University of Birmingham

Computer Science Year 2

Faisal IMH Alrajhi

Year 2 Study Guide

UNIVERSITY OF BIRMINGHAM

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).

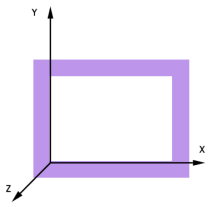


Figure 1.1: Coordinate system assumed throughout module

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

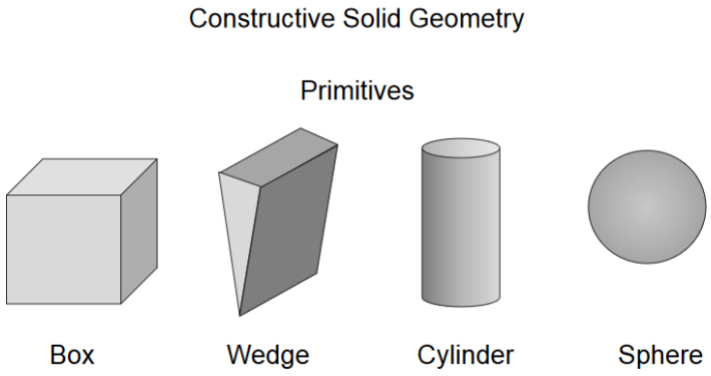


Figure 1.2: Constructive Solid Geometry (CSG) Primitives

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:

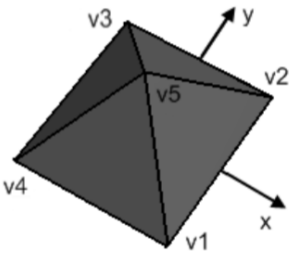
$$\begin{aligned} x &= r\cos\phi\cos\theta \\ y &= r\cos\phi\sin\theta \\ z &= r\sin\phi \\ -\pi &\leq \theta \leq \pi \\ -\pi/2 &\leq \phi \leq \pi/2 \end{aligned} \tag{1.6}$$

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

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.



v1 [1; -1; 0]
v2 [1; 1; 0]
v3 [-1; 1; 0]
v4 [-1; -1; 0]
v5 [0; 0; 2]

f1 : **v1** - **v2** - **v5**
f2 : **v2** - **v3** - **v5**
f3 : **v3** - **v4** - **v5**
f4 : **v4** - **v1** - **v5**
f5 : **v1** - **v4** - **v3** - **v2**

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;

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

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.

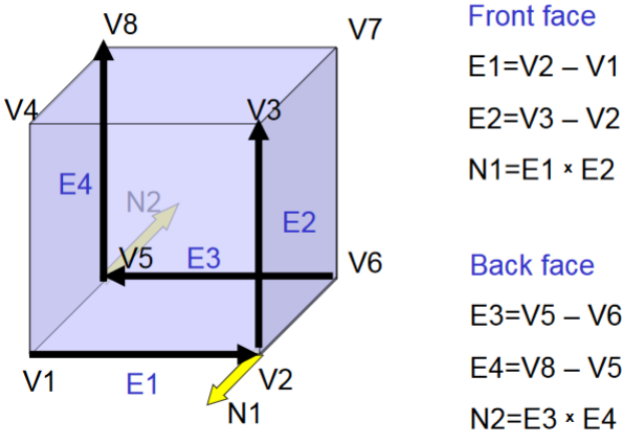


Figure 1.5: Normal Vector of cube from lecture

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.

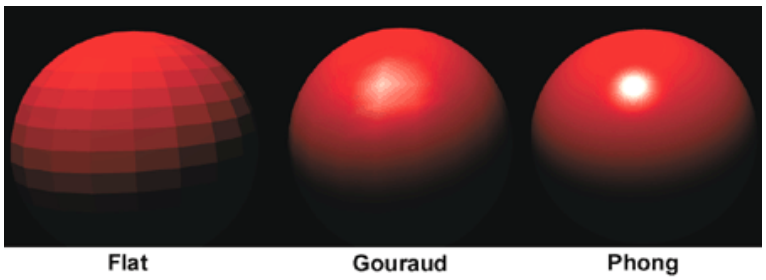


Figure 1.6: Shading Model Differences

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.

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

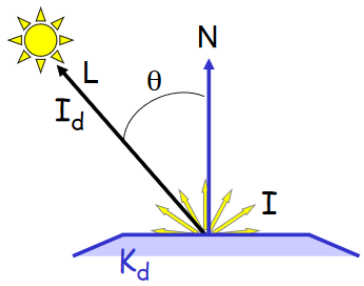


Figure 1.7: Diffuse term overview

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.

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

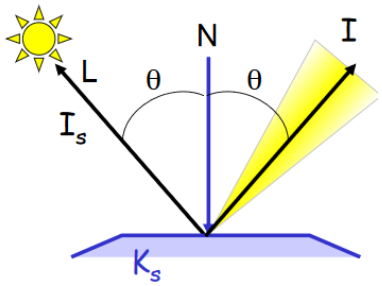


Figure 1.8: Specular term overview

1.3.3 Examples

MATLAB Assignment

Calculating Ambient Term

```
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

```
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

1.5 Texture Mapping

1.6 Past Exam Practice

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