one step)
14 R = n*2*dot(n, -1*normalised_light) + normalised_light;
15 %TO DO: Normalised Reflected Ray
16 normalised_reflection = R / norm(R);
17 %TO DO: Calculate cos_spec, removing negative values
18 cos_light = dot(normal_towardsViewer, normalised_reflection);
19 if (cos_light < 0)
20     cos_light = 0;
21 end
22 %TO DO: compute colour colour of the pixel here and write
23 % values to the corresponding place in image ImS
24 specular = Is.*(Ks.*(cos_light^shininess_factor));
25 for i = 1:size(colour, 3)
26     colour(:,:,i) = colour(:,:,i) + specular(i);
27 end
28
29 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

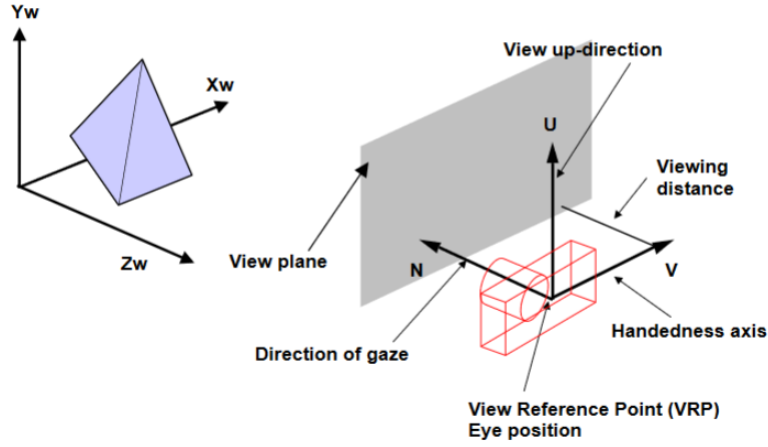


Figure 1.9: Visualised definitions

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).

$$\begin{aligned}
 U_{temp} &= [0 \ 1 \ 0] \\
 N &= TP - VRP \\
 V &= U_{temp} \times N \\
 U &= N \times V
 \end{aligned}$$

We use transformation matrices to move objects from 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.

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



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

$$S_{cam} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.24)$$

- 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

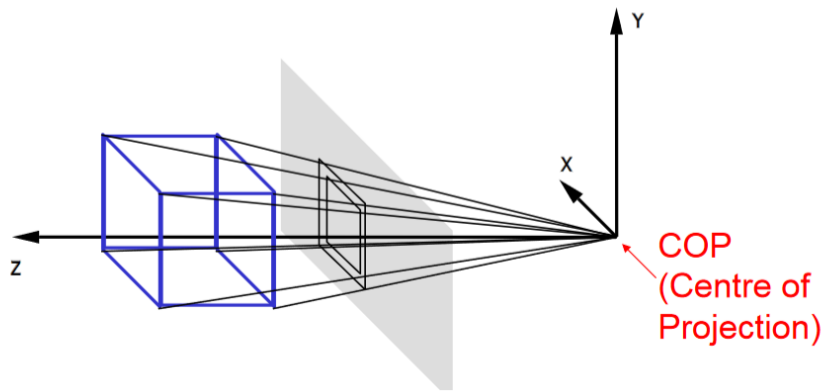


Figure 1.10: Projection Lines meet at the COP

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

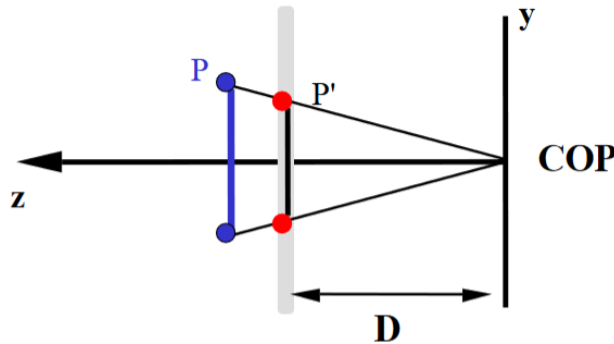


Figure 1.11: COP = 0

$$P_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/D & 0 \end{bmatrix} \quad (1.25)$$

Where  $\mathbf{D}$  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.

$$C = P_{per} * S_{cam} * R_{cam} * T_{cam} \quad (1.26)$$

With the new vector converted to homogenous coordinates after applying  $C$ . All new coordinates should also have their Z coordinate equal to the Z coordinate of the viewing plane.

**COP at  $\mathbf{Z_i0}$**  When the viewing plane is at the origin,  $P_{per}$  changes slightly.

$$P_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/D & 1 \end{bmatrix} \quad (1.27)$$

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:

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

Define the viewing (camera) coordinate system by computing its axes, i.e., direction of gaze  $\mathbf{N}$ , handedness vector  $\mathbf{V}$ , and vector  $\mathbf{U}$  which is the correct up-vector. Camera view reference point is  $\mathbf{VRP} = [60, 30, 100]$ , and the target point  $\mathbf{TP}$  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 Bresenham's 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.

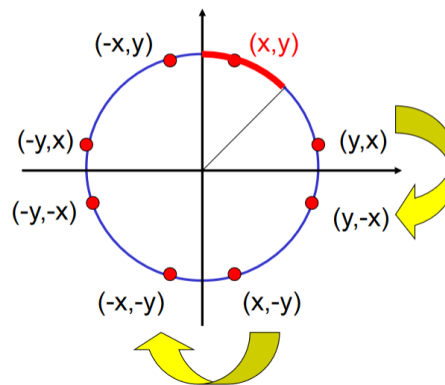


Figure 1.12: Symmetry of circle

### 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

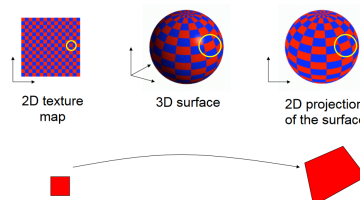


Figure 1.13: Image Mapping flow

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 the 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 field 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 resolution MIP-map is used.

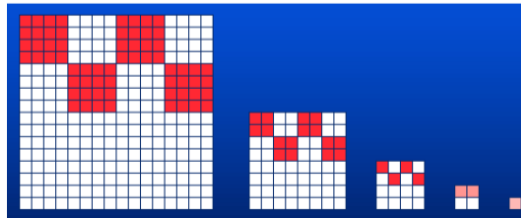


Figure 1.14: MIP-map examples

### 1.5.3 Hidden Surface Removal

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

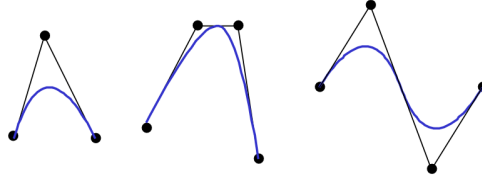


Figure 1.15: Splines with different control points

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

# Chapter 3

## Models of Computation



## Chapter 4

# Introductory Databases

# Chapter 5

## Computer Systems & Architecture

# Chapter 6

## C/C++

## Chapter 7

# Mathematical Techniques for Computer Science

## Chapter 8

# Introduction to Computer Security

## Chapter 9

# Professional Computing

## Chapter 10

# Functional Programming