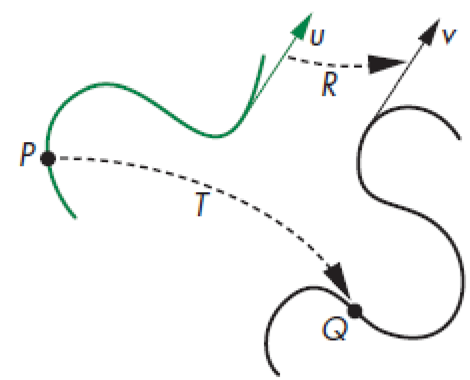
COSC 414/519I: Computer Graphics

2023W2

Shan Du

Affine Transformations

- A transformation is a function that takes a point (or vector) and maps it into another point (or vector).
- $Q = T(P)$ for points
- $v = R(u)$ for vectors
- By using homogeneous



Transformation.

coordinate representations, we can represent both points and vectors as 4-dimensional column matrices.

Affine Transformations

- We can define the transformation using a single function f.
- We can obtain a useful class of transformations if we place restrictions on f. The most important restriction is linearity.
- A function f is a linear function if and only if, for any scalars α and β and any two vertices (or vectors) p and q,

$$f(\alpha p + \beta q) = \alpha f(p) + \beta f(q)$$

Affine Transformations

- The importance of such functions is that if we know the transformations of p and q, we can obtain the transformations of linear combinations of p and q by taking linear combinations of their transformations.
- Using homogeneous coordinates, a linear transformation can always be written in terms of two representations, **u** and **v**, as a matrix multiplication, $\boldsymbol{v} = \boldsymbol{C}\boldsymbol{u}$, where $\boldsymbol{C}$ is a square matrix.

Affine Transformations

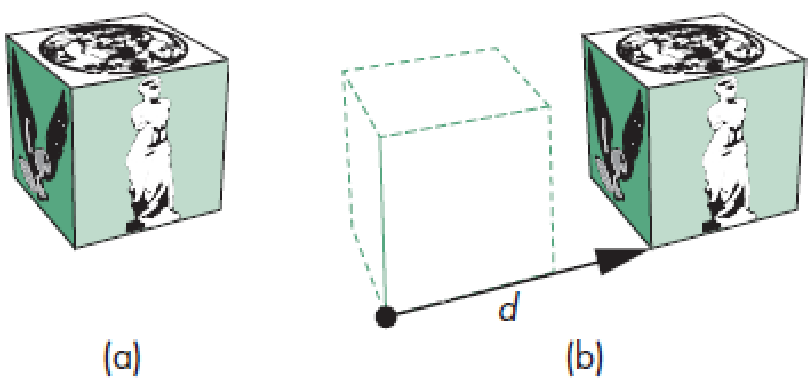
- A linear transformation can be seen as: 1) a change of frames; 2) transformation of vertices within the same frame.
- With homogeneous coordinates, **C** is a 4×4 matrix that leaves the fourth component of a representation unchanged.

$$\boldsymbol{C} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

- Translation is an operation that displaces points by a fixed distance in a given direction.

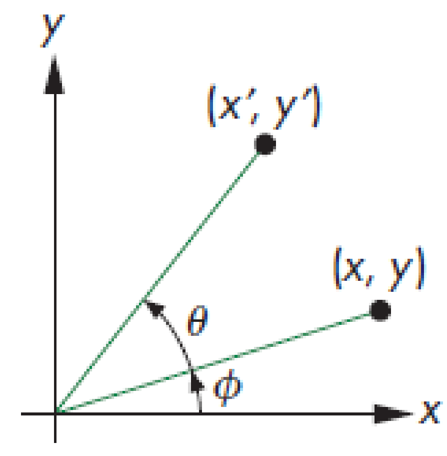
$$P' = P + d$$



Translation. (a) Object in original position. (b) Object translated.

Rotation

- Rotate a point about the origin in a 2D plane.
- Represent (x, y) and (x', y') in polar form.
- $x = \rho cos\Phi$
- $y = \rho sin\Phi$
- $x' = \rho\cos(\theta + \Phi)$
- $y' = \rho\sin(\theta + \Phi)$



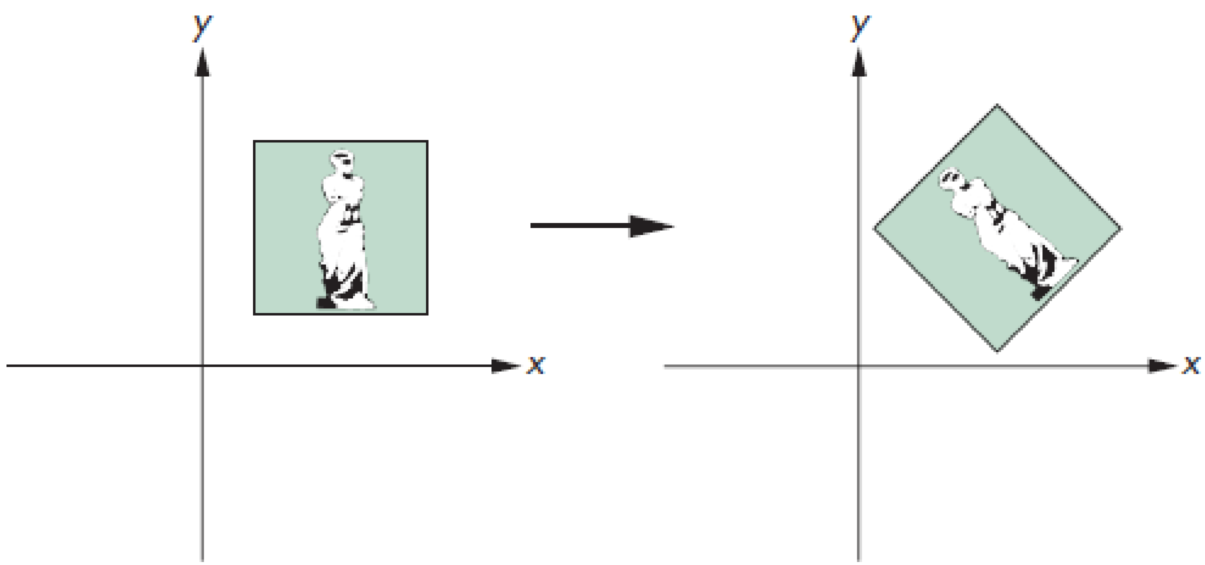
Two-dimensional rotation.

Then

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

Rotation About a Fixed Point

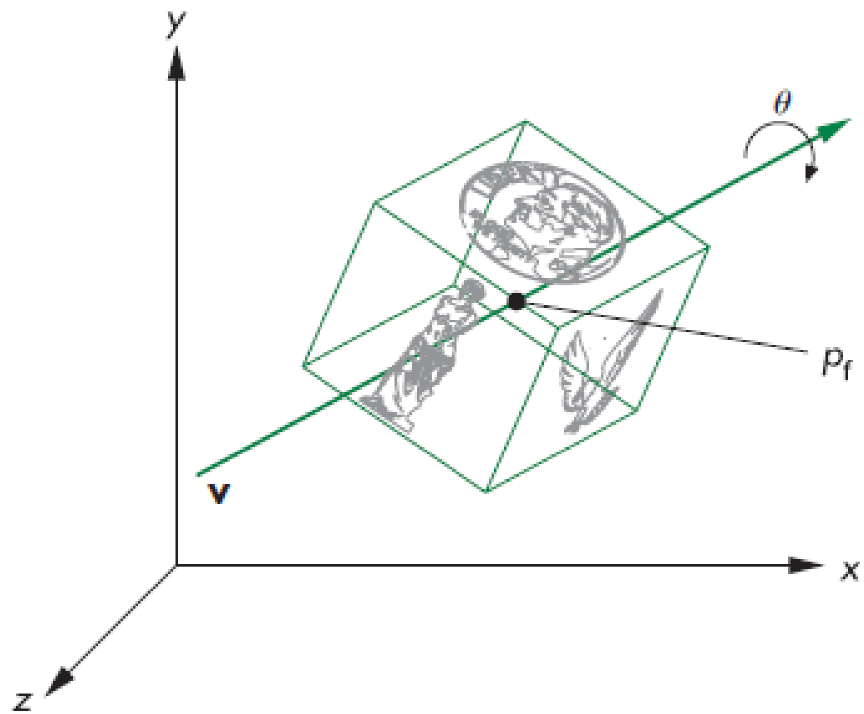
- The center of the rotation is called the fixed point of the rotation.
- Positive rotation - counterclockwise



Rotation about a fixed point.

3D Rotation

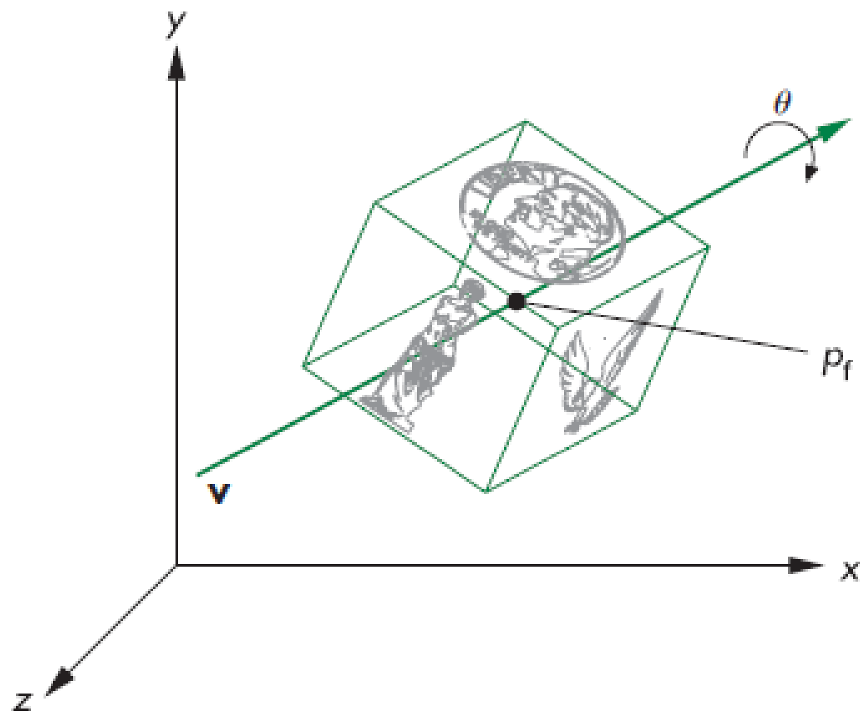
- Rotation in the 2D plane *z=0* is equivalent to a 3D rotation about the *z*-axis. Points in planes of constant *z* all rotate in a similar manner, leaving their *z* values unchanged.



Three-dimensional rotation.

3D Rotation

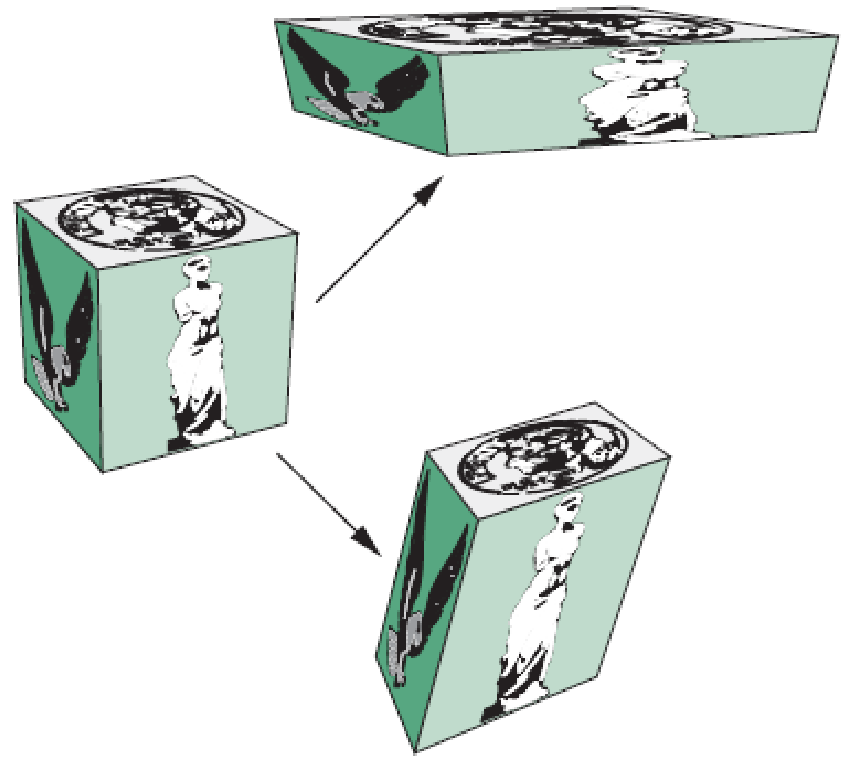
- A general 3D rotation can be specified by: a fixed point P_f, a rotation angle, and a line or a vector about which to rotate.
- Rotation and translation are rigid-body transformations.



Three-dimensional rotation.

Non-rigid-body Transformations

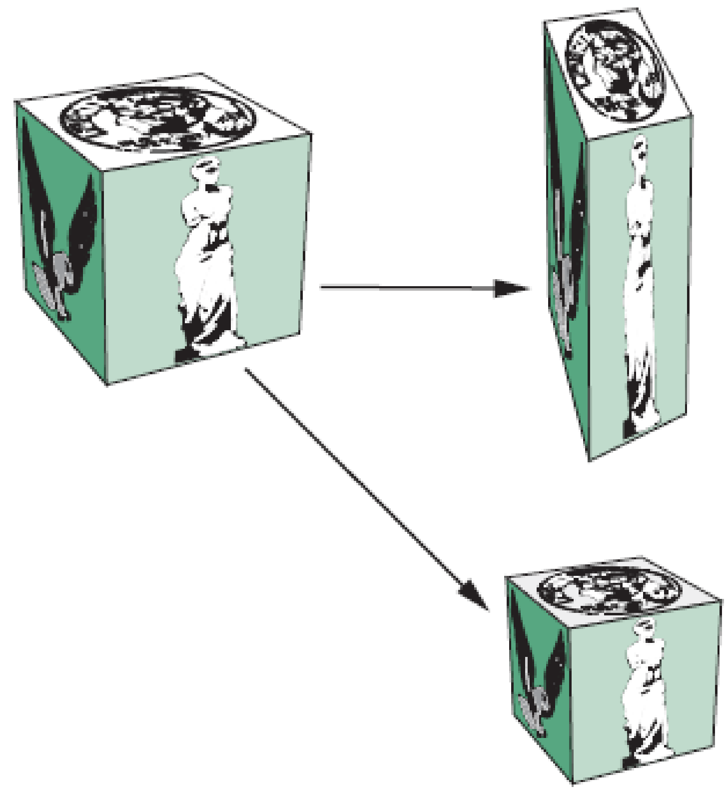
- Transformations that can alter the shape or volume of an object



Non-rigid-body transformations.

Scaling

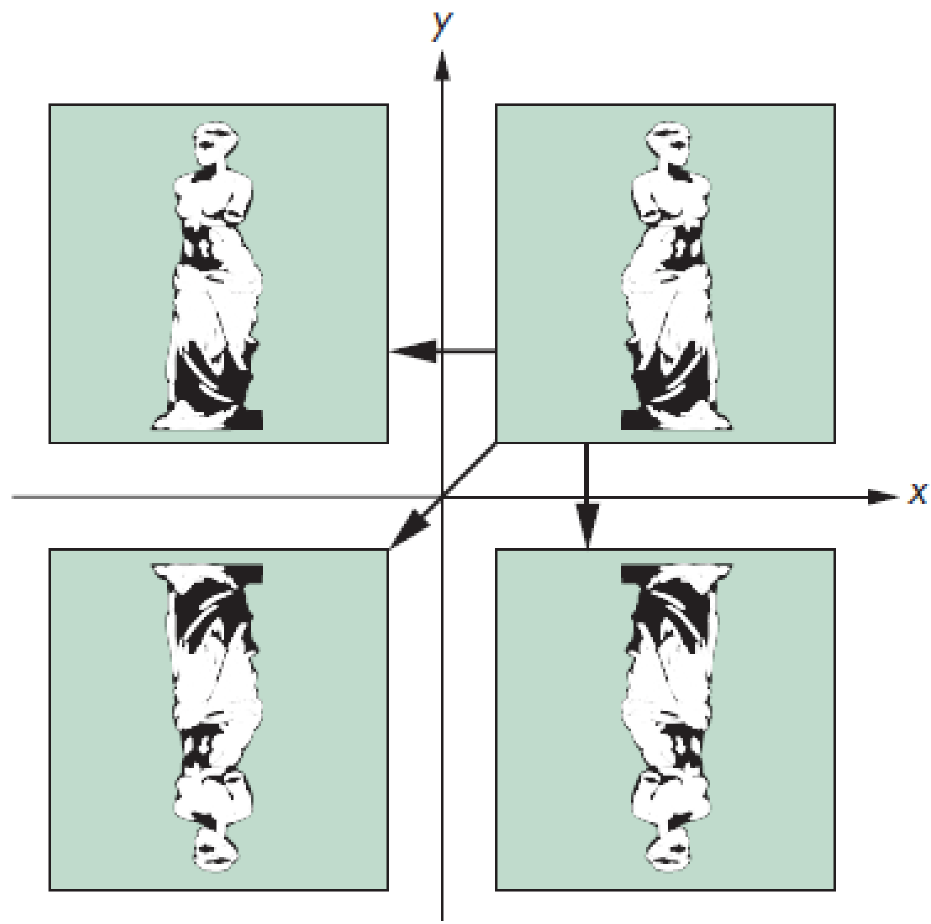
- To specify a scaling, we can specify a fixed point, a direction to scale, and a scale factor (α).
- α>1, the object gets longer in the specified direction.
- $0 \leq \alpha < 1$, the object gets shorter in that direction.



Uniform and nonuniform scaling.

Reflection

- Negative values of α give us reflection about the fixed point, in the scaling direction.



Reflection.

Transformations in Homogeneous Coordinates

- Each affine transformation can be represented by a 4×4 matrix of the form

$$\boldsymbol{A} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

$$P' = P + d$$

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \qquad P' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}, \qquad d = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \\ 0 \end{bmatrix}$$

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \boldsymbol{T}^{-\mathbf{1}} = \begin{bmatrix} 1 & 0 & 0 & -\alpha_x \\ 0 & 1 & 0 & -\alpha_y \\ 0 & 0 & 1 & -\alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

- Assume the fixed point is the origin

$$x' = \beta_x x$$
$$y' = \beta_y y$$
$$z' = \beta_z z$$
$$P' = SP$$

$$\boldsymbol{S} = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \boldsymbol{S^{-1}} = \begin{bmatrix} 1/\beta_x & 0 & 0 & 0 \\ 0 & 1/\beta_y & 0 & 0 \\ 0 & 0 & 1/\beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

- Assume the fixed point is the origin
- The 3D rotation is the independent rotation about the three coordinate axes.
- Rotate about *z*-axis, *z* values unchanged

$$R_z = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

- Rotate about *x*-axis, *x* values unchanged

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

- Rotate about *y*-axis, *y* values unchanged

$$R_y = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

$$R^{-1}(\theta) = R(-\theta)$$

Because all cosine terms are on the diagonal and the sine terms are off-diagonal, and because

$\cos(-\theta) = \cos(\theta)$

$\sin(-\theta) = -\sin(\theta)$

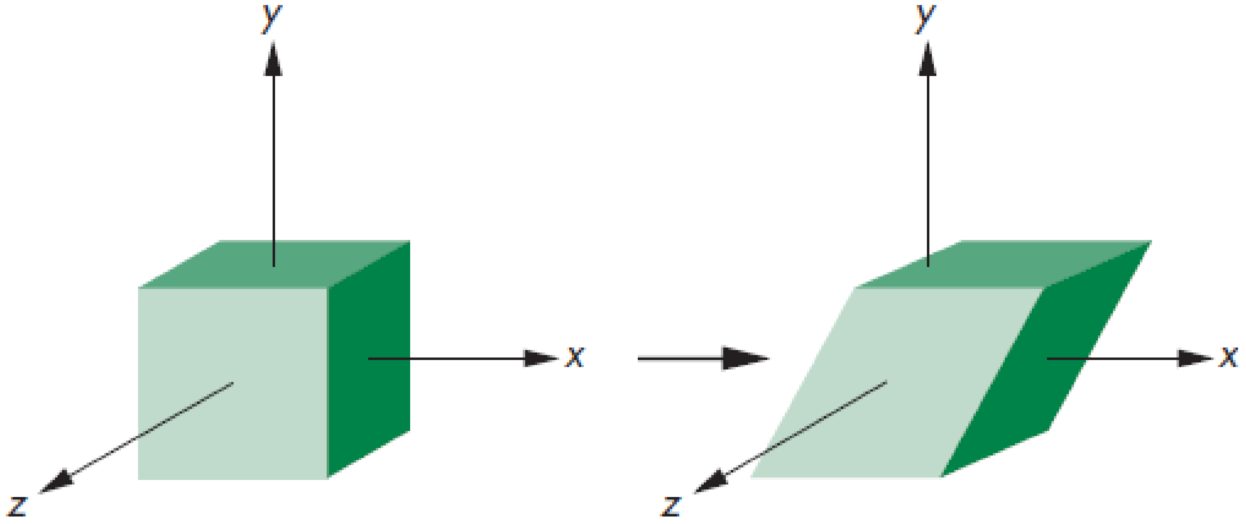
We have

$$R^{-1}(\theta) = R^{T}(\theta)$$

A matrix whose inverse is equal to its transpose is called an orthogonal matrix.

Shear

- Consider a cube centered at the origin, aligned with the axes, and viewed from the positive *z*-axis.
- If we pull the top to the right and the bottom to left, we shear the object in the *x* direction.



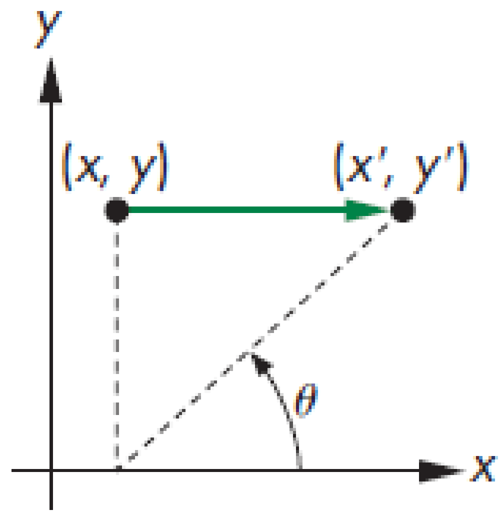
Shear.

Shear

$$x' = x + y\cot\theta$$
$$y' = y$$
$$z' = z$$

Then

$$H_x(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Computation of the shear matrix.

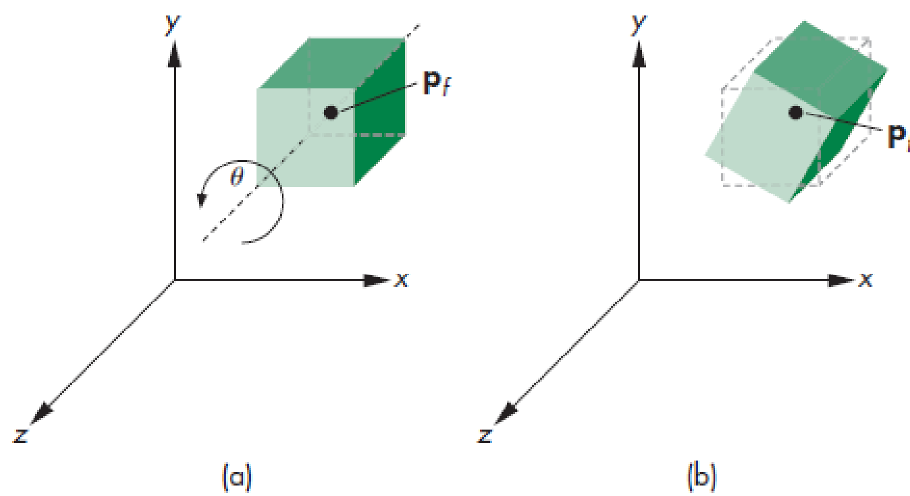
$$H_x^{-1}(\theta) = H_x(-\theta)$$

Concatenation of Transformations

- Suppose we carry out three successive transformations on the homogeneous representations of a point p, creating a new point q.
- Because the matrix product is associative, we can have $\mathrm{q} = CBAp$ to replace $\mathrm{q} = (C(B(Ap)))$.
- If we have many points, we calculate $\mathrm{M} = CBA$ first, then $\mathrm{q} = Mp$.

Rotation About a Fixed Point

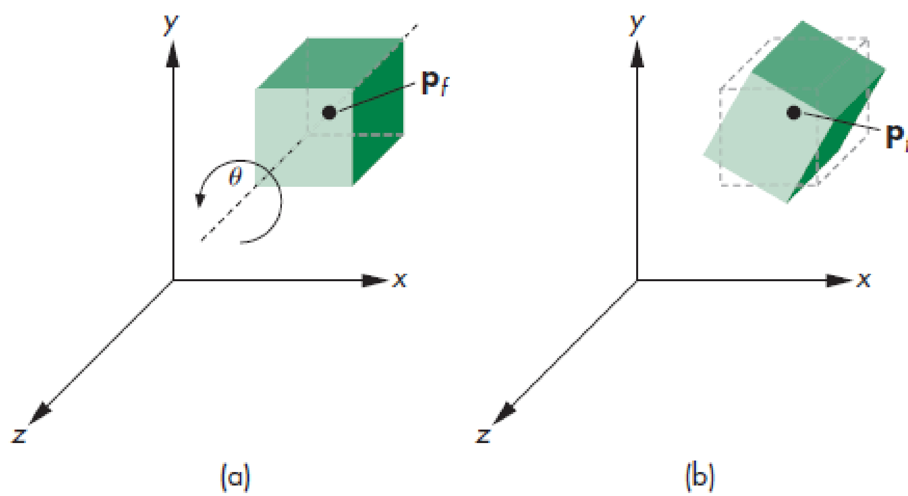
- Consider a cube with its center at P_f and its sides aligned with the axes.
- Rotate the cube with the fixed point at its center P_f.



Rotation of a cube about its center.

Rotation About a Fixed Point

- Move the cube to the origin
- Apply $R_z(\theta)$
- Move the object back to its center



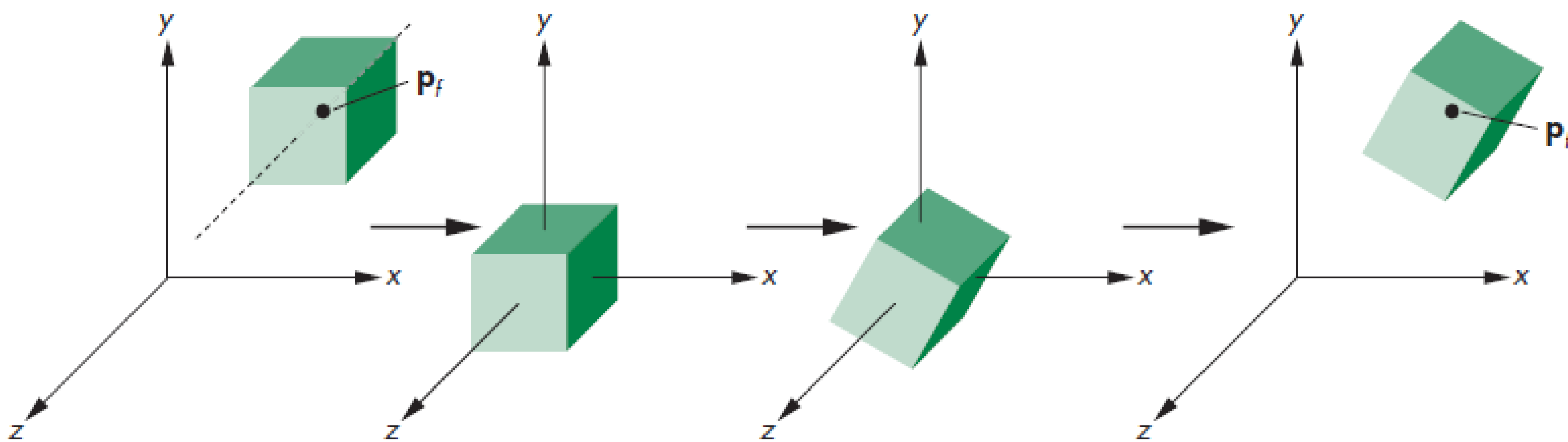
Rotation of a cube about its center.

Sequence of Transformations

The transformations are: $\mathrm{T}(-P_f)$, $R_z(\theta)$, and $\mathrm{T}(P_f)$.

$$M = \mathrm{T}(P_f)R_z(\theta)\mathrm{T}(-P_f)$$

$$M = \begin{bmatrix} cos\theta & -sin\theta & 0 & x_f - x_f cos\theta + y_f sin\theta \\ sin\theta & cos\theta & 0 & y_f - x_f sin\theta - y_f cos\theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

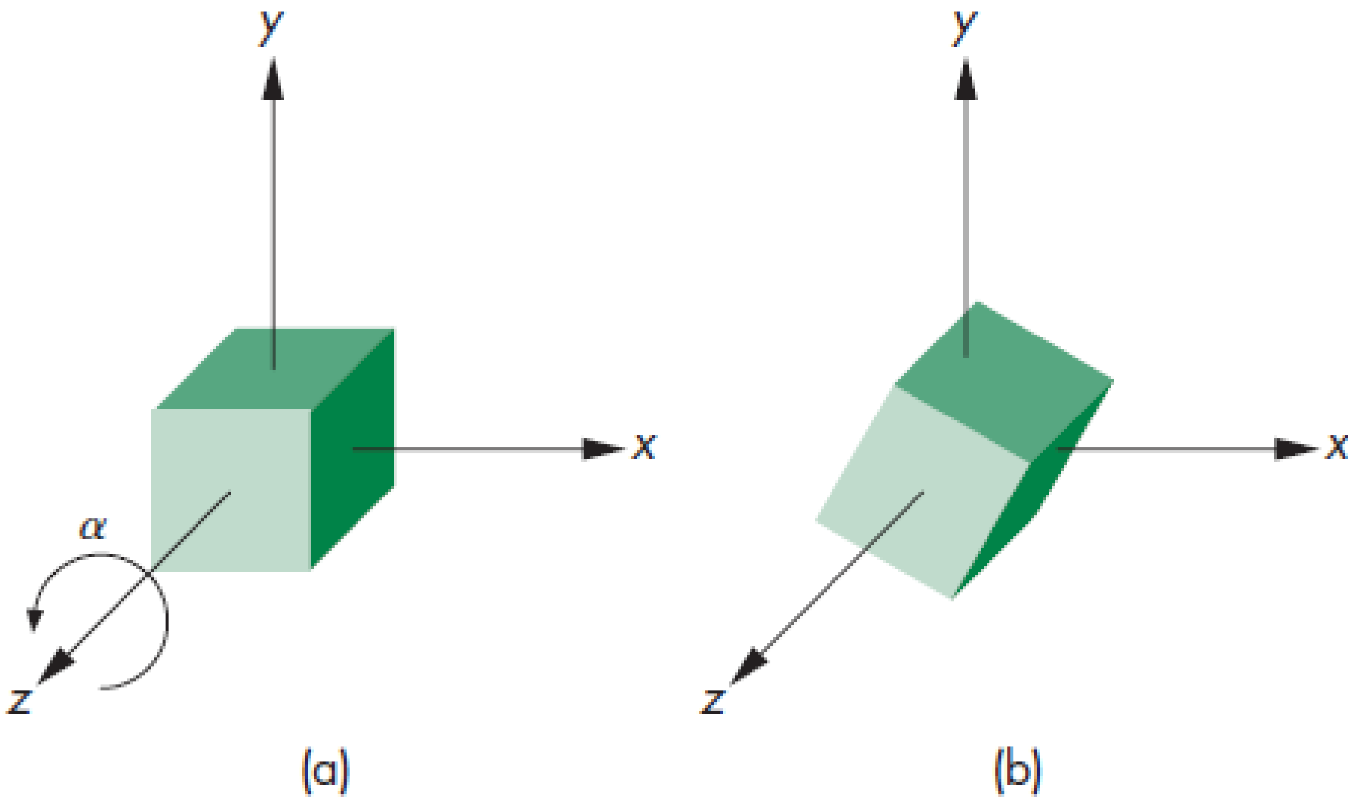


Sequence of transformations.

General Rotation

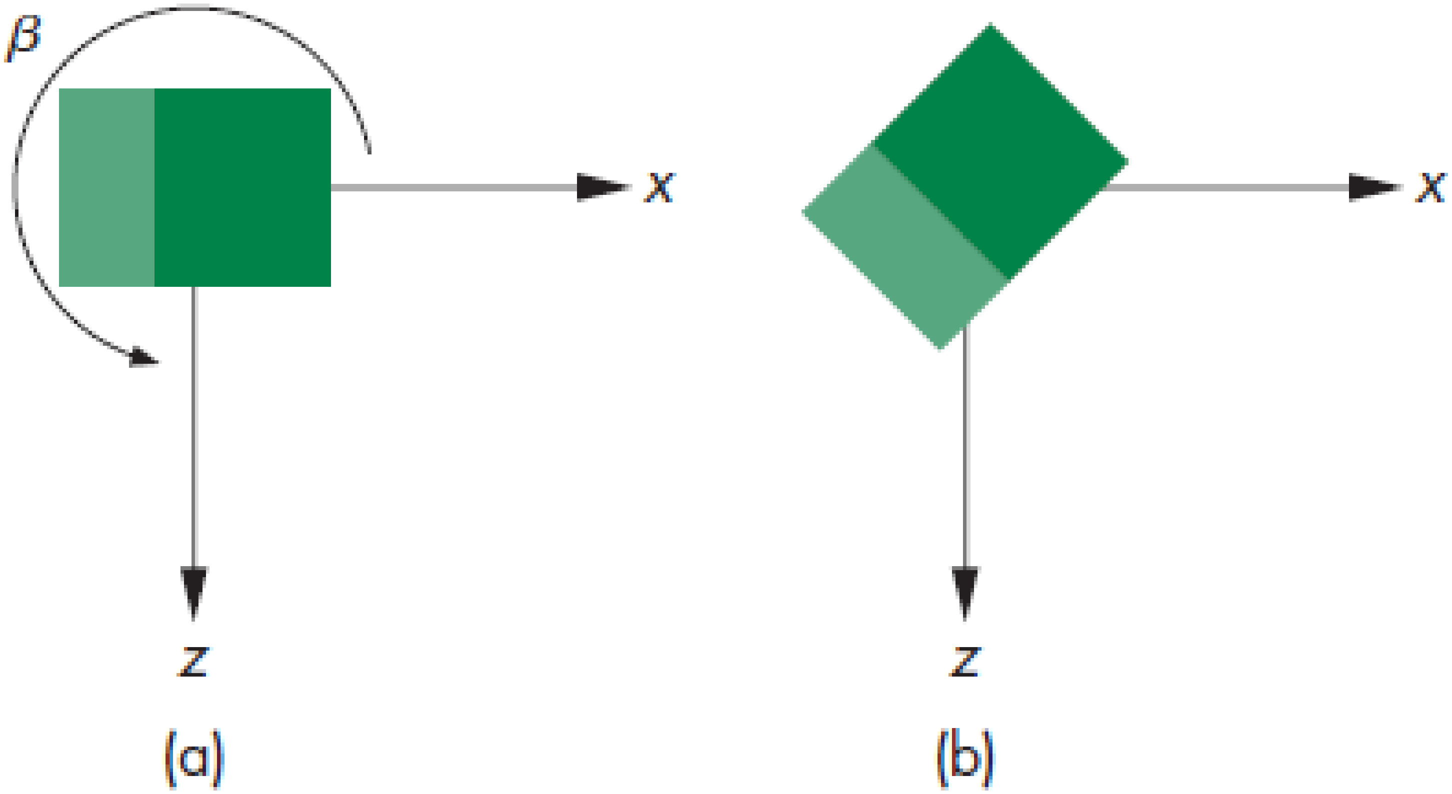
- Rotation about the origin has three degrees of freedom (DOF). We can specify an arbitrary rotation about the origin in terms of three successive rotations about the three axes.
- One way to form the desired rotation matrix is by first doing a rotation about the *z*-axis, then doing a rotation about the *y*-axis, and then with a rotation about the *x*-axis.

General Rotation



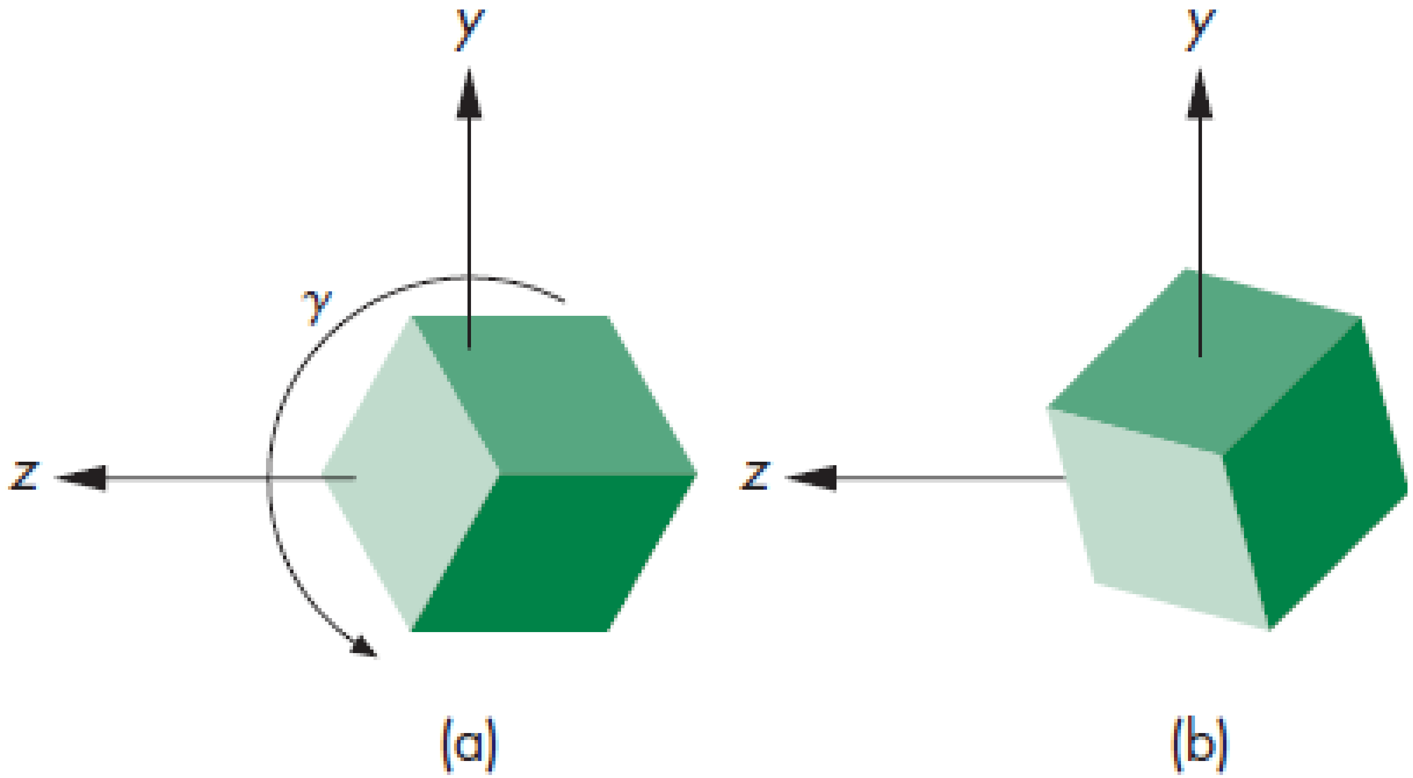
Rotation of a cube about the z-axis. (a) Cube before rotation. (b) Cube after rotation.

General Rotation



Rotation of a cube about the y-axis.

General Rotation



Rotation of a cube about the x-axis.

The Instance Transformation

- Consider a scene composed of many simple object.
- One option is to specify each of these objects, through its vertices, in the desired location with the desired orientation and size.

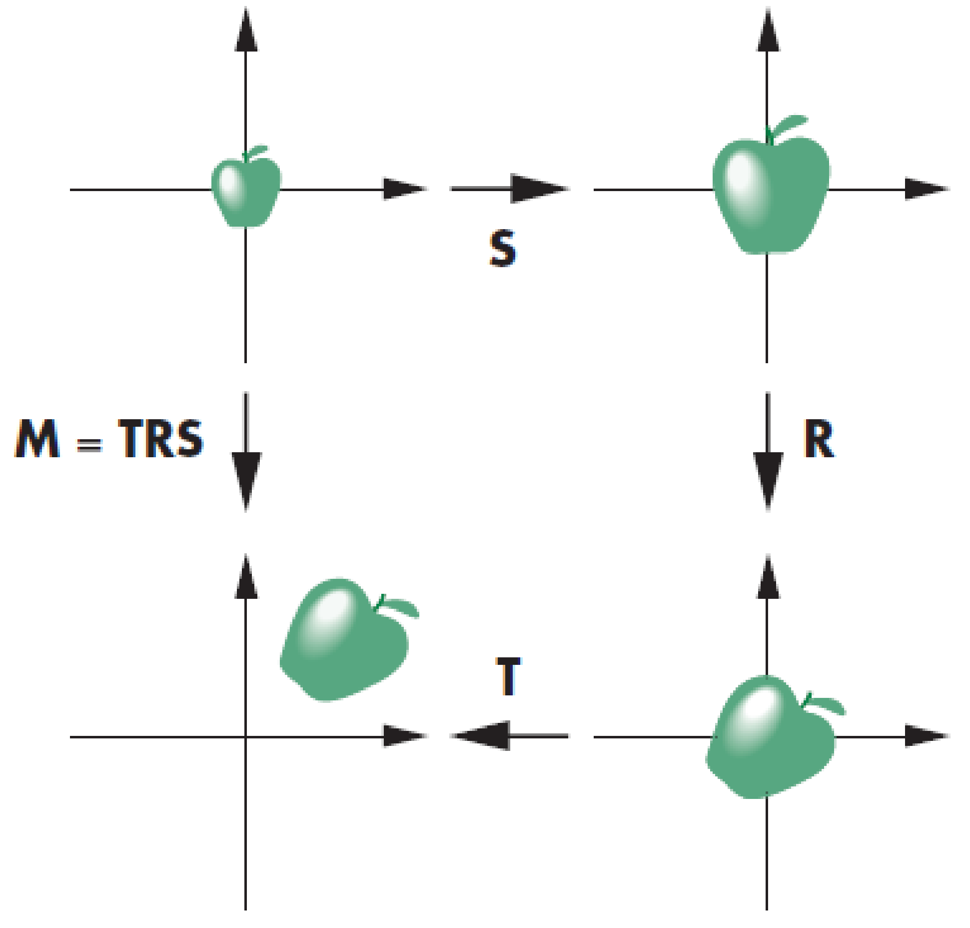
Scene of simple objects.

The Instance Transformation

- A preferred method is to specify each of the object types once at a convenient size, in a convenient place, and with a convenient orientation.
- Each occurrence of an object in the scene is an instance of that object’s prototype, and we can obtain the desired size, orientation, and location by applying an affine transformation - the instant transformation- to the prototype.

The Instance Transformation

- A complex object that is used many times need only to be loaded onto the GPU once.
- Displaying each instance of it requires only sending the appropriate instance transformation to the GPU before rendering the object.



Instance transformation.

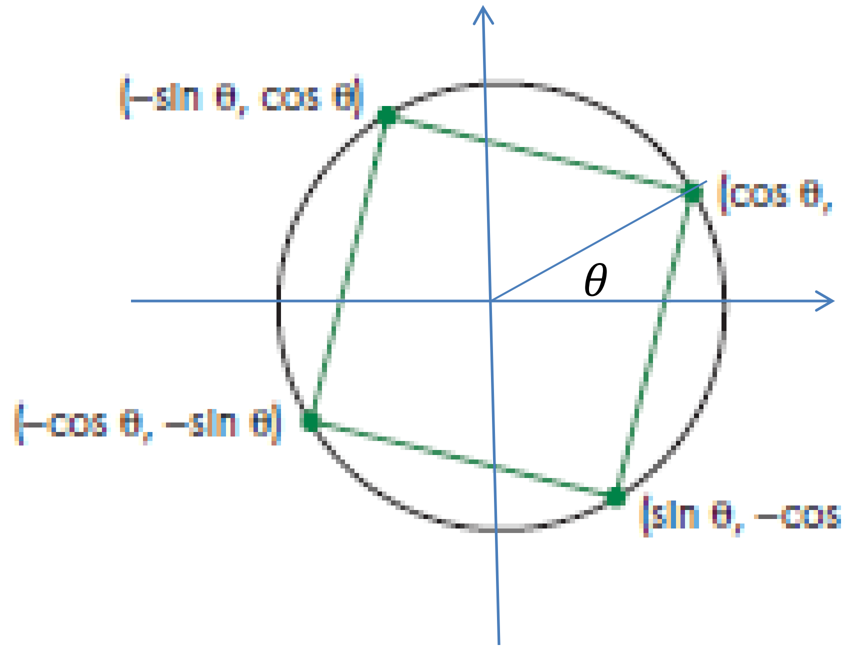
A simple Example

Rotate a square at a constant rate

- How to do?
  - Generate new vertex data periodically
  - Send data to GPU
  - Do rendering each time when we send new data.
- Can we do better?

The Rotating Square

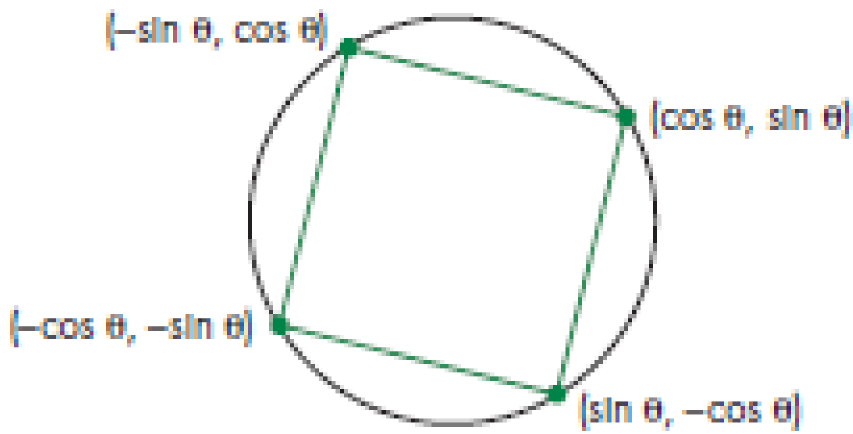
- We draw a unit circle whose radius is 1.
- We find a point on the circumference of the circle with an angle θ.
- This point's location is $(cos\theta, sin\theta)$.



The other three points $(-sin\theta, cos\theta), (-cos\theta, -sin\theta)$, and $(sin\theta, -cos\theta)$ also lie on the unit circle.

The Rotating Square

- These four points are equidistant along the circumference. If we connect the points, we can form a square centered at the origin whose sides are of length $\sqrt{2}$.



The Rotating Square

- We can start with $\theta = 0$, which gives us the four vertices $(0,1), (1,0), (-1,0)\ and\ (0,-1)$. We can send these vertices to GPU by first setting up an array:

```
var vertices = [
  vec2(0, 1),
  vec2(1, 0),
  vec2(-1, 0),
  vec2(0, -1)
];
```

The Rotating Square

- And then sending the array:

```
var bufferId = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);
```

- We can render these data using a render function:

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

The Rotating Square

- If we want to display the square with a different θ, we could compute new vertices and send these vertices to GPU, followed by another rendering.
- If we want the square rotating, put in a loop that increments θ by a fixed amount each time.
- …… but not efficient!!

The Rotating Square

- A better solution:
  - Use a new type of shader variable to transfer data from CPU to variables in the shader – uniform qualified variables.
- Vertex Shader

```
attribute vec4 vPosition;
uniform float theta;

void main()
{
  gl_Position.x = -sin(theta) * vPosition.x + cos(theta) * vPosition.y;
  gl_Position.y =  sin(theta) * vPosition.y + cos(theta) * vPosition.x;
  gl_Position.z = 0.0;
  gl_Position.w = 1.0;
}
```

The Rotating Square

In order to get a value of θ for the shader, we must perform two steps:

- Provide a link between *theta* in the shader and a variable in the application.
- Send the value from the application to the shader.

- Suppose we have a variable in the application:

```
var theta = 0.0;
```

- When the shaders and application are compiled and linked by InitShaders, set a link:

```
var thetaLoc = gl.getUniformLocation(program, "theta");
```

The Rotating Square

- Then send the value of *theta* from the application to the shader by

```
gl.uniform1f(thetaLoc, theta);
```

- We send new values of *theta* to the vertex shader in the *render* function:

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  theta += 0.1;
  gl.uniform1f(thetaLoc, theta);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);

  render();
}
```

The Display Process

- Using the above code, we can only see the initial square. Why?
- We need to examine how and when the display is changed.

The Display Process

- The image is stored as pixels in a buffer maintained by the window system and is periodically and automatically redrawn on the display (about 60 frame per second (fps)).
- Although a browser is being redrawn by the display process, its contents are unchanged until some action takes place that changes pixels in the display buffer.

The Display Process

- In our example, the *onload* event starts the execution of our application with the *init* function. The execution ends in the *render* function that invokes the *gl.drawArray* function.
- At this point, execution of our code is complete and the results are displayed.

The Display Process

- However, to render the rotating square, the recursion puts the drawing inside an infinite loop. We cannot reach the end of our code and we never see changes to the display.

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  theta += 0.1;
  gl.uniform1f(thetaLoc, theta);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);

  render();
}
```

Double Buffering

- Suppose that the color buffer in the framebuffer is a single buffer that holds the colored pixels produced by our application. Then each time the browser repaints the display, we could see its present contents.
- If we change the contents of the framebuffer during a refresh, we may see undesirable artifacts. In addition, if we are rendering more geometry than that can be rendered in a single refresh cycle, we will see different parts of objects on successive refreshes. If an object is moving, its image may be distorted on the display.

Double Buffering

- There is no coupling between when new squares are drawn into the framebuffer and when the framebuffer is redisplayed by the hardware.
- WebGL requires double buffering: front buffer and back buffer.
- The front buffer is for displaying and the back buffer is for rendering.

Double Buffering

- A typical rendering starts with a clearing of the back buffer, rendering into the back buffer, and finishing with a buffer swap.
- How and when the buffer swap is triggered?
  – In the example, we tried to use a recursive call to the render function, we failed to provide a buffer swap so we could not see a change in the display.
  – Use timer or use the function *requestAnimFrame*.

Using a Timer

- Use *setInterval* function to call the render function repeatedly.

```
setInterval(render, 16);
```

- The *render* function will be called after 16ms. An interval of 0ms will cause the render function to be executed as fast as possible. Each time the time-out function completes, it forces the buffers to be swapped, and thus we get an updated display.

Using *requestAnimFrame*

- Because *setInterval* and related function such as *setTimeout*, are independent of the browser, it can be difficult to get a smooth animation.
- One solution is to use *requestAnimFrame* function.

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  theta += 0.1;
  gl.uniform1f(thetaLoc, theta);
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
  requestAnimFrame(render);
}
```

Rotation

```
// RotatedTriangle.js
// Vertex shader program
var VSHADER_SOURCE =
 // x' = x cos b - y sin b
 // y' = x sin b + y cos b                         Equation 3.3
 // z' = z
 'attribute vec4 a_Position;\n' +
 'uniform float u_CosB, u_SinB;\n' +
 'void main() {\n' +
 '  gl_Position.x = a_Position.x * u_CosB - a_Position.y *u_SinB;\n'+
'  gl_Position.y = a_Position.x * u_SinB + a_Position.y * u_CosB;\n'+
 '  gl_Position.z = a_Position.z;\n' +
 '  gl_Position.w = 1.0;\n' +
 '}\n';
```

Rotation

```
// Rotation angle
var ANGLE = 90.0;

function main() {
   ...
  // Set the positions of vertices
  var n = initVertexBuffers(gl);
   ...
  // Pass the data required to rotate the shape to the vertex shader
  var radian = Math.PI * ANGLE / 180.0; // Convert to radians
  var cosB = Math.cos(radian);
  var sinB = Math.sin(radian);

  var u_CosB = gl.getUniformLocation(gl.program, 'u_CosB');
  var u_SinB = gl.getUniformLocation(gl.program, 'u_SinB');
  gl.uniform1f(u_CosB, cosB);
  gl.uniform1f(u_SinB, sinB);
```

Rotation with Matrix

```
// RotatedTriangle_Matrix.js
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'uniform mat4 u_xformMatrix;\n' +
  'void main() {\n' +
  '  gl_Position = u_xformMatrix * a_Position;\n' +
  '}\n';

```

Rotation with Matrix

```
 // Create a rotation matrix
var radian = Math.PI * ANGLE / 180.0; // Convert to radians
var cosB = Math.cos(radian), sinB = Math.sin(radian);

 // Note: WebGL is column major order
var xformMatrix = new Float32Array([
   cosB, sinB, 0.0, 0.0,
  -sinB, cosB, 0.0, 0.0,
   0.0,  0.0, 1.0, 0.0,
   0.0,  0.0, 0.0, 1.0
 ]);

// Pass the rotation matrix to the vertex shader
 var u_xformMatrix = gl.getUniformLocation(gl.program, 'u_xformMatrix');
 ...
gl.uniformMatrix4fv(u_xformMatrix, false, xformMatrix);
```

Rotation with Matrix

`gl.uniformMatrix4fv(location, transpose, array)`

Assign the 4×4 matrix specified by *array* to the uniform variable specified by *location*.

Parameters	location	Specifies the storage location of the uniform variable.
	Transpose	Must be `false` in WebGL.[3]
	array	Specifies an array containing a 4×4 matrix in column major order (typed array).
Return value	None	
Errors	INVALID_OPERATION	There is no current program object.
	INVALID_VALUE	*transpose* is not `false`, or the length of *array* is less than 16.