Exploring Engineering:  
Using OpenGL in Processing

# Dov Kruger

(c)2015 all rights reserved. Licensed to Ad Astra Education

Solar System Images (cylindrical projection): https://drive.google.com/open?id=0ByWFfdXuM\_awZmgzTS1LTkNyRDQ

# Introduction

OpenGL is a high performance graphics API (Applications Programming Interface, a set of functions to control something) that lets programmers control graphics hardware directly and get the full power of high performance hardware.

The graphics hardware on computers today is often far faster than the computer itself. Many boards made by NVIDIA and AMD have thousands of processors, each capable of computing a little slower than the main computer, but since there are so many capable of computing in parallel, the computational power is gigantic. In graphics, this power is used to manipulate all the pixel (dots) on screen fast enough to render animations at 60 frames per second where each frame can have 8 million pixels or more. Graphics is extremely computationally intensive, but it is also easily parallelized because each dot can be computed independently, so parallel computation is a natural approach.

In this unit, you will learn how 3D graphics works so that you can make beautiful 3D scenes. We will not be using the most up-to-date approach to OpenGL which is far faster but also more complicated. The last section of this unit will show you where you can go to learn more.

# 

# 

# First OpenGL Program in Processing

Processing provides functions to draw shapes on the screen in 2D by default. It also provides a way to hook in different drawing engines in the size command. By default you drew to the screen in 2D. Now you will learn that you can draw using a different set of commands that are much more powerful.

The Renderer is a parameter of the size function. The following program shows how to select the OPENGL renderer and draw a sphere on the screen. The sphere is automatically placed at the origin (0,0,0). This is just a quick taste, lots of great details to come!

**void setup() {**

**size(600, 400, P3D);**

**}**

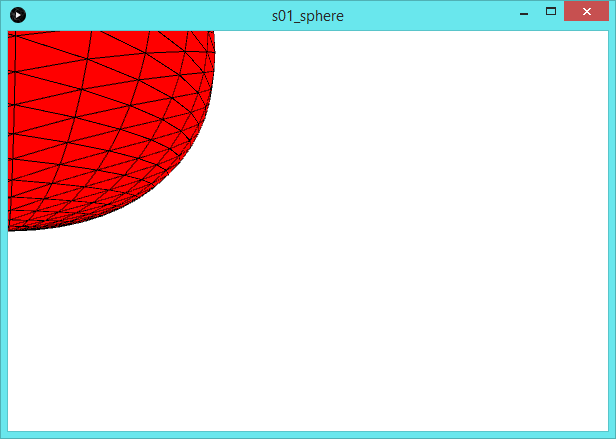
**void draw() {**

**background(255);**

**fill(255,0,0);**

**sphere(200);**

**}**



## 

## 

## Sidebar: Rendering to PDF Files

There are multiple options for rendering available in the size function. These include OPENGL, but also PDF. Rendering to a PDF file is amazing because the output that would otherwise go to a screen can be sent to a pdf file which you can then send to someone. Think of your program as writing a book rather than a movie. The following example shows a program which write the same page (a checkerboard) three times with a different title each time.

import processing.pdf.\*;

void setup() {

**size(800,800, PDF, "test.pdf");**

textSize(100);

}

final int n = 8;

float ang = 0;

void draw() {

background(255);

final int box = width / n;

boolean red = true;

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

if (red) {

fill(255,0,0);

rect(j\*box, i\*box, box, box);

}

red = !red;

}

red = !red;

}

fill(0);

text("Page " + frameCount, 0, 100);

**if (frameCount == 3)**

**exit();**

**PGraphicsPDF pdf = (PGraphicsPDF) g; // Get the renderer**

**pdf.nextPage();**

}

# Step 1: Define a Triangle

To draw in OpenGL, everything is based on triangles. To define a triangle in 3D, use the beginShape() function, specify 3 or more vertexes, then endShape()

void setup() {

size(600, 600, P3D);

}

void draw() {

background(0);

fill(255,0,0);

beginShape();

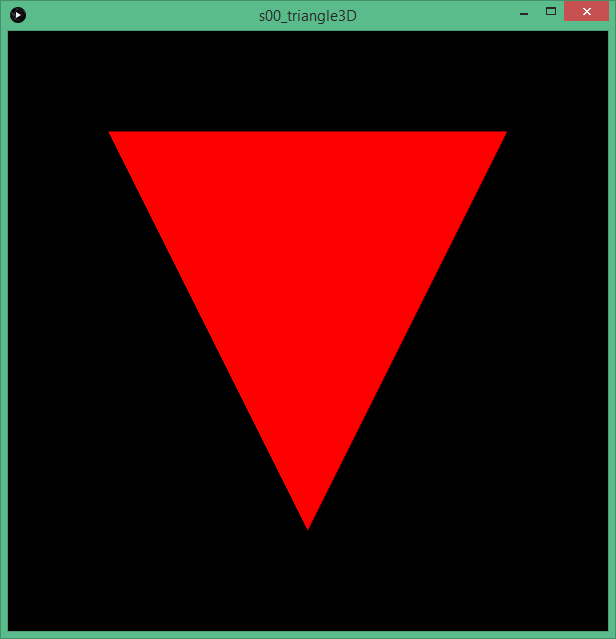
vertex(100,100);

vertex(500,100);

vertex(300, 500);

endShape();

}



# 

# Gouraud Shading

In OpenGL, a triangle does not have to be a single color. With Gouraud shading, triangles can smoothly interpolate between different colors at their vertices. The following triangle has red, green and blue corners.

void setup() {

size(600, 600, OPENGL);

}

void draw() {

background(0);

beginShape();

fill(255,0,0);

vertex(100,100);

fill(0,255,0);

vertex(500,100);

fill(0,0,255);

vertex(300, 500);

endShape();

}



# More Triangles

The beginShape function can take a parameter which allows drawing polygons in different ways. OpenGL will draw just points, the edges, the polygons, and can even share points when triangles are connected in different ways. The following examples are taken from the processing reference manual

| example pic | beginShape(); vertex(30, 20); vertex(85, 20); vertex(85, 75); vertex(30, 75); endShape(CLOSE); |
| --- | --- |
| example pic | beginShape(POINTS); vertex(30, 20); vertex(85, 20); vertex(85, 75); vertex(30, 75); endShape(); |
| example pic | beginShape(LINES); vertex(30, 20); vertex(85, 20); vertex(85, 75); vertex(30, 75); endShape(); |
| example pic | noFill(); beginShape(); vertex(30, 20); vertex(85, 20); vertex(85, 75); vertex(30, 75); endShape(); |
| example pic | noFill(); beginShape(); vertex(30, 20); vertex(85, 20); vertex(85, 75); vertex(30, 75); endShape(CLOSE); |
| example pic | beginShape(TRIANGLE); vertex(30, 75); vertex(40, 20); vertex(50, 75); vertex(60, 20); vertex(70, 75); vertex(80, 20); endShape(); |
| example pic | beginShape(TRIANGLE\_STRIP); vertex(30, 75); vertex(40, 20); vertex(50, 75); vertex(60, 20); vertex(70, 75); vertex(80, 20); vertex(90, 75); endShape(); |
| example pic | beginShape(TRIANGLE\_FAN); vertex(57.5, 50); vertex(57.5, 15);  vertex(92, 50);  vertex(57.5, 85);  vertex(22, 50);  vertex(57.5, 15);  endShape(); |
| example pic | beginShape(QUADS); vertex(30, 20); vertex(30, 75); vertex(50, 75); vertex(50, 20); vertex(65, 20); vertex(65, 75); vertex(85, 75); vertex(85, 20); endShape(); |
| example pic | beginShape(QUAD\_STRIP);  vertex(30, 20);  vertex(30, 75);  vertex(50, 20); vertex(50, 75); vertex(65, 20);  vertex(65, 75);  vertex(85, 20); vertex(85, 75);  endShape(); |
| example pic | beginShape(); vertex(20, 20); vertex(40, 20); vertex(40, 40); vertex(60, 40); vertex(60, 60); vertex(20, 60); endShape(CLOSE); |

# 

# 

# 3D Transformations

OpenGL is a completely different approach to graphics than the 2d graphics you have learned thus far. Instead of drawing commands like line, ellipse, rect, the OpenGL commands allow you to define shapes that exist in 3D. What these look like depends on where your viewpoint is. For example, a triangle viewed in 3D could look like a triangle when viewed face on, but turn by 90 degrees and look from its edge and it looks like a line.

There are three transformations in OpenGL and most 3D graphics systems:

* Translation: move coordinates
* Rotate: spin around an axis
* Scale:

## 

## 

## Translate

Translation moves the origin. All the points subsequently created are moved by the translation. For example, the following program creates a triangle. With translation, the triangle is at the center of the window. Without translation, it is at the top-left.

| void setup() {  size(600, 600, OPENGL);  }  void draw() {  background(0);  fill(255,0,0);  beginShape();  vertex(50,0);  vertex(0,100);  vertex(100, 100);  endShape();  } | void setup() {  size(600, 600, OPENGL);  }  void draw() {  translate(width/2,height/2);  background(0);  fill(255,0,0);  beginShape();  vertex(50,0);  vertex(0,100);  vertex(100, 100);  endShape();  } |
| --- | --- |

## 

## 

## Rotate

In three dimensions, rotation is interesting! The question is, what do you want to rotate around? There are three primary directions:



Processing provides three convenience functions: rotateX, rotateY, rotateZ. There is also a full rotate where you must specify the axis about which to spin. All rotate commands are in ***radians***, not degrees (there are 2π radians in a circle, so 1 radian is about 57 degrees).

The following example code all uses the following:  
**translate(width/2, height/2); // translate origin to center of window  
box(100); // create a cube 100 units on a side**

| rotateX(PI/4); box(100); // 45 degrees |  |  |
| --- | --- | --- |
| rotateY(PI/8); box(100); //22.5 degrees |  |  |
| rotateZ(PI/8); box(100); //22.5 degrees |  |  |
| rotate(PI/8, 1, 1, 0); box(100); //22.5 degrees around axis (1,1,0) which is 45 degrees (halfway between x and y) // rotate around the vector (1,1,0): |  |  |

## 

## 

## Scale

Each dimension x,y,z can be scaled independently. This can stretch shapes in any direction. Scaling equally in all dimensions makes a shape grow or shrink.

Never scale any dimension by 0 or all points will have a zero value in that coordinate. This will often cause the shape to disappear.  
  
The following examples all place different scales in front of the box command below:

**translate(width/2, height/2); // translate origin to center of window**

**box(100); // create a cube 100 units on a side**

| scale(2,1,1);  **box(100);** |  |  |
| --- | --- | --- |
| scale(1,2,1);  box(100); |  |  |

## 

## 

## Combining Transformations

The order of transformations matters -- a lot. Rotating changes the definition of what the coordinates are, so translating before or after makes an enormous difference. Here example, is translating by 100 in the x, and rotating by 45 degrees in both orders:

| rotateZ(PI/4): translate(200,0); |  |
| --- | --- |
| translate(200,0); rotateZ(PI/4); |  |

## Translate and Scale

The scale function changes all subsequent coordinate information. This means that if you scale by 2, not only all the vertices get twice as big in that dimension, but translate as well:

| box(100); |  |
| --- | --- |
| translate(200,200); scale(3,2;  box(100); |  |
| scale(3,2);  translate(200,200); box(100); |  |

# Orbiting without Rotating

If you rotate a shape around a central point, it will also rotate around its center. The table below shows this, and also shows how to compensate by rotating in the reverse direction.

| rotateZ(ang); translate(300,0); |  |
| --- | --- |
| rotateZ(ang); translate(300,0); rotateZ(-ang); |  |

# 

# Sidebar: How transformations work

All 3d transformations.can be represented by a 3 x 4 matrix, Matrices are a rectangular grid of numbers. In this case, in order to create and combine all possible rotations, scale and translations, we need 3 rows of 4 columns each .

In order to multiply the matrix by the vector, sum the product of each element of the row with the column. The last element in each row is multiplied by the fictitious 1.

a b c d x a\*x + b\*y + c\*z + d\*1

e f g h \* y = e\*x + f\*y + g\*z + h\*1

i j k m z i\*x + j\*y + k\*z + m\*1

1

From now on, ignore the last 1 and treat the rightmost column of the matrix as something to add at the end. In other words:

a b c d x a\*x + b\*y + c\*z **+ d**

e f g h \* y = e\*x + f\*y + g\*z **+ h**

i j k m z i\*x + j\*y + k\*z **+ m**

The identity matrix I is equivalent to 1. Multiplying it by any point yields the same point

1 0 0 0 x 1\*x+0\*y+0\*z = x

0 1 0 0 \* y = 0\*x+1\*y+0\*z = y

0 0 1 0 z 0\*x+0\*y+1\*z = z

If we replace the top-left element of the matrix with a number, it multiplies x in the answer. For example, the following matrix will take any point (x,y,z) and turn it into (2x,3y,z)

2 0 0 0 x 2\*x + 0\*y + 0\*z 2x

0 3 0 0 \* y = 0\*x + 3\*y + 0\*z = 3y

0 0 1 0 z 0\*x + 0\*y + 1\*z z

So in general, in order to perform the scale operation by (sx,sy,sz) we need a matrix:

sx 0 0 0

0 sy 0 0

0 0 sz 0

To translate a point by (tx,ty,tz) add these to the rightmost columnm of the matrix:

1 0 0 tx

0 1 0 ty

0 0 1 tz

To rotate around the z axis, x and y will change. The trigonometry is not derived here, but to rotate by angle Θ use the matrix

cosΘ sinΘ 0 0

-sinΘ cosΘ 0 0

0 0 1 0

To rotate around the x axis, the same sin and cosine values are put around the y and z axes

1 0 0 0

0 cosΘ sinΘ 0

0 -sinΘ cosΘ 0

Last, to rotate around y axis:

cosΘ 0 sinΘ 0

0 1 0 0

-sinΘ 0 cosΘ 0

It is possible to rotate around any arbitrary axis of rotation, but the derivation is beyond the scope of this introduction.

To combine any two transformations, multiply the matrices. For example, to scale, then translate:  
  
1 0 0 0

0 0 1 0 0 0 sz 0

0 0 0 1 0 0 0 1

sx 0 0 0 1 0 0 tx

0 sy 0 0 \* 0 1 0 ty =

0 0 sz 0 0 0 1 tz

0 0 0 1 0 0 0 1

sx 0 0 tx

0 sy 0 ty

0 0 sz tz

The fourth row of the matrix is just shown for theoretical purposes. Practically since it is always the same we can throw it out. A 4x4 matrix could be used for other operations such as shearing. Wikipedia summarizes 3d transformations well from a theoretical point of view:

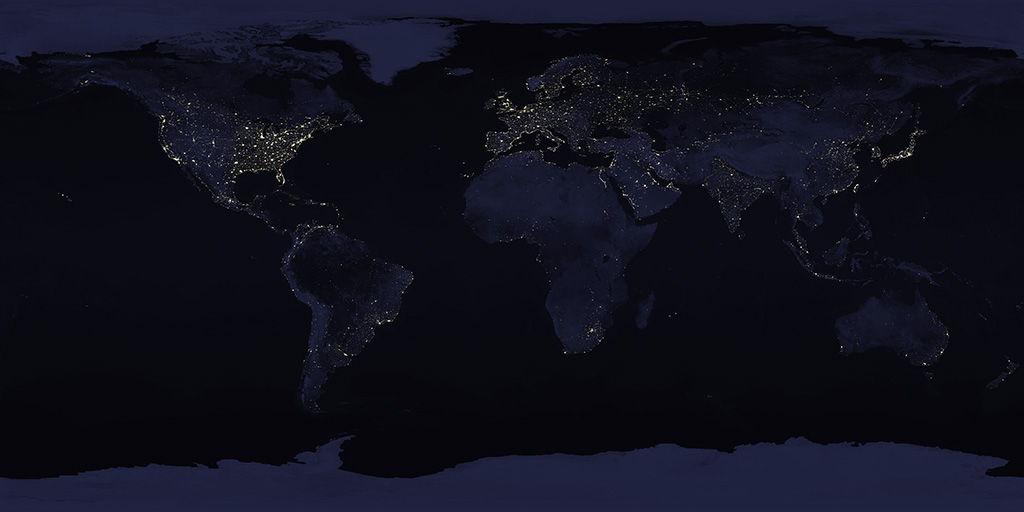
<https://en.wikipedia.org/wiki/Transformation_matrix>

# Textures

OpenGL allows us to attach images to the surfaces of shapes. The simplest way is to attach a complete picture to a sphere. Doing so will warp the picture unless it has been deliberately designed to wrap around.

For this demonstration, select a picture that is in cylindrical coordinates. For this example, we will get some textures for planets from [planetpixelemporium.com](http://planetpixelemporium.com/planets.html).

Here is a sample earth map showing lights on at night in cylindrical projection. When stretched onto a sphere, it looks perfect:



On the other hand, if you wrap a normal picture onto a sphere, it looks distorted:



## Creating a Texture

PShape earth;

void setup() {

size(900, 700, OPENGL);

noStroke();

sphereDetail(50);

earth = createShape(SPHERE, 240);

PImage img = loadImage("earth.jpg");

earth.setTexture(img);

// the larger this number, the better the approximation to a sphere

}

float angle = 0;

void draw() {

background(0); // draw the black of space

lights(); // enable lighting

translate(width/2, height/2); // move to the center

rotateY(angle); // rotate the earth

shape(earth); // draw the earth

angle += 0.01; // compute the next angle

}

## 

# Lighting

Color is a function of light. Without light, we cannot see. The examples so far defined colors that were part of their shapes, but this is not realistic. Real materials reflect light in different ways depending on the texture of the material and the light available.

OpenGL provides lighting options to create fairly realistic scenes. There are four lighting options.

The values v1,v2,v3 specify color either as RGB or HSV depending on the mode

| pointLight(v1,v2,v3, x,y,z) | Shine in all directions |
| --- | --- |
| spotLight(v1,v2,v3,dirx,diry,dirz,angle,conc) | Point in a particular direction with a spread  See red book for complete description, concentration should be from 1..10,000 |
| directionalLight(v1,v2,v3, dirx, diry,dirz) | Colored light at infinity with the given direction causing shadows |
| ambientLight(v1,v2,v3) | Background lighting from all directions |

Once lighting is turned on, colors do not work by themselves. A color is, after all, only visible with light. Lighting takes more time to compute because OpenGL has to compute what light hits each triangle.

# 

# 

# Transforming Multiple Objects Independently

Every time you transform the coordinates in OpenGL, it affects everything from then on.

For example, the following

translate(width/2, height/2);

sphere(50);

box(50);

void draw() {

translate(width/2, height/2,-1000);

pushMatrix(); // remember the current transforation

rotateY(sunRotation);

shape(sun); **// first draw the sun, spinning**

popMatrix(); **// get back to the default transform (undo the rotation of the sun)**

rotateY(earthRotation); **// rotate to the angle of the position in earth’s orbit**

translate(500); **// move out to earth’s orbital position**

shape(earth); **// draw the earth (but it is rotated incorrectly)**

}

void draw() {

translate(width/2, height/2,-1000);

pushMatrix(); // remember the current transforation

rotateY(sunRotation);

shape(sun); // first draw the sun, spinning

popMatrix(); // get back to the default transform (undo the rotation of the sun)

**rotateY(earthOrbit);** **// rotate to the angle of the position in earth’s orbit**

translate(500); **// move out to earth’s orbital position**

**rotateY(-earthOrbit)**

shape(earth); **// draw the earth (but it is rotated incorrectly)**

}

# 

# Camera and Projection

From the beginning, we have said that OpenGL allows programmers to specify what the shapes are. What appears on the screen depends on the point of view. One way of obtaining different points of view is to move the objects of the world around. However, another way is to move the eye that is viewing the scene.

In Processing, the function to set the viewpoint is called camera(). The call to camera with noo parameters defines a default camera looking straight back at the screen from (0,0,200). In order to define an alternate view, for example:

camera(1000,0,0, 0,0,0, 0,1,0); // the camera is at (1000,0,0), looking towards (0,0,0) with positive y upward. The same scene upside-down would be:

camera(1000,0,0, 0,0,0, 0,-1,0);

For example

# 

# 

# Multiple Independent Transformations

Use pushMatrix() and popMatrix() functions to allow drawing in whatever coordinate system without affecting other parts of the drawing

pushMatrix();

// do some transformation

popMatrix(); // get back to the original state

// now you can draw without being affected.

# 

# Taking it Further: Current OpenGL

The OpenGL covered here is just the beginning, and somewhat old. The latest versions allow downloading programs onto the video card, so that performance can be far higher. The price is complexity, which is why this unit up to this point has not covered the latest details.

The slow part of OpenGL is all the function calls, each of which takes time. The first way to improve performance is to reduce the number of calls. One way to do this is to pass an entire array of vertices instead of just once. Thus, instead of:

vertex(x,y,z);

vertex(x2,y2,z2);

…

You can write:

vertex(v);

where v is an array of float containing a number of x,y,z points. A single call to vertex with an array can be the equivalent of hundreds or thousands of vertex calls.

The second performance problem is that the point data has to get from the main computer to the graphics card, and this link is a bottleneck. If there is data that does not change, it can be preloaded on the graphics card. This technology was called VBO (Vertex Buffer Objects), and you can look it up in the red book listed at the end of this unit.

Last, it is possible to run entire graphics programs on the card itself. These programs are called shaders since much of what the do involves complicated coloring and shading, but in fact they can do a lot more. Doing so requires writing the program in

If you want to write higher performance programs, the following table shows some key resources including the last two (FREE BOOKS!)

| [opengl.org](http://opengl.org) | The primary site for OpenGL with the standards, announcements, forums, and other information |
| --- | --- |
| [quick reference to OpenGL 4.4](https://www.khronos.org/files/opengl44-quick-reference-card.pdf) | A one page summary of the latest OpenGL API |
| <https://www.processing.org/tutorials/pshader/> | A tutorial on using shaders in processing |
| [3D Tutorials](http://www.tutorialspoint.com/computer_graphics/3d_transformation.htm) | Tutorials on 3D transformations and other 3D graphics |
| [NeHe Gamedev](http://nehe.gamedev.net/) | A set of tutorials in how to create effects. The tutorial is usually written in C but there are often versions of each example in many different languages. |
| [Red Book](http://www.ics.uci.edu/~gopi/CS211B/opengl_programming_guide_8th_edition.pdf) | A free pdf version of THE book on OpenGL that is very complete |
| [OpenGL Shaders](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwjs-6WK77DJAhUEHT4KHS1zAhUQFgg6MAE&url=http%3A%2F%2Fdeca.cuc.edu.cn%2FCommunity%2Fcfs-filesystemfile.ashx%2F__key%2FCommunityServer.Components.PostAttachments%2F00.00.00.06.49%2FAddison.Wesley.OpenGL.Shading.Language.3rd.Edition.Jul.2009.pdf&usg=AFQjCNHvmH0qtLU8627TpSDFkxg16-2ILQ&sig2=ltrPPRjP4mHW2zwtZWlR_w) | Free pdf of the book describing the new, high performance OpenGL Shaders allow downloading programs to high performance graphics cards. More complexity, but amazing performance and effects on a graphics card that supports it. |

# Quiz

# Exercises

1. Write a program that draws a red square 100 units on a side.
2. Write a program that draws a white hexagon on a black background.
3. Write a program that draws a tetrahedron:
4. Write a program that draws the earth spinning.
5. Modify the program so that earth’s axis is tilted 23.5 degrees to the right.
6. Create the earth-moon system with the moon ¼ the size of the earth, and the moon 30 earth diameters away. Warning: both bodies will have to look pretty small, space is vast.
7. Modify the above system to be unrealistic, with the moon approximately 1 earth diameter away, but make the moon orbit the earth every 28.5 days, and the earth spin every 23h56m. In other words, make the times for all the motion to scale.
8. Create the solar system with the earth, moon, sun and mars (not to scale). You will have to draw the sun first, then turn on lighting.
9. Write a 3d bouncing ball program that has the ball bouncing off walls in the x,y and z direction.