CS 106 Winter 2016
Craig S. Kaplan

Module 06
Three-dimensional graphics

Topics

- Drawing simple shapes (boxes and spheres) in 3D
- Loading and displaying external 3D models
- Three-dimensional transformations
- Simple camera manipulation using `PeasyCam`.

Readings

- *Learning Processing,* Chapter 14

Introduction

In Shiffman's book, 2D and 3D transformations are introduced all in one go in Chapter 14. I prefer to take things one step at a time. We spent plenty of time in the previous module examining geometric context and transformations purely within the 2D world, so that the pictures we draw are still closely related to what you've done before. Now that we've seen those ideas, let's take a peek at how they translate into the third dimension. In some sense, many things in 3D behave identically to 2D (just add one D!), but some things are truly more complicated.

In full generality, modern high-performance 3D graphics programming is extremely complicated. That has more to do with the hardware than with the concepts. We use libraries like Direct3D and OpenGL to tell graphics cards what to do, and those libraries expect your data and programs to be organized in a very particular way so that the graphics card can achieve maximum performance (and let you play the

latest immersive 3D game at the highest possible frame rate). Fortunately, as with many other aspects of programming, Processing does a good job of shielding us from many of these complexities. We can dive in and explore a few parts of 3D graphics, as long as performance isn't a big bottleneck.

Programming in 3D

Let's lay out the differences in Processing between 2D and 3D graphics by walking through the sorts of functions we're accustomed to using in sketches:

- `size()`: The built-in function `size()` is always expected to be the first thing you call in your `setup()` function. We're used to using the two-parameter version of this function, in which you simply tell Processing the width and height you'd like for the sketch. But there's an optional third parameter that tells Processing what kind of graphics to use behind the scenes. To use 3D graphics, we pass in the special value `P3D`:

```
size( 500, 500, P3D );
```

- `ellipse()`, `rect()`: Two of the simplest shapes we can draw in 2D are ellipses, including circles, and rectangles, including squares. These are drawn using the built-in `ellipse()` and `rect()` functions. The 3D equivalents of these functions are `sphere()` and `box()`. They're slightly different from their 2D counterparts—see the online documentation for parameters and examples. For example, `sphere()` takes a floating-point radius as a parameter, and draws a sphere of that radius. If you want an "ellipsoid" (a sphere that's stretched out differently in different directions) you need to put the sphere inside a geometric context.
- `translate()`, `scale()`: All the geometric context functions of the previous module work in 3D as well, with a few simple differences. Translation and

scaling are nearly identical, except that they each take three parameters instead of two, indicating the amount of translation or scaling along each of the three coordinate axes (X, Y *and* Z!).

- `rotate()`: Rotation is mathematically more complicated in 3D than in 2D, and there's no real way to hide that complexity. This is the one place where we have no choice but to learn about the extra details.

  In 2D, the `rotate()` function takes a single parameter, which determines the angle (in radians) by which we want to rotate the world around the origin (0,0).

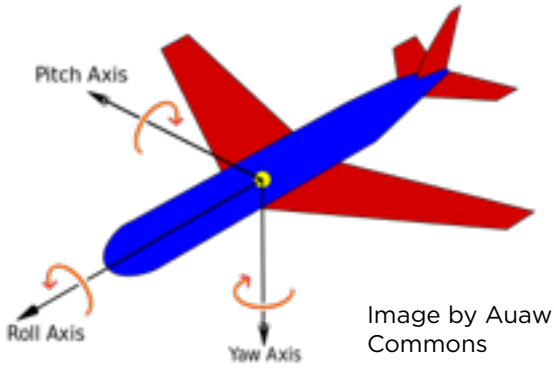

Image by Auawise, Wikimedia Commons

  But in 3D, orientation has more "degrees of freedom". Airplanes provide a standard means of thinking about 3D orientation. An airplane can point more upward or downward (*pitch*); it can drop one wingtip and raise the other (*roll*); and it can turn to face left or right (*yaw*). If we imagine the airplane sitting at the origin of space, with the X axis pointing to its right along one wing, the Y axis sticking out of the top or bottom, and the Z axis pointing forward from the nose, then we can define three "principal rotations" to control its orientation: `rotateX()`, `rotateY()` and `rotateZ()` for pitch, yaw and roll, respectively. These are the transformation functions we use when drawing in 3D in Processing.

Mathematically, even these three rotation functions are far from ideal as a means of expressing orientation, for reasons that aren't worth going into. The most elegant representation of orientation is probably *quaternions*, but we're not going to open that can of worms.

- `pushMatrix()`, `popMatrix()`: Happily, these two functions are "dimensionless": they work exactly the same in 3D as in 2D. They're more about

remembering what you were doing so that you can return to that state later.

Colour and light

How do we control the appearance of objects in 3D? Happily, the simple functions `fill()` and `stroke()` continue to work as expected, and provide a simple means of getting solid-colour 3D shapes on the screen. 3D objects can be rendered using much more complicated material properties as well (in order to make them look shiny, for example), but we won't talk about those. See the Processing documentation for the relevant functions.

Without any additional code, using `fill()` and `stroke()` will result in a flat-looking sketch, because objects will not be shaded in any way. In order to get 3D shading, we must enable at least one light in the scene. The easiest way to do so is via the built-in function `lights()`, which turns on a couple of default lights. Processing offers more fine-grained control over lighting but for many simple sketches the default will be sufficient.

Loading and displaying 3D models

In Module 01 we learned about working with 2D drawings stored in external files. We can use the built-in function `loadShape()` to read an external .svg file, which can be stored in an object of type `PShape`. Then we use the built-in function `shape()` to display that drawing, much as we might do with the `image()` function. These functions freed us from the frustration of composing complicated drawings piece by tedious piece in Processing; instead we can use familiar tools like Adobe Illustrator.

What's great is that these three pieces, `loadShape()`, `shape()` and `PShape`, work equally well in 3D, with external 3D models stored as .obj files.

There's very little to say about 3D models beyond that, since it's so easy to display them. It can be difficult to create high-quality 3D models, but that's beyond the scope of this course. As with SVGs, a 3D model will be defined in its own internal coordinates, and could potentially have any size and position in 3D space. You need to be aware of this size and position in order to transform the model so that it's visible at the right location in your sketch.

One final note: some 3D models come with their own predefined colours and material parameters. It's possible to override these materials and force the model to use the sketch's current fill and stroke—just call the `disableStyle()` method on the `PShape` that represents the 3D model you're drawing.

PeasyCam

When displaying a 3D object, a standard operation you'll want to do is manipulate the 3D view so that you can examine the object from multiple positions. We'd like to be able to tumble (rotate) the object around, and maybe move it from side to side and towards and away from us. These are typical 3D manipulations, but unlike their 2D equivalents they aren't necessarily easy to program. We therefore rely on an external library to implement this simple interface for us. With Processing, the PeasyCam library makes 3D camera manipulation easy. We must `import` the library and create a camera object; from then on, the library takes over the mouse and allows us to manipulate the camera in ways that would be comfortable to any use of 3D graphics software.

Putting together some of the pieces seen in this module, we can write a fairly short sketch that loads and displays a 3D model (one that we assume has been placed in the sketch's `data/` folder)

```
import peasy.*;

PShape airplane;
PeasyCam cam;

void setup()
{
  size( 500, 500, P3D );
  airplane = loadShape( "airplane.obj" );
  airplane.disableStyle();

  cam = new PeasyCam( this, 100 );
}

void draw()
{
  background( 50 );
  lights();
  fill( 255, 255, 0 );

  shape( airplane );
}
```

Example sketch: PeasyAirplane

Several other example sketches can be found in the samples provided with this module.