# CT404

## Graphics & Image Processing

Name: Andrew Hayes
E-mail: a.hayes18@universityofgalway.ie
Student ID: 21321503

2024–09–28

# Contents

# 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.
- `nazre.batool@universityofgalway.ie`
- Office Hours: Thursdays 16:00 – 17:00, CSB-2009.

- Dr. Waqar Shahid Qureshi.
- `waqarshahid.qureshi@universityofgalway.ie`.
- 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

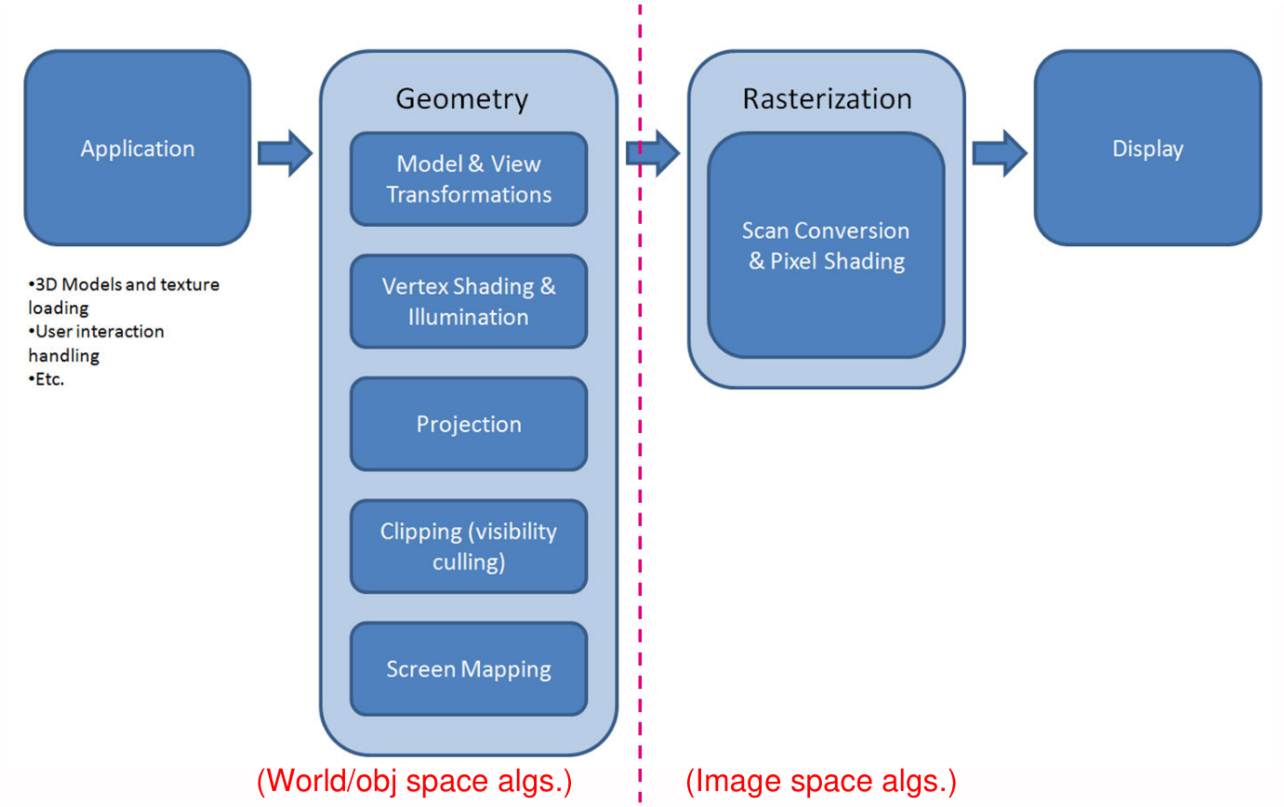


Figure 1: The Real-Time Graphics Pipeline

### 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 $(x', y')$.

$$x' = x + a$$

$$y' = y + b$$

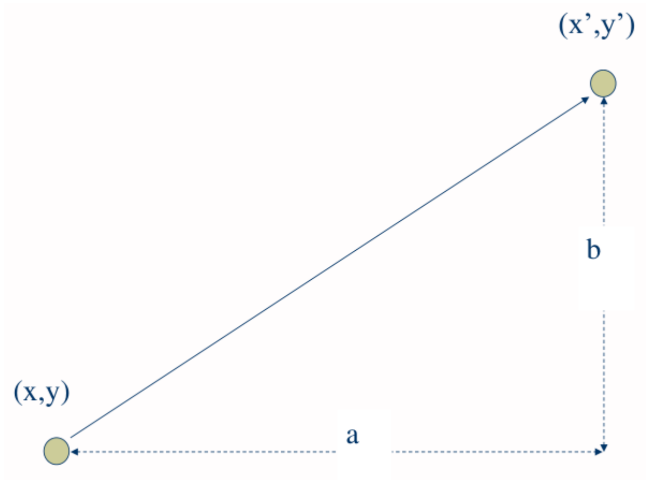


Figure 2: 2D Translation of a Point

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

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

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

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

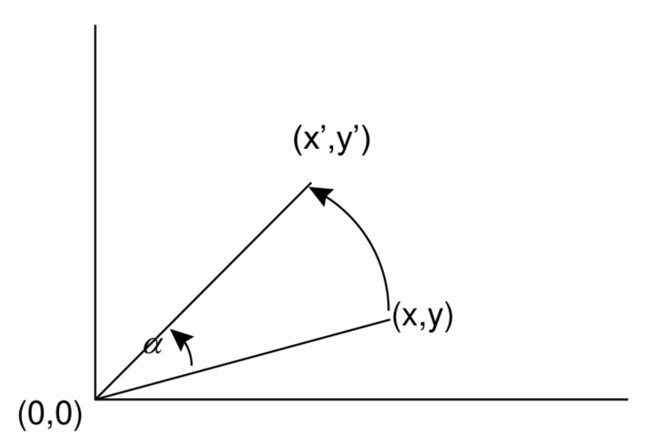


Figure 3: 2D Rotation of a Point

### 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.

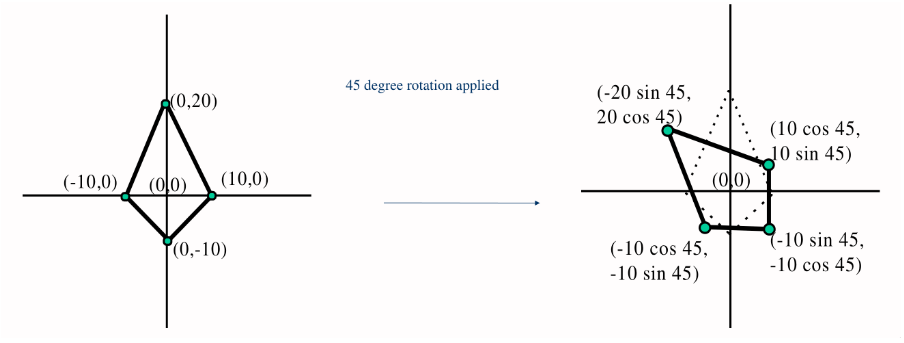


Figure 4: 2D Rotation of an Object

### 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.

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

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

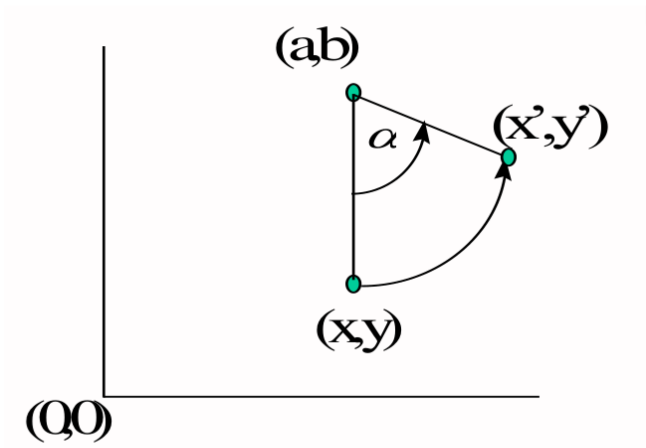


Figure 5: Arbitrary 2D Rotation

### 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

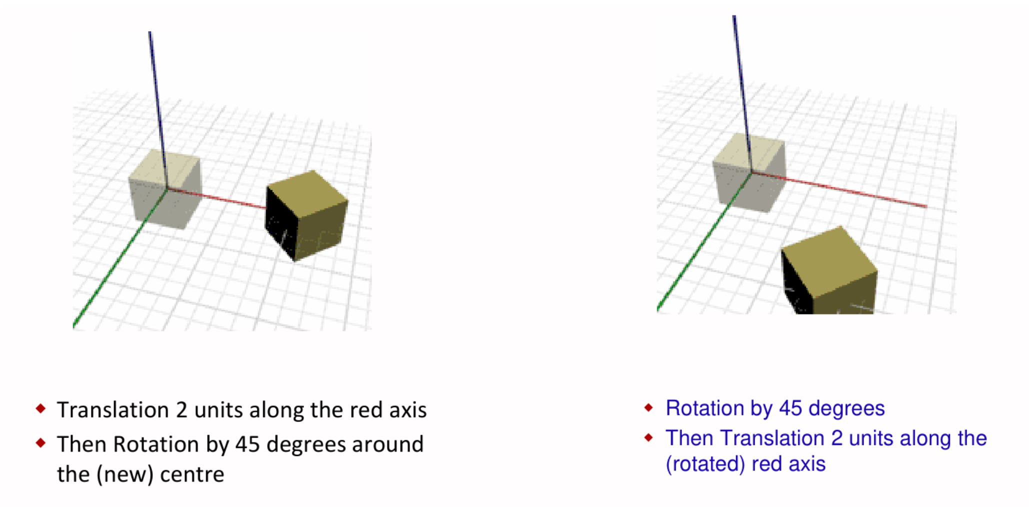


Figure 6: Order of Transformations

## 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.

```
<html>
    <head>
        <script>
            function draw() {
                var canvas = document.getElementById("canvas");
                var ctx = canvas.getContext("2d");
```

```
                ctx.fillStyle = "rgb(200,0,0)";
                ctx.fillRect (10, 10, 55, 50);
                ctx.fillStyle = "rgba(0, 0, 200, 0.5)";
                ctx.fillRect (30, 30, 55, 50);
            }
        </script>
    </head>
    <body onload="draw();">
        <canvas id="canvas" width="150" height="150"></canvas>
    </body>
</html>
```

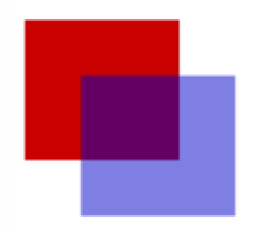

Figure 7: Rendering of the Above HTML Code

### 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.

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

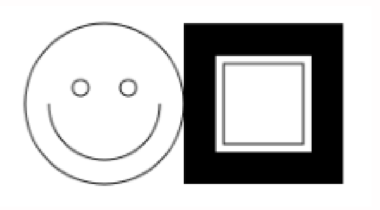

Figure 8: Rendering of the Above JavaScript Code

### 4.1.3 Canvas2D: `drawImage()`

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

```
function draw() {
    var ctx = document.getElementById('canvas').getContext('2d');
    var img = new Image();
    img.src = 'backdrop.png';
    img.onload = function(){
        ctx.drawImage(img,0,0);
        ctx.beginPath();
        ctx.moveTo(30,96);
        ctx.lineTo(70,66);
        ctx.lineTo(103,76);
        ctx.lineTo(170,15);
        ctx.stroke();
    }
}
```

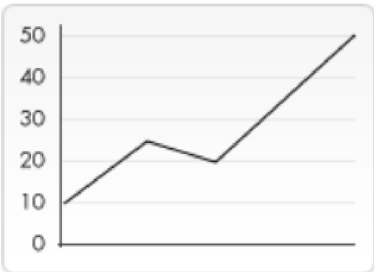


Figure 9: Rendering of the Above JavaScript Code

### 4.1.4 Canvas2D: Fill & Stroke Colours

```
<html>
    <head>
        <script>
            function draw() {
                var canvas = document.getElementById("canvas");
                var context = canvas.getContext('2d');
                // Filled Star
                context.lineWidth=3;
                context.fillStyle="#CC00FF";
                context.strokeStyle="#ffff00"; // NOT lineStyle!
                context.beginPath();
                context.moveTo(100,50);
                context.lineTo(175,200);
                context.lineTo(0,100);
                context.lineTo(200,100);
                context.lineTo(25,200);
                context.lineTo(100,50);
                context.fill(); // colour the interior
                context.stroke(); // draw the lines
            }
        </script>
```

```
    </head>
    <body onload="draw();">
        <canvas id="canvas" width="300" height="300"></canvas>
    </body>
</html>
```

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

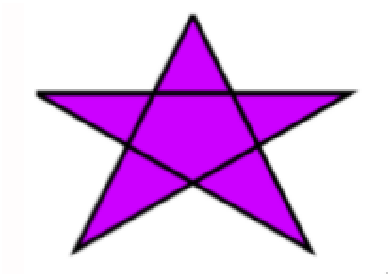

Figure 10: Rendering of the Above JavaScript Code

### 4.1.5 Canvas2D: Translations

```
<html>
    <head>
        <script>
            function draw() {
                var canvas = document.getElementById("canvas");
                var context = canvas.getContext('2d');
                context.save(); // save the default (root) co-ord system
                context.fillStyle="#CC00FF"; // purple
                context.fillRect(100,0,100,100);
                // translates from the origin, producing a nested co-ordinate system
                context.translate(75,50);
                context.fillStyle="#FFFF00"; // yellow
                context.fillRect(100,0,100,100);
                // transforms further, to produce another nested co-ordinate system
                context.translate(75,50);
                context.fillStyle="#0000FF"; // blue
                context.fillRect(100,0,100,100);
                context.restore(); // recover the default (root) co-ordinate system
                context.translate(-75,90);
                context.fillStyle="#00FF00"; // green
                context.fillRect(100,0,100,100);
            }
        </script>
    </head>
    <body onload="draw();">
        <canvas id="canvas" width="600" height="600"></canvas>
    </body>
</html>
```

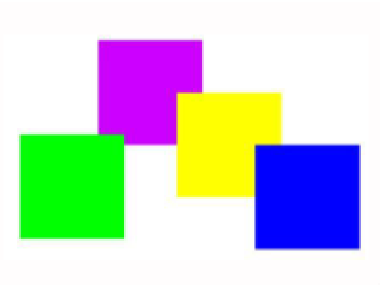

Figure 11: Rendering of the Above JavaScript Code

#### 4.1.6 Canvas2D: Order of Transformations

```
<html>
    <head>
        <script>
            function draw() {
                var canvas = document.getElementById("canvas");
                var context = canvas.getContext('2d');
                context.save(); // save the default (root) co-ord system
                context.fillStyle="#CC00FF"; // purple
                context.fillRect(0,0,100,100); // positioned with TL corner at 0,0
                // translate then rotate
                context.translate(100,0);
                context.rotate(Math.PI/3);
                context.fillStyle="#FF0000"; // red
                context.fillRect(0,0,100,100); // positioned with TL corner at 0,0
                // recover the root co-ord system
                context.restore();
                // rotate then translate
                context.rotate(Math.PI/3);
                context.translate(100,0);
                context.fillStyle="#FFFF00"; // yellow
                context.fillRect(0,0,100,100); // positioned with TL corner at 0,0
            }
        </script>
    </head>
    <body onload="draw();">
        <canvas id="canvas" width="600" height="600"></canvas>
    </body>
</html>
```

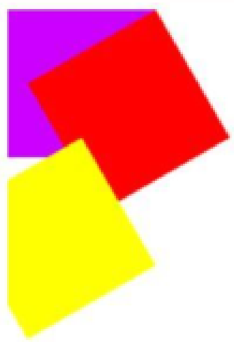

Figure 12: Rendering of the Above JavaScript Code

### 4.1.7 Scaling

```
<html>
    <head>
        <script>
            function draw() {
                var canvas = document.getElementById("canvas");
                var context = canvas.getContext('2d');
                context.fillStyle="#CC00FF"; // purple
                context.fillRect(0,0,100,100); // positioned with TL corner at 0,0
                context.translate(150,0);
                context.scale(2,1.5);
                context.fillStyle="#FF0000"; // red
                context.fillRect(0,0,100,100); // positioned with TL corner at 0,0
            }
        </script>
    </head>
    <body onload="draw();">
        <canvas id="canvas" width="600" height="600"></canvas>
    </body>
</html>
```

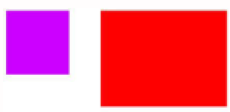

Figure 13: Rendering of the Above JavaScript Code

### 4.1.8 Canvas2D: Programmatic Graphics

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

```
                    context.fillRect(0,0,100,100);
                    context.rotate(2*Math.PI/15);
                }
            }
        </script>
    </head>
    <body onload="draw();">
        <canvas id="canvas" width="600" height="600"></canvas>
    </body>
</html>
```

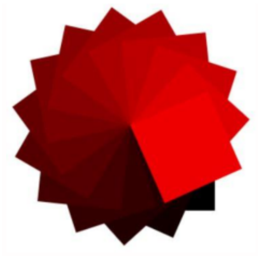

Figure 14: Rendering of the Above JavaScript Code

# 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 $\begin{bmatrix} x & y & z \end{bmatrix}$ by the rotation matrix:

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} R_x$$

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°).

# 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 s 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.

```
<html>
 <head>

  <script src="three.js"></script>
  <script>
    'use strict'

    function draw() {
      // create renderer attached to HTML Canvas object
      var c = document.getElementById("canvas");
      var renderer = new THREE.WebGLRenderer({ canvas: c, antialias: true });

```

```
      // create the scenegraph
      var scene = new THREE.Scene();

      // create a camera
      var fov = 75;
      var aspect = 600/600;
      var near = 0.1;
      var far = 1000;
      var camera = new THREE.PerspectiveCamera( fov, aspect, near, far );
      camera.position.z = 100;

      // add a light to the scene
      var light = new THREE.PointLight(0xFFFF00);
      light.position.set(10, 30, 25);
      scene.add(light);

      // add a cube to the scene
      var geometry = new THREE.BoxGeometry(20, 20, 20);
      var material = new THREE.MeshLambertMaterial({color: 0xfd59d7});
      var cube = new THREE.Mesh(geometry, material);
      scene.add(cube);

      // render the scene as seen by the camera
      renderer.render(scene, camera);
    }
  </script>
 </head>

 <body onload="draw();">
   <canvas id="canvas" width="600" height="600"></canvas>
 </body>
</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`.

```
<html>
 <head>

  <script src="three.js"></script>
  <script>
    'use strict'

    var scene;

    function addGeometryAtPosition(geometry, x, y, z) {
      var material = new THREE.MeshLambertMaterial({color: 0xffffff});
      var mesh = new THREE.Mesh(geometry, material);
```

```
    scene.add(mesh);
    mesh.position.set(x,y,z);
  }

  function draw() {
    // create renderer attached to HTML Canvas object
    var c = document.getElementById("canvas");
    var renderer = new THREE.WebGLRenderer({ canvas: c, antialias: true });

    // create the scenegraph (global variable)
    scene = new THREE.Scene();

    // create a camera
    var fov = 75;
    var aspect = 400/600;
    var near = 0.1;
    var far = 1000;
    var camera = new THREE.PerspectiveCamera( fov, aspect, near, far );
    camera.position.z = 100;

    // add a light to the scene
    var light = new THREE.PointLight(0xFFFF00);
    light.position.set(10, 0, 25);
    scene.add(light);

    // add a bunch of sample primitives to the scene
    // see more here:  https://threejsfundamentals.org/threejs/lessons/threejs-primitives.html

    // args: width, height, depth
    addGeometryAtPosition(new THREE.BoxGeometry(6,4,8), -50, 0, 0);

    // args: radius, segments
    addGeometryAtPosition(new THREE.CircleBufferGeometry(7, 24), -30, 0, 0);

    // args: radius, height, segments
    addGeometryAtPosition(new THREE.ConeBufferGeometry(6, 4, 24), -10, 0, 0);

    // args: radiusTop, radiusBottom, height, radialSegments
    addGeometryAtPosition(new THREE.CylinderBufferGeometry(4, 4, 8, 12), 20, 0, 0);

    // arg: radius
    // Polyhedrons
    // (Dodecahedron is a 12-sided polyhedron, Icosahedron is 20-sided, Octahedron is 8-sided,
    ↪  Tetrahedron is 4-sided)
    addGeometryAtPosition(new THREE.DodecahedronBufferGeometry(7), 40, 0, 0);
    addGeometryAtPosition(new THREE.IcosahedronBufferGeometry(7), -50, 20, 0);
    addGeometryAtPosition(new THREE.OctahedronBufferGeometry(7), -30, 20, 0);
    addGeometryAtPosition(new THREE.TetrahedronBufferGeometry(7), -10, 20, 0);

    // args: radius, widthSegments, heightSegments
    addGeometryAtPosition(new THREE.SphereBufferGeometry(7,12,8), 20, 20, 0);

    // args: radius, tubeRadius, radialSegments, tubularSegments
```

```
      addGeometryAtPosition(new THREE.TorusBufferGeometry(5,2,8,24), 40, 20, 0);

      // render the scene as seen by the camera
      renderer.render(scene, camera);
    }
  </script>
 </head>

 <body onload="draw();">
   <canvas id="canvas" width="600" height="600"></canvas>
 </body>
</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, gay~~ $x$ rotation, $y$ rotation, $z$ rotation).
- The **viewing frustum** (the Field of View (FoV) & clipping planes).

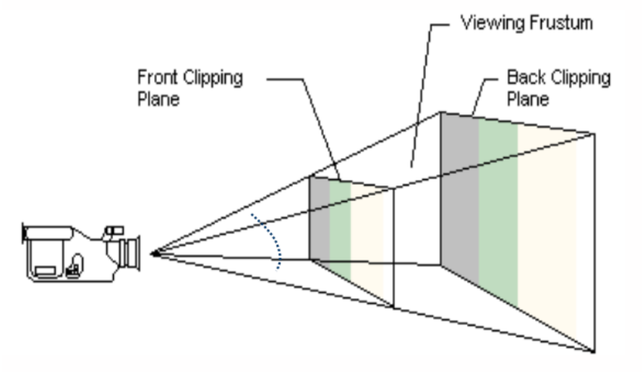


Figure 15: The Viewing Frustum

In Three.js, the FoV can be set differently in the vertical and horizontal directions via the first & second arguments to the constructor 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).

```
<html>
  <head>
  <script src="three.js"></script>
  <script>
    'use strict'

    function draw() {
      // create renderer attached to HTML Canvas object
      var c = document.getElementById("canvas");
      var renderer = new THREE.WebGLRenderer({ canvas: c, antialias: true });

      // create the scenegraph
      var scene = new THREE.Scene();

      // create a camera
      var fov = 75;
      var aspect = 600/600;
      var near = 0.1;
      var far = 1000;
      var camera = new THREE.PerspectiveCamera( fov, aspect, near, far );
      camera.position.set(0, 10, 30);

      // add a light to the scene
      var light = new THREE.PointLight(0xFFFFFF);
      light.position.set(0, 10, 30);
      scene.add(light);

      // add a cylinder
      // args: radiusTop, radiusBottom, height, radialSegments
      var cyl = new THREE.Mesh(
        new THREE.CylinderBufferGeometry(1, 1, 10, 12),
        new THREE.MeshLambertMaterial({color: 0xAAAAAA}) );
      scene.add(cyl);

      // clone the cylinder
      var cyl2 = cyl.clone();

      // modify its rotation by 60 degrees around its z axis
      cyl2.rotateOnAxis(new THREE.Vector3(0,0,1), Math.PI/3);
      scene.add(cyl2);
      // clone the cylinder again
      var cyl3 = cyl.clone();
      scene.add(cyl3);
      // set its rotation directly using "Euler angles", to 120 degrees on z axis
      cyl3.rotation.set(0,0,2*Math.PI/3);

      // render the scene as seen by the camera
      renderer.render(scene, camera);
```

```
    }
  </script>
 </head>

 <body onload="draw();">
   <canvas id="canvas" width="600" height="600"></canvas>
 </body>
</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.

```
<html>
 <head>

  <script src="three.js"></script>
  <script>
    'use strict'

    function draw() {
      // create renderer attached to HTML Canvas object
      var c = document.getElementById("canvas");
      var renderer = new THREE.WebGLRenderer({ canvas: c, antialias: true });

      // create the scenegraph
      var scene = new THREE.Scene();

      // create a camera
      var fov = 75;
      var aspect = 600/600;
      var near = 0.1;
      var far = 1000;
      var camera = new THREE.PerspectiveCamera( fov, aspect, near, far );
      camera.position.set(0, 1.5, 6);

      // add a light to the scene
      var light = new THREE.PointLight(0xFFFFFF);
      light.position.set(0, 10, 30);
      scene.add(light);

      // desk lamp base
      // args: radiusTop, radiusBottom, height, radialSegments
      var base = new THREE.Mesh(
        new THREE.CylinderBufferGeometry(1, 1, 0.1, 12),
        new THREE.MeshLambertMaterial({color: 0xAAAAAA}) );
      scene.add(base);
```

```

      // desk lamp first arm piece
      var arm = new THREE.Mesh(
        new THREE.CylinderBufferGeometry(0.1, 0.1, 3, 12),
        new THREE.MeshLambertMaterial({color: 0xAAAAAA}) );

      // since we want to rotate around a point other than the arm's centre,
      // we can create a pivot point as the parent of the arm, position the
      // arm relative to that pivot point, and apply rotation on the pivot point
      var pivot = new THREE.Object3D();
      // centre of rotation we want
      // (in world coordinates, since pivot is not yet a child of the base)
      pivot.position.set(0, 0, 0);
      pivot.add(arm); // pivot is parent of arm
      base.add(pivot); // base is parent of pivot

      //  translate arm relative to its parent, i.e. 'pivot'
      arm.position.set(0, 1.5, 0);
      //  rotate pivot point relative to its parent, i.e. 'base'
      pivot.rotateOnAxis(new THREE.Vector3(0,0,1), -Math.PI/6);

      // clone a second arm piece (consisting of a pivot with a cylinder as its child)
      var pivot2 = pivot.clone();
      // add as a child of the 1st pivot
      pivot.add(pivot2);
      // rotate the 2nd pivot relative to the 1st pivot (since it's nested)
      pivot2.rotation.z = Math.PI/3;
      // translate the 2nd pivot relative to the 1st pivot
      pivot2.position.set(0,3,0);

      // TEST: we can rotate the 1st arm piece and the 2nd arm piece should stay correct
      pivot.rotateOnAxis(new THREE.Vector3(0,0,1), Math.PI/12);

      // TEST: we can also move the base, and everything stays correct
      base.position.x -= 3;

      // render the scene as seen by the camera
      renderer.render(scene, camera);
    }
  </script>
 </head>

 <body onload="draw();">
   <canvas id="canvas" width="600" height="600"></canvas>
 </body>
</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.

```
<html>

<head>
    <script>
        function attachEvents() {
            document.onkeypress = function (event) {
                var xoffset = 10 * parseInt(String.fromCharCode(event.keyCode || event.charCode));
                draw(xoffset);
            }
        }

        function draw(xoffset) {
            var canvas = document.getElementById("canvas");
            var context = canvas.getContext('2d');

            // remove previous translation if any
            context.save();

            // over-write previous content, with a white rectangle
            context.fillStyle = "#FFFFFF";
            context.fillRect(0, 0, 300, 300);

            // translate based on numerical keypress
            context.translate(xoffset, 0);

            // purple rectangle
            context.fillStyle = "#CC00FF";
            context.fillRect(0, 0, 50, 50);
            context.restore();
        }
    </script>
</head>

<body onload="attachEvents();">
    <canvas id="canvas" width="300" height="300"></canvas>
</body>

</html>
```

Listing 5: Keyboard Handling (Canvas/JavaScript)

## 7.2 Mouse Handling

```
<html>

<head>
    <script>
        var isMouseDown = false;
        function attachEvents() {
            document.onmousedown = function (event) {
                isMouseDown = true;
                draw(event.clientX, event.clientY);
            }
            document.onmouseup = function (event) {
                isMouseDown = false;
            }
            document.onmousemove = function (event) {
                if (isMouseDown) {
                    draw(event.clientX, event.clientY);
                }
            }
        }
        function draw(xoffset, yoffset) {

            var canvas = document.getElementById("canvas");
            var context = canvas.getContext('2d');

            // remove previous translation if any
            context.save();
            // over-write previous content, with a grey rectangle
            context.fillStyle = "#DDDDDD";
            context.fillRect(0, 0, 600, 600);
            // translate based on position of mouseclick
            context.translate(xoffset, yoffset);
            // purple rectangle
            context.fillStyle = "#CC00FF";
            context.fillRect(-25, -25, 50, 50); // centred on coord system
            context.restore();

        }
    </script>
</head>

<body onload="attachEvents(); draw(0,0);">
    hello<br>
    <div id='canvasdiv' style='position:absolute; left:0px; top:0px;'><canvas id="canvas" width="600"
            height="600"></canvas></div>
</body>

</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.

```
<html>

<head>
    <script>

        var x = 0, y = 0;
        var dx = 4, dy = 5;

        function draw() {

            var canvas = document.getElementById("canvas");
            var context = canvas.getContext('2d');

            // remove previous translation if any
            context.save();
            // over-write previous content, with a grey rectangle
            context.fillStyle = "#DDDDDD";
            context.fillRect(0, 0, 600, 600);
            // perform movement, and translate to position
            x += dx;
            y += dy;
            if (x <= 0)
                dx = 4;
            else if (x >= 550)
                dx = -4;
            if (y <= 0)
                dy = 5;
            else if (y >= 550)
                dy = -5;
            context.translate(x, y);
            // purple rectangle
            context.fillStyle = "#CC00FF";
            context.fillRect(0, 0, 50, 50);
            context.restore();

            // do it all again in 1/30th of a second
            window.setTimeout("draw();", 1000 / 30);
        }
    </script>
</head>

<body onload="draw();">
    <canvas id="canvas" width="600" height="600"></canvas>
</body>

</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.

```
<html>
<head>
    <script>
        var x = 0, y = 0;
        var dx = 4, dy = 5;
        var now = Date.now();

        function draw() {
            // do it all again in 1/60th of a second
            window.requestAnimationFrame(draw);

            var elapsedMs = Date.now() - now;
            now = Date.now();

            var canvas = document.getElementById("canvas");
            var context = canvas.getContext('2d');

            // remove previous translation if any
            context.save();

            // over-write previous content, with a grey rectangle
            context.fillStyle = "#DDDDDD";
            context.fillRect(0, 0, 600, 600);

            // perform movement, and translate to position
            x += dx * elapsedMs / 16.7;
            y += dy * elapsedMs / 16.7;

            if (x <= 0)
                dx = 4;
            else if (x >= 550)
                dx = -4;
            if (y <= 0)
                dy = 5;
            else if (y >= 550)
                dy = -5;

            context.translate(x, y);

            // purple rectangle
            context.fillStyle = "#CC00FF";
            context.fillRect(0, 0, 50, 50);
            context.restore();
        }
    </script>
</head>
<body onload="draw();">
    <canvas id="canvas" width="600" height="600"></canvas>
</body>
</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.

```
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.

```
<html>

<head>

    <script src="../../week2/examples/three.js"></script>
    <script>

        'use strict'

        var raycaster, renderer, scene, camera;
        var selectedObject = null;
        var selectableObjects = [];
        var lastMousePos = {x: 0, y: 0};

        function draw() {
            // create renderer attached to HTML Canvas object
            var c = document.getElementById("canvas");
            renderer = new THREE.WebGLRenderer({canvas: c, antialias: true});

            // create the scenegraph
            scene = new THREE.Scene();

            // create a camera
            var fov = 75;
            var aspect = 600 / 600;
            var near = 0.1;
            var far = 1000;
            camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
            camera.position.set(-5, 1.5, 6);

            // add a light to the scene
            var light = new THREE.PointLight(0xFFFFFF);
            light.position.set(0, 10, 30);
            scene.add(light);

            // desk lamp base
            // args: radiusTop, radiusBottom, height, radialSegments
            var base = new THREE.Mesh(
                new THREE.CylinderBufferGeometry(1, 1, 0.1, 12),
                new THREE.MeshLambertMaterial({color: 0xAAAAAA}));
            scene.add(base);
            base.position.set(-5, -2, 0);
            selectableObjects.push(base);
```

```
        base.canTranslate = true; // I added this property

        // desk lamp first arm piece
        var arm = new THREE.Mesh(
            new THREE.CylinderBufferGeometry(0.1, 0.1, 3, 12),
            new THREE.MeshLambertMaterial({color: 0xAAAAAA}));

        // since we want to rotate around a point other than the arm's centre,
        // we can create a pivot point as the parent of the arm, position the
        // arm relative to that pivot point, and apply rotation on the pivot point
        var pivot = new THREE.Object3D();
        pivot.position.set(0, 0, 0); // centre of rotation we want
        pivot.add(arm); // pivot is parent of arm
        base.add(pivot); // base is parent of pivot
        selectableObjects.push(arm);
        arm.canRotate = true; // I added this property

        //    translate arm relative to pivot point
        arm.position.set(0, 1.5, 0);
        //    rotate pivot point relative to the world
        pivot.rotateOnAxis(new THREE.Vector3(0, 0, 1), -Math.PI / 6);

        // second arm piece (consisting of a pivot with a cylinder as its child)
        var pivot2 = pivot.clone();
        pivot.add(pivot2);
        // rotate the 2nd pivot relative to the 1st pivot (since it's nested)
        pivot2.rotation.z = Math.PI / 3;
        // translate the 2nd pivot relative to the 1st pivot
        pivot2.position.set(0, 3, 0);
        var arm2 = pivot2.children[0];
        selectableObjects.push(arm2);
        arm2.canRotate = true; // I added this property

        // args: radius, height, segments
        var lampshade = new THREE.Mesh(
            new THREE.ConeBufferGeometry(1, 0.7, 24),
            new THREE.MeshLambertMaterial({color: 0xAAAAAA})
        );
        var shadePivot = new THREE.Object3D();
        pivot2.add(shadePivot); // lampshade pivot is a child of the 2nd arm pivot
        shadePivot.add(lampshade);
        shadePivot.position.set(0, 3, 0);
        shadePivot.rotation.x = Math.PI;
        selectableObjects.push(lampshade);
        lampshade.canRotate = true; // I added this property

        raycaster = new THREE.Raycaster();

        c.onmousedown = handleMouseDown;
        c.onmousemove = handleMouseMove;
        c.onmouseup = function (e) {
            selectedObject = null;
        };
```

```

            animate();
        }

        function animate() {
            setTimeout(animate, 1000 / 60);

            // render the scene as seen by the camera
            renderer.render(scene, camera);
        }

        function handleMouseDown(e) {
            // handle mouse-clicks on the canvas
            // did the user click a mesh?
            /* note that 0,0 is the centre of the canvas according to WebGL,
                and the canvas extends from (-1,-1) to (1,1)
                but 0,0 is the top-left of the canvas according to e.clientX,e.clientY,
                and the canvas extends from (0,0) to (599,599)
            */
            var x = 2 * (e.clientX - 300) / 600;
            var y = -2 * ((e.clientY - 300) / 600);

            lastMousePos.x = x;
            lastMousePos.y = y;

            // set up and apply the raycaster (we are returned an array of intersection objects)
            raycaster.setFromCamera({x: x, y: y}, camera);
            var intersects = raycaster.intersectObjects(selectableObjects);
            if (intersects.length > 0) {
                var closestObj, closestDist;

                for (var i = 0; i < intersects.length; i++) {
                    /*
                            An intersection has the following properties :
                                    - object : intersected object (THREE.Mesh)
                                    - distance : distance from ray start to intersection (number)
                                    - face : intersected face (THREE.Face3)
                                    - faceIndex : intersected face index (number)
                                    - point : intersection point (THREE.Vector3)
                                    - uv : intersection point in the object's UV coordinates
                                    ↪  (THREE.Vector2)
                    */

                    if (i == 0 || intersects[i].distance < closestDist) {
                        closestObj = intersects[i].object;
                        closestDist = intersects[i].distance;
                    }
                }

                selectedObject = closestObj;
            }
            else
                selectedObject = null;
```

```
        }

        function handleMouseMove(e) {
            if (selectedObject != null) {
                var x = 2 * (e.clientX - 300) / 600;
                var y = -2 * ((e.clientY - 300) / 600);
                // dx,dy is the amount the mouse just moved by in pixels
                var dx = x - lastMousePos.x;
                var dy = y - lastMousePos.y;

                if (selectedObject.canRotate) {
                    // rotate the parent ('pivot') that the object is a child of
                    selectedObject.parent.rotation.x += dx;
                    selectedObject.parent.rotation.z += dy;
                }
                else if (selectedObject.canTranslate) {
                    // translate the object
                    selectedObject.position.x += dx * 4;
                    selectedObject.position.z -= dy * 4;
                }

                lastMousePos.x = x;
                lastMousePos.y = y;
            }
        }
    </script>
</head>

<body onload="draw();">
    <!-- Note that the canvas has been positioned precisely at 0,0 so that mouse positions on the
    ↪ browser
    are the same as mouse positions on the canvas -->
    <canvas id="canvas" width="600" height="600" style="position:absolute; left:0px;
    ↪ top:0px"></canvas>
</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).

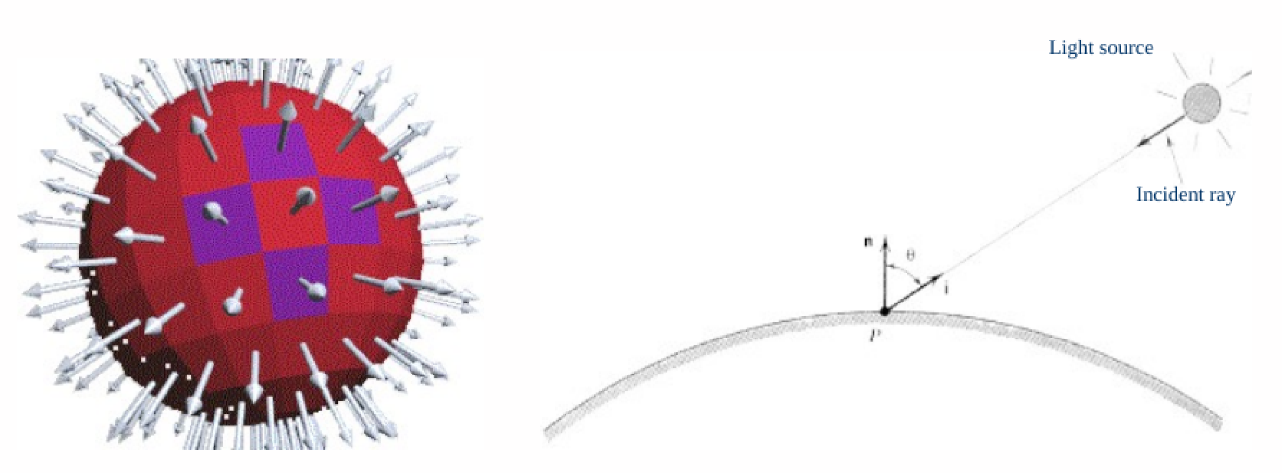


Figure 17: Flat Shading

#### 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.

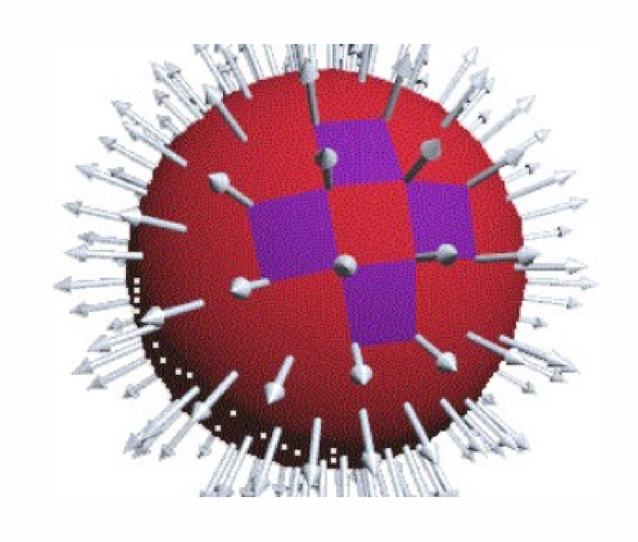

Figure 18: Smooth Shading

### 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.

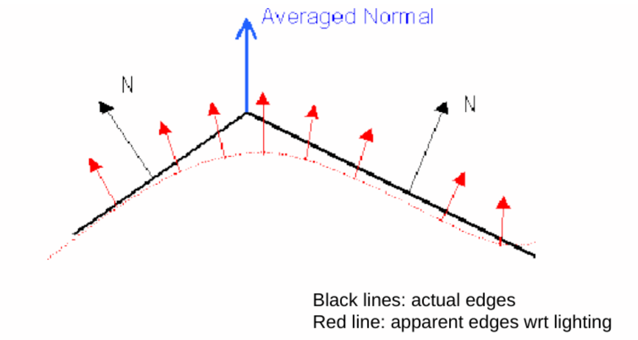


Figure 19: Normal Interpolating (Phong) Shading

## 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).