An (possibly) Interesting Speed Up and a Shortcut

First, the speed up.

Suppose you are trying to stationary draw a very complex background with loops and lots of figures. If you do, the speed of your program may drop to an unacceptable level.

Here is an example from class. This is the file Demo1Slow in the folder Demo1-PGraphics in the folder CC18B.

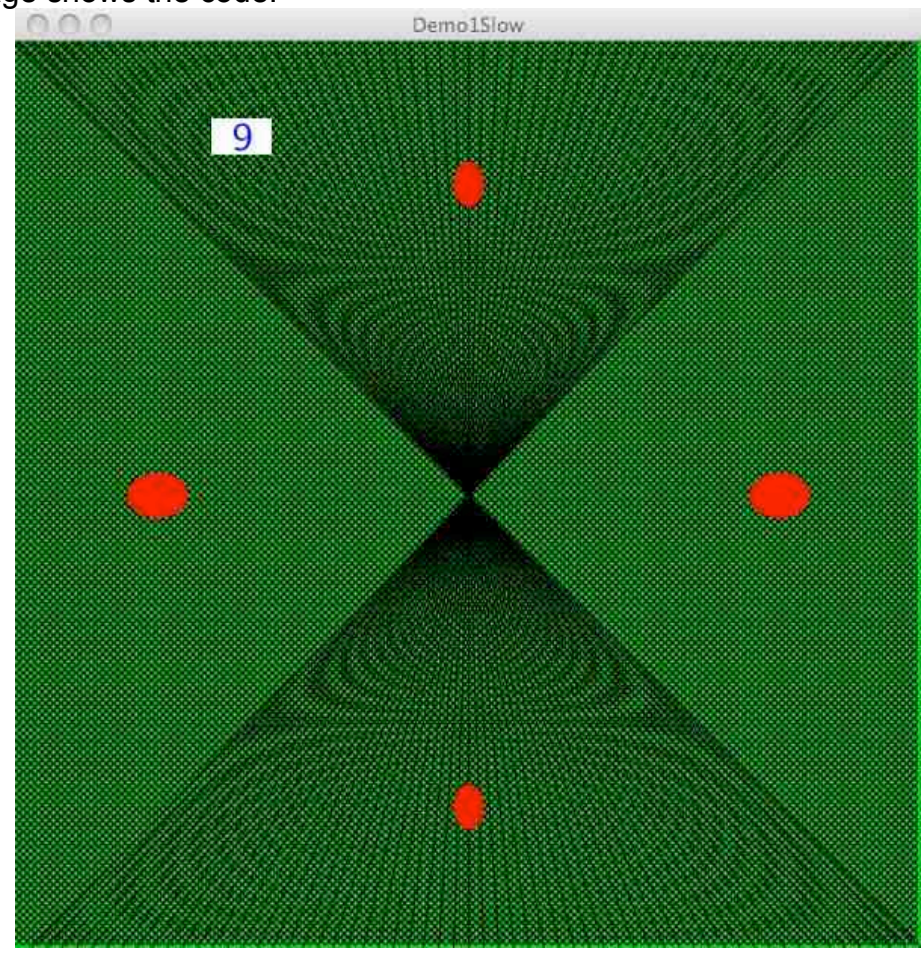
```
void setup()
{
  size(600, 600);
  rectMode( CENTER );
  ellipseMode( CENTER );
  textSize( 24 );
  textAlign( CENTER, CENTER );
  smooth( );
}

void draw()
{
       background( 0, 200, 0 );

       stroke( 0 );
       noFill( );
       for ( int x = 0; x < width; x += 10 )
       {
         line( x, 0, width-x, height);
       }

       for( int r = 0; r < width; r+=5)
       {
         for( int c = 0; c < height; c += 5)
         {
            ellipse( r, c, 4, 4 );
         }
       }

     // draw moving figures
     noStroke( );
     fill( 255,  0, 0 );
     ellipse( width/2, frameCount%height, 20, 30 );
     ellipse( width/2, height - frameCount%height, 20, 30 );
     ellipse( frameCount%width, height/2, 40, 30 );
     ellipse( width - frameCount%width, height/2, 40, 30 );

     // show frame rate
     fill( 255 );
     rect( width/4, height*.105, 40, 24 );
     fill( 0, 0, 255 );
     text( int( frameRate), width/4, height*.10 );
}
```

The main problem is that the code is drawing hundreds of ellipses every frame and this takes a lot of time. The next page shows you a screen shot from Jim's Mac with the frame rate shown in the upper left quadrant:

Again, the reason the frame rate is so slow is that it takes a lot of arithmetic to compute which pixels to use to draw the ellipses. This must be done every frame so the rate is very slow.

Process provides a way to fix this. We can use the `PGraphics` class to render or draw the background image one time in setup() and have it stored in memory so we can just show it like we show a `PImage` object. This means that the arithmetic is done only one time. The trade off is that we use more memory. Remember – when we talk about something being "expensive" the currency we are referring to is either memory or time. We are trading time for memory.

The next page shows the code.

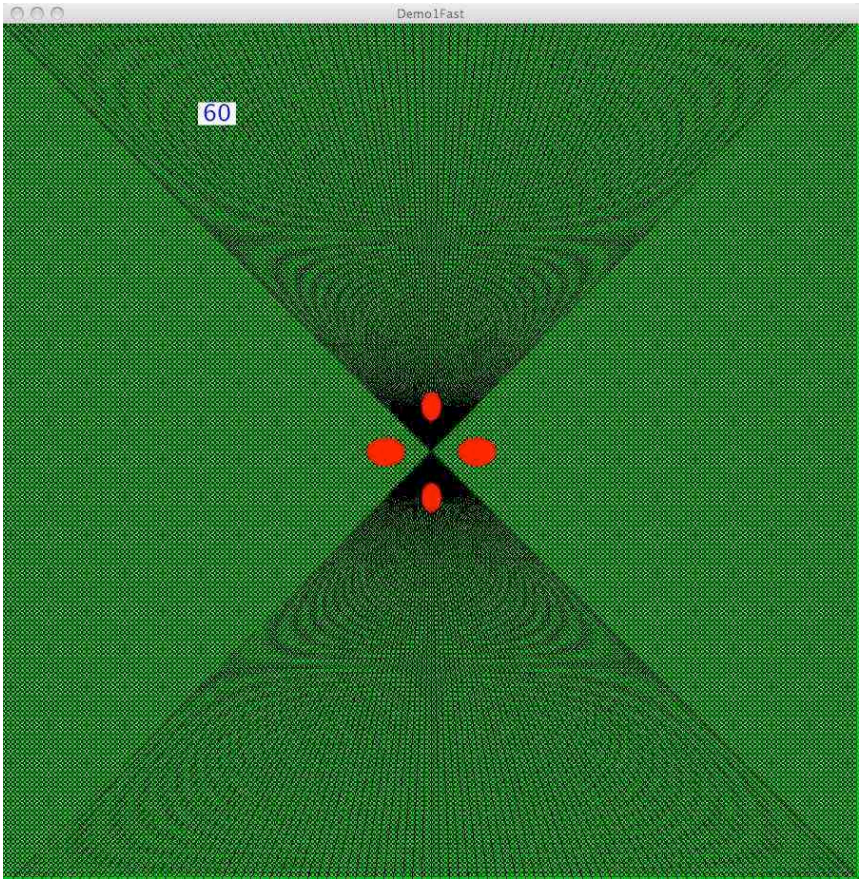


This is the file Demo1Fast in the folder Demo1-PGraphics in the folder CC18B.

```
PGraphics pg;

void setup()
{
  size(900, 900);
  rectMode( CENTER );
  ellipseMode( CENTER );
  smooth( );
  textSize( 24 );
  textAlign( CENTER, CENTER );

  pg = createGraphics(width, height, P2D);
  makeBackground( );
}

// This function makes what is essentially an image
// and stores it for quick retrieval when we want to
// display it.
void makeBackground( )
{
  pg.beginDraw( );
       pg.background( 0, 200, 0 );
       pg.stroke( 0 );
       pg.noFill( );
       for ( int x = 0; x < width; x += 10 )
       {
         pg.line( x, 0, width-x, height);
       }

       for( int r = 0; r < width; r+=5)
       {
         for( int c = 0; c < height; c += 5)
         {
            pg.ellipse( r, c, 4, 4 );
         }
       }
  pg.endDraw( );
}

void draw()
{
     // draw the background
     image( pg, 0, 0 );
     // draw moving figures
     noStroke( );
     fill( 255,  0, 0 );
     ellipse( width/2, frameCount%height, 20, 30 );
     ellipse( width/2, height - frameCount%height, 20, 30 );
     ellipse( frameCount%width, height/2, 40, 30 );
     ellipse( width - frameCount%width, height/2, 40, 30 );

     // show frame rate
     fill( 255 );
     rect( width/4, height*.105, 40, 24 );
     fill( 0, 0, 255 );
     text( int( frameRate), width/4, height*.10 );
}
```

The changes are shown in purple.

Here is the result – check out the frame rate:



If you find your code slowing down due to a complex background you are drawing, consider using this technique.

Jim experimented with using the background( ) function to draw the PGraphics object instead of the image( ) function. The execution speed slowed down quite a bit. If you are after speed, use the image( ) function.

Now the example of code writing code:

Yes, we can have code write code. To do this we use the console window that is at the bottom of the IDE window. Usually this is for debugging. In a "traditional" Java class, all of our output goes here unless we add a lot of code to create a graphics window like Processing provides.

The task is to create a polygon that conforms to the shape of the Carnegie Mellon campus. The first thing we do is get a map The one in the next image is from the University's web site. Hopefully, the university will not mind us using this…

What we are going to do is display this using the `PImage` class and the `image( )` function in a Processing program. Then we will use the `mousePressed( )` function to literally write our code. We will use the following syntax:

```
beginShape( );
  vertex ( 100, 100 );
   // more vertex calls with more arguments
endShape( CLOSE );
```

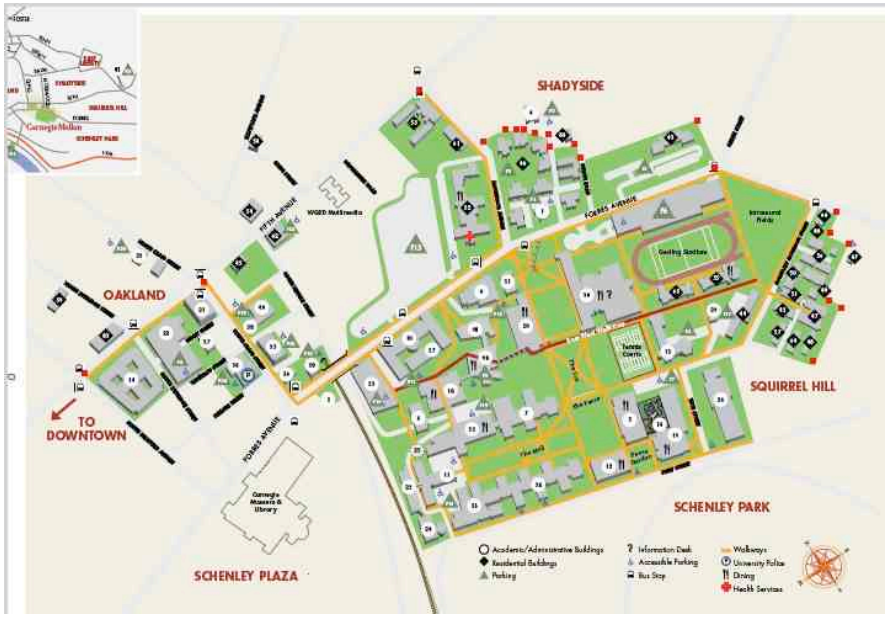
We will get our numbers from the location of the mouse.

Each time the mouse is clicked, Processing will call our `mousePressed( )` function. Inside that function we will use the `println( )` function to write our programming code to the console window. Here is the code we will use:

```
void mousePressed ( )
{
  point( mouseX, mouseY );
  println( "  vertex( "  + mouseX + ", " + mouseY + ");" );
}
```

The `point()` will let us keep track of where we click on the perimeter of the map. The `println( )` will write the argument to the console window. The argument is a series of Strings inside the " "s and the variables `mouseX` and `mouseY`. When the + operator is mixed with Strings, the result of concatenation – literally the gluing together of the Strings and the values of the variables into one long String that is written to the Console window.

Here is what the graphics and the console windows look like part way through a run:
First, the graphics window:



If you look carefully, you will see the red squares marking the places Jim clicked on the image. Each click caused Processing to execute the mousePressed( ) function shown above.

The following output was generated in the console window:

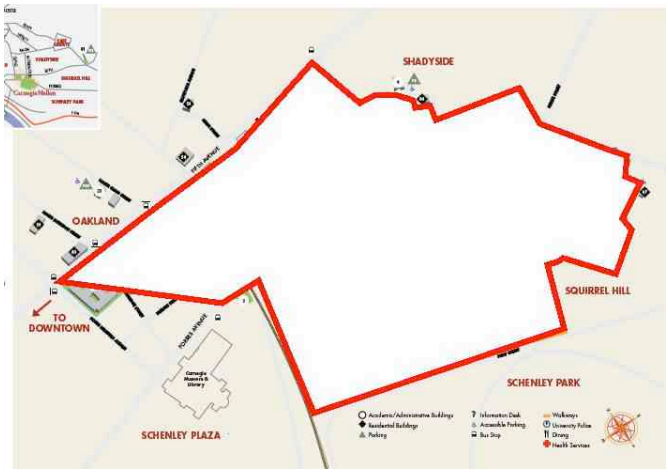
```
beginShape( );
vertex( 72, 307);
vertex( 169, 233);
vertex( 344, 77);
vertex( 396, 119);
vertex( 414, 109);
vertex( 423, 109);
vertex( 429, 109);
vertex( 438, 112);
vertex( 449, 113);
vertex( 450, 123);
vertex( 470, 121);
vertex( 475, 132);
vertex( 570, 102);
vertex( 584, 140);
vertex( 683, 176);
vertex( 678, 193);
vertex( 693, 202);
vertex( 677, 248);
vertex( 687, 254);
vertex( 664, 297);
```

To us this in a program the following must be done:

1. Do a select – all and copy the output in the console window.
2. Paste the console output into a Processing program.
3. add the closing syntax:

```
endShape( CLOSE );
```

Here is the polygon Jim made when he clicked around the entire perimeter of the campus map.
First, fill set to white:



Second, noFill( ) is used:

You may find this a way to save some time when you are working on Homework 12 or your projects