CT404

Graphics & Image Processing

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

2024–09–16

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

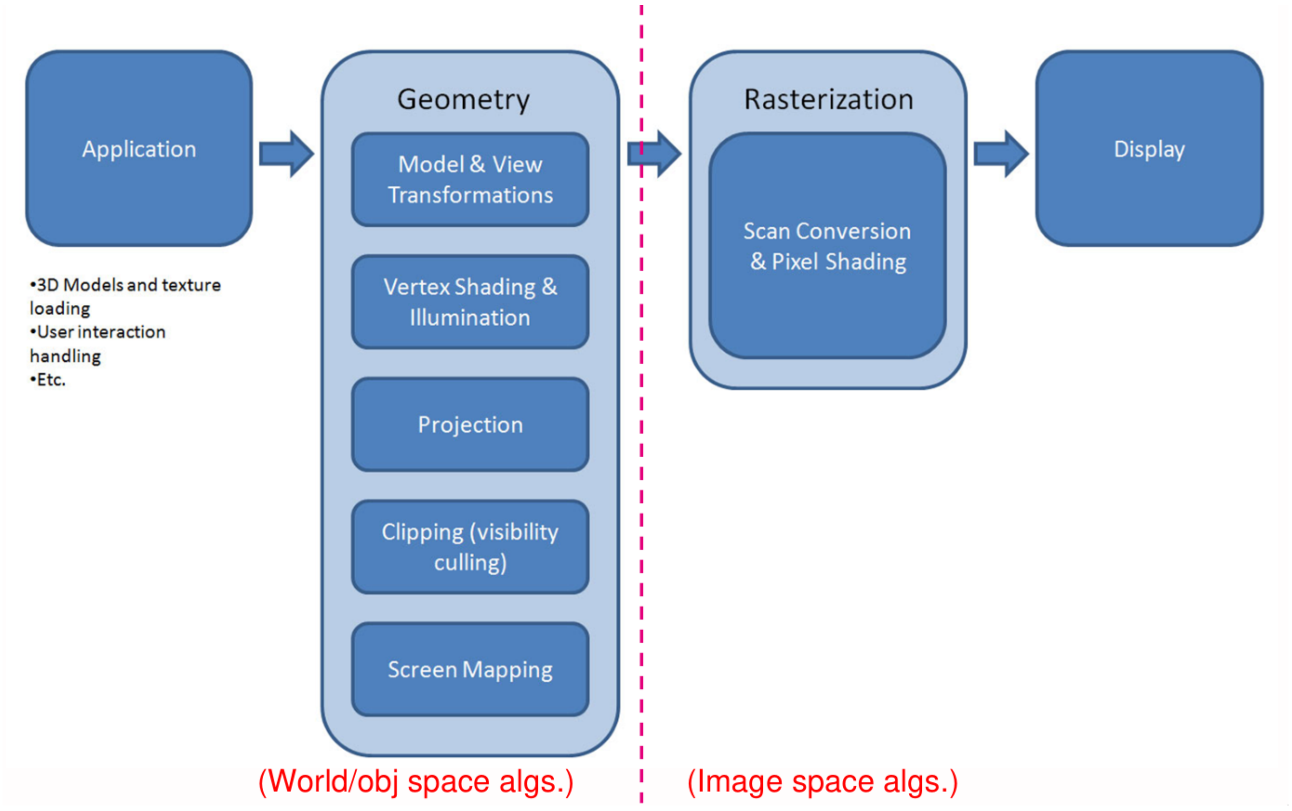

Application
•3D Models and texture loading
•User interaction handling
•Etc.
Geometry
Model & View Transformations
Vertex Shading & Illumination
Projection
Clipping (visibility culling)
Screen Mapping
Rasterization
Scan Conversion & Pixel Shading
Display
(World/obj space algs.)
(Image space algs.)

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').

$$x' = x + a$$

$$y' = y + b$$

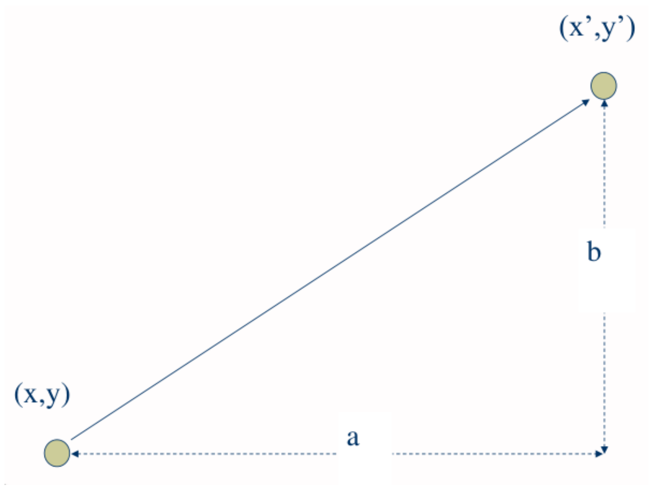


Figure 2: 2D Translation of a Point

3.1.2 2D Rotation of a *Point*

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

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

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

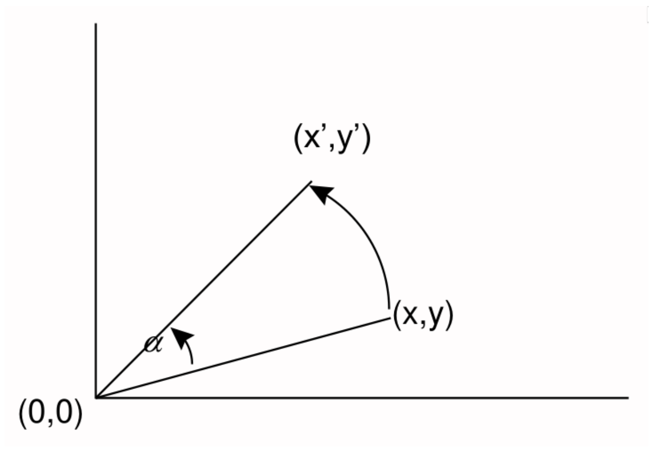


Figure 3: 2D Rotation of a Point

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.

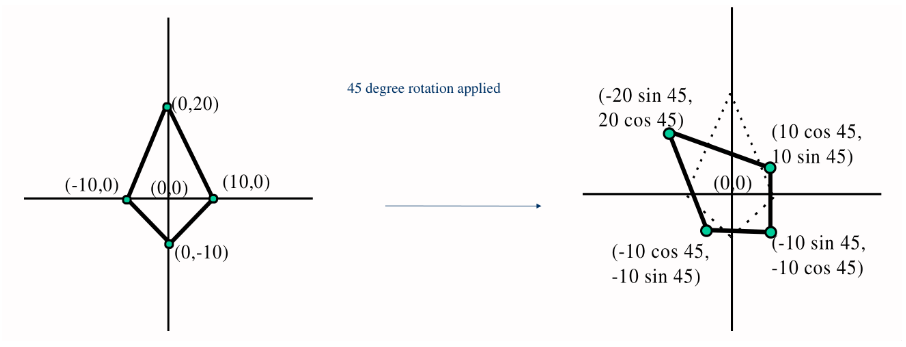


Figure 4: 2D Rotation of an Object

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.

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

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

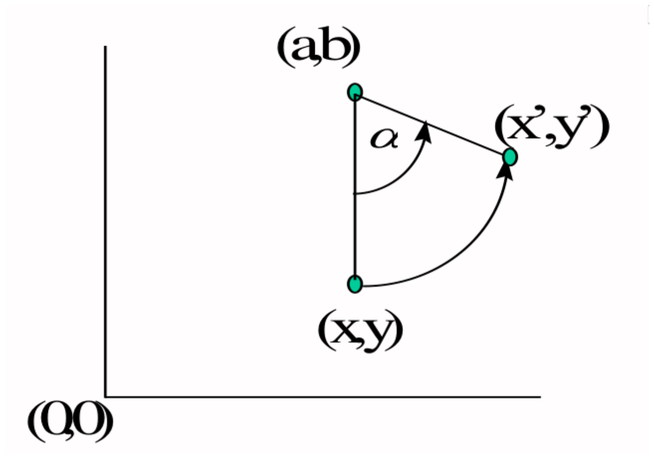


Figure 5: Arbitrary 2D Rotation

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

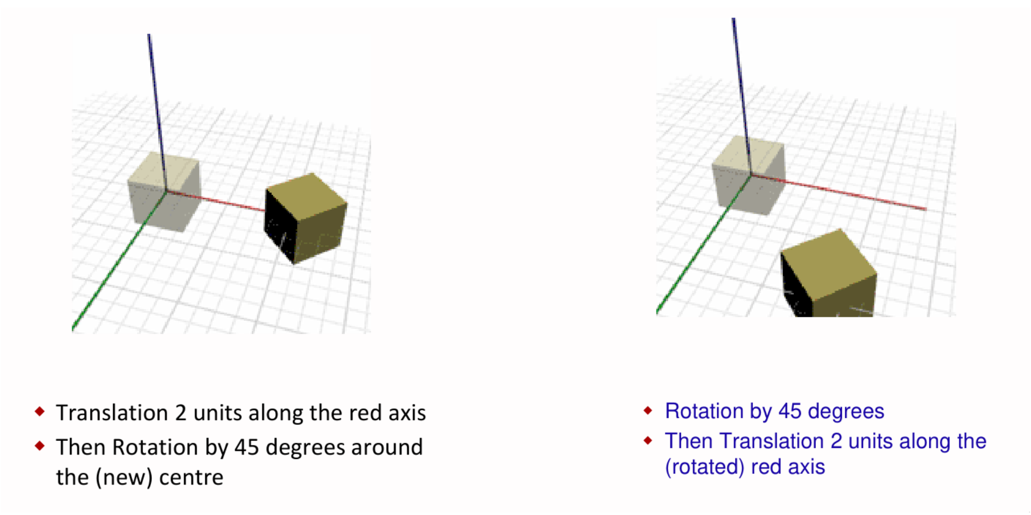
Scaling of an object is achieved by considering each of its vertices in turn, multiplying said vertex's x & y values by the scaling factor. A scaling factor of 2 will double the size of the object, while a scaling factor of 0.5 will halve it. It is possible to have different scaling factors for x & y,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



Translation 2 units along the red axis
Then Rotation by 45 degrees around the (new) centre
Rotation by 45 degrees
Then Translation 2 units along the (rotated) red axis

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>
```

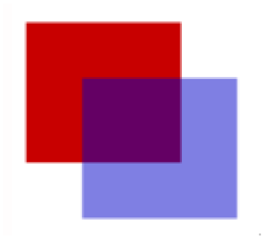


Figure 7: Rendering of the Above HTML Code

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..
}
```

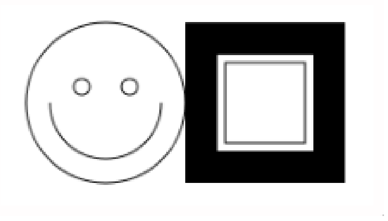


Figure 8: Rendering of the Above JavaScript Code

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();
    }
}
```

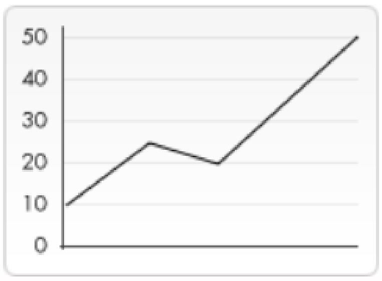


Figure 9: Rendering of the Above JavaScript Code

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`.

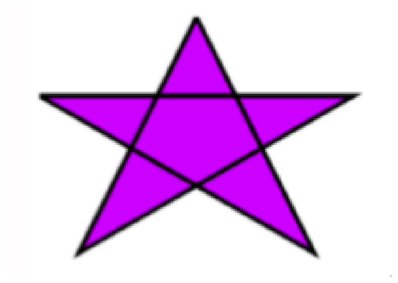


Figure 10: Rendering of the Above JavaScript Code

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>
```

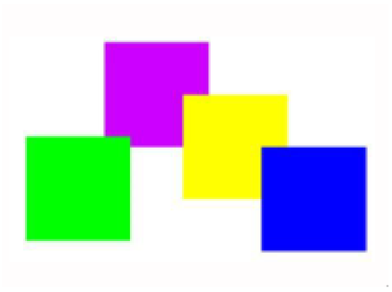


Figure 11: Rendering of the Above JavaScript Code

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>
```

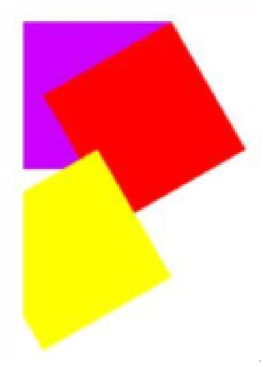


Figure 12: Rendering of the Above JavaScript Code

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>
```

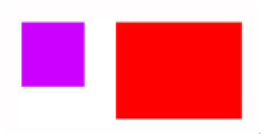


Figure 13: Rendering of the Above JavaScript Code

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>
```

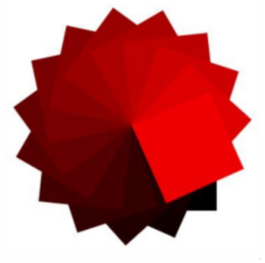


Figure 14: Rendering of the Above JavaScript Code