

---

# CT404

---

## Graphics & Image Processing

---



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Grading . . . . .	1
1.1.1	Reflection on Exams . . . . .	1
1.2	Lecturer Contact Information . . . . .	1
<b>2</b>	<b>Introduction to 2D Graphics</b>	<b>1</b>
2.1	Digital Images – Bitmaps . . . . .	1
2.2	Colour Encoding Schemes . . . . .	1
2.3	The Real-Time Graphics Pipeline . . . . .	2
2.4	Graphics Software . . . . .	2
2.5	Graphics Formats . . . . .	2
<b>3</b>	<b>2D Vector Graphics</b>	<b>2</b>
3.1	Transformations . . . . .	3
3.1.1	2D Translation . . . . .	3
3.1.2	2D Rotation of a <i>Point</i> . . . . .	3
3.1.3	2D Rotation of an <i>Object</i> . . . . .	4
3.1.4	Arbitrary 2D Rotation . . . . .	4
3.1.5	Matrix Notation . . . . .	4
3.1.6	Scaling . . . . .	5
3.1.7	Order of Transformations . . . . .	5
<b>4</b>	<b>2D Raster Graphics</b>	<b>5</b>
4.1	Introduction to HTML5/Canvas . . . . .	5
4.1.1	Canvas: Rendering Contexts . . . . .	5
4.1.2	Canvas2D: Primitives . . . . .	6
4.1.3	Canvas2D: drawImage() . . . . .	7
4.1.4	Canvas2D: Fill & Stroke Colours . . . . .	7
4.1.5	Canvas2D: Translations . . . . .	8
4.1.6	Canvas2D: Order of Transformations . . . . .	9
4.1.7	Scaling . . . . .	10
4.1.8	Canvas2D: Programmatic Graphics . . . . .	10
<b>5</b>	<b>3D Co-Ordinate Systems</b>	<b>11</b>
5.1	Nested Co-Ordinate Systems . . . . .	11
5.2	3D Transformations . . . . .	11
5.2.1	Translation . . . . .	11
5.2.2	Rotation About Principal Axes . . . . .	11
5.2.3	Rotation About Arbitrary Axes . . . . .	12
<b>6</b>	<b>Graphics APIs</b>	<b>12</b>
6.1	Three.js . . . . .	12
6.1.1	3D Primitives . . . . .	13
6.1.2	Cameras . . . . .	15
6.1.3	Lighting . . . . .	15
6.1.4	Nested Co-Ordinates . . . . .	17
6.1.5	Geometry Beyond Primitives . . . . .	19

<b>7</b>	<b>Animation &amp; Interactivity</b>	<b>19</b>
7.1	Handling the Keyboard . . . . .	19
7.2	Mouse Handling . . . . .	20
7.3	Time-Based Animation . . . . .	21
7.4	Raycasting . . . . .	23
7.5	Shading Algorithms . . . . .	26
7.5.1	Flat Shading . . . . .	27
7.5.2	Smooth (Gourard) Shading . . . . .	27
7.5.3	Normal Interpolating (Phong) Shading . . . . .	28
7.6	Shading in Three.js . . . . .	28
7.7	Shadows in Three.js . . . . .	28
7.8	Reflectivity of Materials in Three.js . . . . .	28
<b>8</b>	<b>Image Processing</b>	<b>29</b>
8.1	Introduction to OpenCV . . . . .	29
8.2	Point Transformations . . . . .	29
8.2.1	Histogram Manipulation . . . . .	29
8.2.2	Thresholding . . . . .	31
8.3	Geometric Transformations . . . . .	32
8.3.1	Interpolation . . . . .	32
8.3.2	Warping . . . . .	32
8.3.3	Image Rectification . . . . .	33
8.3.4	Image Registration . . . . .	33
8.3.5	Map Projection . . . . .	33
8.3.6	Image Morphing . . . . .	34
<b>9</b>	<b>Spatial Filtering</b>	<b>35</b>
9.1	Image Filtering for Noise Reduction . . . . .	35
9.2	Image Filtering for Edge Detection . . . . .	36
9.2.1	First Order Derivatives . . . . .	36
9.2.2	Second Order Derivatives . . . . .	39
9.3	Image Filtering in the Frequency Domain . . . . .	40
<b>10</b>	<b>Morphological Image Processing</b>	<b>40</b>
10.1	Basic Morphological Operations . . . . .	41

# 1 Introduction

Textbooks:

- Main textbook: *Image Processing and Analysis* – Stan Birchfield (ISBN: 978-1285179520).
- *Introduction to Computer Graphics* – David J. Eck. (Available online at <https://math.hws.edu/graphicsbook/>).
- *Computer Graphics: Principles and Practice* – John F. Hughes et al. (ISBN: 0-321-39952-8).
- *Computer Vision: Algorithms and Applications* – Richard Szeliski (ISBN: 978-3-030-34371-2).

**Computer graphics** is the processing & displaying of images of objects that exist conceptually rather than physically with emphasis on the generation of an image from a model of the objects, illumination, etc. and the real-time rendering of images. Ideas from 2D graphics extend to 3D graphics.

**Digital Image processing/analysis** is the processing & display of images of real objects, with an emphasis on the modification and/or analysis of the image in order to automatically or semi-automatically extract useful information. Image processing leads to more advanced feature extraction & pattern recognition techniques for image analysis & understanding.

## 1.1 Grading

- Assignments: 30%.
- Final Exam: 70%.

### 1.1.1 Reflection on Exams

“A lot of people give far too little detail in these questions, and/or don’t address the discussion parts – they just give some high-level definitions and consider it done – which isn’t enough for final year undergrad, and isn’t answering the question. More is expected in answers than just repeating what’s in my slides. The top performers demonstrate a higher level of understanding and synthesis as well as more detail about techniques and discussion of what they do on a technical level and how they fit together”

## 1.2 Lecturer Contact Information

- |  |  |
|--|--|
| • Dr. Nazre Batool.  | • Dr. Waqar Shahid Qureshi.  |
| • <a href="mailto:nazre.batool@universityofgalway.ie">nazre.batool@universityofgalway.ie</a> | • <a href="mailto:waqarshahid.qureshi@universityofgalway.ie">waqarshahid.qureshi@universityofgalway.ie</a> . |
| • Office Hours: Thursdays 16:00 – 17:00, CSB-2009.   | • Office Hours: Thursdays 16:00 – 17:00, CSB-3001.   |

# 2 Introduction to 2D Graphics

## 2.1 Digital Images – Bitmaps

**Bitmaps** are grid-based arrays of colour or brightness (greyscale) information. **Pixels** (*picture elements*) are the cells of a bitmap. The **depth** of a bitmap is the number of bits-per-pixel (bpp).

## 2.2 Colour Encoding Schemes

Colour is most commonly represented using the **RGB (Red, Green, Blue)** scheme, typically using 24-bit colour with one 8-bit number representing the level of each colour channel in that pixel.

Alternatively, images can also be represented in **greyscale** wherein pixels are represented with one (typically 8-bit) brightness value (or scale of grey) .

### 2.3 The Real-Time Graphics Pipeline

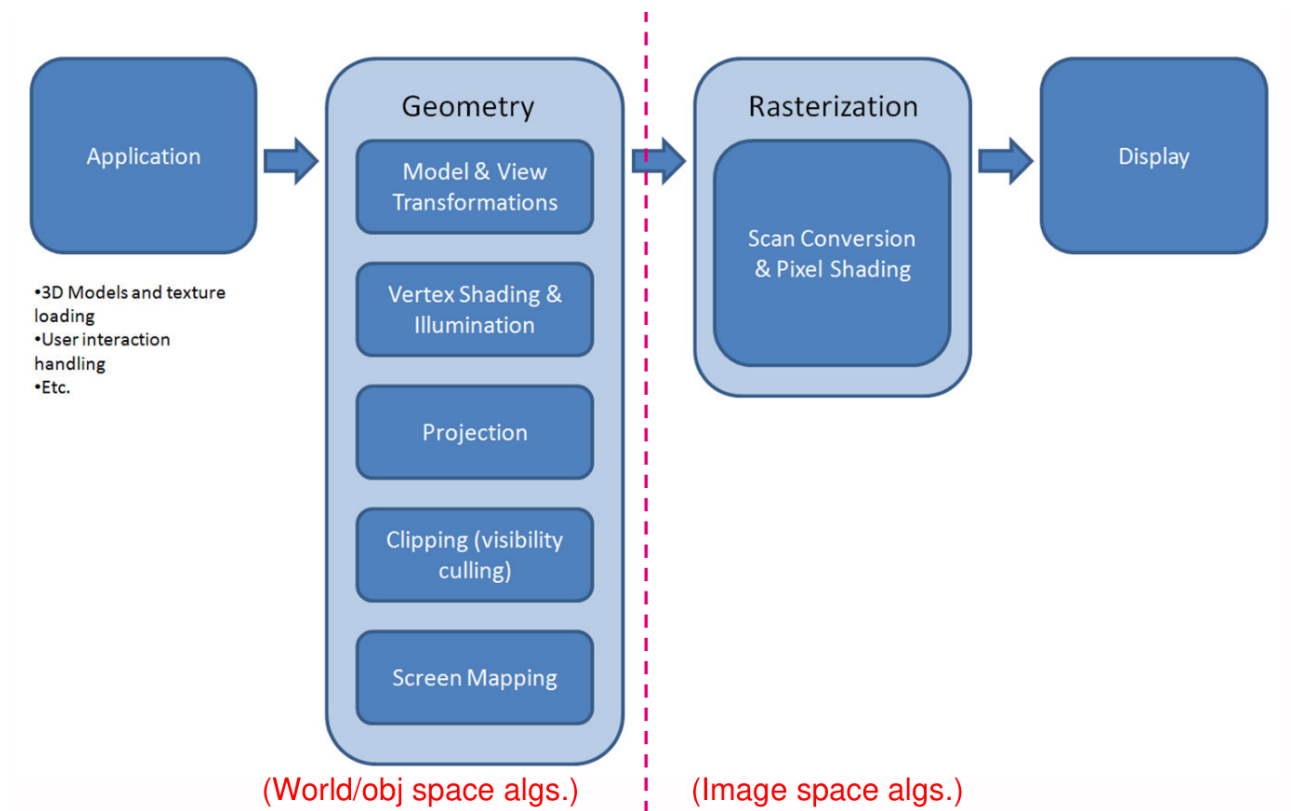


Figure 1: The Real-Time Graphics Pipeline

### 2.4 Graphics Software

The **Graphics Processing Unit (GPU)** of a computer is a hardware unit designed for digital image processing & to accelerate computer graphics that is included in modern computers to complement the CPU. They have internal, rapid-access GPU memory and parallel processors for vertices & fragments to speed up graphics renderings.

**OpenGL** is a 2D & 3D graphics API that has existed since 1992 that is supported by the graphics hardware in most computing devices today. **WebGL** is a web-based implementation of OpenGL for use within web browsers. OpenGL ES for Embedded Systems such as tablets & mobile phones also exists.

OpenGL was originally a client/server system with the CPU+Application acting as a client sending commands & data to the GPU acting as a server. This was later replaced by a programmable graphics interface (OpenGL 3.0) to write GPU programs (shaders) to be run by the GPU directly. It is being replaced by newer APIs such as Vulkan, Metal, & Direct3D and WebGL is being replaced by WebGPU.

### 2.5 Graphics Formats

**Vector graphics** are images described in terms of co-ordinate drawing operations, e.g. AutoCAD, PowerPoint, Flash, SVG. **SVG (Scalable Vector Graphics)** is an image specified by vectors which are scalable without losing any quality.

**Raster graphics** are images described as pixel-based bitmaps. File formats such as GIF, PNG, JPEG represent the image by storing colour values for each pixel.

## 3 2D Vector Graphics

**2D vector graphics** describe drawings as a series of instructions related to a 2-dimensional co-ordinate system. Any point in this co-ordinate system can be specified using two numbers  $(x, y)$ :

- The horizontal component  $x$ , measuring the distance from the left-hand edge of the screen or window.
- The vertical component  $y$ , measuring the distance from the bottom of the screen or window (or sometimes from the top).

### 3.1 Transformations

#### 3.1.1 2D Translation

The **translation** of a point in 2 dimensions is the movement of a point  $(x, y)$  to some other point  $(x', y')$ .

$$x' = x + a$$

$$y' = y + b$$

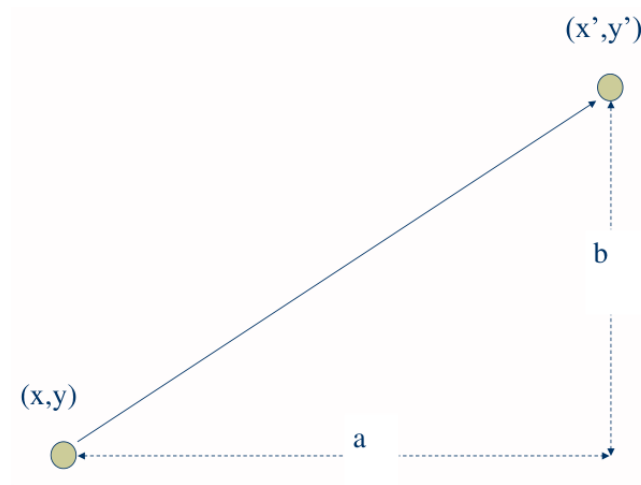


Figure 2: 2D Translation of a Point

#### 3.1.2 2D Rotation of a *Point*

The simplest rotation of a point around the origin is given by:

$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin \theta + y \cos \theta$$

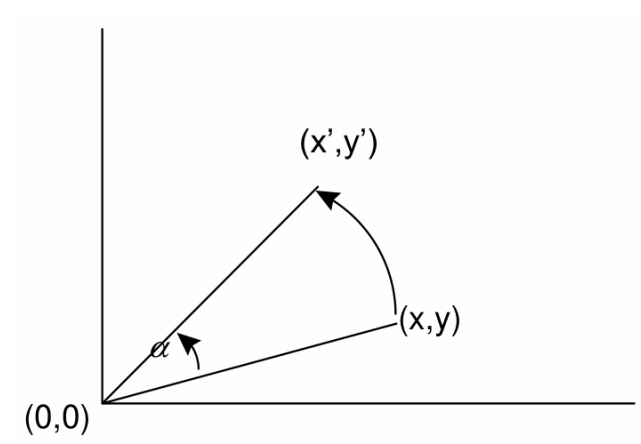


Figure 3: 2D Rotation of a Point

### 3.1.3 2D Rotation of an Object

In vector graphics, **objects** are defined as series of drawing operations (e.g., straight lines) performed on a set of vertices. To rotate a line or more complex object, we simply apply the equations to rotate a point to the  $(x, y)$  co-ordinates of each vertex.

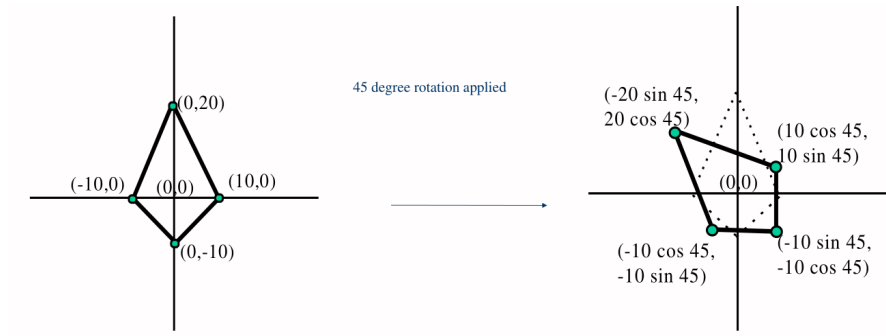


Figure 4: 2D Rotation of an Object

### 3.1.4 Arbitrary 2D Rotation

In order to rotate around an arbitrary point  $(a, b)$ , we perform translation, then rotation, then reverse the translation.

$$x' = a + (x - a) \cos \theta - (y - b) \sin \theta$$

$$y' = a + (x - a) \sin \theta + (y - b) \cos \theta$$

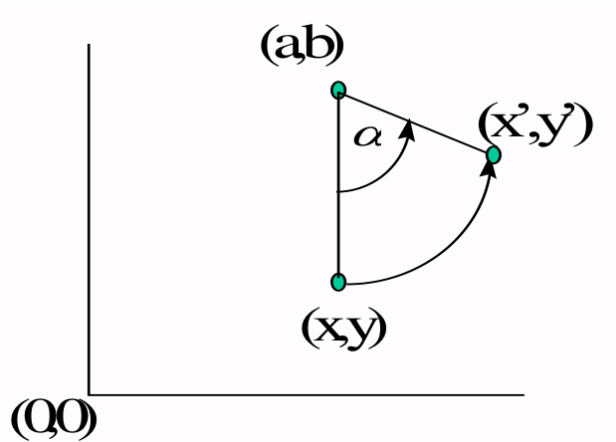


Figure 5: Arbitrary 2D Rotation

### 3.1.5 Matrix Notation

**Matrix notation** is commonly used for vector graphics as more complex operations are often easier in matrix format and because several operations can be combined easily into one matrix using matrix algebra.

Rotation about  $(0, 0)$ :

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$$

Translation:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & 0 & 1 \end{bmatrix}$$

### 3.1.6 Scaling

**Scaling** of an object is achieved by considering each of its vertices in turn, multiplying said vertex's  $x$  &  $y$  values by the scaling factor. A scaling factor of 2 will double the size of the object, while a scaling factor of 0.5 will halve it. It is possible to have different scaling factors for  $x$  &  $y$ , resulting in a **stretch**:

$$x' = x \times s$$

$$y' = y \times t$$

If the object is not centred on the origin, then scaling it will also effect a translation.

### 3.1.7 Order of Transformations

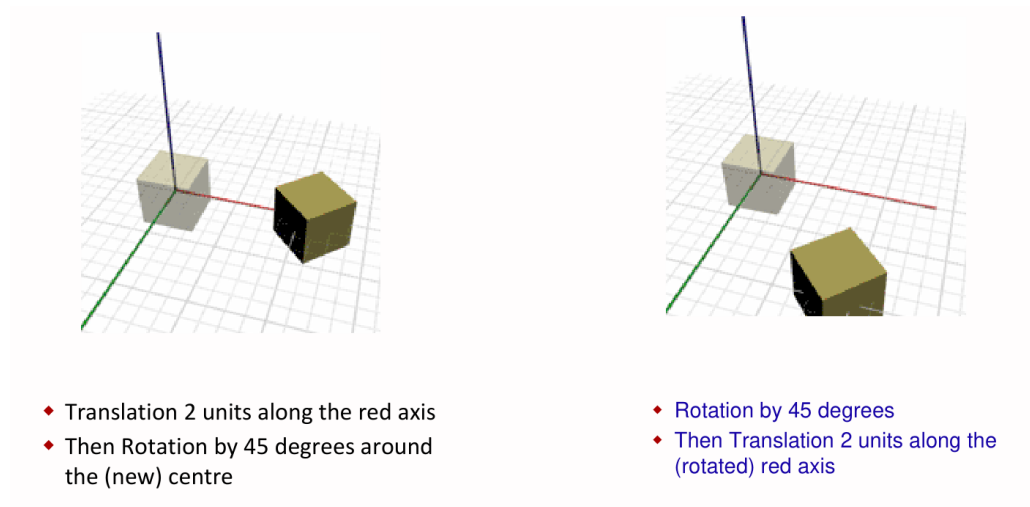


Figure 6: Order of Transformations

## 4 2D Raster Graphics

The raster approach to 2D graphics considers digital images to be grid-based arrays of pixels and operates on the images at the pixel level.

### 4.1 Introduction to HTML5/Canvas

**HTML** or HyperText Markup Language is a page-description language used primarily for website. **HTML5** brings major updates & improvements to the power of client-side web development.

A **canvas** is a 2D raster graphics component in HTML5. There is also a **canvas with 3D** (WebGL) which is a 3D graphics component that is more likely to be hardware-accelerated but is also more complex.

#### 4.1.1 Canvas: Rendering Contexts

`<canvas>` creates a fixed-size drawing surface that exposes one or more **rendering contexts**. The `getContext()` method returns an object with tools (methods) for drawing.

```

1 <html>
2   <head>
3     <script>
4       function draw() {
5         var canvas = document.getElementById("canvas");
6         var ctx = canvas.getContext("2d");

```



```

7         ctx.fillStyle = "rgb(200,0,0)";
8         ctx.fillRect (10, 10, 55, 50);
9         ctx.fillStyle = "rgba(0, 0, 200, 0.5)";
10        ctx.fillRect (30, 30, 55, 50);
11    }
12    </script>
13    </head>
14    <body onload="draw();">
15        <canvas id="canvas" width="150" height="150"></canvas>
16    </body>
17    </html>

```



Figure 7: Rendering of the Above HTML Code

#### 4.1.2 Canvas2D: Primitives

Canvas2D only supports one primitive shape: rectangles. All other shapes must be created by combining one or more *paths*. Fortunately, there are a collection of path-drawing functions which make it possible to compose complex shapes.

```

1  function draw(){
2      var canvas = document.getElementById('canvas');
3      var ctx = canvas.getContext('2d');
4      ctx.fillRect(125,25,100,100);
5      ctx.clearRect(145,45,60,60);
6      ctx.strokeRect(150,50,50,50);
7      ctx.beginPath();
8      ctx.arc(75,75,50,0,Math.PI*2,true); // Outer circle
9      ctx.moveTo(110,75);
10     ctx.arc(75,75,35,0,Math.PI,false); // Mouth (clockwise)
11     ctx.moveTo(65,65);
12     ctx.arc(60,65,5,0,Math.PI*2,true); // Left eye
13     ctx.moveTo(95,65);
14     ctx.arc(90,65,5,0,Math.PI*2,true); // Right eye
15     ctx.stroke(); // renders the Path that has been built up..
16 }

```



Figure 8: Rendering of the Above JavaScript Code

### 4.1.3 Canvas2D: drawImage()

The example below uses an external image as the backdrop of a small line graph:

```

1  function draw() {
2      var ctx = document.getElementById('canvas').getContext('2d');
3      var img = new Image();
4      img.src = 'backdrop.png';
5      img.onload = function() {
6          ctx.drawImage(img,0,0);
7          ctx.beginPath();
8          ctx.moveTo(30,96);
9          ctx.lineTo(70,66);
10         ctx.lineTo(103,76);
11         ctx.lineTo(170,15);
12         ctx.stroke();
13     }
14 }
```

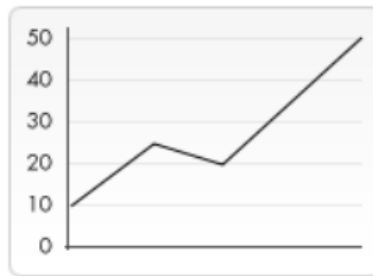


Figure 9: Rendering of the Above JavaScript Code

### 4.1.4 Canvas2D: Fill & Stroke Colours

```

1  <html>
2      <head>
3          <script>
4              function draw() {
5                  var canvas = document.getElementById("canvas");
6                  var context = canvas.getContext('2d');
7                  // Filled Star
8                  context.lineWidth=3;
9                  context.fillStyle="#CC00FF";
10                 context.strokeStyle="#ffff00"; // NOT lineWidth!
11                 context.beginPath();
12                 context.moveTo(100,50);
13                 context.lineTo(175,200);
14                 context.lineTo(0,100);
15                 context.lineTo(200,100);
16                 context.lineTo(25,200);
17                 context.lineTo(100,50);
18                 context.fill(); // colour the interior
19                 context.stroke(); // draw the lines
20             }
21         </script>
```

```

22     </head>
23     <body onload="draw();">
24         <canvas id="canvas" width="300" height="300"></canvas>
25     </body>
26 </html>

```

Colours can be specified by name (red), by a string of the form `rgb(r,g,b)`, or by hexadecimal colour codes `#RRGGBB`.



Figure 10: Rendering of the Above JavaScript Code

#### 4.1.5 Canvas2D: Translations

```

1 <html>
2   <head>
3     <script>
4       function draw() {
5         var canvas = document.getElementById("canvas");
6         var context = canvas.getContext('2d');
7         context.save(); // save the default (root) co-ord system
8         context.fillStyle="#CC00FF"; // purple
9         context.fillRect(100,0,100,100);
10        // translates from the origin, producing a nested co-ordinate system
11        context.translate(75,50);
12        context.fillStyle="#FFFF00"; // yellow
13        context.fillRect(100,0,100,100);
14        // transforms further, to produce another nested co-ordinate system
15        context.translate(75,50);
16        context.fillStyle="#0000FF"; // blue
17        context.fillRect(100,0,100,100);
18        context.restore(); // recover the default (root) co-ordinate system
19        context.translate(-75,90);
20        context.fillStyle="#00FF00"; // green
21        context.fillRect(100,0,100,100);
22      }
23    </script>
24  </head>
25  <body onload="draw();">
26    <canvas id="canvas" width="600" height="600"></canvas>
27  </body>
28 </html>

```

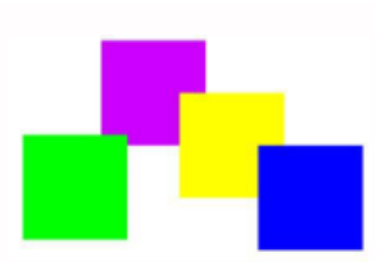


Figure 11: Rendering of the Above JavaScript Code

#### 4.1.6 Canvas2D: Order of Transformations

```

1 <html>
2   <head>
3     <script>
4       function draw() {
5         var canvas = document.getElementById("canvas");
6         var context = canvas.getContext('2d');
7         context.save(); // save the default (root) co-ord system
8         context.fillStyle="#CC00FF"; // purple
9         context.fillRect(0,0,100,100); // positioned with TL corner at 0,0
10        // translate then rotate
11        context.translate(100,0);
12        context.rotate(Math.PI/3);
13        context.fillStyle="#FF0000"; // red
14        context.fillRect(0,0,100,100); // positioned with TL corner at 0,0
15        // recover the root co-ord system
16        context.restore();
17        // rotate then translate
18        context.rotate(Math.PI/3);
19        context.translate(100,0);
20        context.fillStyle="#FFFF00"; // yellow
21        context.fillRect(0,0,100,100); // positioned with TL corner at 0,0
22      }
23    </script>
24  </head>
25  <body onload="draw();">
26    <canvas id="canvas" width="600" height="600"></canvas>
27  </body>
28 </html>

```

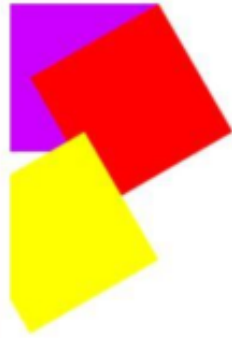


Figure 12: Rendering of the Above JavaScript Code

#### 4.1.7 Scaling

```

1 <html>
2   <head>
3     <script>
4       function draw() {
5         var canvas = document.getElementById("canvas");
6         var context = canvas.getContext('2d');
7         context.fillStyle="#CC00FF"; // purple
8         context.fillRect(0,0,100,100); // positioned with TL corner at 0,0
9         context.translate(150,0);
10        context.scale(2,1.5);
11        context.fillStyle="#FF0000"; // red
12        context.fillRect(0,0,100,100); // positioned with TL corner at 0,0
13      }
14    </script>
15  </head>
16  <body onload="draw();">
17    <canvas id="canvas" width="600" height="600"></canvas>
18  </body>
19 </html>

```

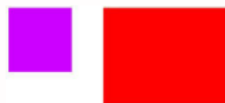


Figure 13: Rendering of the Above JavaScript Code

#### 4.1.8 Canvas2D: Programmatic Graphics

```

1 <html>
2   <head>
3     <script>
4       function draw() {
5         var canvas = document.getElementById("canvas");
6         var context = canvas.getContext('2d');
7         context.translate(150,150);
8         for (i=0;i<15;i++) {
9           context.fillStyle = "rgb("+(i*255/15)+",0,0)";

```

```

10         context.fillRect(0,0,100,100);
11         context.rotate(2*Math.PI/15);
12     }
13 }
14 </script>
15 </head>
16 <body onload="draw();" >
17     <canvas id="canvas" width="600" height="600"></canvas>
18 </body>
19 </html>

```

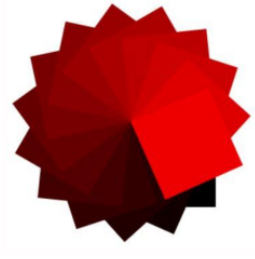


Figure 14: Rendering of the Above JavaScript Code

## 5 3D Co-Ordinate Systems

In a 3D co-ordinate system, a point  $P$  is referred to by three real numbers (co-ordinates):  $(x, y, z)$ . The directions of  $x$ ,  $y$ , &  $z$  are not universally defined but normally follow the **right-hand rule** for axes systems. In this case,  $z$  defined the co-ordinate's distance "out of" the monitor and negative  $z$  values go "into" the monitor.

### 5.1 Nested Co-Ordinate Systems

A **nested co-ordinate system** is defined as a translation relative to the world co-ordinate system. For example,  $-3.0$  units along the  $x$  axis,  $2.0$  units along the  $y$  axis, and  $2.0$  units along the  $z$  axis.

### 5.2 3D Transformations

#### 5.2.1 Translation

To translate a 3D point, modify each dimension separately:

$$x' = x + a_1$$

$$y' = y + a_2$$

$$z' = z + a_3$$

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a_1 & a_2 & a_3 & 1 \end{bmatrix}$$

#### 5.2.2 Rotation About Principal Axes

A **principal axis** is an imaginary line through the "center of mass" of a body around which the body rotates.

- Rotation around the  $x$ -axis is referred to as **pitch**.

- Rotation around the  $y$ -axis is referred to as **yaw**.
- Rotation around the  $z$ -axis is referred to as **roll**.

**Rotation matrices** define rotations by angle  $\alpha$  about the principal axes.

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha \\ 0 & -\sin \alpha & \cos \alpha \end{bmatrix}$$

To get new co-ordinates after rotation, multiply the point  $[x \ y \ z]$  by the rotation matrix:

$$[x' \ y' \ z'] = [x \ y \ z] R_x$$

For example, as a point rotates about the  $x$ -axis, its  $x$  component remains unchanged.

### 5.2.3 Rotation About Arbitrary Axes

You can rotate about any axis, not just the principal axes. You specify a 3D point, and the axis of rotation is defined as the line that joins the origin to this point (e.g., a toy spinning top will rotate about the  $y$ -axis, defined as  $(0, 1, 0)$ ). You must also specify the amount to rotate by, this is measured in radians (e.g.,  $2\pi$  radians is  $360^\circ$ ).

## 6 Graphics APIs

**Low-level** graphics APIs are libraries of graphics functions that can be accessed from a standard programming language. They are typically procedural rather than descriptive, i.e. the programmer calls the graphics functions which carry out operations immediately. The programmer also has to write all other application code: interface, etc. Procedural programming languages are typically faster than descriptive programming languages. Examples include OpenGL, DirectX, Vulkan, Java Media APIs. Examples that run in the browser include Canvas2D, WebGL, SVG.

**High-level** graphics APIs are ones in which the programmer describes the required graphics, animations, interactivity, etc. and doesn't need to deal with how this will be displayed & updated. They are typically descriptive rather than procedural and so are generally slower & less flexible because it is generally interpreted and rather general-purpose rather than task-specific. Examples include VRML/X3D.

### 6.1 Three.js

**WebGL (Web Graphics Library)** is a JavaScript API for rendering interactive 2D & 3D graphics within any compatible web browser without the use of plug-ins. WebGL is fully integrated with other web standards, allowing GPU-accelerated usage of physics & image processing and effects as part of the web page canvas.

**Three.js** is a cross-browser JavaScript library and API used to create & display animated 4D computer graphics in a web browser. Three.js uses WebGL.

```

1 <html>
2 <head>
3
4 <script src="three.js"></script>
5 <script>
6   'use strict'
7
8   function draw() {
9     // create renderer attached to HTML Canvas object
10    var c = document.getElementById("canvas");
11    var renderer = new THREE.WebGLRenderer({ canvas: c, antialias: true });
12

```

```

13 // create the scenegraph
14 var scene = new THREE.Scene();
15
16 // create a camera
17 var fov = 75;
18 var aspect = 600/600;
19 var near = 0.1;
20 var far = 1000;
21 var camera = new THREE.PerspectiveCamera( fov, aspect, near, far );
22 camera.position.z = 100;
23
24 // add a light to the scene
25 var light = new THREE.PointLight(0xFFFF00);
26 light.position.set(10, 30, 25);
27 scene.add(light);
28
29 // add a cube to the scene
30 var geometry = new THREE.BoxGeometry(20, 20, 20);
31 var material = new THREE.MeshLambertMaterial({color: 0xfd59d7});
32 var cube = new THREE.Mesh(geometry, material);
33 scene.add(cube);
34
35 // render the scene as seen by the camera
36 renderer.render(scene, camera);
37 }
38 </script>
39 </head>
40
41 <body onload="draw();">
42   <canvas id="canvas" width="600" height="600"></canvas>
43 </body>
44 </html>

```

Listing 1: “Hello World” in Three.js

In Three.js, a visible object is represented as a **mesh** and is constructed from a *geometry* & a *material*.

### 6.1.1 3D Primitives

Three.js provides a range of primitive geometry as well as the functionality to implement more complex geometry at a lower level. See <https://threejs.org/manual/?q=prim#en/primitives>.

```

1 <html>
2 <head>
3
4 <script src="three.js"></script>
5 <script>
6   'use strict'
7
8   var scene;
9
10  function addGeometryAtPosition(geometry, x, y, z) {
11    var material = new THREE.MeshLambertMaterial({color: 0xffffffff});
12    var mesh = new THREE.Mesh(geometry, material);

```



```

13     scene.add(mesh);
14     mesh.position.set(x,y,z);
15 }
16
17 function draw() {
18     // create renderer attached to HTML Canvas object
19     var c = document.getElementById("canvas");
20     var renderer = new THREE.WebGLRenderer({ canvas: c, antialias: true });
21
22     // create the scenegraph (global variable)
23     scene = new THREE.Scene();
24
25     // create a camera
26     var fov = 75;
27     var aspect = 400/600;
28     var near = 0.1;
29     var far = 1000;
30     var camera = new THREE.PerspectiveCamera( fov, aspect, near, far );
31     camera.position.z = 100;
32
33     // add a light to the scene
34     var light = new THREE.PointLight(0xFFFF00);
35     light.position.set(10, 0, 25);
36     scene.add(light);
37
38     // add a bunch of sample primitives to the scene
39     // see more here: https://threejsfundamentals.org/threejs/lessons/threejs-primitives.html
40
41     // args: width, height, depth
42     addGeometryAtPosition(new THREE.BoxGeometry(6,4,8), -50, 0, 0);
43
44     // args: radius, segments
45     addGeometryAtPosition(new THREE.CircleBufferGeometry(7, 24), -30, 0, 0);
46
47     // args: radius, height, segments
48     addGeometryAtPosition(new THREE.ConeBufferGeometry(6, 4, 24), -10, 0, 0);
49
50     // args: radiusTop, radiusBottom, height, radialSegments
51     addGeometryAtPosition(new THREE.CylinderBufferGeometry(4, 4, 8, 12), 20, 0, 0);
52
53     // arg: radius
54     // Polyhedrons
55     // (Dodecahedron is a 12-sided polyhedron, Icosahedron is 20-sided, Octahedron is 8-sided,
56     // ↪ Tetrahedron is 4-sided)
57     addGeometryAtPosition(new THREE.DodecahedronBufferGeometry(7), 40, 0, 0);
58     addGeometryAtPosition(new THREE.IcosahedronBufferGeometry(7), -50, 20, 0);
59     addGeometryAtPosition(new THREE.OctahedronBufferGeometry(7), -30, 20, 0);
60     addGeometryAtPosition(new THREE.TetrahedronBufferGeometry(7), -10, 20, 0);
61
62     // args: radius, widthSegments, heightSegments
63     addGeometryAtPosition(new THREE.SphereBufferGeometry(7,12,8), 20, 20, 0);
64
65     // args: radius, tubeRadius, radialSegments, tubularSegments

```

```

65     addGeometryAtPosition(new THREE.TorusBufferGeometry(5,2,8,24), 40, 20, 0);
66
67     // render the scene as seen by the camera
68     renderer.render(scene, camera);
69 }
70 </script>
71 </head>
72
73 <body onload="draw();">
74     <canvas id="canvas" width="600" height="600"></canvas>
75 </body>
76 </html>

```

Listing 2: Code Illustrating Some Primitives Provided by Three.js

### 6.1.2 Cameras

3D graphics API cameras allow you to define:

- The camera location ( $x, y, z$ ).
- The camera orientation (straight, ~~ray~~  $x$  rotation,  $y$  rotation,  $z$  rotation).
- The **viewing frustum** (the Field of View (FoV) & clipping planes).

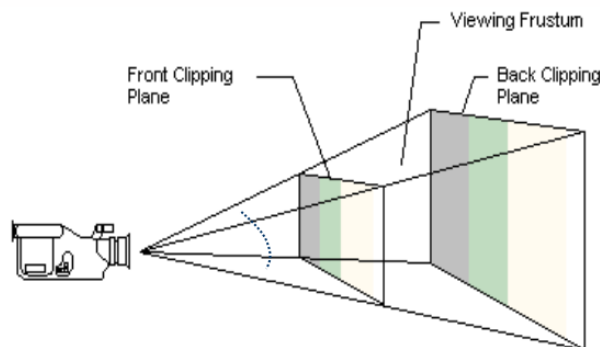


Figure 15: The Viewing Frustum

In Three.js, the FoV can be set differently in the vertical and horizontal directions via the first & second arguments to the constructor (fov, aspect). Generally speaking, the aspect ratio should match that of the canvas width & height to avoid the scene appearing to be stretched.

### 6.1.3 Lighting

Six different types of lights are available in both Three.js & WebGL:

- **Point lights:** rays emanate in all directions from a 3D point source (e.g., a lightbulb).
- **Directional lights:** rays emanate in one direction only from infinitely far away (similar effect rays from the Sun, i.e. very far away).
- **Spotlights:** project a cone of light from a 3D point source aimed at a specific target point.
- **Ambient lights:** simulate in a simplified way the lighting of an entire scene due to complex light/surface interactions – lights up everything in the scene regardless of position or occlusion.

- **Hemisphere lights:** ambient lights that affect the “ceiling” or “floor” hemisphere of objects rather than affecting them in their entirety.
- **RectAreaLights:** emit rectangular areas of light (e.g., fluorescent light strip).

```

1 <html>
2 <head>
3 <script src="three.js"></script>
4 <script>
5   'use strict'
6
7   function draw() {
8     // create renderer attached to HTML Canvas object
9     var c = document.getElementById("canvas");
10    var renderer = new THREE.WebGLRenderer({ canvas: c, antialias: true });
11
12    // create the scenegraph
13    var scene = new THREE.Scene();
14
15    // create a camera
16    var fov = 75;
17    var aspect = 600/600;
18    var near = 0.1;
19    var far = 1000;
20    var camera = new THREE.PerspectiveCamera( fov, aspect, near, far );
21    camera.position.set(0, 10, 30);
22
23    // add a light to the scene
24    var light = new THREE.PointLight(0xFFFFFF);
25    light.position.set(0, 10, 30);
26    scene.add(light);
27
28    // add a cylinder
29    // args: radiusTop, radiusBottom, height, radialSegments
30    var cyl = new THREE.Mesh(
31      new THREE.CylinderBufferGeometry(1, 1, 10, 12),
32      new THREE.MeshLambertMaterial({color: 0xAAAAAA} ) );
33    scene.add(cyl);
34
35    // clone the cylinder
36    var cyl2 = cyl.clone();
37
38    // modify its rotation by 60 degrees around its z axis
39    cyl2.rotateOnAxis(new THREE.Vector3(0,0,1), Math.PI/3);
40    scene.add(cyl2);
41    // clone the cylinder again
42    var cyl3 = cyl.clone();
43    scene.add(cyl3);
44    // set its rotation directly using "Euler angles", to 120 degrees on z axis
45    cyl3.rotation.set(0,0,2*Math.PI/3);
46
47    // render the scene as seen by the camera
48    renderer.render(scene, camera);

```

```

49     }
50   </script>
51 </head>
52
53 <body onload="draw();">
54   <canvas id="canvas" width="600" height="600"></canvas>
55 </body>
56 </html>

```

Listing 3: Rotation Around a Local Origin in Three.js

#### 6.1.4 Nested Co-Ordinates

**Nested co-ordinates** help manage complexity as well as promote reusability & simplify the transformations of objects composed of multiple primitive shapes. In Three.js, 3D objects have a `children` array; a child can be added to an object using the method `.add(childObject)`, i.e. nesting the child object's transform within the parent object. Objects have a parent in the scene graph so when you set their transforms (translation, rotation) it's relative to that parent's local co-ordinate system.

```

1 <html>
2 <head>
3
4   <script src="three.js"></script>
5   <script>
6     'use strict'
7
8     function draw() {
9       // create renderer attached to HTML Canvas object
10      var c = document.getElementById("canvas");
11      var renderer = new THREE.WebGLRenderer({ canvas: c, antialias: true });
12
13      // create the scenegraph
14      var scene = new THREE.Scene();
15
16      // create a camera
17      var fov = 75;
18      var aspect = 600/600;
19      var near = 0.1;
20      var far = 1000;
21      var camera = new THREE.PerspectiveCamera( fov, aspect, near, far );
22      camera.position.set(0, 1.5, 6);
23
24      // add a light to the scene
25      var light = new THREE.PointLight(0xFFFFFF);
26      light.position.set(0, 10, 30);
27      scene.add(light);
28
29      // desk lamp base
30      // args: radiusTop, radiusBottom, height, radialSegments
31      var base = new THREE.Mesh(
32        new THREE.CylinderBufferGeometry(1, 1, 0.1, 12),
33        new THREE.MeshLambertMaterial({color: 0xAAAAAA} ) );
34      scene.add(base);

```

```

35
36 // desk lamp first arm piece
37 var arm = new THREE.Mesh(
38     new THREE.CylinderBufferGeometry(0.1, 0.1, 3, 12),
39     new THREE.MeshLambertMaterial({color: 0xAAAAAA}) );
40
41 // since we want to rotate around a point other than the arm's centre,
42 // we can create a pivot point as the parent of the arm, position the
43 // arm relative to that pivot point, and apply rotation on the pivot point
44 var pivot = new THREE.Object3D();
45 // centre of rotation we want
46 // (in world coordinates, since pivot is not yet a child of the base)
47 pivot.position.set(0, 0, 0);
48 pivot.add(arm); // pivot is parent of arm
49 base.add(pivot); // base is parent of pivot
50
51 // translate arm relative to its parent, i.e. 'pivot'
52 arm.position.set(0, 1.5, 0);
53 // rotate pivot point relative to its parent, i.e. 'base'
54 pivot.rotateOnAxis(new THREE.Vector3(0,0,1), -Math.PI/6);
55
56 // clone a second arm piece (consisting of a pivot with a cylinder as its child)
57 var pivot2 = pivot.clone();
58 // add as a child of the 1st pivot
59 pivot.add(pivot2);
60 // rotate the 2nd pivot relative to the 1st pivot (since it's nested)
61 pivot2.rotation.z = Math.PI/3;
62 // translate the 2nd pivot relative to the 1st pivot
63 pivot2.position.set(0,3,0);
64
65 // TEST: we can rotate the 1st arm piece and the 2nd arm piece should stay correct
66 pivot.rotateOnAxis(new THREE.Vector3(0,0,1), Math.PI/12);
67
68 // TEST: we can also move the base, and everything stays correct
69 base.position.x -= 3;
70
71 // render the scene as seen by the camera
72 renderer.render(scene, camera);
73 }
74 </script>
75 </head>
76
77 <body onload="draw();">
78     <canvas id="canvas" width="600" height="600"></canvas>
79 </body>
80 </html>

```

Listing 4: Partial Desk Lamp with Nested Objects

The above code creates a correctly set-up hierarchy of nested objects, allowing us to:

- Translate the base while the two arms remain in the correct relative position.
- Rotate the first arm while keeping the second arm in the correct position.

### 6.1.5 Geometry Beyond Primitives

In Three.js, the term “low-level geometry” is used to refer to geometry objects consisting of vertices, faces, & normal.

## 7 Animation & Interactivity

### 7.1 Handling the Keyboard

Handling the keyboard involves recognising keypresses and updating the graphics in response.

```

1 <html>
2
3 <head>
4   <script>
5     function attachEvents() {
6       document.onkeypress = function (event) {
7         var xoffset = 10 * parseInt(String.fromCharCode(event.keyCode || event.charCode));
8         draw(xoffset);
9       }
10    }
11
12    function draw(xoffset) {
13      var canvas = document.getElementById("canvas");
14      var context = canvas.getContext('2d');
15
16      // remove previous translation if any
17      context.save();
18
19      // over-write previous content, with a white rectangle
20      context.fillStyle = "#FFFFFF";
21      context.fillRect(0, 0, 300, 300);
22
23      // translate based on numerical keypress
24      context.translate(xoffset, 0);
25
26      // purple rectangle
27      context.fillStyle = "#CC00FF";
28      context.fillRect(0, 0, 50, 50);
29      context.restore();
30    }
31  </script>
32 </head>
33
34 <body onload="attachEvents();">
35   <canvas id="canvas" width="300" height="300"></canvas>
36 </body>
37
38 </html>

```

Listing 5: Keyboard Handling (Canvas/JavaScript)

## 7.2 Mouse Handling

```

1 <html>
2
3 <head>
4   <script>
5     var isMouseDown = false;
6     function attachEvents() {
7       document.onmousedown = function (event) {
8         isMouseDown = true;
9         draw(event.clientX, event.clientY);
10      }
11      document.onmouseup = function (event) {
12        isMouseDown = false;
13      }
14      document.onmousemove = function (event) {
15        if (isMouseDown) {
16          draw(event.clientX, event.clientY);
17        }
18      }
19    }
20    function draw(xoffset, yoffset) {
21
22      var canvas = document.getElementById("canvas");
23      var context = canvas.getContext('2d');
24
25      // remove previous translation if any
26      context.save();
27      // over-write previous content, with a grey rectangle
28      context.fillStyle = "#DDDDDD";
29      context.fillRect(0, 0, 600, 600);
30      // translate based on position of mouseclick
31      context.translate(xoffset, yoffset);
32      // purple rectangle
33      context.fillStyle = "#CC00FF";
34      context.fillRect(-25, -25, 50, 50); // centred on coord system
35      context.restore();
36
37    }
38  </script>
39 </head>
40
41 <body onload="attachEvents(); draw(0,0);">
42   hello<br>
43   <div id='canvasdiv' style='position:absolute; left:0px; top:0px;'><canvas id="canvas"
44     ↪ width="600"
45     height="600"></canvas></div>
46
47 </body>
48 </html>

```

Listing 6: Mouse Handling (Canvas/JavaScript)

### 7.3 Time-Based Animation

Time-based animation can be achieved using `window.setTimeout()` which repaints the canvas at pre-defined intervals.

```

1 <html>
2
3 <head>
4   <script>
5
6     var x = 0, y = 0;
7     var dx = 4, dy = 5;
8
9     function draw() {
10
11       var canvas = document.getElementById("canvas");
12       var context = canvas.getContext('2d');
13
14       // remove previous translation if any
15       context.save();
16       // over-write previous content, with a grey rectangle
17       context.fillStyle = "#DDDDDD";
18       context.fillRect(0, 0, 600, 600);
19       // perform movement, and translate to position
20       x += dx;
21       y += dy;
22       if (x <= 0)
23         dx = 4;
24       else if (x >= 550)
25         dx = -4;
26       if (y <= 0)
27         dy = 5;
28       else if (y >= 550)
29         dy = -5;
30       context.translate(x, y);
31       // purple rectangle
32       context.fillStyle = "#CC00FF";
33       context.fillRect(0, 0, 50, 50);
34       context.restore();
35
36       // do it all again in 1/30th of a second
37       window.setTimeout("draw();", 1000 / 30);
38     }
39   </script>
40 </head>
41
42 <body onload="draw();">
43   <canvas id="canvas" width="600" height="600"></canvas>
44 </body>
45
46 </html>

```

Listing 7: Time-Based Animation with `window.setTimeout()`

However, improved smoothness can be achieved using `window.requestAnimationFrame()` which is called at every window repaint/refresh.



```

1 <html>
2 <head>
3   <script>
4     var x = 0, y = 0;
5     var dx = 4, dy = 5;
6     var now = Date.now();
7
8     function draw() {
9       // do it all again in 1/60th of a second
10      window.requestAnimationFrame(draw);
11
12      var elapsedMs = Date.now() - now;
13      now = Date.now();
14
15      var canvas = document.getElementById("canvas");
16      var context = canvas.getContext('2d');
17
18      // remove previous translation if any
19      context.save();
20
21      // over-write previous content, with a grey rectangle
22      context.fillStyle = "#DDDDDD";
23      context.fillRect(0, 0, 600, 600);
24
25      // perform movement, and translate to position
26      x += dx * elapsedMs / 16.7;
27      y += dy * elapsedMs / 16.7;
28
29      if (x <= 0)
30        dx = 4;
31      else if (x >= 550)
32        dx = -4;
33      if (y <= 0)
34        dy = 5;
35      else if (y >= 550)
36        dy = -5;
37
38      context.translate(x, y);
39
40      // purple rectangle
41      context.fillStyle = "#CC00FF";
42      context.fillRect(0, 0, 50, 50);
43      context.restore();
44    }
45  </script>
46 </head>
47 <body onload="draw();">
48   <canvas id="canvas" width="600" height="600"></canvas>
49 </body>
50 </html>

```

Listing 8: Smoother Time-Based Animation with `window.requestAnimationFrame()`

## 7.4 Raycasting

**Raycasting** is a feature offered by 3D graphics APIs which computes a ray from a start position in a specified direction and identifies the geometry that the ray hits.

```
1  renderer = new THREE.WebGLRenderer({ canvas: c, antialias: true });
```

The following example illustrates the use of raycasting/picking and rotation/translation based on mouse selection and mouse movement. It also illustrates how nested co-ordinate systems have been used to make the lamp parts behave correctly.

```
1  <html>
2
3  <head>
4
5      <script src="../../week2/examples/three.js"></script>
6      <script>
7
8          'use strict'
9
10         var raycaster, renderer, scene, camera;
11         var selectedObject = null;
12         var selectableObjects = [];
13         var lastMousePos = {x: 0, y: 0};
14
15         function draw() {
16             // create renderer attached to HTML Canvas object
17             var c = document.getElementById("canvas");
18             renderer = new THREE.WebGLRenderer({canvas: c, antialias: true});
19
20             // create the scenegraph
21             scene = new THREE.Scene();
22
23             // create a camera
24             var fov = 75;
25             var aspect = 600 / 600;
26             var near = 0.1;
27             var far = 1000;
28             camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
29             camera.position.set(-5, 1.5, 6);
30
31             // add a light to the scene
32             var light = new THREE.PointLight(0xFFFFFF);
33             light.position.set(0, 10, 30);
34             scene.add(light);
35
36             // desk lamp base
37             // args: radiusTop, radiusBottom, height, radialSegments
38             var base = new THREE.Mesh(
39                 new THREE.CylinderBufferGeometry(1, 1, 0.1, 12),
40                 new THREE.MeshLambertMaterial({color: 0xAAAAAA}));
41             scene.add(base);
42             base.position.set(-5, -2, 0);
43             selectableObjects.push(base);
```

```

44     base.canTranslate = true; // I added this property
45
46     // desk lamp first arm piece
47     var arm = new THREE.Mesh(
48         new THREE.CylinderBufferGeometry(0.1, 0.1, 3, 12),
49         new THREE.MeshLambertMaterial({color: 0xAAAAAA}));
50
51     // since we want to rotate around a point other than the arm's centre,
52     // we can create a pivot point as the parent of the arm, position the
53     // arm relative to that pivot point, and apply rotation on the pivot point
54     var pivot = new THREE.Object3D();
55     pivot.position.set(0, 0, 0); // centre of rotation we want
56     pivot.add(arm); // pivot is parent of arm
57     base.add(pivot); // base is parent of pivot
58     selectableObjects.push(arm);
59     arm.canRotate = true; // I added this property
60
61     // translate arm relative to pivot point
62     arm.position.set(0, 1.5, 0);
63     // rotate pivot point relative to the world
64     pivot.rotateOnAxis(new THREE.Vector3(0, 0, 1), -Math.PI / 6);
65
66     // second arm piece (consisting of a pivot with a cylinder as its child)
67     var pivot2 = pivot.clone();
68     pivot.add(pivot2);
69     // rotate the 2nd pivot relative to the 1st pivot (since it's nested)
70     pivot2.rotation.z = Math.PI / 3;
71     // translate the 2nd pivot relative to the 1st pivot
72     pivot2.position.set(0, 3, 0);
73     var arm2 = pivot2.children[0];
74     selectableObjects.push(arm2);
75     arm2.canRotate = true; // I added this property
76
77     // args: radius, height, segments
78     var lampshade = new THREE.Mesh(
79         new THREE.ConeBufferGeometry(1, 0.7, 24),
80         new THREE.MeshLambertMaterial({color: 0xAAAAAA})
81     );
82     var shadePivot = new THREE.Object3D();
83     pivot2.add(shadePivot); // lampshade pivot is a child of the 2nd arm pivot
84     shadePivot.add(lampshade);
85     shadePivot.position.set(0, 3, 0);
86     shadePivot.rotation.x = Math.PI;
87     selectableObjects.push(lampshade);
88     lampshade.canRotate = true; // I added this property
89
90     raycaster = new THREE.Raycaster();
91
92     c.onmousedown = handleMouseDown;
93     c.onmousemove = handleMouseMove;
94     c.onmouseup = function (e) {
95         selectedObject = null;
96     };

```

```

97     animate();
98 }
99
100
101 function animate() {
102     setTimeout(animate, 1000 / 60);
103
104     // render the scene as seen by the camera
105     renderer.render(scene, camera);
106 }
107
108 function handleMouseDown(e) {
109     // handle mouse-clicks on the canvas
110     // did the user click a mesh?
111     /* note that 0,0 is the centre of the canvas according to WebGL,
112        and the canvas extends from (-1,-1) to (1,1)
113        but 0,0 is the top-left of the canvas according to e.clientX,e.clientY,
114        and the canvas extends from (0,0) to (599,599)
115     */
116     var x = 2 * (e.clientX - 300) / 600;
117     var y = -2 * ((e.clientY - 300) / 600);
118
119     lastMousePos.x = x;
120     lastMousePos.y = y;
121
122     // set up and apply the raycaster (we are returned an array of intersection objects)
123     raycaster.setFromCamera({x: x, y: y}, camera);
124     var intersects = raycaster.intersectObjects(selectableObjects);
125     if (intersects.length > 0) {
126         var closestObj, closestDist;
127
128         for (var i = 0; i < intersects.length; i++) {
129             /*
130                 An intersection has the following properties :
131                 - object : intersected object (THREE.Mesh)
132                 - distance : distance from ray start to intersection (number)
133                 - face : intersected face (THREE.Face3)
134                 - faceIndex : intersected face index (number)
135                 - point : intersection point (THREE.Vector3)
136                 - uv : intersection point in the object's UV coordinates
137                   ↪ (THREE.Vector2)
138             */
139             if (i == 0 || intersects[i].distance < closestDist) {
140                 closestObj = intersects[i].object;
141                 closestDist = intersects[i].distance;
142             }
143         }
144
145         selectedObject = closestObj;
146     }
147     else
148         selectedObject = null;

```

```

149     }
150
151     function handleMouseMove(e) {
152         if (selectedObject != null) {
153             var x = 2 * (e.clientX - 300) / 600;
154             var y = -2 * ((e.clientY - 300) / 600);
155             // dx,dy is the amount the mouse just moved by in pixels
156             var dx = x - lastMousePos.x;
157             var dy = y - lastMousePos.y;
158
159             if (selectedObject.canRotate) {
160                 // rotate the parent ('pivot') that the object is a child of
161                 selectedObject.parent.rotation.x += dx;
162                 selectedObject.parent.rotation.z += dy;
163             }
164             else if (selectedObject.canTranslate) {
165                 // translate the object
166                 selectedObject.position.x += dx * 4;
167                 selectedObject.position.z -= dy * 4;
168             }
169
170             lastMousePos.x = x;
171             lastMousePos.y = y;
172         }
173     }
174 </script>
175 </head>
176
177 <body onload="draw();">
178     <!-- Note that the canvas has been positioned precisely at 0,0 so that mouse positions on the
179     ↪ browser
180     are the same as mouse positions on the canvas -->
181     <canvas id="canvas" width="600" height="600" style="position:absolute; left:0px;
182     ↪ top:0px"></canvas>
183 </body>
</html>

```

Listing 9: Controllable Desk Lamp

## 7.5 Shading Algorithms

The colour at any pixel on a polygon is determined by:

- The characteristics (including colour) of the surface itself.
- Information about light sources (ambient, directional, parallel, point, or spot) and their positions relative to the surface.
- *Diffuse & specular* reflections.

Classic shading algorithms include:

- Flat shading.
- Smooth shading (Gourard).

- Normal Interpolating Shading (Phong).



Figure 16: Different Shading Algorithms

### 7.5.1 Flat Shading

**Flat shading** calculates and applies directly the shade of each surface, which is calculated via the cosine of the angle of incidence ray to the *surface normal* (a **surface normal** is a vector perpendicular to the surface).

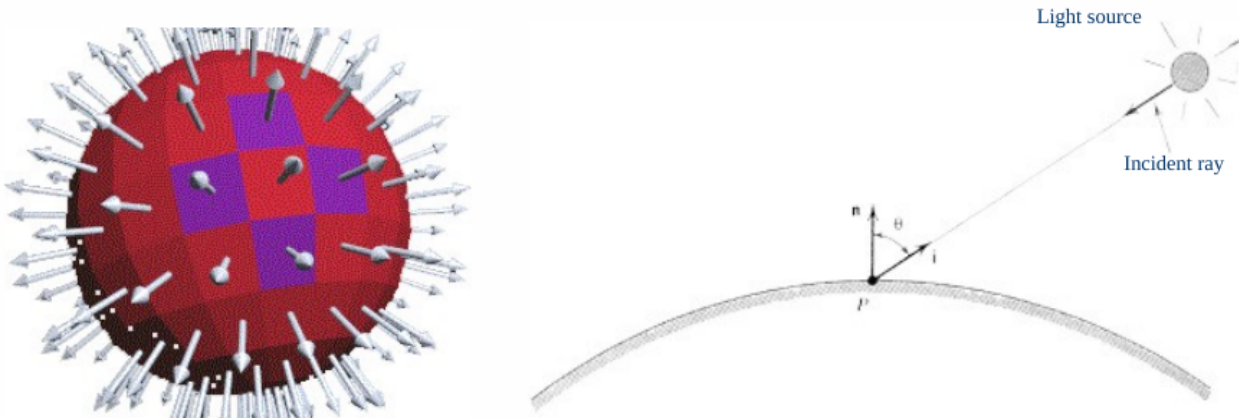


Figure 17: Flat Shading

### 7.5.2 Smooth (Gourard) Shading

**Smooth (Gourard) shading** calculates the shade at each vertex, and interpolates (smooths) these shades across the surfaces. Vertex normals are calculated by averaging the normals of the connected faces. Interpolation is often carried out in graphics hardware, making it generally very fast.

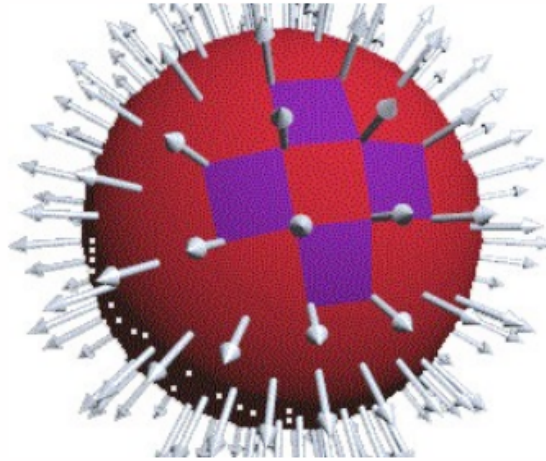


Figure 18: Smooth Shading

### 7.5.3 Normal Interpolating (Phong) Shading

**Normal interpolating (Phong) shading** calculates the normal at each vertex and interpolates these normals across the surfaces. The light, and therefore the shade at each pixel is individually calculated from its unique surface normal.

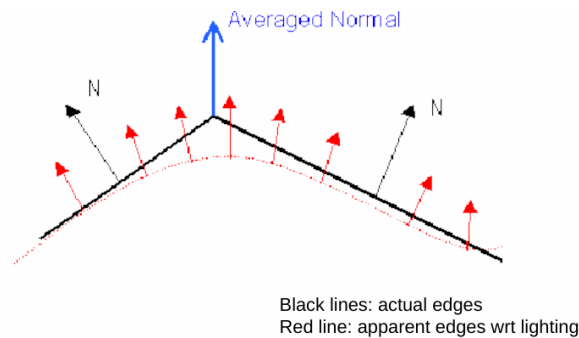


Figure 19: Normal Interpolating (Phong) Shading

## 7.6 Shading in Three.js

In Three.js, **materials** define how objects will be shaded in the scene. There are three different shading models to choose from:

- `MeshBasicMaterial`: none.
- `MeshPhongMaterial` (with `flatShading = true`): flat shading.
- `MeshLambertMaterial`: Gourard shading.

## 7.7 Shadows in Three.js

Three.js supports the use of shadows although they are expensive to use. The scene is redrawn for each shadow-casting light, and finally composed from all the results. Games sometimes use fake “blob shadows” instead of proper shadows or else only let one light cast shadows to save computation.

## 7.8 Reflectivity of Materials in Three.js

There are a variety of colour settings in Three.js

- **Diffuse colour** is defined by the colour of the material.
- **Specular colour** is the colour of specular highlights (in Phong shading only).
- **Shininess** is the strength of specular highlights (in Phong only).
- **Emissive colour** is not affected by lighting.

## 8 Image Processing

The difference between Graphics, Image Processing, & Computer Vision is as follows:

- **Graphics** is the processing & display of images of objects that exist conceptually rather than physically, with emphasis on the generation of an image from a model of the objects, illumination, etc. and an emphasis on the rendering efficiency for real-time display and/or realism.
- **Image Processing** is the processing & analysis of images of the real world, with an emphasis on the modification of the image.
- **Computer Vision** uses image processing, techniques from AI, & pattern recognition to recognise & categorise image data and extract domain-specific information from these images.

Image processing techniques include:

- **Image Enhancement:** provide a more effective display of data for visual interpretation or increase the visual distinction between features in the image.
- **Image Restoration:** correction of geometrics distortions, de-noising, de-blurring, etc.
- **Feature Extraction:** the extraction of useful features, e.g. corners, blobs, edges, & lines, the extraction of image segments & regions of interest, and the subtraction of background to extract foreground objects.

Image processing applications include industrial inspection, document image analysis, traffic monitoring, security and surveillance, remote sensing, scientific imaging, medical imaging, robotics and autonomous systems, face analysis and biometric, & entertainment

### 8.1 Introduction to OpenCV

**Open Source Computer Vision Library (OpenCV)** is an open source computer vision & machine learning software library which is available for Python, C++, JavaScript, & Java. It is a good choice for high-performance image processing and for making use of pre-built library functions.

You can also write image processing code directly using Canvas & JavaScript or Matlab Image Processing & Computer Vision toolboxes.

### 8.2 Point Transformations

**Point transformations** modify the value of each pixel.

#### 8.2.1 Histogram Manipulation

A **histogram** is a graphical representation of the distribution of pixel intensity values in an image. It displays the number of pixels for each intensity level, allowing us to understand the image's overall brightness & contrast. Histogram manipulation utilises the following processes:

- **Contrast Stretching:** enhancing the contrast of an image by spreading out the intensity values across the available range, with the goal of making the distinction between different pixel values more apparent. **Uniform expansion** / **linear stretch** is a method of contrast stretching that assigns a proportional number of grey levels to both frequently & rarely occurring pixel values. It aims to ensure that every pixel intensity has a chance to



occupy the full range of values, and can be applied either globally (to the entire image) or locally (to smaller regions).

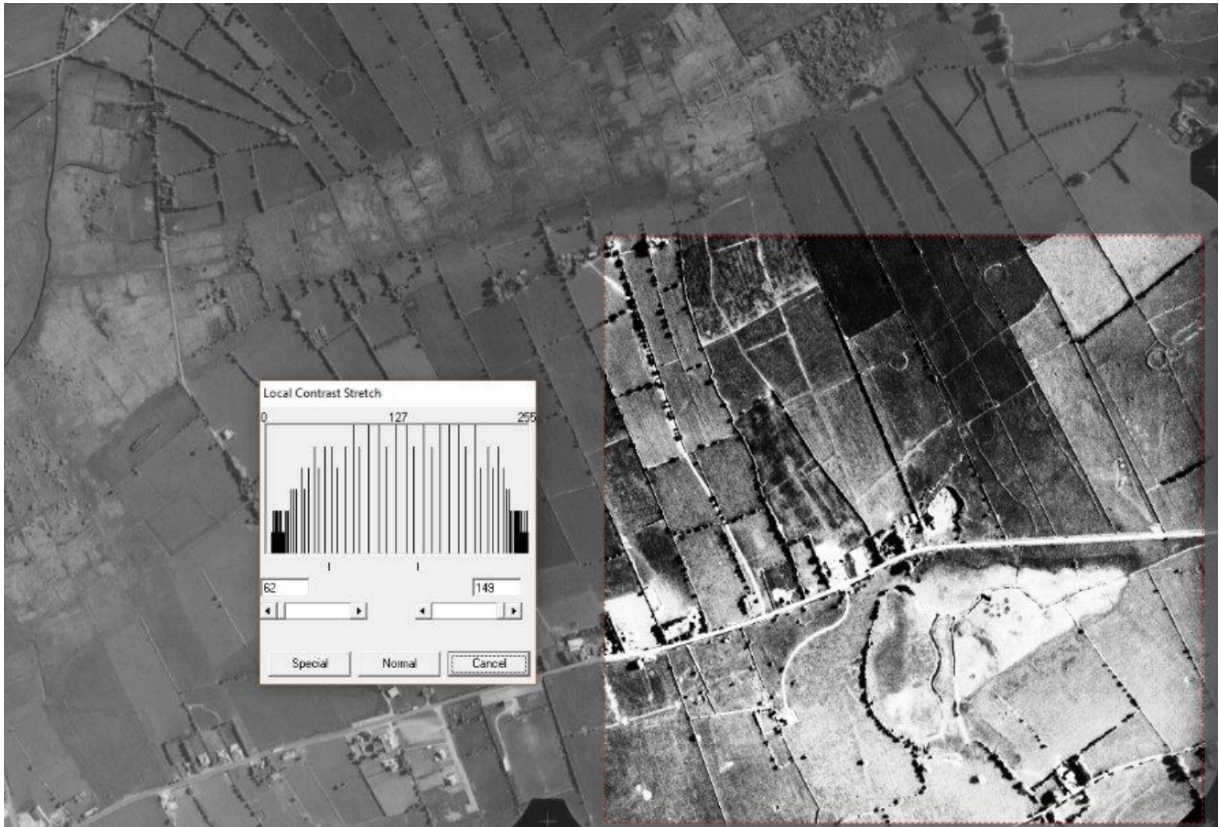


Figure 20: Aerial Photograph with a Local Histogram Stretch Applied

- **Histogram Equalisation** is a process that adjusts the intensity distribution of an image to achieve a uniform histogram. It enhances contrast by redistributing pixel values, which helps in revealing more details, particularly in images where some pixel values are too dominant or too rare.

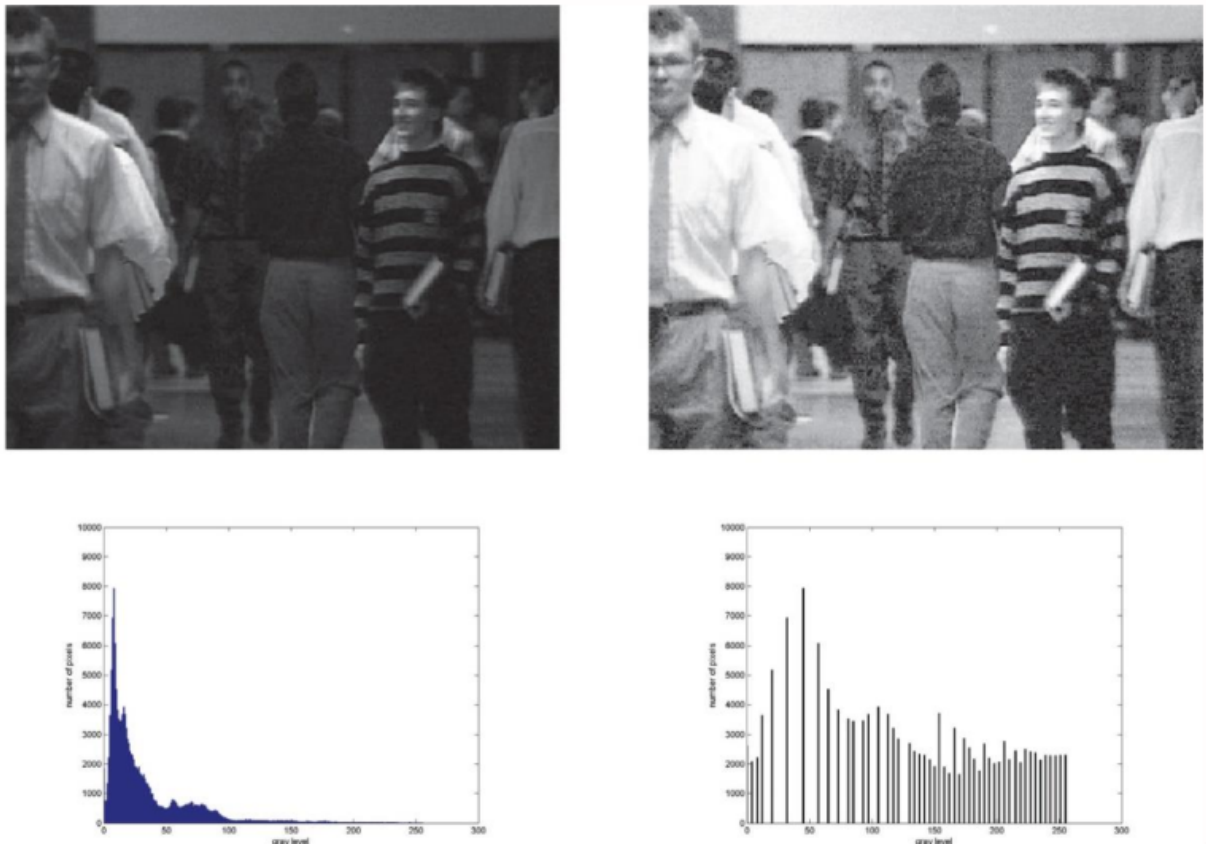


Figure 21: Histogram Equalisation Applied with Discretisation Effects

### 8.2.2 Thresholding

**Thresholding** is a simple segmentation technique that is very useful for separating solid objects from a contrasting background. All pixels above a determined threshold grey level are assumed to belong to the object, and all pixels below that level are assumed to be outside that object (or vice-versa). The selection of the threshold level is very important: problems of over-segmentation exist (false negatives) and under-segmentation (false positives).

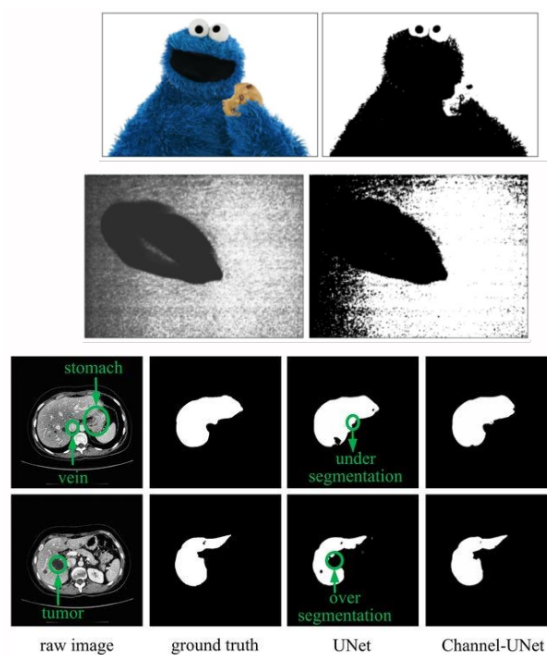


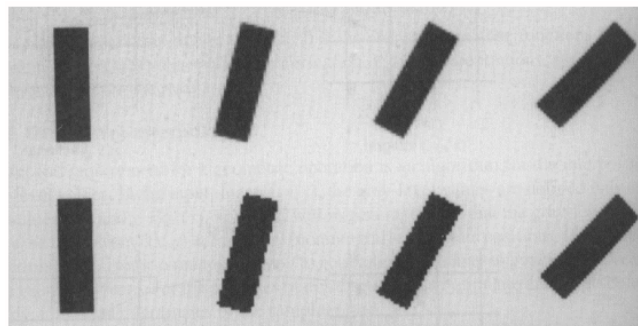
Figure 22: Thresholding Examples

### 8.3 Geometric Transformations

#### 8.3.1 Interpolation

As a 2D array of values, a digital image can be thought of as a sampling of an underlying continuous function. Interpolation estimates this underlying continuous function by computing pixel values at real-valued co-ordinates where the estimated continuous function must coincide with the sampled data at the sample points. Types of interpolation include:

- Nearest-neighbour interpolation.
- Bilinear interpolation which calculates a distance-weighted average of the four pixels closest to the target sub-pixel position  $(v, w)$ .
- Bicubic interpolation which is more accurate but costly, where derivatives of the underlying function are also estimated.



Using a rotation operation as an example, we can see how bilinear interpolation is superior to nearest neighbour interpolation

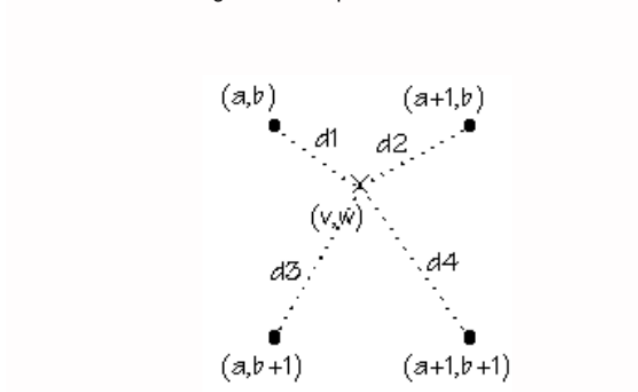


Figure 23: Interpolation Examples

#### 8.3.2 Warping

**Warping** consists of arbitrary geometric transformations from real-valued co-ordinates  $(x, y)$  to real-valued co-ordinates  $(x', y')$  where the mapping function  $f(x, y)$  specifies the transformation, or *warping*. Applications include image rectification, image registration, map projection, & image morphing.

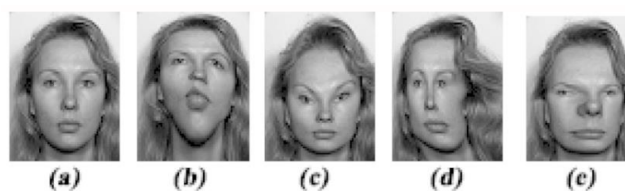


Figure 24: Warping Examples

### 8.3.3 Image Rectification

**Image rectification** (part of camera calibration) is a standard approach to geometric correction consisting of displacement values for specified control points in the image. Displacement of non-control points is determined through *interpolation*. For example, take a photograph of a rectangular grid and then determine the mapping required to move output control points back to known undistorted positions.

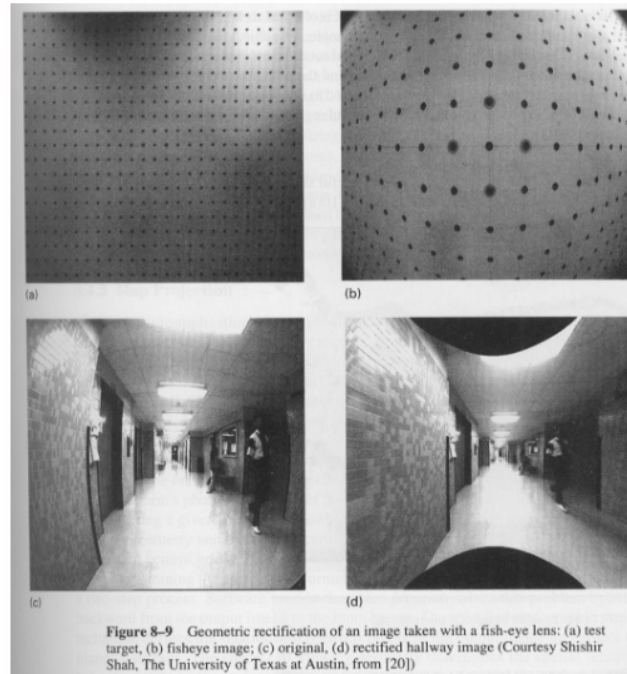


Figure 25: Image Rectification Examples

### 8.3.4 Image Registration

**Image registration** is the process of transforming different sets of data into one co-ordinate system. Geometric operations are applied to images for the purposes of comparison, monitoring, measurement, etc. and have many applications in medical imaging.

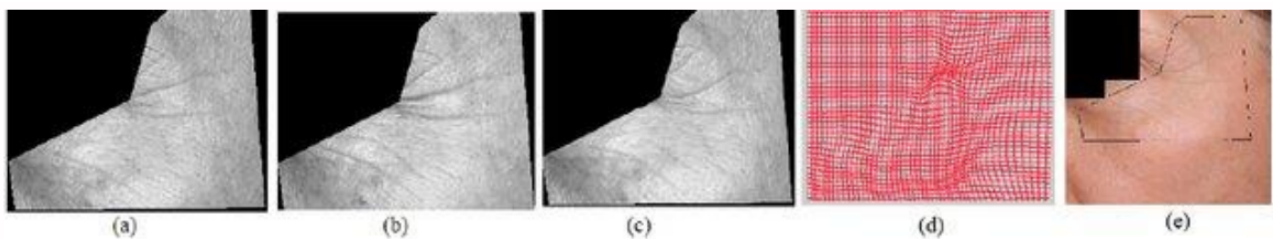


Figure 26: Image Registration Examples

### 8.3.5 Map Projection

Aerial or spaceborne images of the surface of a planet may be rectified into photomaps. Both oblique and orthogonol photographs require correction for this application due to the shape of the surface being imaged.

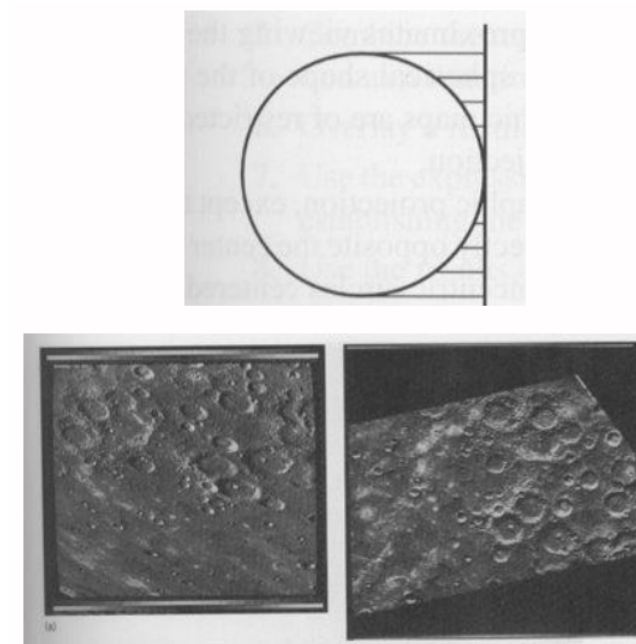


Figure 27: Map Projection Examples

### 8.3.6 Image Morphing

**Image morphing** gradually transforms one image into another over a number of animation frames. It involves a dissolve from one image to the other (i.e., gradual change of pixel values) as well as an incremental geometric operation using control points (e.g., nostrils, eyes, chin, etc.).

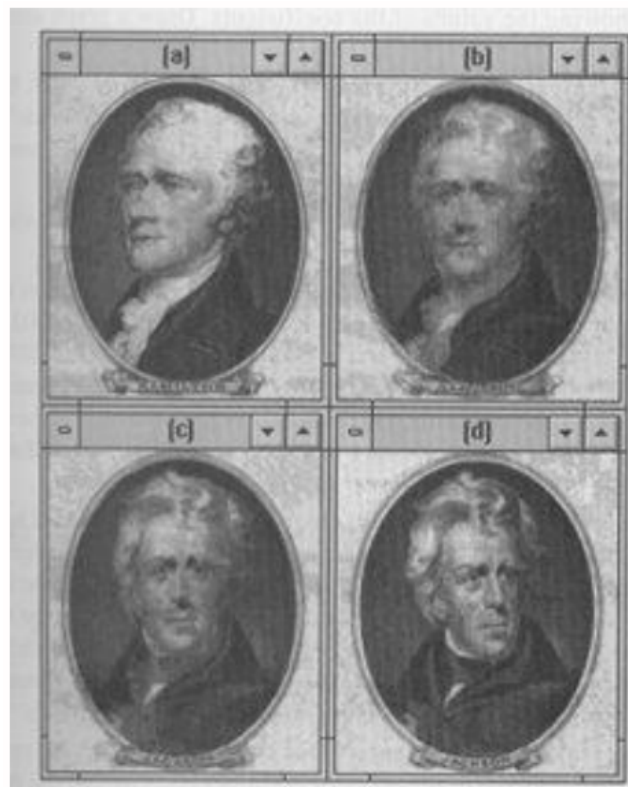


Figure 28: Image Morphing Examples



## 9 Spatial Filtering

Spatial filtering is a fundamental local operation in image processing that is used for a variety of tasks, including noise removal, blurring, sharpening, & edge detection. It establishes a moving window called a **kernel** which contains an array of coefficients or weighting factors. The kernel is then moved across the original image so that it centres on each pixel in turn. Each coefficient in the kernel is multiplied by the value of the pixel below it, and the addition of each of these values determines the value of the pixel in the output image corresponding to the pixel in the centre of the kernel.

- **Smoothing kernels** perform an averaging of the values in a local neighbourhood and therefore reduce the effects of noise. Such kernels are often used as the first stage of pre-processing an image that has been corrupted by noise in order to restore the original image.
- **Differentiating kernels** accentuate the places where the signal is changing in value rapidly. They are used to extract useful information from images, such as the boundaries of objects, for purposes such as object detection.



Figure 29: Spatial Filtering: Smoothing Filters

For symmetric kernels with real numbers and signals with real values (as is the case with images), convolution is the same as cross-correlation.

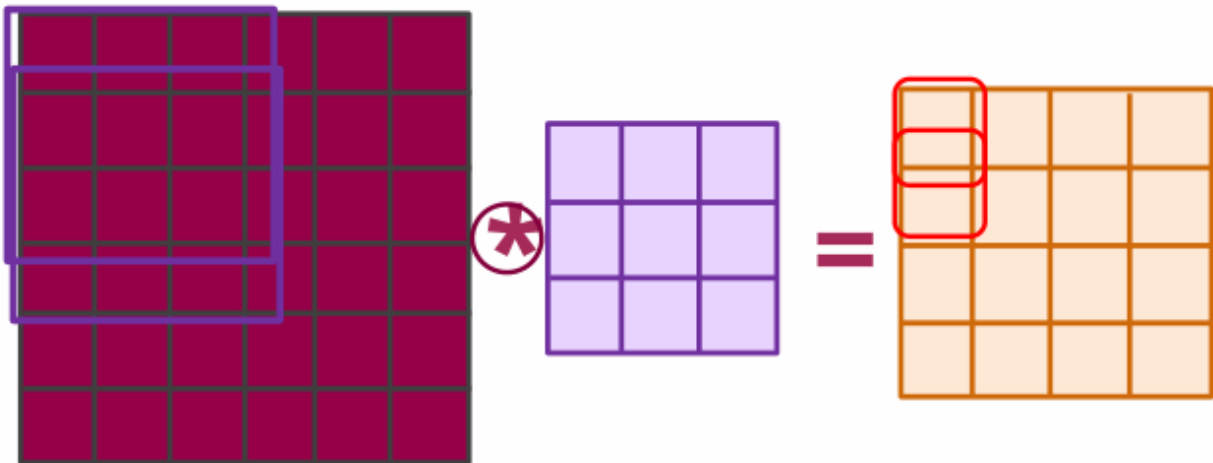


Figure 30: Convolution Operation Denoted by \*

### 9.1 Image Filtering for Noise Reduction

We typically use **smoothing** to remove *high-frequency noise* without unduly damaging the larger *low-frequency* objects of interest. Commonly used smoothing filters include:

- **Blur:** averages a pixel and its neighbours.
- **Median:** replaces a pixel with the median (rather than the mean) of the pixel and its neighbours.

- **Gaussian:** a filter that produces a smooth response (unlike blur/“box” & median filtering) by weighting more towards the centre.

A typical “classical” (pre-deep learning) computer vision pipeline consists of the following steps:

1. Clean-up / Pre-processing:

- Reduce noise (smoothing kernels).
- Remove geometric/radiometric distortion.
- Emphasise desired aspects of the image, e.g., edges, corners, blobs, etc. (differentiating kernels, feature detectors).

2. Segmentation:

- Identify / extract objects of interest.
- Sometimes the entire image is of interest, so the task is to separate it into non-overlapping regions.
- Most likely leverages domain-specific knowledge.
- Not always needed in deep learning based approaches.

3. Measurement:

- Quantify appropriate measurements on segmented objects.
- Might not be needed in deep learning based approaches.

4. Classification:

- Assign segmented objects to classes.
- Make decision etc.

## 9.2 Image Filtering for Edge Detection

Consider a horizontal slice across the image: **edge detection** filters are essentially performing a differentiation of the grey level with respect to distance, i.e., “how different is a pixel to its neighbours?”. Some filters are akin to *first derivatives* while others are more akin to *second derivatives*.

**Edge detection** is a common early step in image segmentation (often preceded by noise reduction). Edge detection determines how different pixels are from their neighbours: abrupt changes in brightness are interpreted as the edges of objects. Differentiating kernels can represent **first order** or **second order** derivatives. Differentiating kernels for edge detection can also be classified as **gradient magnitude** or **gradient direction**.

### 9.2.1 First Order Derivatives

The general image processing pipeline is as follows:

Smoothing (to reduce noise) → Derivative (so that noise is not accentuated)

Most differentiating kernels are built by combining these two operations. First order derivatives include:

- 1D Gaussian derivative kernels.
- 2D Gaussian derivative kernels: image gradients

$$\nabla f(x, y) = \left[ \frac{\partial f}{\partial x} \frac{\partial f}{\partial y} \right]^T$$

In image processing, an image can be represented as a 2D function  $I(x, y)$  where  $x$  &  $y$  are spatial co-ordinates and  $I$  is the pixel intensity at those co-ordinates. The first-order derivatives in the  $x$ - &  $y$ -directions are defined as:

$$\frac{\partial I}{\partial x} \text{ and } \frac{\partial I}{\partial y}$$

These derivatives measure the rate of change of intensity in the horizontal & vertical directions respectively. The concept is the same as differentiating a function, but applied to discrete pixel values, usually using finite differences or convolution with derivative filters.

The **Prewitt operator** is the simplest 2D differentiating kernel. It is obtained by convolving a 1D Gaussian derivative kernel with a 1D box filter in the orthogonal direction. It is used to estimate the gradient of an image's intensity by highlighting regions with high spatial intensity variation, making it useful for detecting edges & boundaries. It uses two  $3 \times 3$  convolution kernels to approximate the first-order derivatives of the image in the horizontal & vertical directions.

$$\begin{aligned} \text{Prewitt}_x &= \frac{1}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \otimes \frac{1}{2} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} = \frac{1}{6} \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \\ \text{Prewitt}_y &= \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \otimes \frac{1}{2} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} = \frac{1}{6} \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \end{aligned}$$

The **Sobel operator** is more robust than the Prewitt operator as it uses the Gaussian  $\sigma^2 = 0.5$  for the smoothing kernel:

$$\begin{aligned} \text{Sobel}_x &= \text{gauss}_{0.5}(y) \otimes \text{gauss}_{0.5}(x) = \frac{1}{4} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \otimes \frac{1}{2} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} = \frac{1}{8} \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & 1 \end{bmatrix} \\ \text{Sobel}_y &= \text{gauss}_{0.5}(x) \otimes \text{gauss}_{0.5}(y) = \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \otimes \frac{1}{2} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} = \frac{1}{8} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \end{aligned}$$

The **Scharr operator** is similar to the Sobel operator but with a smaller variance  $\sigma^2 = 0.375$  in the smoothing kernel.



### Magnitude Images using First Order Derivatives

Calculate “magnitude images” of the directional image gradients in the following image:



Figure 31: Example Image



Figure 32: Partial Derivative in the  $x$  Direction



Figure 34: The Magnitude of the Gradient

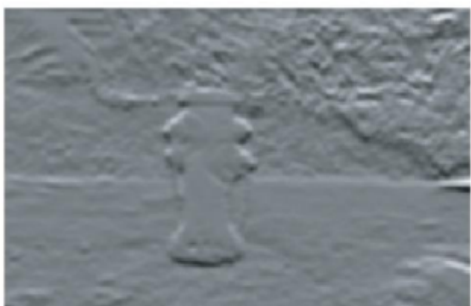


Figure 33: Partial Derivative in the  $y$  Direction



Figure 35: The Phase of the Gradient

The partial derivatives of the image in the  $x$  &  $y$  directions together form the two components of the gradient of the image.

### Edge Detection by Thresholding Magnitude Images

Calculate edges by thresholding “magnitude images” of the directional image gradients.

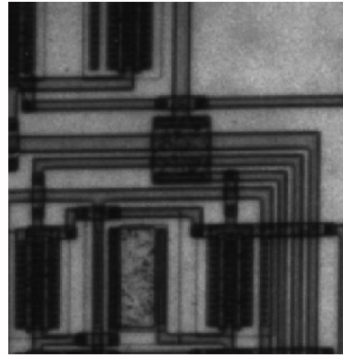


Figure 36: Input Greyscale Image

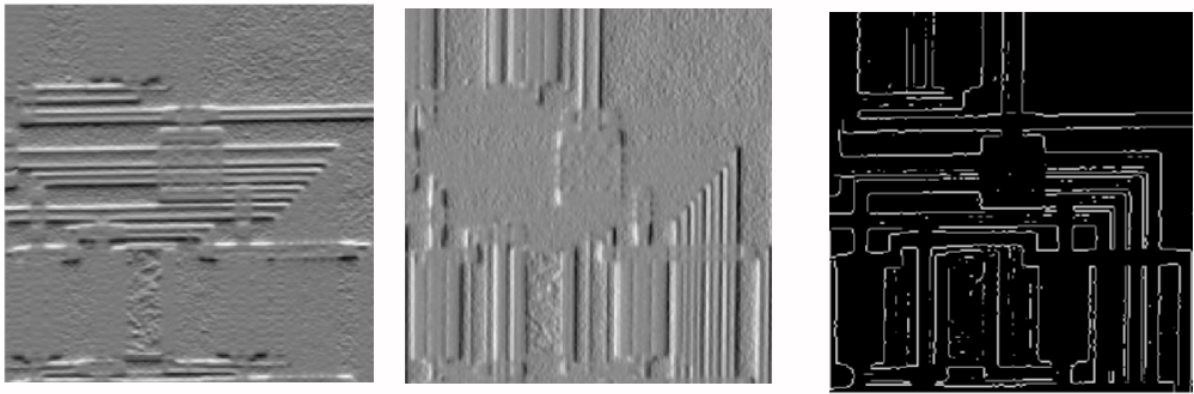


Figure 37: Prewitt Operator (Vertical Gradient, Horizontal Gradient, Thresholding Gradient Magnitude)

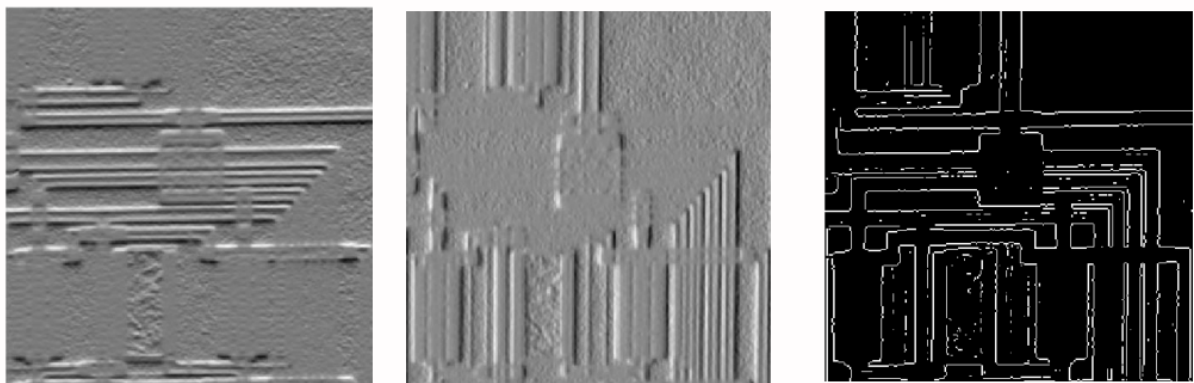


Figure 38: Sobel Operator (Vertical Gradient, Horizontal Gradient, Thresholding Gradient Magnitude)

#### 9.2.2 Second Order Derivatives

For a function (or image) of two variables, the **second order derivative** in the  $x$  &  $y$  directions can be obtained by convolving with the appropriately oriented **second-derivative kernel**. For a function or image  $I(x, y)$ , the second-order derivatives measure how the rate of change of the function changes, represented by  $\frac{\partial^2 I(x, y)}{\partial x^2}$  for the second derivative in the  $x$ -direction and  $\frac{\partial^2 I(x, y)}{\partial y^2}$  for the second derivative in the  $y$ -direction.

In image processing, second-order derivatives can be computed using specialised convolution kernels that act as second-derivative operators:

$$\frac{\partial^2 I(x, y)}{\partial x^2} = I(x, y) \otimes \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$$

$$\frac{\partial^2 I(x, y)}{\partial y^2} = I(x, y) \otimes \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$$

These kernels detect changes in intensity by convolving with the image. When applied, they help identify where the intensity values change significantly, highlighting potential edges or other features.

These kernels are second-order derivatives because they involve convolving the function with a differentiating kernel twice. This can be seen in 1D by convolving the function with the non-centralised difference operator, then convolving the result again with the same operator:

$$(f(x) \otimes \begin{bmatrix} 1 & -1 \end{bmatrix}) \otimes \begin{bmatrix} 1 & -1 \end{bmatrix} = f(x) \otimes (\begin{bmatrix} 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \end{bmatrix}) = f(x) \otimes \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$$

Note that these 2D derivatives are not isotropic or symmetric.

COME BACK TO LAPLACIAN OPERATOR

### 9.3 Image Filtering in the Frequency Domain

Any signal, discrete or continuous, periodic or non-periodic, can be represented as a sum of sinusoidal waves of different frequencies and phases which constitute the frequency domain representation of that signal.

## 10 Morphological Image Processing

The term **morphology** refers to shape. Morphological image processing assumes that an image consists of structures that can be handled by mathematical set theory. It is normally applied to binary (black & white) images, e.g. after applying thresholding. A **set** is a group of pixels; different set operations can be performed on this set of pixels. Applications for morphological image analysis generally involve image analysis at a very small scale (i.e., small regions of pixels) e.g., medical image processing, scientific image processing, industrial inspection, etc.

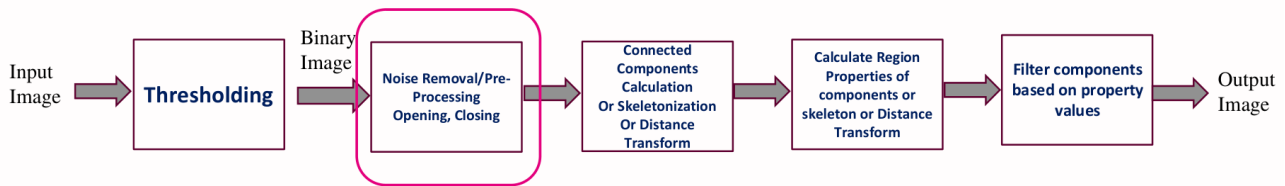


Figure 39: A General Morphological Image Processing Pipeline

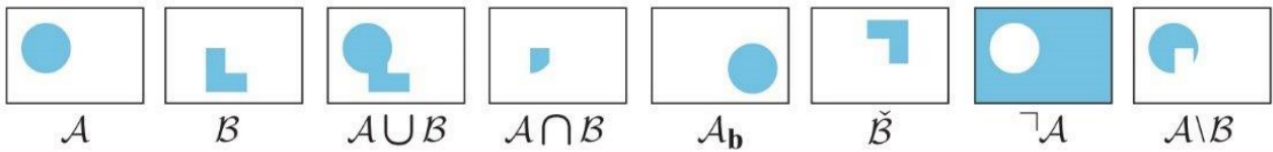


Figure 40: Basic Set Operators (From left to right: union, intersection, shift of  $A$  by some amount  $b$ , reflection about the origin (assumed to be at the center), complement, & set difference)

We can describe morphological differences in terms of intersections with test sets called **structuring elements (SEs)**. SEs are of a much smaller size than the original image. Morphological operations transform an image, i.e. changing a

pixel from black to white (or vice-versa) if a defined structuring element “fits” at that point. The shape + size of the structuring element directly affect the information about the image obtained by the operation.

### Structuring Elements Example

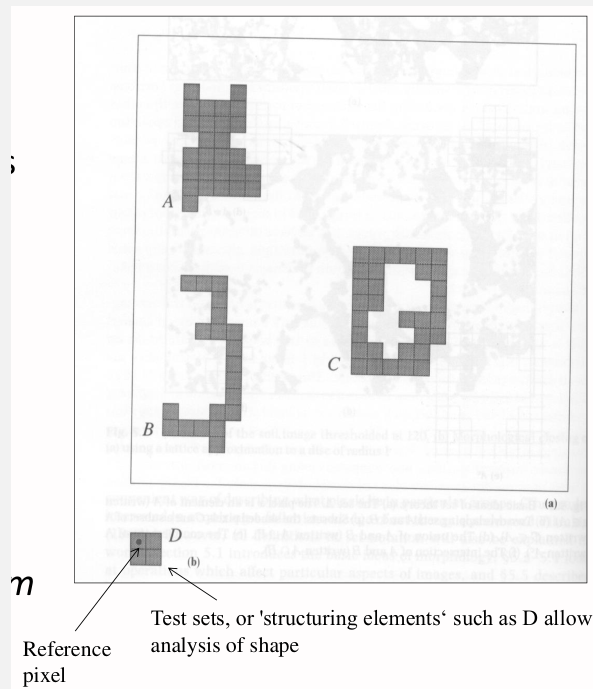


Figure 41: Structuring Elements Example

The three black objects *A*, *B*, & *C* are very similar in size (i.e., number of pixels) but different in morphology/shape. E.g., several possible translations of *D* will fit into *A*, but none will fit into *B*.

## 10.1 Basic Morphological Operations

Basic morphological operations are performed by set operations between a given image and a structuring element, e.g., erosion, dilation, opening, closing.