Modeling

RNTWARI@GMAIL.COM

Hierarchical Modeling

- A complex object can be made up of simpler objects, which can in turn be made up of even simpler objects,
- Is continuous until it bottoms out with simple geometric primitives that can be drawn directly
- This is called **hierarchical modeling.**
- We will see that the transforms that were studied in the previous section play an important role in hierarchical modeling.

Building Complex Objects

- When drawing an object, use the coordinate system that is most natural for the object
- Usually, we want an object in its natural coordinates to be centered at the origin, (0,0), or at least to use the origin as a convenient reference point
- Then, to place it in the scene, we can use a scaling transform, followed by a rotation, followed by a translation to set its size, orientation, and position in the scene.
  - Recall that transformations used in this way are called modeling transformations.

Building Complex Objects

- The transforms are often applied in the order scale, then rotate, then translate, because scaling and rotation leave the reference point, (0,0), fixed
- Once the object has been scaled and rotated, it's easy to use a translation to move the reference point to any desired point in the scene.
- Remember that in the code, the transformations are specified in the opposite order from the order in which they are applied to the object and that the transformations are specified before drawing the object.
- So in the code, the translation would come first, followed by the rotation and then the scaling.
    - Modeling transforms are not always composed in this order, but it is the most common usage.

Building Complex Objects

- The modeling transformations that are used to place an object in the scene should not affect other objects in the scene.
- To limit their application to just the one object, we can save the current transformation before starting work on the object and restore it afterwards
- How this is done differs from one graphics API to another, but let's suppose here that there are subroutines *saveTransform*() and *restoreTransform*() for performing those tasks.
  - That is, *saveTransform* will make a copy of the modeling transformation that is currently in effect and store that copy. It does not change the current transformation; it merely saves a copy.
  - Later, when *restoreTransform* is called, it will retrieve that copy and will replace the current modeling transform with the retrieved transform.

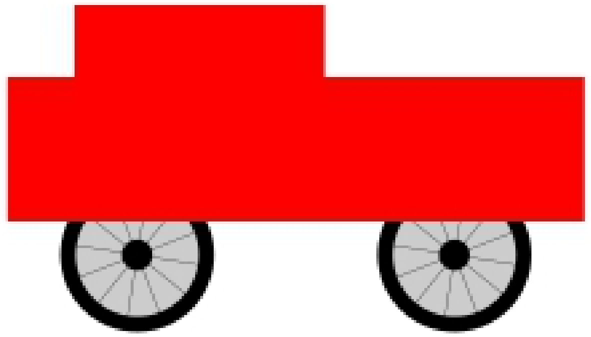
Building Complex Objects

- Typical code for drawing an object will then have the form:

```
saveTransform()
translate(dx,dy) // move object into position
rotate(r)        // set the orientation of the object
scale(sx,sy)     // set the size of the object
    .
    .  // draw the object, using its natural coordinates
    .
restoreTransform()
```

Building Complex Objects

- Suppose that we want to draw a simple 2D image of a cart with two wheels



Building Complex Objects

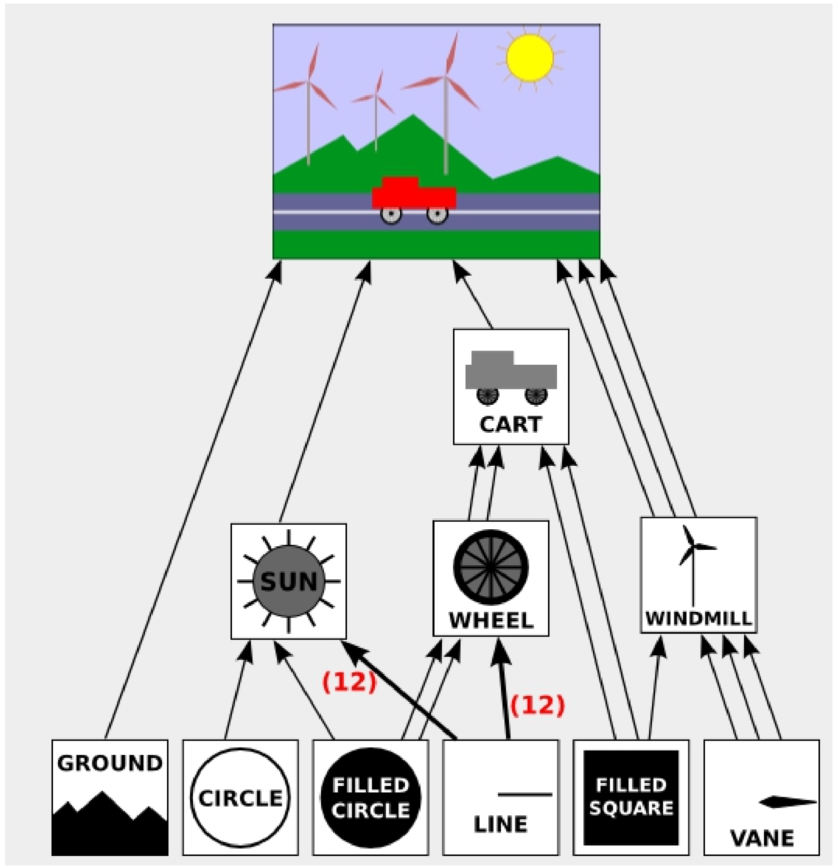
- Here is pseudocode for a subroutine that draws the cart in its own coordinate system:

```
subroutine drawCart() :
    saveTransform()       // save the current transform
    translate(-1.65,-0.1) // center of first wheel will be at (-1.65,-0.1)
    scale(0.8,0.8)        // scale to reduce radius from 1 to 0.8
    drawWheel()           // draw the first wheel
    restoreTransform()    // restore the saved transform
    saveTransform()       // save it again
    translate(1.5,-0.1)   // center of second wheel will be at (1.5,-0.1)
    scale(0.8,0.8)        // scale to reduce radius from 1 to 0.8
    drawWheel(g2)         // draw the second wheel
    restoreTransform()    // restore the transform
    setDrawingColor(RED)  // use red color to draw the rectangles
    fillRectangle(-3, 0, 6, 2)       // draw the body of the cart
    fillRectangle(-2.3, 1, 2.6, 1)  // draw the top of the cart
```

Scene Graphs

- Logically, the components of a complex scene form a structure
- In this structure, each object is associated with the sub-objects that it contains. If the scene is hierarchical, then the structure is hierarchical.
- This structure is known as a scene graph. A scene graph is a tree-like structure, with the root representing the entire scene, the children of the root representing the top-level objects in the scene, and so on
- We can visualize the scene graph for our sample scene:

Scene Graphs



- In this drawing, a single object can have several connections to one or more parent objects
- Each connection represents one occurrence of the object in its parent object
- For example, the "filled square" object occurs as a sub-object in the cart and in the windmill. It is used twice in the cart and once in the windmill
- Each arrow in the picture can be associated with a modeling transformation that places the sub-object into its parent object

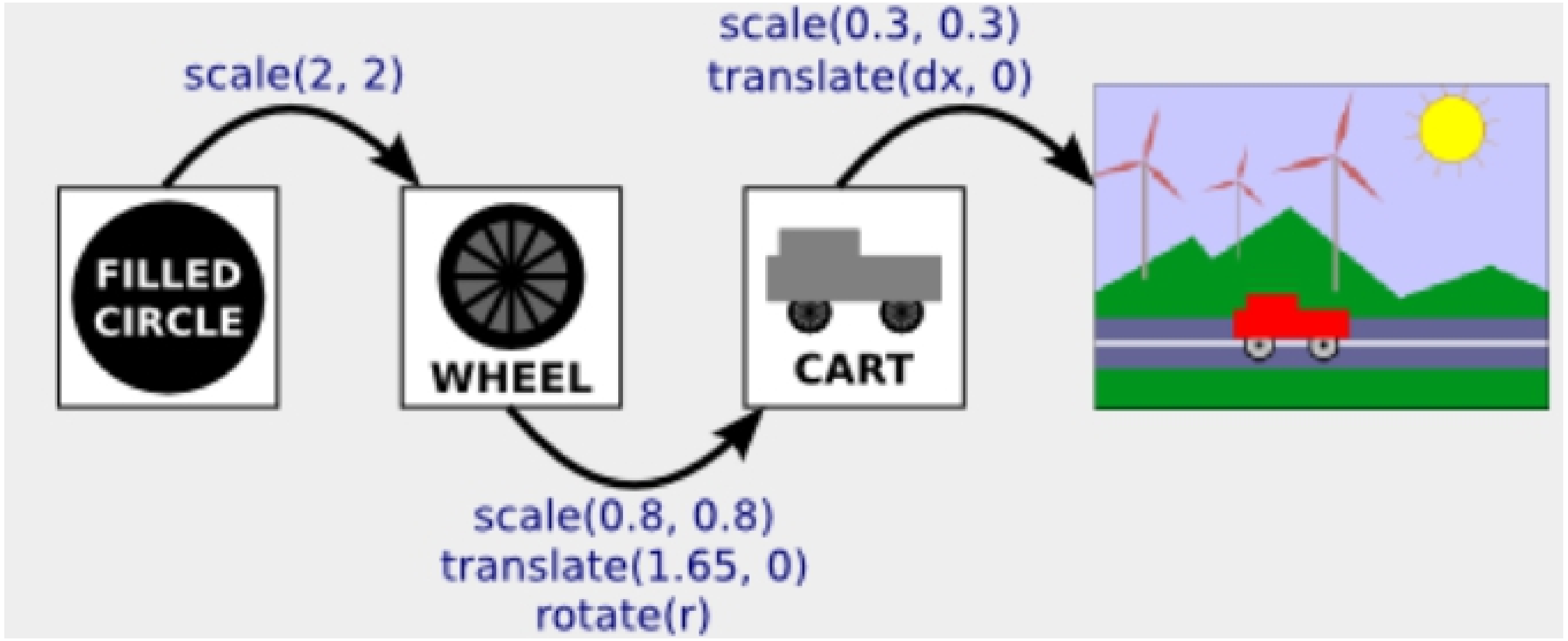
The Transform Stack

- Suppose that you write a subroutine to draw an object.
- At the beginning of the subroutine, you use a routine such as *saveTransform*() to save a copy of the current transform. At the end of the subroutine, you call *restoreTransform*() to reset the current transform back to the value that was saved
- Now, in order for this to work correctly for hierarchical graphics, these routines must actually use a stack of transforms
- (Recall that a stack is simply a list where items can be added, or "pushed," onto one end of the list and removed, or "popped," from the same end.)
- A stack has exactly the structure that is needed to implement these operations.

The Transform Stack

- Some graphics APIs come with transform stacks already defined
- For example, the HTML canvas API for 2D graphics, we'll see that it includes functions named *save*() and *restore*() that are actually *push* and *pop* operations on a stack.
- These functions are essential to implementing hierarchical graphics for an HTML canvas.
- Let's try to bring this all together by considering how it applies to a simple object in a complex scene: one of the filled circles that is part of the front wheel on the cart in our example scene.
- Here is a rearranged part of the scene graph for that scene, with labels added to show the modeling transformations that are applied to each object:

The Transform Stack



The Transform Stack

- The rotation amount for the wheel and the translation amount for the cart are shown as variables, since they are different in different frames of the animation
- When the computer starts drawing the scene, the modeling transform that is in effect is the identity transform, that is, no transform at all.
  - As it prepares to draw the cart, it saves a copy of the current transform (the identity) by pushing it onto the stack. It then modifies the current transform by multiplying it by the modeling transforms for the cart, *scale*(0.3,0.3) and *translate*(dx,0).
- When it comes to drawing the wheel, it again pushes the current transform (the modeling transform for the cart as a whole) onto the stack, and it modifies the current transform to take the wheel's modeling transforms into account.
- Similarly, when it comes to the filled circle, it saves the modeling transform for the wheel, and then applies the modeling transform for the circle.

HTML Canvas Graphics

- Most modern web browsers support a 2D graphics API that can be used to create images on a web page
- The API is implemented using JavaScript, the client-side programming language for the web
- The visible content of a web page is made up of "elements" such as headlines and paragraphs. The content is specified using the HTML language.
- A "canvas" is an HTML element. It appears on the page as a blank rectangular area which can be used as a drawing surface

HTML Canvas Graphics

- a canvas element is created with code of the form:

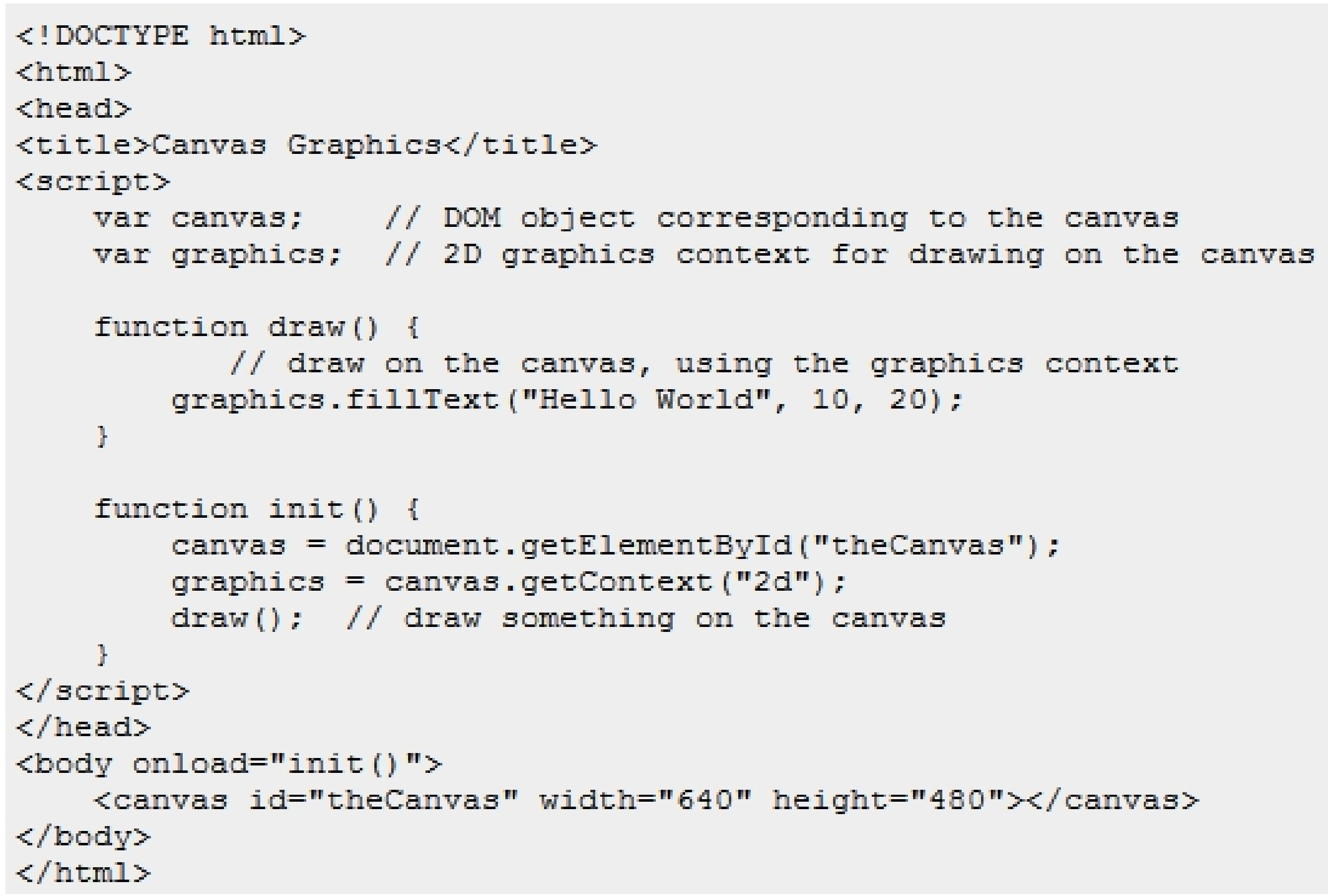
```
<canvas width="800" height="600" id="theCanvas"></canvas>
```

- The *width* and *height* give the size of the drawing area, in pixels. The id is an identifier that can be used to refer to the canvas in JavaScript
- To draw on a canvas, you need a graphics context. A graphics context is an object that contains functions for drawing shapes
- It also contains variables that record the current graphics state, including things like the current drawing color, transform, and font
- A typical starting point is:

```
canvas = document.getElementById("theCanvas");
graphics = canvas.getContext("2d");
```

- The first line gets a reference to the canvas element on the web page, using its id. The second line creates the graphics context for that canvas element

HTML Canvas Graphics



```
<!DOCTYPE html>
<html>
<head>
<title>Canvas Graphics</title>
<script>
    var canvas;    // DOM object corresponding to the canvas
    var graphics;  // 2D graphics context for drawing on the canvas

    function draw() {
           // draw on the canvas, using the graphics context
        graphics.fillText("Hello World", 10, 20);
    }

    function init() {
        canvas = document.getElementById("theCanvas");
        graphics = canvas.getContext("2d");
        draw();  // draw something on the canvas
    }
</script>
</head>
<body onload="init()">
    <canvas id="theCanvas" width="640" height="480"></canvas>
</body>
</html>
```

Shapes

- The default coordinate system on a canvas is the usual: The unit of measure is one pixel; (0,0) is at the upper left corner; the x-coordinate increases to the right; and the *y*-coordinate increases downward.
- The range of x and y values are given by the *width* and *height* properties of the <canvas> element.
- The unit of measure is one pixel at typical desktop resolution with no magnification. If you apply a magnification to a browser window, the unit of measure gets stretched.
- And on a high-resolution screen, one unit in the default coordinate system might correspond to several actual pixels on the display device.

Shapes

- The canvas API supports only a very limited set of basic shapes.
- The only basic shapes are rectangles and text. Other shapes must be created as paths.
- Shapes can be stroked and filled. That includes text:
  - When you stroke a string of text, a pen is dragged along the outlines of the characters; when you fill a string, the insides of the characters are filled.
- It only really makes sense to stroke text when the characters are rather large.

Shapes

- Here are the functions for drawing rectangles and text, where *graphics* refers to the object that represents the graphics context:
  - graphics.fillRect(x,y,w,h) — draws a filled rectangle with corner at (*x*,*y*), with width *w* and with height *h*. If the width or the height is less than or equal to zero, nothing is drawn.
  - graphics.strokeRect(x,y,w,h) — strokes the outline of the same rectangle.
  - graphics.clearRect(x,y,w,h) — clears the rectangle by filling it with fully transparent pixels, allowing the background of the canvas to show. graphics.fillText(str,x,y) — fills the characters in the string *str*. The left end of the baseline of the string is positioned at the point (*x*,*y*).
  - graphics.strokeText(str,x,y) — strokes the outlines of the characters in the string

Shapes

- A path can be created using functions in the graphics context. The context keeps track of a "current path.“
- Paths can contain lines, Bezier curves, and circular arcs. Here are the most common functions for working with paths:
  - graphics.beginPath() — start a new path. Any previous path is discarded, and the current path in the graphics context is now empty. Note that the graphics context also keeps track of the current point, the last point in the current path. After calling graphics.beginPath(), the current point is undefined.
  - graphics.moveTo(x,y) — move the current point to (x,y), without adding anything to the path. This can be used for the starting point of the path or to start a new, disconnected segment of the path.
  - graphics.lineTo(x,y) — add the line segment starting at current point and ending at (x,y) to the path, and move the current point to (x,y)

Shapes

- graphics.bezierCurveTo(cx1,cy1,c2x,cy2,x,y) — add a cubic Bezier curve to the path. The curve starts at the current point and ends at (x,y). The points (cx1,cy1) and (cx2,cy2) are the two control points for the curve. (Bezier curves and their control points were discussed in Subsection 2.2.3.)
- graphics.quadraticCurveTo(cx,cy,x,y) — adds a quadratic Bezier curve from the current point to (x,y), with control point (cx,cy).
- graphics.arc(x,y,r,startAngle,endAngle) — adds an arc of the circle with center (x,y) and radius r. The next two parameters give the starting and ending angle of the arc. They are measured in radians.
- graphics.closePath() — adds to the path a line from the current point back to the starting point of the current segment of the curve. (Recall that you start a new segment of the curve every time you use moveTo.)

Shapes

- Creating a curve with these commands does not draw anything. To get something visible to appear in the image, you must fill or stroke the path.
- The commands *graphics.fill*() and *graphics.stroke*() are used to fill and to stroke the current path.
- If you fill a path that has not been closed, the fill algorithm acts as though a final line segment had been added to close the path.
- When you stroke a shape, it's the center of the virtual pen that moves along the path.
- So, for high-precision canvas drawing, it's common to use paths that pass through the centers of pixels rather than through their corners.

Shapes

- For example, to draw a line that extends from the pixel with coordinates (100,200) to the pixel with coordinates (300,200), you would actually stroke the geometric line with endpoints (100.5,200.5) and (100.5,300.5). We should look at some examples. It takes four steps to draw a line:

```
graphics.beginPath();          // start a new path
graphics.moveTo(100.5,200.5);  // starting point of the new path
graphics.lineTo(300.5,200.5);  // add a line to the point (300.5,200.5)
graphics.stroke();             // draw the line
```

- Remember that the line remains as part of the current path until the next time you call graphics.beginPath().

Shapes

- Here's how to draw a filled, regular octagon centered at (200,400) and with radius 100:

```
graphics.beginPath();
graphics.moveTo(300,400);
for (var i = 1; i < 8; i++) {
    var angle = (2*Math.PI)/8 * i;
    var x = 200 + 100*Math.cos(angle);
    var y = 400 + 100*Math.sin(angle);
    graphics.lineTo(x,y);
}
graphics.closePath();
graphics.fill();
```

Shapes

- The function graphics.arc() can be used to draw a circle, with a start angle of 0 and an end angle of 2*Math.PI. Here's a filled circle with radius 100, centered at 200,300:

```
graphics.beginPath();
graphics.arc( 200, 300, 100, 0, 2*Math.PI );
graphics.fill();
```

- To draw just the outline of the circle, use graphics.stroke() in place of graphics.fill(). You can apply both operations to the same path. If you look at the details of graphics.arc(), you can see how to draw a wedge of a circle:

```
graphics.beginPath();
graphics.moveTo(200,300);   // Move current point to center of the circle.
graphics.arc(200,300,100,0,Math.PI/4);  // Arc, plus line from current point.
graphics.lineTo(200,300);  // Line from end of arc back to center of circle.
graphics.fill();  // Fill the wedge.
```

Stroke and Fill

- Attributes such as line width that affect the visual appearance of strokes and fills are stored as properties of the graphics context. For example, the value of graphics.lineWidth is a number that represents the width that will be used for strokes.
  - graphics.lineWidth = 2.5; // Change the current width.
- The change affects subsequent strokes. You can also read the current value:
  - saveWidth = graphics.lineWidth; // Save current width.
- The property graphics.lineCap controls the appearance of the endpoints of a stroke. It can be set to "round", "square", or "butt". The quotation marks are part of the value. For example,
  - graphics.lineCap = "round";
- Similarly, graphics.lineJoin controls the appearance of the point where one segment of a stroke joins another segment; its possible values are "round", "bevel", or "miter".

Colour

- Color is controlled by the values of the properties graphics.fillStyle and graphics.strokeStyle.
- The graphics context maintains separate styles for filling and for stroking. A solid color for stroking or filling is specified as a string.
- Valid color strings are ones that can be used in CSS, the language that is used to specify colors and other style properties of elements on web pages.
- Many solid colors can be specified by their names, such as "red", "black", and "beige".
- An RGB color can be specified as a string of the form "rgb(r,g,b)", where the parentheses contain three numbers in the range 0 to 255 giving the red, green, and blue components of the color. Hexadecimal color codes are also supported, in the form "#XXYYZZ" where XX, YY, and ZZ are two-digit hexadecimal numbers giving the RGB color components.

Colour

- For example,

```
graphics.fillStyle = "rgb(200,200,255)"; // light blue
graphics.strokeStyle = "#0070A0"; // a darker, greenish blue
```

- The style can actually be more complicated than a simple solid color: Gradients and patterns are also supported. As an example, a gradient can be created with a series of steps such as

```
var lineargradient = graphics.createLinearGradient(420,420,550,200);
lineargradient.addColorStop(0,"red");
lineargradient.addColorStop(0.5,"yellow");
lineargradient.addColorStop(1,"green");
graphics.fillStyle = lineargradient;  // Use a gradient fill!
```

Colour

- The first line creates a linear gradient that will vary in color along the line segment from the point (420,420) to the point (550,200).
- Colors for the gradient are specified by the *addColorStop* function: the first parameter gives the fraction of the distance from the initial point to the final point where that color is applied, and the second is a string that specifies the color itself.
- A color stop at 0 specifies the color at the initial point; a color stop at 1 specifies the color at the final point.
- Once a gradient has been created, it can be used both as a fill style and as a stroke style in the graphics context.

Fonts

- the font that is used for drawing text is the value of the property graphics.font. The value is a string that could be used to specify a font in CSS.
- As such, it can be fairly complicated, but the simplest versions include a font-size (such as 20px or 150%) and a font-family (such as serif, sans-serif, monospace, or the name of any font that is accessible to the web page). You can add italic or bold or both to the front of the string. Some examples:

```
graphics.font = "2cm monospace";  // the size is in centimeters
graphics.font = "bold 18px sans-serif";
graphics.font = "italic 150% serif";   // size is 150% of the usual size
```

- The default is "10px sans-serif," which is usually too small. Note that text, like all drawing, is subject to coordinate transforms. Applying a scaling operation changes the size of the text, and a negative scaling factor can produce mirror-image text.

Transforms

- A graphics context has three basic functions for modifying the current transform by scaling, rotation, and translation. There are also functions that will compose the current transform with an arbitrary transform and for completely replacing the current transform:

  graphics.scale(sx,sy) — scale by sx in the x-direction and sy in the y-direction.

  graphics.rotate(angle) — rotate by angle radians about the origin. A positive rotation is clockwise in the default coordinate system.

  graphics.translate(tx,ty) — translate by tx in the x-direction and ty in the y-direction.

  graphics.transform(a,b,c,d,e,f) — apply the transformation x1 = a*x + c*y + e, and y1 = b*x + d*y + f.

  graphics.setTransform(a,b,c,d,e,f) — discard the current transformation, and set the current transformation to be x1 = a*x + c*y + e, and y1 = b*x + d*y + f.

- Note that there is no shear transform, but you can apply a shear as a general transform. For example, for a horizontal shear with shear factor 0.5, use

  graphics.transform(1, 0, 0.5, 1, 0, 0)

Transforms

- To implement hierarchical modeling, you need to be able to save the current transformation so that you can restore it later.
- Unfortunately, no way is provided to read the current transformation from a canvas graphics context.
- However, the graphics context itself keeps a stack of transformations and provides methods for pushing and popping the current transformation.
- In fact, these methods do more than save and restore the current transformation.
- They actually save and restore almost the entire state of the graphics context, including properties such as current colors, line width, and font (but not the current path):

Transforms

- Operations include;
  - graphics.save() — push a copy of the current state of the graphics context, including the current transformation, onto the stack.
  - graphics.restore() — remove the top item from the stack, containing a saved state of the graphics context, and restore the graphics context to that state.
- Using these methods, the basic setup for drawing an object with a modeling transform becomes:

```
graphics.save();          // save a copy of the current state
graphics.translate(a,b);  // apply modeling transformations
graphics.rotate(r);
graphics.scale(s,s);
     ….  // Draw the object!
graphics.restore();       // restore the saved state
```

Transforms

- Note that if drawing the object includes any changes to attributes such as drawing color, those changes will be also undone by the call to graphics.restore().
- In hierarchical graphics, this is usually what you want, and it eliminates the need to have extra statements for saving and restoring things like color.
- Now that we know how to do transformations, we can see how to draw an oval using the canvas API.
- Now that we know how to do transformations, we can see how to draw an oval using the canvas API.
- Suppose that we want an oval with center at (x,y), with horizontal radius r1 and with vertical radius r2. The idea is to draw a circle of radius 1 with center at (0,0), then transform it.
- The circle needs to be scaled by a factor of r1 horizontally and r2 vertically. It should then be translated to move its center from (0,0) to (x,y).
  - We can use graphics.save() and graphics.restore() to make sure that the transformations only affect the circle.

Transforms

- Recalling that the order of transforms in the code is the opposite of the order in which they are applied to objects, this becomes:

```
graphics.save();
graphics.translate( x, y );
graphics.scale( r1, r2 );
graphics.beginPath();
graphics.arc( 0, 0, 1, 0, Math.PI );  // a circle of radius 1
graphics.restore();
graphics.stroke();
```

Transforms

- Recalling that the order of transforms in the code is the opposite of the order in which they are applied to objects, this becomes:

```
graphics.save();
graphics.translate( x, y );
graphics.scale( r1, r2 );
graphics.beginPath();
graphics.arc( 0, 0, 1, 0, Math.PI );  // a circle of radius 1
graphics.restore();
graphics.stroke();
```

Images

- An image on a web page is specified by an element in the web page source such as

```
<img src="pic.jpg" width="400" height="300" id="mypic">
```

- The src attribute specifies the URL from which the image is loaded.
- The optional id can be used to reference the image in JavaScript. In the script,

```
image = document.getElementById("mypic");
```

- gets a reference to the object that represents the image in the document structure

Images

- Once you have such an object, you can use it to draw the image on a canvas. If graphics is a graphics context for the canvas, then image is drawn with its upper left corner at (x,y) by:

  graphics.drawImage(image, x, y);

- Both the point (x,y) and the image itself are transformed by any transformation in effect in the graphics context
- This will draw the image using its natural width and height (scaled by the transformation, if any)
- With this version of drawImage, the image is scaled to fit the specified rectangle.
- Now, suppose that the image you want to draw onto the canvas is not part of the web page? Use the document object to create an img element:

  newImage = document.createElement("img");

Images

- An img element needs a src attribute that specifies the URL from which it is to be loaded. For example,

  newImage.src = "pic2.jpg";

- As soon as you assign a value to the src attribute, the browser starts loading the image. The loading is done asynchronously; that is, the computer continues to execute the script without waiting for the load to complete.
- You need to assign a function to the image's onload property before setting the src. That function will be called when the image has been fully loaded.
- Putting this together, here is a simple JavaScript function for loading an image from a specified URL and drawing it on a canvas after it has loaded:

Images

```
function loadAndDraw( imageURL, x, y )
{
    var image = document.createElement("img");
    image.onload = doneLoading;
    image.src = imageURL;
    function doneLoading() {
        graphics.drawImage(image, x, y);
    }
}
```

End

- Questions???