CS 106 Winter 2016
Craig S. Kaplan

Module 05

Geometric Context

Topics

- Using the `translate()`, `rotate()` and `scale()` functions to manipulate geometric context
- Using `pushMatrix()` and `popMatrix()` to preserve (and discard) context
- Combining multiple transformations, order of operations
- Nesting transformations and hierarchical modelling
- Special effects from iterated transformations

Readings

- *Learning Processing,* Sections 14.1, 14.5, 14.7, 14.8, 14.9

Introduction

Let's begin with a simple programming task that will demonstrate the benefits of "geometric context". You're asked to create a sketch that draws a simple house in the middle of the sketch window. After a bit of thought, you come up with the following code.

```
void setup()
{
  size( 500, 500 );
}

void drawHouse()
{
  fill( #BFB375 );
  rect( 150, 200, 200, 150 );

  fill( #3E362F );
  triangle(
    250, 100, 120, 200, 380, 200 );
}

void draw()
{
  background( 255 );
  drawHouse();
}
```

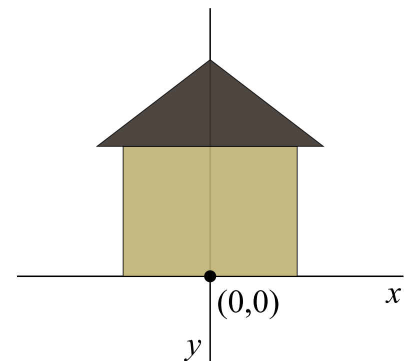
Everything's great, until you're asked to move the house slightly to the left. That small change forces you to rewrite every line of code that has coordinates in it. That's both frustrating (there could be many lines of code affected) and confusing (did you remember to change *all three* X coordinates in the call to `triangle()`? And who knows if the person asking will be satisfied—maybe they'll want to move it yet again.

Of course, with a bit of programming skill we can take an important conceptual leap. Let's add arguments to the `drawHouse()` function that allow the house to be moved around.

```
void setup()
{
  size( 500, 500 );
}

void drawHouse( float tx, float ty )
{
  // Facade
  fill( #BFB375 );
  rect( 150+tx, 200+ty, 200, 150 );

  // Roof
  fill( #3E362F );
  triangle(
    250+tx, 100+ty,
    120+tx, 200+ty,
    380+tx, 200+ty );
}

void draw()
{
  background( 255 );
  drawHouse( -20, 0 );
}
```

That's definitely better for our picky customer: the exact position of the house is determined in just one place, and can easily be adjusted with low risk of introducing bugs. Plus, there's another big benefit that we get immediately: we can easily draw multiple houses by adding just one line of code each time.

Once we accept that the "caller" of `drawHouse()` will decide where to put the house, doesn't it make more sense to build the house in a more convenient coordinate system? Instead of being forced to think about the coordinates of the window where the house will be drawn, I'd prefer to envision it in a coordinate system that's convenient for house-design purposes.



```
void setup()
{
  size( 500, 500 );
}

void drawHouse( float tx, float ty )
{
  // Facade
  fill( #BFB375 );
  rect( -100+tx, -150+ty, 200, 150 );

  // Roof
  fill( #3E362F );
  triangle(
    tx, -250+ty,
    -130+tx, -150+ty,
    130+tx, -150+ty );
}

void draw()
{
  background( 255 );
  drawHouse( 0.5*width, 0.75*height );
}
```

This code has the advantage that the “position” of the house (i.e., the values we pass in as arguments to `drawHouse()`) is defined relative to a useful part of the house drawing itself: the bottom centre. That makes it easier to visualize where we’re putting it. And don’t worry, we’ll simplify the code further soon.

What if the picky customer also wanted to *scale* the house? Well, we can certainly imagine adding a scaling argument to `drawHouse()`, to go along with the translation arguments. But the code is going to get very messy very quickly. But the real problems will start when we try to incorporate rotation as well:

- For starters, every function in the style of drawHouse() will require five parameters: two for translation, two for scaling, and one for rotation. That gets old very fast.
- Every time we want to place a point in the plane, we’ll have to transform from “object coordinates” to

"world coordinates", which will require a lot more typing and be highly error-prone.
- Computing rotations is difficult—it's all sines and cosines, and we certainly want to avoid peppering our code with those.
- Rotated objects are much harder to describe. A rotated rectangle is still a polygon. But what's a rotated ellipse?

Geometric context

The solution, which is a classic idea from computer graphics, is to introduce *geometric context*. For our purposes, a geometric context will consist of the sequence of transformations that we plan to apply to every object that we're drawing.

The first smart thing we do is to make the current geometric context a global variable. It's not a variable you define or see, it's hiding somewhere within Processing. And it's not really something you need to think about explicitly. The context is part of the program's overall state. The second smart idea ties in with the first one: every drawing function, like `ellipse()`, `rect()`, `line()`, vertex(), and so on, knows about the current context and uses it to transform objects before they're drawn on the screen.

It turns out that geometric context can be represented very compactly and elegantly using a *matrix* (a 2D array of numbers). You may have encountered matrices if you ever took a linear algebra course. But you definitely don't need to know about them for this course.

This setup neatly separates transformation and geometry. You can set up a current transformation by altering the geometric context, and then draw objects without having to track things like translation manually. The trick is that you have a greater responsibility as a programmer to have a mental model of what the context is and how it changes.

Enough philosophizing. There are three main transformation functions that alter geometric context: `translate()`, `rotate()`, and `scale()`. The `translate()` function takes two arguments, which act like `tx` and `ty` in the examples above. The meaning of the function is something like "Hey Processing, everything you draw from now own should be offset by these amounts".

With that in mind, we can rewrite the translated house example as follows:

```
void setup()
{
  size( 500, 500 );
}

void drawHouse()
{
  fill( #BFB375 );
  rect( -100, -150, 200, 150 );

  fill( #3E362F );
  triangle(
    0, -250, -130, -150, 130, -150 );
}

void draw()
{
  background( 255 );
  translate( 0.5*width, 0.75*height );
  drawHouse();
}
```

Ah, that's much nicer. Notice how the `drawHouse()` function doesn't need to be aware of any transformations that might be applied to the drawing. The drawing can be created in whatever "local coordinates" are most convenient for drawing houses, and we trust whoever's asking for a house to establish the appropriate geometric context before drawing. The information on geometric context is passed between these two pieces of code indirectly, through a global variable that we don't see. (So we get a lot of convenience in exchange for a little bit of mystery.)

Note that every time `draw()` is called, the geometric context is re-initialized. So if you have some context you always want to work in, you should establish it from scratch at the start of `draw()`.

Pushing and popping

It's very important to realize that the effects of these transformation functions are *permanent*, at least until the end of the frame. For example, let's say we wanted to draw a row of circles using only the `translate()` function. It might look right to try this:

```
void setup()
{
  size( 600, 100 );

  for( int idx = 0; idx < 6; ++idx ) {
    translate( 50 + 100*idx, 50 );
    ellipse( 0, 0, 100, 100 );
  }
}
```

That probably doesn't do what you want. The first circle will have the correct translation of (50,50). But the second will *combine* the new translation of (150,50) with the existing translation, yielding (200,100). The third circle will be even more off. The call to `translate()` doesn't just affect the next ellipse to be drawn, it changes the underlying geometric context.

Now, there is a quick fix that will get around this. The idea is to recognize explicitly that transformations accumulate and adjust these translations accordingly:

```
void setup()
{
  size( 600, 100 );
  translate( 50, 50 );

  for( int idx = 0; idx < 6; ++idx ) {
    ellipse( 0, 0, 100, 100 );
    translate( 100, 0 );
  }
}
```

That's pretty clever, but it's potentially confusing because every ellipse depends on a whole sequence of accumulated transformations. It's better to have a way to set the graphics context separately for each ellipse, and "revert" to the previous context afterwards. Processing lets us do that with the functions `pushMatrix()` and `popMatrix()`. The function `pushMatrix()` can be thought of as "set aside the current geometric context, and make a new copy that I can play with". Then, when you're done using this pushed context, you can throw it away with `popMatrix()`, and return to whichever context was in place before. Even better, note that you can push as many times as you want, temporarily setting up layers of sub-contexts that will be discarded later. With that in mind, here's a less confusing version of the row-of-ellipses code:

The names `pushMatrix()` and `popMatrix()` are derived from the fact that the context is represented internally by a stack of matrices. You don't need to know that, but it might help explain these mysterious names.

```
void setup()
{
  size( 600, 100 );

  for( int idx = 0; idx < 6; ++idx ) {
    pushMatrix();
    translate( 50 + 100*idx, 50 );
    ellipse( 0, 0, 100, 100 );
    popMatrix();
  }
}
```

In fact, this example embodies a pretty standard model for using geometric context. To draw some

objects in context, we typically write code of this form:

```
pushMatrix();
// A sequence of translate(), rotate(),
// and scale() calls.
applySomeTransformations();
// Anything that draws objects
drawSomeStuff();
popMatrix();
```

Combining transformations

So far, the examples we've seen have relied almost exclusively on translation. I did that deliberately: it's fairly easy to see how multiple translations might combine. Things get more challenging (but also more powerful!) when we start to combine different kinds of transformations together.

One thing that makes `scale()` and `rotate()` more complex is that they operate *relative to a point*. When you rotate the world, there's a point you're rotating around; when you scale, you're scaling inward towards a point (or outward from it). You might have experienced this in practice in tools like Adobe Illustrator, where the rotation and scaling tools let you specify that point manually. It will help to keep these descriptions in mind:

- `rotate( theta )`: from now on, draw everything in a context that has been rotated by an angle `theta` around the point (0,0). The angle is given in *radians*.
- `scale( sx, sy )`: from now on, draw everything in a context that has been scaled by a factor of `sx` in the x direction and `sy` in the y direction relative to the point (0,0). That is, every point (x,y) will be transformed to (sx*x, sy*y). (Note that `scale( s )` is equivalent to `scale( s, s )`.)

The other problem is that order matters. If we have a sequence of transformations we'd like to apply to an object, we have to choose the right order in which to apply them.

As an example, let's try to draw a rotated ellipse in the centre of a sketch. We'll assume that actual call to `ellipse()` will use (0,0) as the centre of the ellipse, so that we're forced to handle translation and rotation using geometric context.

```
void setup()
{
  size( 150, 150 );
  // Do some transformations here.
  // But which ones?
  ellipse( 0, 0, 100, 50 );
}
```

It's pretty clear that we'll need some combination of a translation and rotation to get the ellipse to the right location and orientation. But which transformations, and in what order? (Exercise: work this out!)

As a further complication, suppose that the only permitted drawing operation is to create a circle of diameter 100. Can we still draw a 200 x 100 ellipse? Yes, if we use the correct `scale()` operation. But note that this "non-uniform scaling" operation doesn't necessarily do what you want, particularly when it comes to stroke widths. Unfortunately, there isn't really a way to avoid that. It's generally better to build any non-uniform scaling into the shape itself (e.g., by changing the size arguments passed to `ellipse()`) than to use `scale()` with different absolute scaling values in x and y. (Note that `scale(-1,1)` and `scale(1,-1)` are both useful—they flip the world across a vertical and horizontal axis, respectively, without non-uniform stretching.)

In general, if we assume that we've got an object that's drawn in "local coordinates" (say, centred on (0,0)) and we want to scale, rotate, and translate it into place in a sketch, we prefer this order of operations:

```
pushMatrix();
translate( tx, ty );
rotate( angle );
scale( s );
drawTheObject();
popMatrix();
```

One or more of these function calls can of course be omitted if they're not needed.

As a quick example, we can finally correct the discrepancy between Processing's coordinate system (origin in the top level of the sketch, axes point right and down), and a coordinate system that we might find more familiar from mathematics (origin at the centre of the sketch, axes point right and up):

```
void draw()
{
  translate( width/2, height/2 );
  scale( 1, -1 );

  // Do everything else in the
  // sketch here.
}
```

This transformation will work fine if we're drawing everything in the window ourselves. However, there are a few things we can draw for which Processing does a lot of work internally: text, images (`PImage`), and illustrations (`PShape`). Processing draws these with the understanding that y points down. If we manually change that, things like images will be drawn upside-down. We'd need to apply further corrections to (un-)reflect these objects.

Let's look at one further example. Suppose we want to create a sketch in which an image is drawn in the centre of the window and it rotates around its own centre. We start with simple code to draw an image in the centre of the window.

```
PImage img;

void setup()
{
  size( 400, 400 );
  img = loadImage( "titania150.jpg" );
}

void draw()
{
  background( 255 );
  translate( width/2, height/2 );
  image( img, 0, 0 );
}
```

When we run this sketch we discover that it doesn't do what we want. The problem, of course, is that an image is drawn relative to its top-left corner, not its centre. To centre the image on the screen, we'd need to offset the translation by half the size of the image. Let me also prepare some code that will allow us to do rotation.

```
PImage img;
float angle;

void setup()
{
  size( 400, 400 );
  img = loadImage( "titania150.jpg" );
  angle = 0.0;
}

void draw()
{
  background( 255 );
  translate( width/2, height/2 );
  translate( width/2 - img.width/2,
    height/2 - img.height/2 );
  image( img, 0, 0 );
  angle += 0.01;
}
```

We could also have used the built-in function `imageMode()` to recentre images.

So far, so good—the image will be in the correct location. It seems natural simply to find the right spot

to stick in `rotate( angle )`. But where? It turns out that no matter where you try to rotate, it won't work. You'll be able to rotate around the corner of the image, or the corner of the sketch, but not the centre of the image.

The solution is one step more complicated. We must break the transformation sequence down into *three* steps. First, we move the image so that its centre lies at (0,0). Now the image is in a spot where we can rotate it around its own centre. Finally, we can move the image so it lies at the centre of the sketch window. Putting these three steps together (and remembering that we need to write them in reverse order to how we want them applied!), we end up with this code:

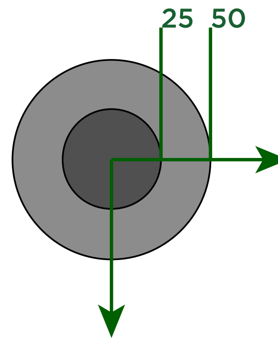
```
PImage img;
float angle;

void setup()
{
  size( 400, 400 );
  img = loadImage( "titania150.jpg" );
  angle = 0.0;
}

void draw()
{
  background( 255 );
  translate( width/2, height/2 );
  rotate( angle );
  translate(
    -img.width/2, -img.height/2 );
  image( img, 0, 0 );
  angle += 0.01;
}
```

Example sketch: RotateImage

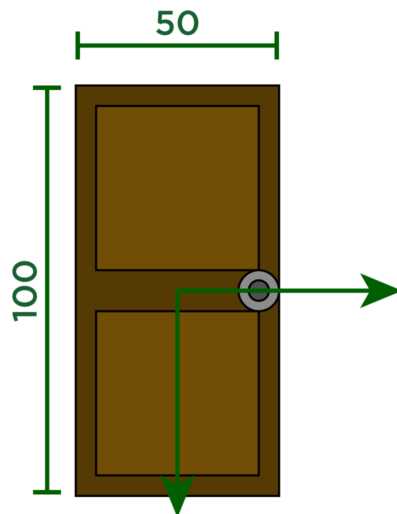
Hierarchical transformations and hierarchical modeling

Let's say we want to draw a more complicated house than the one that opened this module. We'll start small, by designing a humble doorknob:

```
void doorknob()
{
  fill( 140 );
  ellipse( 0, 0, 100, 100 );
  fill( 80 );
  ellipse( 0, 0, 50, 50 );
}
```



Next, the plan is draw a complete door, with two inset panels:



We can start out in the usual way:

```
void door()
{
  fill( #553A03 );
  rect( -25, -50, 50, 100 );
  fill( #714D05 );
  rect( -20, -45, 40, 40 );
  rect( -20, 5, 40, 40 );
  // Now draw the doorknob.
}
```

We still need to add the doorknob to the code above. We *could* copy and paste the doorknob code into the `door()` function, and modify all coordinates and sizes so it's compatible. But we've already written a `doorknob()` function. And we can exploit geometric context to re-use it inside of `door()`:

```
void door()
{
  fill( #553A03 );
  rect( -25, -50, 50, 100 );
  fill( #714D05 );
  rect( -20, -45, 40, 40 );
  rect( -20, 5, 40, 40 );

  pushMatrix();
  translate( 20, 0 );
  scale( 0.1 );
  doorknob();
  popMatrix();
}
```

This approach is elegant and principled. We prepare and push a *sub-context* that embeds the doorknob in the world (i.e., the coordinate system) of the door. Then we can go ahead and drop in a call to `doorknob()` and it gets transformed into the right location. Better yet, we can go back later and rewrite the doorknob code, and every door will automatically be upgraded. This is a bit like using Symbols in Adobe Illustrator (have you seen those?)—a symbol behaves like a helper function in Processing. When you drop a symbol into a new document, it's like pushing a sub-

context, calling a function, and then popping the context.

Of course, we can continue this embedding process indefinitely. A house might have two doors side-by-side, not to mention some windows. A street could be drawn as a sequence of houses, and so on. This powerful approach to creating objects is usually called *hierarchical modelling*: an object might contain some explicit drawing commands, but also some uses of transformed sub-objects. The power of hierarchical modelling is that each object can be drawn without thinking about how it might play a role in a larger drawing. The place where you *use* that sub-drawing is where you decide how to transform it into position.

Example sketch: HierarchicalStreet