Transformations
Texture Mapping

lbg@dongseo.ac.kr

2D Transformations

Translation

Rotation

Scale

Shear

Transform

applyMatrix()
popMatrix()
printMatrix()
pushMatrix()
resetMatrix()

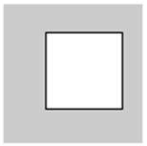
translate()
rotate()
scale()

rotateX()
rotateY()
rotateZ()
shearX()
shearY()

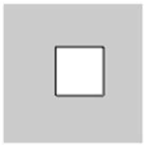
translate

Name translate()

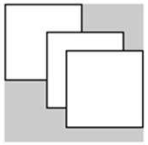
Examples



```
translate(30, 20);
rect(0, 0, 55, 55);
```



```
// Translating in 3D requires P3D
// as the parameter to size()
size(100, 100, P3D);
// Translate 30 across, 20 down, and
// 50 back, or "away" from the screen.
translate(30, 20, -50);
rect(0, 0, 55, 55);
```



```
rect(0, 0, 55, 55);  // Draw rect at original 0,0
translate(30, 20);
rect(0, 0, 55, 55);  // Draw rect at new 0,0
translate(14, 14);
rect(0, 0, 55, 55);  // Draw rect at new 0,0
```

Description Specifies an amount to displace objects within the display window. The `x` parameter specifies left/right translation, the `y` parameter specifies up/down translation, and the `z` parameter specifies translations toward/away from the screen. Using this function with the `z` parameter requires using P3D as a parameter in combination with size as shown in the above example.

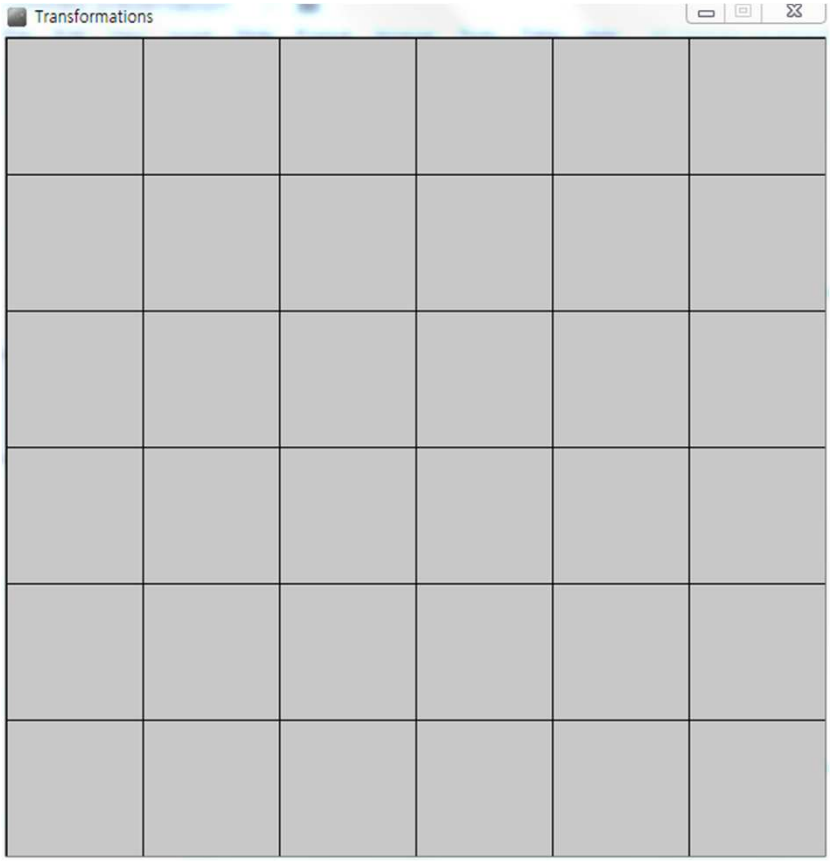
Transformations are cumulative and apply to everything that happens after and subsequent calls to the function accumulates the effect. For example, calling `translate(50, 0)` and then `translate(20, 0)` is the same as `translate(70, 0)`. If `translate()` is called within `draw()`, the transformation is reset when the loop begins again. This function can be further controlled by using `pushMatrix()` and `popMatrix()`.

Syntax

```
translate(x, y)
translate(x, y, z)
```

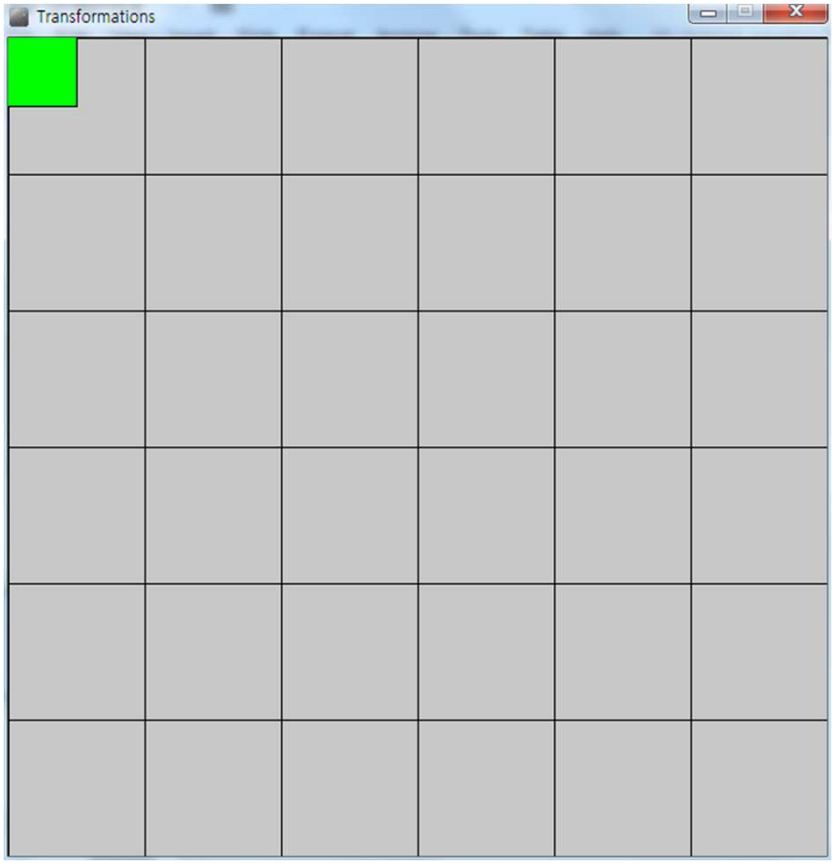
Translation

```
void setup() {
  size(600, 600);
}

void draw() {
  background(200);
  drawAxis(100);
}

void drawAxis(int w) {
  for(int x=0; x<width; x+=w)
    line(x, 0, x, height);
  for(int y=0; y<height; y+=w)
    line(0, y, width, y);
}
```



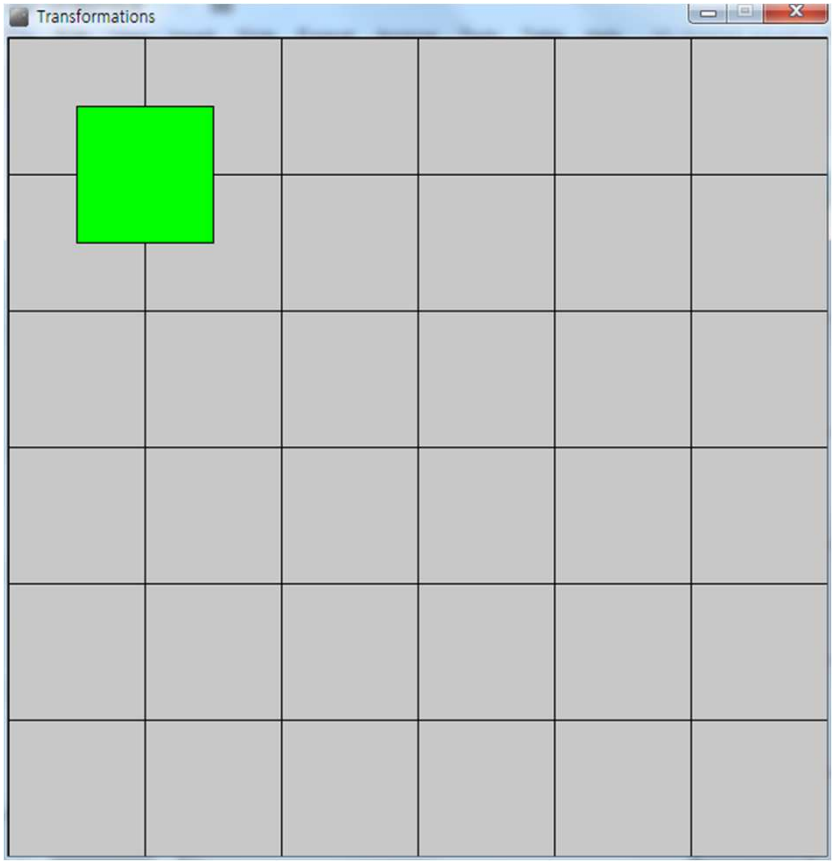
Translation

```
void setup() {
  size(600, 600);
  rectMode(CENTER);
}

void draw() {
  background(200);
  drawAxis(100);

  fill(0, 255, 0);
  rect(0, 0, 100, 100);
}

void drawAxis(int w) {
  for(int x=0; x<width; x+=w)
    line(x, 0, x, height);
  for(int y=0; y<height; y+=w)
    line(0, y, width, y);
}
```



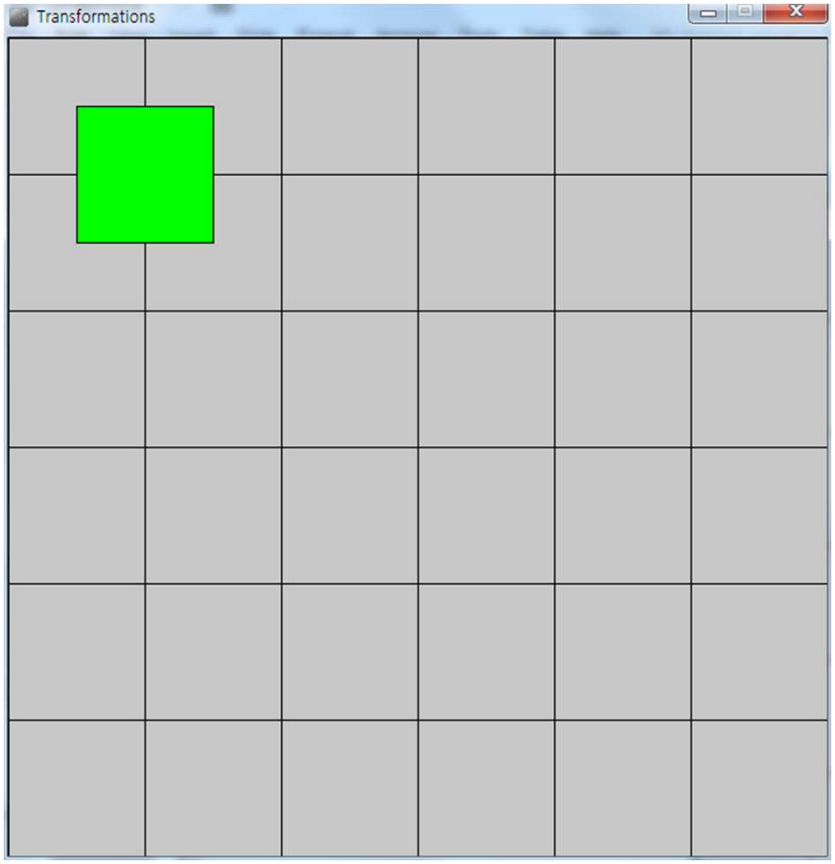
Translation

```
void setup() {
  size(600, 600);
  rectMode(CENTER);
}

void draw() {
  background(200);
  drawAxis(100);

  translate(100, 100);

  fill(0, 255, 0);
  rect(0, 0, 100, 100);
}

void drawAxis(int w) {
  for(int x=0; x<width; x+=w)
    line(x, 0, x, height);
  for(int y=0; y<height; y+=w)
    line(0, y, width, y);
}
```



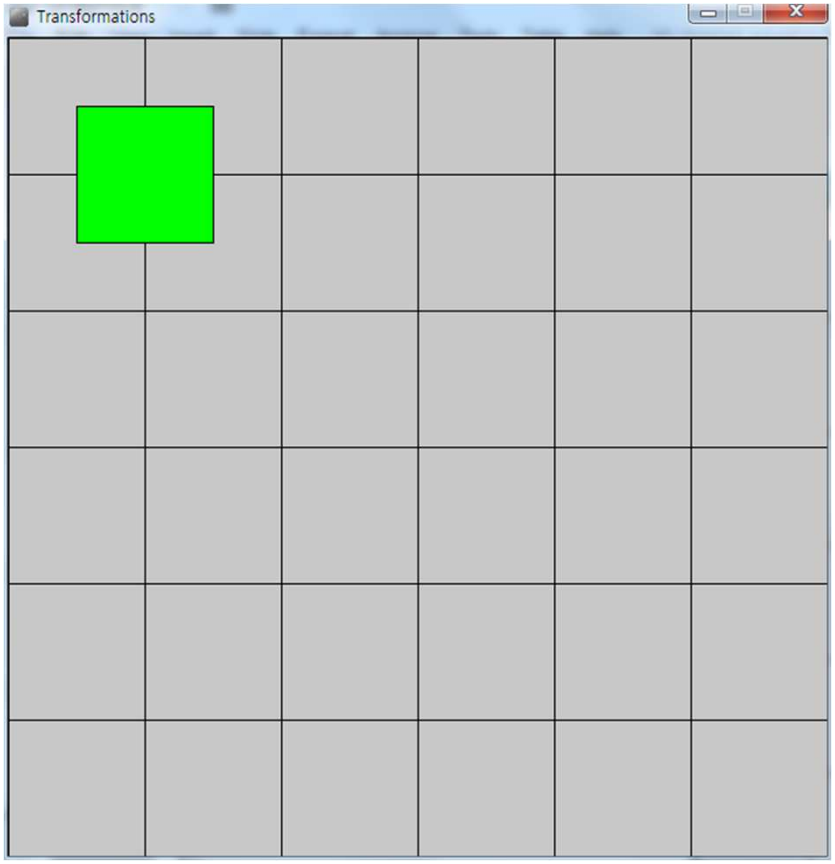
Translation

```
void setup() {
  size(600, 600);
  rectMode(CENTER);
}

void draw() {
  background(200);
  drawAxis(100);

  translate(100, 100);

  fill(0, 255, 0);
  rect(0, 0, 100, 100);
}

void drawObject(int r, int g, int b, int w, int h) {
  fill(r, g, b);
  rect(0, 0, w, h);
}

void drawAxis(int w) { … }
```



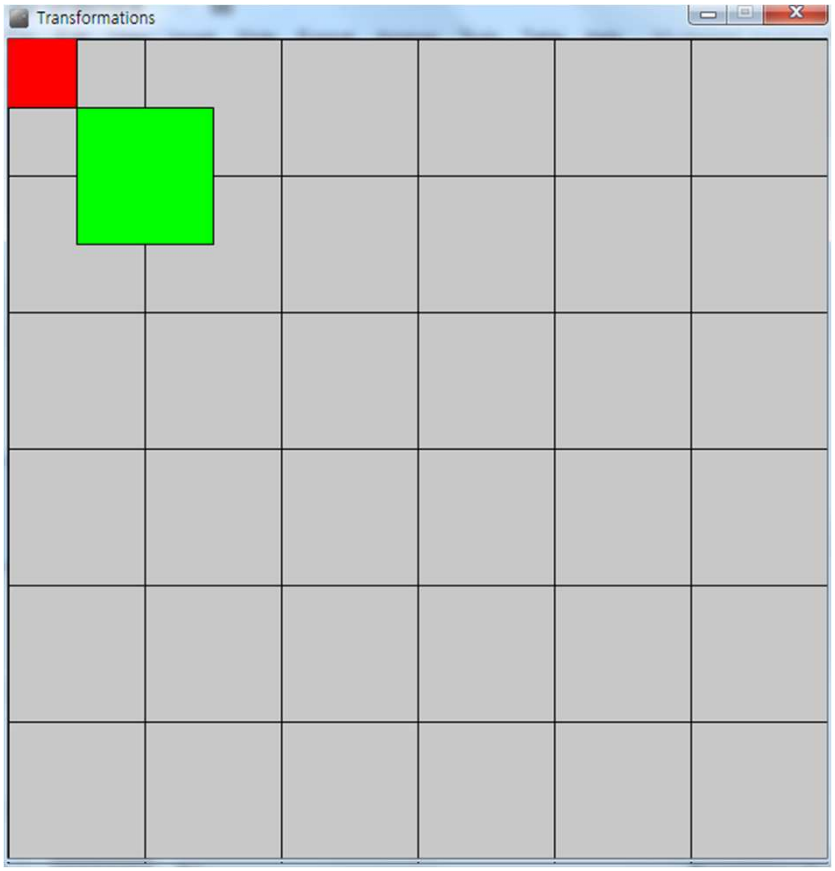
Translation

```
void setup() {
  size(600, 600);
  rectMode(CENTER);
}

void draw() {
  background(200);
  drawAxis(100);

  translate(100, 100);
  drawObject(0, 255, 0, 100, 100);
}

void drawObject(int r, int g, int b, int w, int h) {
  fill(r, g, b);
  rect(0, 0, w, h);
}

void drawAxis(int w) { … }
```



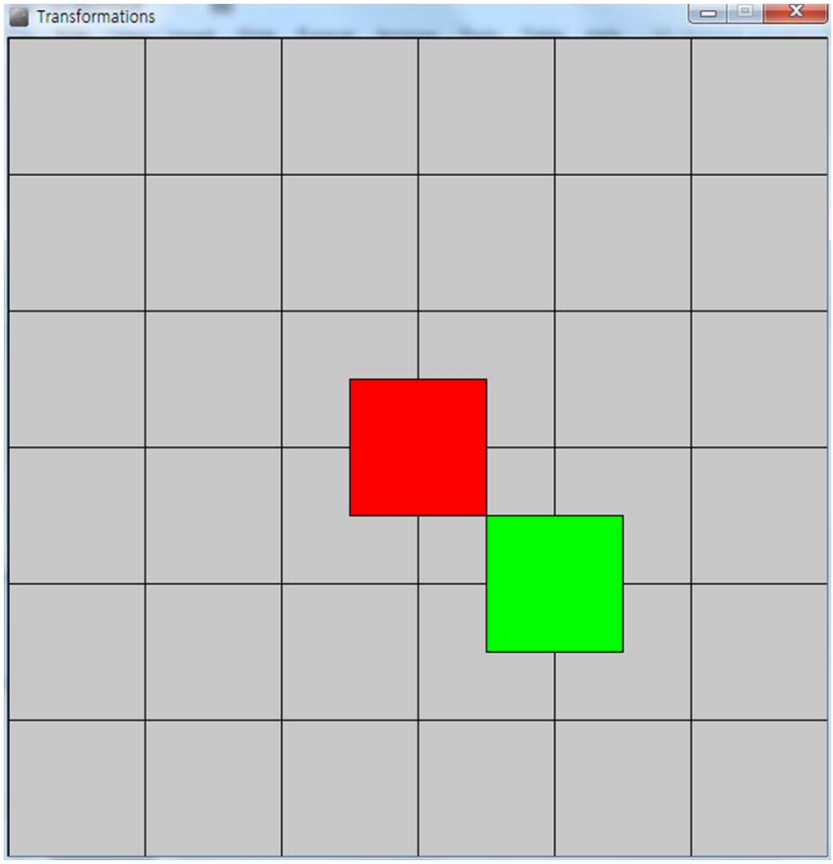
Translation

```
void setup() {
  size(600, 600);
  rectMode(CENTER);
}

void draw() {
  background(200);
  drawAxis(100);


  drawObject(255, 0, 0, 100, 100);

  translate(100, 100);
  drawObject(0, 255, 0, 100, 100);
}

void drawObject(int r, int g, int b, int w, int h) { … }

void drawAxis(int w) { … }
```



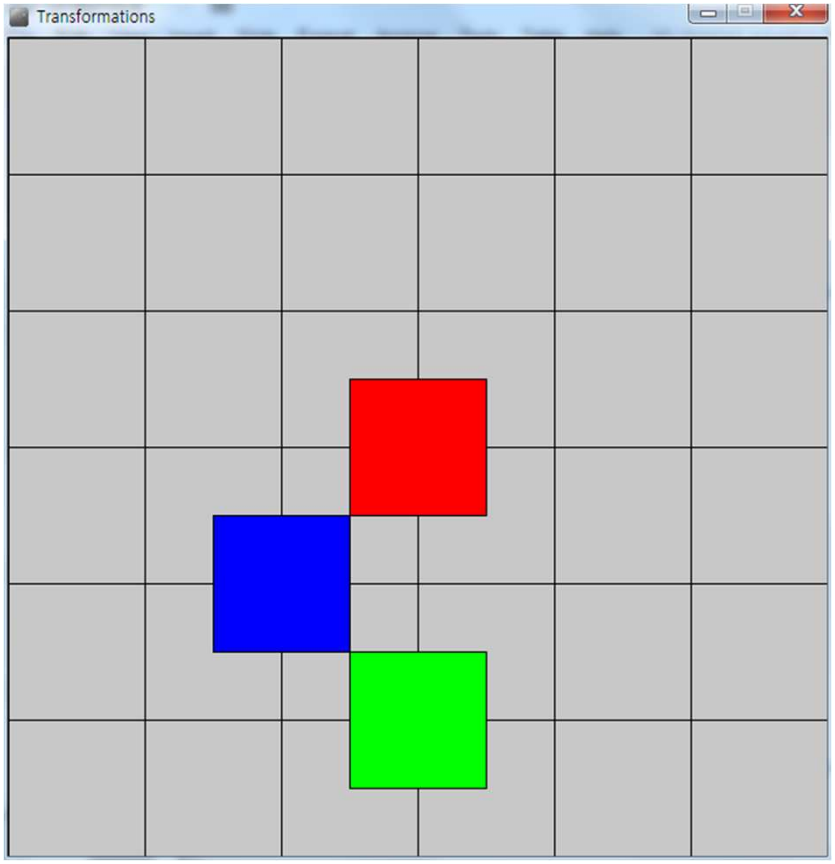
Translation

```
void setup() {
  size(600, 600);
  rectMode(CENTER);
}

void draw() {
  background(200);
  drawAxis(100);

  translate(300, 300);
  drawObject(255, 0, 0, 100, 100);

  translate(100, 100);
  drawObject(0, 255, 0, 100, 100);
}

void drawObject(int r, int g, int b, int w, int h) { … }

void drawAxis(int w) { … }
```



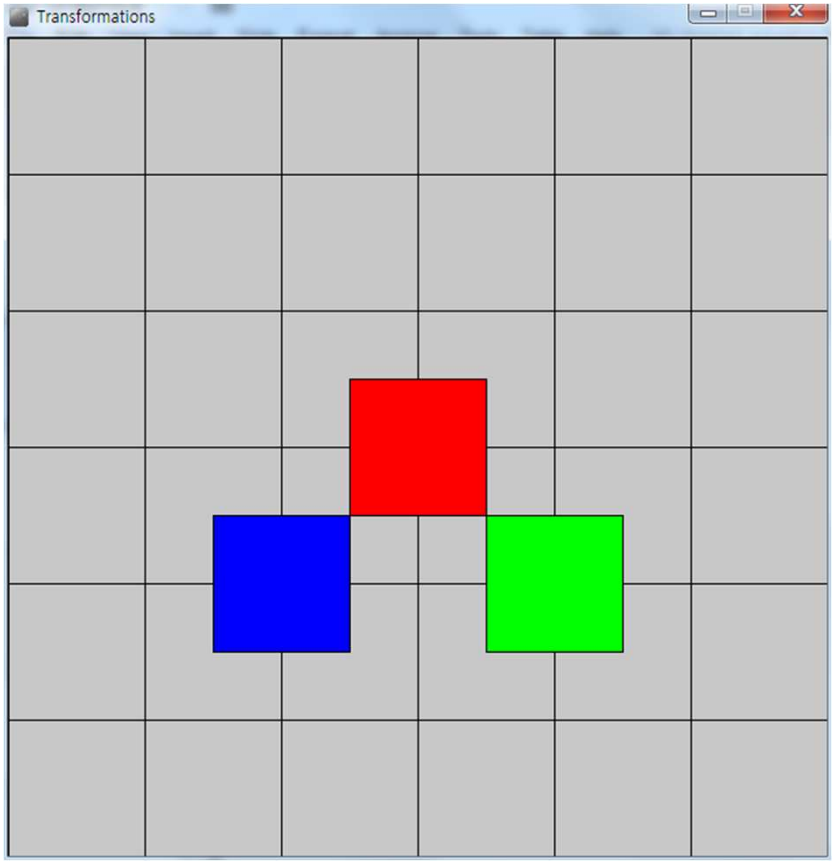
Translation

```
void setup() {
  size(600, 600);
  rectMode(CENTER);
}

void draw() {
  background(200);
  drawAxis(100);

  translate(300, 300);
  drawObject(255, 0, 0, 100, 100);

  translate(-100, 100);
  drawObject(0, 0, 255, 100, 100);

  translate(100, 100);
  drawObject(0, 255, 0, 100, 100);
}

void drawObject(int r, int g, int b, int w, int h) { … }

void drawAxis(int w) { … }
```



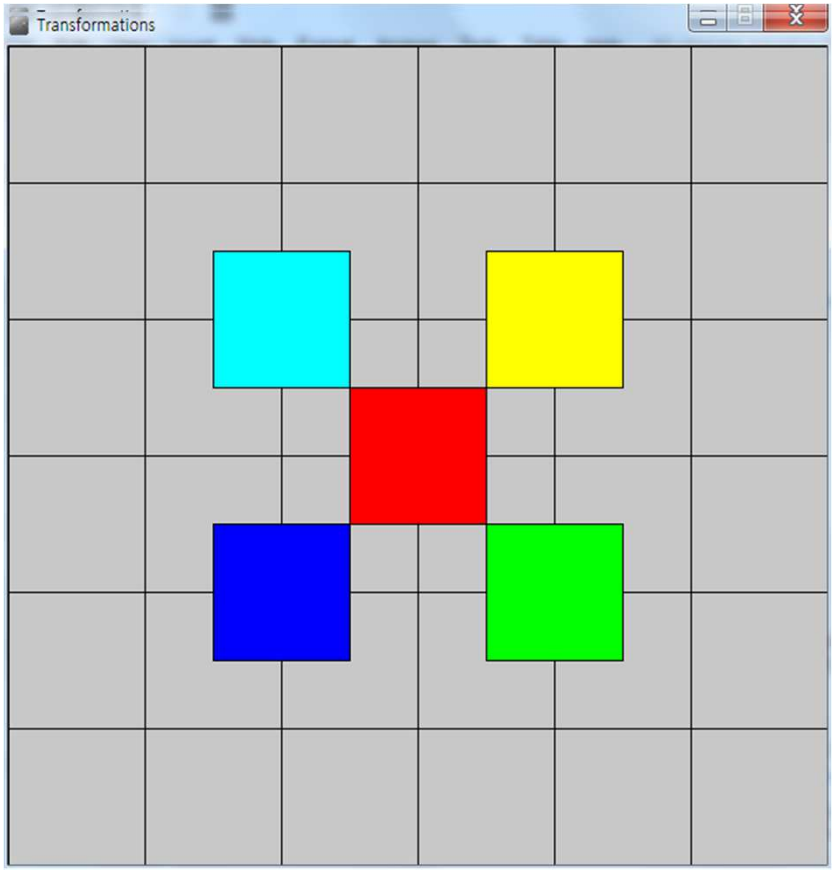
Translation

```
void setup() { … }

void draw() {
  backgrouxis(100);

  translate(300, 300);
  drawObject(255, 0, 0, 100, 100);

  pushMatrix()
    translate(-100, 100);
    drawObject(0, 0, 255, 100, 100);
  popMatrix();

  translate(100, 100);
  drawObject(0, 255, 0, 100, 100);
}

void drawObject(int r, int g, int b, int w, int h) { … }
void drawAxis(int w) { … }
```



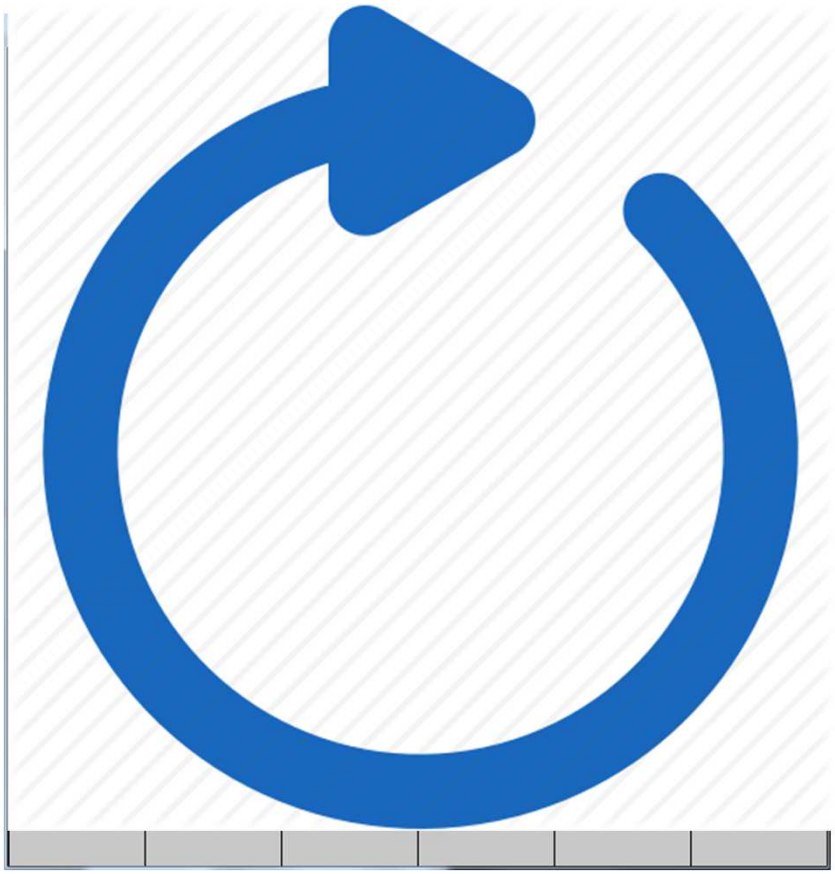
Translation

```
void setup() { … }

void draw() {
  background(200);
  drawAxis(100);

  translate(300, 300);
  drawObject(255, 0, 0, 100, 100);

  pushMatrix()
    translate(100, -100);
    drawObject(255, 255, 0, 100, 100);
  popMatrix();

  pushMatrix()
    translate(-100, -100);
    drawObject(0, 255, 255, 100, 100);
  popMatrix();

  pushMatrix()
    translation(-100, 100);
    drawObject(0, 0, 255, 100, 100);
  popMatrix();

  translation(100, 100);
  drawObject(0, 255, 0, 100, 100);
}

void drawObject(int r, int g, int b, int w, int h) { … }
void drawAxis(int w) { … }
```



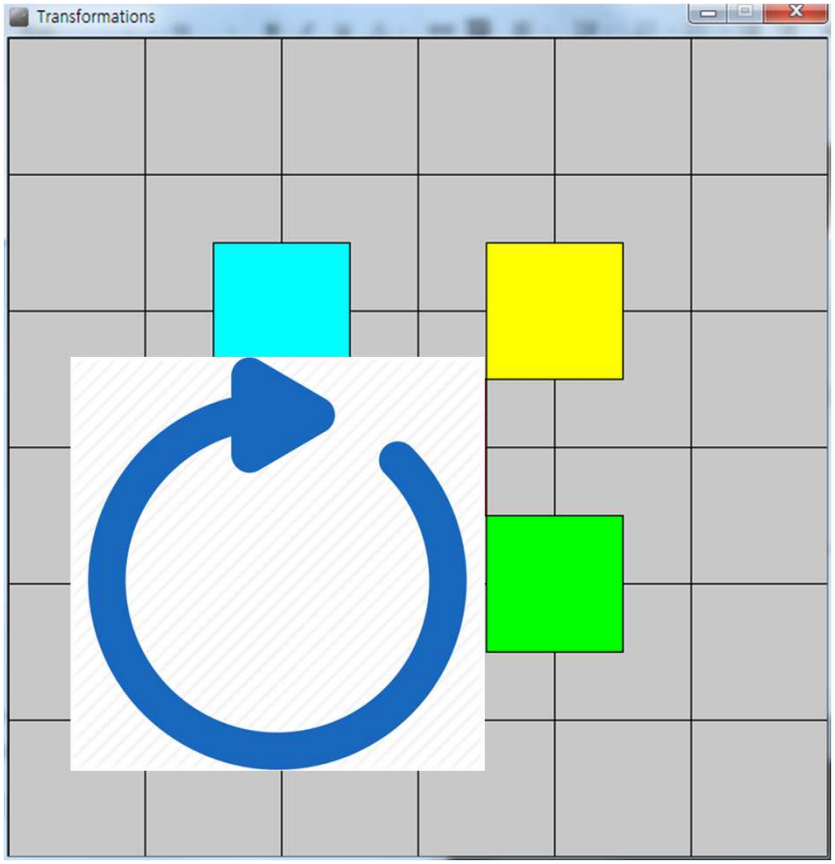
Translation

```
float theta=0.;
void setup() { … }

void draw() {
  background(200);
  drawAxis(100);

  translate(300, 300);
  rotate(theta);
  drawObject(255, 0, 0, 100, 100);

  pushMatrix()
    …
  pushMatrix()
    translate(-100, 100);
    drawObject(0, 0, 255, 100, 100);
  popMatrix();

  translate(100, 100);
  drawObject(0, 255, 0, 100, 100);
  theta+=0.01;
}

void drawObject(int r, int g, int b, int w, int h) { … }
void drawAxis(int w) { … }
```



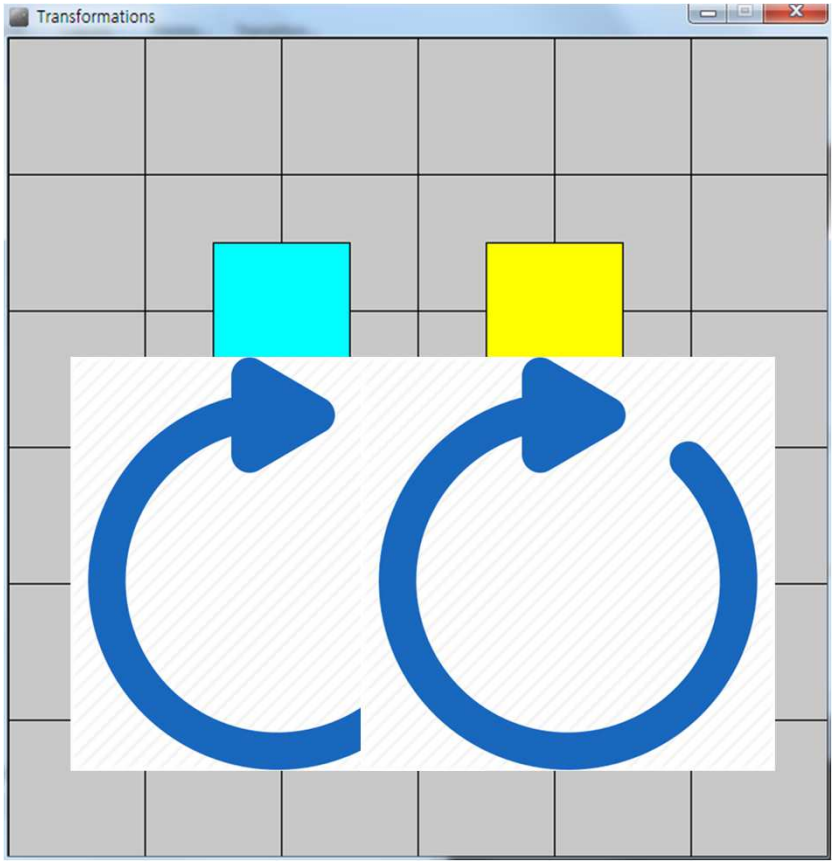
Translation

```
float theta=0.;
void setup() { … }

void draw() {
  background(200);
  drawAxis(100);

  translate(300, 300);
  drawObject(255, 0, 0, 100, 100);

  pushMatrix()
    …
  pushMatrix()
    translate(-100, 100);
    rotate(theta);
    drawObject(0, 0, 255, 100, 100);
  popMatrix();

  translate(100, 100);
  drawObject(0, 255, 0, 100, 100);
  theta+=0.01;
}

void drawObject(int r, int g, int b, int w, int h) { … }
void drawAxis(int w) { … }
```



Translation

```
float theta=0.;
void setup() { … }

void draw() {
  background(200);
  drawAxis(100);

  translate(300, 300);
  drawObject(255, 0, 0, 100, 100);

  pushMatrix()
    …
  pushMatrix()
    translate(-100, 100);
    rotate(theta);
    drawObject(0, 0, 255, 100, 100);
  popMatrix();

  translate(100, 100);
  rotate(theta)
  drawObject(0, 255, 0, 100, 100);
  theta+=0.01;
}
void drawObject(int r, int g, int b, int w, int h) { … }
void drawAxis(int w) { … }
```



https://www.processing.org/tutorials/transform2d/

The Transformation Matrix

Every time you do a rotation, translation, or scaling, the information required to do the transformation is accumulated into a table of numbers. This table, or matrix has only a few rows and columns, yet, through the miracle of mathematics, it contains all the information needed to do any series of transformations. And that's why the `pushMatrix()` and `popMatrix()` have that word in their name.

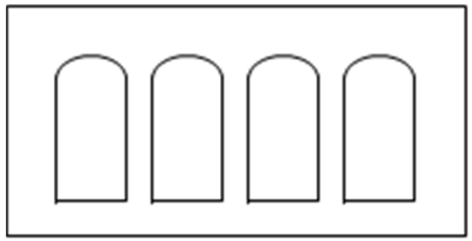
Push and Pop

What about the push and pop part of the names? These come from a computer concept known as a stack, which works like a spring-loaded tray dispenser in a cafeteria. When someone returns a tray to the stack, its weight pushes the platform down. When someone needs a tray, he takes it from the top of the stack, and the remaining trays pop up a little bit.
In a similar manner, `pushMatrix()` puts the current status of the coordinate system at the top of a memory area, and `popMatrix()` pulls that status back out. The preceding example used `pushMatrix()` and `popMatrix()` to make sure that the coordinate system was "clean" before each part of the drawing. In all of the other examples, the calls to those two functions weren't really necessary, but it doesn't hurt anything to save and restore the grid status.
Note: in Processing, the coordinate system is restored to its original state (origin at the upper left of the window, no rotation, and no scaling) every time that the `draw()` function is executed.

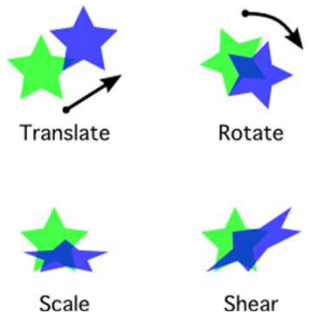
Introduction to transformations

Transformations are useful for:

- Composing a more complex object using many instances of a single form.
- Creating a complex object from a single "motif".
- Moving an object to get a different view of it.
- Moving an object to animate it.



Mathematics Notation



in 3D, let P = (P_x, P_y, P_z, 1)

P can be *transformed* into Q = (Q_x, Q_y, Q_z, 1) by applying a *transformation* T

$$Q = T(P)$$

There are an infinite number of possible transformations

- Some transformations are rather simple
- Some transformations are rather complex
- Some transformations distort the shape of an object
- Some transformations maintain the relative shape of an object

Affine Transformations

There is a special class of transformations that maintain the relative shape of objects

(the *affine transforms*).

All affine transformations are *linear*. Therefore, they can be written as:

$$Q_x = m_{11}P_x + m_{12}P_y + m_{13}P_z + m_{14}$$
$$Q_y = m_{21}P_x + m_{22}P_y + m_{23}P_z + m_{24}$$
$$Q_z = m_{31}P_x + m_{32}P_y + m_{33}P_z + m_{34}$$
$$1 = 1$$

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Standard affine transformations

- Translation
- Scaling
- Rotation
- Shearing

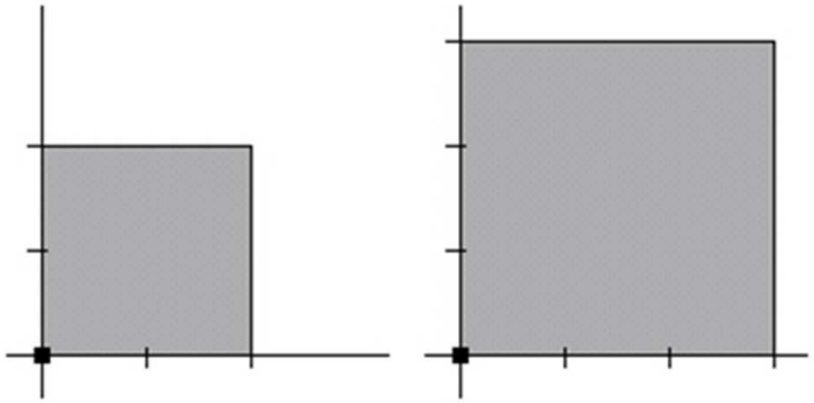
Translation

- a displacement
- a move in a linear direction
- it is not affected by the location of the origin

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Scaling

- to enlarge (or shrink)
- a point at the origin is never affected by scaling
- objects always scale about the origin



$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Rotation

- to move in a circular direction
- to rotate about the origin
- the angle is always considered a counter-clockwise rotation.
- derivation:

$Q_x = r\cos(q+f)$

$= r(\cos(q)\cos(f) - \sin(q)\sin(f))$

$= r\cos(q)\cos(f) - r\sin(q)\sin(f)$

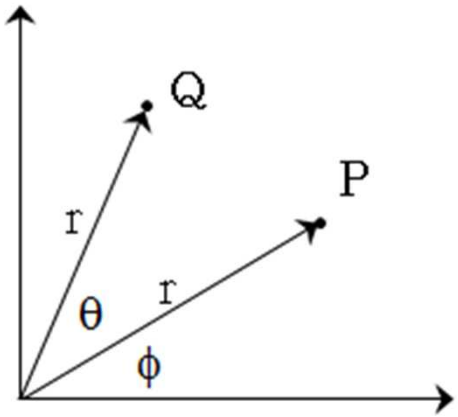
$= P_x\cos(q) - P_y\sin(q)$

$Q_y = r\sin(q+f)$

$= r(\sin(q)\cos(f) + \cos(q)\sin(f))$

$= r\sin(q)\cos(f) + r\cos(q)\sin(f)$

$= P_x\sin(q) + P_y\cos(q)$



Rotation

- Rotation about Z

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

- Rotation about Y

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

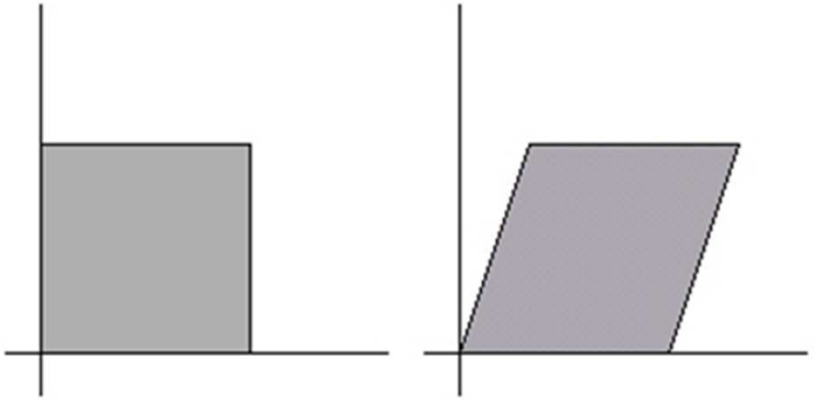
- Rotation about X

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

where c = cos(angle), s = sin(angle)

Shearing

- to shear in x: ($P_x + hP_y$, P_y, P_z); slant vertically
- to shear in y: (P_x, gP_x+P_y, P_z); slant horizontally



$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ g & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

resetMatrix

- The resetMatrix function replaces the current matrix with the identity matrix.
- All transformations are reset when draw() begins again.
- The equivalent function in OpenGL is glLoadIdentity().

translate

Specifies an amount to displace objects within the display window. The x parameter specifies left/right translation, the y parameter specifies up/down translation, and the z parameter specifies translations toward/away from the screen. Using this function with the z parameter requires using P3D as a parameter in combination with size as shown in the above example.

Transformations are cumulative and apply to everything that happens after and subsequent calls to the function accumulates the effect. For example, calling translate(50, 0) and then translate(20, 0) is the same as translate(70, 0). If translate() is called within draw(), the transformation is reset when the loop begins again. This function can be further controlled by using pushMatrix() and popMatrix().

Syntax

translate(x, y), translate(x, y, z)

Parameters

x float: left/right translation,
y float: up/down translation,
z float: forward/backward translation

rotate

Rotates the amount specified by the angle parameter. Angles must be specified in radians (values from 0 to TWO_PI), or they can be converted from degrees to radians with the radians() function.

The coordinates are always rotated around their relative position to the origin. Positive numbers rotate objects in a clockwise direction and negative numbers rotate in the couter clockwise direction. Transformations apply to everything that happens afterward, and subsequent calls to the function compound the effect. For example, calling rotate(PI/2.0) once and then calling rotate(PI/2.0) a second time is the same as a single rotate(PI). All transformations are reset when draw() begins again.
Technically, rotate() multiplies the current transformation matrix by a rotation matrix. This function can be further controlled by pushMatrix() and popMatrix().

Syntax

rotate(angle)

Parameters

angle float: angle of rotation specified in radians

scale

Increases or decreases the size of a shape by expanding and contracting vertices. Objects always scale from their relative origin to the coordinate system. Scale values are specified as decimal percentages. For example, the function call scale(2.0) increases the dimension of a shape by 200%.
Transformations apply to everything that happens after and subsequent calls to the function multiply the effect. For example, calling scale(2.0) and then scale(1.5) is the same as scale(3.0). If scale() is called within draw(), the transformation is reset when the loop begins again. Using this function with the z parameter requires using P3D as a parameter for size(), as shown in the third example above. This function can be further controlled with pushMatrix() and popMatrix().

Syntax

scale(s)
scale(x, y)
scale(x, y, z)

Parameters

s float: percentage to scale the object

Specifics

Given a sequence of elementary transformations that together form a single complex transformation,
if you change the order of the elementary transformations, you get a different complex transformation

There is typically more than one sequence of transformations that will produce the same effect,
but often one sequence is easier to create than the others.

The multiplication of the matrices is always (CT*m), not (m*CT),
so the order of the function calls must be in reverse order
from the "conceptual" ordering of the transformations.

always (CT*m), not (m*CT),

Specifics

For the translate, rotate, and scale functions,
each builds a 4x4 transformation matrix using the supplied arguments
and then multiplies it times the CT (current transform).
For example, the translate function might look something like the following:

```
void translate(float dx, float dy, float dz) {
  float m[4][4] =
  {{ 1.0, 0.0, 0.0, dx },
   { 0.0, 1.0, 0.0, dy },
   { 0.0, 0.0, 1.0, dz },
   { 0.0, 0.0, 0.0, 1.0}};
  CT = CT*m; // pseudocode
}
```

Matrix Stack

Problem : Matrix multiplication is computationally expensive.

The multiplication of two 4x4 matrices requires 64 multiples and 48 additions

Solution : Avoid duplicate matrix multiplication by saving transformations that will be needed again.

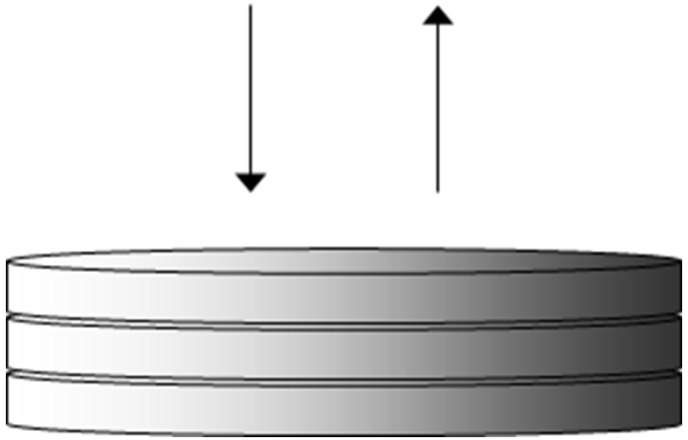
A stack is a good way to store a series of transformations for later use

OpenGL implementation

OpenGL maintains a stack that can hold a minimum of 32 matrices.
The matrix at the top of the stack is always the CT (Current Transform)

The following two commands manipulate the matrix stack:

```
pushMatrix();
popMatrix();
```



OpenGL implementation

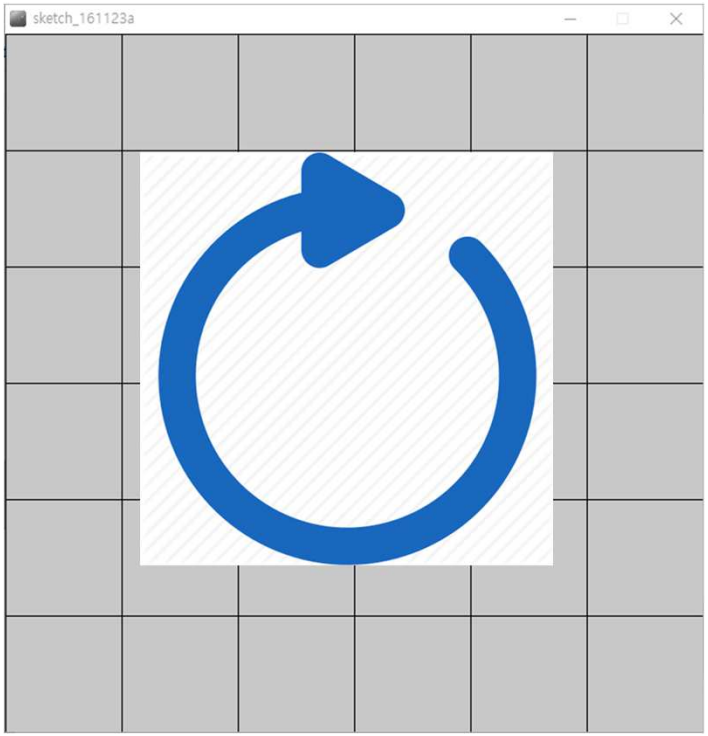
Assuming that the stack is implemented as an array, these two functions look like the following pseudocode:

```
// The stack begins with an Identity matrix on top
int stackTop = 1;
Matrix4x4   Identity = {{1, 0, 0, 0},{0, 1, 0, 0},{0, 0, 1, 0},{0, 0, 0, 1}};
Matrix4x4    MatrixStack[32] = {Identity, {0}};

void pushMatrix(void) {
  MatrixStack[stackTop] = MatrixStack[stackTop-1];
  stackTop++;
}

void popMatrix(void) {
  stackTop--;
}
 // The CT (current transform) is always MatrixStack[stacktop-1]
```

Matrix Stack

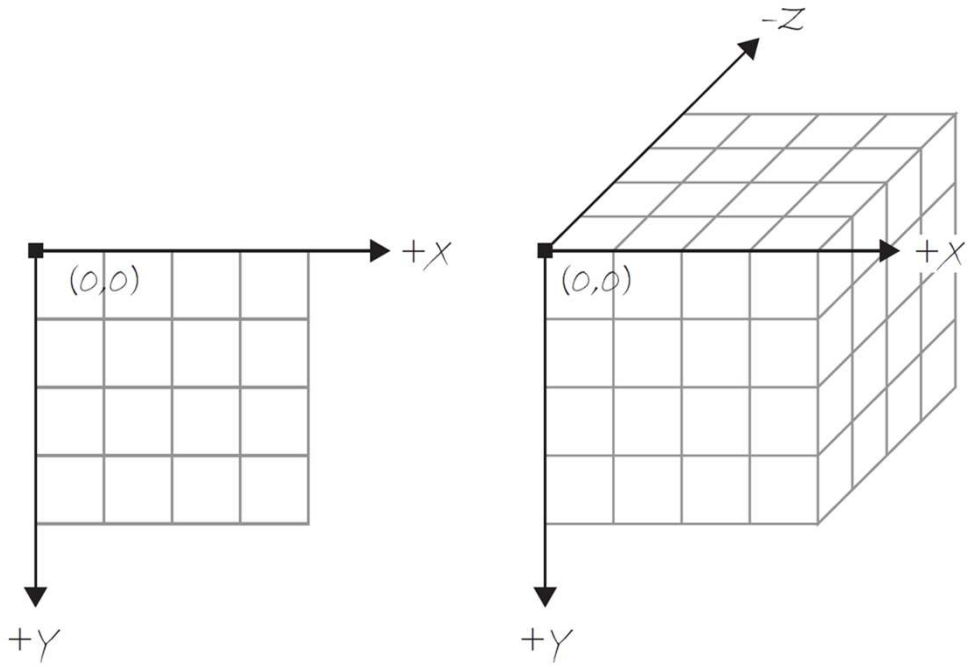
Statements	stack[0]	stack[1]

resetMatrix()
background(200);
drawAxis(100);
 	
translate(300, 300);
rotate(theta);
fill(0, 255, 0);
rect(0, 0, 100, 100);
 	
fill(0, 0, 255);
pushMatrix();
translate(-100, 0);
ellipse(0, 0, 50, 50);
popMatrix();
 	
translate(100, 0);
ellipse(0, 0, 50, 50);
theta+=0.01;
	M_0=I
	M_0
	M_0
	
	M_0=M_0*$T_{300,300}$
	M_0=M_0*R_{theta}
	M_0
	M_0
	
	M_0
	
	M_0
	
	M_0=M_0*$T_{100,0}$
	M_0
	M_0
		
		M_1=M_0
		M_1=M_1*$T_{-100,0}$
		M_1



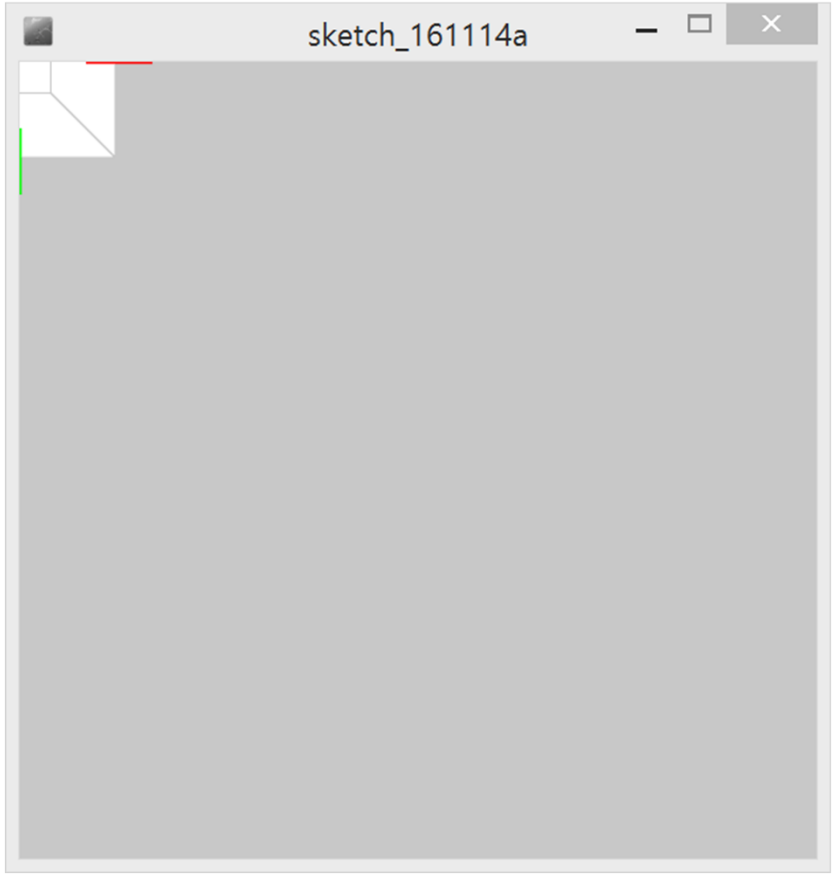
3D Coordinate System in Processing

size(500, 500, P3D);



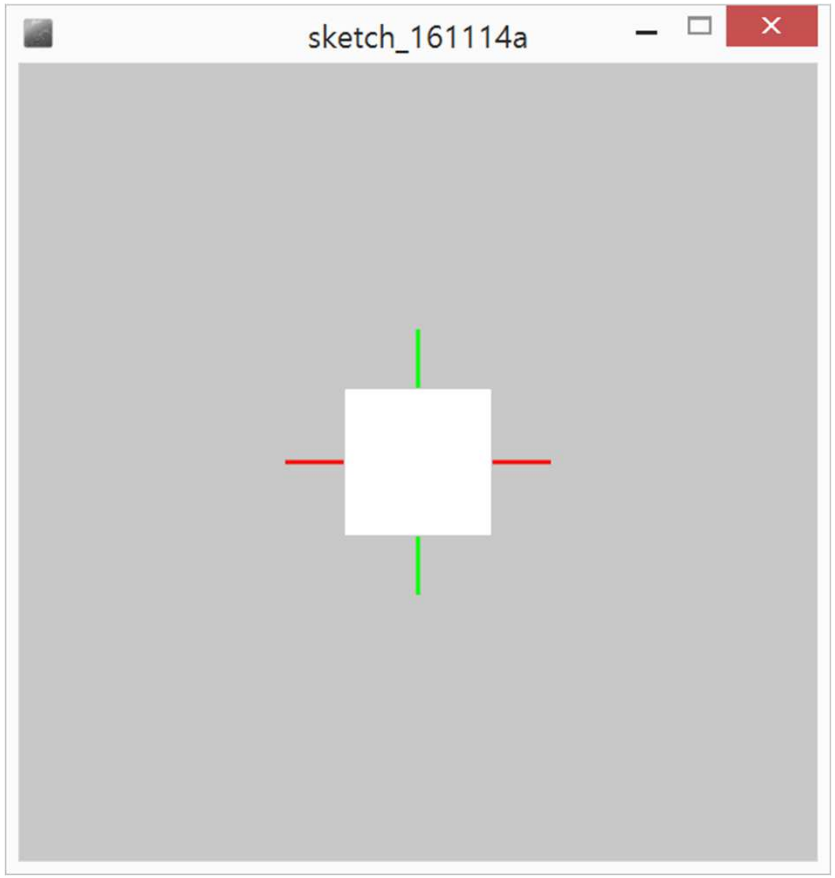
```
void setup() {
  size(600, 600, P3D);
}

void draw() {
  background(200);

  drawAxis(100);
  strokeWeight(1);
  stroke(200);
  box(100);
}
```

```
void drawAxis(int w) {
  strokeWeight(3);
  stroke(255, 0, 0);
  line(-w, 0, 0, w, 0, 0);
  stroke(0, 255, 0);
  line(0, -w, 0, 0, w, 0);
  stroke(0, 0, 255);
  line(0, 0, -w, 0, 0, w);
}
```



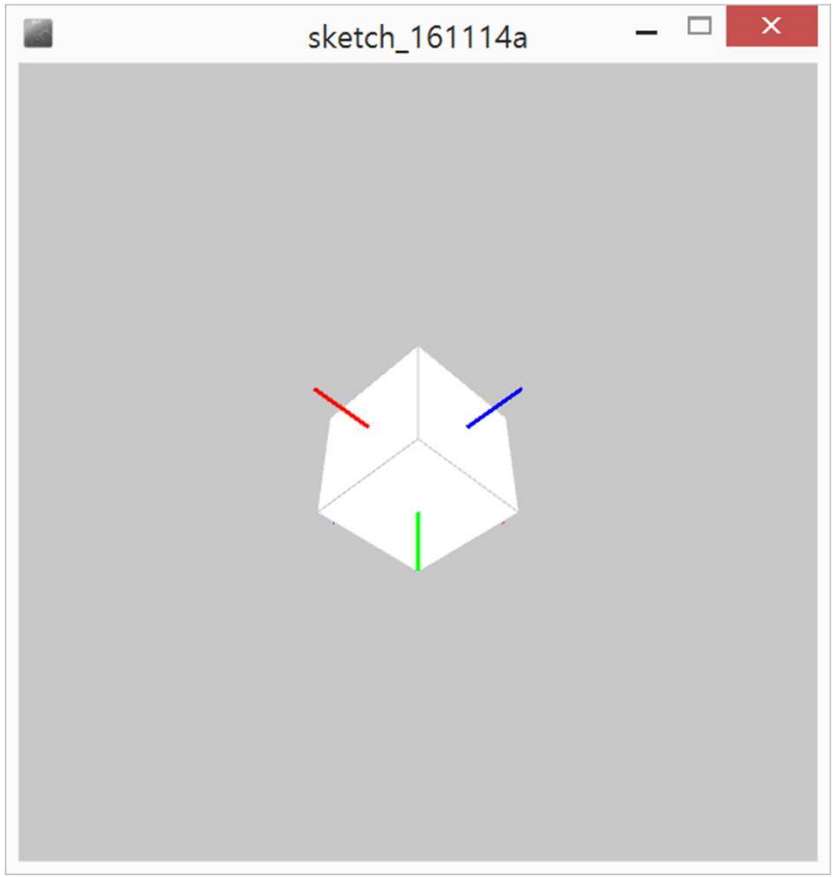
```
void setup() {
  size(600, 600, P3D);
}

void draw() {
  background(200);
  translate(width/2, height/2);

  drawAxis(100);
  strokeWeight(1);
  stroke(200);
  box(100);
}
```

```
void drawAxis(int w) {
  strokeWeight(3);
  stroke(255, 0, 0);
  line(-w, 0, 0, w, 0, 0);
  stroke(0, 255, 0);
  line(0, -w, 0, 0, w, 0);
  stroke(0, 0, 255);
  line(0, 0, -w, 0, 0, w);
}
```



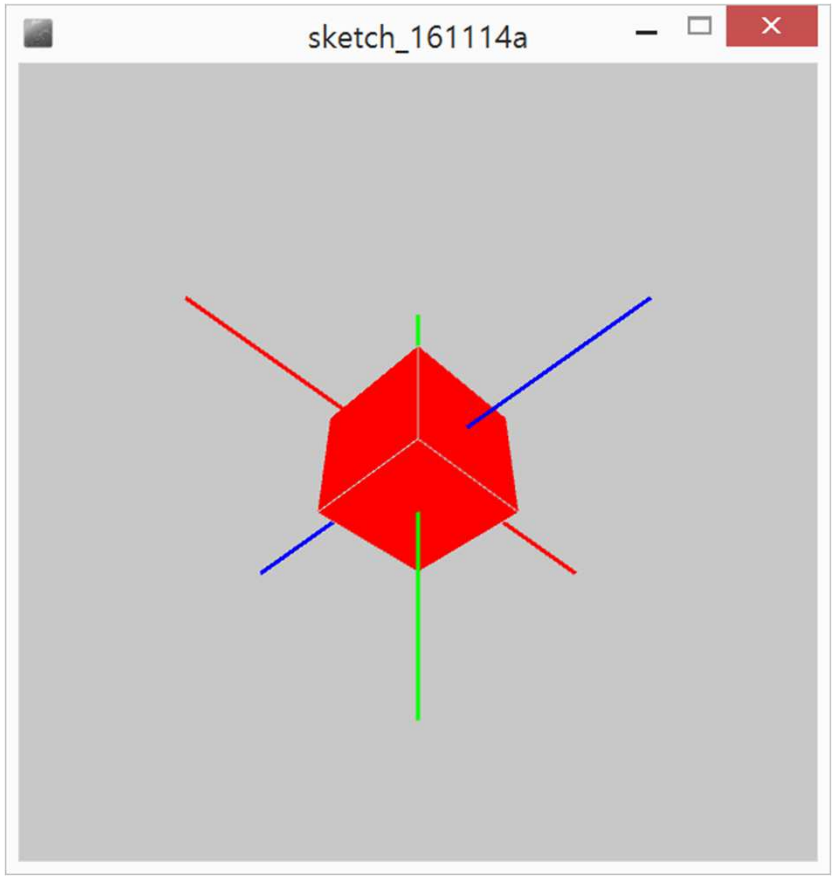
```
float rotx = PI/4;
float roty = PI/4;

void setup() {
  size(600, 600, P3D);
}

void draw() {
  background(200);
  translate(width/2, height/2);
  rotateX(rotx);
  rotateY(roty);

  drawAxis(100);
  strokeWeight(1);
  stroke(200);
  box(100);
}
```

```
void drawAxis(int w) {
  strokeWeight(3);
  stroke(255, 0, 0);
  line(-w, 0, 0, w, 0, 0);
  stroke(0, 255, 0);
  line(0, -w, 0, 0, w, 0);
  stroke(0, 0, 255);
  line(0, 0, -w, 0, 0, w);
}

void mouseDragged() {
  float rate = 0.01;
  rotx += (pmouseY-mouseY) * rate;
  roty += (mouseX-pmouseX) * rate;
}
```



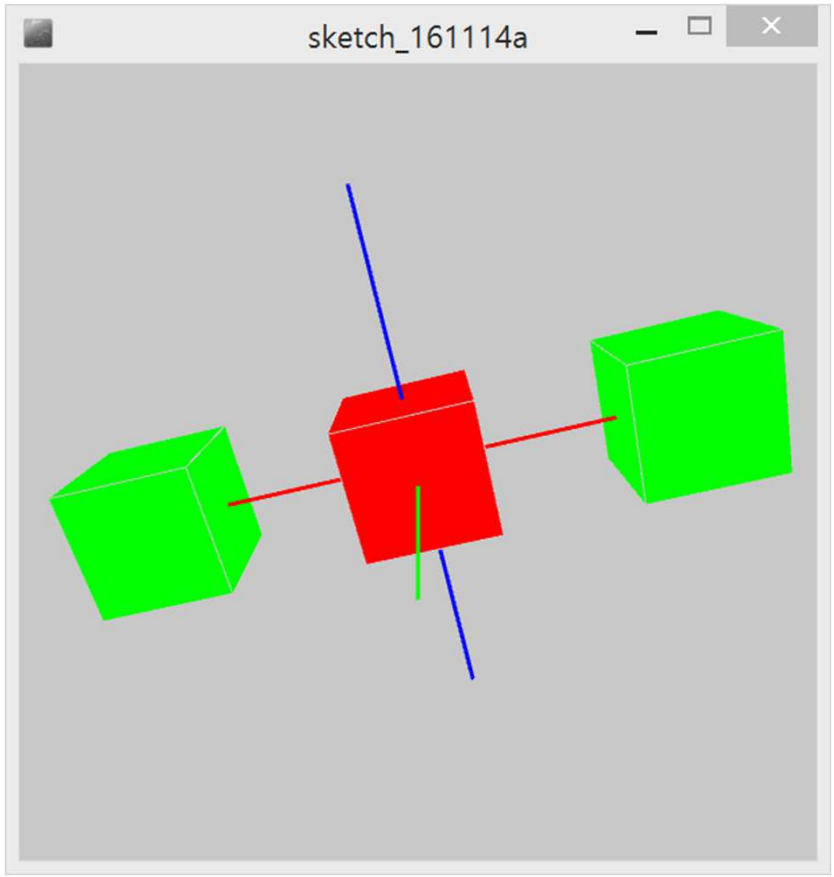
```
float rotx = PI/4;
float roty = PI/4;

void setup() {
  size(600, 600, P3D);
}

void draw() {
  background(200);
  translate(width/2, height/2);
  rotateX(rotx);
  rotateY(roty);

  drawAxis(200);
  drawBox(255, 0, 0, 100);
}
```

```
void drawBox(int r, int g, int b, int w) {
  strokeWeight(1);
  stroke(200);
  fill(r, g, b);
  box(w);
}

void mouseDragged() { … }

void drawAxis(int w) { … }
```

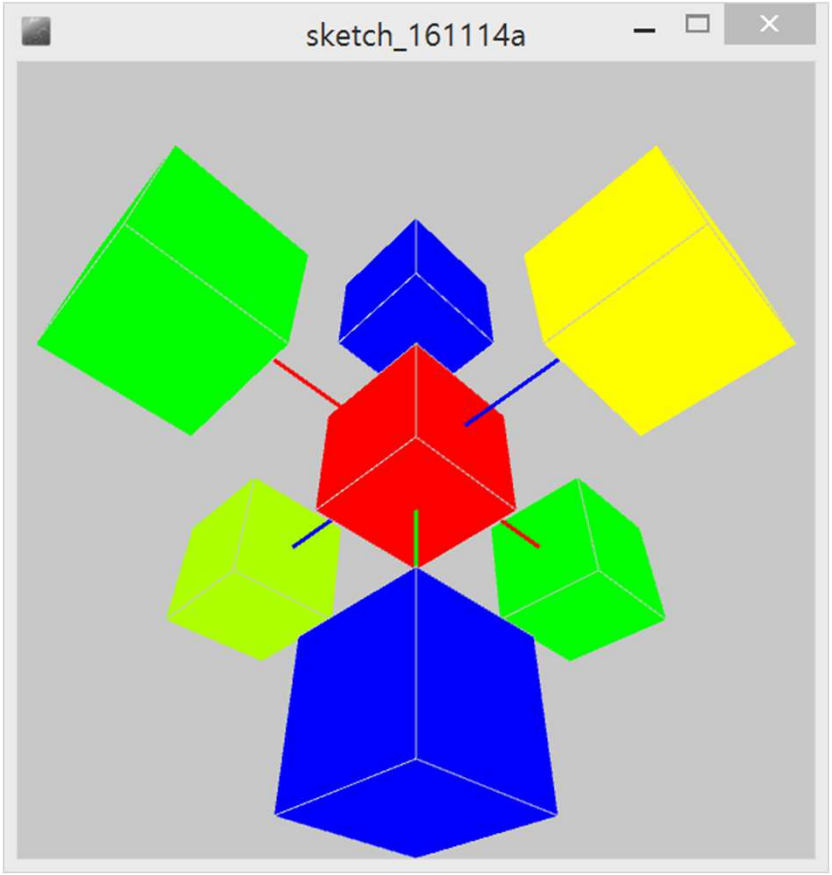


```
float rotx ..

void setup() { .. }

void draw() {
  background(200);
  translate(width/2, height/2);
  rotateX(rotx);
  rotateY(roty);

  drawAxis(200);
  drawBox(255, 0, 0, 100);

  pushMatrix();
    translate(-200, 0, 0);
    drawBox(0, 255, 0, 100);
  popMatrix();

  translate(200, 0, 0);
  drawBox(0, 255, 0, 100);
}
```

```
void mouseDragged() { … }
void drawAxis(int w) { … }
void drawBox(int r, int g, int b, int w) { … }
```

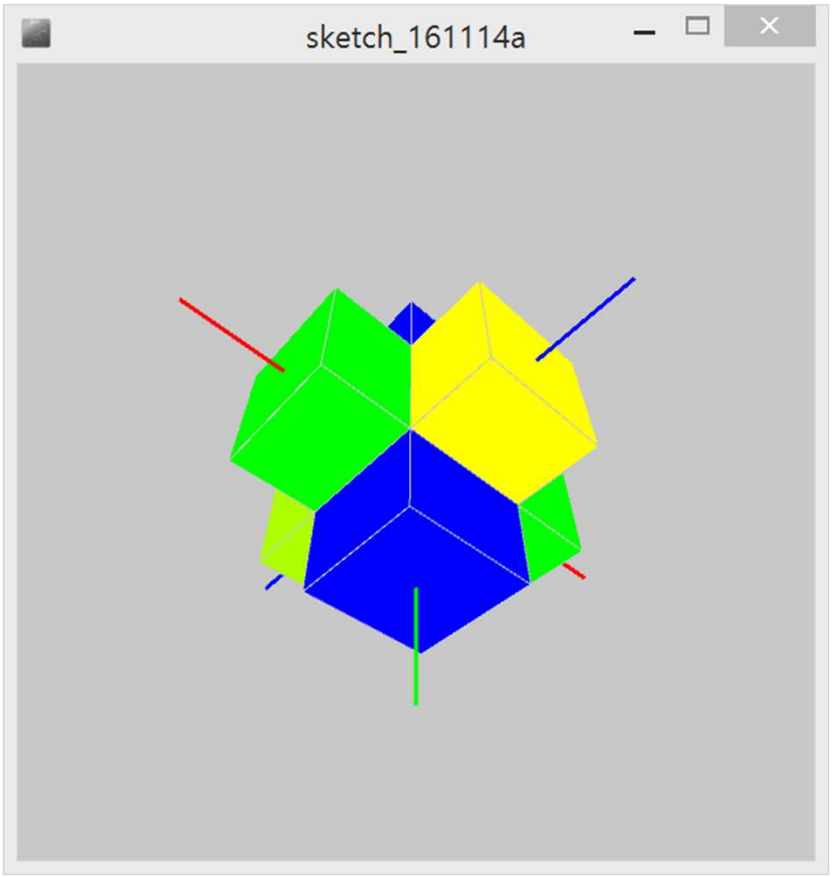


```
float rotx ..
void setup() { .. }

void draw() {
  background(200);
  translate(width/2, height/2);
  rotateX(rotx);
  rotateY(roty);

  drawAxis(200);
  drawBox(255, 0, 0, 100);

  pushMatrix();
    translate(-100, 0, 0);
    drawBox(0, 255, 0, 100);
  popMatrix();

  pushMatrix();
    translate(100, 0, 0);
    drawBox(0, 255, 0, 100);
  popMatrix();

  pushMatrix();
    translate(0, -100, 0);
    drawBox(0, 0, 255, 100);
  popMatrix();

  pushMatrix();
    translate(0, 100, 0);
    drawBox(0, 0, 255, 100);
  popMatrix();

  pushMatrix();
    translate(0, 0, -100);
    drawBox(255, 255, 0, 100);
  popMatrix();

  pushMatrix();
    translate(0, 0, 100);
    drawBox(255, 255, 0, 100);
  popMatrix();
}
void mouseDragged() { … }
void drawAxis(int w) { … }
void drawBox(int r, int g, int b, int w) { … }
```



```
float rotx ..
float dist=0.;
float step=0.1;
void setup() { .. }

void draw() {
  background(200);
  translate(width/2, height/2);
  rotateX(rotx);
  rotateY(roty);

  drawObject(dist);
  dist+=step;
  if(dist>=200) step=-0.1;
  else if(dist<=0) step=0.1;
}
```

```
void drawObject(float d) {
  drawAxis(200);
  drawBox(255, 0, 0, 100);

  pushMatrix();
    translate(-d, 0, 0);
    drawBox(0, 255, 0, 100);
  popMatrix();

  pushMatrix();
    translate(d, 0, 0);
    drawBox(0, 255, 0, 100);
  popMatrix();

  pushMatrix();
   translate(0, -d, 0);
    drawBox(0, 0, 255, 100);
  popMatrix();

  pushMatrix();
    translate(0, d, 0);
    drawBox(0, 0, 255, 100);
  popMatrix();

  pushMatrix();
    translate(0, 0, -d);
    drawBox(0, 255, 255, 100);
  popMatrix();

  pushMatrix();
    translate(0, 0, d);
    drawBox(0, 255, 255, 100);
  popMatrix();
}
```




```
void mouseDragged() { … }
void drawAxis(int w) { … }
void drawBox(int r, int g, int b, int w) { … }
```

Projection Transformations

- Converts objects defined in 3D space into objects defined in 2D space.
- There are two basic types of 3D to 2D projects:

  - **orthographic** : a parallel project of 3D values onto a 2D plane;
  useful for engineering drawings

  - **perspective** : the way your eye sees the normal world around you;
  used for animation and visual simulation

camera

Examples

```
size(100, 100, P3D);
	noFill();
	background(204);
	camera(70.0, 35.0, 120.0, 50.0, 50.0, 0.0, 0.0, 1.0, 0.0);
	translate(50, 50, 0);
	rotateX(-PI/6);
	rotateY(PI/3);
	box(45);
```

Description

Sets the position of the camera through setting the eye position, the center of the scene, and which axis is facing upward. Moving the eye position and the direction it is pointing (the center of the scene) allows the images to be seen from different angles. The version without any parameters sets the camera to the default position, pointing to the center of the display window with the Y axis as up. The default values are `camera(width/2.0, height/2.0, (height/2.0) / tan(PI*30.0 / 180.0), width/2.0, height/2.0, 0, 0, 1, 0)`. This function is similar to `gluLookAt()` in OpenGL, but it first clears the current camera settings.

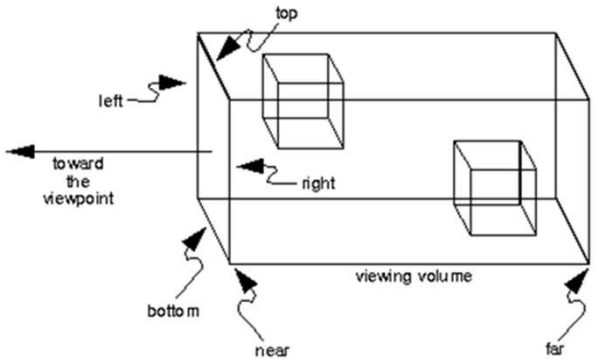
Syntax

```
camera()
	camera(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)
```

ortho

Examples

```
size(100, 100, P3D);
noFill();
ortho(-width/2, width/2, -height/2, height/2); // Same as ortho()
translate(width/2, height/2, 0);
rotateX(-PI/6);
rotateY(PI/3);
box(45);
```



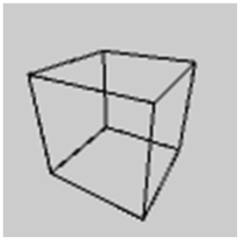
Description

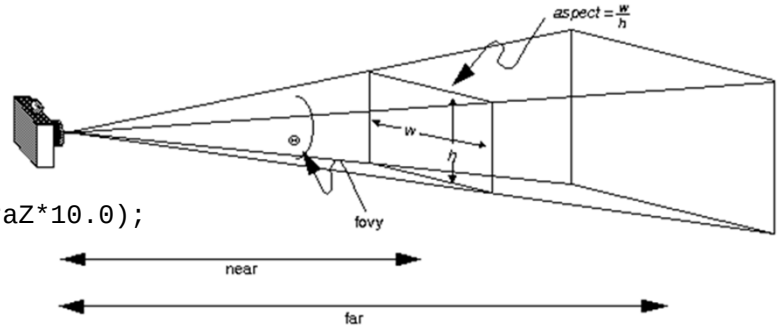
Sets an orthographic projection and defines a parallel clipping volume. All objects with the same dimension appear the same size, regardless of whether they are near or far from the camera. The parameters to this function specify the clipping volume where left and right are the minimum and maximum x values, top and bottom are the minimum and maximum y values, and near and far are the minimum and maximum z values. If no parameters are given, the default is used: ortho(-width/2, width/2, -height/2, height/2).

Syntax

```
ortho()
ortho(left, right, bottom, top)
ortho(left, right, bottom, top, near, far)
```

perspective





Examples

```
// Re-creates the default perspective
        size(100, 100, P3D);
        noFill();
        float fovy = PI/3.0;
        float cameraZ = (height/2.0)/tan(fovy/2.0);
        perspective(fovy, float(width)/float(height), cameraZ/10.0, cameraZ*10.0);
        translate(50, 50, 0);
        rotateX(-PI/6);
        rotateY(PI/3);
        box(45);
```

Description

Sets a perspective projection applying foreshortening, making distant objects appear smaller than closer ones. The parameters define a viewing volume with the shape of truncated pyramid. Objects near to the front of the volume appear their actual size, while farther objects appear smaller. This projection simulates the perspective of the world more accurately than orthographic projection. The version of perspective without parameters sets the default perspective and the version with four parameters allows the programmer to set the area precisely. The default values are: perspective(PI/3.0, width/height, cameraZ/10.0, cameraZ*10.0) where cameraZ is ((height/2.0) / tan(PI*60.0/360.0));

Syntax

```
perspective()
perspective(fovy, aspect, zNear, zFar)
```

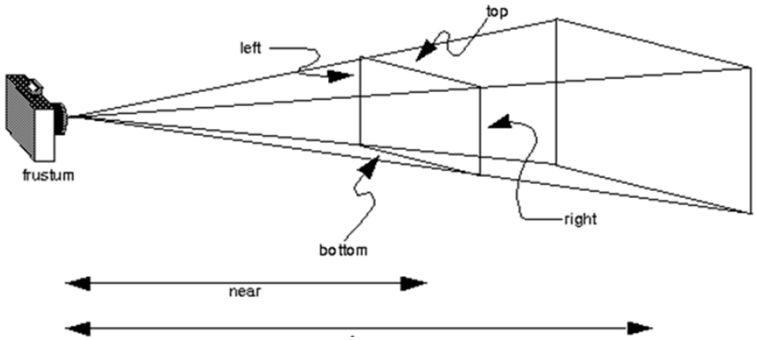
Parameters

`fovy` float: field-of-view angle (in radians) for vertical direction
`aspect` float: ratio of width to height
`zNear` float: z-position of nearest clipping plane
`zFar` float: z-position of farthest clipping plane

frustum

Examples

```
size(100, 100, P3D);
	noFill();
	background(204);
	frustum(-10, 0, 0, 10, 10, 200);
	rotateY(PI/6);
	box(45);
```



Description

Sets a perspective matrix as defined by the parameters.

A frustum is a geometric form: a pyramid with its top cut off. With the viewer's eye at the imaginary top of the pyramid, the six planes of the frustum act as clipping planes when rendering a 3D view. Thus, any form inside the clipping planes is rendered and visible; anything outside those planes is not visible.

Setting the frustum has the effect of changing the perspective with which the scene is rendered. This can be acheived more simply in many cases by using **perspective()**.

Note that the near value must be greater than zero (as the point of the frustum "pyramid" cannot converge "behind" the viewer). Similarly, the far value must be greater than the near value (as the "far" plane of the frustum must be "farther away" from the viewer than the near plane).

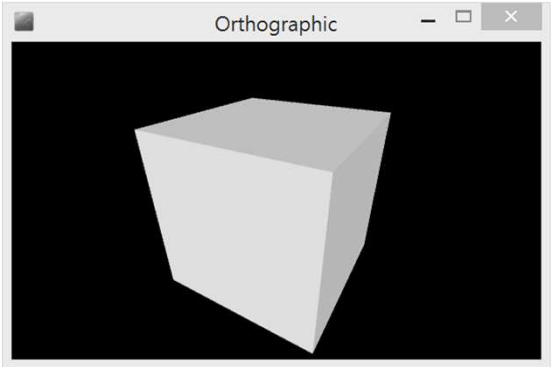
Works like glFrustum, except it wipes out the current perspective matrix rather than multiplying itself with it.

Syntax

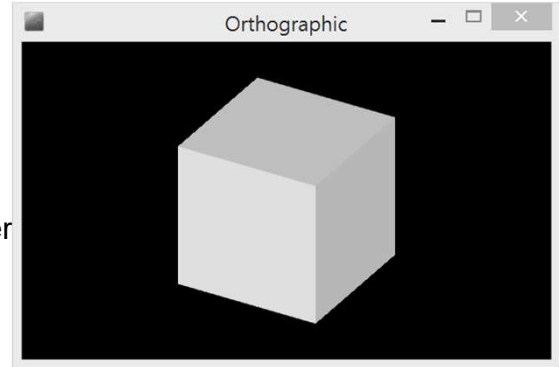
```
frustum(left, right, bottom, top, near, far)
```

Perspective vs. Ortho

```
/**
 * Perspective vs. Ortho
 *
 * Move the mouse left to right to change the "far"
 * parameter for the perspective() and ortho() functions.
 * This parameter sets the maximum distance from the
 * origin away from the viewer and will clip the geometry.
 * Click a mouse button to switch between the perspective and
 * orthographic projections.
 */


boolean showPerspective = false;

void setup() {
  size(600, 360, P3D);
  noFill();
  fill(255);
  noStroke();
}
```

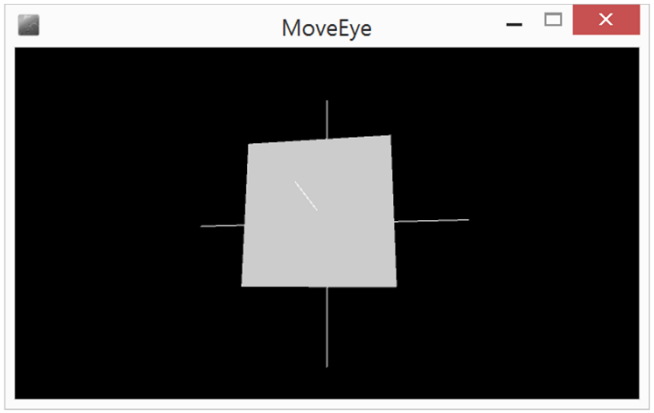



```
void draw() {
  lights();
  background(0);
  float far = map(mouseX, 0, width, 120, 400);
  if (showPerspective == true) {
    perspective(PI/3.0, float(width)/float(height), 10, far);
  } else {
    ortho(0, width, 0, height, 10, far);
  }
  translate(width/2, height/2, 0);
  rotateX(-PI/6);
  rotateY(PI/3);
  box(180);
}

void mousePressed() {
  showPerspective = !showPer
}
```



Camera lifts up

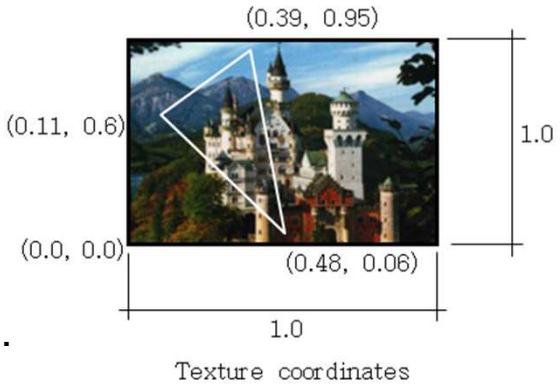
```
// Move Eye. by Simon Greenwold.
// The camera lifts up (controlled by mouseY) while looking at the same point.
void setup() {
  size(640, 360, P3D);
  fill(204);
}

void draw() {
  lights();
  background(0);
  camera(30.0, mouseY, 220.0, // eyeX, eyeY, eyeZ
         0.0, 0.0, 0.0, // centerX, centerY, centerZ
         0.0, 1.0, 0.0); // upX, upY, upZ
  noStroke();
  box(90);
  stroke(255);
  line(-100, 0, 0, 100, 0, 0);
  line(0, -100, 0, 0, 100, 0);
  line(0, 0, -100, 0, 0, 100);
}
```



Texture Mapping

- Definition : A 2-dimensional (2D) texture map is an image that is applied to one side of a 3-dimensional polygon.

- Problems :
  The image and the 3D polygon typically do not have the same shape (most polygons are not rectangles).
  The image and the 3D polygon typically do not have the same size



- Solution :
  specify the *portion* of the image that is to be used to "paint" the polygon.
  Since we need to specify a subset of the whole image,
  it is natural to use percentages of the image dimensions.
  The percentages are specified as values between 0 (0%) and 1.0 (100%).

texture

Examples

```
size(100, 100, P3D);
	noStroke();
	PImage img = loadImage("laDefense.jpg");
	beginShape();
	texture(img);
	vertex(10, 20, 0, 0);
	vertex(80, 5, 100, 0);
	vertex(95, 90, 100, 100);
	vertex(40, 95, 0, 100);
	endShape();
```

Description

Sets a texture to be applied to vertex points. The `texture()` function must be called between `beginShape()` and `endShape()` and before any calls to `vertex()`. This function only works with the P2D and P3D renderers.

When textures are in use, the fill color is ignored. Instead, use `tint()` to specify the color of the texture as it is applied to the shape.

Syntax

```
texture(image)
```

Parameters

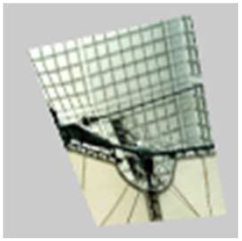
`Image PImage : reference to a PImage object`

textureMode

Examples

```
size(100, 100, P3D);
noStroke();
PImage img = loadImage("laDefense.jpg");
textureMode(IMAGE);
beginShape();
texture(img);
vertex(10, 20, 0, 0);
vertex(80, 5, 100, 0);
vertex(95, 90, 100, 100);
vertex(40, 95, 0, 100);
endShape();
```

```
size(100, 100, P3D);
noStroke();
PImage img = loadImage("laDefense.jpg");
textureMode(NORMAL);
beginShape();
texture(img);
vertex(10, 20, 0, 0);
vertex(80, 5, 1, 0);
vertex(95, 90, 1, 1);
vertex(40, 95, 0, 1);
endShape();
```



Description

Sets the coordinate space for texture mapping. The default mode is `IMAGE`, which refers to the actual coordinates of the image. `NORMAL` refers to a normalized space of values ranging from 0 to 1. This function only works with the P2D and P3D renderers.

With `IMAGE`, if an image is 100 x 200 pixels, mapping the image onto the entire size of a quad would require the points (0,0) (100, 0) (100,200) (0,200). The same mapping in `NORMAL` is (0,0) (1,0) (1,1) (0,1).

Syntax

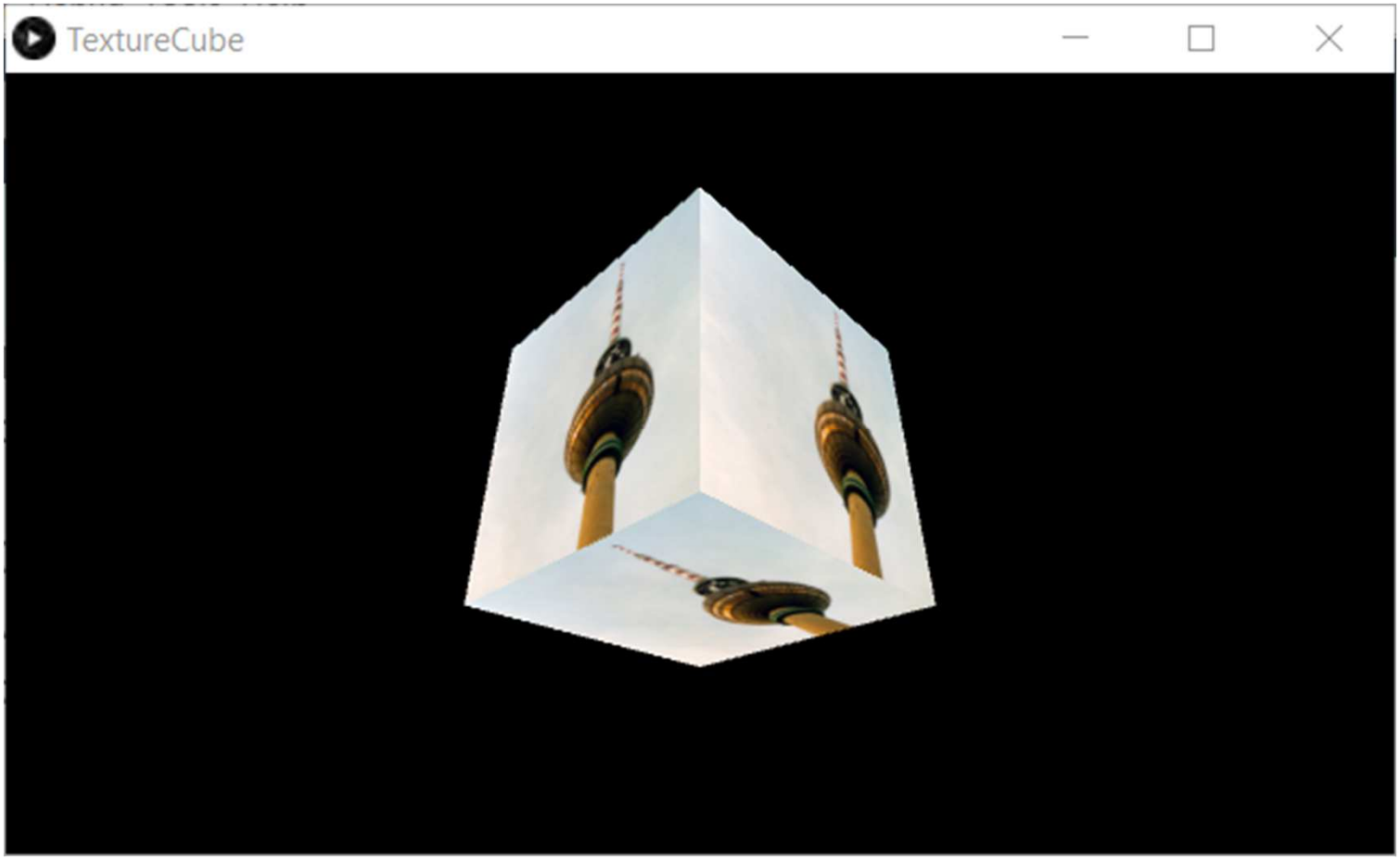
```
textureMode(mode)
mode int: either IMAGE or NORMAL
```

```
PImage img;
void setup() {
  size(300, 300, P2D);
  img = loadImage("berlin-1.jpg");
  textureMode(NORMAL);
}

void draw() {
  background(0);
  translate(width/2, height/2);
  rotate(map(mouseX, 0, width, -PI, PI));
  if (mousePressed) {
    textureWrap(REPEAT);
  } else {
    textureWrap(CLAMP);
  }
  beginShape();
  texture(img);
  vertex(-100, -100, 0, 0);
  vertex(100, -100, 2, 0);
  vertex(100, 100, 2, 2);
  vertex(-100, 100, 0, 2);
  endShape();
}
```

TexturedCube

```
PImage tex;
float rotx = PI/4;
float roty = PI/4;

void setup() {
  size(640, 360, P3D);
  tex = loadImage("berlin-1.jpg");
  textureMode(NORMAL);
  fill(255);
  stroke(color(44,48,32));
}

void draw() {
  background(0);
  noStroke();
  translate(width/2.0, height/2.0, -100);
  rotateX(rotx);
  rotateY(roty);
  scale(90);
  TexturedCube(tex);
}
```

```
void TexturedCube(PImage tex) {
  beginShape(QUADS);
  texture(tex);

  // Given one texture and six faces, we can easily set up the uv coordinates such that four of the faces
  // tile "perfectly" along either u or v, but the other two faces cannot be so aligned.
  // This code tiles "along" u, "around" the X/Z faces and fudges the Y faces - the Y faces are arbitrarily
  // aligned such that a rotation along the X axis will put the "top" of either texture at the "top"
  // of the screen, but is not otherwised aligned with the X/Z faces. (This just affects what type of symmetry
  // is required if you need seamless tiling all the way around the cube)

  // +Z "front" face
  vertex(-1, -1,  1, 0, 0);
  vertex( 1, -1,  1, 1, 0);
  vertex( 1,  1,  1, 1, 1);
  vertex(-1,  1,  1, 0, 1);

  // -Z "back" face
  vertex( 1, -1, -1, 0, 0);
  vertex(-1, -1, -1, 1, 0);
  vertex(-1,  1, -1, 1, 1);
  vertex( 1,  1, -1, 0, 1);

  // -X "left" face
  vertex(-1, -1, -1, 0, 0);
  vertex(-1, -1,  1, 1, 0);
  vertex(-1,  1,  1, 1, 1);
  vertex(-1,  1, -1, 0, 1);
  endShape();
}

void mouseDragged() {
  float rate = 0.01;
  rotx += (pmouseY-mouseY) * rate;
  roty += (mouseX-pmouseX) * rate;
}
```


TextureCube